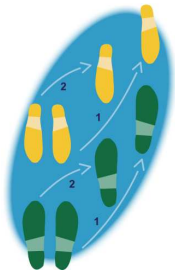
Ideals : evolvability of software-intensive high-tech systems : a collaborative research project on maintaining complex embedded systems

Ideals: evolvability of software-intensive high-tech systems

A collaborative research project on maintaining complex embedded systems

Embedded Systems Institute (The Netherlands)



Ideals: evolvability of software-intensive high-tech systems

A collaborative research project on maintaining complex embedded systems

Editors:

Remco van Engelen ASML
Jeroen Voeten Embedded Systems Institute

Publisher:

Embedded Systems Institute, Eindhoven, The Netherlands

Embedded Systems
INSTITUTE

Publisher:
Embedded Systems Institute
TU/e Campus, Den Dolech 2
P.O. Box 513, 5600 MB Eindhoven
Eindhoven, The Netherlands

Corrected : December 6, 2007

Keywords:
evolvability; software; high-tech systems; aspect-oriented programming; aspect-oriented design; model-driven engineering

ISBN: 978-90-78679-03-5

The Ideals project has been executed under the responsibility of the Embedded Systems Institute, and is partially supported by the Netherlands Ministry of Economic Affairs under the SenterNovem TS program (grant TSIT3003).

Foreword

The ancient Greeks already stated that the only constant is change, and this is still true in today's world. The world of embedded systems is of course no exception to this, and in many ways it is even characterized by the rapid rate of innovation. Technology changes at an increasing pace, customers and markets change, the environment in which machines must operate changes. ASML, as the leading manufacturer of advanced technology systems for the semiconductor industry, is both an enabler of many innovations and a continuous innovator of its own products. We introduce multiple new products every year and we continuously update our existing products already installed at customers with performance enhancements.

Not all innovations are big leaps ahead, grand re-designs or complete new concepts. True, such disruptive changes are needed every now and then to remove bottlenecks in a system or to replace a core technology that has reached its limits of innovation, but these large changes always jeopardize the investments that have established the system as efficient, reliable and robust. Many innovations are therefore on a smaller scale, changing individual parts of a system to improve a part of its performance. The combination of many small changes still leads to a valuable improvement of the overall system performance in all areas, while each individual change is easier to control.

This is the reason why ASML, like any other producer of complex embedded systems, needs the ability to make changes to our systems in an efficient and controlled manner. Evolvability is the quality of a system to be amenable to change, and this was the focus of the Ideals project: to find new ways to reduce the cost and lead time of changes in complex embedded systems, and to find innovative ways to introduce these changes into existing systems.

The systems that we develop at ASML are extremely complex semiconductor manufacturing machines that transfer nanometer circuit patterns onto silicon wafers to make semiconductor chips. The software in such a system is key in operating a multitude of subsystems, from material handling robots, highly accurate pattern/wafer positioning devices, state of the art opto-mechanical projection lenses to production control & optimization systems.

The complexity of semiconductor machines is growing at an ever increasing rate. Drivers for this increase are first and foremost the increasing product specifications: ever smaller circuit dimensions, placed with tighter accuracies at higher processing speeds, to satisfy the semiconductor roadmap for higher performance and higher capacity IC's combined with lower costs and higher energy efficiency. Breakthroughs in any of the core technologies need to be implemented into reliable production systems as fast as possible. As an example, the recent introduction of immersion technology to extend the performance of our systems at existing wavelengths, took only a few years and has led to some of the fastest production ramp-ups of new machines in the history of ASML. But also, as the cost price of an individual machine grows, so does the need to be able to continuously specialize its operation to the operation of the owner.

This leads to higher built-in ability to adapt the system, meaning a greater complexity in terms of ways the system can be operated. It also means a higher demand for upgrades to the system during its life-time, to keep it up to date with the changes in its intended usage.

So, we see two trends: growing system complexity and an increasing rate of innovation. These trends are conflicting: complexity makes changes more difficult to design, implement and test; continuous change makes it difficult to create and maintain a stable platform that can be used as a basis for reducing complexity. This effect can be clearly felt in the growth of the effort, lead time and cost that needs to be spent on each new product.

Four years ago ASML has formulated the clash of these two conflicting trends as the basis of the Ideals research project. The project results are now helping us to reduce the cost of changes to our software system, by allowing us to easily re-apply already existing solutions to many common problems during changes. It has also provided us valuable insights into the use of models to capture and analyze the impact of changes.

At the outset of the Ideals project, it was clear to ASML that we needed the help of academic partners to research this topic. But driving academic research based on an industrial problem, and ensuring that the results of the academic research are actually impacting the reality in industry, is not a trivial task. We have therefore sought the cooperation of the Embedded Systems Institute (ESI) to ensure that the research method of using industry-as-a-laboratory would be successful. Together with the academic partners, ESI has created a true spirit of collaboration between academics and industry, and helped to make this project worthwhile for both industry and academics. ASML is the first to be able to benefit from the fruits of this cooperation, but we are confident that other industries can follow.

ASML is pleased by the results of this project by itself, but equally pleased with the fact that it has proven again that a cooperative research setting where different partners bring in unique expertise, combined with a strong focus on practical industrial relevance, can create breakthrough innovations. We hope to see more of these innovations in the future.

Ir. Harry Borggreve
Senior Vice President of Development and Engineering
ASML Netherlands B.V.
Veldhoven, The Netherlands
November 2007

Preface

This book is already the third volume in a series started a year ago by the Embedded Systems Institute to report on the results of ESI research projects. This particular issue is devoted to the outcome of the Ideals project, which addresses the problem of evolvability for industrial embedded control systems, and is carried out by ASML, Delft University of Technology, Eindhoven University of Technology, University of Twente, CWI and ESI. The project started in 2003 and will finish in the beginning of 2008.

Evolvability is one the most challenging problems of high-tech industry. Typically, the time and effort required to modify and extend complex embedded systems is huge and unpredictable, posing a severe threat to meeting the ever-tighter time-to-market constraints. Consequently, it is of great importance to finds ways to improve system evolvability. This is one of the central themes of the ESI Research Agenda. Following the ESI approach to applied research, the Ideals project has been organized as an industry-as-laboratory project, where industry provides the experimental platform to develop and validate new methods, techniques and tools. ESI is unique in the application of this research format to the problems of embedded systems engineering. It has proven to be most successful in producing substantial results leading both to industrial innovation and high-quality academic output.

Ideals is the second project for which we have been fortunate enough to have ASML as the carrying industrial partner. As in the preceding Tangram project, access to the fascinating and technologically sophisticated world of high-tech silicon lithography provided a stimulating opportunity for the research consortium to develop and try out new ideas and solutions. In this context the application of techniques such as aspect-oriented programming and model-driven engineering have emerged as main tools to overcome important problems, especially for handling the fragmentation problems of software code for crosscutting concerns and for obtaining effective design abstractions.

The project participants have shown great commitment and their contributions have led to the success of the Ideals projects, for which I would like to expressly thank them. ASML and the Dutch Ministry of Economic Affairs, provided the financial basis for carrying out the project, and their support is gratefully acknowledged. We hope that with this book we can share the most important results and insights with a wider industrial and scientific community.

Prof. dr. Ed Brinksma
Scientific Director & Chair
Embedded Systems Institute
The Netherlands
November 2007

Contents

Chapter 1

Ideals: an introduction to the project and the book

Authors: Remco van Engelen, Jeroen Voeten

1.1 Introduction

High-tech systems such as wafer scanners, medical MRI[1] scanners, electron microscopes, and copiers, are typically not developed from scratch. Instead, new generations of such machines are based on older versions, where new features and capabilities are added; high-tech systems evolve over time. This proces of evolution is often driven by the required changes in the key performance parameters of such systems. As an example, driven by Moore's law, the key performance parameters of a wafer scanner are tightened from one generation to the next. These parameters mainly concern the dimensions of patterns of electronic circuits that are mapped onto a wafer and the number of wafers that are processed per hour. Even a small change in these key performance parameters can have a huge impact on the design and implementation of the embedded system that controls the wafer scanner. An important reason is that physical dependencies and effects that could be ignored in the past have to be compensated for in the next generation. This is done by mirroring them in the embedded system where they appear as (new) interactions between (new) system components. Another consequence is that the performance requirements of these components are tightened at the same time, making even more adaptations necessary. Finally, life-cycle requirements may result in a major overhaul of the existing components of the embedded system.

Evolvability poses one of the most difficult challenges the high-tech industry is currently facing. The time and effort required to modify and extend a complex em-

[1] Magnetic Resonance Imaging.

bedded system is typically huge and unpredictable, thereby severely threatening time-to-market and time-to-quality constraints. It is therefore more and more important to make embedded systems better evolvable. This is exactly the goal of the Ideals project: *developing methods, techniques and tools reducing the lead time and effort to maintain complex embedded systems*, where the focus is on embedded software. Ideals is an applied industrial-academic research project. Coordinated by the Embedded Systems Institute, ASML together with different research institutes in the Netherlands have collaborated on achieving the research goal.

This book gives an overview of the results of the Ideals project. This introductory chapter introduces the project (Section 1.2), analyzes the problem statement (Section 1.3) and introduces the two main research directions in which solutions have been developed: Aspect-oriented software design (Section 1.4) and Model-driven engineering (Section 1.5). These sections also introduce the corresponding book chapters in which detailed project results are described. The concluding chapter of this book (Chapter 10) describes the industrial impact of the project, the lessons learned, and draws the final conclusions. This introductory chapter together with the concluding chapter are self-contained and can be read without having to study the chapters describing the detailed results.

1.2 The Ideals project

The Ideals Project is an industrial-academic research and development project managed by the Embedded Systems Institute. The goal of Ideals is to develop methods, techniques and tools to make embedded software better evolvable. In Ideals, researchers and engineers from ASML have worked closely together with researchers of Delft University of Technology, Eindhoven University of Technology, the University of Twente, the Center for Mathematics and Computer Science, and the Embedded Systems Institute. The project started in September 2003, lasted until February 2008, and was financially supported by the Netherlands Ministry of Economic Affairs.

Industry-as-laboratory

The academic-industrial cooperation in Ideals took place in a setting called *industry-as-laboratory* [85]. This means that the actual industrial setting is used as a laboratory, akin to a physical or chemical laboratory, where new theories, ideas, and hypotheses, mostly coming from the academic partners in the project, are tested, evaluated, and further developed. This setting provides a realistic environment for experimenting with ideas and theories. Moreover, the industry-as-laboratory setting facilitates the transfer of knowledge from academia to industry, and it provides direct feedback about the applicability and usefulness of newly developed academic theories, which may again lead to new academic research questions. But, of course, in such a setting also care should be taken that the normal industrial processes are not disrupted.

ASML

For Ideals, the laboratory has been provided by ASML. ASML is the leading global company for lithography systems for the semiconductor industry. Their wafer scanner machines, which involve highly complex configurations of embedded systems with extreme requirements regarding performance and precision, provided a demanding and stimulating laboratory environment.

An example of the evolvability challenge that ASML faces can be found in one of the most crucial lithography system components: the projection optics. This complex system of lenses is used to project the original circuit pattern, with a size of roughly 10 by 10 centimeters and containing lines as small as 180 nanometer (1/300th of the width of a human hair), onto a silicon wafer (a large disc with a radius of 200 or 300 millimeters), while reducing the image by a factor 4, producing images on the wafer with line widths down to 45 nanometers. The quality of the projection determines the performance of the resulting IC, and thereby its value. The projection optics is not a static system: it contains a number of controls that allow tuning of the projection result to compensate for e.g. distortion in the original circuit pattern or changes in temperature or air pressure. The embedded system uses a set of sensors to sample all factors influencing the lens performance, calculate the optimal settings for the lens and drive the actuators to control the lens.

Over a period of 5 years, as the minimum exposed line width for leading edge lithography machines shrunk from 95 to 40 nanometers, the number of controls in the used projection optics subsystems grew from 5 to 60. This meant that more sensors had to be introduced and needed to be sampled, more complex models needed to be used to calculate optimal settings for the lens, and more actuators needed to be controlled. This was not a single step, but in fact a gradual growth in complexity in 5 or 6 steps during these 5 years. Every step resulted in a commercial product which targeted an intermediate line width used by the IC industry to continuously improve chip capacity and performance. Therefore, each step had to be delivered on time, work reliably and be cost effective to implement and maintain. How to manage and design such gradual changes that ultimately transform a subsystem without exploding implementation and integration costs, is a challenge that ASML faces not only for the projection optics, but in many more domains. It can be compared to the challenge posed to the Dutch Rijkswaterstaat organization to perform complete upgrades of complex highway intersections, while keeping them open for daily traffic with minimal disturbance, all at an acceptable cost.

1.3 Evolvability - problem analysis and solution directions

The evolvability problem for ASML can be stated as: *the effort and lead time to maintain and improve the embedded (software) system of a wafer scanner is too large.* In the

Ideals project we identified two major causes for this, both related to the decomposition of complex embedded (software) systems.

For complex embedded systems, the gap between the system specification (describing the desired properties in terms of behavior and key performance drivers) and the implementation (consisting of a huge number of interacting hardware and software components) is very large. As a result, people are not able to understand or verify how these interacting components together satisfy the system specification. Also, people cannot construct an implementation of such a magnitude in a single step. To deal with this complexity designers create intermediate entities (such as subsystems, modules and components) and break up the large verification and synthesis step in a sequence of manageable intermediate steps. We will call these intermediate entities artifacts. Each artifact is characterized with its own specification and design, and may in itself be further decomposed into smaller artifacts.

This process of decomposition is typically guided by a number of *concerns* [2]. Some concerns are the requirements or use cases of the system: often specific artifacts are created for each of them to have a clear assignment of responsibilities. Another group of concerns are the interfaces of the system: often specific artifacts are created as abstractions of external elements (hardware or other software components).

In principle, all these concerns could be treated equal. But when two concerns are decomposed into independent artifacts, but they have some sort of relation and hence a need for interaction, a choice must be made where to put the interaction in the decomposition. As an example, if two requirements refer to each other, and both are assigned to a separate artifact, who should be responsible for the shared part of the requirements? Placing the shared part in either artifact leads to a decomposition where one requirement is completely described in one artifact, but the other is described in two artifacts. Placing the shared part in a new, separate artifact, leads to a decomposition where both requirements are described in two artifacts. Usually the relation is put into one of the two artifacts, and a choice is made for which concern locality is considered more important. This leads to a phenomenon known as *dominant decomposition*, where some concerns have a better locality in the decomposition than others, because they are deemed more important.

As a consequence, concerns with a lot of relations but that are deemed less important end up scattered over the large number of artifacts of more important concerns. We call these scattered concerns *crosscutting concerns*, since they intersect the decomposition of the dominant or *core concerns* in a number of places. The first major cause of the large effort and lead time to maintain the embedded control system of a wafer scanner is in these crosscutting concerns, and the insufficient means and methods to efficiently deal with them in the design and implementation phases.

The second major cause of the large effort and lead time to maintain complex embedded systems is a lack of proper abstractions for the artifacts, such that the decomposition is effective. With effective we mean that one can understand and reason about each artifact without having to consider its further decomposition into constituents and

[2] A concern is a general term that refers to any particular piece of interest or focus in a system.

that one is able to reason about the interaction and combined properties of all artifacts. To be effective, the specifications (abstractions) of artifacts should describe (only) the properties that are essential to understand the system as a whole, in a compact and precise manner. Unfortunately, in practice, artifact specifications are typically of a low abstraction level, inconsistent, ambiguous or imprecise, making it very difficult to effectively use them as a basis for reasoning about system-level properties.

In Subsections 1.3.1 and 1.3.2 these major causes are explored in more detail. This exploration is followed in Sections 1.4 and 1.5 by the main solution directions of the Ideals project, i.e. *Aspect-oriented software design* and *Model-driven engineering*.

1.3.1 Crosscutting concerns

Crosscutting concerns (*CCC*'s), are those concerns (requirements, use cases, interfaces) in a system that have no clear locality in the chosen decomposition. These concerns are not cleanly decomposed from the rest of the system in either the design or the implementation and therefore recur in several artifacts. Typical examples of CCC's come from requirements and use cases related to the testing, integration and (field) support of systems[3]. While these concerns are usually not dominant in the decomposition, they are crucial to the success of a complex system and have many relations to all other concerns. Some concrete examples are:

- the ability to monitor the activity of a system during operation (tracing);
- correct and consistent handling of errors (exception handling and recovery);
- uniform access control to different capabilities of the system (licensing or user privileges).

In case a piece of functionality has to be adapted or newly developed, these crosscutting concerns have to be implemented as well. These additional concerns distract designers from focusing on their key assignment (the core concerns). A common way to design and implement a crosscutting concern is using *idioms*, where the crosscutting concern is described as a set of typical patterns to be applied in the design or implementation of the core artifacts. The intention is that the application of these idioms is both easy to do and easy to recognize in other artifacts; all instantiations of the idiom are largely the same and only limited adaptation to the location where it is applied is required. In practice however, the wide-spread use of these idioms means that the crosscutting concern is handled in a great many places (the *scattering* effect), while the interleaving of these idioms with parts dealing with the core concerns means that identifying and working with the core concern is more difficult (the *tangling* effect). Both these effects lead to engineering inefficiencies when adapting both core concerns as well as crosscutting concerns, see Figure 1.1. The problems are most manifest at the implementation level, where the idioms are recognizable as code patterns or templates, and often less at the

[3] In a broader sense: all system life-cycle activities.

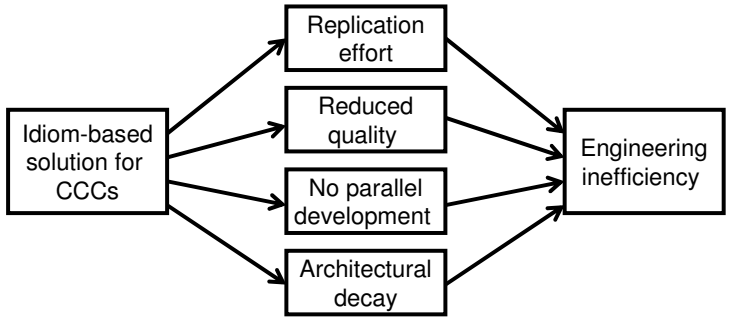


Figure 1.1: Consequences of an idiom-based solution for crosscutting concerns (CCC's).

design level. At the design level concern interactions are often left implicit; they are not described at all or in a very informal way (e.g., 'the usual tracing must be applied'). The reasons for engineering inefficiency, as depicted in Figure 1.1, are as follows:

- **Replication effort** Although the description of an idiom is usually well localized, its instantiations are by nature replicated over many places. Thus, the implementation, adaptation and testing of idiom instances is performed time and again. This takes a lot of time and effort, sometimes because of the sheer number of instantiations, sometimes because of the complexity of an instantiation, and sometimes because of both.

- **Reduced quality** Idioms have to be instantiated by hand by a software engineer. This is an error-prone activity, especially when idiom descriptions are informal and ambiguous, or when many instantiations with slight variations have to be made. This results in extra integration effort and duration to detect and correct these errors. Additionally, since typically examples of crosscutting concerns come from life-cycle requirements such as product integration and testing requirements, any remaining errors in the idiom instantiations negatively affect the efficiency of the processes to create and support a product.

- **No parallel development** The core functionality and the crosscutting concerns cannot be developed in parallel, since they are integrated into the same artifact. In addition it is difficult to out-source the development of a piece of functionality or to use commercial off-the-shelf components, since the idioms used for the crosscutting concerns should also be applied in the outsourced or bought soft-

ware. Hence parallel development is complicated, having a negative impact on effort and lead time.

- **Architectural decay** The possibility to modify the design or implementation of a crosscutting concern itself is hindered by the sheer number of idiom instantiations that already exist in a system. A change in an idiom either implies updating all instantiations of the old idiom (costing a large amount of effort and time) or accepting that multiple versions of the idiom exist in the system (hindering the ease of recognition and consistent use of the crosscutting concern). Not changing the idiom means that the crosscutting concern cannot be adapted to follow the evolution in its requirements, leading to suboptimal solutions or workarounds in the system. Both accepting multiple versions of an idiom or not changing an idiom at all leads to architectural decay of the whole system, making maintenance as a whole gradually more expensive.

In the Ideals project aspect-oriented software design (*AOSD*) techniques were investigated as a means to deal with these issues. The promise of AOSD is to allow a localized treatment of crosscutting concerns at both the design and implementation level. The research field of aspect-oriented software design together with the topics addressed in the Ideals project are explained in Section 1.4.

1.3.2 Missing effective abstractions

The second major cause we identified for the large effort and lead time to maintain complex embedded systems is the lack of effective abstractions of the decomposition artifacts of a system. This means that even if a decomposition achieves a good locality with respect to all concerns involved, it is still cumbersome to reason about the properties of an artifact, based on the descriptions of its constituents.

Each decomposition artifact has its own specification and design. The design describes the way the artifact is built from lower-level interacting artifacts. The specification abstracts the essential properties that characterize these lower-level artifacts as a whole. In this way one can understand and reason about the artifact without having to consider its constituents and similarly one is able to reason about the interaction with other artifacts. To be effective, a specification of an artifact should describe the properties that are essential to understand the system as a whole. For real-time embedded systems this implies that next to structure one should also focus on behavior, timing, performance and accuracy properties. Furthermore, the specification of an artifact should be compact and precise and its design should not be too complex (implying that it contains a restricted number of artifacts and interactions). Finally, the specification and design should be consistent in the sense that their relation is clear and precise.

The effective use of abstractions in the design process brings many benefits. Unfortunately, these benefits are typically not experienced by industrial practitioners. Design documentation that is supposed to provide insight, is typically of a low abstraction level, is inconsistent, ambiguous and imprecise. Hence the design documentation does

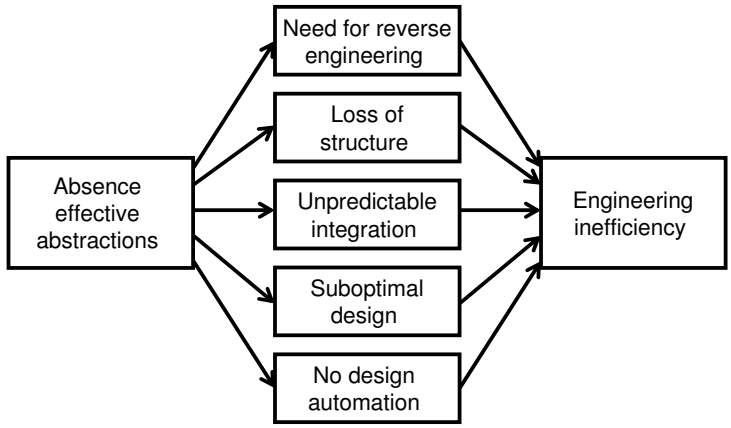


Figure 1.2: Consequences of ineffective abstractions.

not provide the required effective abstractions, leading to engineering inefficiency as shown in Figure 1.2.

- **Need for reverse engineering** In case a system has to be adapted (functionality or interactions have to be added or the performance has to be improved) one has to understand the 'big picture' of the design. This in order to determine how the change should be incorporated in such a way that the system remains structured and understandable. Typically only a few architects have this 'big picture' in their mind, but it is not explicitly available in the documentation and is not shared by the majority of designers. As a result designers spend a lot of time and effort in trying to (re-)construct this 'big picture'. Shedding light on this 'big picture' is precisely what effective abstractions are meant for. Existence of such abstractions would make reverse engineering less needed and more effective.

- **Loss of structure** An important goal of effective abstractions is to keep a system understandable by structuring it. Design artifacts are to be designed in such a way that they have limited interactions with and dependencies on other artifacts. This allows modifications to be carried out locally, e.g., within one or a few artifacts. However, if abstractions are not explicitly available or not consistent, the intended structure is very difficult to retrieve (see also the previous item). As a result dependencies are introduced that cross the intended artifact boundaries, a phenomenon sometimes referred to as architectural decay, and changes to one artifact can cause an unpredictable chain of required changes to other artifacts.

- **Unpredictable integration** Typically a lot of design documentation is produced, but this documentation is mainly in the form of text and structure diagrams,

which does not allow system behavior to be verified properly. The reason is that dynamic, concurrent, or real-time behavior is just too difficult to understand from textual documents and structure diagrams. In addition undocumented (hidden) dependencies between system modules may exist. As a result many design errors only show up during system integration when the system is actually used and the impact of the hidden dependencies becomes visible. System integration is typically late because all implementations of all components have to be ready. Early integration by mixing implementations and executable specifications is not supported if specifications are informal or ambiguous. Many of these problems can be avoided if adequate system abstractions are available.

- **Suboptimal design** Design solutions typically have a 'sweet spot' in which their performance/resource ratio is optimal. For instance, assigning a piece of functionality to embedded software or digital hardware in a clumsy way, can yield a complex solution that is expensive to build (both in terms of effort and material costs). Without proper abstractions and optimization tools the odds are low of designing a solution in or around this 'sweet spot'. Once a suboptimal design is obtained, it is very difficult to get rid of it by making a fundamentally different design. Organizational conservatism is a very important reason for this, but also the fact that such a major design step requires one to return to the original specifications (which are not explicitly present) and explore design alternatives (which is not supported). As a result, designers (have to) push the design performance while leaving the design architecture the same, thereby increasing complexity and drifting even further away from the sweet spot.

- **No design automation** Explicitly capturing the design intent in the form of precise abstractions allows the application of automated tools. Tools exist to verify whether a design behaves correctly, to predict performance and timing properties, to transform specifications into implementations and to explore design alternatives. These tools can have a huge impact on design efficiency, simply because they are fast and can produce reliable results in a reproducible way. Without them, a lot of manual work has be carried out, which is error-prone and time-consuming.

The major goal of model-driven engineering is to attack engineering inefficiency by introducing models as first-class citizens in the design trajectory. These models serve as explicit abstractions that are intended to complement traditional forms of design documentation. The research field of model-driven engineering together with the topics addressed in the Ideals project are explained in Section 1.5.

1.4 Aspect-oriented software design

Before we begin exploring the solution direction researched in the Ideals project, we can already formulate the first research question in this area:

Q-1 How relevant and real are the perceived problems with an idiom-based solution for crosscutting concerns, as depicted in Figure 1.1? How can we identify and quantify these problems?

A clear understanding of the problems caused by idiom-based solutions help in formulating the requirements and constraints to alternatives. A quantification of the problems helps to balance the cost of introducing an alternative to the benefits that can be gained.

1.4.1 AOSD in a nutshell

The goal of AOSD is to formally capture the interaction between the core concern code (called the *base program*) and the crosscutting concern code in an *aspect*: a modular implementation of the crosscutting concern. This interaction can be characterized by answering two questions: what should the crosscutting functionality do and when should it occur in a base program? The two answers form the two parts of an aspect: the *advice* captures what-should-be-done, the *pointcut* captures when-it-should-be-done.

In order for the advice to be truly independent from the base program, which allows it to be applied to many different base programs and thus solve the crosscuttingness need, it needs to have a very clear and limited interface or abstraction of the base program[4]. This interface is called the *joinpoint*. A joinpoint typically contains some generic abstractions that are available for every base program, by virtue of its chosen programming language(s), run-time environment(s) or coding standards. A joinpoint can also provide additional abstractions depending on the type or contents of the used pointcut. Figure 1.3 gives an overview of an aspect and how it relates to base programs. We will talk about the process of 'applying' an aspect in more detail in Section 1.4.3. Given this sketch of what AOSD is, we can formulate the second research question on this topic:

Q-2 (How) does AOSD contribute to a better handling of crosscutting concerns? How much does it help and is it practically useful in an industrial context? What are problems that may be introduced as a result of introducing AOSD?

1.4.2 Variability support in AOSD

An important part of research question Q-2 (Subsection 1.4.1) warrants extra attention: practical usefulness. In order for AOSD to be a useful paradigm in practice, it must be able to support a wide variety of crosscutting concerns and support variability within a single crosscutting concern. In industrial contexts, with large embedded systems, there will always be a need for slight adaptation of the implementation of a crosscutting concern for a specific domain, platform, (sub-)application or product life cycle phase. We will show how pointcuts and advices support this variation.

[4] If the advice would not need any interface to the base program, it is questionable whether the concern is truly crosscutting, since there seems to be no relation between the two implementations. In such a case, modularization can be achieved using more traditional decomposition techniques.

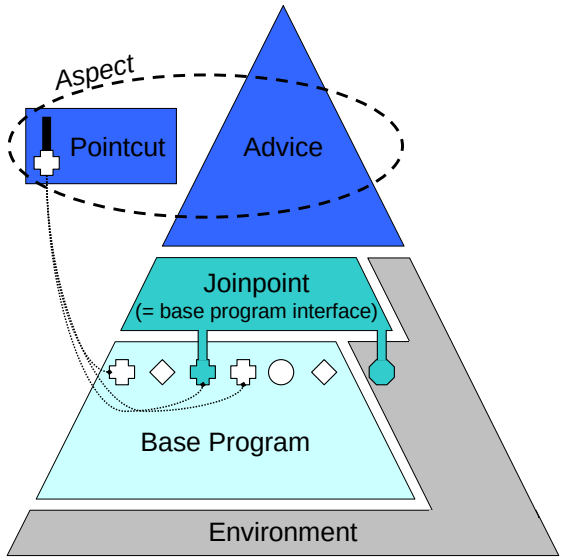


Figure 1.3: Parts of an aspect and their relations.

Pointcuts are a formal means to specify when an advice should be applied. A basic set of primitive properties is provided, together with a Boolean algebra to combine the primitives into more complex expressions. The wealth of the primitive properties determines the expressiveness of the pointcut formalism, and the amount of supported variability in specifying when advices should be applied. Examples of categories of primitives are:

- **Static** Also called syntactical or structural properties, this category contains primitives that relate to the definition of the base program: the entities (functions, variables, classes, et cetera) it consists of. Primitives can be used to select entities based on e.g., name (exact or matching a regular expression), type, scope, or any other static property. Primitives can also be used to query properties of entities (type, existence, size, canonical name, ...) for more complex selection criteria. In order to be broadly applicable, these primitives are based on the static program model of the program environment used or coding conventions that exist (and are expected to be used consistently).

- **Run-time** Also called semantical or dynamic properties, this category contains primitives that relate to the execution of the base program. Examples are prim-

itives to select functions executing within the calling hierarchy of another function, properties of the process or thread, or current values of variables and arguments. These primitives are usually based on the semantical model of the program environment used. These properties require run-time support for evaluating pointcuts to determine if a pointcut is applicable in the base program instance.

- **Meta-data** When a required concept has neither a consistent static representation nor a run-time identification, meta-data like annotations can be used to identify an entity. An example is when a domain concept like 'performance-critical function' cannot be directly linked to a language construct in the program environment or a consistent naming convention, all functions in the domain concept could be annotated with a specific annotation that asserts that they are performance-critical. Using the meta-data, an aspect can either be applied or rather be refrained from being applied to performance critical functions in a consistent and modular way.

An Advice expresses what the crosscutting functionality should do, i.e., it is a piece of code to be executed [5] in each joinpoint. An advice can be forced to use the same program environment as the base program, or, if the AOSD tool set allows this, it could also use a different program environment that is more suitable to the domain of the crosscutting concern. The expressiveness of advices is determined by the program environment used for the advice.

An advice should be as independent as possible from the base programs it will be added to later. As an example, an advice is free to use modules (or libraries or services) of the run-time system, independent from the ones used by the base program, but it should introduce the interfaces of the modules it depends on itself: it should not be dependent on the base program to provide these. Complete independence from the base program is usually impossible to achieve. Typically some information from the base program and the location where it is applied (like the name of a function or module) is needed in the advice. This interface between an advice and the base program is formalized in the joinpoint. There are three types of properties a joinpoint can provide to an advice:

- **Generic properties** These properties are automatically available to all advice code in every joinpoint. They are typically provided by the program environment used by the base program (e.g., every entity has a name, or was declared in a specific module or file), or by generic conventions (e.g., naming conventions may link the publicly available names of entities to a module name).

- **Pointcut-type specific properties** These properties are only available based on the type of the pointcut. As an example, pointcuts may identify functions (for which the arguments and return type is available) or identify variables (for which

[5] Strictly speaking an advice is not only executable code; it can also contain declarative code. However, with declarative code there are usually other mechanisms for modularization which are just as good or better to use, so it is questionable if aspects should be used for purely declarative advices.

the type and value is available). An advice may rely on these properties if it specifies the types of pointcuts it can be applied on. The availability of the property for all relevant pointcut types is again guaranteed by the program environment or by generic conventions.

- **Domain specific properties** Both types of properties so far rely on the program environment or conventions to ensure that a specific property is available. However, sometimes an advice needs an interface to a concept that the program environment does not support, but is domain specific. As an example, if an advice has the need to re-initialize the base program, it should require an initialization function in the interface of the joinpoint. This function can not be identified automatically, but must be identified explicitly by the pointcut[6].

We see that we have a number of options in supporting variability in both pointcuts and advices. The more categories we choose, the more complex the interaction between pointcuts and advices can become (especially for domain specific properties in advices, which require a precise way for advices and pointcuts to establish whether they are compatible), or between the base program and the aspect (especially for meta-data properties in pointcuts, which requires an extension to the base program environment to support annotations and the definition of these annotations). For each category we choose, we can further choose the specific properties supported in that category. Providing more categories and properties therein gives more expressiveness, at the price of greater complexity. We can formulate our third research question concerning AOSD as finding the balance between expressiveness and complexity:

Q-3 What is the required level of variability in crosscutting concerns in practice? What AOSD techniques for pointcut and advice expressiveness do we need to support this variability? How can these techniques be used in practice?

1.4.3 Applying AOSD in practice

So far, we have not discussed how the behavior of an aspect is actually added to the behavior of the base program. There are a number of alternatives for this:

- **Weaving** We can take the source code of the base program and add extra code to it that implements the behavior of the aspect. The combined program is then presented to the compiler to create an executable version of the base program including the aspect behavior. This *source level* combination of base program and aspect is called weaving, and it has a number of note-worthy characteristics:

[6] Another interesting example in this context is error handling. In a program environment with exception support, the means to signal an error is a generic property provided by the program environment. In a program environment without exception support, the means to signal an error becomes domain specific, for instance by assigning an error code to a specific variable. Which variable to use must then be captured by the pointcut and provided to the advice; advices that might want to report errors require a pointcut that guarantees them an error variable to use for this purpose.

  - It requires that the program language of the advice can be easily translated to the program language of the base program. Typically the program language of the advice would be the same as that of the base program, with a few extensions.
  - It has only a crude support for run-time properties in pointcuts, since weaving is done at compile time. It can support run-time properties by adding advice code in all possible locations and guarding them with checks that skip the advice code if the run-time requirements are not met, but this can incur severe performance penalties.
  - It can be easily compared to an idiom-based solution for a crosscutting concern, since both are visible in the source code. This makes it more easy to contrast the two approaches in terms of quality, effort, and run-time impact, and to debug the process.
  - It can easily support deployment to multiple target platforms using a single aspect tool set, by using portable code for the aspect weaving and different compilers after the weaving process.

- **Binary augmentation** We can also take the output from the compilation of the base program (either to native machine code or some type of byte code targeting a virtual machine) and add extra instructions to it that implement the behavior of the aspect. This *executable level* combination is called binary augmentation. In contrast to weaving, its characteristics are:

  - Some concepts from the program environment that are used in e.g., pointcuts may be difficult to extract reliably from the compiled base program (e.g., scoping rules, variable names or types and annotations). This is especially the case for native compiled code (as opposed to code compiled to target a virtual machine): instruction sets support less abstractions than high level languages.
  - The program language of the advice can be (very) different from the base program, as long as the advice can be compiled to the target platform.
  - It has the same difficulties with run-time properties as weaving.
  - It is more difficult to examine the impact of the aspects without using special tooling and target platform expertise. This may make debugging more complex, and the run-time impact more difficult to understand.
  - It can support aspects for different source languages using a single aspect tool set, if all these languages are compiled to the same platform.

- **Run-time interception** The third option is to perform a standard instrumentation of the base program (either through weaving or binary augmentation), independent of the actual aspect(s) to be applied to the base program. Then, at run-time, an aspect engine is used to intercept all interesting activities in the base program

and to execute the relevant advices. This option is similar in characteristics to binary augmentation, with the following exceptions:

- It has no problems with run-time properties.
- It is very flexible to add or change aspects, without changing anything to the binary of the base program.
- It is very expensive in terms of run-time overhead, due to the extra layer of the run-time aspect engine.

We consider source level weaving to be the best option for aspect-oriented software development in complex embedded systems. This is because low performance overhead and the possibility to understand and debug the impact of aspects at the programming language abstraction level, are considered of paramount importance. In a situation where a virtual machine is used and multiple source languages are used, binary augmentation could be considered, as it reduces the cost of the aspect tool set (at a possibly acceptable performance penalty).

As a result of the Ideals project, ASML started the design and implementation of a weaver for the C language that can be used within the ASML software. Although formally not part of the Ideals project, the project made use of the results of the research project in the area of Aspect Oriented Software Design (especially into the contribution and practical usability of AOSD), and was aimed at actually introducing an AOSD methodology and tool set into a complex embedded system. By organizing this project as a transfer project from research into the industry, thereby involving the research partners, the research project could in return learn from the insights and questions of the introduction project to trigger new research within the Ideals project. We will therefore include some of the results of this transfer project in this book, to answer the following research question:

Q-4 What are the important design constraints and quality attributes for an AOSD tool set for use in complex embedded systems?

1.4.4 Migration to an AOSD solution

Knowing a solution to the problems caused by an idiom-based way of implementing crosscutting concerns is immediately helpful for new developments that are not based on existing designs. However, usually complex embedded systems change or grow through evolution of existing designs (after all, this was the motivation for the Ideals project in the first place). It is therefore very important that any solution can be introduced into legacy designs (and implementations) in a controlled, and preferably automated, manner. This leads us to formulate the final research question in the area of AOSD:

Q-5 How can we support the migration of an idiom-based solution for crosscutting concerns to an AOSD based solution? Can we do this fully automatically?

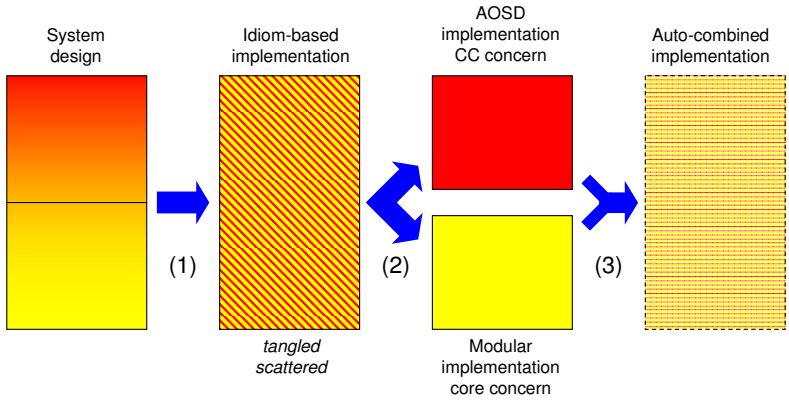


Figure 1.4: Migration path of an existing system to an AOSD based system.

In Figure 1.4 we depict the migration process of a system originally designed using an idiom-based solution (step 1), to one using an aspect-based solution and tool set (step 3). The central direction of the research was to find out if the same approach used to investigate the variability present in the existing system (to answer research question Q-3 in Subsection 1.4.2) could also be used to support the migration of the existing system, by splitting the existing code into the future base program and the aspect definition (step 2). Such a migration has to be performed with minimal risk of course, since testing complex embedded systems is very hard, especially when making the many changes related to changing a crosscutting concern's implementation. Using source level weaving, we can strive to obtain textual equivalence of original source code and the output of the weaver code, which would automatically prove the equivalence of the two solutions. Any method with a less than 100% proven equivalence would increase the (testing) cost of migration.

1.4.5 Research performed

The detailed results of the Ideals research into AOSD are described in the following chapters:

- Chapter 2 describes a method for studying idiom-based implementations of crosscutting concerns. In particular, it analyses a seemingly simple concern, tracing, and shows that it exhibits significant variability, despite the use of a prescribed idiom. It further discusses the consequences of this variability in terms of how AOSD could help prevent it, how it paralyzes (automated) migration efforts, and which aspect language features are required in order to obtain precise and concise

aspects. Hence, this chapter addresses research questions Q-3 and Q-5 (Pages 13 and 15).

- Chapter 3 addresses research question Q-1 (Page 10) by presenting an analysis of the use of an idiom-based solution for exception handling. In particular it focuses on evaluating the fault-proneness of this idiom: it presents a characterization of the idiom, a fault model accompanied by an analysis tool, and empirical data. The findings show that the idiom is indeed fault-prone, supporting the analysis that an idiom-based solution for crosscutting concerns leads to reduced quality.

- Chapter 4 discusses the so-called aspect interference problem, one of the remaining challenges of AOSD: aspects may interfere unexpectedly with the behavior of the base code or other aspects. Especially interference among aspects is difficult to prevent, as this may be caused solely by the composition of aspects that behave correctly in isolation. This chapter explains the problem of behavioral conflicts among aspects at shared join points, and illustrates it with 2 aspects found in the actual ASML software system. It presents an approach for the detection of behavioral conflicts that is based on a novel abstraction model for representing the behavior of an advice. Hence, this chapter addresses research question Q-2 (Page 10).

- Chapter 5 relates to research questions Q-2, Q-3 and Q-4 (Pages 10, 13 and 15), since it elaborates on the design of an industrial-strength AOSD system (a language and a weaver) for complex embedded software. It gives an analysis on the requirements of a general purpose AOSD language that can handle crosscutting concerns in embedded software, and a strategy on working with aspects in a large-scale software development process. It shows where established AOSD techniques fail to meet some of these requirements, and proposes new techniques to address them. In conclusion, it presents a short evaluation of the language and weaver as applied in the software development process of ASML. This chapter is the result of a joint project by the Ideals team and ASML to transfer knowledge from the Ideals research project into industry.

- In Chapter 10 it is shown what the impact of the research described in the above chapters is in practice, and how this impact was achieved. It thus addresses all the research questions Q-1 through Q-5.

1.5 Model-driven engineering

As explained in the previous section, the focus of AOSD techniques is on (embedded) software at the implementation level of abstraction. During the course of the Ideals project we tried to broaden this perspective by considering model-driven engineering (MDE) techniques. An important reason for this was the growing interest within ASML

to apply these techniques to improve the efficiency of the engineering process (see also Subsection 1.3.2).

In the engineering process, communication between engineers about various heterogeneous concerns takes place at various abstraction levels. The communication at the higher levels of abstraction usually manifests itself in the form of documents and drawings that vaguely relate to each other. At the lowest level, this communication manifests in the form of well related physical deliveries like boards, computers and byte code files. In the model-driven engineering vision, models will replace the higher level communication artifacts, enabling the systematic derivation of the lowest level artifacts. Hence MDE refers to the systematic use of models as primary engineering artifacts throughout the engineering life cycle.

MDE is an open approach that embraces various technological domains in a uniform way. In this view, other model-oriented initiatives, such as model-driven Architecture (MDA), domain-specific modeling (DSM), model-integrated Computing (MIC), model-driven software development (MDSD) and model-driven development (MDD), are concrete instances of MDE. To give an example, the Object Management Group's (OMG) MDA initiative [68] is a standardized MDE approach that specifies formalization and automation of a pre-defined development process, which is structured based on the PIM (Platform Independent Model) - PSM (Platform Specific Model) classification. Moreover, MDA relies on the OMG's modeling technologies, most notably the Meta Object Facility (MOF). The term MDE was first proposed and defined by Kent [63] as a generalization of MDA that includes the notion of development process and model space. According to Kent, a model space contains a particular set of models providing chosen perspectives on the system to be constructed, as well as appropriate mappings among these models. The model space and the development process are closely related: The artifacts or models developed by a particular process are intrinsic to the definition of that process and vice versa. In MDE, the notion of model space is extended beyond the abstraction dimension of the PIM-PSM classification. A number of generic dimensions of this space can be identified in the literature: abstraction, paradigm and concerns to name a few. The direct consequence of such a rich model space is heterogeneity of models in MDE.

Model-driven engineering thus implies dealing with a model space, typically consisting of a set of heterogeneous models. An example of how such a model space could look like[7] is shown in Figure 1.5. The squares depict the different models in the space. Dependent on the engineering discipline, models address different concerns. For instance, the software discipline focusses on the logic of an application, while the hardware discipline addresses the execution platform on which this application is deployed[8]. But even within one engineering discipline one may consider different concerns. For instance, the application logic typically consists of a part dealing with the

[7] This is only a preliminary idea of what such a model space might be. Charting the (desired) model space has only recently started at ASML.

[8] Model-driven engineering techniques typically make an explicit distinction between application logic platform and execution platform so that the platform and the application logic can evolve relatively independently [73].

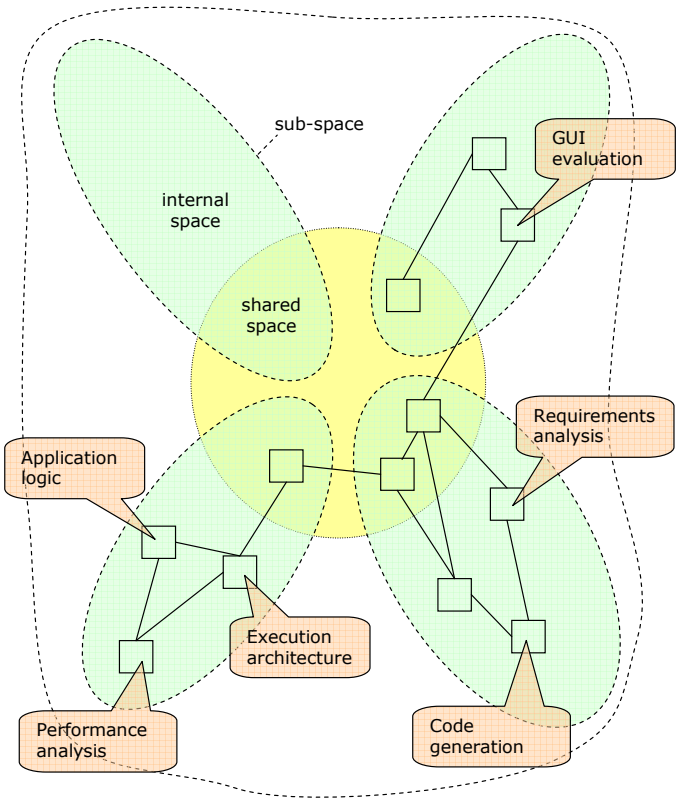


Figure 1.5: An impression of a model space.

flow of data through the system, while another part deals with the reactive control. Next to the focus on different concerns, models are made for different purposes. For instance, a model can be meant for design review, functional verification, timing and performance analysis, design-space exploration, refinement, code generation or testing.

Models are expressed in modeling languages supporting a so-called model of computation. Different models of computation focus on different concerns for different purposes. For instance, the flow of data through a system is well captured by Kahn process networks or synchronous data flow networks that allow the trade-offs between different deployments on the execution platform to be analyzed and that support an automatic mapping on this platform. On the other hand, (hierarchical) state-machine models excel in expressing, analyzing and synthesizing reactive event-driven behavior.

Models in a model space are related to each other. Relations, depicted as lines between the squares in Figure 1.5, can take many forms. For instance, a model can be an abstraction of another model by focusing on one specific concern. Vice versa, a model can be refinement of another model or might be automatically generated from another model. Also, models may be able to exchange information, for instance by passing messages.

When applying MDE in a Large Scale Industrial context (such as ASML) it is attractive to structure the model space in an hierarchical way. As shown in Figure 1.5 models are grouped into sub-spaces depicted as ovals. Sub-spaces have shared spaces that are visible to other sub-spaces and internal spaces that are only visible within the sub-space. Within one subspace, modeling languages are used that best suit the nature of the system modeled in that sub-space. The shared spaces allow relations to be defined between models in different sub-spaces. To make this feasible, formats and semantics of such shared models must be standardized in some way. For instance, one might require such models to conform to a standardized communication interface (an approach that is adopted by Ptolemy [86] and which is treated more formally in [54]). Another possibility is to apply only a set of standardized languages in the shared space, and have language transformations to transform to the specific models as used in the internal spaces (as described in [27]).

Clearly, model-driven engineering covers a vast research area with many challenging research questions:

Q-6 What models of computation (modeling languages and tools) are required to support the design of high-tech systems. What models play a role at what levels of abstraction? What languages should play in role in a shared space? Should we target one set of standardized languages or should we target a standardized communication interface?

Q-7 How to predict or analyze the properties of interest, especially when different models of computation are involved?

Q-8 How to keep models that involve the same concern consistent? For instance, how to keep models at different abstraction levels consistent?

Q-9 How to transform a model into a more refined one? How to weave models addressing different concerns together? How to do this in a predictable (property-preserving) way?

In the Ideals project we have only started to explore this area. At the start of this exploration, we did not have ‘the big picture’ of this field, nor could we articulate the proper research questions. Based on a number of case studies, each touching upon different concerns and purposes of model-driven engineering, the insight in the field grew. An important result of this exploration is the overview of the field you are currently reading. Detailed results are described in the following chapters:

- Chapter 6 focuses on the modeling of a coordination concern in a concise and formalized way. It is shown how such a model can be transformed automatically into a model expressed in terms of the execution platform primitives. This latter model can on its turn be transformed into executable code. Hence this chapter addresses research topics Q-6 and Q-9 as described above.

- Chapter 7 focuses on the modeling of a light control concern of a wafer scanner. Key issue is to capture the logic of this application and the underlying architecture in separate abstract executable models. By combining these models, a model suitable for analyzing the timing properties of the system is obtained. This model allows design trade-offs to be made in a systematic way. In addition such an application model can be transformed automatically into a (prototype) software implementation that runs on the target. The executable models are expressed in the POOSL language. To incorporate this language into a possible future MDE model space, a UML counterpart is being developed together with a UML to POOSL transformation. This transformation allows one to combine an application model created in UML with a platform model created in POOSL and analyze this combined model. Hence this chapter addresses research questions Q-6, Q-7 and Q-9 described above.

- Chapter 8 deals with a sequencing concern. The chapter introduces a technique to formally specify constraints on the possible sequences of function calls from a given program together with tools to check the consistency between multiple specifications and between a specification and an implementation. The focus of this chapter is thus on research questions Q-6 and Q-8.

- Chapter 9 focuses on the migration of supervisory machine control architecture towards an alternative approach based on task-resource models. This is done by capturing the essential control architecture information in model and by re-implementing this model based on the alternative approach. This chapter thus addresses research questions Q-6 and Q-9.

- In Chapter 10 it is shown what the impact of the research described in the above chapters is in practice, and how this impact was achieved. It thus addresses all the research questions Q-6 through Q-9.

Chapter 2

Simple crosscutting concerns are not so simple[1]

Authors: Magiel Bruntink, Arie van Deursen, Maja D'Hondt, Tom Tourwé

Abstract This chapter describes a method for studying idiom-based implementations of crosscutting concerns, and our experiences with it in the context of a real-world, large-scale embedded software system. In particular, we analyze a seemingly simple concern, tracing, and show that it exhibits significant variability, despite the use of a prescribed idiom. We discuss the consequences of this variability in terms of how aspect-oriented software development techniques could help prevent it, how it paralyzes (automated) migration efforts, and which aspect language features are required in order to obtain precise and concise aspects. Additionally, we elaborate on the representativeness of our results and on the usefulness of our proposed method.

2.1 Introduction

The lack of certain languages features, such as aspects or exception handling, can cause developers to resort to the use of idioms [2] for implementing crosscutting concerns. Idioms (informally) describe an implementation of required functionality, and can often be found in manuals, or reference code bodies. A well-known example is the *return-code idiom* we have studied in a realistic setting in [18]. It is used in languages such as C to implement exception handling. It advocates the use of error codes that are returned by functions when something irregular happens and caught whenever functions are

[1] This chapter is based on an article in the Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD'07) [13].

[2] Synonyms are code templates, coding conventions, patterns, et cetera.

invoked. Idioms are also used purposefully as a means of design reuse, for instance in the case of (design) patterns [19, 26].

Using idioms can result in various forms of code duplication [15]. Despite this duplication, idiom-based implementations are not guaranteed to be consistent across the software, however. Several factors may give rise to variability in the use of the idiom. Some variability, which is essential, occurs if there is a deliberate deviation from the idiom, for example in order to deal with specific needs of a subsystem, or to deal with special cases not foreseen in the idiom description. In addition to this, variability will occur accidentally due to the lack of automated enforcement (compilers, checking tools), programmer preference or skills, changing requirements and idiom descriptions, and implementation errors.

In this chapter, we are interested in the answer to the following question:

> *Is the idiom-based implementation of a crosscutting concern sufficiently systematic such that it is suitable for an aspect-oriented solution (with appropriate pointcuts and advice)?*

While answering this question is an endeavor too ambitious for this chapter, we do take an important step towards an answer by addressing the following sub questions: First, can we analyze the variability of the idiom-based implementation of a crosscutting concern? And secondly, can we determine the aspect language abstractions required for designing aspects that succinctly express the common part and the variability of a crosscutting concern?

We have encountered a number of examples of idiomatically implemented crosscutting concerns [15, 17, 18]. Several more are mentioned in the literature [25, 23]. The questions we ask in this chapter need to be answered in order to start migrating these crosscutting concerns to aspect-oriented solutions.

We present a generally-applicable method for analyzing the occurrence of variability in the idiom-based implementation of crosscutting concerns, that will help us answer these questions. We show the results of applying this method in order to analyze the *tracing* idiom in four selected components (ranging from 5 to 31 KLOC) of a 15 million Line C software system that is fully operational and under constant maintenance. Tracing is one of the ubiquitous examples from aspect-oriented software development (AOSD), and although it is a relatively simple idiom, we show that it exhibits significant and unexpected variability.

The structure of this chapter is as follows. The next section briefly presents our method for analyzing variability by describing each individual step. Sections 2.3–2.6 then describe how we applied each step on the selected components of our subject system in order to analyze the tracing idiom's variability. Section 2.7 then presents a discussion of the repercussions of these results. Section 2.8 presents our conclusions.

2.2 A method for analyzing idiom variability

This section proposes the general approach we use to acquire a deep understanding of the variability in the idiom-based implementation of a crosscutting concern, and explains how to use this understanding in subsequent aspect specification and design phases.

2.2.1 Idiom definition

The aim of this step is to provide a definition that is as clear and unambiguous as possible for the idiom that we want to study. The input for this (manual) step is typically found in the documentation accompanying the software, by means of code inspections, or by discussions with developers. In this respect, this step closely resembles the *Skim the Documentation, Read all the Code in One Hour* and *Chat with the Maintainers* patterns discussed in the *First Contact* cluster of [28].

While this step may seem simple, in our experience idiom descriptions in coding standard manuals often leave room for interpretation. When presenting our results, it happened more than once that developers started a heated debate on whether a particular use of the idiom was valid or not.

2.2.2 Idiom extraction

In this step, the code implementing the idiom is automatically extracted from the source code. This requires that the idiom code is recognized, and hence the output of the previous step is used as input for this step. The result of this step is similar to a slice [101], albeit that the extracted code does not necessarily need to be executable. Nevertheless, the extracted code can be compiled and analyzed by standard tools, and it is much smaller than the original code, allowing us to scale up to large systems.

Naturally, the complexity of this step is strongly dependent on the idiom: idioms that are relatively independent of the code surrounding them are easy to extract using simple program transformations, whereas idioms that are highly tangled with the other code require much more work.

2.2.3 Variability modeling

In this step, we describe which properties of the idiom can vary and indicate which variability we will target in our analysis. It is important to note that we do not require a description of variabilities that actually occur in the source code. We only need to know where we can expect variabilities, given the definition of the idiom. For example, variability in the tracing idiom under investigation can occur in the specific macro that is used to invoke the tracing functionality. In practice, it might turn out that the same macro is used consistently throughout the source code, or it might not.

Additionally, it is preferable to model different levels of variability separately in order to understand them fully, and subsequently to consider combinations. For example, in the tracing idiom there is the aforementioned variability in the way the tracing functionality is invoked, but also variability in the way the function parameters are converted to strings before being traced.

Finally, we do not require all possible variability to be modeled. As we discuss later, we only study part of the variability of the tracing idiom, while other parts are not considered. This is no problem if this is taken into account when discussing the results of the analysis. In other words, these results can be seen as a lower bound of the amount of variability that occurs.

2.2.4 Variability analysis

This step forms the core of our method, as it analyzes the variabilities actually present in the source code. This is achieved by taking the extracted idiom code, and analyzing it considering the variabilities that were modeled in the previous step. We are particularly interested in finding out how properties that can vary are typically related. For example, is it the case that tracing macro *m* is always invoked with either parameter c_1 or c_2, but never with c_3? Answering such questions can help us in designing the simplest aspect that captures all combinations as occurring in practice.

To analyze such relations between variable properties we use formal concept analysis (FCA) [44]. FCA is a mathematical technique for analyzing data which takes as input a so-called *context*. This context is basically a matrix containing a set of *objects* and a set of *attributes* belonging to these objects. The context specifies a binary relation that signals whether or not a particular attribute belongs to a particular object. Based on this relation, the technique finds maximal groups of objects and attributes — called a *concept* — such that

- each object of the concept shares the attributes of the concept;
- every attribute of the concept holds for all of the concept's objects;
- no other object outside the concept has those same attributes, nor does any attribute outside the concept hold for all objects in the concept.

Intuitively, a concept corresponds to a maximal 'rectangle' in the context, after permutation of the relevant rows and columns.

The resulting concepts form a lattice and therefore we can use relations between concepts, as well as characteristics of the concepts themselves, to get statistics and interpret the results.

2.2.5 Aspect design

In this step, we determine the required abstractions in aspect languages, which can be nearly directly distilled from the results of the variability analysis in the previous step. This step is discussed in detail in [13].

```
1  int f(chuck_id* a, scan_component b) {
2      int result = OK;
3      char* func_name = "f";
4      ...
5      trace(CC, TRACE_INT, func_name, "> (b = %s)",
6          SCAN_COMPONENT2STR(b));
7      ...
8      trace(CC, TRACE_INT, func_name, "< (a = %s) = %d",
9          CHUCK_ID_ENUM2STR(a), result);
10     return result;
11 }
```

Figure 2.1: Code fragment illustrating the tracing idiom at ASML.

2.3 Defining the tracing idiom

The idiom we study in the chapter is the ASML tracing idiom. Tracing is a seemingly simple idiom, used at development-time to facilitate debugging or any other kind of analysis. The base code is augmented with tracing code that logs interesting events (such as function calls), such that a log file is generated at runtime. The simplicity of the idiom is reflected in its simple definition: “Each function should trace the values of its input parameters before executing its body, and should trace the values of its output parameters before returning”

The ASML documentation describes the basic implementation version of the idiom, which looks as in Figure 2.1. The `trace` function is used to implement tracing and is a variable-argument function. The first four arguments are mandatory, and specify the following information:

1. the component in which the function is defined;
2. whether the tracing is internal or external to that component;
3. the function for which the parameters are being traced;
4. a `printf`-like format string that specifies the format in which parameters should be traced.

The way in which each of these four parameters should be passed on to the `trace` function is described by the standard, but not enforced. For example, some components follow the standard and use the `CC` constant, which always holds the component’s name, to specify the name, while others actually hardcode the name with a string representing the name (as in `"CC3"`). Similarly, the `func_name` variable should be used to specify the name of the function whose parameters are being traced. Since `func_name` is a local variable, however, different functions might use different names for that variable (`f_name`, for instance). The structure of the format string is also not fixed, and developers are thus free to construct strings as they like.

The optional arguments for `trace` are the input or output parameters that need to be traced. If these parameters are of a complex type (as opposed to a basic type like `int` or `char`), they need to be converted to a string representation first. Often, a dedicated function or macro is defined exactly for this purpose. In Figure 2.1, `SCAN_COMPONENT2STR` and `CHUCK_ID_ENUM2STR` are two such examples. Developers can choose to trace individual fields of struct instead of using a converter function, however.

Although the idiom described above is the standard idiom, some development teams define special-purpose tracing macro's, as a wrap around the basic idiom. These macro's try to avoid code duplication by filling in the parameters to `trace` in the standard way beforehand. Typically, tracing implementations by means of such macro's thus require fewer parameters, although sometimes extra parameters are added as well, for example to include the name of the file where tracing is happening.

It should be clear from this presentation that the tracing idiom precisely prescribes what information should be traced, but that the way in which this information is provided is not specified. Hence, we can expect a lot of variability, as we will discuss in Section 2.5.

2.4 Extracting the tracing idiom

Extraction of the tracing idiom out of the source code is achieved by using a combination of a code analysis tool, called CodeSurfer,[3] and a code transformation tool, called ASF+SDF [6]. The underlying idea is that the analysis tool is used to identify all idiom-related code in the considered components and that this information is passed on to the transformation tool that extracts the idiom code from the base code. The end result is a combination of the base code without the idiom-related code, and a representation of the idiom code by itself.

2.5 Modeling variability in the tracing idiom

Tracing is generally considered as a very simple example of a crosscutting concern that can be captured in an aspect easily. This is confirmed by the fact that we can express the requirements for tracing in one single sentence, and hence we could expect an aspect to be simple as well. However, the tracing idiom we consider here is significantly more complex than the simple example often mentioned and than the requirement would reveal. Rather, it represents a good example of what such an at first sight simple idiom looks like in a real-world setting.

The following characteristics of the tracing idiom distinguish it from a simple logging concern:

- A simple logging aspect typically weaves in log calls at the beginning and end of a function, and often only logs the fact that the function has been entered

[3] www.grammatech.com

and has been exited. The tracing idiom described above also logs the values of actual parameters and the module in which the function is defined. Moreover, it differentiates between input and output parameters, which have to be traced differently.

- Tracing the values of actual parameters passed to a C function is a quite complex matter. Basic types such as `int` or `bool` can be printed easily, but more complex types, such as `struct` and `enum`, are a different story. These should be converted to a string-based representation first, which differs for different instances of `struct` and `enum`. Moreover, certain fields of a struct may be relevant in the context of a particular function, but may not be relevant elsewhere. Hence, the printed value depends on the context in which the type is used, and not only on the type itself.

- The conversion of complex types to a string representation is quite different in C than in Java, or any other modern programming language. C does not provide a default `toString` function, as do all Java classes, for example. Consequently, a special-purpose converter method for complex types needs to be provided explicitly. Additionally, since C does not support overloading of function names, each converter function needs to have a unique name.

These issues, together with the way tracing is invoked as explained in Section 2.3, show that variability can occur at many different levels. In the remainder of this chapter, however, we will focus on *function-level* and *parameter-level* variability. The variability present on those levels possibly has the biggest impact on the definition of aspects for the tracing concern.

At the function-level, the variability occurs in the specific way the tracing functionality is invoked. This depends on four different properties: the name of the tracing function that is used (for example `trace`), the way the component name and the function name are specified (by using `CC` and `func_name`, for example), and whether internal or external tracing is used. More properties are considered when a different tracing idiom requires more parameters when it is called, for example the name of the file in which the traced function is defined.

At the parameter-level, the variability involves the different ways in which a parameter of a particular kind is traced. This level of variability is discussed in detail in [13], and not considered further in this chapter.

2.6 Analyzing the tracing idiom's variability

As shown in Table 2.1, our experiments involve 4 different components, comprising 83,000 lines of non-white lines of C code. These components define 704 functions in total. The table also lists the different number of ways in which tracing is invoked, i.e., the different tracing macros that are used, as well as the different component names and function names that are specified. The numbers clearly show the variability present in

	CC1	CC2	CC3	CC4	global
LOC	29,339	17,848	31,165	4,985	83,337
functions	328	134	174	68	704
tracing macro's	1	1	2	1	2
component names	2	3	1	2	6
function names	3	1	1	1	3

Table 2.1: Basic statistics of the analyzed components.

the idiom at the function level, since globally 2 different tracing macro's, 6 different ways to specify the component name and 3 different ways for specifying the function name are used.

The goal of our analysis is to identify, at the function level, which functions invoke tracing in the same way. Analyzing this allows us to make headway into answering our key question, since it shows us where the implementation is systematic and what is variable. Since FCA, introduced in Section 2.2.4, is capable of identifying meaningful groupings of elements, we use it in our variability analysis.

The FCA algorithm needs to be set up before it can be applied, i.e., we need to define the objects and attributes of the input context. The next subsection explains how this is achieved for our experiment. Subsequently we describe the results of running FCA on each of the components separately, as well as on all components together. This will allow us to discuss the variability within a single component, as well as the between different components.

2.6.1 Setting up FCA for analyzing tracing

We first explain how objects and attributes are chosen for our experiment, and how we run the FCA algorithm. Afterwards, we explain how we interpret the results.

Objects and attributes

For studying function-level variability, the objects and attributes are chosen such that all functions that invoke tracing in the same way are grouped. Hence, the objects we use in the FCA context are the names of all functions defined in the components we consider. The attributes are different instantiations of the four properties used to invoke tracing, as discussed in Section 2.3. A sample context is shown in Table 2.2.

Applying FCA

Once the context is set up, the algorithm can be applied. We use Lindig's Concepts tool to compute the actual concepts [74]. The context is specified in a file in a specific format, which we generate using ASF+SDF and the extracted tracing representation files. The tool can output the resulting concepts in a user-defined way, and we tune the

	trace	CC_TRACE	TRACE_INT	TRACE_EXT	CC	func_name	f_name
f	√	-	√	-	√	√	-
g	-	√	-	-	-	√	-
h	√	-	-	√	√	-	√
i	-	√	-	-	-	√	-
j	√	-	√	-	√	√	-

Table 2.2: Example FCA context for function-level variability.

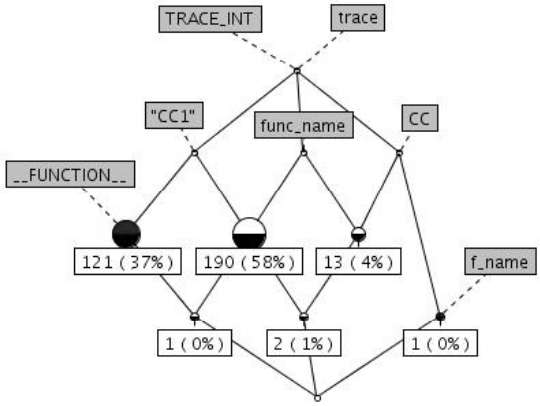


Figure 2.2: Function-level variability in the CC1 component.

results so that they can be read into a Scheme environment. This allows us to reason about the results using Scheme scripts.

An alternative is to use the ConExp tool[4], which requires a slightly different input format, but that can visualize the concepts (and the resulting lattice) so that it can be inspected easily. The graphical representations of lattices in this chapter are obtained by this tool.

Interpreting the results

From running the FCA algorithm, we obtain a concept lattice that shows the different concepts identified and the relation between them. An example lattice appears in Figure 2.2. Each dot in the lattice represents a concept, and the lines connecting the dots

[4]http://conexp.sf.net

represent concepts that are related because they share objects and/or attributes.

While traversing a lattice from top to bottom, following the edges that connect concepts, attributes are gradually added to the concepts, and objects are removed from them. The top concept contains all objects and all attributes shared by all objects (if any), whereas the bottom concept contains all attributes and all objects shared by all attributes (if any). At some point in the lattice, a concept contains objects that are not contained within any of its sub-concepts. Those objects are the concept's *own objects*. The attributes associated with the own objects of a concept are always "complete", in the sense that in the input context passed to the FCA algorithm, the own objects precisely are related to precisely those attributes.

A concept with own objects represents a single variant for invoking tracing, or a single variant for converting a particular type. In the first case, for example, the own objects are functions, all these functions share the same (complete) set of attributes, an no other attribute is shared by these functions. In Figure 2.2, the concepts with own objects are denoted by nodes whose bottom half is colored black and whose size is proportional to the number of own objects they contain. They also have white labels indicating the number of own objects and the percentage of own objects with respect to the total number of objects of the concept. The largest concepts contains 190 own object, which are functions in this case.

We observe that a particular kind of variability occurs when either input and output tracing in the same function are invoked in a different way, or a single type is converted using two different converter functions. Such situations, which are in most cases clearly examples of accidental variability, immediately show up in the concept lattice. They are embodied by concepts with own objects that have at least one parent concept with own objects. Indeed, such concepts have more attributes than is necessary, hence some of these attributes are different variations for the same property. As an example, consider again Figure 2.2 and observe the two concepts in the lower left part that contain 1 and 2 own objects, respectively. From their positions in the lattice, it can be derived that the leftmost concept uses both `__FUNCTION__` and `func_name` for specifying the function name when tracing, and the other concept `"CC1"` and `CC` for specifying the component name.

2.6.2 Function-level variability

The upper half of Table 2.3 presents the results of analyzing the function-level variability in the four components we consider. The first row of data contains the total number of concepts that are found by the FCA algorithm. The second row lists the number of different tracing invocations that are found (i.e., the total number of concepts containing own objects). The third row then lists the number of functions that implement the standard tracing idiom as described in ASML's coding standards (i.e., the number of own objects found in the concept with attributes `trace`, `CC`, `TRACE_INT` or `TRACE_EXT` and `func_name`), and the last row presents the percentage of those functions with respect to the total number of functions in the component.

	CC1	CC2	CC3	CC4	total	global
Function-level variability						
#concepts	11	6	24	2	43	47
#tracing variants	6	4	19	2	31	29
#functions w. std. tracing	13	1	26	0	40	40
% of total functions	4	0.7	15	0		5.7

Table 2.3: Function-level variability results for 704 functions.

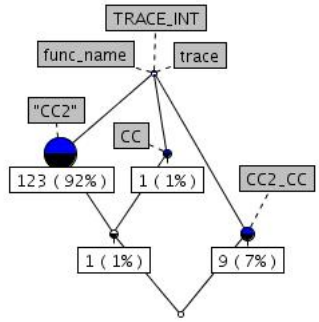


Figure 2.3: Function-level variability in the CC2 component.

The most striking observation revealed by these results is that only 5.7% (40 out of 704) of all functions invoke tracing in the standard way, as described in Section 2.3. This immediately raises the question why developers do not adhere to the standard. Maybe a new standard for invoking tracing should be considered? Can we observe candidate standards in our results?

Looking at the second row in the upper half of Table 2.3, we see that 29 different tracing variants are used in the four components. If we consider each component separately, we find 31 variants in total. This difference can be explained by the fact that 3 components invoke tracing according to the standard idiom, and that the functions of these components doing so are all grouped in one single concept when considering the components together. This results in one concept replacing three other concepts, hence the reduction with two concepts. Reversing this reasoning also means that there is no other way of invoking tracing that is shared by different components, or in other words, all components invoke tracing by using their own variant(s). Consequently, we can not select one single variant that can be considered as the standard among these 29 variants, with the other variants being simple exceptions to the general rule. This is confirmed by looking at the lattices.

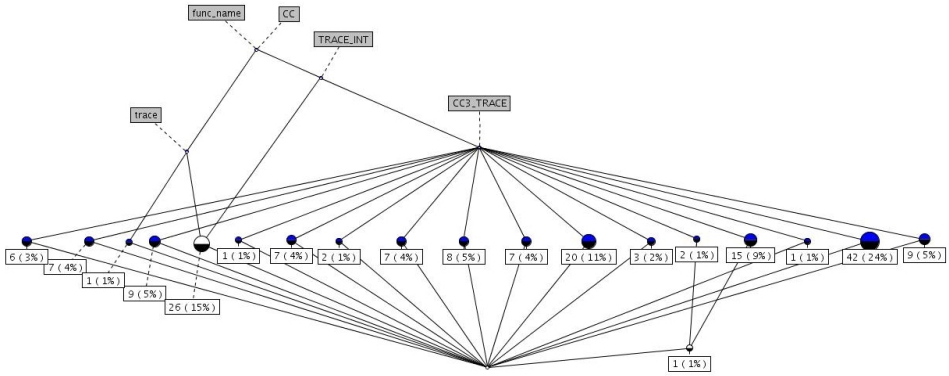


Figure 2.4: Function-level variability in the CC3 component.

Looking at Figures 2.2 and 2.3, it is clear that both components use a similar tracing variant implemented by most functions (190 or 58% functions in the case of CC1, 123 functions or 92% in the case of CC2). Additionally, CC1 has yet another "big" variant that uses the `__FUNCTION__` preprocessor token instead of the variable `func_name`. This variant is used in 121 functions (37%).

Figures 2.4 and 2.5 show significantly different results. The CC4 component implements only two tracing variants, implemented by 31 and 37 functions respectively. The difference between the two variants is that one is an extension of the other: one variant uses `CC4_LINE` to denote the component name, whereas the other uses both `CC4_LINE` and `CC4_CC`. The CC3 component implements 19 different variants, and none can be selected as the most representative or resembles the variants of another component. The variability in this case stems from the fact that the CC3 component defines its own macro for invoking tracing, and that this macro requires one extra argument, namely the name of the file in which is defined the function that is being traced. This is clearly visible in the lattice: each concept corresponding to a specific tracing variant that corresponds to a specific file in the source code, contains an extra attribute that denotes the constant used in the trace call corresponding with the file. Interestingly, although CC3 defines its own macro, it is also the component that uses the standard idiom the most. Whether the mixing of the standard idiom with the dedicated macro is a deliberate choice or not is an issue that remains to be discussed with the developers.

Summarizing, we can state that very few functions implement the standard tracing variant, that no other standard variant can be identified that holds for all components, but that within one single component a more common variant can sometimes be detected.

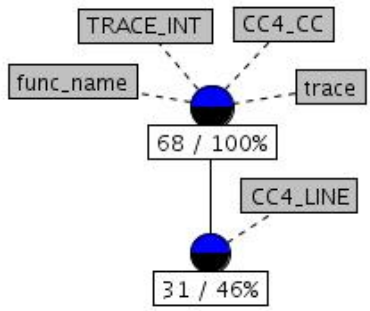


Figure 2.5: Function-level variability in the CC4 component.

The previous subsection discussed an example of accidental variability in the CC1 component. A similar situation occurs in the CC2 component, as can be seen in Figure 2.3, where one function uses `CC` and `"CC2"`. The CC3 component contains one variant that is accidental, as confirmed by the ASML developers, consisting of a copy/paste error when passing a constant representing the file name in invoking the `CC3_trace` macro.

2.7 Discussion and evaluation

This section discusses the implications of variability caused by idiom-based development from the perspective of the migration of legacy systems. Whereas the discussion in the previous section concerned the essential variability, the discussion here is based on the occurrence of accidental variability. First, however, we discuss the consequences of taking into account additional variability that was not considered in our analysis.

2.7.1 Further variability

It is important to note that we have only considered function-level and parameter-level [5] variability in our experiments, and in our discussion above. However, the tracing idiom has other characteristics that we did not analyze in depth, and these characteristics make the idiom richer. Hence, more features might be needed in an aspect language then the ones we described above if we wish to express ASML's tracing idiom in an aspect.

For example, ASML code distinguishes between input and output parameters. Our analysis did not make that distinction and considered input and output tracing together.

[5] See [13] for the results of the parameter-level variability analysis.

Although this allowed us to detect accidental variabilities that we would not have discovered otherwise, it also prevented us from considering the impact on an aspect implementation. An aspect needs to know which parameters are input and which are output in order to construct the appropriate input and output trace statements. An aspect weaver could extract such information from the source code using data-flow analysis, and could make it available in the aspect language, for example.

Other characteristics that we did not consider but that are relevant for such a discussion include the position of the input and output trace statements in the original code (do they always occur right at the beginning and at the end of a function's execution?), the tracing of other variables besides parameters (such as local and/or global variables), the order in which the parameters are traced, and the format string that is used, together with the format types for parameters contained within that string.

Clearly, the results we obtained can thus be seen as a lower bound of the real amount of variability present in the tracing idiom's implementation. Since the variability we found was considerable already, we arrive at our claim that simple crosscutting concerns do not exist, at least not for software systems of industrial size.

2.7.2 Migration of idioms to aspects

Given that an aspect-oriented solution has benefits over an idiom-based solution, it is relevant to study the risks involved in migrating the idiom-based implementation to an aspect-oriented implementation.

In general, migrating code of an operational software system is a high-risk effort. Although one of the biggest contributors to this risk is the scale of the software system, in our case this can be dealt with by approaching the migration of tracing incrementally [8], for instance on a component-per-component basis. However, other sources of risk need to be accounted for: the migrated code is of course expected to be functionally equivalent to the original code.

Our findings concerning variability of idiom-based concern implementations introduce an additional risk dimension. In particular, accidental variability is a complicating factor. Ignoring such variabilities by defining an aspect that only implements the essential variability means we would be changing the functionality of the system. A particular function that does not execute tracing as its first statement but only as its second or third statement, might fail once an aspect changes that behavior, for example, when originally a check on null pointers preceded the tracing of a pointer value. So this risk is real even with functionality that is seemingly side-effect free, as is the tracing concern, and will become higher when the functionality does involve side-effects.

On the other hand, migrating the idiom including its accidental variability is undesirable as well: aspect-oriented languages are not well-equipped for expressing accidental variability and the resulting aspect-oriented solution quickly converges to a *one-aspect-per-function* solution. So the issue boils down to a trade-off between minimizing the risk on the one hand, and on the other hand reducing the variability in favor of uniformity, in order to reach a reasonable aspect-oriented solution.

At the moment, we do not have an answer to the question how to migrate idioms of legacy systems with a high degree of accidental variability — at this point we do not even know what a *high degree* of accidental variability is, nor do we know whether automated migration towards aspects is feasible at all in practice, if a simple aspect such as tracing already exposes difficult problems. This discussion only serves to point out that the risk is present and that there are currently no processes or tools available for minimizing the risks. Nevertheless, we can say that in the particular context of ASML, the initial proposal for dealing with the migration risk is to (1) confirm or refute the detected accidental variability, (2) eliminate the confirmed accidental variability in the idiom-based implementation of the legacy system incrementally and test if the resulting implementation is behavior-preserving by comparing the compiled code, (3) remove the remaining idiom-based implementation of the crosscutting concern, and (4) represent the idiom and its essential variability as aspects.

2.8 Concluding remarks

In this chapter, we have studied "tracing in the wild" using idiom-based development. It turns out that for systems of industrial size, tracing is not as simple as one might think: in the code we analyzed, the idiom used for implementing the tracing concern exhibits remarkable variability. Part of this variability is accidental and due to typing errors or improper use of idioms, which could be seen as a plea for using aspect-oriented techniques. A significant part of the variability, however, turns out to be essential: aspects must be able to express this variability in pointcuts or advice. Even with our partial analysis of the variability of the so-called "trivial" tracing concern, we discover the need for quite general language abstractions that probably no aspect language today can provide entirely, and certainly not in the context of an industrial system. This will only worsen when more variability is considered or more complex concerns are investigated.

In summary, this chapter makes the following contributions:

1. We proposed a method to assess the variability in idiom-based implementation of (crosscutting) concerns.

2. We have shown how existing tools for source code analysis and transformation, and for formal concept analysis can be combined and refined to support the variability analysis process.

3. We presented the results of applying the method on selected components of a large-scale software system, showing that significant variability is present.

4. We discussed the implications of the accidental variability caused by idiom-based development in the context of crosscutting concerns from the perspective of legacy system migration.

Chapter 3

Discovering faults in idiom-based exception handling [1]

Authors: Magiel Bruntink, Arie van Deursen, Tom Tourwé

Abstract In this chapter, we analyse the exception handling mechanism of a state-of-the-art industrial embedded software system. Like many systems implemented in classic programming languages, our system uses the popular return-code idiom for dealing with exceptions. Our goal is to evaluate the fault-proneness of this idiom, and we therefore present a characterization of the idiom, a fault model accompanied by an analysis tool, and empirical data. Our findings show that the idiom is indeed fault prone, but that a simple solution can lead to significant improvements.

3.1 Introduction

A key component of any reliable software system is its exception handling. This allows the system to detect errors, and react to them correspondingly, for example by recovering the error or by signalling an appropriate error message. As such, exception handling is not an optional add-on, but a sine qua non: a system without proper exception handling is likely to crash continuously, which renders it useless for practical purposes.

Despite its importance, several studies have shown that exception handling is often the least well understood, documented and tested part of a system. For example, [97] states that more than 50% of all system failures in a telephone switching application

[1] This chapter is based on an article in the Proceedings of the 28th International Conference on Software Engineering (ICSE 2006) [18].

are due to faults in exception handling algorithms, and [75] explains that the Ariane 5 launch vehicle was lost due to an un-handled exception.

Various explanations for this phenomenon have been given.

First of all, since exception handling is not the primary *concern* to be implemented, it does not receive as much attention in requirements, design and testing. [91] explains that exception handling design degrades (in part) because less attention is paid to it, while [21] explains that testing is often most thorough for the ordinary application functionality, and least thorough for the exception handling functionality. Granted, exception handling behavior is hard to test, as the root causes that invoke the exception handling mechanism are often difficult to generate, and a combinatorial explosion of test cases is to be expected. Moreover, it is very hard to prepare a system for all possible errors that might occur at runtime. The environment in which the system will run is often unpredictable, and errors may thus occur for which a system was not prepared.

Second, exception handling functionality is crosscutting in the meanest sense of the word. [76] shows that even the simplest exception handling strategy takes up 11% of an application's implementation, that it is scattered over many different files and functions and that it is tangled with the application's main functionality. This has a severe impact on understandability and maintainability of the code in general and the exception handling code in particular, and makes it hard to ensure correctness and consistency of the latter code.

Last, older programming languages, such as C or Cobol, that do not explicitly support exception handling, are still widely used to develop new software systems, or to maintain existing ones. Such explicit support makes exception handling design easier, by providing language constructs and accompanying static compiler checks. In the absence of such support, systems typically resort to systematic coding idioms for implementing exception handling, as advocated by the well-known *return code* technique, used in many C programs and operating systems. Idioms are also referred to as 'boilerplate code', 'template', 'pattern', or 'recipe' in programming jargon. On the one hand, they are fragments of code that occur frequently and are repetitive, hence tedious, to reproduce. For example, even for the most simple C program that produces any output one has to explicitly include the standard input/output library. On the other hand, they represent common, well-tested and scrutinized solutions to frequently occurring programming problems. Design patterns [43] are examples of the latter case. At both extremes, the use of idioms is a manual implementation technique, and hence can be considered to be a fault-prone and effort intensive practice compared to automatic code generation. As shown in [17], such idioms are not scalable and compromise correctness.

In this chapter, we focus on the exception handling mechanism of a wafer scanner The system consists of approximately 15 million lines of C code, and is developed and maintained using a state-of-the-art development process. It applies (a variant of) the return code idiom consistently throughout the implementation. The central question we seek to address is the following: 'how can we reduce the number of implementation faults related to exception handling implemented by means of the return code idiom?'.

In order to answer this general question, a number of more specific questions needs to be answered:

1. What kinds of faults can occur? Answering this question requires an in-depth analysis of the return code idiom, and a fault model that covers the possible faults to which the idiom can lead. A fault model is a model that is used in software testing to specify which faults can occur in a program. A fault model describes how faults can lead to failures.

2. Which of these faults do actually occur in the code? A fault model only predicts which faults can occur, but does not say which faults actually occur in the code. By carefully analyzing (automatically) the subject system, an estimate of the probability of a particular fault can be given.

3. What are the primary causes of these faults? The fault model explains *when* a fault occurs, but does not explicitly state *why* it occurs. Because we need to analyse the source code in detail for detecting faults, we can also study the causes of these faults, as we will see.

4. Can we eliminate these causes, and if so, how? Once we know why these faults occur and how often, we can come up with alternative solutions for implementing exception handling that help developers in avoiding such faults. An alternative solution is only a first step, (automated) migration can then follow.

We believe that answers to these questions are of interest to a broader audience than the original developers of our subject system. Any software system that is developed in a language without exception handling support will suffer the same problems, and guidelines for avoiding such problems are more than welcome. In this chapter we offer experience, an analysis approach, tool support, empirical data, and alternative solutions to such projects.

This chapter first discusses the return code idiom and obtains a precise characterization of the idiom in Section 3.2. Section 3.3 further formalizes this characterization into a fault model. Next, Section 3.4 presents SMELL, a source code analysis tool that is capable of detecting the faults defined in the fault model. We applied SMELL to several ASML components and report on the results in Section 3.5. A discussion of the results follows in Section 2.7. Finally, Section 3.7 concludes the chapter.

3.2 Characterizing the return code idiom

The central question we seek to answer is how we can reduce the number of faults related to exception handling implemented by means of the return code idiom. To arrive at the answer, we first of all need a clear description of the workings of (the particular variant of) the return code idiom at ASML. We use an existing model for exception handling mechanisms (EHM) [69] to distinguish the different components

```
int f(int a, int* b) {
   int r = OK;
   bool allocated = FALSE;
   r = mem_alloc(10, (int *)b);
   allocated = (r == OK);
   if((r == OK) && ((a < 0) || (a > 10))) {
      r = PARAM_ERROR;
      LOG(r,OK);
   }
   if(r == OK) {
      r = g(a);
      if(r != OK) {
         LOG(LINKED_ERROR,r);
         r = LINKED_ERROR;
      }
   }
   if(r == OK)
      r = h(b);
   if((r != OK) && allocated)
      mem_free(b);
   return r;
}
```

Figure 3.1: Exception handling idiom.

of the idiom. This allows us to identify and focus on the most error-prone components in the next sections. Furthermore, expressing our problem in terms of this general EHM model makes it easier to apply our results to other systems using similar approaches.

An exception at ASML is ‘any abnormal situation found by the equipment that hampers or could hamper the production’. Exceptions are logged in an *event log*, that provides information on the machine history to different stakeholders (such as service engineers, quality assurance department, et cetera).

The EHM itself is based on two requirements:

1. a function that detects an error should log that error in the event log, and recover it or pass it on to its caller.

2. a function that receives an error from a called function must provide useful context information (if possible) by *linking* an error to the received error, and recover the error or pass it on to the calling function.

An error that is detected by a function is called a *root* error, while an error that is linked to an error received from a function is called a *linked* error.

If correctly implemented, the EHM produces a tree of related consecutive errors in the event log. This tree is referred to as the *error link tree*, and resembles a exception chain produced by the Java virtual machine.

Because the C programming language is used, and C does not have explicit support for exception handling, each function in the source code follows the *return code* idiom. Figure 3.1 shows an example of such a function. We will now discuss this approach in more detail.

An exception representation *defines what an exception is and how it is represented*. A *singular* representation is used, in the form of an *error variable* of type `int`. Line 2 in Figure 3.1 shows a typical example of such an error variable, that is initialized to the `OK` constant. This variable is used throughout the function to hold an *error value*, i.e., either `OK` or any other constant to signal an error. The variable can be assigned a constant, as in Lines 7 and 14, or can be assigned the result of a function call, as in Lines 4, 11 and 18. If the function does not recover from an error itself, the value of the error should be propagated through the caller by the `return` statement (Line 21).

Exception raising is *the notification of an exception occurrence*. Different mechanisms exist, of the *explicit control-flow transfer* variant is used: if a root error is encountered, the error variable is assigned a constant (see Lines $6-9$), the function logs the error, stops executing its normal behavior, and notifies its caller of the error.

Logging occurs by means of the `LOG` function (Line 8), where the first argument is the new error encountered, which is linked to the second argument, that represents the previous error value. The function treats root errors as a special case of linked errors, and therefore the root error detected at Line 8 is linked to the previous error value, `OK` in this case.

Explicit guards are used to skip the normal behavior of the function, as in Lines 10 and 17. These guards check if the error variable still contains the `OK` value, and if so, execute the behavior, otherwise skip it. Note that such guards are also needed in loops containing function calls.

If the error variable contains an error value, this value propagates to the `return` statement, which notifies the callers of the function.

3.3 A fault model for exception handling

Based on the characterization presented in the previous section, we establish a fault model for exception handling by means of the return code idiom in this section. The fault model defines when a fault occurs, and includes failure scenarios which explain what happens when a fault occurs.

The return code idiom relies on the fact that when an error is received, the corresponding error value should be logged and should propagate to the `return` statement. The programming language constructs that are used to implement this behavior are function calls, return statements and log calls.

We define three categories of faults:

Category 1: A function raises a new error but fails to perform appropriate logging.

Category 2: A function properly links a new error value *y* to the received error value *x*, but then fails to return the new error value *y* (return *z* instead).

Category 3: A function receives an error value *x* and does not link a new error value to *x* in the log, but does return an error value *y* that is different from *x*.

For a detailed description of the fault model we refer to [18].

3.4 SMELL: Statically detecting exception handling faults

Based on the fault model we developed SMELL, the *State Machine for Error Linking and Logging*, which is capable of statically detecting violations to the return code idiom in the source code, and is implemented as a CodeSurfer[2] plug-in. We want to detect faults statically, instead of through testing as is usual for fault models, because early detection and prevention of faults is less costly [5, 20], and because testing exception handling is inherently difficult.

3.4.1 Implementation

SMELL statically analyses executions of a function in order to prove the truth of any one of the logic formulas of our fault model. The analysis is static in the sense that no assumptions are made about the inputs of a function. Inputs consist of formal or global variables, or values returned by called functions.

We represent an execution of a function by a finite path through its control-flow graph. Possibly infinite paths due to recursion or iteration statements are dealt with as follows. First, SMELL performs intra-procedural analysis, i.e., the analysis stays within a function and does not consider functions it may call. Therefore recursion is not a problem during analysis. Intra-procedural analysis does not impact SMELL's usefulness, as it closely resembles the way developers work with the code: they should not make specific assumptions about possible return values, but should instead write appropriate checks after the call. Second, loops created by iteration statements are dealt with by caching analysis results at each node of the control-flow graph. We discuss this mechanism later.

The analysis performed by SMELL is based on the evaluation of a deterministic (finite) state machine (SM) during the traversal of a path through the control-flow graph. The SM inspects the properties of each node it reaches, and then changes state accordingly. A fault is detected if the SM reaches the *reject* state. Conversely, a path is free of faults if the SM reaches the *accept* state.

The error variable is a central notion in the current implementation of SMELL. An error variable, such as the `r` variable in Figure 3.1, is used by a programmer to keep track of previously raised errors. SMELL attempts to identify such variables automatically based on a number of properties. Unfortunately, the idiom used for exception handling does not specify a naming convention for error variables. Hence, each programmer picks his or her favorite variable name, ruling out a simple lexical identification of these variables. Instead, a variable qualifies as an error variable in case it satisfies the following properties:

- it is a local variable of type `int`,
- it is assigned only constant (integer) values or function call results,

[2] www.grammatech.com

- it is not passed to a function as an actual parameter, unless in a log call,
- no arithmetic is performed using the variable.

Note that this characterization does not include the fact that an error variable should be returned or that it should be logged. We deliberately do not want the error variable identification to depend on the correct use of the idiom, as this would create a circular dependency: in order to verify adherence to the idiom, the error variable needs to be identified, which would need strict adherence to the idiom to start with.

Most functions in the source base use at most one error variable, but in case multiple are used, SMELL considers each control-flow path separately for each error variable. Functions for which no error variable can be identified are not considered for further analysis. We discuss the limitations of this approach at the end of this section.

The definition of the SM was established manually, by translating the informal rules in the manuals to appropriate states and transitions. Describing the complete SM would require too much space. Therefore we limit our description to the states defined in the SM, and show a subset of the transitions by means of example runs.

The following states are defined in the SM:

Accept and **Reject** represent the absence and presence of a fault on the current control-flow path, respectively.

Entry is the start state, i.e., the state of the SM before the evaluation of the first node. A transition from this state only occurs when an initialization of the considered error variable is encountered.

OK reflects that the current value of the error variable is the OK constant. Conceptually this state represents the absence of an exceptional condition.

Not-OK is the converse, i.e., the error variable is known to be anything but OK, though the exact value is not known. This state can be reached when a path has taken the true branch of a guard like `if(r != OK)`.

Unknown is the state reached if the result of a function call is assigned to the error variable. Due to our limitation to intra-procedural analysis, we conservatively assume function call results to be unknown.

Constant is a parametrised state that contains the constant value assigned to the error variable. This state can be reached after the assignment of a literal constant value to the error variable.

All states also track the error value that was last written to the log file. This information is needed to detect faults in the logging of errors. Since we only perform intra-procedural analysis, we set the last logged value to *unknown* in the case of an Unknown state (i.e. when a function was called). We thus assume that the called function adheres to the idiom, which allows us to verify each function in isolation. Faults in these called functions will still be detected when they are being checked.

While traversing paths of the control-flow graph of a function, the analysis caches results in order to prevent infinite traversals of loops and to improve efficiency by eliminating redundant computations. In particular, the state (including associated values of parameters) in which the SM reaches each node is stored. The analysis then makes sure that each node is visited at most once given a particular state. The same technique is used by Engler *et al.* in [39].

3.4.2 Example faults

The following three examples show how the SM detects faults from each of the categories in the fault model. States reached by the SM are included in the examples as comments, and where appropriate the last logged error value is mentioned in parentheses. First, consider the code snippet in Figure 3.2.

```
1 int calibrate(int a) { // Entry
2   int r = OK;          // OK
3   r = initialise();    // Unknown
4   if(a == 1)
5     LOG(RANGE_ERROR, OK); // Reject
6   ...
7 }
```

Figure 3.2: Example of fault category 1.

A fault of category 1 possibly occurs on the path that takes the true branch of the `if` statement on Line 4. If the function call at Line 3 returns with an error value, say `INIT_ERROR`. The call to the LOG function on Line 5 logs RANGE_ERROR to the OK value, and since OK is different from `INIT_ERROR`, erroneous logging has been done, and a fault of category 1 occurs.

SMELL detects this fault as follows, starting in the Entry state on Line 1. The initialization of `r`, which has been identified as an error variable, causes a transition to the OK state on Line 2. The assignment to `r` of the function call result on Line 3 results in the Unknown state. On the true branch of the `if` statement on Line 4, a (new) root error is raised. The cause of the fault lies here. SMELL reaches the Reject state at Line 5 because if an error value (`INIT_ERROR`) would have been returned from the call to the `initialise` function, it is required to link the `RANGE_ERROR` to the `INIT_ERROR`, instead of linking to OK.

The function in Figure 3.3 exhibits a fault of category 2 on the path that takes the true branch of the `if` statement. Again, suppose an INIT_ERROR comes out of the function call at Line 3, then the function correctly performs LOG(`ALIGN_ERROR`, `INIT_ERROR`). The fault consists of the function not returning `ALIGN_ERROR`, but `INIT_ERROR`, because after linking to the received error, the new error value is not assigned to the error variable.

Again SMELL starts in the Entry state, and subsequently reaches the OK state after the initialization of the error variable `r`. The `initialize` function is called at Line

```
1 int align() {        // Entry
2   int r = OK;        // OK
3   r = initialise();  // Unknown
4   if(r != OK)        // Not-OK
5     LOG(ALIGN_ERROR, r); // Not-OK (ALIGN_ERROR)
6   return r;          // Reject
7 }
```

Figure 3.3: Example of fault category 2.

3, and causes SMELL to enter the Unknown state. Taking the true branch at Line 4 implies that the value of `r` must be different from `OK`, and SMELL records this by changing to the Not-OK state. At Line 5 an `ALIGN_ERROR` is linked to the error value currently stored in the `r` variable. SMELL then reaches the return statement, which causes the error value to be returned that was returned from the `initialize` function call at Line 3. Since the returned value differs from the logged value at this point, SMELL transits to the Reject state.

```
1 int process(int a) { // Entry
2   int r = OK;       // OK
3   r = initialise(); // Unknown
4   if(a == 2) {
5     r = PROCESS_ERROR; // Reject
6   }
7   ...
8   return r;
9 }
```

Figure 3.4: Example of fault category 3.

Category 3 faults are similar to category 2, but without any logging taking place. Suppose that an INIT_ERROR comes out of the function call at Line 3 in Figure 3.4. For the path taking the true branch of the `if` statement a value different from `INIT_ERROR` will be returned, i.e., `PROCESS_ERROR`.

Until the assignment at Line 5 the SM traverses through the same sequence of states as for the previous examples. However, the assignment at Line 5 puts SMELL in the Reject state, because the previously received error value has been overwritten. A category 3 fault is therefore manifest.

3.4.3 Fault reporting

SMELL reports the presence of faults using a more fine-grained view of the source code than the fault model. While the fault model takes a black box perspective, i.e., regarding behavior only at the interface level, SMELL reports detected faults using a white box perspective, i.e., considering the implementation level details of a function.

The white box perspective is considered to be more useful when interpreting actual fault reports, which developers may have to process.

In the following we present a list of 'low-level faults', or programmer mistakes, that SMELL reports to its users. For each programmer mistake we mention here the associated fault categories from the fault model. SMELL itself does not report these categories to the user. To help users interpreting the reported faults, SMELL prints the control-flow path leading up to the fault, and the associated state transitions of the SM.

function does not return occurs when a function declares and uses an error variable (i.e., assigns a value to it), but does not return its value. If present, SMELL detects this fault at the return statement of the function under consideration. This can cause category 2 or 3 faults.

wrong error variable returned occurs when a function declares and uses an error variable but returns another variable, or when it defines multiple error variables, but only returns one of them and does not link the others to the returned one in the appropriate way. This can cause category 2 or 3 faults.

assigned and logged value mismatch occurs when the error value that is returned by a function is not equal to the value last logged by that function. This can cause category 2 faults.

not linked to previous value occurs when a `LOG` call is used to link an error value to a previous value, but this latter value was not the one that was previously logged. If present, SMELL detects this fault at the call site of the log function. This causes category 1 faults.

unsafe assignment occurs when an assignment to an error variable overwrites a previously received error value, while the previous error value has not yet been logged. Clearly, if present SMELL detects this fault at the assignment that overwrites the previous error value. These faults can be category 1, 2 or 3.

3.4.4 Limitations

Our approach is both formally unsound and incomplete, which is to say that our analysis proves neither the absence nor the presence of 'true' faults. In other words, both false negatives (missed faults) or false positives (false alarms) are possible. False negatives for example occur when SMELL detects a fault on a particular control-flow path, and stops traversing that path. Consequently, faults occurring later in the path will go unnoticed. The unsoundness property and incompleteness properties do not necessarily harm the usefulness of our tool, given that the tool still allows us to detect a large number of faults that may cause much machine down-time, and that the number of false positives remains manageable. The experimental results (see Section 3.5) show that we are currently within acceptable margins.

	reported	false positives	limitations	validated
CC1 (3 kLoC)	32	2	4	26 (13)
CC2 (19 kLoC)	72	20	22	30
CC3 (15 kLoC)	16	0	3	13
CC4 (14.5 kLoC)	107	14	13	80
CC5 (15 kLoC)	9	1	3	5
total (66.5 kLoC)	236	37	45	154 (141)

Table 3.1: Reported number of faults by SMELL for five components.

3.5 Experimental results

3.5.1 General remarks

Table 3.1 presents the results of applying SMELL on 5 relatively small components. The first column lists the component that was considered together with its size, column 2 lists the number of faults reported by SMELL, column 3 contains the number of false positives we manually identified among the reported faults, column 4 shows the number of SMELL limitations (as discussed in the previous section) that are encountered and automatically recognized, and finally column 5 contains the number of validated faults, or 'true' faults.

Four of the five components are approximately of the same size, but there is a striking difference between the numbers of *reported* faults. The number of reported faults for the CC3 and CC5 components is much smaller than those reported for the CC2 and CC4 components. When comparing the number of *validated* faults, the CC4 component clearly stands out, whereas the number for the other three components is approximately within the same range.

Although the CC1 component is the smallest one, its number of validated faults is large compared to the larger components. This is due to the fact that a heavily-used macro in the CC1 component contains a fault. Since SMELL is run after macro expansion, a fault in a single macro is reported at every location where that macro is used. In this case, only 13 faults need to be corrected (as indicated between parenthesis), since the macro with the fault is used in 14 places.

The number of validated faults reported for the CC5 component is also interestingly low. This component is developed by the same people responsible for the EHM implementation. As it turns out, even these people violate the idiom from time to time, which shows that the idiom approach is difficult to adhere to. However, it is clear that the CC5 code is of better quality than the other code.

Overall, we get 236 reported faults, of which 45 (19 %) are reported by SMELL as a limitation. The remaining 191 faults were inspected manually, and we identified 37 false positives (16 % of reported faults). Of the remaining 154 faults, 141 are unique, and so in other words, we found 2.1 true faults per thousand lines of code.

3.5.2 Fault distribution

A closer look at the 141 validated faults shows that 13 faults are due to a function not returning, 28 due to the wrong error variable being returned, 54 due to unsafe assignments, 10 due to incorrect logging, and 36 due to an assigned and logged value mismatch.

The *unsafe assignment* fault occurs when the error variable contains an error value that is subsequently overwritten. This kind of fault is by far the one that occurs the most (54 out of 141 = 38%), followed by the *assigned and logged value mismatch* (36 out of 141 = 26%). If we want to minimize the exception handling faults, we should develop an alternative solution that deals with these two kinds of faults.

Accidental overwriting of the error value typically occurs because the control flow transfer when the exception is raised is not implemented correctly. This is mostly due to a forgotten guard that involves the error variable ensuring that normal operation only continues when no exception has been reported previously. An example of such a fault is found in Figure 3.4.

The second kind of fault occurs in two different situations. First, as exemplified in Figure 3.3, when a function is called and an exception is received, a developer might link an exception to the received one, but forgets to assign the linked exception to the error variable. Second, when a root error is detected and a developer assigns the appropriate error value to the error variable, he might forget to log that value.

3.5.3 False positives

The number of false positives is sufficiently low to make SMELL useful in practice. A detailed look at these false positives reveals the reasons why they occur and allows us to identify where we can improve SMELL.

Of the 37 false positives identified, 23 are due to an incorrect identification of the error variable, 7 are due to SMELL getting confused when multiple error variables are used, 4 occur because an infeasible path has been followed, and 3 false positives occur due to some other (mostly domain-specific) reason.

These numbers indicate that the largest gain can be obtained by improving the error variable identification algorithm, for example by trying to distinguish error variables from ‘ordinary’ error variables. Additionally, they show that the issue of infeasible paths is not really a large problem in practice.

3.6 Discussion

In our examples, we found 2.1 deviations from the return code idiom per 1000 lines of code. In this section, we discuss some of the implications of this figure, looking at questions such as the following: How does the figure relate to reported defect densities in other systems? What, if anything, does the figure imply for system reliability? What does the figure teach us on idiom and coding standard design?

Representativeness A first question to be asked is to what extent our findings are representative for other systems.

The software under study has the following characteristics:

- It is part of an embedded system in which proper exception handling is essential.
- Exception handling is implemented using the return code idiom, which is common for C applications.
- Before release, the software components in question are subjected to a thorough code review.
- The software is subjected to rigorous unit, integration, and system tests.

In other words, we believe our findings hold for software that is the result of a state-of-the-art development process and that uses an exception handling mechanism similar to the one we considered.

The reason so many exception handling faults occur is that current ways of working are not effective in finding such faults: tool support is inadequate, regular reviews tend to be focused on 'good weather behavior' — and even if they are aimed at exception handling faults these are too hard to find, and testing exception handling is notoriously hard.

Defect density What meaning should we assign to the value of 2.1 exception handling faults per 1000 lines of code (kLoC) we detected?

It is tempting to compare the figure to reported defect densities. For example, an often cited paper reports a defect density between 5 and 10 per kLoC for software developed in the USA and Europe [36]. More recently, in his ICSE 2005 state-of-the-art report, Littlewood states that studies show around 30 faults per kLoC for commercial systems [77].

There are, however, several reasons why making such comparisons is questionable, as argued, for example, by [41]. First, there is neither consensus on what constitutes a defect, nor on the best way to measure software size in a consistent and comparable way. In addition to that, defect density is a product measure that is derived from the process of finding defects. Thus, 'defect density may tell us more about the quality of the defect finding and reporting process than about the quality of the product itself' [41, p.346]. This particularly applies to our setting, in which we have adopted a new way to search for faults.

The consequence of this is that no conclusive statement on the relative defect density of the system under study can be made. We cannot even say that our system is of poorer quality than another with a lower reported density, as long as we do not know whether the search for defects included a hunt for idiom errors similar to our approach.

What we can say, however, is that a serious attempt to determine defect densities should include an analysis of the faults that may arise from idioms used for dealing with crosscutting concerns.

Reliability We presently do not know what the likelihood is that an exception handling fault actually leads to a failure, such as an unnecessary halt, an erroneously logged error value, or the activation of the wrong exception handler. As already observed by Adams in 1984, more faults need not lead to more failures [1]. We are presently investigating historical system data to clarify the relation between exception handling faults and their corresponding failures. This, however, is a time consuming analysis requiring substantial domain knowledge in order to understand a problem report, the fault identified for it (which may have to be derived from the fix applied) and to see their relation to the exception handling idiom.

Idiom design The research we are presenting is part of a larger, ongoing effort in which we are investigating the impact of crosscutting concerns on embedded C code [14, 17]. The traditional way of dealing with such concerns is by devising an appropriate coding idiom. What implications do our findings have on the way we actually design such coding idioms?

One finding is that an idiom making it too easy to make small mistakes can lead to many faults spread all over the system. For that reason, idiom design should include the step of constructing an explicit fault model, describing what can go wrong when using the idiom. This will not only help in avoiding such errors, but may also lead to a revised design in which the likelihood of certain types of errors is reduced.

A second lesson to be drawn is that the possibility to check idiom usage automatically should be taken into account: static checking should be designed into the idiom. As we have seen, this may otherwise require complex analysis. Our analysis requires computationally expensive program dependence graphs, while it would be much preferable to suffice with relatively simple abstract syntax trees.

3.7 Concluding remarks

Our contributions are summarized as follows. First, we provided empirical data about the use of an exception handling mechanism based on the return code idiom in an industrial setting. This data shows that the idiom is particularly error prone, due to the fact that it is omnipresent as well as highly tangled, and requires focused and well-thought programming. Second, we defined a series of steps to regain control over this situation, and answer the specific questions we raised in the introduction. These steps consist of the characterization of the return code idiom in terms of an existing model for exception handling mechanisms, the construction of a fault model which explains when a fault occurs in the most error prone components of the characterization, the implementation of a static checker tool which detects faults as predicted by the fault model, and the introduction of an alternative solution, based on experimental findings, which is believed to remove the faults most occurring.

We feel these contributions are not only a first step toward a reliability check component for the return code idiom, but also provide a good basis for (re)considering

exception handling approaches when working with programming languages without proper exception handling support. We showed that when designing such idiom-based solutions, a corresponding fault model is a necessity to assess the fault-proneness, and the possibility of static checking should be seriously considered.

Chapter 4

Detecting behavioral conflicts among crosscutting concerns

Authors: Pascal Dürr, Lodewijk Bergmans, Mehmet Akşit

Abstract Aspects have been successfully promoted as a means to improve the modularization of software in the presence of crosscutting concerns. Within the Ideals project, aspects have been shown to be valuable for improving the modularization of idioms (see also Chapter 1). The so-called *aspect interference problem* is considered to be one of the remaining challenges of aspect-oriented software development: aspects may interfere with the behavior of the base code or other aspects. Especially interference among aspects is difficult to prevent, as this may be caused solely by the composition of aspects that behave correctly in isolation. A typical situation where this may occur is when multiple advices are applied at the same, or *shared*, join point. In this chapter we explain the problem of behavioral conflicts among aspects at shared join points, illustrated by aspects that represent idioms: *Parameter checking* and *Error propagation*. We present an approach for the detection of behavioral conflicts that is based on a novel abstraction model for representing the behavior of advice. The approach employs a set of conflict detection rules which can be used to detect both generic conflicts as well as domain or application specific conflicts. One of the benefits of the approach is that it neither requires the application programmers to deal with the conflict models, nor does it require a background in formal methods for the aspect programmers.

4.1 Introduction

Aspect-Oriented Programming (AOP) aims at improving the modularity of software in the presence of crosscutting concerns. AOP languages allow independently programmed aspects to *superimpose* behavior (so-called *advice*) on a base program. Unfortunately, the increased composition possibilities may also cause undesired emerging behavior. This is not necessarily due to a wrong implementation of the individual aspects; the composition of independently programmed aspects may cause emerging conflicts. These conflicts are caused by unexpected behavioral interactions. The most common situation where this occurs is when multiple advices are superimposed at the same join point; we call this a *shared join point*. Note that interference between aspects may also occur in other places without shared join points, but here we concentrate on this case. We define a *behavioral conflict* as emerging behavior that conflicts with the originally intended behavior (cf. requirements) of one or more of the involved modules.

Conventional techniques for guarding consistency are not equally applicable to aspects. This is mainly because aspect composition is *implicit*: each aspect is defined independently of the others, potentially at different times and by different people. The composition of advices can happen 'by coincidence', certainly the programmers of the individual aspects cannot always be aware that this will happen. Also, tracing a possible conflict back to an aspect specification, can become hard using conventional techniques. Recently, reasoning about the correctness of a system after superimposing multiple aspects at the same or *shared join point* [80], has been considered as an important problem to address [71, 72, 53].

In this chapter we explain and motivate the problem of behavioral conflicts with an example from idioms used by ASML, represented by aspects (Section 4.2). The chapter presents an approach to the detection of behavioral conflicts that is applicable for most, if not all, AOP languages (Section 4.3). We also discuss the application of this method to the Composition Filters [3] approach whose declarative approach to aspect definitions improves automatic detection of behavioral conflicts (Section 4.4).

4.2 Motivation

There are numerous examples of behavioral conflicts between aspects, see for example [32]. In this section we present an example that is based on the idioms applied within ASML's wafer scanner software.

We present here two aspects[1] that we have identified, namely *Parameter Checking* and *Error Propagation*. Currently, *Parameter Checking* has indeed been implemented –in part of the system– as an aspect using *WeaveC*. This is not (yet) the case for *Error Propagation*.

[1] Please note that the example aspects presented here are slightly altered for reasons of confidentiality. However, this does not affect the essence of the examples.

4.2.1 Parameter checking

Parameter Checking verifies pre- and post-conditions on parameters of C functions. Parameters can be one of three types; input, output and in- and output. This distinction depends on whether a parameter is read, written or both. Two checks are employed to verify the validity of the parameters. First, function input and in- and output pointer parameters, should not be empty (i.e., not *null*) at the start of a function. If the input parameter pointer is *null*, it could yield a fatal error whenever this parameter is accessed. Second, every output pointer parameter, must be *null* at the start of a function. An output parameter is a pointer to a memory location that is written in the function body. If such a parameter points to a memory location that is already in use, this might accidentally override data, which is undesired. An example of the parameter checking concern, implemented as idiom, applied to the function *compare_data()* is shown in Listing 4.1.

```
1  static int compare_data(
2      const DATA_struct*       p1,
3      const DATA_struct*       p2,
4      bool*                    changed_ptr)
5  {
6     int result = OK;
7    /* Check preconditions */
8     if (result == OK && p1 == NULL)
9     {   result = INVALID_INPUT_PARAMETER_ERROR; }
10    if (result == OK && p2 == NULL)
11    {   result = INVALID_INPUT_PARAMETER_ERROR; }
12    if (result == OK && changed_ptr != NULL)
13    {   result = INVALID_OUTPUT_PARAMETER_ERROR; }
14    // code that compares the structures and sets the changed_ptr boolean
          accordingly
15    return result;
16 }
```

Listing 4.1: Example of Parameter Checking code.

This function compares the two input parameters *p1* and *p2*, and sets the boolean output parameter *changed_ptr* accordingly. At Lines 8 to 13, the checks for the input and output parameters are shown. Typically, the parameter checking concern accounts for around 7% of the number of statements in the code, although the exact percentage varies among components.

4.2.2 Error propagation

The C programming language does not offer a native exception handling mechanism. The typical way to implement exception handling in C is to use the return value of a function. The function returns *OK* in case of success and an error number in case of failure. This means that the caller of the function should always inspect the return value and verify that it is *OK*. If not, it should either handle the error or return the error to its caller.

Error Propagation includes: (a) passing the error state through a so-called *error variable* and as the return value of the function, (b) ensuring that no other actions are

performed in an error state, and (c) logging error states. Listing 4.2 shows an example of such an exception handling scheme.

```
1  static int compare_data(
2      const DATA_struct*        p1,
3      const DATA_struct*        p2,
4      bool*                     changed_ptr)
5  {
6     int result = OK;
7     if (result == OK)
8     {
9        result = example_action1(...);
10         if(result != OK)
11         {   LogError(result); }
12    }
13    return result;
14 }
```

Listing 4.2: Example of Error Propagation code.

The code in Listing 4.2, first (Line 6) initializes the error variable, *result*, to hold the current error state. To determine whether to continue with normal execution, a check is placed which guards the execution (Line 7). In this case this might seem useless as the error variable already contains OK, however these are coding guideline templates, and as the code evolves such a check might be required if another statement is inserted before Line 7. Next, a call to a regular function ($example\_action1(...)$) is performed (Line 9). If an error is detected (Lines 10-11), this error is logged. Finally the *error variable* is returned at Line 13.

It is out of the scope of this chapter to elaborate on the alternatives for exception handling. It is however obvious to see that this exception handling idiom contributes substantially to the amount of code. Depending upon the component, this may even be up to 25% of the number of lines of code. The error handling domain can be divided into three main elements: detection, propagation and handling. We will focus here on propagation. Detection and handling of errors is highly context dependent, and thus re-factoring this into an aspect is hard, and perhaps not even desirable. Error propagation on the other hand follows a more common pattern which can be re-factored into an aspect more easily.

4.2.3 An aspect-based design

We will now discuss how to refactor the concerns above into an AOP solution. Aspect *ParameterChecking* should check the input and output pointer parameters of each function to ensure the contract of the function. We implement this functionality as an advice, called *check*. Aspect *ErrorPropagation* implements the following elements: check whether we are not in erroneous state and if so execute the original call. If this call yields an error state, it must be logged. Similar to the checking concern, we also implement the functionality of propagation as an advice, called *propagate*. Figure 4.1 illustrates the application of both aspects to a (base) system.

At the top of the picture the two aspects and their advices are shown, namely, *check* and *propagate*. The figure also shows our example C function, *compare_data(...)* as

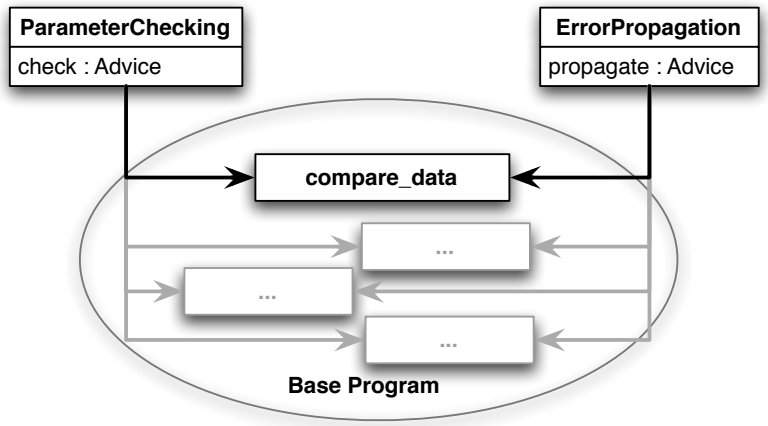


Figure 4.1: Parameter checking and error propagation aspects on a base program.

one of the functions that form the base system. The arrows show where each advice is applied. In this case, both advices are superimposed on the same join point, the function *compare_data()*. However, as the aspects implement coding conventions, there are many such shared join points. This is indicated by the gray arrows and rectangles. Now assume we would execute *propagate* before *check*; in this case, the errors detected by *check* are never propagated to the caller.

If we examine this conflict more carefully we see that it is caused by an interaction between the two advices. The *propagate* advice reads the *error variable* to determine the current error state and can subsequently write the *error variable* and log if an error is detected. Advice *check* verifies that the arguments are valid, and possibly sets the *error variable*. In this case the presence of the conflict depends on a specific ordering of advices, but there are conflicts where the ordering does not matter.

Now let us elaborate more on the concerns and the conflict between them. Individually, both aspects are consistent with their requirements and therefore they can be considered sound. From the language compiler point of view, the program with either orderings of advices can be considered as a valid program with no errors, there are no syntactical or structural problems. However, once these aspects are applied at the same join point, new behavior emerges. Such new behavior may be undesired behavior, in which case we call it a composition (-caused) conflict.

It is therefore necessary to develop techniques and tools that can statically (at compile time) reason about the (potential) behavioral conflicts between aspects, to avoid unexpected behavior during the execution of the system. However, there are aspects which rely on dynamic information. Statically checking such aspects may not be sufficient, and a runtime extension is required to detect these dynamic conflicts. In [35], we present such an extension, but this is out of the scope of this chapter.

In the remainder of this chapter we present an approach to detect behavioral con-

flicts that is generally applicable for most, if not all, AOP languages. This has been worked out for both composition filters (as implemented in the Compose* language and tools) and AspectJ. The approach is also independent of the base language, and applies equally well to Java or C# as it does to C.

4.3 Methodology

To reason about the behavior of advices and detect behavioral conflicts between them, we need to introduce a formalization that enables us to express behavior, and conflict detection rules over that behavior. A formalization of the complete behavior of advices in general would be too complicated to achieve and to reason with. Therefore, we propose an abstraction that can represent the essential behavior of advices, but without too many details, such that it can be used to detect behavioral conflicts between advices.

Our approach is based on a *resource-operation* model to represent the relevant semantics of advice. This is a simple abstraction model that can represent both concrete, low-level, behavior as well as abstract high-level behavior. This resembles the idea of Abstract Data Types [62]: representing an abstraction through its operations. Our approach to conflict detection resembles the Bernstein conditions [4] for stating concurrency requirements. A similar approach is also used for detecting and resolving (concurrency) conflicts in transaction systems, such as databases [78]. However, our approach generalizes these domain-specific approaches.

A conflict among advices can only occur if there is an interaction between them; we model this interaction as operations on one or more shared resources. A conflict can then be modeled as the occurrence of a certain pattern of operations on a shared resource. In the remainder of this chapter we will explain the model intuitively, based on the previously presented example. [29] offers a formal description of the resource model.

Figure 4.2 presents the semantic analysis process and the relationships to the base system and advice. We use this image as a guideline through Sections 4.3.1 to 4.3.3.

4.3.1 Pointcut designator analysis

At the top of Figure 4.2, one can see a set of aspects. These aspects contain advices and pointcut designators. There is also a base program with a set of classes (*ClassA*...*ClassZ*). The aspects and base system specifications are inputs of the *Pointcut Designator Analysis*(PDCA) phase. During this phase all pointcut designators are evaluated with respect to the base program. This results in a set of join points with a mapping to all superimposed advice(s). For conflict analysis, we only need to consider join points with more than one super-imposed advice.

4.3.2 Abstraction

The result of phase PCDA is a sequence of advices per shared join point[2]. This sequence is used in the next phase: *Advice Behavior Abstraction*. The other input for

[2]In the case that the ordering is not, or only partially, known, we can select one specific ordering (e.g., the one that the aspect compiler would choose), or iterate over all valid orderings.

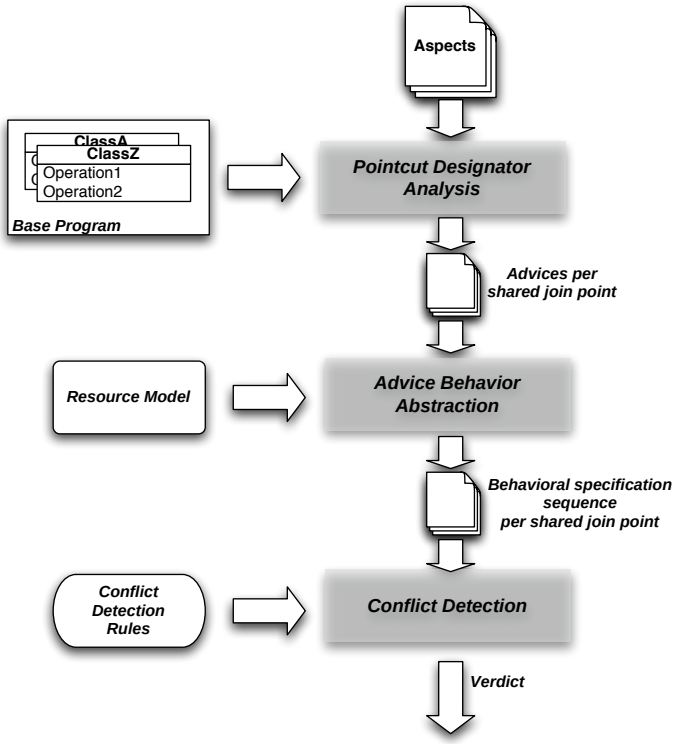


Figure 4.2: An overview of the approach.

this phase is the resource-operation specification. During the abstraction phase, the sequence of advices is transformed into a sequence of resource-operation tuples per shared join point. Next we will discuss the notion of *Resources* and *Operations* and show instantiations of these notions for the example that we explained in Section 4.2.

Resources

A resource can represent either a concrete property of the system, such as a *field* or the *arguments* of a method, or an abstract property which may, or may not have a one to one mapping to elements in the system. Such elements can be domain specific or even application specific. One such resource is the *error variable* in the example. Advice *check* verifies the arguments and alters the *error variable*, if it detects a bad input or output parameter. Advice *propagate* must ensure that if an error is set, it is logged and should be propagated to the caller. There is thus a clear dependency between these advices w.r.t. the *error variable*.

Operations

As explained previously, both *check* and *propagate*, access the *errorvariable*. Advice *Check reads* the *arguments* and the *errorvariable* and possibly writes the *errorvariable*. Advice *propagate reads* (Line 7 of Listing 4.2), *writes* (Line 9 of Listing 4.2) and *reads* (Lines 10 and 11 of Listing 4.2) the *errorvariable*, to determine whether an error has occurred. We model these as *read* and *write* operations on the *errorvariable* resource.

Although the most primitive actions on shared resources are *read* and *write* operations, if desired by the programmer, we allow such actions to be modeled at a higher level of abstraction. These more abstract operations can be derived from a specific domain, e.g., the 'P' and 'V' operations on a semaphore. Operations can even be application specific.

4.3.3 Conflict detection

The *Conflict Detection* phase expects two inputs. The first input is the sequence of operations, per resource and per shared join point. The second input is the set of conflict detection rules. This phase determines for each shared join point, and for each sequence of operations upon a (shared) resource, whether this sequence yields a conflict.

Conflict detection rules

A conflict detection rule is a pattern matching expression on the possible (combination of) operations on a resource, which is specified as a matching expression on a trace of the operations per resource. This rule can be expressed either as an *assertion pattern*; a combination of operations that *must* occur on a resource, or as a *conflict pattern*; a combination of operations that are not allowed.

In the example, the occurrence of a conflict is specified as: 'if the sequence of operations on resource *errorvariable* ends with a *write* or contains at least two succeeding *writes*'. The conflict detection rules can be expressed in any suitable matching language, such as temporal logic, (extended) regular expressions or a predicate based language. For instance, we can formulate these two conflict patterns as the following extended regular expression: $(write\$)|(write;write)$.

Conflict analysis

For each shared join point, there is a sequence of operations per shared resource. In the example, this is the *errorvariable* resource. Now assume that operation sequence *read*, *write* and *read* (which are caused by concern *Error Propagation*) are carried out on the *errorvariable* resource at a shared join point. And that subsequently a *read* and *write* operation (caused by the parameter checking concern) is carried out on *errorvariable* resource. The resulting sequence then is: *read write read read read write* This would match the conflict detection rule: $((write\$)|(write;write))$, in which case the verdict of the conflict detection process is: 'conflict'. In case of detecting an error, several actions can be carried out, such as reporting the conflict to the programmer.

4.4 Application within the Composition Filters model

The Composition Filters execution model

The Composition Filters model is an enhancement of (mostly) object-based languages, which offers improved modularity for (crosscutting) concerns. Compose* is a specific language that adopts the composition filters model. It adopts a declarative language and well defined (compositional) semantics to compose an application. It is beyond the scope of this chapter to discuss all features and elements of Compose* and the composition filters model. For a more detailed explanation we refer to [3] and [92]. It suffices to state that *filter modules* and *filters* are composed sequentially (optionally with the use of partial ordering specifications), and added (superimposed) at various places within a program. A filter encapsulates a specific behavior, and is best compared to advice.

A Compose* implementation of the example

We explain the Composition Filters model using the example of concern *ParameterChecking* presented earlier in this chapter. Listing 4.3 shows the implementation of concern *ParameterChecking* in Compose*. In Compose*, the basic abstraction is a *concern*. This is a generalization of both (object-oriented) classes and aspects. In this example, the concern *ParameterChecking* corresponds to an aspect that implements a crosscutting concern, i.e., contract enforcement of the function parameters. In the context of the composition filters model, *superimposition* denotes the—potentially crosscutting—composition of 'aspect' program units with 'base' program units [3].

```
1  concern ParameterChecking
2  {
3     filtermodule check
4     {
5        internals
6           checker : ParameterChecker;
7        conditions
8           inputwrong : checker.inputParametersAreInvalid();
9           outputwrong : checker.outputParametersAreInvalid();
10       inputfilters
11          paramcheckfilter : ParameterChecking = {
12                                                                  inputwrong || outputwrong
                                                                        => [*.compare_data]
                                                                        *.* }
13    }
14
15    superimposition
16    {
17       selectors
18          sel = {Class | isClassWithName(Class, 'CC.CX.FS')};
19       filtermodules
20          sel <- check;
21    }
22 }
```

Listing 4.3: Source of the ParameterChecking concern.

[3] Note that we use the terms aspect and base only as relative roles of program units, not fixed, since we assume a symmetrical model of AOP.

In Listing 4.3, filter module *Check* is defined (Lines 3-13). A filter module is the unit of superimposition, best compared to advice (or a group of advice and related declarations). A filter module can contain *internals* (instantiated for each join point) and *externals* (instances shared among all join points where this filter module is superimposed) variable declarations, condition declarations (these are boolean functions defined in the base code), and declarations of *input filters* and *output filters*. Filters observe and manipulate incoming, resp. outgoing messages, and are discussed in a bit more detail below). This filter module is superimposed on class *CC.CX.FS* (Lines 15-21). Filter module *Check* has an internal variable, *checker* which is an instance of utility class *ParameterChecker* (Line 6). Filter module *check* also declares two conditions, *inputwrong* and *outputwrong* (Lines 8 and 9).

Filter module *Check* has a single input filter, *paramcheckfilter*. A filter declaration consists of the filter identifier (*paramcheckfilter*), a filter type (*ParameterChecking*), and an initialization expression (between curly brackets). A filter type is best compared to a library aspect that encapsulate the actions to take when accepting respectively rejecting the initialization expression. This filter is evaluated as follows. First conditions *inputwrong* and *outputwrong* and evaluated. If any of these conditions yield a true value, the message is tried to match to the matching part (on the right side of the '=>'), in this case $[*.compare\_data]$[4]. If the current message matches, filter *paramcheckfilter* is said to *accept*, and will execute the corresponding action of the filter type. In this case this will set the appropriate error state. In all other cases the filter will reject, the corresponding reject action of the filter type does not take any action (and would result in the message continuing to the next filter, if any).

As a second aspect, we show how the error propagation concern can be constructed. This is shown in Listing 4.4. The concern *ErrorPropagation* defines one filter module *propagate*, which consists of an input filter named *errorpropagationfilter* of type *ErrorPropagation*. The filter, defined on Line 6, matches all messages, and thus will always execute the accept action of the filter. The accept action of *ErrorPropagation* ensures that all calls only continue if there is no error, and that if an error is detected, it will be logged and properly propagated.

```
1  concern ErrorPropagationConcern
2  {
3     filtermodule propagate
4     {
5        inputfilters
6           errorpropagationfilter : ErrorPropagation = { [*] }
7     }
8
9     superimposition
10    {
11       selectors
12          sel = {Class | isClass(Class)};
13       filtermodules
14          sel <- propagate;
15    }
```

[4]Here we only select one specific message for demonstration purposes. As *Parameter Checking* implements an idiom, a more general selection is used in the actual implementation.

```
16 }
```

Listing 4.4: Source of the ErrorPropagation concern.

Filter module *propagate* is to be superimposed on all classes in the system, as defined in Lines 12 and 14.

Automatically deriving a behavioral specification

In Composition Filters the behavioral specification is split into two parts. The first part is filter type specific, encapsulating the behavior of the accept and reject actions. The second part is filter instance specific, depending on the condition, matching and substitutions elements of each filter. The latter part can be automatically derived by inspecting the declarative language of the filters. We consider the evaluation of a condition to be a *read* operation on the resource representing this condition. We also inspect the matching and substitution parts to determine whether the *target* resp. the *selector* is *read* or *written*.

Inspecting the filter type to determine its behavior specification is not trivial. Filter types and the corresponding filter actions are implemented in a regular (Turing-complete) base language. These languages suffer from the general problems with analysis. However, as the filter types are defined as first class and are highly reusable we only have to express the behavioral specification while writing the filter type. It is important to realize that the behavior specification is attached to a specific filter action. Each filter which executes this action, either when accepting or rejecting, thus includes this specification. Such a behavioral specification will usually be written while developing these filter actions. This specification can subsequently be used for each instantiation of the filter type.

Detecting conflicts

We now show how the two concerns described in Listings 4.3 and 4.4 are used to detect the example conflict between these concerns. After resolving the superimposition of both concerns we can identify shared join points. In this case there is just one: the class *CC.CX.FS*. At this join point, two filter modules are superimposed: *Propagate* and *Check*. These filter modules have to be composed in some order. Any order can be chosen, but here we consider the following order exposing the conflict we are interested in: *propagate* and *check*. After composing these filter modules, we obtain the following sequence of filters:

```
1 errorpropagationfilter : ErrorPropagation = { [*] }
2 paramcheckfilter : ParameterChecking = { inputwrong || outputwrong =>
3                                          [*.compare_data] *.* }
```

Listing 4.5: Example filter order.

Filter *errorpropagationfilter* will always accept, and thus we have two possible paths: either filter *paramcheckfilter* accepts and filter action *check* is executed, or the filter rejects, in which case it will execute filter action *continue*. The two possible traces for this example are as follows:

1. $errorpropagationfilter \xrightarrow{propagate} paramcheckfilter \xrightarrow{check}$

2. $errorpropagationfilter \overset{propagate}{\longrightarrow} paramcheckfilter \overset{continue}{\longrightarrow}$

For each of these traces we generate a new set of resource usage tables. After all filter actions have been translated to operations on resources, we obtain, for each trace, a sequence of operations per resource. Table 1 shows the result of trace 1. We can see that a conflicting situation arises for resource *errorvariable*, since the last *write* is not followed by a *read* operation. This matches the first part of the regular expression: $(write\$)|(write;write)$.

action	selector	arguments	inputwrong	outputwrong	errorvariable
errorpropagationfilter: propagate					read write read read
paramcheckfilter:check	read	read	read	read	read write

Table 4.1: Filter actions mapped to operation traces for trace 1.

Using the Composition Filters approach we showed that a declarative language enables automatic reasoning about behavioral conflicts among aspects. This approach has been implemented as the *SECRET* module in the Compose* toolset.

4.5 Conclusion

This chapter presents a novel approach for detecting behavioral conflicts between aspects. Our approach defines the behavior of advice in terms of operations on an (abstract) resource model. We first analyze this behavior and represent the behavior at each (shared) join point, according to our conflict detection model. Next we verify this representation against a set of conflict and assertion rules. The resource-operation model allows us to express knowledge about the behavior of advice at both concrete and abstract levels.

We showed an actual behavioral conflict caused by crosscutting concerns that have been identified in the ASML context. We foresee a need for tooling that checks for consistency and detects conflicts between aspects, before AOSD technology can be successfully applied in an industrial context. As aspect technology is incorporated in large and complex systems (and is used to implement not only systemic crosscutting concerns) but also to implement more component specific concerns, there will be an even stronger need to have verification tools for avoiding conflicts between aspects, such as presented in this chapter. In this chapter we presented a generic model for behavioral conflict detection and a way to tailor this model to detect domain or application specific conflicts as well.

The presented approach is generic and can be applied to most, if not all, AOP languages. We briefly discussed the application of the approach to the Compose* language. In Compose* it is possible to exploit its declarative advice specifications so that

the programmer normally does not need to annotate the program.

We believe the approach presented in this chapter offers a powerful and practical means of establishing behavioral conflict detection with a minimal amount of explicit behavior specifications from the programmer. The approach has been implemented and tested within the Compose* and CAPE tool sets. The *Common Aspect Proofing Environment(CAPE)* is an initiative of the European Network of Excellence.

Chapter 5

An overview of Mirjam and WeaveC

Authors: István Nagy, Remco van Engelen, Durk van der Ploeg

Abstract In this chapter, we elaborate on the design of an industrial-strength aspect-oriented programming language and weaver for large-scale software development. First, we present an analysis on the requirements of a general purpose aspect-oriented language that can handle crosscutting concerns in ASML software. We also outline a strategy on working with aspects in large-scale software development processes. In our design, we both re-use existing aspect-oriented language abstractions and propose new ones to address the issues that we identified in our analysis. The quality of the code ensured by the realized language and weaver has a positive impact both on maintenance effort and lead-time in the first line software development process. As evidence, we present a short evaluation of the language and weaver as applied today in the software development process of ASML.

5.1 Introduction

One of the primary goals of the Ideals project is to develop methods and tools to improve the handling of crosscutting concerns, such as tracing and profiling. As a solution, a proof-of-concept aspect-oriented [38] language for the C language [64] and weaver (called WeaveC[1]) have been proposed and developed by Durr et al. [33]. Subsequently, a case study has been carried out on a representative component of the ASML software to asses the usability of WeaveC for four particular crosscutting concerns. The outcome of the case-study has shown that these four crosscutting concerns are

[1] Here, the name WeaveC refers both to the AOP language and weaver.

manifested in a significant amount of code (20-40%) besides the code representing the original concerns of the component. By using the CoCoMo model [5], the case-study estimated 5-10% effort reduction and 2-3% lead-time reduction for the code developed with WeaveC, as compared to a non aspect-oriented solution in the C programming language. Although these numbers may seem to be small, it is important to note that all software development (approximately 500 software developers) is affected by the above mentioned and other crosscutting concerns.

Briefly, the case-study demonstrated the industrial maturity of aspect-oriented programming by means of a successful proof-of-concept. As a consequent step, a transfer project was initiated by the industrial partner with the following goals:

- Develop a general purpose aspect-oriented language[2] for the C language that is capable of modularizing crosscutting concerns within ASML software.
- Implement a robust, industrial strength weaver for the proposed aspect-oriented language.
- Develop a way of working (i.e. micro-process) that describes the necessary developer roles, activities and artifacts to deal with crosscutting concerns (by means of the above described programming language and weaver).

The remainder of the chapter is organized as follows: Section 5.2 presents an analysis to identify the requirements of a general purpose AOP language. We also outline a strategy on working with aspects in large-scale software development processes. Section 5.3 discusses the concepts of the proposed aspect-oriented language, weaver and micro-process. Section 5.4 presents the results of an evaluation on the proposed language and weaver. Finally, Section 5.5 draws conclusions.

5.2 Problem analysis

In this section, we present an analysis to identify the requirements towards a general purpose aspect-oriented language to handle crosscutting concerns in the ASML context. The term ‘ASML context’ covers two important design considerations: (a) the existing solution designs (i.e., idioms, see Section 2.2.1) in Chapter 2 of crosscutting concerns and (b) the way of working of software developers within ASML. In the following subsections, we elaborate further on these considerations.

5.2.1 Boundaries of modularization

An important concern of the analysis is to explore the boundaries of modularization of the current solution design of crosscutting concerns. The aim of this step is to determine the set of necessary AOP language abstractions and variation points required

[2]The proof-of-concept AOP language had only a limited set of language abstractions that were necessary for the execution of the case-study.

for the proper modularization. Our objective is not to come up with new language abstractions; in contrast, our objective is to re-use the existing language concepts of aspect-oriented languages as much as possible. For this purpose, we iterate over the essential concepts of a reference model of AOP languages [79, 7]:

```
 1) static ZDSPTI_module_handle ti_handle ;                          //(2)
 2)   ...
 3)   int ZDAPSF_startup ( int sim_mode, boolean caching)
 4)   {
 5)     int result = 0 ;
 6)
 7)     THXAtrace("KI", (1), "ZDAPSF_startup", "> ("                 //(1)
 8)         "sim_mode = %i, caching = %b"                            //(1)
 9)         ")"                                                      //(1)
10)         , mode, caching                                         //(1)
11)       );                                                        //(1)
12)     ...
13)     ZDSPTI_timing_handle timing_hdl = NULL ;                    //(3)
14)     if ( result == OK )                                         //(2)
15)       result = ZDSPTI_register_module("ZDAPSF", &ti_handle);    //(2)
16)     ...
17)     ZDSPTI_timing_in(ZDSPTI_func_timing_hdl, ti_handle,         //(3)
                                 "ZDAPSF_startup", &timing_hdl);  //(3)
18)     ...
19)     if ( result == 0 )
20)        {... }
21)   }
```

Listing 5.1. Illustration of three particular concern instances - Tracing, Setup of Timing and Start of Timing - as they may[a] appear in a particular C function in ASML software. Respectively, these concerns are represented by the lines marked (1), (2) and (3).

[a] The concern instances presented here are slightly altered for reasons of confidentiality. However, this does not affect the essence of the examples.

Base Language Aspect-oriented languages are considered to be extensions of generic programming languages. The fact that ASML software is a legacy system written in the C programming language, restricts the base language to C. Note that the choice of base language will restrict further the design space of other aspect-oriented language abstractions (e.g,. types of join points).

Join points In AOP, (behavioral) join point corresponds to events in the control flow of a program. The example of Listing 5.1 shows that the concern instances typically appear at the beginning of the execution of a function. In other words, the example concern instances need to be executed before (and also after) the occurrence of the

event ‘execution of a function’. Considering the current idioms that realize the crosscutting concerns, we identified two necessary types of join point: *call* and *execution* of a function.

Context of Join points Listing 5.1 shows that various types of context information are required for the modularization of a crosscutting concern. For instance, the concern instance of Tracing refers to the *name* of the function in Line 7, and *formal parameters* of the function in Line 10. The concern instance of Setup of Timing refers to a *local variable* of the function (‘result’) in Line 14. The same concern instance refers to the *name of the module* of the function (‘ZDASP’) in Line 15. Besides these types of context information, we identified that the following types of context information are required: *global parameters* of a function, *types of parameters and local variables, return type of functions* and the fact whether a parameter or local variable behaves as an *input and/or output parameter* in the function (derived information from data-flow analysis).

Context and join points related to pointcuts AOP languages generally make use of the above described context information, not only in the parametrization of crosscutting behavior but also in the designation process of join points (i.e. in pointcut or query languages of AOP languages). Obviously, we will also use these in the designation process of join points.

Context Originated from Build Process In our investigation, we recognized that information derived from the build process is also used in the idioms that realize crosscutting concerns in software. For instance, the concern instance Tracing refers to the component code ("KI") in Line 7; this information is determined from the target of the build process. In a broader view, when crosscutting concerns are woven into different products of a product line, product (and platform) specific information also needs to be addressed in the modularization. Hence, information about product and platform can serve as variation points; these are typically originated from the configuration of the build processes.

Advices and Variation Points in Advices Advices are the units of AOP languages to formulate the crosscutting behavior in terms of the instructions of generic programming languages. All the earlier described types of context information — i.e., the join point, the properties and relationships of a join point, and the platform and product specific information from the build process — can serve as variation points in the formulation of an advice. Normally, these variation points can be *directly* used/referred to through pseudo variables (e.g. `thisJoinPoint` in [65]) and/or through parameters that are extracted from pointcuts and passed to advice bodies (e.g. context exposure in AspectJ).
Besides the direct references and usage, we recognized *indirect* usages of the above listed types of context information. For instance, the string literal ‘`mode = %i,`

`caching = %b`' in Line 8 contains the name of formal parameters and format specifier characters based on the type of the formal parameters. That is, the formal parameters (as join point context) are not explicitly referred to as variations points (unlike in Line 10) but their properties need to be used to *compute* a variation point (cf. the string literal in Line 8) in the notation of the advice-concept.

The fact that we need to deal with a legacy system may have further constraints on the design space of the advice-concept: the crosscutting behavior should be formulated in terms of instructions that can express calls to existing software libraries. For instance, the calls `THXAtrace()` in Lines 7-10 and `ZDSPTI_timing_in ()` in Line 17 are such 'legacy' calls to libraries that realize tracing and timing.

A consequence of this constraint is that the concept of variable-length argument list - denoted by '...' in C - needs to be treated as variation points in the notation of an advice. For instance, in Listing 5.1, the concern instance of tracing is represented by the call `THXAtrace("KI" , (1) , "ZDAPSF_startup" , "> (" "sim_mode = %i, caching = %b" ")", sim_mode, caching)` with 7 arguments, in Lines 7-10. The arguments of this call statement depends the formal parameters `mode` and `caching` of the function `int ZDAPSF_startup ( int sim_mode, boolean caching)` This means that the concern instance of tracing in a different function context will be represented by a call statement with different format string and different number of arguments. For instance, the function `ZDAPSF_shutdown()` without any parameter will have the tracing call `THXAtrace("KI" , (1) , "ZDAPSF_startup" , "> (" ")")` with only 5 arguments. To address this problem, the notation of the advice concept needs to able to deal with the concept variable-length argument list as a variation point. This variable-length argument list can always be derived from the actual join point and its context (cf. the base function to be woven and it formal parameters).

Note that re-implementing the tracing library with a suitable interface is not an option either, as it would introduce other maintenance problems related to legacy systems.

Aspectual States/Aspect Instances In Listing 5.1, both the concern instances of Setup of Timing and Start of Timing use the variable `ti_handle` that represents the module handler of Timing. The concern instance of Timing uses another variable, called `timing_hdl` in Line 17. The difference between these two variables is apparent from the example already: `ti_handle` is declared as a global static variable in Line 1. This means that `ti_handle` can (and will) be used in every function of the module; thus, it can be *shared* among different concern instances that appear in different functions. In contrast, the variable `timing_hdl` is declared as a local variable of the function in Line 13; hence it is *local* only to those concern instances that appear only within the same functions. In terms of aspect-oriented programming, we identified the need of two types of aspectual states: one which is shared only among those advices that are woven at the same join point (*per join point*) and one which is globally shared among every advice (i.e., *per aspect*).

Note that there are many other fundamental concepts, e.g. ordering of advices, context exposure, et cetera, in the design space of aspect-oriented languages. Naturally,

we consider these concepts as parts of our design space; however, the discussion on the motivation for each particular language concept is beyond the scope of this chapter.

5.2.2 Quality aspects of concrete syntax for large-scale development

In the previous section, we discussed the boundaries of modularization that determines the necessary expressiveness of the language - i.e., abstract-syntax of a language, in terminology of Domain Specific Languages [42]. However, besides necessary expressiveness, there are various quality aspects on the concrete syntax that are significant from the point of view of large-scale software development. In this section, we discuss these.

Predictability (in design phase) Predictability ensures developers that certain properties are held during the development of software. This is crucial in large-scale development to prevent mistakes and errors already in the design phase as soon as possible. Besides, when introducing new technology into a large organization, ensuring predictability in the design phase is well-motivated for the following two reasons: (1) it can lessen the risk of improper use (and its undesired side effect) of the new technology; (2) it can reduce the fear of using a new technology among the developers (e.g., by providing well-controlled means for the new technology).

To this aim, we are going to use well-known language mechanisms, such as type-checking, enforcement of declarative completeness, and an extensive set of syntactic and semantic rules in the design of the aspect-oriented language[3].

Evolvability Evolvability is an important software quality factor that indicates the ability of developing programs incrementally. In general, evolvability facilitates extending an application towards new requirements mostly by reusing previously written modules with minimal or no modification. By ensuring evolvable specification, we expect to reduce the complexity of the code caused by the frequent appearance of the phenomenon called ‘deviations from standard functionality’ in a large-scale development.

To this aim, we are going to provide language abstractions that (1) can be re-used in different specifications and (2) can support the re-use of existing specifications.

Extendibility In principle, language constructs that provide means of parametrization can positively contribute to the extendibility of a language. Providing ease of extendibility for a language is beneficial in large-scale development for two reasons. First, due to time-to-market pressure the language (and also the weaver) needs to be incrementally delivered. Secondly, rolling out a new version of a language and weaver is not a trivial task as it affects many ongoing software development activities; hence, a

[3]These language mechanisms obviously require adequate compiler and/or run-time support.

large number of development modules maintain dependencies to aspect specifications. This means that change requests towards the language should result in changes in the notation as *minimal* and *isolated* as possible.

Comprehensibility We define comprehensibility as the ability to understand the meaning of a program by just looking at its source code. Comprehensibility can be influenced by the programming style and the notation of the abstractions of the adopted language, such as how the program units are modularized, where the references in the units are specified, and the style of notation that reflect the underlying computational paradigm. Although this quality aspect had less significance compared to the previous ones in our list, we needed to take into account the fact that developers at ASML have a stronger background on procedural and object-oriented languages (with strong typing) compared to logic or functional languages.

5.2.3 Expected way of working

In the previous two sections, we discussed the requirements towards an aspect specification language used in large-scale software development. However, when introducing aspect-oriented programming (or any other new technology) into a large organization, a clear and sound strategy on the expected way of working is also necessary, besides the required means and artifacts. In this section, we discuss our strategy on how aspects are intended to be used in the software development process of ASML.

To make sure that developers can benefit from the advantages[4] of aspect-oriented programming, the objective is that *every* developer should be able to *use* aspects easily with a minimal learning curve. On the other hand, to minimize the possible danger of the improper use[5] of AOP, only *a few* of the developers are allowed to *write* and *release* news aspects. To this aim, our objective is to provide generic aspects with highly and easily customizable interfaces:

- Developers are expected to use aspects in a standard way (seamlessly, by enabling them in the build process).
- Most users will use aspects with their standard functionality.
- Some users will want to deviate from the standard functionality. In other words, they want to customize the functionality of generic aspects according to their special needs. For instance, some users will want to ‘switch off’ the tracing of certain time-critical functions.
- This customization should be minimal and rely on design information added to the source code, as most users are not allowed to modify aspects or write their own one.

[4] E.g., locality of changes, consistency and clarity of code, et cetera.
[5] E.g., undesired side-effects in the control-flow, ‘patching by aspects’, et cetera.

5.3 Mirjam, an aspect-oriented language extension for C

In this section, we outline the important constructs and characteristic properties of the realized aspect-oriented language, called Mirjam. Due to a lack of space, we cannot iterate over the full design space of Mirjam. The interested readers can find a more detailed introduction and description of the language in [9].

5.3.1 Aspect

The main unit of modularization is the aspect specification file. An aspect specification may contain two language constructs (see Listing 5.2): *context declaration closure* and *aspect declaration*.

```
 1)   context{
 2)       #include "THXAtrace.h"
 3)       #define FALSE 0
 4)
 5)       typedef int boolean;
 6)   }
 7)
 8)   aspect SimpleAspect
 9)   {
10)       advice someAdvice() before (FunctionJP JP)
11)       {
12)         boolean tracing_flag = FALSE;
13)         THXAtrace(JP.module.name, JP.name, tracing_flag, "> ");
14)      }
15)     ...
16)   }
```

Listing 5.2. Illustration of the context declaration closure and aspect declaration in an aspect specification file.

The context declaration closure is a placeholder for a standard C declaration. We will use the declarations in C in the context closure to ensure *declarative completeness* in the notation of advices. The context closure in the listing above has two preprocessor directives in Lines 2 and 3, and a typedef declaration in Line 5. To resolve the preprocessor directives, the aspect specification is first preprocessed by a standard C preprocessor. The declarations in the context closure are used later in the specification of the crosscutting behavior (i.e., advice in term of Mirjam) in Lines 11 and 12. Note that the function call 'THXAtrace' (in Line 13) is declared in the included THXAtrace.h file.

The aspect declaration part is a container type of program element: it works as a name space. The contained language abstractions (query, advice- and binding-declarations) can be referred to through this name space.

5.3.2 Queries as pointcuts

The language concept of pointcut is realized by the language abstraction *query* in Mirjam. A query in Mirjam returns a set of join points. A join point is a *tuple* that contains an arbitrary number of (but at least one) tuple elements. A tuple element can be of two types: *join point location* and *join point context*. As the names suggest, the type of join point location describes the place where we want to weave in crosscutting behavior (e.g., execution of a function), while the type of join point context describes the context of the weaving. We can use the context information for (1) either refining of the weaving location or (2) customizing the crosscutting behavior to be woven to certain weave contexts.

```
1)   int a;
2)   int f(int b, int c) {
3)      ...;
4)   }
5)   int g(double d) {
6)
7)   }
```

Listing 5.3. An example of base code that we will use in the rest of the discussion to illustrate how the query language of Mirjam works.

We will use a number of examples of queries to present the characteristic features of the query language of Mirjam:

```
1)   query Q1() provides (FunctionJP JP)
2)   {
3)      JP: true;
4)   }
5)
6)   query Q2() provides (FunctionJP JP)
7)   {
8)         JP: JP.name =~ "f.*";
9)   }
10)
11)  query Q3() provides (FunctionJP JP, Variable@JP V)
12}  {
13)     JP: true;
14)     V : true;
15)  }
16)
17)  query Q4() provides (FunctionJP JP, Variable@JP int V)
18)  {
19)     JP: tuple(JP) in Q1() && |JP.formalParameters()| >= 1;
20)     V : V in JP.formalParameters();
21)  }
```

```
22)
23)   query Q5() provides (FunctionJP JP, Variable@JP[] V)
24)   {
25)      JP: tuple(JP) in Q1() && |JP.formalParameters()| >= 1;
26)      V : V == JP.formalParameters() ;
27)   }
```

Listing 5.4. Examples of query-declarations in Mirjam.

As an example of a query, consider the query declared between Lines 1 and 4 in Listing 5.4. The query declaration starts with the keyword `query` followed by an identifier ('Q1') and a list of possible formal parameters. After the formal parameters we define a set of tuple variables preceded by the keyword `provides`. The list of tuple variables describes the elements and type of tuples that the given query provides. In Lines 1-4, the query Q1 provides tuples with one tuple element, the type of this tuple element is FunctionJP. The type FunctionJP represents the type of the join point location of the execution of a function. Inside the query, a tuple condition needs to be defined for each tuple variable. In Line 3, the tuple condition 'true' means that every particular function execution will satisfy this query; i.e., there is no further restriction on the tuple variable JP. The resulting set of tuples of query Q1 executed on the base code of Listing 5.3 is {(f),(g)}[6]. In Lines 6-9, the query Q2 is similar to Q1 except that it has a more restrictive tuple condition in Line 8: the name of the executing function shall start with the prefix 'f' (defined by the regular expression `'f.*'`). Similar to the notation of object-oriented programs, the dot operator can be used to access instance variables and execute methods of certain types in Mirjam. The result set of Q2 on the given base code is (f), as there is only one function that can satisfy this condition.

In Lines 11-15, the query Q3 will provide set of tuples with two tuple elements. The first tuple variable is of type 'function execution'. The second, new tuple variable is of type Variable, which is a type of join point context. More precisely, the type Variable can represent global variables, formal parameters and local variables of an executing function. In Line 11, the declaration Variable@JP means that we expect the variables in the scope of the function execution of JP; that is, there is direct link between a particular function and variable in the result set of the query. Note that a second tuple condition 'true' has been declared for the second tuple variable in Line 14. When a query is evaluated, the query engine iterates over the possible values of both tuple variable types. During the iteration, if there is a combination of particular values of tuple variables that can satisfy both tuple conditions (based on the available base code), a tuple is created and added to the result set of query. In short, the query Q3 returns an ordered set of tuples (i.e. the Descartes-product) of all functions and variables related with those functions: {(f,a), (f,b), (f,c), (g,a),(g,d)} based on the code of Listing 5.3.

In Lines 17-21, the query Q4 illustrates a bit more complex use of queries. The tuple variable V in Line 17 is declared with a type restriction: the C type of the variable

[6]For the sake of simplicity, the letters f and g represent the execution of functions f and g.

is restricted to int. In Line 19, we show a possible re-use of a previously declared query: the tuple variable JP is converted to a tuple by the conversion operator 'tuple'. The membership relation 'in' specifies that JP (as converted to a tuple of one element) should be in the result set of the query Q1. The membership relation ('in') is not the only way to re-use queries; the equivalence relations can also be used with query calls. Queries can be called with formal parameters to perform selection or projection (in terms of tuple relational calculus [24])[7] on previously defined queries. In Line 19, the second part of the tuple condition specifies that the executing function needs to have at least one formal parameter. The second tuple condition in Line 20 specifies that V is a formal parameter of JP. The execution of Q4 on the provided base code of Listing 5.3 results in the following set of tuples: {(f,b), (f,c)}.

In Lines 23-27, the query Q5 is slightly modified version of the query Q4. Q5 differs from Q4 in two places. First, the type of the tuple variable V is an array type denoted by the symbols '[ ]' in Line 23. Note that this tuple variable has no C type specification either. Secondly, the tuple condition uses the equivalence relationship ('==') instead of the membership relationship ('in'). This means that Q5 will provide tuples with two elements: a tuple element of a function-execution and a tuple element of an array of variables. The equivalence relationship indicates that the array of variables must be the array of formal parameters of the corresponding function. This means the result set of Q5 is { (f, {b,c}), (g,{d}) } based on the code of Listing 5.3.

5.3.3 Advices

Similar to other AOP languages, the program unit that specifies the crosscutting behavior is called *advice* in Mirjam. The crosscutting behavior is defined in terms of instructions of the C programming language.

```
1)   advice printIntParam(Variable@JP int V) before (FunctionJP JP)
2)   {
3)    printf("In module %s, function %s executes with argument %s=%d",
4)               JP.module.name, JP.name, V.name, V);
5)   }
```

Listing 5.5. An example of advice-declaration.

As an example of an advice, consider the advice declared in Listing 5.5. The declaration starts with the keyword 'advice' and is followed by an identifier and a (possibly empty) list of formal parameters. A formal parameter has to have a type specifier in Mirjam and may have a type specifier in C. The formal parameters are followed by the type of the advice that specifies whether the advice is to be executed before or after the execution of a join point. Finally, the last element in the signature of the advice is the actual join point variable in the form of a formal parameter (FunctionJP JP). In the

[7] Defining the formal semantics of queries is beyond the scope of this chapter.

body of the advice, we refer to the properties of the join point through this variable in Line 4. In the same line, we also use the parameter V as a formal parameter similar to the formal parameters used in C.

Note that the correct use of the format specifier (e.g., %d in Line 3) cannot be checked just like in normal C code. For instance, we could use %f that would have resulted in an incorrect execution. This piece of advice exemplifies another possible problem as well: if we coupled it with the query Q4 (of Listing 5.4) the advice would be woven two times on the function `f`, as Q4 returns {(f,b), (f,c)}. This means that the execution of the program would have two output messages, see Listing 5.6.

```
6)  In module ZZ, function f executes with argument b=5
7)  In module ZZ, function f executes with argument c=11
```

Listing 5.6. Illustration of output messages from advice executions.

This execution is correct because Q4 exposes only formal parameters of type int. However, if the query is not restrictive enough, formal parameters with other types than int (e.g. double, pointer-type, et cetera) can be queried too. This would result in incorrect output messages, since the advice expects a parameter of type int[8]. In addition, having the same message two times, with only a small difference at the end, is not considered a neat solution: a single printf statement could handle a variable number of arguments. So the question is how to specify the statement printf with variable number of arguments of different types that depend on the weaving context? Note that we discussed the same problem in Section 5.2.1 at 'Variation Points in Advices'.

5.3.4 Advice generators

To handle the category of problems described above, we introduced the language construct *advice generator*. These are special built-in constructions that can be used in the body of an advice and are recognizable by the form `@type<...>@`. The first element of a generator is usually an iterator of the form `[iterator: expression]`. The iterator contains the name of an array variable and the (local) name for each element, and returns a list of expressions. The `expression` will be evaluated for each element. Finally, the `type` says how the expressions are combined and what the outcome of the advice generator is[9].

[8] To prevent this sort of errors, we do type checking between the type of the actual tuple values and the type of the advice-parameter.

[9] In other (typically, functional) languages one can achieve the same functionality by using the list manipulator functions *map* and *join* on lists.

```
 1)   advice printParams(Variable@JP[] Args) before (FunctionJP JP)
 2)   {
 3)     printf("Function %s executes with argument(s) "
 4)       @StringConstant<
 5)       [arg in Args:
 6)             strConcat(arg.name, "=",formatString(arg))],
 7)        "; " //semicolon as separator
 8)       >@
 9)       ,JP.name
10)       ,@VarargExpression<[arg in Args: formatExpression(arg)]>@
11)     );
12)   }
```

Listing 5.7. An example application of two advice generators.

For an example application of advice generators, consider Listing 5.7. In Lines 4-8, the advice generator `@StringConstant` will generate a format string for an array of variables passed to the advice (Args)[10]. In Line 6, the built-in function `strConcat` creates a piece of string from the name of a variable (cf. formal parameter of a function), from the equation sign ('=') and from the result of the function `formatString` returning the format specifier of the C type of the variable. In Line 5, this concatenation function is applied on each variable in the array of the variables ('arg in Args'). This will result in a list of a piece of format string for each formal parameter. The resulting list of format strings per parameter will be combined into a final string literal by the StringConstant generator, using the semicolon as a separator. Similarly, a variable-length argument list will be created by the advice generator `@VarargExpression` in Line 10. If we couple this advice (Listing 5.7) with the query Q5 (in Listing 5.4) the advice is woven once on the functions f and g, as Q5 returns (f, {b,c}), (g,{d}). This means that the execution of the program would have two output messages, see Listing 5.8.

```
1)   Function f executes with argument(s) b=5; c=11
2)   Function g executes with argument(s) d=3.1415
```

Listing 5.8. Illustration of output messages from advice executions with advice generators.

5.3.5 Bindings

The third language construct that can be declared within an aspect is the *binding closure*. The binding closure contains typically one or more *binding definitions* introduced

[10]This list of parameters is derived from the context of join point: they are e.g., representing formal parameters of a function. In the following section, we will show both the query that realizes this derivation and the binding-definition that couples the advice printParams with that query.

by the keyword `foreach`. One binding definition can couple a query with one or more *binding entities*. Different types of binding entities exist in Mirjam: for example, the binding entity `apply` binds advices to queries, and passes the result of the queries to the advices using a parameter-passing mechanism. When more than one advices are to be applied to the same join point, one can specify the order of their application by the binding entity `order` per join point. The binding entity `error` and `warn` can raise errors, when certain situations occur in the base code. Finally, the binding closure and binding entities may also contain variable declarations, called binding variables, to share states between advices. In the following subsections, we will show the use of each of these language constructs.

```
 1)   aspect VerySimpleTracing
 2)   {
 3)     query Q5() provides (FunctionJP JP, Variable@JP[] V)
 4)     {...}
 5)
 6)     advice printParams(Variable@JP[] Args)before(FunctionJP JP)
 7)     {...}
 8)
 9)     binding
10)     {
11)       foreach (thisFunc, vars) in Q5
12)       {
13)           apply on thisFunc {
14)             printParams(vars);
15)           }
16)       }
17)     }
18)   }
```

Listing 5.9. An example of binding and an illustration of passing context information to advices.

For an example of a binding-declaration, consider Listing 5.9. In Line 3, the query Q5 provides a set of tuples of a join point variable (FunctionJP JP) and a context variable (Variable@JP V). In Line 11, the binding definition iterates over the result set of Q5: `thisFunc` and `vars` will represent the two variables of each tuple. For each iteration step, these variables hold the values of the actual tuple elements. In Lines 13 and 14, the actual values of `thisFunc` and `vars` are passed as parameters to the advice in the advice-call. Finally, in Line 6, the variables are ‘received’ and they are used in the body of the advice. This type of parameter passing works in a similar way to the macro substitution mechanism of the C preprocessor.

5.3.6 Bindings variables

As we mentioned in the previous section, the binding closure and binding entities may also contain variable declarations, called *binding variables*, to share states between the executions of advices.

```
 1)   binding {
 2)      static int ti_handle;
 3)
 4)      foreach (startup) in startupFunction()
 5)      {
 6)         apply on startup {
 7)            moduleRegistration(ti_handle);
 8)         }
 9)      }
10)
11)      foreach (f) in timedExecution()
12)      {
13)        apply on f {
14)           int timer;
15)
16)           functionTimingStart(ti_handle, timer);
17)           functionTimingStop(timer);
18)        }
19)      }
20)   }
```

Listing 5.10. Illustration of the use of binding variables.

Consider the variables `static int ti_handle` and `int timer` in the Lines 2 and 14 of Listing 5.10. One can use binding variables *per aspect* (like `ti_handle` in Line 2) and *per join point* (like `timer` in Line 13). 'Per aspect' binding variables, e.g., `ti_handle`, are shared between all advices within a binding. This means that if one advice changes `ti_handle`, the change can be observed in all other advices of the aspect. 'Per join point' binding variables, e.g. `timer`, are shared only among those advices that are called from the binding entity where the variable is declared. For example, `timer` is shared only between the execution of the advices `functionTimingStart` and `functionTimingStop` in each function where they are applied together.

5.3.7 Annotations

Annotations in WeaveC are used to attach semantic meaning to syntactical constructs in a program written in the programming language C. The annotation mechanism is designed such that the following objectives are met:

Easy of use An application of an annotation to a program should have minimal overhead, both in terms of the effort spent by a developer and of the impact on the program listing. The syntax should be unobtrusive (not disturb the natural flow of the program listing) and similar to the syntax of similar C concepts (type modifiers like storage class, const et cetera).

Flexible for different usage The application of annotations is only limited by the scope of tools that can process them, so the mechanism should place few restrictions and allow for varied use.

Allow checking of the correct application of annotations This means that it must be detectable by a tool if an annotation is applied at the wrong location (i.e., to a function while it makes no sense for a function), with the wrong arguments (i.e., with a value for a specific field while the annotation has no such field) or multiple times (when it makes no sense to do so for the annotation). It does not need to be possible to check for unexpected absence of annotations (i.e., a function has no annotation while one is always expected) or illegal combinations of annotations or values of fields of annotations (i.e. a function can not have both annotations 'a' and 'not_a'). These checks, if desired, must be performed by the tools that make use of the annotations.

Easy to hide from a program This should be the case for instance if the program is to be read by tools that are unaware of this annotation mechanism. Such hiding should be possible by a standard-compliant C pre-processor or a standard text processing facility.

Robust Common typing errors in the declaration or application of an annotation should not lead to undesirable parsing of the actual program code.

5.3.8 An example of using annotations

In this section, we outline of how the annotation mechanism of WeaveC can be used.

```
 1)   // --- an annotation-declaration in .c files ---
 2)   /*$
 3)      annotation{
 4)        boolean value;
 5)      } trace[module,function,parameter,variable,type] = {
 6)        value = TRUE
 7)      };
 8)   $*/
 9)
10)   // --- an annotation-application in .c files ---
11)   int critical_function()
12)   /*$trace(FALSE)$*/
13)   {...}
```

```
14)
15)   // --- a query referring to an annotation-application  ---
16)   query allTraceableFuncs() provides (FunctionJP JP)
17)   {
18)        JP:  (JP.name != "main") &&
19)             (!( (JP.$trace?)&&(JP.$trace.value == false) ))
20)   }
```

Listing 5.11. Illustration of the declaration, application and use of annotations.

The declaration and application of annotations are denoted by the tokens `/*$` and `$*/`, in the style of Splint [40]. In Listing 5.11, the annotation `trace` is declared between Lines 2-8. The annotation declaration has one boolean field called `value` (in Line 4) and it can be applied on modules, functions, formal parameters, variable declarations and type declarations, as indicated in Line 5. In Line 6, we specify that the default value for the field `value` is TRUE[11]. An application of the annotation `trace` is illustrated in Line 12: the application is part of the signature of the function declaration `int critical_function()`. Finally, the query `allFunctions()` shows a use of annotations for determining join points in Mirjam, in Lines 18-19. Only those function executions will be designated that do not have the name "main" and do not have the application of the annotation 'trace' with the value FALSE.

5.4 Evaluation

Dürr et al. [34] carried out an experiment on quantifying the benefits of using aspect-oriented programming by means of the above defined language and weaver. The setup of the complete experiment and all statistical data about the results can be found in [34]. In this section, we provide only a summary about the setup and the results of the experiment.

The goal of the experiment was to determine whether using AOP speeds up the development and maintenance of the ASML codebase. The experiment consisted of five change scenarios in two sets related to use of the concern Tracing. Twenty software developers participated in the experiment[12]; the participants were split into two groups. The most important reason to do the splitting was to verify that the two sets of change scenarios were equivalent.

Although it is hard to extract significant results from the experiment due to small number of participants, the following conclusions can be definitely supported from the results of the experiment:

- Adding tracing to a function takes considerably less time with AOP than without.

[11] This is a standard predefined macro. The weaver has a built-in C preprocessor to resolve macro substitutions within annotation declarations and applications.

[12] The experiment was actually combined with training on the use of aspects and WeaveC.

- Removing tracing from a function takes more time with AOP than without. This is probably caused by the (first) usage of annotations.
- Selectively tracing parameters in a function, takes less time with AOP than without.
- Adding tracing to a function manually introduces significantly more errors than with AOP.
- Changing the signature of a function manually introduces significantly more errors than with AOP.

Besides this experiment, we are continuously working on the evaluation of the language and weaver by different means. At the moment of writing, software developers and engineers (approximately 70-80 participants) - who are currently using the language and weaver - are participating in a survey to evaluate the tool in the view of usability and other quality concerns. We consider the feedback both from the experiment and survey crucial in driving our design on the upcoming features of both the language and the weaver.

5.5 Conclusions

Currently, WeaveC is part of the standard build process of the ASML software. WeaveC is used in 42 components today; in these components 298 targets are generated based on 1007 woven source files. Software developers do not need to write tracing and timing code anymore. The aspect files that realize these crosscutting concerns are part of the external interface of a standard software component. The weaving of these aspects is enabled by make-files [95] in the build process. Besides, developers can add annotations to their base code to customize the standard tracing functionality when needed.

Mirjam and WeaveC can fulfill the promises of the proof-of-concept weaver. The quality of the code ensured by Mirjam and WeaveC has positive impacts both on the maintenance effort and lead-time in the first line software development process. The increased quality of the code also improves lead-time and reduces errors in terms of the analysis of problems/machine performance, especially during integration and field problems. As a result, the forth line software development can also be done faster and better, which therefore reduces integration time and improves the response to field problems.

Chapter 6

Modeling the coordination idiom

Authors: Teade Punter, Marc Hamilton, Tanja Gurzhiy, Louis van Gool

Abstract This chapter reports about a case study on modeling and model transformation of a specific ASML idiom, called coordination. The study showed that it is possible to define generic transformation rules that can be applied for simple and complex instances of this particular idiom. We think that the approach presented in this chapter is applicable to other idioms as well. However, this requires clear criteria to identify the idioms. Furthermore, the approach requires specific personal skills to enable the modeling and transformation approach.

6.1 Introduction

This chapter reports about a case study on modeling and model transformation of a specific ASML idiom, called coordination.

In general, modeling and transformation are advocated to achieve a high level of abstraction in complex system design. This provides an organization, its system architects and engineers a better overview on their system, meanwhile focusing on the most relevant system elements. This argument was also the driver for our case study. Defining the idiom on a higher level of abstraction would enable ASML to maintain its design of coordination concerns in a better way.

Abstraction is a way to hide the implementation details of particular functionality. A well-known example of abstraction levels - that deals with communication in computer and network protocols - is the ISO OSI model that consists of seven layers [104].

In systems and software engineering such layering is common, though only few methods make this explicit [103]. We are aware of OMG's MDA initiative that distinguishes platform-independent models (PIMs) and platform-specific models (PSMs), which is quoted in many papers of academic researchers. However, we found that this distinction is not implemented yet in industrial practice. We use in this paper a terminology that distinguishes abstract and concrete level, without providing an exact definition. Figure 6.1 provides an intuitive overview of abstraction levels we found at ASML. The figure points out that we take a relatively stable abstract design model which focuses on the required functional semantics using the coordination idiom, which we transform to a concrete design level, where platform issues are addressed. The idiom thus reflects the implementation-dependent concepts.

The main purpose of models in system development in the high-tech industry, is to help engineers understand the interesting aspects of a future system. Models are therefore widespread used by engineers in a variety of disciplines. For example, hardware engineering applies models in notations/languages like VHDL. The expected benefits of modeling and transformations are: a) the reduction of development effort (generate a large part of the implementation) and b) the ability to manage system complexity (work at higher abstraction levels).

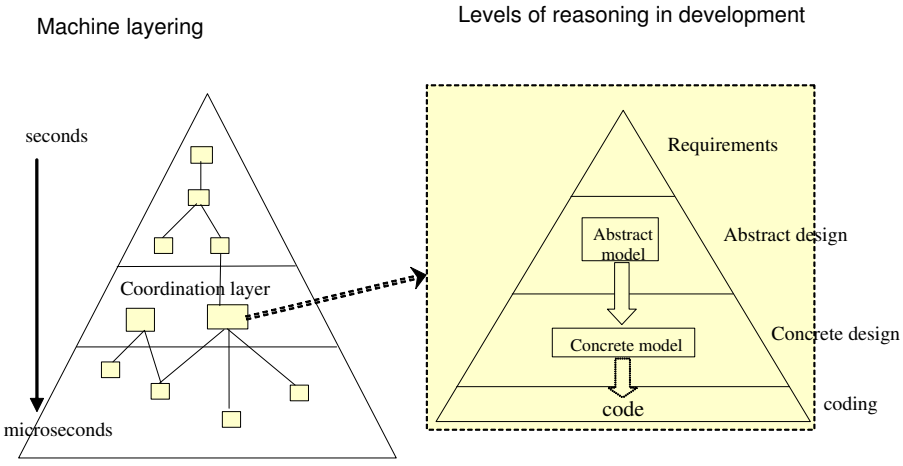


Figure 6.1: The abstraction levels in the ASML design flow (right) for a subsystem in the machine.

If we position the work of this chapter in the model space as shown in Figure 1.5 of Section 1.5 we act within a subspace concerning so-called *logical action* components on models of different abstraction levels. The goal of our work is to examine the feasibility to model the coordination concern on an abstract level and to transform this

abstract model into a more concrete one, see also Figure 6.2. This chapter is further organized as follows. In Section 6.2 we present the coordination idiom that we applied in our modeling and transformation case study. Section 6.3 introduces our approach and its considerations to model and transform the idiom. Section 6.4 provides the case study findings, and Section 6.5 summarizes these findings into conclusions.

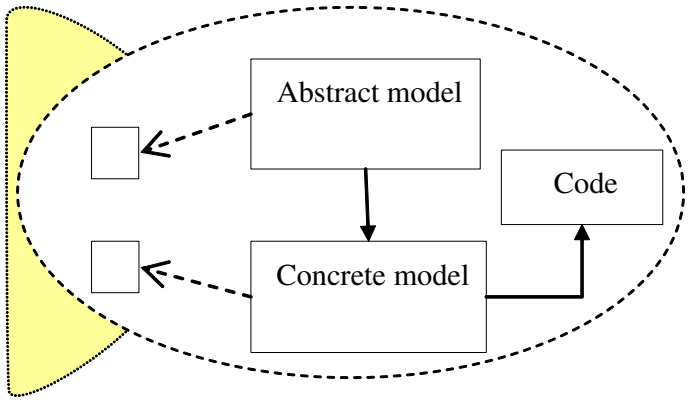


Figure 6.2: Relating this work to the general model space in Chapter 1.

6.2 Coordination idiom

The coordination idiom deals with the definition of services which are algorithmic combinations of machine actions that a wafer scanner can perform. Such services are called *logical actions*. A subsystem in the coordination layer defines services in terms of machine actions, called *physical actions* and the *subsystems* that can perform these actions.

The invocation of a logical action results in a *logical behavior*, which consists of algorithmically combined invocations of *physical actions* which result in *physical behaviors* of parallel executing *subsystems*. A group of physical behaviors that is executed simultaneously in parallel as part of a logical behavior is called a *behavior set*. Behavior sets are independent and may execute concurrently. However, a series of behavior sets for a single logical behavior is guaranteed to execute without interruption of physical behaviors of other logical behaviors. This enables engineers to think about the behavior of logical actions in isolation.

A logical behavior is not as straightforward as a simple sequence of physical actions, but can contain loops and decisions that are based on intermediate results. An important aspect of coordination is that logical behaviors are not executed in isolation, but may execute concurrently if they do not need a certain subsystem at the same mo-

ment. Sometimes one wants to prevent that a subsystem is affected by another logical behavior, although one does not need the subsystem to perform any physical behavior but only remain in its current state. This can be guaranteed by performing so-called passive behavior on that resource in some behavior set. This ensures that during the execution of the behavior set, the passive subsystem will not execute any other physical behavior. Together with the definition of the coordination idiom that assumes that logical behaviors do not interfere, a tetris-like execution of physical behaviors results [45].

The coordination platform implementation manages the asynchronous invocation and the parametrization of the subsystem functions that correspond to the physical actions and keeps track of the data flow to allow for combining the results into a logical action result. In this platform, a logical behavior has to be programmed as a flat sequence of (invocations of) subsystem functions. During execution, these functions are queued at the corresponding subsystems until a result is needed from some previously queued function or until all functions are queued. Furthermore, the platform allows during execution to skip queuing of functions until a certain one in the sequence. The latter enables the realization of conditional execution.

In the abstract definition of logical behavior, physical actions can be combined algorithmically and parallelism and loops can be explicitly defined as such. To allow for a generic approach in mapping this to the coordination platform implementation, we define in the concrete level two kinds of actions: functional actions and control actions. A functional action corresponds to a call of a function corresponding to a physical action. It determines which function is executed and (as part of the behavior set) which subsystems have to be passive during its execution. Control actions are used to implement the flow of queuing of functions. These control actions are used to assign values to variables, evaluate guards, mark some point in the sequence and skip the sequence until such a mark.

Currently, a subsystem in the coordination layer is implemented in C-code using a generic coordination platform code framework, while its higher level specification is written down in Word documents without using specific modeling techniques. The interfaces between a coordination subsystem and the general coordination framework result in logical action related code scattered over multiple C modules. Although at a high level of abstraction, the required functionality can be defined in a clear and easy to understand way, the realization of the coordination subsystems is labor-intensive, time-consuming and error-prone. For this reason, we want to consider a model-based generative approach, starting from models at a high level of abstraction and ultimately reaching the code level.

Overall, such a generative approach results in a ‘model-to-text’ transformation. Looking in more detail, such a transformation consists of two major parts:

1. The transformation of abstract level concepts (physical action, passive action, conditional execution, loops, behavior sets) to concrete level concepts (functional actions, control actions).

2. The transformation of concrete level concepts into corresponding code represen-

tation (C-language: functions, defines et cetera).

In general, one separates these concerns when developing generators in an MDE context. In our opinion, transformations of concepts can best be realized as model-to-model transformations, and the generation of a particular representation as a much more straightforward model-to-text transformation. Our work has primarily focused on the high level steps in the generative approach, which we expected to be the most challenging parts of the generation process. It includes:

- The representation of abstract level concepts in a model.
- The representation of concrete level concepts in a model.
- The transformation of abstract level concepts into concrete level concepts.

6.3 Case study

The goal of our case study was to investigate the feasibility of model transformation by considering the coordination concern. The work in our case study can be distinguished in two separate, but related, activities, namely: modeling and transformation.

6.3.1 Modeling

The modeling activity aimed at defining models at abstract and concrete levels. At both levels the models related to structure as well as to behavioral issues. Structure was modeled by using UML's class diagrams. The dynamic behavior of the coordination idiom was shown by using activity diagrams. In this chapter we focus on behavior mainly because this was the most challenging part with respect to transformation.

The models at abstract level were derived from documentation on global software functionality and design as well as about component functionality and application interfaces. Because we wanted to transform models of an abstract level into models on a concrete level, we also needed models of the latter level. The activity and class models of the concrete level were derived from documentation on component functionality and application interface component design as well as from an existing framework. This framework is software that allows synchronized execution of physical scans in a distributed system.

Consider a system which behavior is shown in Figure 6.3. In this system, the logical actions are executed in a concurrent way. This concurrency is demonstrated by two subsystems – `NetworkConnection` and `DisplayIcon`. `DisplayIcon` is responsible to indicate the status of the connection; either on or off. The test starts with a call of `CHECK_CONNECTION` on the `DisplayIcon`. The sub-system `NetworkConnection` is in the 'passive' phase. In general, the sub-systems can be in passive and in active phases, indicated by stereotypes 'passive' and 'active' respectively. If a physical action

is called, a ‘passive’ phase of the sub-system makes sure that no other physical actions can use the `NetworkConnection` during this call. The result of the call is stored in variable `CNT`. After that we check if the variable `CNT` is equal to `DISCONNECTED`. If expression %`CNT`%=`DISCONNECTED` is true, variable ENC is set to `CONNECT` and is used as an input for the `ESTABLISH_CONNECTION` on the `NetworkConnection` subsystem. After that, the system tests again if the `DisplayIcon` gets the connection. If the `DisplayIcon` result `CNT` is equal to `CONNECTED`, the test is terminated by return value `OK` (default). If it is not the case, we are sure that the indicator is out of order and we terminate by return value `OUT_OF_ORDER` [52]. The concrete activity diagrams

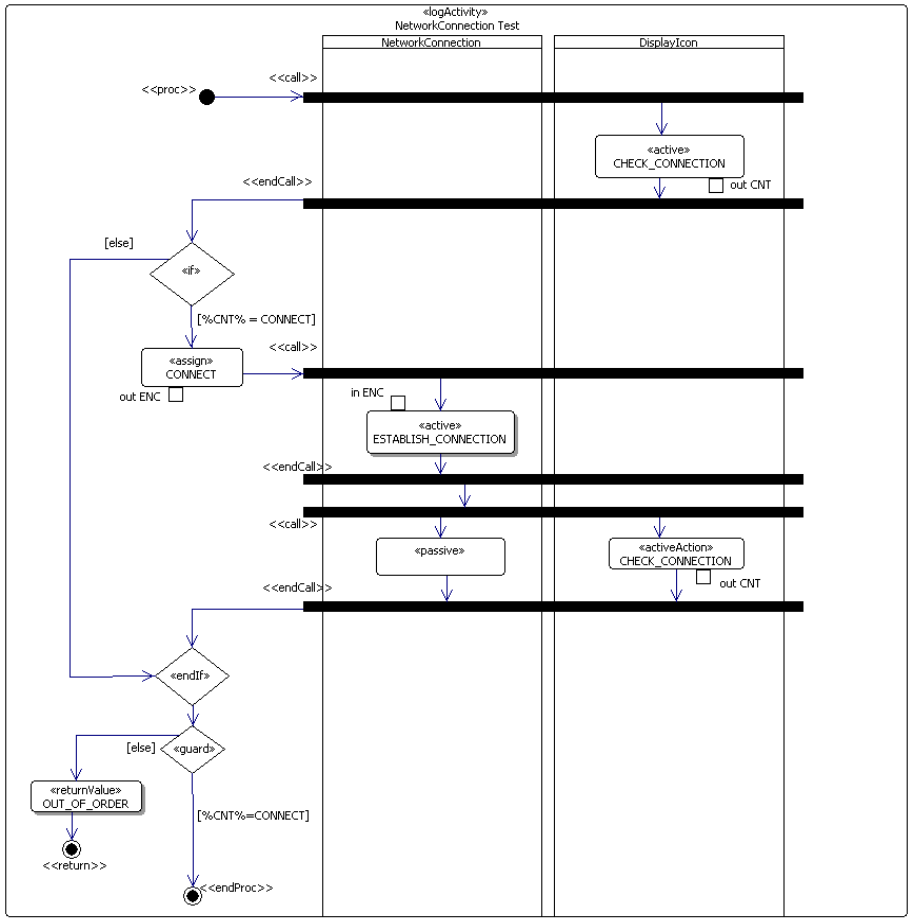


Figure 6.3: Example of an abstract level model. Taken from: [52].

of the `NetworkConnection` test describes the queuing of the physical actions. The control actions are also part of the diagram – they are depicted on the left (e.g., IF1, ASSIGN2, et cetera) and physical actions are depicted on the right. This concrete model is meant to be the target model that we expect to have after we apply the model transformation to the abstract model.

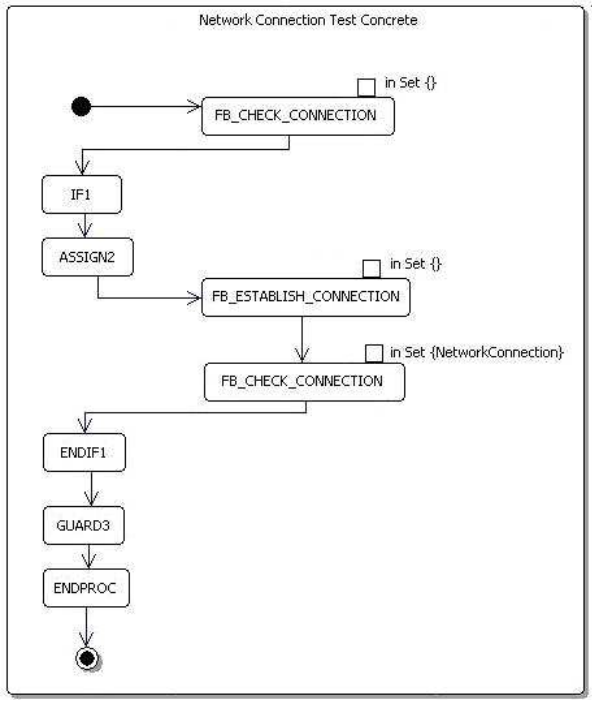


Figure 6.4: Example of a concrete level model. Taken from: [52].

The modeling of the logical and the concrete model was supported by using a profile. This UML mechanism extends meta-models by means of stereotypes, tagged values and constraints. We applied profiling, as opposed to meta-modeling because of the expectation that (re)using standardized languages as much as possible improves communication within the organization. Furthermore, tools specifically designed for a standardized language like UML can already offer advanced support for a domain specific language (DSL) that is based on this language.

6.3.2 Transformation

A model transformation takes as input a (source) model and produces as output a (target) model. Both models may be restricted by the requirement that they conform to a

source or target meta-model. A *model transformation* can be carried out to refine the model into a model of a lower abstraction level or from a lower abstraction level to a higher one, e.g. reverse engineering. In addition, the abstraction level may remain the same during the transformation, in case the model is re-factored. A transformation takes a source model as input and transforms it into a target model or other artifact, which often means that source and target are different. However, in case of a model redesign the source and target model are the same. (Software) synthesis is a particular form of model transformation during which a model is transformed into an implementation, e.g., in the form of code.

We have chosen the Query View Transformation (QVT) as a set of rules to define the transformation. QVT was chosen because it is an upcoming standard [83], advocated by the Object Management Group (OMG), as well as that we wanted to have confirmation about the usefulness of this standard.

A *query* is a well-defined expression of a query language (e.g., OCL) that is evaluated over a model. It is used to identify and select the required elements from the input model when one wants to map elements from input model to output model. *Views* are the models which are obtained from other models. A *transformation* takes an input model and generates an output model. A transformation of a platform independent model to a platform specific model is an example of transformation.

The QVT specification proposes a hybrid approach, which combines declarative (Relations) and imperative (Operational) languages. A declarative approach to transformation describes the goal in terms of relations between the source and target patterns and contrasts with an imperative approach, which defines explicit intermediate steps to reach the goal. During the analysis phase of our case study we found that the Relations language, and with that the declarative part of QVT, is only theoretically applicable to models. It is not yet implemented in state-of-the-art tooling for QVT, e.g., Borland® Together® Architect 2006 [52].

We found that the imperative part of QVT is partly implemented in Borland® Together® Architect 2006. Therefore, we applied this imperative part in our transformation. We also looked at the compliance of Borland® Together® Architect 2006to OMG standards.

An imperative language has some side-effects and forces the programmer to be explicit about the sequence of steps to be taken when it is executed. The structure (class) models could be relatively easily transformed from source to target models by using OMG's Query View Transformation (QVT) standard. It was more difficult to transform the behavioral (activity) diagrams. Besides the fact that unconstrained use of activity diagrams can easily lead to incomprehensible models, their transformation to a specific target platform is not feasible in general. It would require the target platform to support the behavior of arbitrary complex activity diagrams, which cannot and should not be expected from a dedicated platform. Arbitrary concepts have to be detailed to a level that can unambiguously be interpreted by the platform.

This is the reason why we defined language constructs that helped us to decompose behavioral descriptions of source into target models. This is addressed as Composi-

	Abstract level	Concrete level
Class diagrams	2	6
Activity diagrams	2	17

Table 6.1: Number of diagrams produced during the case study.

tional language [45]. This language is compositional subset of UML 2.0 activity diagrams and consists of any activity that can be built with patterns that we called `proc`, seq, `assign`, `guard`, `if` and `call`. For practical implementation we defined additional features in the Together tool to enable the right sequencing in practice [52].

6.4 Findings and interpretation

The model transformation exercise started with specifying the high-level design of coordination as a set of UML2.0 Activity diagrams and Class diagrams. Next step was to specify the system at the concrete level of abstraction as a set of UML2.0 Activity diagrams and Class diagrams too, see Section 6.3.1. The number of diagrams that was constructed is presented in Table 6.1. With respect to our complete transformation approach we have the following findings:

Understandability The use of a language that is compositional helps in the construction of specifications that are easy to understand. This principle holds under the assumption that the language's constructs are (semantically) rich enough for a designer to express his or her thoughts in a clear and concise way. The models were evaluated by two ASML designers to examine their understandability. The designers considered the abstract model to be complete enough to describe the cases in an existing system and to be used for the model transformation. As expected, the concrete model was perceived as complex and only understandable for a platform specialist. This is caused by the low abstraction level of the model. Although those comments were made, it was concluded that the models are understandable. They can be used as input and as expected output for the transformation. The structure (class) models could be transformed relatively easy from higher level of abstraction to an implementation by using OMG's Query View Transformation (QVT) standard. However it was more difficult to transform the behavioral (activity) diagrams as stated already in Section 6.3.2.

Effort The modeling and transformation definition activities took nine months. Two academic researchers were working during this time period and needed in total 15 person months of work. One defined the transformation language, the other drew the models and implemented the transformation rules in Borland Together.

Reuse of transformation Transformations from models of an abstract level to a concrete level require transformation rules. Therefore, we reconstructed a simple as well as a complex instance of the coordination idiom. We found out that our transformation rules could be applied to the simple as well as to a very complex instance: both instances could be transformed from abstract to concrete models.

Tool/standard compliance The modeling and QVT-transformation was implemented by a state-of-the-art tool: Borland® Together® Architect 2006. Incompatibilities between the Borland® Together® Architect 2006standard implementation and the MDA standard exist and they concern the UML2.0 specification as well as the QVT specification. Most of the problems could be overcome using the extensibility capabilities of the tool set. However, the lack of UML2 compliance has resulted in alternatives for modeling at abstract level (e.g. loops).

6.5 Conclusions

This chapter reports about a case study on the feasibility of modeling and model transformation of a specific idiom, called coordination. We conclude that this modeling and transformation is possible. Based on these case study experiences, ASML has already built a prototype that generates production quality code.

However, applying this kind of model-based approach in practice requires special attention on the following issues:

Specific skills Developing a coherent high level language for a certain domain and defining transformations for such a language to other languages requires skills that are not generally available in a software engineering population.

Correctness In our case study, the combination of experience and testing was sufficient to come to a meaningful setup. A formal proof of correctness would require formalization of the semantics on all levels of abstraction involved. In practice, such a level of rigor will not be present and the confidence in the transformations will have to be based on application and experience.

Immaturity of tools The tools used in this project were not fully compliant with the UML and QVT standards. In general, most tools have only limited compliance with the relatively new standards in this area. Given the current state of maturity, the evolution of modeling tools will require regular reconsideration of modeling and transformation solutions created.

We end this chapter with the advantages of our approach and with future challenges:

Advantages The high level models make it easy to understand the design of a system in contrast to reading the code. The generative approach guarantees that the code indeed corresponds to the high level description. The high level model, capturing the essential semantics, thus becomes a primary artifact in the development process.

Challenges The case study showed that we could define generic transformation rules that could be applied for simple and complex instances of this particular idiom. We think that the approach presented in this chapter is applicable to other idioms as well. However, the identification of such idioms requires clear criteria, which are not available yet. It is also not clear what problems can be expected with both modeling and transformations when integrating idioms at a high abstract level.

Chapter 7

Embedded systems modeling, analysis and synthesis

Authors: Mark van den Brand, Luc Engelen, Marc Hamilton, Andriy Levytskyy, Jeroen Voeten

Abstract Model-driven engineering (MDE) refers to the systematic use of models as primary engineering artifacts throughout the engineering life cycle. In this chapter we will show how executable models can be used to aid the design process of a light control subsystem of a wafer scanner. We make an explicit distinction between the model of the application logic and the platform on which it is deployed. It will be shown how the performance of this subsystem can be predicted in an early phase of the design process, before the system is implemented in terms of hardware and software components. The executable model in addition allows a prototype software implementation to be derived from it automatically in a predictable way. The executable model is expressed in POOSL, a special-purpose modeling language targeting real-time embedded systems. To allow an embedding in a future MDE environment, an experiment is performed to express a similar model in the general-purpose modeling language UML from which the executable models can be derived through model transformations. These transformations further allow one to combine an application model created in UML with a platform model created in POOSL and analyze this combined model.

7.1 Introduction

One benefit of using models as a primary artifact in the development process is the ability to evaluate certain aspects of the system in an early stage. In embedded systems in particular, a re-occurring problem is the balance between hard- and software.

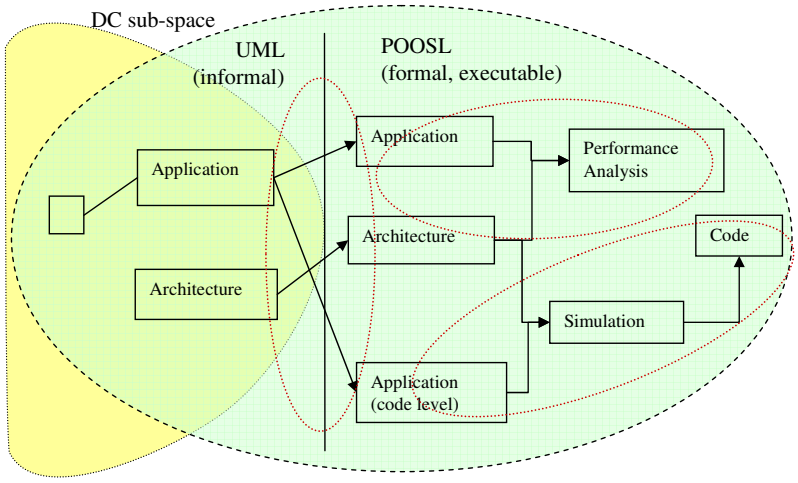


Figure 7.1: Positioning of the chapter in a model space.

This problem motivated the construction of a high-level performance analysis model that clearly separates the hardware architecture from the application logic, using the executable modeling language POOSL. However, the results of such a model analysis are only of limited use if the corresponding realization does not match the assumptions made in the model. Clearly there is a need to synthesize a model into a realization while preserving the essential properties of the model. This has driven the development of an executable model from which an implementation can be generated in a property-preserving way. A large portion of such executable models are based on information that was generally available in design documentation or in the heads of the engineers involved. In fact, the POOSL models are merely a formal representation of available information. It is attractive to capture such information at one shared place in a model space. Other models share this information by retrieving it from this single source. Based on this idea, we started to investigate how this can work in a model space as shown in Section 1.5. The idea is to capture the essential properties of a system in UML, abstracting from the specifics of the POOSL formalism. The first step to cover in this investigation is then to derive the POOSL performance analysis model from the UML model using model transformations. The resulting transformations then also allow one to combine an application model created in UML with a platform model created in POOSL and to analyze this combined model. The cases described in this chapter are retrieved from a sub-space that concentrates on the light control system in the ASML wafer scanner, depicted in Figure 7.1. The different cases are indicated by the dotted ovals.

This chapter is organized as follows. The performance modeling exercise is explained in Section 7.2. Section 7.3 gives a brief overview of the modeling language POOSL and how it was used to model the light control subsystem. Section 7.4 focus on the predictable generation of code from executable models. Embedding these executable models in an MDE environment based on UML is subject of Section 7.5. Conclusions and are drawn in Section 7.6 and direction for future work are sketched.

7.2 Performance modeling of the light control subsystem

The goal of the performance modeling exercise was to investigate design alternatives together with their impact on the timing performance of the light control subsystem of a wafer scanner. The goal of this subsystem is to generate a sequence of light pulses such that a pattern of an electronic circuit is projected on a silicon wafer. The timing performance and especially the duration between two consecutive light pulses must be controlled very accurately. Major difficulty is that concurrently to the hard real-time processes of emitting these light pulses, soft real-time processes are running that take care of communication and negotiation with other subsystems. Deploying these processes on the hardware platform in such a way that all timing requirements are met was a major design challenge.

To be able to investigate alternative deployment strategies we constructed both a POOSL model of the light control application logic and a separate POOSL model of the hardware platform, following the Y-chart approach [67, 102] as depicted in Figure 7.2.

An application, modeled as a set of communicating application processes, and the platform on which the application runs are modeled separately. A mapping assigns the different processes to appropriate processing units of the platform and communications between the processes on corresponding communication resources. Together the application model, the platform model and the mapping form a system model which performance properties can be analyzed, either analytically or by simulation. For the light control subsystem, these performance properties concern the (maximal) latency between two light pulses and the utilization of the platform resources such as busses and processors. Based on these analysis results, one can decide to adapt the application, the architecture or the mapping. This can even be computer-supported, for instance by utilizing automated design-space exploration techniques [82]. Once analysis results are satisfactory, the application can be mapped on the architecture, preferable in an automated way.

The performance modeling exercise was very very valuable. The model provided insight in overall subsystem behavior. Design documentation provided many details, but did not shed light on the big picture. The design documentation specified a number of separate message sequence charts, but the (timed) behavior emerging from their collaborations was unclear, also because the design documents were informal and not executable. The executable model in POOSL combined the separate pieces into one

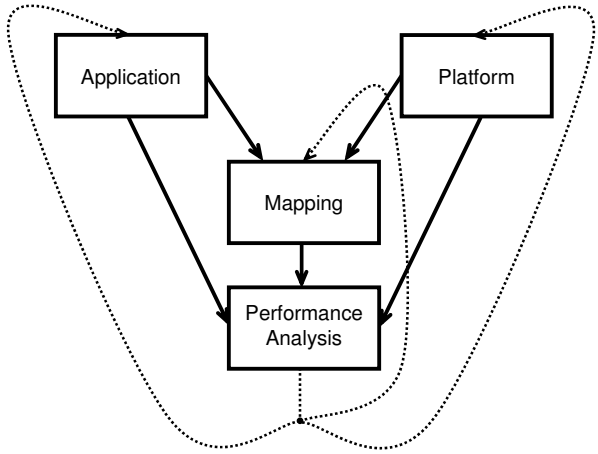


Figure 7.2: The Y-chart approach.

unified whole. The immediate feedback obtained from the model eased communication with the design team. It allowed us to understand and verify the light control subsystem before it was implemented in terms of hardware and software components. The mapping of the application onto the platform was investigated resulting in a concrete task priority scheme. Also, a timing problem concerning the forwarding of application data to an FPGA was found. Although designers new that the realization behaved incorrectly once and a while, the root course for this problem was not know. This root course was discover in the executable model which also provided inspiration for a robust solution and possibilities for verification. Last, but not least, an expected timing bottleneck in an on-board communication switch turned out to be a non-issue. On the other hand, cache misses turned out to cause a major timing problem. Based on this identification, strategies to minimize cache misses were developed.

7.3 The POOSL language

POOSL is a formal modeling language for complex real-time embedded systems. The language form the core of the SHE (Software/Hardware Engineering) methodology [89, 96]. The language combines a data part with a process part. The data part is based upon the concepts of traditional sequential object-oriented programming languages such as C++ and Smalltalk. The process part is based on a probabilistic and timed extension of the CCS process algebra. A specification in POOSL consists of a collection of asynchronous concurrent processes that communicate synchronously by passing messages over channels. Processes can contain, operate on and exchange data objects. A key

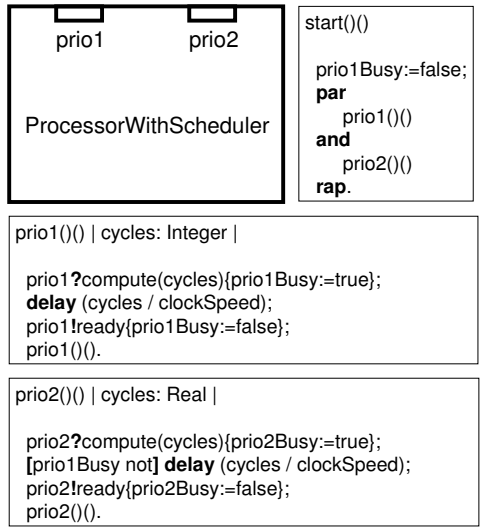


Figure 7.3: A platform model with a processor and scheduler.

feature of POOSL is its expressivity. By offering a set of blending orthogonal language constructs (such as parallel composition, guards, recursion, aborts, interrupts and delays), intricate dynamic behavior can be described in a compact way. Once POOSL models have been constructed, they can be executed and interesting design properties can be determined. The execution of POOSL models is defined by a formal semantics, which allows the model to be verified with respect to correctness and performance properties. The interactive model editing and simulation tool called SHESim supports the SHE methodology and the POOSL language. The tool is used to incrementally specify and modify classes of data and processes in a graphical way. It further allows models to be simulated, several types of model viewers to be opened and message-sequence charts to be generated automatically. In addition the execution engine Rotalumis allows very large model to be executed and analyzed and forms the basis for an automatic correctness-preserving mapping on the target implementation platform [57].

We used the POOSL language to make a concrete model following the Y-chart approach explained in the previous section. Both application processes and platform resources were modeled as POOSL processes. The mapping of application processes onto these resources was modeled by means of the synchronous message passing mechanism of POOSL. Different deployments could thus simply be explored by modifying channels between application-level and platform-level processes.

An example of a platform model consisting of a processor with a real-time operating system is depicted in Figure 7.3.

The rectangle denoted *ProcessorWithScheduler* is a process that has two ports

named *prio1* and *prio2*. Its models a processor on which a pre-emptive real-time operating system runs that has two priorities. The highest priority corresponds to port *prio1* and the lowest to port *prio2*. Application processes (not depicted in the figure) connect to one of these ports, depending on the priority on which they should run. The behavior of the *ProcessorWithScheduler* is also shown in Figure 7.3. The process starts by executing a method (a sort of function) called *start()()*. In this method a variable *prio1Busy* (global to the process) is initialized to false. This variable indicates whether any task is running at the corresponding priority. The two methods *prio1()()* and *prio2()()* are started in parallel. Method *prio1()()* deals with requests from an application process to perform a computation at priority 1 taking a certain number of clock cycles. Blocking statement *prio1?compute(cycles)* receives message *compute* from port *prio1* with *Integer* parameter *cycles*. Upon reception of such a message the *prio1Busy* variable is set to true and a time delay is performed in accordance with the requested number of cycles. After having performed the delay, the requesting processes is sent a *ready* message and the *prio1Busy* variable is set to false. Notice that only one request on port prio1 can be serviced at the same time (a task of a certain priority cannot pre-empt a task of the same priority). Method *prio2()()* deals with requests on port *prio2* and is almost similar to method *prio1()()*. The difference is in the guarded delay *[prio1Busy not]delay(cycles/clockSpeed)* indicating that the delay may only be performed in case no task with a higher priority is running. If the guard is closed when a part of the delay has already been performed, carrying out the remainder of the delay is temporarily stopped until the guard opens up again (because the higher priority task is finished). This models the fact that higher-priority tasks are allowed to pre-empt lower-priority tasks. Notice that the combination of the language constructs allow quite a complex piece of behavior to be expressed in a very compact way, thereby increasing the speed of modeling. We will come back to this issue of expressivity in Section 7.5 where the embedding in UML is discussed.

7.4 Predictable code generation

The main purpose of executable model explained in Section 7.2 is performance analysis. In the MDE vision however, it should also be possible to derive implementations from models in a mechanized way. Therefore we investigated whether automatic code generation from light control models is feasible.

In general, performance models cannot be directly used to generate code. The main reason is that their purpose is analysis and not synthesis. This means that these models are made in such a way that the properties of interest can effectively and efficiently be derived from them. To do so, all non-interesting details are left out of these models. The application processes in the light control performance model do not contain details of a feed-back control algorithmic, for example. Instead such an algorithm is modeled by sending a *compute* message to the platform model (as depicted in Figure 7.3) and by waiting for the *ready* reply. When used for code generation, such a model must be refined by incorporating the algorithmic details.

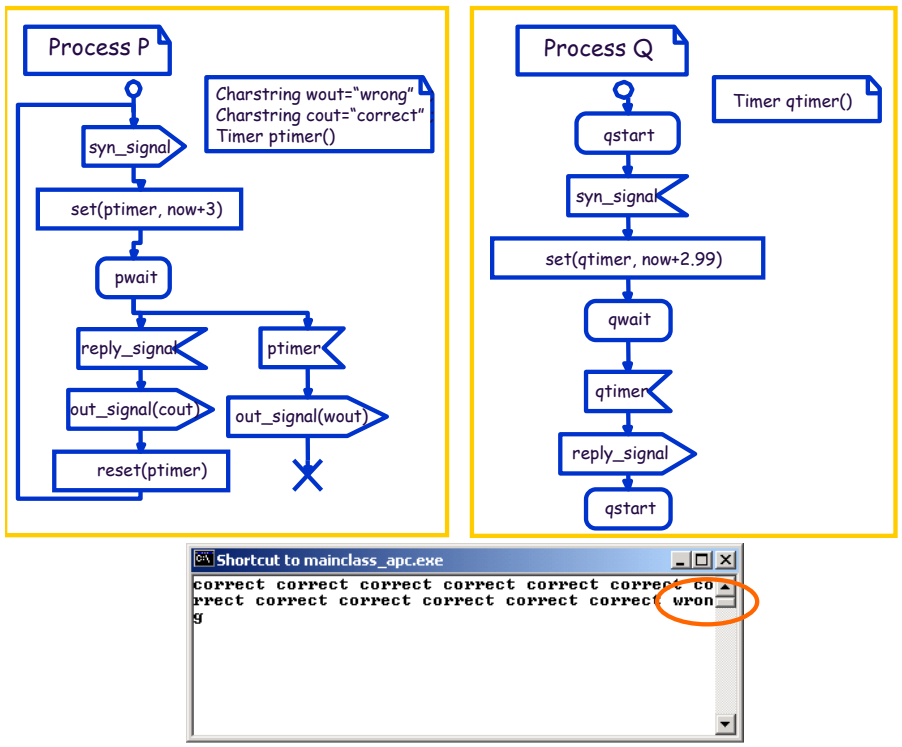


Figure 7.4: Unpredictable code generation in TAU G2.

As explained in Section 1.3.2, models are used to predict properties of a future realization. With respect to code generation this means that properties of the model should be respected by the code generator. If this is not the case, the implementation may show unexpected behaviors, which is of course not desirable. Especially for real-time systems predictable code generation is a major challenge [57]. To illustrate this, consider an example of a model of two communication parallel processes P and Q shown in Figure 7.4.

Process P starts by sending a synchronization signal to process Q after it set a timer to expire after exactly 3 seconds. It then waits for a reply signal from process Q to arrive. In case this happens, it outputs a *correct* message and otherwise, if the timer has expired, it output a *wrong* message. Upon reception of the synchronization signal, process Q returns the reply signal after precisely 2.99 seconds. We created this

model using the TAU G2 tool released by Telelogic [94]. Model simulation revealed that the reply signal is *always* received by process *P* before its timer expires, which is correct. Hence process *P* just produces a sequence of *correct* messages. In the automatically generated implementation this property is not satisfied however, as can be seen in Figure 7.4. Major issue is that the virtual notion of time in the model is simply replaced by a physical notion of time in the implementation (the timer in the example is simply implemented by an operating system timer). As a result, not only the timing of the implementation differs from that of the model, but even the ordering of events can be disturbed by the code generator [58]! To address this issue we applied a predictable implementation approach [57]. The crux of this approach is that the virtual time in the model is not replaced by, but *synchronized to* the physical time of the platform. As a consequence the ordering of events is preserved and the timing properties are preserved up to a small deviation.

A student carried out the code-generation experiment as part of an ASML internship. Starting point was the executable performance model. This model allowed the student to quickly understand the light control subsystem and refine and extend it to a level of detail from which code could be produced. The experiment was successful in the sense that the resulting code executed correctly on the ASML development platform. Since the code is based on a real-time variant of the Rotalumis simulator, the code efficiency (timing and footprint) should be improved. Also distributed implementations on multi-processor systems is not possible yet. In any case, both the performance modeling and code generation experiments showed that engineering with executable models is very well possible and promising.

7.5 UML embedding

Different models of computation are required to capture different concerns for different purposes, as we also explained in Section 1.5. To be able to construct a realization or to be able to predict properties thereof, it is important to be able to combine these models together in a consistent way. In the performance modeling exercise (see Section 7.2) we achieved this by composing a model of the application logic with a model of the implementation platform. These models, however, are both expressed in the formal modeling language POOSL. To study the combination of different modeling languages, we tested the idea to capture the application model in UML instead, and to develop automatic transformations to the (original) formal model. Another important rational behind this idea is to avoid capturing the same information at different locations in the model space; shared information should be captured at one single location. This shared information should form the root from which models for dedicated purposes (e.g. design review, verification or performance analysis) can be derived. As an additional advantage, such a transformation potentially minimizes the number of different languages designers have to be acquainted with.

To test these ideas, the POOSL performance model of the light control subsystem, as explained in Section 7.2, was used as a case study. A mapping from POOSL to

the UML was made to get acquainted with the constructs offered by the UML for the modeling of behavior and to explore the gap between the UML and POOSL. After this initial exploratory phase, data and processes were investigated separately. The main focus in both investigations is on modeling behavior using only the standard language constructs offered by the UML.

In the remainder of this section, instances of meta classes of the UML meta model, constructs of the POOSL language, and names mentioned in examples that identify processes, ports or other objects are italicized.

Data classes

First, modeling the behavior of data classes was investigated. We considered four ways of modeling this behavior offered by the UML.

- Incorporating the original POOSL code using *OpaqueBehaviour*.
- Declarational specification using OCL.
- Operational specification using *Activity* diagrams.
- Operational specification using a textual representation of *Activities*.

Incorporating the original POOSL code in a UML model using *OpaqueBehaviour* was not an acceptable option, since knowledge of POOSL would be required to create and maintain the model.

The declarational specification of behavior using OCL uses contracts specifying preconditions and post-conditions for each method of a data class. Although these contracts give a precise and detailed description of the behavior of the methods, an automated transformation to POOSL is not feasible.

The biggest advantage of the operational specification using *Activity* diagrams is the fact that *Actions* in the UML correspond with expressions in POOSL. This makes a straightforward automated transformation within reach. The downside of the operational specification is the size of the diagrams. Even the diagrams describing simple behavior can grow very large.

This downside of the operational specification of behavior can be tackled by using a textual counterpart of the graphical *Activity* diagrams. Such a textual representation is called a surface language.

The graphical *Activity* diagrams have been chosen as a starting point of the transformation instead of a textual alternative. A suitable textual alternative is not available and creating one was not possible given the resources.

Process classes

The mapping from POOSL to the UML was used to create two UML models that specify the behavior of the processes of the POOSL performance model. In the first

version, all behavior is modeled using *Activities*. The second version divides behavior into two categories. State-independent behavior is modeled using *Activities*, whereas state-dependent behavior is modeled using *StateMachines*.

The first version of the model has two important properties. The transformation to POOSL is straightforward, because of the correspondence between *Actions* in the UML and statements in POOSL. However, because of the similarities, the added value of the graphical approach is questionable.

Figure 7.5 shows three *Activities* that model the behavior of a process called *ProcessQueue*, as is done in the first version of the model. This process simultaneously receives objects from a port called *in* and sends them via a port called *out*. The fact that objects are sent and received concurrently is modeled by the *Activity* named *start*. The *Activity* named *receive* specifies that after receiving an object, the process stores it in a datastructure called *DataQueue*. As long as this *DataQueue* contains objects, these objects are retrieved from the *DataQueue* one by one and sent to another process via the port called *out*, as is modeled by the *Activity* named *send*.

The most important property of the second version of the model is the separation between state-independent behavior and state-dependent behavior. This separation makes a clear distinction between process behavior that can occur at any time and that is always triggered by another process, and process behavior which occurrence depends on previous actions.

Figure 7.6 and Figure 7.7 show the state-dependent behavior and the state-independent behavior of the process called *ProcessQueue*, respectively. These figures illustrate the approach taken for the second version of the model. They describe the same behavior as the *Activities* in Figure 7.5. Figure 7.6 consists of a *StateMachine* and an *Activity*. The *Activity* describes the *Effect* of the rightmost *Transition* in the *StateMachine*. The figure shows that the process sends objects via port *out*, as long as the *DataQueue* is not empty. The *Activity* in Figure 7.7 can be called at any time during the life cycle of process *ProcessQueue*. Processes that call this *Activity* pass an object as argument. This object is then stored in the *DataQueue* of the process *ProcessQueue*.

Mismatches between the languages

When modeling the behavior of the processes of the performance model of the light control subsystem, it turned out that some constructs offered by POOSL have no counterpart in the UML. The *guarded delay* used in the example of Section 7.3 is an example of such a construct. We found no suitable counterpart and chose to model the process *ProcessorWithScheduler* on a lower level of abstraction.

Figure 7.8 shows the *StateMachine* that models the behavior of this adapted version of the process. To save space, textual alternatives are used to describe the *Effects* of the *Transitions*, instead of the usual graphical representations. The adapted version of the process uses an instance of the data class *PriorityQueue*, called *PQ*, to store the amount of time that has to be spent on the computation associated with each priority. When the process receives a message named *compute* from the port *prio1*, it stores the integer

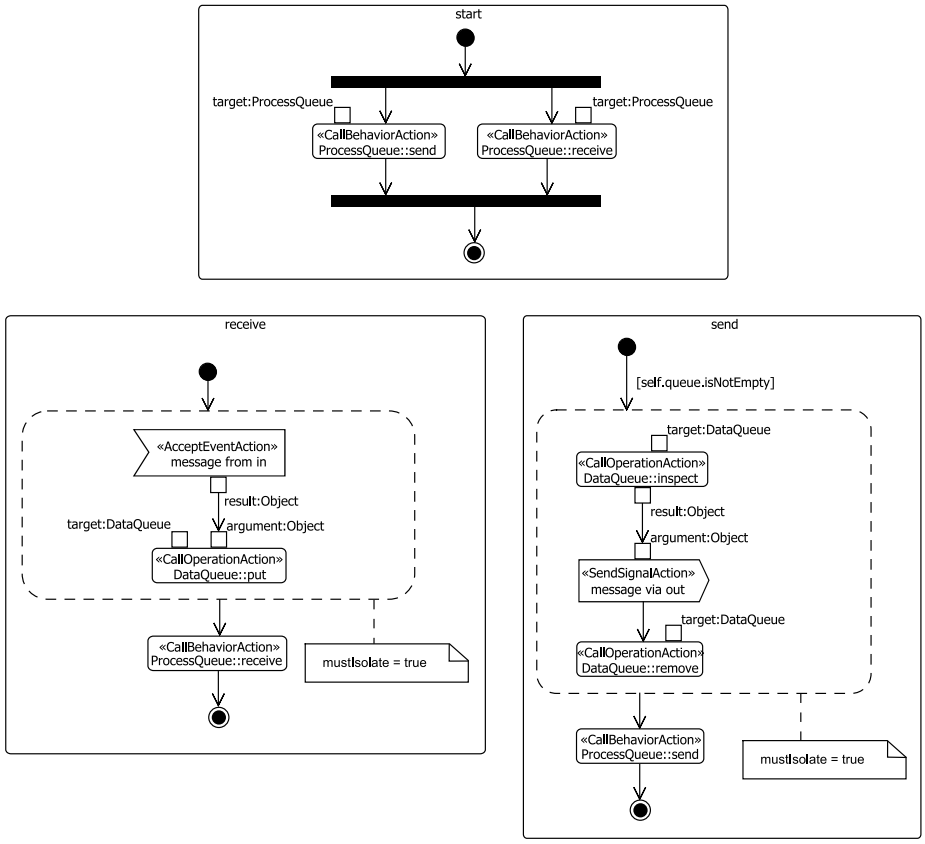


Figure 7.5: Behavior of the class *ProcessQueue* modeled using *Activities*.

argument passed with the message in the *PriorityQueue* at the slot associated with the first priority. The behavior concerning the port *prio*2 is analogue to the behavior described for port *prio1*. When the *PriorityQueue* is not empty, the process delays for a fixed number of milliseconds, given by the integer constant called *grain*, and then decreases the value associated with the highest priority. When the value associated with a priority is decreased to zero, a message named *ready* is sent via the corresponding port.

Formal semantics

A transformation from a subset of the UML to POOSL provides a way to tie the formal semantics of POOSL to this subset and give it a formal foundation. As an alternative, the work of Hooman and Van der Zwaag [55] could be studied as a starting point for

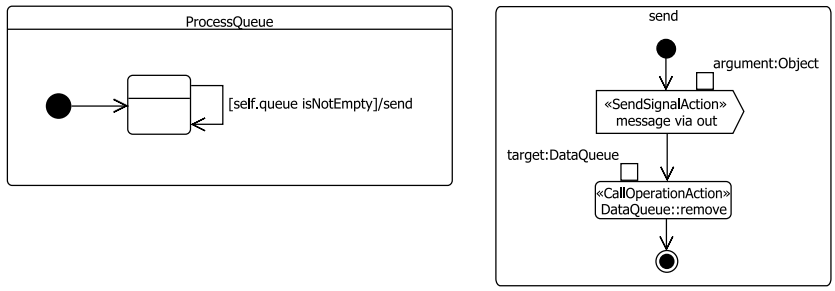


Figure 7.6: State-dependent behavior of the class *ProcessQueue* modeled using an *Activity* and a *StateMachine*.

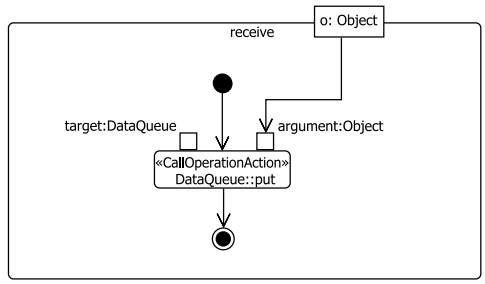


Figure 7.7: State-independent behavior of the class *ProcessQueue* modeled using an *Activity*.

a formal description of the semantics of the subset of the UML used in the second version of the UML model of light control. Having independent descriptions of both the semantics of POOSL and the semantics of this subset paves the way for proving that the transformation preserves semantics.

7.6 Conclusions and future work

Although not all research has concluded yet and many new questions rise, conclusions can be drawn based on the results thus far.

- We experienced that both graphical and textual concrete syntaxes have their pros and cons. Since making models as concise as possible is an important issue, it is desirable to provide the best means of interacting with the model. Therefore it is desirable to combine textual syntax with graphical syntax.

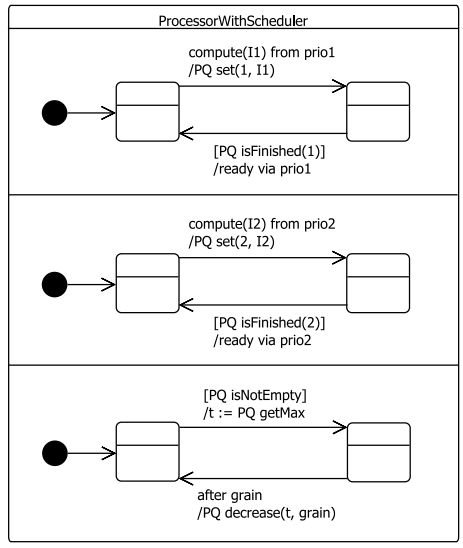


Figure 7.8: *StateMachine* describing the behavior of the process *ProcessorWithScheduler*.

- The separation between state-dependent behavior and state-independent behavior seems to result in more concise models. It appears that the use of the most appropriate paradigms prevails above the use of a single paradigm, as long as the relations among them are well-defined.
- There are several semantic constructs that can be expressed easily using the domain specific language POOSL but result in awkward constructions in the general purpose UML language. A similar result was found when evaluating the POOSL code for a calibration function against its more concise description in Matlab. Clearly, the expressive power of DSL's is an important aspect in the trade-off between using domain specific languages and general purpose languages.
- The model-based performance predictions turned out to match closely with the properties of the actual system that resulted from the traditional engineering process. This demonstrates that even without code generation, validation and verification at a high level are very useful.
- The semantics of the language(s) used, especially for shared models, must be well-defined. UML in its current state suffers from many unclear semantic issues, either by erroneous or incomplete specifications or by explicitly delegating semantics to the application domain.
- Property-preserving code generation as applied in this chapter is feasible, but in

its current form still has a number of limitations. Generated implementations are not resource-efficient enough for the embedded systems domain. In addition, distributed and mixed hard and soft real-time systems cannot be dealt with yet.

In general, MDE in a large industrial context raises many questions that give rise to additional research. Until this stage, the research has mostly been explorative simply because the area of MDE is still immature. With respect to future work we therefore address only those issues that follow from the results described in this chapter.

- The use of surface languages seems to lie in their possibility to combine graphical and textual languages. The exploration of this topic was stopped during our research. However, understanding the possible role of surface language in an MDE context is an interesting topic for future research.

- The property-preserving code-generation method for the POOSL language must be improved. Increasing resource efficiency and dealing with distributed and mixed hard and soft real-time applications are challenging directions for future research.

- One goal of the work on embedding a formal language in UML is to capture shared modeling information at one single location in the space of models. Another goal is to combine different concerns expressed in different languages together in a semantically sound way. Although an automatic transformation from UML to POOSL has not been developed yet, we are confident that this is very well possible. However, to verify the feasibility of a shared information model, a transformation to at least one other formalism (targeting e.g. formal verification) should be developed.

- The semantics of the shared models at ASML that will be expressed in UML must be made precise. The work in [55] is an attractive starting point for this.

- It is important to find the appropriate models of computation that are relevant to the engineering process of ASML. The challenge of further research is to identify the appropriate models of computation and to understand their semantic relations.

- It should be investigated what information is shared among the engineering activities at ASML. The results can be used to define a unifying modeling language that can be used for shared models.

Chapter 8

Designing and documenting the behavior of software

Authors: Gürcan Güleşir, Lodewijk Bergmans, Mehmet Akşit

Abstract The development and maintenance of today's software systems is an increasingly effort-consuming and error-prone task. A major cause of this problem is the lack of formal *and* human-readable documentation of software design. In practice, software design is often informally documented (e.g. texts in a natural language, 'boxes-and-arrows' diagrams without well-defined syntax and semantics, et cetera), or not documented at all. Therefore, the design cannot be properly communicated between software engineers, it cannot be formally analyzed, and the conformance of an implementation to the design cannot be formally verified.

In this chapter, we address this problem for the design and documentation of the behavior implemented in procedural programs. We introduce a solution that consists of three components: The first component is a graphical language called VisuaL, which enables engineers to specify constraints on the possible sequences of function calls from a given program. Since the specifications may be inconsistent with each other, the second component of our solution is a tool called CheckDesign, which automatically verifies the consistency between multiple specifications written in VisuaL. The third component is a tool called CheckSource, which automatically verifies that a given implementation conforms to the corresponding specifications written in VisuaL.

This solution has been evaluated empirically through controlled experiments with 71 participants: 23 professional developers of ASML, and 49 Computer Science M.Sc. students. These experiments showed that, with statistical significance of 0.01, the solution reduced the effort of typical maintenance tasks by 75% and prevented one error per 140 lines of source code. Further details about these results can be found in Section 10.5.

8.1 Introduction

VisuaL and the associated tools are outcome of our close collaboration with ASML. Our collaboration with ASML was divided in four phases: In the first phase, we identified a number of effort-consuming and error-prone tasks in software development and maintenance processes. In the second phase, we developed VisuaL and the tools to automate these tasks. In the third phase, we conducted formal experiments to evaluate CheckSource. In the fourth and the final phase, ASML initiated a half-year project to embed VisuaL and the tools into their software development and maintenance processes. This project was ongoing at the time of writing this chapter.

The remainder of this chapter is structured as follows: In Section 8.2, we present some common problems of today's software engineering practice. In Section 8.3, we explain our approach for improving the software engineering practice. In Section 8.4, we introduce VisuaL, and illustrate it with seven use cases. In Sections 8.5, and 8.6, we respectively introduce CheckDesign and CheckSource. We conclude with Section 8.7.

8.2 Obstacles in the development of embedded software

In the first phase of the Ideals project, we investigated the software engineering process of ASML, and identified a number of general problems. In this section, we explain these problems.

8.2.1 Informal documentation of software design

Natural languages are frequently used in the industrial practice, for documenting the design of software. For instance, we have seen several design documents containing substantial text in English, written in a 'story-telling' style. Although the unlimited expressive power is an advantage of using a natural language, this freedom unfortunately allows for ambiguities and imprecision in the design documents.

In addition to the texts in a natural language, design documents frequently contain figures that illustrate various facets of software design, such as the structure of data, flow of control, decomposition into (sub)modules, et cetera. These figures provide valuable intuition about the structure of software. However, typically such figures cannot be used as precise specifications of the actual software, since they are abstractions with no well-defined mapping to the final implementation in source code.

As we discuss in Sections 8.2.2 and 8.2.3, ambiguous and informal software design documents are a major cause of excessive manual effort and human errors during software development and maintenance.

8.2.2 Obstacles in the software development process

In Figure 8.1, we illustrate a part of the software development process of ASML, showing four steps:

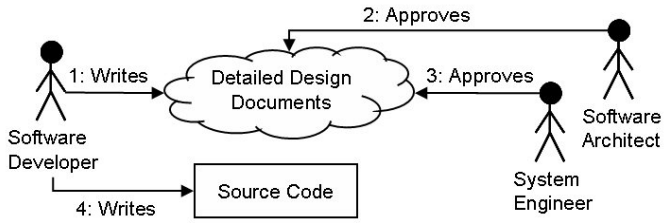


Figure 8.1: This figure shows part of the software development process at ASML.

In the first step, a software developer writes detailed design documents about the new feature that she will implement. The detailed design documents are depicted as a cloud to indicate that they are informal and potentially ambiguous.

In the second step, a software architect reviews the documents. If the architect concludes that the design of the new feature 'fits' the architecture of software, then she approves the design documents.

In the third step, a system engineer reviews the design documents. If the system engineer concludes that the new feature 'fits' the electro-mechanical parts of the system, and fulfills the requirements, then she approves the design documents.

In the fourth step, the developer implements the feature by writing source code. The source code is depicted as a regular geometric shape (i.e. rectangle in this case) to indicate that the source code is written in a formal language.

After the feature is implemented, it is not possible to conclude with a large certainty that the source code is consistent with the design documents, because the design documents are informal and potentially ambiguous. Therefore, the following problems may arise:

- The structure of the source code may be inconsistent with the structure approved by the software architect, because the architect may have interpreted the design differently than the software developer.
- The implemented feature may not 'fit' the electro-mechanical parts of the system, because the system engineer may have interpreted the design differently than the software developer. In such a case, the source code is defective.

8.2.3 Obstacles in the software maintenance process

In Figure 8.2, we illustrate a part of the software maintenance process of ASML, showing five steps: In the first step, a developer receives a change request (or a problem report) related to the implementation of an existing feature. If the developer concludes that the change request has an impact on the detailed design, then she accordingly modifies the detailed design documents, in the second step. If the design documents are modified, then a software architect and a system engineer review and approve the modified design documents, in the third and the fourth steps. In the fifth step, the developer implements the change by modifying the existing source code.

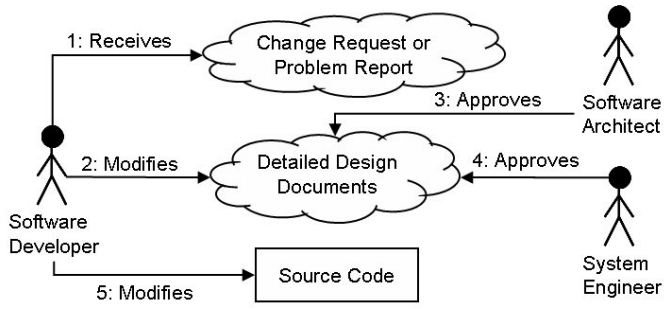


Figure 8.2: This figure shows part of the software maintenance process at ASML.

In practice, engineers can follow shortcuts in the maintenance process explained above, because they are often urged to decrease the time-to-market of a product. They can skip the second, third, or fourth steps, because the design documents are not a part of the product that is shipped to customers. This shortcut leads to the following problems:

- The source code 'drifts away' from the design documents. More precisely, the design that is implemented in the source code becomes substantially different than the design that is written in the documents. In such a case, the design documents become useless, because the source code is the only artifact that 'works', and the design documents do no longer provide any useful information about the source code.
- Since the design documents become useless, a developer has to directly read and understand the source code, whenever she needs to modify software. Consequently, maintenance becomes more effort-consuming and error-prone, because the developer is constantly exposed to the whole complexity and the lowest level details of software.
- Since the design documents become useless, the software architect and the system engineer cannot effectively control the quality of software during evolution, which results in the same problems listed in Section 8.2.2.
- Since the design documents become useless, the initial effort spent by the developer to write the design documents, and the effort spent by the software architect and the system engineer to review them, are no longer utilized.

The problems explained so far in Section 8.2 are more broadly explained in Section 1.3.2.

8.2.4 The scope in this chapter

The scope of the problems that we explained so far is too broad to be effectively addressed by a single solution. Therefore, we communicated with the engineers of ASML to determine a sub-scope that is narrow enough to be effectively addressed, general

enough to be academically interesting, and important enough to have industrial relevance. As a result, we chose to restrict our scope to the design and documentation of the control flow within C functions. In the remainder of this section, we explain the reason for this choice.

Abstractly speaking, the manufacturing machines produced by ASML perform certain operations on some input material. These operations must be performed in a sequence that satisfies certain temporal constraints, otherwise the machines cannot fulfill one or more of their requirements. For example, a machine must clean the input material *before* processing it, otherwise the required level of mechanical precision cannot be achieved during processing; loss of precision results in defective output material.

In software, the input material is modeled as a data structure, and each operation is typically implemented as a function that can read or write instances of the data structure. The possible sequences of operations are determined by the control flow structure of a separate function that calls the functions corresponding to the operations.

During software maintenance, the engineers of ASML frequently change the control flow structure of functions, and unintentionally violate the temporal constraints. These violations result in software defects. Finding and repairing these defects is effort-consuming and error-prone, because (a) the constraints are either not documented at all, or poorly documented, as explained in Section 8.2.1, and (b) there are no explicit means for the engineers to tell them if and where the constraints are violated.

Based on these observations, we decided to find a better way to document the temporal constraints, and to develop tools that can help engineers in finding and repairing the defects. As a result, we developed a solution that consists of a graphical language *VisuaL*, a tool for verifying internal consistency of the design *CheckDesign*, and a tool for verifying the consistency between the design and the source code, *CheckSource*.

VisuaL is a graphical language for expressing temporal constraints on operations in a system, in particular on the operations within a specified function body. It aims at being both intuitive (through a UML-style visual notation), precise (a VisuaL diagram can be mapped to a formal representation of automata), and evolution-proof (through the use of wildcards, one can specify only necessary ordering constraints).

8.3 Solution approach

In this section, we explain how our solution (i.e. VisuaL, CheckDesign, and CheckSource) can be used during software development and maintenance. The details of the solution are presented throughout Sections 8.4-8.7.

8.3.1 Adapting the software development process

We present the software development process in which our solution is used, in two steps: (1) the software design process, and (2) the software implementation process.

The Software Design Process

In Figure 8.3, we illustrate the software design process, in which VisuaL and CheckDesign are used. This process consists of four steps:

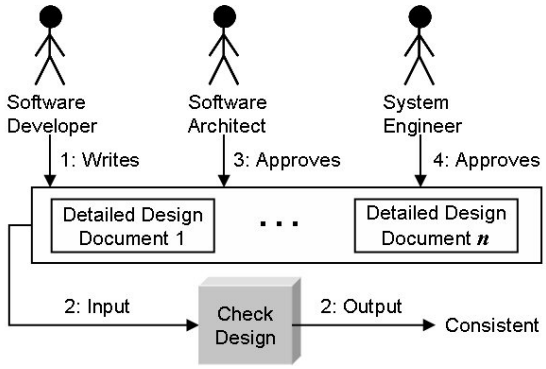


Figure 8.3: This figure shows the design process with VisuaL and CheckDesign.

In the first step, a software developer writes detailed design documents about the feature that she will implement. She uses VisuaL for writing the documents. Therefore, the resulting documents are formal and unambiguous.

In the second step, CheckDesign automatically verifies the consistency between those documents that apply to the same function. If the documents are not consistent, CheckDesign outputs an error message that contains information for locating and resolving the inconsistency. Note that in the original development process (see Section 8.2.2), design level verification was not possible due to the informal and potentially ambiguous documentation.

If CheckDesign outputs a success message, a software architect and a system engineer review and approve the design documents, in the third and fourth steps. Thus, an important requirement is that 'The design documents should be easily read and understood by humans'.

The software implementation process

Figure 8.4 shows the software implementation process in which the formal design documents and CheckSource are used. This process consists of two steps:

In the first step, a software developer implements the feature by writing source code.

In the second step, CheckSource verifies the consistency between the source code and the design documents. If the source code is inconsistent with the documents, CheckSource outputs an error message that contains information for locating and resolving the inconsistency.

An inconsistency can be resolved through one of the following scenarios:

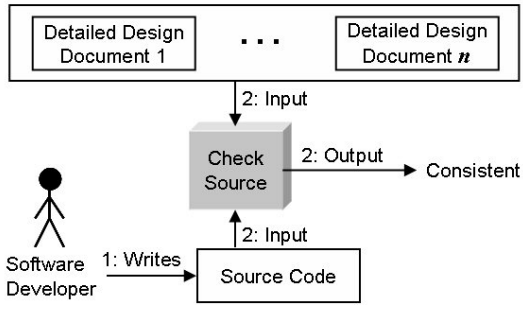


Figure 8.4: This figure shows the implementation process with formal design documents and CheckSource.

- The developer decides that the inconsistency is due to a defect in the source code, so she repairs (i.e. modifies) the source code, and then reruns CheckSource.
- The developer decides that the inconsistency is due to a defect in design documents, so she repairs the design documents and then performs the second, third, and the fourth steps of the design process (see Figure 8.3). After these steps, she reruns CheckSource.
- The developer decides that the inconsistency is due to the defects in both the design documents and the source code. So she repairs the design documents and then performs the second, third, and the fourth steps of the design process (see Figure 8.3). After these steps, she repairs the source code and reruns CheckSource.

The design and implementation processes presented above address the problems listed in Section 8.2.2.

8.3.2 Improving the software maintenance process

Whenever a developer receives a change request (or a problem report) about the implementation of an existing feature, she decides whether the change request has an impact on the detailed design. If the developer decides that there is no such impact, then she directly implements the request by following the implementation process depicted in Figure 8.4. If the developer decides that the change request has an impact on the detailed design, then she realizes the change request by following the design process depicted in Figure 8.3. Subsequently, she implements the change by following the implementation process depicted in Figure 8.4. The maintenance process explained in this section addresses the problems listed in Section 8.2.3. In Sections 8.4, 8.5, and 8.6, we respectively present VisuaL, CheckDesign, and CheckSource.

8.4 VisuaL

VisuaL is a graphical language for specifying constraints on the possible sequences of function calls from a given C function. In this section, we explain VisuaL by presenting the specification of seven example constraints, each demonstrating a distinct 'primitive' usage of the language.

8.4.1 Example 1: 'At least one'

Figure 8.5 shows a specification of the following constraint:

C1: In each possible sequence of function calls from the function `f`, there must be *at least one* call to the function `g`.

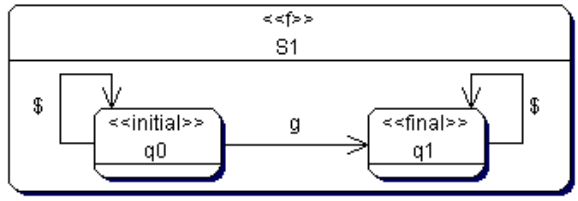


Figure 8.5: An example specification demonstrating the usage of 'At least one'.

The outer rectangle (i.e. the rectangle with the stereotype <<f>>) defines –a view on– the control-flow behavior as implemented by (the body of) function `f`. The label `S1` is the name (i.e. identifier) of the specification. The arrows represent the function calls from `f`, and the inner rectangles (e.g., the rectangle that is labeled with `q0`) represent locations within the control flow of `f`.

Inside the outer rectangle, there is a structure consisting of the arrows and the inner rectangles. We call such a structure a **pattern**.

The stereotype <<f>> means 'each possible sequence of function calls from the function `f` must be *matched by the pattern*[1], otherwise the constraint that is represented by the specification is not satisfied'.

The rectangle `q0` represents the beginning of a given sequence of function calls, because it has the stereotype <<initial>>. We call this the **initial rectangle**. There must be exactly one initial rectangle in each VisuaL specification.

The `$`-labelled arrow originating from `q0` matches each function call from the beginning of a sequence, until a call to `g` is reached. This 'until' condition is due to the existence of the `g`-labelled arrow originating from the same rectangle (i.e. `q0`).

In general, a `$`-labelled arrow matches a function call, if and only if this call cannot be matched by the other arrows *originating from the same rectangle*. In VisuaL, no two arrows originating from the same rectangle can have the same label.

[1] We precisely define this later in this section.

Note the difference between the $-labelled arrow pointing to q0 and the $-labeled arrow pointing to q1: the former arrow can match a call to any function except g, whereas the latter arrow can match a call to any function (i.e. including g), since q1 has no other outgoing arrow.

During the matching of a given sequence of function calls, if the first call to g is reached, then this call is matched by the arrow labeled with g. If there are no more calls in the sequence, then the sequence **terminates** at q1, because the last call of the sequence is matched by an arrow that points to q1.

If there are additional calls after the first call to g, then each of these calls is matched by the $-labelled arrow pointing to q1, hence the sequence eventually terminates[2] at q1.

A given sequence of function calls is **matched by a pattern**, if and only if the sequence terminates at a rectangle with the stereotype <<final>>. We call such a rectangle **final rectangle**. There can be zero or more final rectangles in a VisuaL specification.

8.4.2 Example 2: 'Immediately followed by'

Figure 8.6 shows a specification of the following constraint:

C2: In each possible sequence of function calls from f, the first call to g, if it exists, must be *immediately followed by* a call to h.

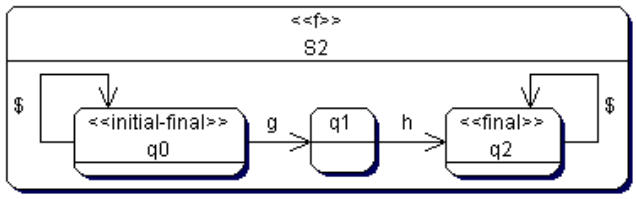


Figure 8.6: An example specification demonstrating the usage of 'Immediately followed by'.

In Figure 8.6, the stereotype <<initial-final>> means that q0 has both <<initial>> and <<final>> stereotypes.

8.4.3 Example 3: 'Each'

Figure 8.7 shows a specification of the following constraint:

C3: In each possible sequence of function calls from f, *each* call to g must be immediately followed by a call to h.

[2] Infinite sequences of function calls are out of the scope of this chapter, because VisuaL is *not* a language for specifying constraints on the execution of possibly non-terminating programs. This topic is already studied in [22].

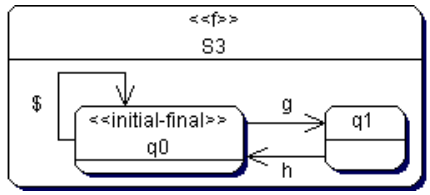


Figure 8.7: An example specification demonstrating the usage of 'each'.

8.4.4 Example 4: 'Until'

Figure 8.8 shows a specification of the following constraint:

C4: In each possible sequence of function calls from `f`, each function call must be a call to `g`, *until* a call to `h` is reached.

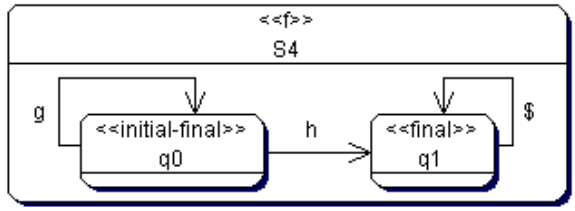


Figure 8.8: An example specification demonstrating the usage of 'until'.

8.4.5 Example 5: 'Not'

Figure 8.9 shows a specification of the following constraint:

C5: In each possible sequence of function calls from `f`, a call to `g` must *not* exist.

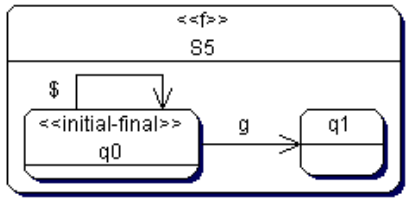


Figure 8.9: An example specification demonstrating the usage of 'not'.

Note that `q1` does not have the stereotype <<final>>, and no arrow originates from `q1`. We call such a rectangle **trap rectangle**. For a given sequence *seq* of function calls, if a call *c* in *seq* is matched by an arrow pointing to a trap rectangle *tr*, then either of the following scenarios occurs:

- *c* is the last call in *seq*. Since *tr* does not have the stereotype <<final>>, *seq* is not matched by the pattern.

- *c* is not the last call in *seq*. In this case, for each remaining call in *seq*, there is no matching arrow in the pattern. Therefore, *seq* is not matched by the pattern.

To sum up, if a sequence 'visits' a trap rectangle, then the sequence cannot be matched by the pattern.

8.4.6 Example 6: 'Or'

Figure 8.10 shows the specification of the following constraint:

C6: In each possible sequence of function calls from `f`, the first function call, if exists, must be a call to `g` *or* `h`.

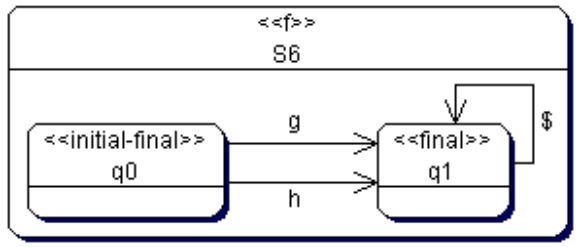


Figure 8.10: An example specification demonstrating the usage of 'or'.

8.4.7 Example 7: 'And'

Figure 8.11 shows the specification of the following constraint:

C7: In each possible sequence of function calls from `f`, there must be at least one call to `g`, *and* the first call to `g` must be immediately followed by a call to `h`.

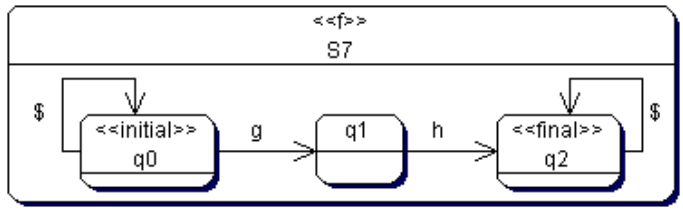


Figure 8.11: An example specification demonstrating the usage of 'and'.

8.5 CheckDesign

Using VisuaL, one can create multiple specifications each representing a different constraint on the same function. For example, each of the seven specifications in Section 8.4 represents a different constraint on the same function: `f`.

When creating multiple VisuaL specifications to express different constraints on the same function, it must be ensured that the specifications are **consistent**: There is at least one possible implementation of the function, such that the implementation satisfies each of the constraints. If there is no possible implementation of the function that satisfies all specified constraints, then the VisuaL specifications are **inconsistent**.

For example, the specifications S1 (Figure 8.5) and S5 (Figure 8.9) are *in*consistent: If an implementation of the function `f` satisfies the constraint C1 (Section 8.4.1), then this implementation cannot satisfy the constraint C5 (Section 8.4.5). Conversely, if an implementation of the function `f` satisfies C5, then this implementation cannot satisfy C1. Hence, it is impossible to implement `f`, such that the implementation satisfies both C1 and C5.

Manually finding and resolving an inconsistency among multiple VisuaL specifications is an effort-consuming and error-prone task. CheckDesign can reduce the effort and prevent the errors. CheckDesign takes a finite set of VisuaL specifications, and automatically finds out whether the specifications are consistent or not. If the specifications are not consistent, CheckDesign outputs an error message that can help in understanding and resolving the inconsistency.

8.6 CheckSource

After creating consistent VisuaL specifications, a developer typically writes source code to implement the specifications. For example, after creating the specification S2 (Figure 8.6), a developer may implement the function `f` as shown in Listing 8.1.

```
1  void f(int i)
2  {
3      g();
4      if(i)
5      {
6          h();
7      }
8  }
```

Listing 8.1: An example implementation of the function f in C.

A function and a corresponding specification may be inconsistent with each other. For example, the function shown in Listing 8.1 is inconsistent with the specification S2 (Figure 8.6): There are two possible sequences of function calls from `f`, and these sequences are $seq_1 =$<g, h> and $seq_2 =$<g>. Although seq_1 is matched by the pattern of S2, seq_2 cannot be matched by this pattern. Therefore, this implementation (Listing 8.1) is inconsistent with S2, which indicates that the implementation does not satisfy the constraint C2.

Manually finding and resolving inconsistencies between a function and its specification is an effort-consuming and error-prone task. CheckSource can reduce the effort and prevent errors. CheckSource takes a function and a corresponding VisuaL specification as the input, and finds out whether they are consistent or not. If they are consistent, then CheckSource outputs a success message, else an error message that is useful for understanding and resolving the inconsistency.

8.7 Conclusions

To conclude, we summarize the possible use cases of VisuaL, CheckDesign, and CheckSource:

- A software engineer can use VisuaL for designing the control flow of a new function to be implemented. After the engineer creates the VisuaL specifications, she can use CheckDesign for verifying that the specifications are consistent with each other.
- A software engineer can use CheckSource to automatically verify that a given function is consistent with the corresponding specifications. Whenever a specification or the function evolves, the verification can be automatically repeated.
- A software engineer can also use VisuaL for designing an additional feature of an existing function. The additional feature can be designed either within the existing specifications, or as a separate specification besides the existing ones. In either case, CheckDesign can be used for ensuring the consistency between the specifications, and CheckSource can be used for ensuring the consistency between the specifications and the implementation.
- Whenever a function evolves, a software engineer can use CheckSource for automatically detecting inconsistencies between the function and the specifications. Such an inconsistency indicates either a bug in the source code, or an outdated specification.
- A software engineer can use VisuaL to distinguish anticipated bugs from features. She can specify the 'illegal' sequences of function calls together with the 'legal' sequences of function calls. For example, any sequence that visits `q3` of S7 (Figure 8.11) is illegal, and any sequence that terminates at `q2` of S7 is legal.
- If VisuaL specifications are kept consistent both with each other and with the source code, then these specifications can be used during code inspections. If engineers suspect a bug in the function call sequences, then they can abstract away from details such as data flow, and focus on the function call sequence, by inspecting only the VisuaL specifications.

Chapter 9

Model-driven migration of supervisory machine control architectures[1]

Authors: Bas Graaf, Sven Weber, Arie van Deursen

Abstract Supervisory machine control is the high-level control in advanced manufacturing machines that is responsible for the coordination of manufacturing activities. Traditionally, the design of such control systems is based on finite state machines. An alternative, more flexible and maintainable approach is based on task-resource models. This chapter describes an approach for the migration of supervisory machine control architectures towards this alternative approach. We propose a generic migration approach based on model transformations that includes normalization of legacy architectures before their actual transformation.

9.1 Introduction

In this chapter we consider the migration of supervisory machine control (SMC) architectures towards a product-line approach that, amongst others, supports model-driven development and code generation. In practice, adopting such techniques requires architectural changes. When migrating towards a product line, such a migration needs to be applied repeatedly to migrate different product versions into product-line members. Therefore, ideally, one would like to make such a migration reproducible by automatically transforming one architecture into another. In this chapter we investigate how

[1] This chapter is based on an article in the Journal of Systems and Software [50].

this can be done using model transformation technology.

This chapter was motivated by the prototype migration of the SMC architecture of a wafer scanner. We use this wafer scanner as a running example to illustrate the migration of a legacy architecture, based on finite state machines (FSMs), to a new architecture that is based on task-resource systems (TRSs). This migration is spurred by the fact that a TRS-based SMC architecture, as opposed to an FSM-based one, is declarative, separates concerns, and supports run-time dependent decisions [81]. As a result, the maintainability and flexibility of the migrated software systems is improved.

We consider the start and end point of the migration as different architectural views [60]. We refer to these views as the source and target view respectively. In our migration approach we use the models underlying these views to consolidate and reuse as much existing design knowledge as possible. As such, we consider migration to constitute a series of model transformations, which we implemented using Model Driven Architecture[2](MDA). It should be noted that we only consider the actual migration approach; the paradigms for the migration start point and end point are prescribed by our industrial case.

In order to define a reproducible mapping and perform the migration, we define practical transformation rules in terms of patterns associated with the source and target meta models. These transformation rules are practical in the sense that they are based on an actual migration as performed manually by an expert. Based on this migration, we have formulated generic, concern-based transformation rules. These rules are defined using a model transformation language making our approach automated. Due to practical reasons, which are mainly associated with the informal use of modeling languages in industry [47, 70], we first normalize the legacy models before applying our model transformations.

Although we focus on the migration of the SMC architecture of a particular manufacturing system, a wafer scanner, the contributions of this chapter are applicable to similar (paradigm) migrations of supervisory control components in general. The presented industrial results serve as a proof a concept, additional migrations have to be performed before the results can be properly quantified. The experiences as outlined in this chapter are, to a lesser extent, relevant for all software architecture migrations that can be seen as model transformation problems. Note that this chapter merely gives an overview of the work done. See Graaf et al. [50] for additional details.

9.2 Migration context

The machine control context is clarified in Figure 9.1. From a supervisory perspective, (sub)frames, transducers and associated regulative controllers form mechatronic subsystems that execute manufacturing activities to add value to products. The recipe- and customer-dependent routing of multi-product flows, with varying optimization criteria, constitutes one of the key (supervisory) control issues. Moreover, advanced manu-

[2] http://www.omg.org/mda (June 2007).

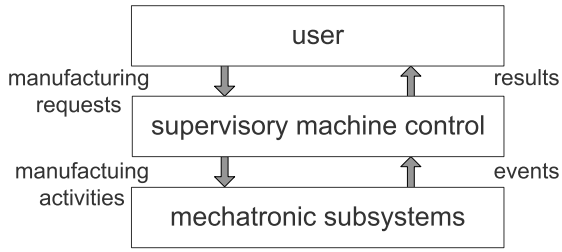


Figure 9.1: Machine control context.

facturing machines must respond correctly and reproducibly to manufacturing requests, run-time events and results. Consequently, to interpret manufacturing requests and to ensure feasible machine behavior, a supervisory machine control component is required to coordinate the execution of manufacturing activities [90, 93, 81].

In practice, a high-level manufacturing request is translated into valid low-level machine behavior using multiple, consecutive control-layers. This is supported by recursive application of the control context from Figure 9.1: manufacturing activities of one level become manufacturing requests for the next level until the level of the mechatronic subsystems.

In this chapter we consider a wafer scanner as a representative example of an advanced manufacturing machine. Wafer scanners are used in the semiconductor industry and perform the most critical step in the manufacturing process of integrated circuits. We use a request that can be handled by one of the ASML wafer scanner SMC components as a running example. Execution of the 'unload wafer' request results in a wafer to be removed from one of the wafer stages by the (un)load robot.

In advanced manufacturing machines, multiple manufacturing activities - and sequences hereof - may fulfil a particular request and, in turn, multiple mechatronic subsystems may be available to perform a particular activity. That is, multiple alternatives exist that require the selection of a specific subset of both manufacturing activities and mechatronic subsystems to fulfil a given manufacturing request. As a result the following concerns play an import role in the design of the SMC system of a wafer scanner:

Setups The execution of a manufacturing activity may leave a subsystem in a state that is not compatible with a subsequent activity. Then, an additional subsystem state transition, a setup, has to be performed.

Resource Usage For some manufacturing activities subsystems have to be claimed exclusively. Afterwards, they have to be released again.

Concurrent Execution Independent activities have to be performed in parallel to improve performance.

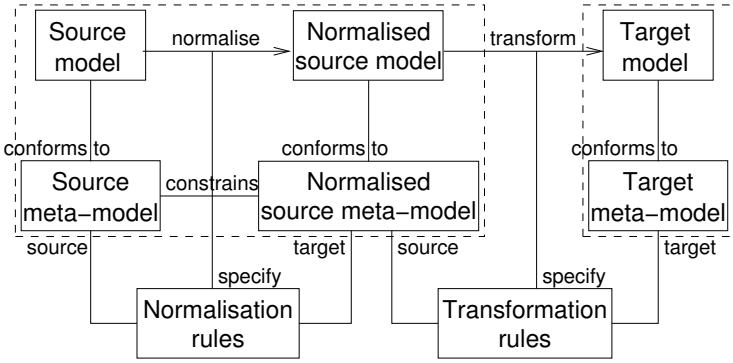


Figure 9.2: Generic two-phased migration approach.

Synchronous Execution Synchronization is required between subsequent (dependent) activities. Additionally, subsystem state transitions may require synchronization as well (e.g., to prevent collisions of physically moving subsystems).

Conditional Execution Depending on certain machine conditions different execution paths might need to be activated.

During migration, sequence-dependent setups, subsystem usage, concurrent execution, synchronous execution and conditional execution are concerns that need to be addressed. To this end, we defined concern-based transformation rules that map these concerns from the legacy to the new architecture.

9.3 Model-driven migration

We propose the migration approach as shown in Figure 9.2. It uses a two-step process that includes a normalization and transformation step.

Because of tool limitations and the generally informal use of modeling paradigms and languages in industry [47, 70] a multitude of models becomes conceivable that all have the same intended meaning. This makes directly translating a source model into a target model inherently difficult. As such, we introduce an intermediate normalization step that uses a set of normalization rules to obtain a normalized source model. The normalization rules are defined as mappings from the source meta model to the normalized source meta model. Next, a set of transformation rules can be applied to transform a normalized source model into the target model. These transformation rules are defined as mappings from the normalized source meta model to the target meta model.

Although the approach is generic, our industrial case imposes some practical restrictions on the enabling technologies. Spurred by the fact that the existing architecture documentation contained source models (partly) in Unified Modeling Lan-

guage[3](UML) state charts, we decided to implement the different steps of our migration approach using MDA technologies.

9.4 Migration source

We consider FSMs as the given starting point for the migration. Using FSMs, the set of possible machine behaviors is considered to form a language. A discrete supervisory FSM is synthesized that restricts this language by disabling a subset of events to enforce valid machine behavior. This requires the behavior in all possible states for all requests to be specified explicitly using (conditional) state transitions with associated triggers (events), and effects or state actions (manufacturing activities). When using this paradigm, concurrent execution is the result of independent parts of concurrently executing state machines that can optionally share events to synchronize. Consequently, multiple FSMs are used per controller (typically one for each type of request).

Our source models are specified using UML state chart diagrams, for which the UML specification provides a meta model and a set of well-formed-ness rules, specified in Object Constraint Language[4](OCL). Using this meta model, UML state machines can be constructed that model behavior as a traversal of a graph of state nodes interconnected by transition arcs. For a detailed description of the semantics of the various elements in this meta model we refer the reader to the UML specification [84].

As an example of how this meta model is used in practice, consider the state machine in Figure 9.3, which correspond to the unload wafer request. Such state machines are the source models for the migration. Figure 9.3 illustrates the use of two distinct resource usage patterns for WS (wafer stage) and UR (unload robot) in the unload wafer request: for WS only an available Event (WS available) and release Action (release WS) are specified, for UR also a claim Action (claim UR) has been specified. Furthermore, observe that after the actual transfer of the wafer (TRANSFER_FINISHED) the alternative completion sequences of subsequent activities, which are associated with the UR_moved and WS_moved events, are specified exhaustively.

Even from our example request it becomes clear that, in practice, concerns are addressed using a multitude of idioms and constructs. This is the main reason for the introduction of our normalization step.

9.5 Normalization rules

Normalized source models have to comply to a set of well-formed-ness rules. Most importantly, concerns have to be specified in a uniform way. We have defined standardized idioms for the concerns identified in Section 9.2. We introduce these idioms by the example of Figure 9.4. Normalization involves modifying source models to remove any violation of these well-formed-ness rules. Note that, due to the diversity of

[3] http://www.uml.org (June 2007).

[4] http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL (June 2007).

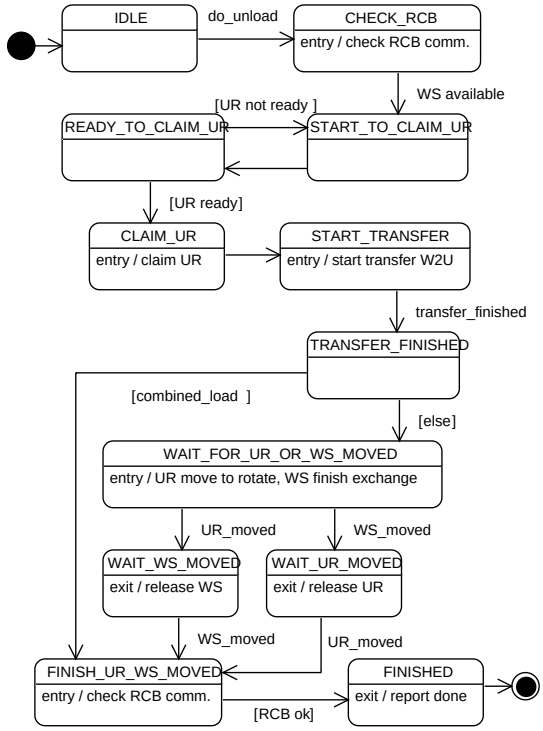


Figure 9.3: Unload wafer request.

the idioms used in the source models, normalization is performed manually in our case study.

Table 9.1 lists the stereotypes that we define as part of the SMC profile. Next to stereotypes, the profile also defines a number of constraints. Listing 9.1 lists one of these constraints, specified in OCL as an invariant over the UML meta model (*C1*).

```
-- C1: state entry actions are actions that execute manufacturing
  activities ($\text{i.e.}$, without stereotype)
context State inv:
   entry.stereotype->isEmpty
```

Listing 9.1: Well-formedness rule of the SMC profile, in OCL.

Stereotype	baseClass	description
≪wait≫	State	wait for resource state
≪claim≫	Action	claim resource action
≪release≫	Action	release resources action
≪available≫	Guard	resource available guard
≪available≫	Event	resource becomes available event

Table 9.1: SMC profile stereotypes.

Intuitively, the normalization is context dependent and requires (some) domain knowledge. Moreover, the normalization rules not only depend on the specific source paradigm but also on the modeling conventions as encountered in the specific (industrial) migration context. Therefore, we illustrate the normalization step by defining the used context-specific normalization rules for our case study.

Subsystem setups In the source model, subsystem state consistency is ensured by specifying setup transitions for every possible subsystem state at design-time. In practice, this is not done exhaustively. Instead, domain-knowledge is used to limit the number of setup related alternative transitions. Although subsystem setups can be performed automatically using the TRS paradigm and, thus, do not need to be specified explicitly, we do preserve them during the normalization step. This in fact ensures that the migrated control system mimics the behavior of the legacy control system exactly. When reconsidering Figure 9.3 and 9.4, the move to rotate Action is in fact a resource setup.

Subsystem usage The pattern to address the 'subsystem usage' concern is best understood from one of the orthogonal regions in the composite state in Figure 9.4. Before a manufacturing activity (e.g., finish exchange) that requires a certain subsystem (WS) is executed, a choice pseudo-state is entered. Then, if the required resource is available ([WS available]), it is claimed (claim WS) by the transition towards the state in which the manufacturing activity is executed (FINISH). Otherwise, a state (WAIT_FOR_WS) is entered that is only left when an event occurs indicating the resource has become available (WS available). The resource is claimed (claim WS) on the transition triggered by that event. Once the manufacturing activity is performed, claimed resources are released again by a release action that is executed when exiting the state (release). This pattern can easily be generalized.

We use the stereotypes defined by the SMC profile (Table 9.1) to distinguish between Actions, Guards, Events, and States related to the use of subsystems and those related to the execution of manufacturing activities (to which no stereotypes are applied). normalization introduces stereotypes for specific model elements that are related to the subsystem usage concern. Furthermore, from Figure 9.3, and its normalized counterpart in Figure 9.4, it can be seen that additional model elements are introduced to complete the pattern described above.

Synchronous execution Synchronization between subsequent manufacturing activities in the source models is simply achieved by their order in the state machine. Furthermore, synchronization between subsystem state transitions is not modeled at this level. As such, no specific idiom is used to specify this concern.

Concurrent execution In the original source models, concurrency was often modelled using States, including Actions that *start* two or more manufacturing activities and separate transition paths for all possible completion sequences, which are enabled by completion Events. As an example, consider state WAIT_FOR_UR_OR_WS_MOVED in Figure 9.3 in which two manufacturing activities are started. The possible completion sequences are specified exhaustively by replication of the associated completion events (UR_MOVED and WS_MOVED). Because those events can only be associated with their corresponding manufacturing activities using naming conventions, such an approach complicates the determination of the scope of concurrent execution. Therefore, we require that concurrency is modeled using a concurrent CompositeState containing (orthogonal) regions. This implies that during normalization, manufacturing activities are mapped to CompositeStates when they are started in a single State node and alternative completion sequences are specified exhaustively. Figure 9.4 shows the normalized version of this concurrency idiom, where the two resource usage patterns are executed in parallel.

Conditional execution The idiom for conditional execution is more complicated. First, we require it to be specified using a choice Pseudostate with two outgoing Transitions. One specifies some condition as a Guard; the other specifies [else] as a Guard. Furthermore, we require 'proper' nesting of conditional activation paths in a state machine. This means that we require pairs of corresponding, alternative paths through the state machine to be merged one at a time (using junction Pseudostates), and in reverse order.

Without this requirement for proper nesting, finding the set of States, and thus Actions, which are enabled when some Guard evaluates to true would become rather complicated.

9.6 Target meta model

We consider TRS as the given paradigm for the end-point of the migration. This endpoint is based on a research prototype [81]. Using the TRS paradigm, a manufacturing request is translated into valid machine behavior in two phases. First, upon arrival of a manufacturing request, a scheduling problem in the context of that request is instantiated during a planning phase. For this, the request is interpreted through rules that operate on capabilities (resource types) and behaviors (task types). Here, a manufacturing activity corresponds to a task and a mechatronic subsystem to a resource. The first phase results in an hierarchical digraph that consists of tasks and their (precedence)

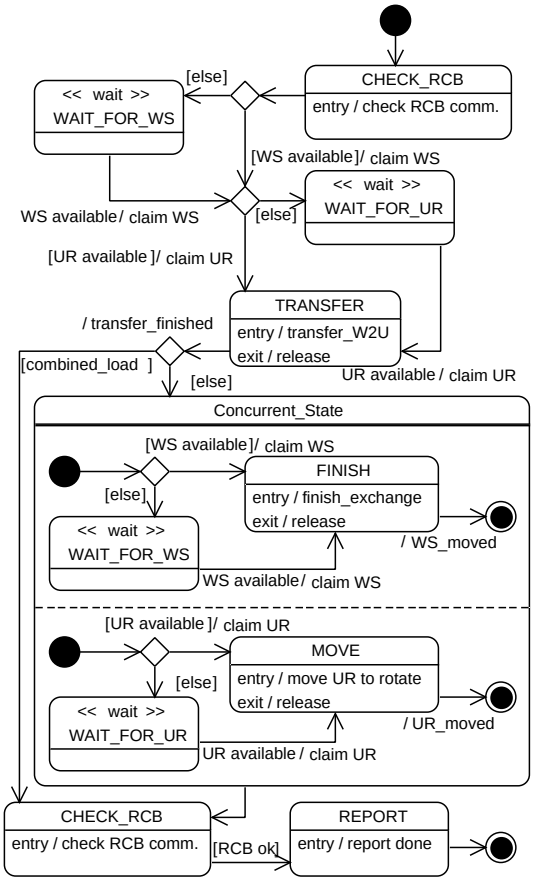


Figure 9.4: Normalized unload wafer request.

relations. Nodes in this graph can be composite to either denote a set of tasks that all need to be executed or to denote a set of tasks of which only one will be executed based on some condition. Second, a scheduling phase constructively assigns tasks in this digraph to specific resources over time [99, 81]. This results in a fully timed, coordinated TRS that can be dispatched for execution.

The end-point for our migration is a product-line architecture, in which the decisional responsibilities are assigned to three generic and reusable components: Planner, Scheduler, and Dispatcher. This product-line architecture offers variability with respect to tasks and resources and can be instantiated for a specific controller by implementing two application specific modules that define the specific system under control and implement the interfacing with lower-level components.

In order to define our target models, we introduce a governing target meta model as depicted in Figure 9.5. There, SystemDefinition serves as a root element. This system definition consists of a static and dynamic part. The static part defines the available Behaviours, Resources and Capabilities of the system under control. These are used to model types of manufacturing activities, subsystems, and types of subsystems. In addition, to address the subsystem usage concern, it defines which capabilities are required by which behavior. Furthermore, the corresponding beginState and endState are specified in CapabilityUsage. These states are, for instance, used to determine sequence dependent setups.

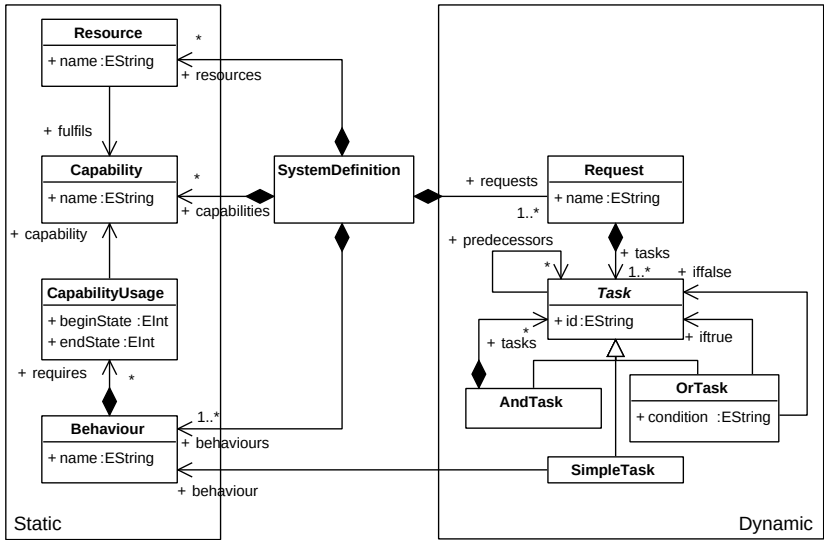


Figure 9.5: Target meta model.

The dynamic part of Figure 9.5 represents the rules for uniquely mapping a manufacturing Request to SimpleTasks, which are of a specific Behaviour, and assigning Resources that fulfil a required Capability. Every Task includes a set of (direct) predecessors, that is, other Tasks that need to be executed before it can be dispatched. This relation is used to (dis)allow concurrency and imply synchronization; in principle all tasks are executed in parallel, unless prevented by the predecessor relation. Conditional execution can be specified using OrTasks, that contain two Tasks (iftrue and iffalse) that may be composite. The evaluation of its condition determines which one will be dispatched. Finally, to cluster Tasks that *all* need to be performed, an AndTask can be used.

9.7 Transformation

The transformation rules are defined as mappings from a normalized source meta model (i.e., our UML profile) to a TRS meta model. For their definition we used the following strategy. First, we indicate how elements in the normalized source meta model are related to the primary elements of the target meta model. Second, for each of the identified SMC concerns we define and tailor transformation rules to relate the corresponding patterns in the normalized source model and the target model. These rules are described reasoning backwards, meaning that for each of the elements of the target meta model we explain for what source model patterns they will be created.

We defined all transformations in the Atlas Transformation Language [61](ATL). Its transformation engine can be used in combination with MetaObject Facility[5](MOF)-based models and meta models serialized with XML Metadata Interchange[6]. As our source meta model we used the MOF-UML meta model available from the Object Management Group[7](OMG) [84]. To create source models, we can simply use a UML modeling tool that supports XMI export. For the target meta model we used Eclipse Modeling Framework[8] (EMF).

9.7.1 Basic Target Model Elements

SimpleTask and Behaviour SimpleTasks correspond to manufacturing activities, and Behaviours correspond to *types* of manufacturing activities in SMC systems. Therefore, to create SimpleTasks and Behaviours in the target model we need to identify Actions corresponding to manufacturing activities in the source model.

Our UML SMC profile specifies that an Action that corresponds to a manufacturing activity has no stereotype and is executed as a State entry Action (see `C1` in Listing 9.1). For every such Action, a SimpleTask needs to be created in the target model.

Resource and Capability To create Resources and Capabilities we need to identify mechatronic subsystems in the source models. However, in the FSM paradigm, mechatronic subsystems are not modeled explicitly. Hence, the source model does not contain elements that directly correspond to Resources and Capabilities. We can, however, take advantage of the fact that in the FSM paradigm, subsystems are explicitly claimed. We create Resources in the target model based on Actions that claim a specific subsystem, that is, Actions to which the ≪claim≫ stereotype has been applied. Furthermore, for every resource we simply create a separate Capability (Resource type).

SystemDefinition and Request The SystemDefinition root element in a target model contains all required elements that define the domain specific part of an SMC controller.

[5] http://www.omg.org/mof (June 2007).
[6] http://www.omg.org/mda/specs.htm#XMI (June 2007).
[7] http://www.omg.org (June 2007).
[8] http://www.eclipse.org/emf (June 2007).

As such, this element corresponds to a complete source model.

A Request encompasses rules that determine how that particular manufacturing request, such as our unload wafer from Figure 9.3, is planned. Planning rules involve a set of Tasks and corresponding predecessor relations. Additionally, a Task can be an AndTask or an OrTask. In the source model, a complete state machine is used to specify how a manufacturing request is to be executed. So, we create a Request element in the target model for every StateMachine in the source model.

9.7.2 Concern-Based Transformation Rules

Resource usage To address the resource usage concern we need to relate Behaviours to the Resources and Capabilities (resource types) they require. In the target meta model, CapabilityUsage elements are used to this end. However, we cannot derive the CapabilityUsage elements in the target model directly, since our source models only contain dynamic information. Consequently, we will have to derive them indirectly instead.

For each subsystem usage pattern, as described in Section 9.5 we conclude that the subsystems claimed at that point are required for the corresponding manufacturing activity. These are all the subsystems that are claimed after the previous release action. In the target model, CapabilityUsage elements are then defined connecting the corresponding Behaviour and Capabilities. For our unload wafer request, for instance, this results in the definition of a CapabilityUsage element relating the transfer W2U behavior to the WS capability.

Resource setups In the target model, setups are automatically inserted by the generic (solving) part of the product-line architecture. This is done at run-time, based on mismatching beginState and endState attributes of the CapabilityUsage element. To some extent, these could be derived from the explicitly specified setups in the source model.

Synchronous execution The target model defines precedence relations between those Tasks that require synchronization (within the same Request). In principle, these relations follow from the execution order of the manufacturing activities and the corresponding Actions within a normalized state machine. In addition, (virtual) resources can be used for external synchronization.

For synchronization within a Request, predecessor relations are created for every task by searching for its set of (direct) predecessor tasks.

Concurrent execution The normalized pattern for concurrency, as discussed in Section 9.5, is a CompositeState with orthogonal regions. To address the concurrent execution concern we need to identify instances of such patterns in the source model.

We defined a transformation rule that creates an AndTask for every concurrent CompositeState in the source model except for the top CompositeState of the StateMachine. Basically, the predecessors relation is the mechanism used in the target model

to (dis)allow concurrency: if two tasks are not related by the transitive closure of the predecessors relation, they can execute concurrently. Now, these potentially concurrent tasks are executed as soon as execution of their predecessors has finished and the required resources are available. In turn, this also implies that a task can have multiple (concurrent) predecessors.

Conditional Execution As discussed in Section 9.5, the normalized source model uses a state with two outgoing guarded transitions to specify conditional execution. Every two alternative conditional branches in a source model are mapped to an OrTask in the target model. This OrTask contains two subtasks (iftrue and iffalse), which may be composite and represent the two conditionally executed branches following a State with two outgoing guarded Transitions. Subsequently, for the creation of those subtasks, we need to find all model elements that map to a task in each of the branches.

9.7.3 Transformation Results

In total, we needed approximately 300 lines of ATL code to implement all the necessary transformation rules and helpers for the transformation step of our migration approach. Once the source model, source meta model, target meta model, and transformation module are defined and located, the ATL transformation engine generates the target model (e.g., a system definition) in its serialized form. We used a UML activity diagram to visualize the dynamic part of the target model in Figure 9.6.

Note that, we merely use UML notation to *represent* part of the task resource model. As such, the semantics are not identical to that of UML activity graphs, but only similar. We represent Tasks as Activities stereotyped with the resource they require. The transition represent predecessor relationships (in reverse direction). For AndTasks we use fork Pseudostates (represented by the horizontal black bar). A complete AndTask is thus represented by the subgraph that starts with a fork and ends when the two concurrent paths are joined. OrTasks are represented using choice Pseudostates (represented by a diamond with two outgoing arrows). Similar to the AndTask a complete OrTask is thus represented by the subgraph that starts with a choice Pseudostate and ends when the two conditional paths are joined. For convenience we did not explicitly represented the join of the two concurrent paths (i.e., using another horizontal bar); they are joined in the same node (the diamond with three incoming arrows) as the conditional paths.

9.8 Evaluation

Scalability With respect to the scalability of our approach we can safely state that our experiments are of the same order of magnitude as full-fledged component migrations for real-world wafer scanner applications. More concretely, the two requests that were migrated as a proof of concept account for approximately 10-20% of the source code for our SMC components. The application of our transformation rules to the two representative examples presented in this chapter requires less than 10 seconds to complete

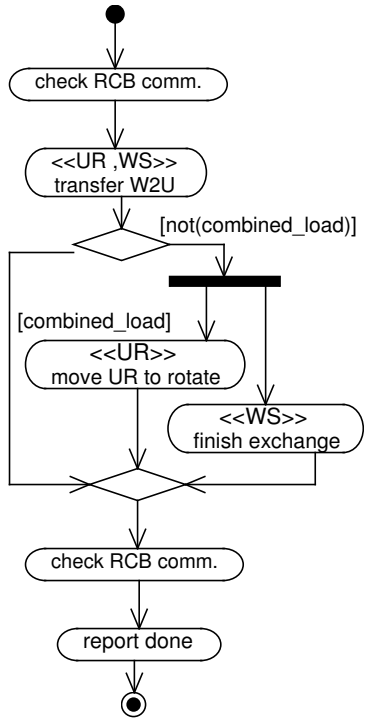


Figure 9.6: Results for unload wafer request.

on a 1.7 GHz notebook. Furthermore, we expect the execution time to be linear with respect to the number of requests. More important for the execution time is the nesting depth of conditional paths. For our industrial case we have not encountered requests with deeper nesting than our example requests.

Effectiveness Our model-driven approach requires that implicit design decisions and design knowledge is consolidated and made explicit for the definition of meta models and transformation rules. As such, the application of our approach to the SMC components of our case study increased the general understanding of concerns and the associated implications (and difficulties) surrounding the architecture migration of SMC systems. Moreover, the need for experts on both the domain and the target paradigm was confined to the definition of the normalization and transformation rules.

The effectiveness of both the MDA approach and our model-driven migration approach depends partially on the ability of modeling, transformation and code generation tools to cooperate. As such, standards involved with the MDA, such as MOF, UML, and particularly XMI, play an important role. In practice, the availability of different versions of these specifications made it difficult to setup an appropriate tool

chain. For instance, we could not use the latest version of our UML modeling tool (i.e., 'Poseidon for UML') because the UML meta model it uses, was incompatible with the ATL transformation engine. Although we took the liberty of selecting tools that were able to cooperate, we still needed to implement some additional transformations using Extensible Stylesheet Language Transformations[9](XSLT) to overcome some incompatibilities between the various tools. In industry it will not always be possible to select a specific set of tools for the migration given practical considerations such as licensing, support, and training costs.

Apart from tool support, the required human intervention during the normalization step also determines the effectiveness of our migration approach. The complexity of the normalization step depends on the number of constraints that the restricted source meta model adds to the legacy source meta model (if present). Here, a trade-off applies: fewer constraints make the transformation, which is typically automated, more complex because more specification alternatives have to be covered. For instance, if we would allow Actions corresponding to manufacturing activities to occur as Actions on Transitions, searching for predecessors would become much more complicated. On the other hand, the normalization step requires less effort in that case.

Extensibility Currently, our transformation rules do not handle synchronization across different requests. This could prove to be a limitation for the large scale application of our transformation rules. To this end, we would have to (at least) extend our profile to include a special type of Event to denote external events for such inter-request dependencies.

The overall extensibility of our migration approach is demonstrated by using source models with two distinct origins for our experiments. In the case of the unload wafer request we used the available architecture documentation of the involved SMC component. This documentation contained UML state chart diagrams for the component's requests, including our example request.

The 'back-end' of our approach can be extended as well by steps that further process the result of our model transformations. We already mentioned the generation of documentation. Another possible extension is the generation of source code to actually generate the System Definition module of the product-line architecture. Both can be specified using model transformations.

Note that we did not yet consider the domain specific interface modules of the product-line architecture. However, this only constitutes a minor hurdle since we can simply encapsulate the existing source code bodies for each behavior (preserving interface functionality and behavior).

[9] http://www.w3.org/TR/xslt (June 2007).

9.9 Conclusions

In this chapter we formulated the migration of supervisory machine control (SMC) systems as a model transformation problem. The starting point is an SMC architecture based on finite state machines (FSMs); the end point is a product-line SMC architecture based on task-resource systems (TRSs). Our approach supports the generic migration of the product-line members.

We demonstrated that the development framework for the Model Driven Architecture[10](MDA) can be successfully applied in a migration context as well: migration can be seen as a series of model transformations. We proposed a generic two-phased, model-driven migration approach that uses distinct normalization and transformation steps to derive the modules required to instantiate the TRS product-line architecture for a particular (sub)system. The normalization step is crucial in overcoming semi-formal, incomplete and ambiguous specifications as well as tool and language limitations. This normalization step requires domain knowledge and manual effort, but makes our approach suited for industrial application.

The industrial case that motivated this chapter imposes not only the source and target paradigms but places practical constraints on the enabling technologies as well. Starting from Unified Modeling Language[11](UML), we selected technologies compatible with the MDA to setup a convenient tool-chain that supports the definition and manipulation of models. Using this tool chain, several requests from different SMC components have been migrated as a proof of concept. The experiences we gained from this exercise indicate that the application of model transformations not only increases the understandability of such a migration, but also reduces the need for domain experts.

[10] http://www.omg.org/mda (June 2007).
[11] http://www.uml.org (June 2007).

Chapter 10

Industrial impact, lessons learned and conclusions

Authors: The Ideals research team

10.1 Introduction

The Ideals project is an applied research project that is carried out in an industry-as-laboratory setting, with ASML as the carrying industrial partner. The industry-as-laboratory setting provides a realistic industrial environment in which research ideas and theories can be validated and tested. Methods and techniques that are developed in the Ideals project have been described in the previous chapters and an overview of the results is found in Chapter 1. Their development has resulted in several papers and articles in scientific proceedings and journals, Ph.D. theses, presentations at scientific workshop and conferences and tutorials. A list of the scientific papers and articles is found in Appendix A. This concluding chapter focuses on the industrial relevance of the research results. The industrial relevance of new methods and techniques is demonstrated by a proof-of-concept showing that the principles work, and could be used by industry. Several transfer projects were initiated for those research areas that demonstrated their industrial maturity by means of a successful proof-of-concept.

This Chapter is organized as follows. In Section 10.2 we give an overview of the maturity phases of an industry-as-laboratory project. Based on these phases we discuss the industrial impact of the methods and techniques that were developed in the project in Sections 10.3–10.6. Some general lessons learned when guiding researchers through the phases of industrial evidence are given in Section 10.7. Final conclusions are given in Section 10.8.

10.2 Phases of industrial evidence

Different levels of proof-of-concept, and different phases in demonstrating evidence and transferring knowledge can be recognized. The Embedded Systems Institute distinguishes between six phases, which range from pure academic research to full operational use in industry. In the successive phases, the scale and level of reality of the proof-of-concept case studies increases, while the involvement of the industrial partner grows from none to complete involvement in the operational phase; see Figure 10.1.

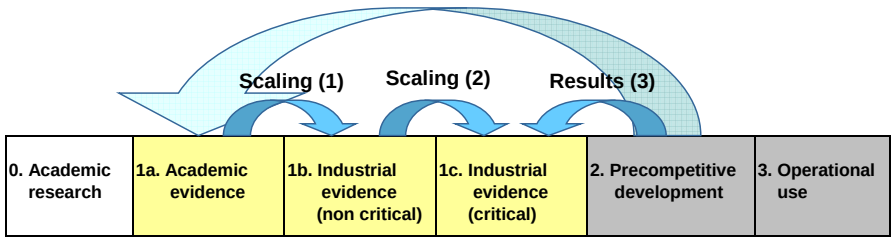


Figure 10.1: The transfer process.

Phase 0: Academic research Universities and research institutes perform research in a more or less autonomous setting. This research may be triggered by industrial problems, but more often it is just curiosity driven academic research. Industry-as-laboratory projects of the Embedded Systems Institute are not directly involved here, nor are transfer projects. This research may lead to future industrial relevance.

Phase 1a: Academic evidence Academic evidence refers to a proof-of-concept of newly developed methods, techniques and tools in an academic environment. Emphasis in this phase is on making the theory work, and showing that it may potentially solve an industrial problem. The methods and tools are applied to an artificial academic problem, or to a simplified industrial problem, in a well-defined and controlled environment. Aspects like scalability of the method, or usability of the tools, do not play a role yet. Typically, some evidence of this kind is a starting point for an industry-as-laboratory project like Ideals.

Phase 1b: Industrial evidence, non critical A realistic problem of the industrial partner with limited size and complexity is the starting point for showing industrial evidence. Project members and researchers are involved; the industrial partner may actively cooperate but his main task is to state the problem, provide a case for this problem, and provide the necessary domain knowledge so that the project members get a clear view on the problem and the solution area. The case study should show the proof-of-concept, and demonstrate the feasibility and benefits,

when possible but not necessarily quantitatively, of the newly developed techniques and tools in a real industrial case. This case should be non-critical, in the sense that the daily business of the industrial partner may in no way depend on it. In this phase the case may concern an old problem for which a solution has already been developed with other means.

Phase 1c: Industrial evidence, critical The newly developed techniques and tools are used in pilot projects in the real industrial context, where the outcome does matter. The focus is on scalability, embedding in, and impact on existing processes, and demonstrating quantitative evidence of their usefulness. Issues and aspects that do not directly relate to the main functionality of the new techniques and tools, such as usability, performance, reliability, availability of documentation and manuals, help desk, and training, are getting more important. Also the importance of knowledge consolidation and transfer increases. Since many of these issues are important for, and in addition specific to the industrial partner, whereas they are less interesting from a research perspective, the involvement of the industrial partner increases.

To organize the aspects mentioned above, a *transfer project* is initiated for each successful research area, which has clearly shown industrial evidence of benefits. The aim of a transfer project is, as the name suggests, to transfer knowledge, methodology, and tools from the research project to the industrial partner. Since this goes beyond the proof-of-concept goal of the research project, and since the major part of the manpower for this activity is provided by the industrial partner, transfer projects are decoupled from the research project.

Phase 2: Precompetitive development The main target of this phase is to prepare the methodology, techniques and tools so that they can be institutionalized within the industrial partner. This involves a seamless continuation of the activities of the transfer project, but with much less involvement of the research-project members; they may give support for specific requests, but they are generally not involved anymore. The pilot projects are gradually taken over by real users, adapting and deploying the methodology and supporting tools in their daily development activities. The activities during this phase include re-factoring of tools, documentation, user training, and configuration management, making the methodology and supporting tooling ready to be rolled out in the whole organization.

Phase 3: Operational use The full roll out of the methodology, techniques and tooling into the organization of the industrial partner takes place in this phase. Parallel to the daily usage of the techniques and tools, the support aspects like training, knowledge consolidation, tool maintenance and support, and configuration management are institutionalized within the industrial partner. For the research project there is no role anymore in this phase.

Ideally, a research topic passes through these six phases from academic research to operational use in industry. But typically, there are several feedback loops. The results in any phase can trigger new research questions, and also new problems may trigger new research subjects. Of all these new research ideas, many never make it to operational use.

10.3 Industrial impact - TRAPPEL

The goal of TRAPPEL, the Transfer Project for Error Linking, is to leverage results of research on the improvement of ASML error linking — results that were discussed in Chapter 3 of this book. Faults in the error linking implementation have a negative impact on system availability. This project will consist of the development of BELL (Better Error Linking and Logging), a tool that is capable of detecting error linking faults in source code. Furthermore, TRAPPEL will introduce BELL in a pilot development project, and provide the necessary training. The intended use of BELL is similar and complementary to the use of the QA/C tool. Code that is synchronized with the code repository is checked by BELL, and error reports are generated that have to be processed by developers.

Motivation

Error linking (as defined in Chapter 14 of ASML's Software Architecture User Manual) is currently being implemented by hand, and without automatic checking. This practice has resulted in a relatively high number of faults in the error linking implementation. Research in the Ideals project has shown an average fault rate of 2.1 faults per 1 KLOC (229 KLOC examined). Table 10.1 gives an overview of those results, which were also discussed in detail in Chapter 3.

Faults in the error linking implementation have a negative impact on system availability. A typical error linking fault breaks the chain of errors that starts at a root error and is propagated upwards through the system. Diagnosing system failure becomes more time consuming because of these broken links, since they prevent the identification of a root error.

Faults in the error linking implementation can be detected by an automatic tool. Introducing such a tool in the build process (similar and complementary to the QA/C tool) will alert developers of the presence of error linking faults. By requiring that the detected faults are fixed by developers the fault rate for error linking can be reduced.

We have therefore proposed that an automatic checking tool for error linking is included in the ASML build process. This tool can be based on a prototype checking tool for error linking that has already been developed by the Ideals project (the SMELL tool also discussed in Chapter 3).

Scope

TRAPPEL will consist of the development by ASML of an automatic tool, called BELL, for the detection of faults in the implementation of error linking. The tool will process C sources as they are checked into the source repository by a developer. Each error linking fault will result in a report that the developer can read and use to fix the fault.

The tool will be introduced in a pilot project that is developing (new) software. This pilot project will use BELL during development to find and repair error linking faults that BELL discovers. Members of the pilot project provide feedback to the TRAPPEL tool developers.

Specifically out of scope for TRAPPEL is the adoption of BELL outside of the pilot project and the complete integration into the ASML development process. This will be part of a follow-up project, for which a plan will need to be developed by ASML in case such further adoption is desired. It was however advised that ASML already considers these issues, in parallel to the TRAPPEL project, so that appropriate action can be taken once TRAPPEL is delivered.

Status

TRAPPEL has passed phase 1b (non-critical industrial evidence) of maturity, and is currently being moved into phase 1c (critical industrial evidence).

The State Machine for Error Linking and Logging (SMELL) tool has been developed by researchers from CWI, Delft University of Technology, and Eindhoven University of Technology within the context of the Ideals project. It is a proof-of-concept tool capable of detecting various faults in the implementation of the error linking idiom as used by ASML today. The tool has been applied to various ASML components, and the results obtained confirm that building such a tool is both feasible and useful for ASML. As SMELL is only a proof-of-concept tool, it currently has a number of limitations, i.e. situations in which it does not know what to do precisely. These limitations are however not inherent to the techniques used, but are primarily technical in nature and can be attacked by the new tool developed by this project.

Table 10.1 presents the results of applying SMELL to five relatively small ASML components. The first column lists the component that was considered together with its size, Column 2 lists the number of faults reported by SMELL, Column 3 contains the number of false positives we manually identified among the reported faults, Column 4 shows the number of SMELL limitations that are encountered and automatically recognized, and finally Column 5 contains the number of validated faults, or ‘true’ faults. The validation of faults was performed jointly by researchers and ASML developers.

Overall, we get 236 reported faults, of which 45 (19 %) are reported by SMELL as a limitation. The remaining 191 faults were inspected manually by both researchers and ASML developers, and we identified 37 false positives (16 % of reported faults). Of the remaining 154 faults, 141 are unique, and so in other words, we found 2.1 true faults per thousand lines of code.

	reported	false positives	limitations	validated
CC1 (3 kLoC)	32	2	4	26 (13)
CC2 (19 kLoC)	72	20	22	30
CC3 (15 kLoC)	16	0	3	13
CC4 (14.5 kLoC)	107	14	13	80
CC5 (15 kLoC)	9	1	3	5
total (66.5 kLoC)	236	37	45	154 (141)

Table 10.1: Reported number of faults by SMELL for five ASML components.

Outlook

The infrastructure developed in the context of the TRAPPEL project is not only useful within this project only, but can be useful for follow-up projects and new research activities as well. First, BELL can be extended in order to check many more coding conventions and idioms, such as tracing, parameter checking, memory handling and state updates. Second, the detailed code analysis performed by BELL is particularly useful for idiom migration, where the current way of working is improved by introducing aspect-oriented solutions for the idioms mentioned above.

10.4 Industrial impact - WeaveC

The problem of crosscutting concerns in software development has been recognized for a long time, and made explicit in e.g. [66, 100]. Also, the approach to provide explicit support in programming languages, as well as other development methods and artifacts has been explored in research [37]. In other words, the AOSD community had already passed through most of the maturity phases of the industry-as-laboratory approach. However, one of the phases that is still under-developed is the evidence of industrial application in critical projects (phase 1c). Another thing lacking in state-of-the-art AOP technology -at least at the start of the ideals project- were industrial quality AOP tools for the C programming language. This situation is illustrated by Figure 10.2, in the row with label ‘general AOP technology’. This figure shows the phases of maturity of research results in an industrial context.

At the start of the Ideals project, the notion of crosscutting concerns in ASML source code had already been identified, and it was known that aspect-oriented technology could potentially address this. Hence one of the first activities was to investigate whether AOP technology was useful in a context of complex embedded systems, typically based on the C programming language. This required going through most phases of the maturity model again, but this time focused on the particular issues in such an environment:

> **Phase 0:** Investigate whether AOP techniques are applicable in a C context, and if so, how to deal with the specifics of C (and other procedural languages).

Phase 1a: Illustrate how AOP for C could work, based on a number of representative examples (which were derived from the identified crosscutting concerns within the ASML source code).

Phase 1b: Construction of a prototype weaver and implementation of aspects, to demonstrate that the technology is realizable, and show that it solves the actual crosscutting concerns, on a representative amount of source code. It was shown that for a particular component, with three aspects the number of statements was reduced by 26%.

Phase 1c: This phase consisted of a case study, which investigated a set of issues to be solved to enable critical industrial application of this technology [31]. These issues were: migration to AOP, availability of mature tooling, ability to switch off aspects, understandability, run-time performance, compile-time performance, and the ability to debug. For each issue it was demonstrated how it was solved or could be solved. The results of the case study convinced ASML to start the development of their own industrial quality weaver.

Phase 2: ASML and —to a lesser extent— the University of Twente embarked on a so-called transfer project, reported in Chapter 5, to apply AOP technology in production software, in the setting of large-scale embedded C programs.

Phase 3: ASML has started to apply the AOP technology in production software (also briefly reported upon in Chapter 5). As soon as the technology became available, software engineers needed to be trained in the concepts and usage of WeaveC. As a part of this training, we developed two practical sessions that were in fact controlled experiments to assess the distinction in effort and errors made between the conventional idiom and a solution based on AOP, for a set of typical maintenance tasks. Brief results of this experiment have been described in Chapter 5; for more elaborate results of the experiments we refer to [34].

A spin-off transfer project was initiated by the University of Twente and ASML, based on the results achieved during phases 1 and 2 of the maturity model of WeaveC. ASML was developing a new system using the .NET framework. The University of Twente had developed an AOP solution, based on the Composition Filters model, with a prototype implementation for the .NET framework. This prototype and the experiences with WeaveC enabled us to quickly advance to phase 1b of the maturity model. The issues that were voiced by the stakeholders of the .NET-based project where similar to those for the C language (mentioned above). However, since all of the code in the .NET-based project was newly created, a migration path was not necessary. In the spin-off project, partially funded by ASML, we advanced to phase 1c. At the time of writing it is still unsure whether ASML will adopt the tooling and move to phase 2.

Meanwhile, the actual *research* questions within Ideals with respect to AOP, focused on problems related to large-scale application of AOP technology; as such, these questions did not directly come from the state-of-practice (since large-scale application of AOP technology, especially in embedded systems is still rare). On the other

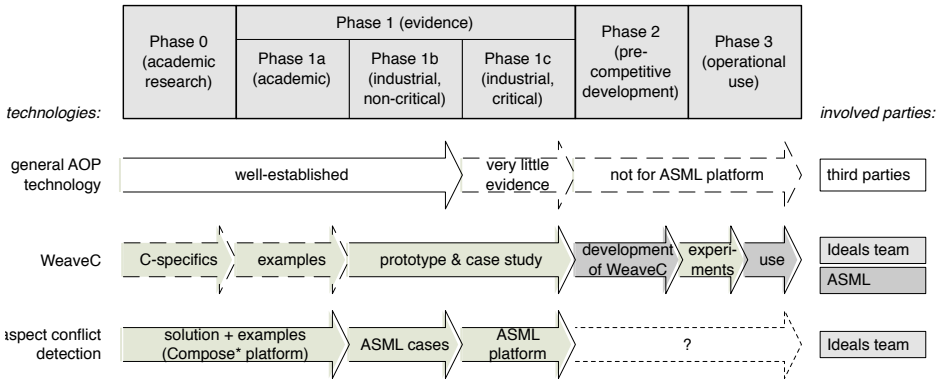


Figure 10.2: An overview of the maturity of AOP research.

hand, they come from one of the first objections raised by software engineers when encountering AOP technology: 'How can I be sure that a certain aspect does not accidently conflict with my program?'. We focused on a situation that is particularly hard to detect, and thus had the highest priority to address. This situation occurs when the combination of two or more aspects is conflicting. This is a research question that in fact applies to general AOP technology. We demonstrated that the detection of conflicts is also applicable for two common (ASML) idioms: error handling and contract enforcement. This research is reported in Chapter 4.

The maturity process of this research developed as follows. First, in a purely academic setting, a solution to achieve the detection of behavioral conflicts between aspects was created. That solution was tested on a number of small examples. It was implemented and tested in the mostly academic Compose* platform [3], covering phases 0 and 1a of the maturity model. Secondly, we tested the solution for realistic aspects based on real-world idiom (i.e. phase 1b). At the time of writing this chapter, we are in progress to allow the tooling that performs the conflict detection to work in a C-based environment. Once this is the case, we can run the conflict detection on industrial case studies of substantial size (i.e. phase 1c). Only after this has been achieved, will we have acquired sufficient evidence to convince a company such as ASML to include such checks in their production software (moving to phases 2 and 3).

10.5 Industrial impact - Angel

The development and maintenance of today's software systems is an increasingly effort-consuming and error-prone task. A major cause of this problem is the lack of formal *and* human-readable documentation of software design. In practice, software design

is often informally documented, or not documented at all. Therefore, the design cannot be properly communicated between software engineers, it cannot be systematically analyzed, and the conformance of an implementation to the design cannot be verified. In Chapter 8, we addressed this problem for the design and documentation of the behavior implemented in procedural programs. We introduced a solution that consists of three components: (1) A graphical language called VisuaL, which enables engineers to specify constraints on the possible sequences of function calls from a given program; (2) a tool called CheckDesign, which automatically verifies the consistency between multiple specifications written in VisuaL; and (3) a tool called CheckSource, which automatically verifies that a given implementation conforms to the corresponding specifications written in VisuaL.

After we developed VisuaL, our purpose was to find out whether VisuaL is expressive enough to specify design constraints in real-life, and whether a software engineer can use VisuaL efficiently. To find preliminary answers to these questions, we trained a software engineer of ASML, who has 15 years of professional experience. After a 1-hour training, the engineer created three VisuaL specifications in approximately 2.5 hours. In terms of size, each of these specifications corresponds to two A4 pages of regular design documentation.

The first specification created by the engineer contains 11 nodes and 19 edges. To create this specification, and to draw it using Borland Together, the engineer spent 80 minutes in total. In Table 10.2, the data for each of the three specifications is listed.

Specifications	**# Rectangles**	**# Arrows**	**Effort in minutes**
Spec1	11	19	80
Spec2	11	23	47
Spec3	10	20	28

Table 10.2: Data of the VisuaL Specifications.

Using the data presented in Table 10.2, one can calculate that the engineer spent on the average 160, 83, and 56 seconds per rectangle or arrow while creating Spec1, Spec2, and Spec3, respectively. This calculation indicates that the engineer quickly gained speed in creating specifications.

To create the specifications, the engineer had to rigorously analyze the relationship between the implementation, the detailed design, the architecture, and the requirements of the software component. This rigorous analysis enabled him to find one defect, which had to be repaired in the next release, four design anomalies that required restructuring and maintenance, and one undocumented feature. Two weeks earlier, the component in which the engineer found these problems had been maintained by himself, and reviewed by two of his colleagues.

This initial experience indicates that VisuaL can be expressive enough to be useful, at least to some extent, in an industrial context.

To evaluate CheckSource, we conducted a controlled experiment with 21 M.Sc.

computer science students, and repeated this experiment with 23 professional software developers of ASML.

During both experiments, the participants worked with (a) three C functions that were selected from the source code of ASML by the software engineer mentioned above, and (b) the VisuaL specifications that were created by the software engineer.

We injected an inconsistency between each function and the corresponding VisuaL specification, and then we requested the participants to restore the consistency by modifying the functions, such that each function would conform to the corresponding specification.

We formulated the following hypotheses:

- $H_0{}^1$: CheckSource does not have any effect on the amount of effort spent by the participants.
- $H_0{}^2$: CheckSource does not have any effect on the number of errors made by the participants.

Based on the data we collected and the statistical tests we performed, we successfully rejected both $H_0{}^1$ and $H_0{}^2$, in both the student and developer experiments, at the significance level 0,01.

In the student experiment, CheckSource reduced the effort spent by an average student by 50%. In addition, CheckSource prevented approximately one error per 100 lines of source code.

In the developer experiment, CheckSource reduced the effort spent by an average developer by 75%. In addition, CheckSource prevented approximately one error per 140 lines of source code.

Based on the positive results of the experiments and the positive subjective opinions of the ASML developers who participated in the experiment, ASML decided to invest in a transfer project, which is called Angel.

10.6 Industrial impact - Model-driven engineering

The most important result of the model-driven engineering research within Ideals (but also within Tangram [98]) is that it put model-driven engineering on the ASML roadmap for system and software development. By carrying out these projects and through many intensive discussions, the vision on model-driven engineering was sharpened and the challenges to apply MDE techniques in large scale industrial contexts became clear. A broad awareness was created on how model-driven engineering techniques will improve future engineering efficiency. A large follow-up research program to stimulate a company-wide introduction of model-driven engineering techniques is currently being developed. The exploration we performed in the area of model-driven engineering, delivered a number of concrete results, next to the Angel transfer project described in Section 10.5:

- The performance modeling case study for the light control subsystem as described in Chapter 7 was very successful. The model provided insight in the overall subsystem behavior. Traditional design documentation provided many details, but did not shed light on the big picture. The design documentation specified a number of separate message sequence charts, but the (timed) behavior emerging from their collaborations was unclear, also because the design documents were informal and not executable. The executable model combined the separate pieces into one unified whole. The immediate feedback obtained from the model eased communication with the design team. It allowed us to understand and verify the light control subsystem before it was implemented in terms of hardware and software components. The mapping of the application onto the hardware platform was investigated in one unifying model resulting in a concrete task priority scheme. Also, a timing problem concerning the forwarding of application data to a hardware device was found. Although designers new that the realization behaved incorrectly once and a while, the root course for this problem was not know. This root course was discover in the executable model which also provided inspiration for a robust solution and possibilities for verification. Last, but not least, an expected timing bottleneck in an on-board communication switch turned out to be non issue. On the other hand, cache misses turned out to cause a major timing problem. Based on this identification, strategies to minimize cache misses was developed. In summary, the modeling exercise demonstrated the feasibility to accurately predict system behavior and the impact of design decisions in an early phase of the design process. In this sense this work has passed phase 1b (non-critical evidence) of industrial evidence.

- The feasibility to generate code from models in a property-preserving way has been demonstrated (see Chapter 7). In their current form, the applied techniques still have their limitations however. Generated implementations are not resource-efficient enough for the embedded systems domain and distributed and mixed hard and soft real-time systems cannot be dealt with yet. As such this part of the research is still in phase 1a (academic evidence) of maturity. However, the results achieved were promising enough to motivate ASML in continuing along the same research lines by defining a number of Master's projects.

- Even in the relatively small research exercises, the diversity of models has become apparent. To be able to construct a realization or to be able to predict properties thereof, it is important to be able to combine these models in a consistent way. Inspired by the available experience with programming in a large scale industrial context, we strive to an approach that avoids copying information (single point of definition) and that minimizes the number of different languages designers have to be acquainted with. One implication is the ability to allow for different models of computation to interact with each other. As an example we studied the retrieval of information for a performance analysis model from a broader, general purpose UML model. Although an automated transformation

has not been developed yet, we are confident that this is well possible. However, to verify the feasibility of a shared information model, a transformation to at least one other formalism (targeting e.g. formal verification) should be developed, thereby re-using the same information. In this sense this part of the research is only in the early phase of academic evidence. Already as an important result, the search and identification of the relevant models of computation has become a key issue in the ASML road-map for model-driven engineering and in newly defined research proposals.

- The work on modeling the coordination concern and transforming this model into an implementation (see Chapter 6) has passed phase 1b (industrial evidence, non critical) of maturity and has shown the practical feasibility of the developed techniques. Due to these results, ASML has invested in carrying this research further. A prototype tool has been developed from which a software component can be derived automatically, starting from an abstract model of this concern. This prototype is already used in the software development process within ASML, although some improvements can still be made. Hence this work is currently in phase 2 (pre-competitive development) of industrial evidence. The goal is to further develop this prototype into a 'shared technology item' that is rolled out in the organization, bringing the results in the final phase (operational use) of maturity.

- In Chapter 9 we describe an approach to (partly) automate the migration of SMC systems in a wafer scanner. This approach is based on model-driven software development technologies. We developed a meta model for the target of the migration and specified model transformations to automate the migration of legacy models to a new control architecture. The use of such an approach has the advantage that the design knowledge incorporated in legacy designs is not lost. In this particular case this was especially useful, because to reduce risk one goal of the migration was to preserve the behavior of the legacy SMC components; optimizations that are made possible by the new architecture are only applied later.

  We applied our approach to two manufacturing requests (of two different SMC components). The source models for our migration approach were obtained from the documentation of the legacy SMC components. Here, we took into account the informal use of modeling in industry. For instance, different model patterns were used to model the same concern in different models. This makes these type of models not directly suited for the application of (automated) model transformations. Therefore, we introduced a normalization step to obtain source models in which standardized idioms are applied for various concerns. To this end, we identified important SMC concerns and provided a normalized idiom for each of them. After application of a normalization step, which is carried out manually, normalized models serve as source models for our model transformation and target models for the new SMC architecture are automatically generated.

We applied our approach to real SMC components and used the actual design documentation of these components to obtain the source models for our approach. Furthermore, we collaborated with some of the engineers involved in the migration of these components. As such, this part of our research can be classified as having passed maturity phase 1b (industrial evidence, non-critical).

10.7 Lessons learned on industrial impact

During the Ideals project various concepts and methods were developed. As explained in the previous sections, some of them were implemented by ASML, while others did not reach this level of maturity.

The phases of industrial evidence helped both the researchers and the ASML representatives to express and reason about the status of the research, the expectations and the follow-up steps. The phases of industrial evidence, however, do not explain why certain research topics reached the level of operational use and others did not.

In order to create a better insight in the elements that influence the growth towards industrial application of an idea, an evaluation of the various experiments and transfer projects was carried out. The so-called PPSF (Power, Persuasiveness, Support, Feasibility) model [2] was very useful for this purpose.

This paragraph addresses the lessons learned based on a combination of the various phases of industrial evidence and the PPSF model. The PPSF model shows the relationship between the key elements that are crucial for the development of an idea, see Figure 10.3. Besides the core elements of *idea* and *idea-owner*, the following elements can be distinguished:

Power (of the idea-owner). The power or standing of the idea-owner is crucial and can be based on admiration, actions, motives or trust. The role and position of the idea owner are dominant factors.

Persuasiveness (of the idea). Besides the perceptible added value, issues like style, persuasion, appeal and fit for use, are possible indicators for the potential of an idea. Facts, arguments, reasoning and statements emphasize the persuasiveness of the idea.

Support. The support for an idea is directly related to the position and role of the idea-owner and is dictated by stakeholders. The degree to which stakeholders can identify themselves with the intentions of the idea-owner and the degree to which the idea meets their needs, intentions and concerns determines their support.

Feasibility. Feasibility is the most concrete element and is often examined in an early phase. Making an inventory and an analysis based on facts are activities to prove the feasibility of an idea.

Based on the elements of the PPSF model we are able to clarify the lessons when evolving through the different stages of industrial maturity:

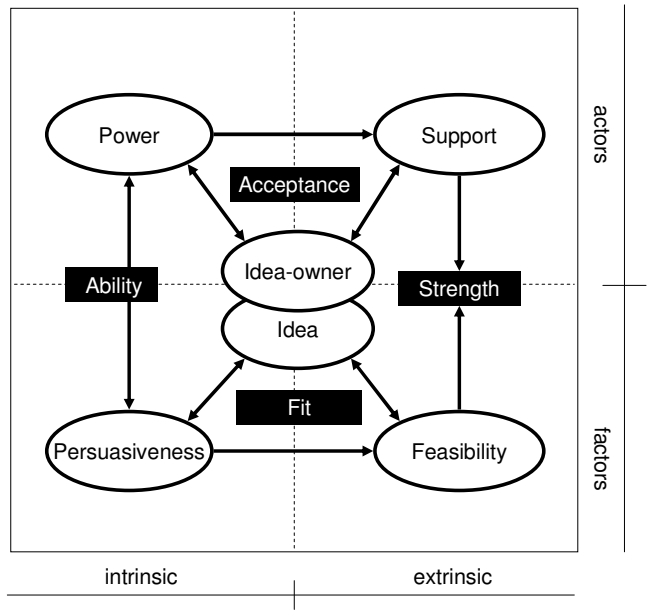


Figure 10.3: Power, Persuasiveness, Support, Feasibility (PPSF) model.

Power. Universities and Ph.D.-students in particular, play an important role in the industry-as-laboratory approach. From an industrial point of view, Ph.D.-students are relatively unknown. The 'power' of a Ph.D.-student, especially at the beginning of the project, is therefore rather limited. To compensate for this lack of power, it is important that others with sufficient industrial standing take part in the development and introduction of the idea. Research fellows from the Embedded Systems Institute together with respected industrial architects and group leaders, who are committed to the project, can fulfill this role.

Persuasiveness (of the idea). For academics, writing and publishing papers in a dedicated technical community is common practice. The publications are often theoretical, conceptual and focused on a specific topic. In an industrial environment it is hard to understand what the impact of a new conceptual idea will be. Just presenting the concept itself is therefore not enough. The industrial context in which the idea is going to be used must be part of the presentation. Furthermore it is important to present the idea in a way that the industrial stakeholders on higher and middle management understand the message.

Support. Understanding the problem is half the solution. Various stakeholders, on different levels and positions in the organization, must understand how the idea contributes to the problems they are facing. Higher management and middle

management must be able to position the potential solution in a broader long-term context. In the phases of academic evidence and industrial evidence (non-critical and critical), dedicated road-maps on company or department level are needed to prevent that solutions will be perceived as isolated and as a technology-push.

Additional to the road-maps, support is needed from middle management to initiate and execute experiments. In this, group leaders, project leaders, process owners and architects on a middle management level, are all stakeholders. They must be aware of the support that is needed, how results affect their own work and what the possible next steps will be.

Last but certainly not least, support is needed from engineers who are involved in the experiments. They apply the new concepts, methods and tools and can reflect on the operational added value compared to the current way of working. Positive reactions to the experiments encourage future support on all levels.

Feasibility. The feasibility of the idea is directly coupled to the phases of industrial evidence. At the end of each phase an explicit go / no-go moment should be created so that project management can decide whether and how to continue.

In the phase of academic evidence a lot of emphasis is on the concept itself and on a qualitative and quantitative analysis of the problem domain. Their combination gives insight in the possible added value of the solution.

In the next (non-critical industrial evidence) phase, simple and isolated industrial cases with sufficient relevance are selected to carry out the experiments. Focus is on the functionality of the solution and not on industrial constraints such as usability, scalability et cetera.

In the final (critical industrial evidence) phase of the research project, the industrial cases become more complex and also the industrial constraints are taken into account. In this phase a lot of emphasis is on concrete and quantitative results.

The PPSF model was used at the end of the Ideals project to create insight in the elements that influence the development of an idea during the various phases of maturity. Suggestion for future industry-as-laboratory research projects is that the model is used, in a more explicit way, at the end of each phase of industrial evidence. This in order to get a better insight in, and control on, the factors that influence the development of the idea in the next phase of creating industrial evidence.

10.8 Conclusions

The goal of the Ideals project was to provide practically useful innovations that reduce the effort and lead time to maintain and improve complex embedded systems in general, and ASML's lithography system in particular. The results, as presented in the

previous chapters and reviewed for their practical impact in this chapter, are a number of methods, techniques and tools with different maturity levels.

From an ASML point of view, we can conclude that significant results have been achieved:

- The Ideals project has effectively demonstrated that traditional idiom-based approaches clearly limit the development efficiency, both in terms of effort and quality. It has also demonstrated that for a subset of crosscutting concerns, an aspect-oriented programming solution can provide a modular implementation with much better development efficiency characteristics. This was exactly the motivation for ASML to invest in the WeaveC transfer project, to create an AOP methodology and tool set for embedded systems using the C language.

- The WeaveC transfer project introduced an AOP methodology, tool set and training to ASML developers. Developers who use WeaveC have indicated in a survey that they clearly experience the benefits in terms of reduction of workload and errors. This succes can be demonstrated by the fact that the WeaveC tool is now used in a significant, and continuously growing, portion of the ASML software[1], on multiple target platforms and in multiple application domains. A clear benefit was the pilot-style introduction: while the methodology and tool set were still under construction, it was supplied to a limited set of customers with dedicated support who gave valuable feedback in its use. The next step in the introduction will be a broad rollout of a much more mature version of the methodology and tool set, allowing ASML to reap the benefits on an even larger scale. At this moment the aspects in use address variations of a logging concern (function parameter tracing). Potential application for other aspects is being investigated.

- Even if an aspect-oriented programming solution is not (yet) feasible for all crosscutting concerns, the research into measuring the reduced quality impact of using idioms has demonstrated methods for detecting two typical classes of faults in ASML's software: error linking faults and activity sequencing faults. These methods have clear potential as they can be used to increase the efficiency of code reviews and improve system quality.

- In multiple case studies the Ideals project has demonstrated how software designs can be made much more useful by basing them on effective modeling abstractions: these designs form the basis of analysis and synthesis, to provide insight into key performance parameters and to enable transformations to actual implementations or to allow automated validation of implementations. These demonstrations were performed in real-life domains, and the analysis and synthesis possibilities were verified with industrial end-users to be relevant, accurate and time-saving.

[1] At the time of writing of this Chapter, the WeaveC tool is applied to more than 1,000 source files and this number grows by approximately 250 source files per month.

Not all of the ambitions from the start of the Ideals project could be realized. The intention was to explore the similarity between aspect-oriented programming (merging base programs with different aspects to form a complete program) and model-driven architecture (merging different models to form a detailed model). However, the practical differences in concepts, implementations and maturity, limited the synergy that could be gained from researching these two areas in a single research project. Other concrete areas that are still open at the end of the Ideals project are:

- There are still several crosscutting concerns for which no modular implementation strategy could be developed. The most obvious example is exception handling, which seems to defy a modular implementation because of its very detailed interaction with the control flow of the base program.
- Migration of existing idiom-based code to an aspect-based solution proved to be much more difficult than expected. A workable solution has yet to be devised. Part of the problem is the large number of faults that is usually present in idiom implementations: should these faults be fixed or migrated? But even with no faults remaining, idioms typically still have a large amount of (harmless) variations that must all be supported and mapped to a single/canonical version.
- Model-driven engineering is a very heterogeneous domain, with a broad range of definitions, opinions and visions. The 'big picture' of model-driven engineering in large-scale industrial systems still has to be defined. It requires the identification of the required modeling paradigms as well as the establishment of their relations. Furthermore, it should become clear what the prerequisites are for successful application of the diverse activities that are associated with model-driven engineering. Hence a lot of future research is still necessary before model-driven engineering techniques will dominate the engineering process within ASML.

The industry-as-laboratory style of research adopted in the Ideals project has resulted in succes, both by academic standards (publications and Ph.D. dissertations) and industrial standards (accepted and introduced innovations that bring value). There are some key factors for making this research style work effectively:

- It is very important for the academic researchers to be located on the site of the industrial partner for a significant part of their time. Continuous interaction with a large variety of people is crucial for proper understanding of the industrial problems and selection of viable solution areas. Understanding the problem must not be limited to the technical problem itself, but must also involve the context in which it occurs, the possible reactions of stakeholders to different solution directions, the constraints that are placed on solutions, et cetera. Only during the research it becomes clear which people are relevant for this understanding and for defining these constraints, so it is crucial that these people can be approached and involved in an effective and efficient way: a researcher that is on-site on a regular basis is able to have quick interactions with these people, and has better chances of producing a practically accepted solution.

- Transfer of research results to industry not only requires a clear solution with a proven result, but also someone who positions himself as the problem owner for the adoption of the solution. This problem owner should be convinced about the merits of the proposed solution, and must provide enough focus to the continual balance between the required activities and the available resources. For a subject like crosscutting concerns, this has turned out to be difficult, but not impossible as everybody feels a little bit of the problem. It especially demonstrates to be difficult for a subject like MDE, where many people feel the need, but where the solution is not very crisp yet and prone to priority-debates.

- Clear phases of research (as described in Section 10.2) help in keeping everybody focussed on the progression of research to practical results. Initially they help academic partners to provide the relative quiet in which to fully understand and explore the problem, and the ability to generate solution directions without a lot of immediate detailed discussions about advantages and disadvantages. But they also provide a means to step up the involvement from the industrial party, first through qualitative assessments, later through fact-based pilots and finally in real-life implementations. Although the transition at the end is sometimes difficult to manage, since it concerns the handover of project responsibility from research project to industrial partner, it is very worthwhile to maintain everyone's involvement till the end. The industrial partner benefits from the researcher partners' experiences with solution alternatives and prototype implementations. The research partners benefit from the larger scale on which a technology is applied, allowing for better setup of experiments to validate theories, and which often brings up interesting and unexpected new research questions.

Some research results are firmly embedded in the ASML environment, ready for wider adoption in the embedded software industry. Some research results are finished from a research point of view, and are ready for adoption by ASML. Some research results primarily provide more questions and future research topics, not a practically applicable methodology or technology yet. We hope that the network of partners that joined the project will also play a role in the further dissemination and growth of all the Ideals research results.

Appendix A

Ideals publications

1. Bruntink M. Analysis and transformation of idiomatic crosscutting concerns in legacy software systems. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pp. 499–500. IEEE Computer Society Press, 2007.

2. Bruntink M., van Deursen A., D'Hondt M., and Tourwé T. Simple Crosscutting Concerns Are Not So Simple – Analysing Variability in Large-scale Idioms-based Implementations. In *Proceedings of the Sixth International Conference on Aspect-Oriented Software Development (AOSD'07)*, pp. 199–211. ACM Press, March 2007.

3. Bruntink M. Linking Analysis and Transformations Tools with Source-based Mappings. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pp. 107–116. IEEE Computer Society Press, September 2006.

4. Bruntink M., van Deursen A., and Tourwé T. Discovering Faults in Idiom-Based Exception Handling. In *Proceedings of the International Conference on Software Engineering (ICSE'06)*, pp. 242–251. ACM Press, 2006.

5. Bruntink M. Aspect Mining Using Clone Class Metrics. In *Proceedings of the 2004 Workshop on Aspect Reverse Engineering (co-located with the 11th Working Conference on Reverse Engineering (WCRE'04))*. November 2004. Published as CWI technical report SEN-E0502, February 2005.

6. Bruntink M., van Deursen A., and Tourwé T. Isolating Idiomatic Crosscutting Concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*, pp. 37–46. IEEE Computer Society, 2005.

7. Bruntink M., van Deursen A., van Engelen R., and Tourwé T. On the Use of Clone Detection for Identifying Cross Cutting Concern Code. *IEEE Transactions*

on Software Engineering, volume 31(10):pp. 804–818, 2005.

8. Bruntink M., van Deursen A., van Engelen R., and Tourwé T. An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. In *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 200–209. IEEE Computer Society, 2004.

9. Bruntink M., van Deursen A., and Tourwé T. An initial experiment in reverse engineering aspects from existing applications. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)*, pp. 306–307. IEEE Computer Society, 2004.

10. Durr P.E.A., Bergmans L.M.J., and Aksit M. Static and Dynamic Detection of Behavioral Conflicts between Aspects. In *Proceedings of the 7th Workshop on Runtime Verification*, number 4839 in LNCS, pp. 38–50. Springer Verlag, Vancouver, Canada, March 2007.

11. Durr P., Gulesir G., Bergmans L., Aksit M., and van Engelen R. Applying AOP in an Industrial Context. In *Workshop on Best Practices in Applying Aspect-Oriented Software Development*. March 2006.

12. Durr P., Bergmans L., and Aksit M. Reasoning about Semantic Conflicts between Aspects. In R. Chitchyan, J. Fabry, L. Bergmans, A. Nedos, and A. Rensink, editors, *Proceedings of ADI'06 Aspect, Dependencies, and Interactions Workshop*, pp. 10–18. Lancaster University, July 2006.

13. Gool L., Punter T., Hamilton M., and Engelen R. Compositional MDA. In Nierstrasz, O. et al, editor, *Proceedings of ACM/IEEE Models 2006*, volume 4199 of *Lecture Notes in Computer Science*, pp. 126–139. Springer, 2006.

14. Graaf B. *Model-Driven Evolution of Software Architectures*. Ph.D. thesis, Delft University of Technology, November 2007.

15. Graaf B., Weber S., and van Deursen A. Model-driven migration of supervisory machine control architectures. *Journal of Systems and Software*, 2008. Doi: 10.1016/j.jss.2007.06.007.

16. Graaf B., Weber S., and van Deursen A. Migrating supervisory control architectures using model transformations. In G. Visaggio, G. Antonio Di Lucca, and N. Gold, editors, *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pp. 151–160. IEEE Computer Society, 2006.

17. Graaf B., Weber S., and van Deursen A. Migration of supervisory machine control architectures. In R. Nord, N. Medvidovic, R. Krikhaar, J. Stafford, and J. Bosch, editors, *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*, pp. 261–262. IEEE CS, November 2005.

18. Gulesir G., Bergmans L., Durr P., and Nagy I. Separating and managing dependent concerns. LATE 2005 workshop at AOSD 2005.

19. Gurzhiy T. Model Transformations using QVT - Feasibility Analysis by Implementation. SAI Technical Report 2006056, Eindhoven University of Technology, August 2006.

20. Huang J., Voeten J., and Corporaal H. Predictable real-time software synthesis. *Real-Time Systems Journal*, volume 36(3):pp. 159–198, 2007.

21. Huang J., Voeten J., Groothuis M., Broenink J., and Corporaal H. A model-driven design approach for mechatronic systems. In IEEE Computer Society, editor, *Proceedings of the 7th International Conference on Application of Concurrency to System Design – ACSD-07*, pp. 127–136. 2007.

22. Huang J. and Voeten J. Predictable model-driven design for real-time embedded systems. In *Proceedings of Bits & Chips conference.* 2007.

23. Punter T. and Gool L. Experience Report On Maintainability Prediction at Design Level. Ideals technical report, Embedded Systems Institute, October 2005.

24. Punter T., Voeten J., and Huang J. Quality of Model Driven Engineering. In *Model-Driven Software Development: Integrating Quality Assurance.* To appear.

Appendix B

List of authors

Joris van den Aker
Embedded Systems Institute
joris.van.den.aker@esi.nl

Mehmet Akşit
University of Twente
aksit@ewi.utwente.nl

Lodewijk Bergmans
University of Twente
bergmans@ewi.utwente.nl

Mark van den Brand
Eindhoven University of Technology
M.G.J.v.d.Brand@tue.nl

Magiel Bruntink
Centrum voor Wiskunde en Informatica
magiel.bruntink@cwi.nl

Arie van Deursen
Delft University of Technology
Arie.vanDeursen@tudelft.nl

Pascal Dürr
University of Twente
durr@ewi.utwente.nl

Luc Engelen
Eindhoven University of Technology
l.j.p.engelen@tue.nl

Remco van Engelen
ASML
remco.van.engelen@asml.com

Louis van Gool
ICT NoviQ
louis.van.gool@ict.nl

Bas Graaf
Delft University of Technology
b.s.graaf@tudelft.nl

Gürcan Güleşir
University of Twente
gulesirg@ewi.utwente.nl

Tanja Gurzhiy
ASML
tanja.gurzhiy@asml.com

Marc Hamilton
ASML
Marc.Hamilton@asml.com

Maja d'Hondt
IMEC
dhondtm@imec.be

Andriy Levytskyy
ASML
andriy.levytskyy@asml.com

Durk van der Ploeg
ASML
durk.van.der.ploeg@asml.com

Tom Tourwé
Sirris
tom.tourwe@sirris.be

Sven Weber
ASML
sven.weber@asml.com

Paul Klint
Centrum voor Wiskunde en Informatica
paul.klint@cwi.nl

István Nagy
ASML
istvan.nagy@asml.com

Teade Punter
Embedded Systems Institute
teade.punter@esi.nl

Jeroen Voeten
Embedded Systems Institute
Eindhoven University of Technology
j.p.m.voeten@tue.nl

Ideals partners (institutions and companies):

ASML	Veldhoven, The Netherlands
Centrum voor Wiskunde en Informatica	Amsterdam, The Netherlands
Embedded Systems Institute	Eindhoven, The Netherlands
Delft University of Technology	Delft, The Netherlands
Eindhoven University of Technology	Eindhoven, The Netherlands
University of Twente	Enschede, The Netherlands

For more information: office@esi.nl

References

[1] Adams E.N. Optimizing Preventive Service of Software Products. *IBM Journal of Research and Development*, volume 28(1):pp. 2–14, 1984.

[2] Bekkering T., Glas H., and Klaassen D. *Management van processen - Succesvol realiseren van complexe initiatieven*. Het Spectrum, 2004.

[3] Bergmans L. and Akşit M. Principles and Design Rationale of Composition Filters. In R.E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pp. 63–95. Addison-Wesley, Boston, 2005.

[4] Bernstein A.J. Program Analysis for Parallel Processing. *IEEE Trans. on Electronic Computers*, volume EC-15:pp. 757–762, October 1966.

[5] Boehm B.W. *Software Engineering Economics*. Prentice-Hall, 1981.

[6] van den Brand M., van Deursen A., Heering J., de Jong H.A., de Jonge M., Kuipers T., Klint P., Moonen L., Olivier P.A., Scheerder J., Vinju J.J., Visser E., and Visser J. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pp. 365–370. Springer, 2001.

[7] Brichau J. and Haupt M. Survey of Aspect-Oriented Languages and Execution Models.

[8] Brodie M.L. and Stonebraker M. *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*. Morgan Kaufmann, 1995.

[9] Brouwer S. and Nagy I. Mirjam. Internal document 107731, ASML, 2007.

[10] Bruntink M. Aspect Mining Using Clone Class Metrics. In *Proceedings of the 2004 Workshop on Aspect Reverse Engineering (co-located with the 11th Working Conference on Reverse Engineering (WCRE'04))*. November 2004. Published as CWI technical report SEN-E0502, February 2005.

[11] Bruntink M. Linking Analysis and Transformations Tools with Source-based Mappings. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pp. 107–116. IEEE Computer Society Press, September 2006.

[12] Bruntink M. Analysis and transformation of idiomatic crosscutting concerns in legacy software systems. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pp. 499–500. IEEE Computer Society Press, 2007.

[13] Bruntink M., van Deursen A., D'Hondt M., and Tourwé T. Simple Crosscutting Concerns Are Not So Simple – Analysing Variability in Large-scale Idioms-based Implementations. In *Proceedings of the Sixth International Conference on Aspect-Oriented Software Development (AOSD'07)*, pp. 199–211. ACM Press, March 2007.

[14] Bruntink M., van Deursen A., van Engelen R., and Tourwé T. An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. In *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 200–209. IEEE Computer Society, 2004.

[15] Bruntink M., van Deursen A., van Engelen R., and Tourwé T. On the Use of Clone Detection for Identifying Cross Cutting Concern Code. *IEEE Transactions on Software Engineering*, volume 31(10):pp. 804–818, 2005.

[16] Bruntink M., van Deursen A., and Tourwé T. An initial experiment in reverse engineering aspects from existing applications. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)*, pp. 306–307. IEEE Computer Society, 2004.

[17] Bruntink M., van Deursen A., and Tourwé T. Isolating Idiomatic Crosscutting Concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*, pp. 37–46. IEEE Computer Society, 2005.

[18] Bruntink M., van Deursen A., and Tourwé T. Discovering Faults in Idiom-Based Exception Handling. In *Proceedings of the International Conference on Software Engineering (ICSE'06)*, pp. 242–251. ACM Press, 2006.

[19] Buschmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley series in Software design patterns. John Wiley & Sons, 1996.

[20] Bush M. Improving software quality: the use of formal inspections at the JPL. In *Proceedings of the International Conference on Software Engineering*, pp. 196–199. IEEE Computer Society, 1990.

[21] Christian F. *Exception handling and tolerance of software faults*, chapter 4, pp. 81–107. John Wiley & Sons, 1995.

[22] Clarke E.M., Grumberg O., and a. Peled D. *Model Checking*. The MIT Press, 1999.

[23] Coady Y., Kiczales G., Feeley M., and Smolyn G. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of the Joint European Software Engineering Conference (ESEC'01) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'01)*, pp. 88–98. ACM Press, June 2001.

[24] Codd E. A Relational Model of Data for Large Shared Data Banks. volume 13, pp. 377–387. 1970.

[25] Colyer A. and Clement A. Large-scale AOSD for middleware. In *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD'04)*, pp. 56–65. ACM Press, New York, NY, USA, 2004. doi:http://doi.acm.org/10.1145/976270.976279.

[26] Coplien J. *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1991.

[27] Csertán G., Huszerl G., Majzik I., Pap Z., Pataricza A., and Varró D. VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models. In *ASE 2002: 17th IEEE International Conference on Automated Software Engineering*, pp. 267–270. IEEE Press, 2002.

[28] Demeyer S., Ducasse S., and Nierstrasz O. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

[29] Durr P., Bergmans L., and Aksit M. Technical Report: Formal model for SECRET. Technical report, University of Twente, 2005.

[30] Durr P., Bergmans L., and Aksit M. Reasoning about Semantic Conflicts between Aspects. In R. Chitchyan, J. Fabry, L. Bergmans, A. Nedos, and A. Rensink, editors, *Proceedings of ADI'06 Aspect, Dependencies, and Interactions Workshop*, pp. 10–18. Lancaster University, July 2006.

[31] Durr P., Gulesir G., Bergmans L., Aksit M., and van Engelen R. Applying AOP in an Industrial Context. In *Workshop on Best Practices in Applying Aspect-Oriented Software Development*. March 2006.

[32] Durr P., Staijen T., Bergmans L., and Aksit M. Reasoning about Semantic Conflicts between Aspects. In *EIWAS '05: The 2nd European Interactive Workshop on Aspects in Software*. Brussel, Belgium, September 2005.

[33] Durr P.E.A. and Bergmans L.M.J. High-level Design of WeaveC. Internal document, ASML, 2006.

[34] Durr P.E.A., Bergmans L.M.J., and Aksit M. Initial Results for Quantifying AOP. Technical Report TR-CTIT-07-71, Centre for Telematics and Information Technology, University of Twente, 2007.

[35] Durr P.E.A., Bergmans L.M.J., and Aksit M. Static and Dynamic Detection of Behavioral Conflicts between Aspects. In *Proceedings of the 7th Workshop on Runtime Verification*, number 4839 in LNCS, pp. 38–50. Springer Verlag, Vancouver, Canada, March 2007.

[36] Dyer M. The Cleanroom Approach to Quality Software Development. In *Proceedings of the 18th International Computer Measurement Group Conference*, pp. 1201–1212. Computer Measurement Group, 1992.

[37] Elrad T., Filman R.E., and Bader A. Aspect-Oriented Programming. *Comm. ACM*, volume 44(10):pp. 29–32, October 2001.

[38] Elrad T., Filman R.E., and Bader A. Aspect-Oriented Programming: Introduction. *Communications of the ACM*, volume 44(10):pp. 29–32, October 2001.

[39] Engler D.R., Chelf B., Chou A., and Hallem S. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *4th Symposium on Operating System Design and Implementation*, pp. 1–16. USENIX Association, 2000.

[40] Evans D. and Larochelle D. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, Jan/Feb 2002.

[41] Fenton N.E. and Pfleeger S.L. *Software Metrics: A rigorous and Practical Approach*. PWS Publishing Company, second edition, 1997.

[42] Fowler M. Language Workbenches: The Killer-App for Domain Specific Languages? Technical report, June 2005.

[43] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns*. Addison-Wesley, 1995.

[44] Ganter B. and Wille R. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.

[45] Gool L., Punter T., Hamilton M., and Engelen R. Compositional MDA. In Nierstrasz, O. et al, editor, *Proceedings of ACM/IEEE Models 2006*, volume 4199 of *Lecture Notes in Computer Science*, pp. 126–139. Springer, 2006.

[46] Graaf B. *Model-Driven Evolution of Software Architectures*. Ph.D. thesis, Delft University of Technology, November 2007.

[47] Graaf B., Lormans M., and Toetenel H. Embedded Software Engineering: The State of the Practice. *IEEE Software*, volume 20(6):pp. 61–69, November–December 2003.

[48] Graaf B., Weber S., and van Deursen A. Migration of supervisory machine control architectures. In R. Nord, N. Medvidovic, R. Krikhaar, J. Stafford, and J. Bosch, editors, *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*, pp. 261–262. IEEE CS, November 2005.

[49] Graaf B., Weber S., and van Deursen A. Migrating supervisory control architectures using model transformations. In G. Visaggio, G. Antonio Di Lucca, and N. Gold, editors, *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pp. 151–160. IEEE Computer Society, 2006.

[50] Graaf B., Weber S., and van Deursen A. Model-driven migration of supervisory machine control architectures. *Journal of Systems and Software*, 2008. Doi: 10.1016/j.jss.2007.06.007.

[51] Gulesir G., Bergmans L., Durr P., and Nagy I. Separating and managing dependent concerns. LATE 2005 workshop at AOSD 2005.

[52] Gurzhiy T. Model Transformations using QVT - Feasibility Analysis by Implementation. SAI Technical Report 2006056, Eindhoven University of Technology, August 2006.

[53] Hannemann J., Chitchyan R., and Rashid A. Analysis of Aspect-Oriented Software, Workshop report. In *ECOOP 2003 Workshop Reader*. Darmstadt, Germany, July 2003.

[54] Hardebolle C., Boulanger F., Marcadet D., and Vidal-Naquet G. A generic execution framework for models of computation. In *Model-Based Methodologies for Pervasive and Embedded Software*, pp. 45–54. 2007.

[55] Hooman J. and van der Zwaag M.B. A semantics of communicating reactive objects with timing. *International Journal on Software Tools for Technology Transfer*, volume 8(2):pp. 97–112, 2006.

[56] Huang J. and Voeten J. Predictable model-driven design for real-time embedded systems. In *Proceedings of Bits & Chips conference.* 2007.

[57] Huang J., Voeten J., and Corporaal H. Predictable real-time software synthesis. *Real-Time Systems Journal*, volume 36(3):pp. 159–198, 2007.

[58] Huang J., Voeten J., Florescu O., van der Putten P., and Corporaal H. *Predictability in Real-Time System Development*, chapter 8, pp. 167–183. Kluwer Academic Publishers, 2005.

[59] Huang J., Voeten J., Groothuis M., Broenink J., and Corporaal H. A model-driven design approach for mechatronic systems. In IEEE Computer Society, editor, *Proceedings of the 7th International Conference on Application of Concurrency to System Design – ACSD-07*, pp. 127–136. 2007.

[60] IEEE-1471. IEEE Recommended Practice for Architectural Description of Software Intensive Systems. IEEE Std 1471–2000, 2000.

[61] Jouault F. and Kurtev I. Transforming Models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*. 2005.

[62] Kapur D. and Mandayam S. Expressiveness of the operation set of a data abstraction. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 139–153. ACM Press, New York, NY, USA, 1980. doi:http://doi.acm.org/10.1145/567446.567460.

[63] Kent S. Model Driven Engineering. In *Integrated Formal Methods: Third International Conference, LNCS 2335)*, pp. 286–298. Springer Berlin / Heidelberg, 2002.

[64] Kernighan B.W. and Ritchie D.M. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[65] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., and Griswold W. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming, June 18-22*, pp. 327–353. June 18-22 2001.

[66] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.M., and Irwin J. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pp. 220–242. Springer, 1997.

[67] Kienhuis B., Deprettere E., Vissers K., and van der Wolf P. Quantitative Analysis of Application-Specific Dataflow Architectures. In *1997 International Conference on Application-Specific Systems, Architectures, and Processors (ASAP '97)*, pp. 338–349. IEEE Computer Society, 1997.

[68] Kleppe A.G., Warmer J., and Bast W. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[69] Lang J. and Stewart D.B. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM Transactions on Programming Languages and Systems*, volume 20(2):pp. 274 – 301, 1998.

[70] Lange C.F.J., Chaudron M.R.V., and Muskens J. In Practice: UML Software Architecture and Design Description. *IEEE Software*, volume 23(2):pp. 40–46, March 2006.

[71] Leavens G.T. and Clifton C. Foundations of Aspect-Oriented Languages Workshop. In *Foundations of Aspect-Oriented Languages Workshop*, volume 3. AOSD, 2004.

[72] Leavens G.T. and Clifton C. Foundations of Aspect-Oriented Languages Workshop. In *Foundations of Aspect-Oriented Languages Workshop*, volume 4. AOSD, 2005.

[73] Lewis G. and Wrage L. Approaches to Constructive Interoperability. Technical Report CMU/SEI-2004-TR-020 ESC-TR-2004-020, Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005.

[74] Lindig C. and Snelting G. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pp. 349–359. ACM Press, 1997.

[75] Lions J.L. ARIANE 5 Flight 501 Failure. Technical report, ESA/CNES, 1996.

[76] Lippert M. and Lopes C.V. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the International Conference on Software Engineering*, pp. 418 – 427. IEEE Computer Society, 2000.

[77] Littlewood B. Dependability assessment of software-based systems: state of the art. In *Proceedings of the International Conference on Software Engineering*, pp. 6–7. ACM Press, 2005. doi:http://doi.acm.org/10.1145/1062455.1062461.

[78] Lynch N.A., Merritt M., Weihl W.E., and Fekete A. *Atomic Transactions: In Concurrent and Distributed Systems*. Morgan Kaufmann, 1993.

[79] Nagy I. *On the Design of Aspect-Oriented Composition Models for Software Evolution.* Ph.D. thesis, University of Twente, June 2006.

[80] Nagy I., Bergmans L., and Aksit M. Composing aspects at shared join points. In R. Hirschfeld, R. Kowalczyk, A. Polze, and M. Weske, editors, *Proceedings of International Conference NetObjectDays, NODe2005*, volume P-69 of *Lecture Notes in Informatics*. Springer-Verlag, Erfurt, Germany, Sep 2005.

[81] van den Nieuwelaar N. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Ph.D. thesis, Technische Universiteit Eindhoven, 2004.

[82] Noonan L. and Flanagan C. Utilising evolutionary approaches and object-oriented techniques for design space exploration. In *Euromicro Conference on Digital System Design*, pp. 346–352. IEEE Computer Society Press, 2006.

[83] OMG. OMG. http://www.omg.org/cgi-bin/doc?ad/2002-4-10.

[84] OMG. OMG Unified Modeling Language Specification, Version 1.4. http://www.omg.org/docs/formal/01-09-67.pdf, June 2007.

[85] Potts C. Software-Engineering Research Revisited. *IEEE Software*, volume 10(5):pp. 19–28, September/October 1993.

[86] Ptolemy. Http://ptolemy.eecs.berkeley.edu/.

[87] Punter T. and Gool L. Experience Report On Maintainability Prediction at Design Level. Ideals technical report, Embedded Systems Institute, October 2005.

[88] Punter T., Voeten J., and Huang J. Quality of Model Driven Engineering. In *Model-Driven Software Development: Integrating Quality Assurance.* To appear.

[89] van der Putten P. and Voeten J. *Specification of Reactive Hardware/Software Systems - The Method Software/Hardware Engineering*. Ph.D. thesis, Eindhoven University of Technology, The Netherlands, 1997.

[90] Ramadge P. and Wonham W. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, volume 25(1):pp. 206–230, 1987.

[91] Robillard M. and Murphy G.C. Regaining Control of Exception Handling. Technical Report TR-99-14, Department of Computer Science, University of British Columbia, 1999.

[92] Roo A. *Towards More Robust Advice: Message Flow Analysis for Composition Filters and its Application*. Master's thesis, University of Twente, March 2007.

[93] Sabuncuoglu I. and Bayiz M. Analysis of reactive scheduling problems in a job-shop environment. *European Journal of operational research*, volume 126:pp. 567–586, 2000.

[94] Tau generation 2. Http://www.taug2.com/.

[95] The Open Group. IEEE Std 1003.1, The make utility. The Open Group Base Specifications Issue 6, 2004.

[96] Theelen B., Florescu O., Geilen M., Huang J., van der Putten P., and Voeten J. Software/Hardware Engineering with the Parallel Object-Oriented Specification Language. In *ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 139–148. IEEE Computer Society, 2007.

[97] Toy W.N. Fault-tolerant design of local ESS processors. In *Proceedings of IEEE*, pp. 1126–1145. IEEE Computer Society, 1982.

[98] Tretmans G.E. *Tangram: Model-based integration and testing of complex high-tech systems*. Embedded Systems Institute, 2007.

[99] Viennot G.X. Heaps of Pieces, I: Basic definitions and combinatorial lemmas. In *Proceedings of the Colloque de combinatoire énumérative (UQAM 1985), Montreal, Canada*, volume 1234 of *Lecture Notes in Mathematics*, pp. 321–350. Springer, 1986.

[100] Wegner P. and Doyle J. Editorial: strategic directions in computing research. *ACM Computing Surveys*, volume 28(4):pp. 565–574, 1996.

[101] Weiser M. Program Slicing. *IEEE Transactions on Software Engineering*, volume 10(4):pp. 352–357, Jul 1984.

[102] van Wijk F., Voeten J., and ten Berg A. *An Abstract Modeling Approach Towards System-Level Design-Space Exploration*, chapter 22, pp. 167–183. Kluwer Academic Publishers, 2003.

[103] Zachmann. The Zachmann Institute for Framework Advancement. http://www.zifa.com.

[104] Zimmermann H. OSI Reference Model - The ISO Model of Architecture for Open System Communication. In IEEE, editor, *IEEE Transactions on Communication*, volume Com-28, Nr. 4, pp. 425–432. April 1980.

Embedded Systems
INSTITUTE

Ideals:
evolvability of software-intensive high-tech systems

A collaborative research project on maintaining complex embedded systems

Summary

The Ideals project is an industrial-academic research project managed by the Embedded Systems Institute in Eindhoven, The Netherlands. For a period of four years, researchers and engineers from ASML have worked closely together with researchers from Delft University of Technology, Eindhoven University of Technology, the University of Twente, the Center for Mathematics and Computer Science, and the Embedded Systems Institute. The research objective was to develop methods, techniques and tools to improve the design efficiency of software-intensive high-tech systems. Special attention has been paid to the evolvability characteristics of these systems in an environment characterized by continuous functional and technological change.

ASML, the leading global company for lithography systems for the semiconductor industry, has provided the industry-as-laboratory setting to develop and validate the project results. Driven by Moore's law, their highly complex systems are subject to constant change to deal with the ever-shrinking lithography dimensions. Given this background, ASML could provide a challenging and inspiring environment to develop new methods, improve upon existing approaches, and to validate the proposed solutions to systems of industrial size and complexity.

This book summarizes the results of the Ideals project. Methods are presented for two key areas of interest: (1) aspect-oriented programming, and (2) model-driven engineering. The methods have been validated in industrial practice with the latest wafer scanners, demonstrating substantial improvements in design and engineering efficiency.

Embedded Systems Institute

The Embedded Systems Institute is a public-private partnership founded in 2002 by a number of Dutch universities and companies: Delft University of Technology, Eindhoven University of Technology, University of Twente, TNO, ASML, Océ and Philips. Its mission is to advance industrial innovation and academic excellence in embedded systems engineering. A broad research program is executed, mostly based on cases from industrial practice. The research is carried out in partnership with academia, industrial partners and other research institutes. ESI also maintains an extensive knowledge dissemination program, including a wide variety of courses from system architecting and engineering.

Visit us at **www.esi.nl**