Ideals : an introduction to the project and the book

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.

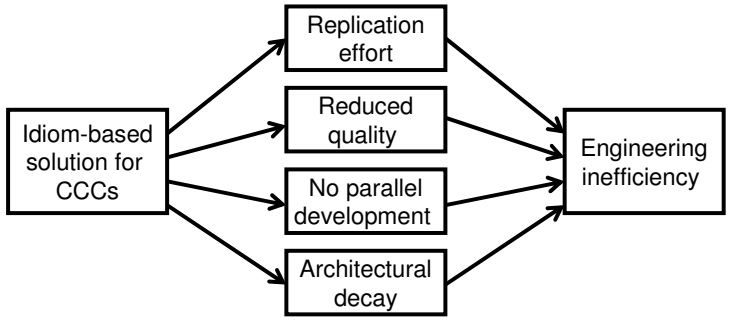

Figure 1.1: Consequences of an idiom-based solution for crosscutting concerns (CCC's).

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

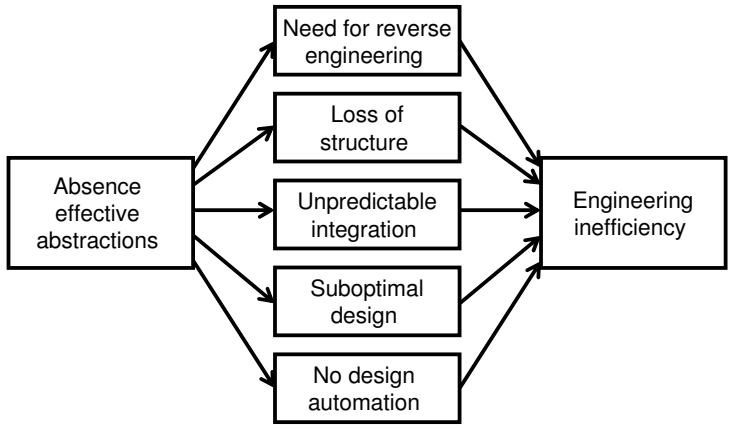

Figure 1.2: Consequences of ineffective abstractions.

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.

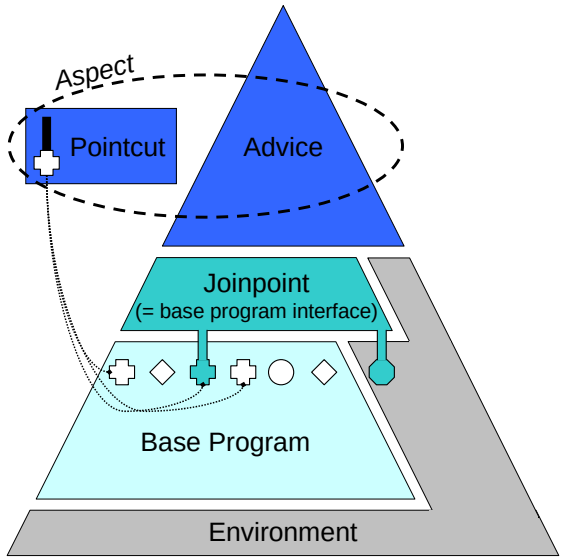

Figure 1.3: Parts of an aspect and their relations.

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?

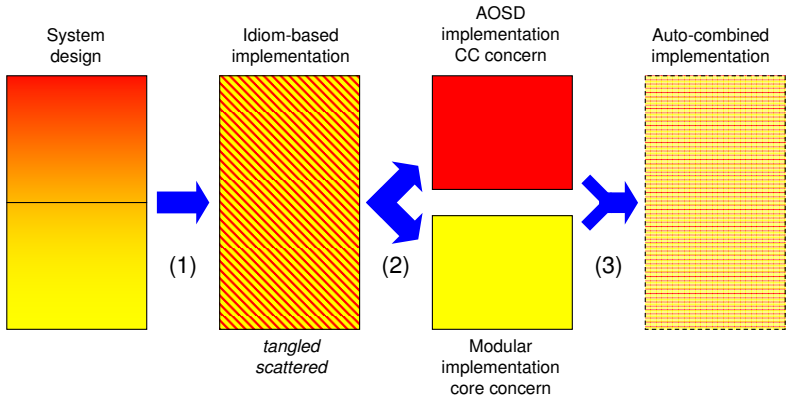

Figure 1.4: Migration path of an existing system to an AOSD based system.

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].

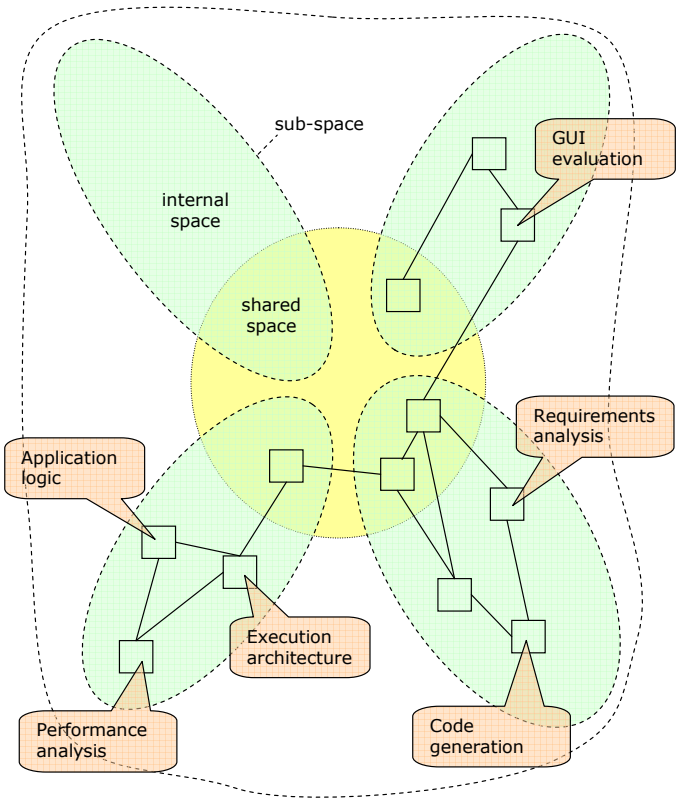

Figure 1.5: An impression of a model space.

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.