# Linking Model-Driven Development and Software Architecture: A Case Study

Anders Mattsson, *Member*, *IEEE*, Björn Lundell, *Member*, *IEEE*, Brian Lings, and Brian Fitzgerald

**Abstract**—A basic premise of model-driven development (MDD) is to capture all important design information in a set of formal or semiformal models, which are then automatically kept consistent by tools. The concept, however, is still relatively immature and there is little by way of empirically validated guidelines. In this paper, we report on the use of MDD on a significant real-world project over several years. Our research found the MDD approach to be deficient in terms of modeling architectural design rules. Furthermore, the current body of literature does not offer a satisfactory solution as to how architectural design rules should be modeled. As a result developers have to rely on time-consuming and error-prone manual practices to keep a system consistent with its architecture. To realize the full benefits of MDD, it is important to find ways of formalizing architectural design rules, which then allow automatic enforcement of the architecture on the system model. Without this, architectural enforcement will remain a bottleneck in large MDD projects.

**Index Terms**—Case study, model-driven development, software architecture.

## 1 Introduction

Model-driven development (MDD) [45] is an emerging discipline [4], [44] where the prevalent software-development practices in the industry are still immature [17]. The success of MDD in practice is currently an open question [17], and there is a lack of proven real-world usage of MDD in large industrial projects.

A basic premise of MDD is to capture all important design information in a set of formal or semiformal models that are automatically kept consistent by tools. This raises the level of abstraction at which the developers work, which can eliminate time-consuming and error-prone manual work in keeping different design artifacts consistent [17]. An important design artifact in any software development project, with the possible exception of very small projects, is the software architecture [6]. An important part of any architecture is the set of architectural design rules. We define architectural design rules as the rules, specified by the architect(s), that need to be followed in the detailed design of the system. The importance of architectural design rules has been further highlighted by recent research on software architecture [20], [21], [24], [26], [50], which has acknowledged that a primary role of the architecture is to capture the architectural design decisions. An important part of these design decisions consists of architectural design rules. In an MDD context, the design of the system is captured in models of the system; therefore, architectural design rules in an MDD context specify rules that the models of the system have to conform to. There is, however, no satisfactory solution in the current body of literature on how to model architectural design rules.

Given the above concerns—the lack of real-world rigorous validation of the MDD approach and the absence of guidelines as to how architectural design rules can be modeled—our research objective here was to investigate the application of MDD in a large project and to assess the extent to which architectural design rules could be smoothly integrated into the process.

This paper is organized as follows: In the next section, we review the literature on architectural design rules and, especially, research relevant to modeling architectural design rules and MDD. In Section 3, we present the research approach adopted in this study. In Section 4, we present the findings of a case study, and finally, in Section 5, we discuss our conclusions and implications for theory and practice.

## 2 Modeling Architectural Design Rules and MDD

In order to validate our research objective, we conducted a detailed review of the relevant literature, using the specific approach proposed in [32]:

- Keyword searches were done in IEEE Xplore, ACM Portal, ScienceDirect, SpringerLink, Inspec, CiteSeer, and Google Scholar.
- Having identified relevant articles, we performed backward and forward reference search and backward and forward author search.
- During the search, additional relevant keywords were used to refine the search process.

The primary goal of the review was to investigate the role of architectural design rules and how these rules can be

- *A. Mattsson is with Combitech AB, PO Box 1017, SE-551 11 Jönköping, Sweden. E-mail: anders.mattsson@combitech.se.*
- *B. Lundell and B. Lings are with the University of Skövde, PO Box 408, SE-541 28 Skövde, Sweden. E-mail: {bjorn.lundell, brian.lings}@his.se.*
- *B. Fitzgerald is with Lero-the Irish Software Engineering Research Centre, University of Limerick, Ireland. E-mail: bf@ul.ie.*

*Manuscript received 4 Feb. 2008; revised 9 June 2008; accepted 6 Aug. 2008; published online 15 Oct. 2008.*
*Recommended for acceptance by R. Taylor.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-02-0058.*
*Digital Object Identifier no. 10.1109/TSE.2008.87.*

modeled, especially in relation to MDD. We also wanted to find out what had been reported on architectural practices in large industrial projects and how this related to our case study. Overall, there are very few case studies on MDD in industrial projects, e.g., [3], [48], but, to the best of our knowledge, there are no case studies that illuminate architectural work practices in such projects. The main conclusions of this literature review were the following:

- Architectural design rules are important design artifacts for which there is no direct support in MDD.
- There is no satisfactory solution on how to model architectural design rules in the current body of literature.

We discuss these findings in more detail in the sections below.

### 2.1 The Role of Architectural Design Rules

The IEEE has established a set of recommended practices for the architectural description of software-intensive systems [19], which are followed by several architectural design methods:

- Attribute-Driven Design (ADD) [6], [53] developed at CMU/SEI,
- The Rational Unified Process (RUP) 4+1 views [25], [27] developed and commercialized by Rational Software, now IBM,
- The QASAR method [8], [9], [10] developed by the RISE research group at the University of Karlskrona/Ronneby,
- Siemens' 4 Views (S4V) method [18], [47], developed at Siemens Corporate Research,
- Business Architecture Process and Organization (BAPO/CAFCR) developed primarily at Philips Research [1], [52], and
- Architectural Separation of Concerns (ASC) [43] developed at Nokia Research.

The purpose of the architecture is to guide and control the design of the system so that it meets its quality requirements. Bass et al. [6] are unequivocal in stating the importance of an architectural approach:

> *The architecture serves as the blueprint for both the system and the project developing it. It defines the work assignments that must be carried out by design and implementation teams and it is the primary carrier of system qualities such as performance, modifiability, and security—none of which can be achieved without a unifying architectural vision. Architecture is an artefact for early analysis to make sure that the design approach will yield an acceptable system. Moreover, architecture holds the key to post-deployment system understanding, maintenance, and mining efforts. In short, architecture is the conceptual glue that holds every phase of the project together for all of its many stakeholders.*

A common understanding in architectural methods is that the architecture is represented as a set of components related to each other [42], [46]. The components can be organized into different views focusing on different aspects of the system. Different methods propose different views; typical views are a view showing the development structure (e.g., packages and classes), a view showing the runtime structure (processes and objects), and a view showing the resource usage (processors and devices). In any view, each component is specified with the following:

- an interface that documents how the component interacts with its environment,
- constraints and rules that have to be fulfilled in the design of the component,
- allocated functionality, and
- allocated requirements on quality attributes.

A typical method of decomposition (see, for instance, [6], [53], and [9]) is to select and combine a number of patterns that address the quality requirements of the system and use them to divide the functionality in the system into a number of elements. Child elements are recursively decomposed in the same way down to a level where no more decomposition is needed, as judged by the architect. The elements are then handed over to the designers who detail them to a level where they can be implemented. For common architectural patterns such as Model-View-Controller, Blackboard, or Layers [13], this typically means that you decompose your system into subsystems containing different kinds of classes (such as models, views, and controllers). However, the instantiation into actual classes is often left to the detailed design, for two main reasons:

1. Functionality will be added later, either because it was missed or because a new version of the system is developed, so more elements will be added later that also have to follow the design patterns decided by the architect.
2. It is not of architectural concern. The concern of the architect is that the design follows the selected architectural patterns and not to do the detailed design.

This means that a substantial part of the architecture consists of design rules on what kinds of elements, with behavioral and structural rules and constraints, there should be in a certain subsystem.

The importance of architectural design rules is also highlighted in current research in software architecture, which is focused on the treatment of architectural design decisions as first-class entities [20], [21], [24], [26], [50], where architectural design decisions impose rules and constraints on the design together with rationale. However, there is not yet any suggestion on how to formally model these design rules. The current suggestion is to capture them in text and to link them to the resulting design. This may be sufficient for rules stating the existence of elements ("ontocrisis" in [24]) in the design, such as a subsystem or an interface, since the architect can put the actual element (i.e., a certain subsystem) into the system model at the time of the decision. It is, however, not sufficient for rules on potentially existing elements ("diacrisis" in [24]), such as rules on what kinds of elements, with behavioral and structural rules and constraints, there should be in a certain subsystem, since the actual elements are not known at the time when the design decision is made. Instead, the rule-based design occurs later in the detailed design phase and involves other persons, potentially even in a different version of the system.

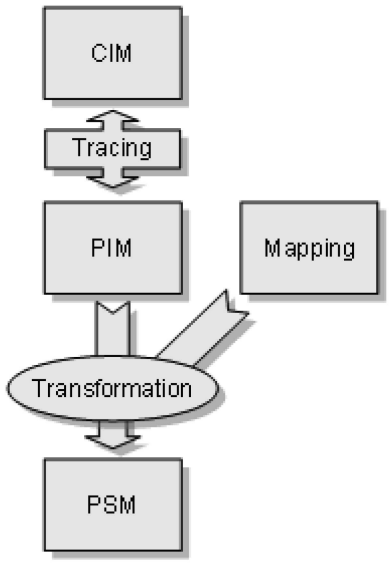



Fig. 1. An overview of MDA.

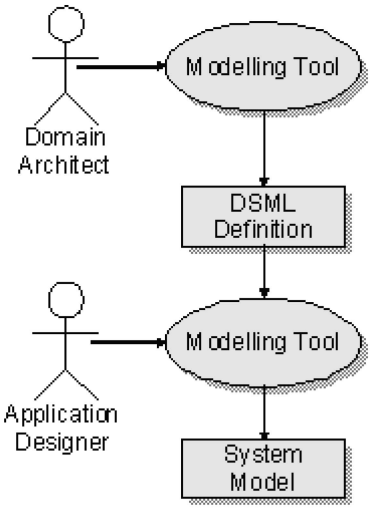



Fig. 2. Domain Specific Modeling.

### 2.2 MDD and Architectural Design Rules

The basic idea of MDD is to capture all important design information in a set of formal or semiformal models that are automatically kept consistent by tools. The purpose is to raise the level of abstraction at which the developers work and to eliminate time-consuming and error-prone manual work in keeping different design artifacts consistent [17].

MDD requires that the work products produced and used during development are captured in models to allow automation of noncreative tasks such as transformation of models into code or conformance checks between different design artifacts. There exist several approaches to MDD, such as OMG's MDA [40], Domain Specific Modeling (DSM) [22], [49], and Software Factories [16] from Microsoft.

MDA prescribes that three models or sets of models shall be developed as illustrated in Fig. 1:

1. The Computationally Independent Model(s) (CIM) captures the requirements of the system.
2. The Platform-Independent Model(s) (PIM) captures the systems functionality without considering any particular execution platform.
3. The Platform-Specific Model(s) (PSM) combines the specifications in the PIM with the details that specify how the system uses a particular type of platform. The PSM is a transformation of the PIM using a mapping either on the type level or at the instance level. A type-level mapping maps types of the PIM language to types of the PSM language. An instance-level mapping uses marks that represent concepts in the PSM (such as a process or a Corba object). When a PIM shall be deployed on a certain platform, the marks are applied to the elements of the PIM before the transformation.

MDA does not prescribe any particular language to be used for the models, but UML [41] is proposed as one possibility. There is also an accompanying OMG standard, Metaobject Facility (MOF), which can be used to describe metamodels for modeling languages. MDA does not directly address architectural design or how to represent the architecture, but the architecture has to be captured in the PIM or in the mapping since the CIM captures the requirements and the PSM is generated from the PIM using the mapping.

Another approach to MDD is DSM. The basic idea of DSM is that instead of using a general-purpose language such as UML to model your system you define and use a language that is specifically well suited to define systems in a narrow domain. In Fig. 2, the basic approach of DSM is illustrated. A DSM Language (DSML) is defined that captures the main concepts in the domain of an application. This DSML definition is then used as input into a language-configurable modeling tool in which the system is modeled in the DSML. Examples of this approach are described in [22] and [49].

The DSML is defined by an abstract syntax, a concrete syntax, semantics, a mapping between the abstract syntax and the concrete syntax, and a mapping between the abstract syntax and the semantics. The abstract syntax is typically defined in a model that defines the concepts of the domain, relationships, and integrity constraints. For this one can, for instance, use UML and OCL. Since this is a model that in turn defines elements of a language for a model, it is called a metamodel [2]. The concrete syntax of the language is comprised of the visual symbols that represent the concepts in the metamodel. To provide this and the mapping to the abstract syntax, one must specify symbols for the model elements in the metamodel. The semantics of the language define the meaning of the concepts in the abstract syntax. Defining semantics and mapping these to the abstract syntax can be done by providing translations of the metamodel into another language with defined semantics, such as a programming language. DSM can be seen as a special case of MDA where a domain-specific language is defined for use in the PIM.

Software factories is an MDD approach that incorporates the DSM idea but goes further since it provides a method for building complete customized tool chains for product

families using extensible tools. Such a tool chain is called a *software factory*.

Although neither the DSM nor the software factory approach directly address the problem of how to model architectural rules, it is interesting in that they allow you to naturally specify rules on the system model in the DSML definition. In fact, that is basically what the DSML definition is—a set of rules that have to be followed when building the system model. These rules are, however, not the architectural design rules, they are the rules of a domain-specific language.

### 2.3 Modeling Architectural Design Rules

There are a large number of Architectural Description Languages (ADL) [34], [35], [36], including UML, specified for describing the architecture of software systems. These typically allow one to specify components with relations and interfaces together with functional and structural constraints. They do not, however, provide any means to specify constraints or rules on groups of conceptual components only partly specified by the architect that are intended to be instantiated and detailed by designers. For instance, in the project reported on in this study, the architects needed to specify a set of rules on behavior and relations on a conceptual component called arcComponent without knowing which specific arcComponents would be relevant. Rather, they were to be identified and designed by the designers according to the rules stated by the architects.

The problem of modeling design rules is essentially the same problem as modeling the solution part of a design pattern since the solution specifies rules to follow in the design. There are a number of suggestions on how to formally model design pattern specifications [7], [14], [29], [33], [37]. They are, however, all limited in what kind of rules they can formalize, typically only structural rules. In addition, all approaches except [33] require the architect to use mathematical formalisms such as predicate logic and set theory that may be unfamiliar or hard to understand for both architects and developers.

There are also some approaches on how to model the usage of architectural design patterns or architectural styles in a system model, such as [38], [55]. However, they only address the problem of how to show that an architectural rule has been followed and not the problem of how to specify the rule.

## 3 Research Approach

Much of the research on the application of software methods to date has relied on postal surveys to investigate factors using statistical techniques. This is undoubtedly useful, but it is often beneficial to complement this with a "thick" description, which provides a more detailed and nuanced description of the factors at play in a particular context.

This research is based on experience at Combitech AB (www.combitech.se), a Swedish supplier of services within system development, system integration, information security, and system safety. Combitech, a wholly owned subsidiary of Saab AB, can be found in 20 locations in Sweden and the company has more than 800 employees. Customers are primarily from the defense and telecommunications industry, as well as government offices and authorities responsible for infrastructure in society.

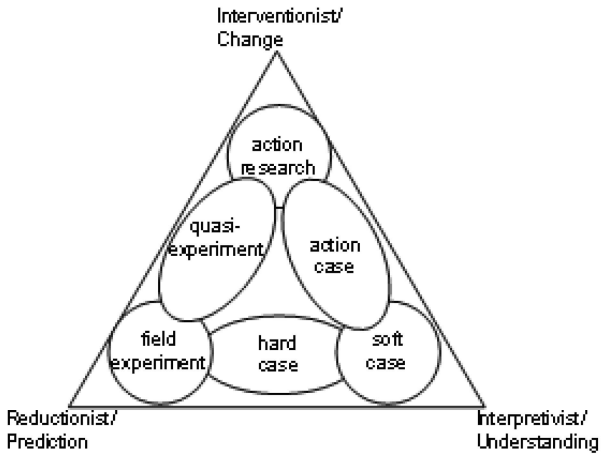



Fig. 3. A framework for integrating research perspectives (based on Braa and Vidgen [12]).

Braa and Vidgen [11] propose a useful framework (Fig. 3) integrating positivist, interpretivist, and critical research approaches. Briefly summarizing, in Fig. 3, the apexes of the triangle represent the different perspectives and outcomes of the research. Thus, from the positivist perspective, a reductionist approach would be followed to produce the desired outcome, which is that of prediction. From the interpretivist perspective, on the other hand, the primary motivation would be that of understanding, while, from the critical interventionist perspective, the motivation would be one of change.

Given the lack of research to date on the application of MDD in real industrial projects and, more specifically, the modeling of architectural design rules, this study was concerned with achieving an increased understanding of this process. Bearing this in mind, an interpretivist approach that sought to inductively develop a richer understanding based on a deep analysis of a single case was deemed appropriate. Also, as it represents uncharted territory to a large extent, the study was also motivated by an interventionist desire to achieve successful change in this real organizational problem given the lack of any road map documenting how this can be successfully achieved.

Given these objectives, a hybrid of the interventionist/change and interpretivist/understanding perspectives was appropriate. Braa and Vidgen locate a number of hybrid research approaches where a mixture of perspectives is motivating the research, and in the case of a mixture of interventionist/change and interpretivist/understanding perspectives, as in this study, the Action Case approach is deemed appropriate. As can be seen in Fig. 3, the Action Case approach is a hybrid of the Soft Case and Action Research approaches, each of which is discussed in turn below.

In Soft Case research, an interpretivist approach is adopted. The concern is more with gaining understanding and insight [51]. It is our belief that, in this area where little exists by way of successful exemplars, the appropriate approach is an in-depth study that a single case provides,

what has been termed the "revelatory case" [54]. A single-case strategy is also strongly recommended by Mintzberg, who poses the very apt question "what, for example, is wrong with samples of one?" [39]. One of the limitations of this study might appear to be the fact that it is based on a single case and, thus, there is limited scope for generalization. However, Lee and Baskerville [30] identify a fundamental and long-standing problem with the type of generalization based on the type of statistical sampling frequently sought in research, namely, the problem of attempting to generalize to any other settings beyond the current one. Following this conventional model, researchers have suggested increasing the sample size or the number of case study organizations, but Baskerville and Lee argue cogently for the ultimate futility of this flawed strategy.

Action research originates from the work of Lewin [31], and several "flavors" have emerged. At heart, however, there is general agreement on a number of essential characteristics: It is a highly participative approach, which implies a close intertwining between researchers and practitioners intervening on real problems in real contexts, with two primary outcomes: an action outcome in terms of a (hopefully) beneficial intervention in the organization and a research outcome in terms of a contribution to research on the phenomenon in question. It is also a longitudinal cyclical process of intervention and reflection, with any learning fed back in successive action research cycles. e.g., [5], [23], [28].

These characteristics were very much present in this research: The primary author is currently the lead engineer in MDD and software architecture at Combitech. He has 18 years of experience with development of embedded real-time systems from a wide range of organizations in the automotive, defense, medical, telecom, and automation industries. In the last 13 of these years, he has alternated between the roles of a software architect and a mentor in software architecture and MDD in several projects. The research reported in this paper was motivated by problems in architectural work practices experienced in these projects.

The overall objective of this research was to identify potential improvements to architectural practices in an MDD context based on tensions between MDD practices and architectural practices as revealed in an industrial case where architectural enforcement was an important issue. That is, a case where there were a relatively large number of developers (more than 20) that had to follow an unfamiliar architecture. Questions to be answered by this research were:

- How were the rules documented?
- How were they communicated and enforced?
- How did the actual work practices regarding architectural design rules affect the overall development process?

The research focused on architectural practices in a development project that was executed over a two-year period. The analysis is based on a rich set of system and project documentation, collected from the configuration management tool used by the project, including

- the architecture design rule document and the system model revealing how the architecture was documented and how the detailed design was done,
- documents defining the development process of the project,
- architecture review protocols with comments and actions revealing problems in the interpretation of the architecture and the effort to correct them (there were 120 review protocols with an average of 13 remarks in each protocol), and
- project plans and progress reports showing the planned and actual effort for activities such as architectural reviews and rework. (These also contain a number of metrics such as how many modules there were at each defined module status at the time of the progress report. One such metric was how many modules there were waiting for architectural review. The progress reports covered the construction phase with an interval of approximately one week.)

Further, the primary author has drawn from experience related to participation in the project being investigated where he had the overall responsibility for work practices and tools. He has also drawn from experiences gained through participation in several other MDD projects, where he has either acted as an architect or as a mentor to the architect(s).

## 4 Integrating Architectural Design Rules in an MDD Project

This section describes the research findings, beginning with a description of the research context.

### 4.1 Action-Case Context

At the start of the project, Combitech faced the challenging task of developing a software platform for a new generation of digital TV set-top boxes for the Digital Video Broadcasting (DVB) standard. The development had to be done in 18 months under strict quality requirements and on a completely new customized hardware platform developed in parallel by another company. At the time of the project, Combitech was a Swedish consultancy company specializing in services for developing embedded real-time systems. Combitech had approximately 250 consultants of whom about 75 where involved in the project. The total effort of the project was 100+ person years expended over a 24-month period, with the first delivery after 18 months and two more deliveries with additional functionality and corrections during the last six months. The project was distributed across five sites in the south of Sweden where the geographical distances between sites ranged between 1.5 and 4 hours one-way travel by car. The two architects were stationed in the same site, which meant that the other sites had to rely heavily on remote architectural support.

Combitech developed the platform as a phased fixed-price (price negotiated for each phase) assignment for a customer company. Combitech had full responsibility for the software development, but the customer wanted control of the architecture of the software, so the architecture team was actually managed by the customer and not by Combitech, although they were Combitech employees. This meant that there was a need for a counterpart on the Combitech side, making sure that the architecture also was

feasible within the budget. This was the project role of the first author of this paper: technically responsible within the Combitech project management team, with prime responsibility to negotiate the architecture with the architecture team on the customer side.

### 4.2 Project Challenges

The project faced a number of challenges:

- There was a short time to market. Since Combitech was in competition with other developers, the product had to be ready at a fixed date.
- Since the hardware was to be developed in parallel with the software, there would be little time to integrate the two, leading to a significant risk of errors and misunderstandings that would have to be handled by very late changes in the software. Therefore, Combitech needed to test the software on a range of platforms, from a host PC to real target hardware with several intermediate hardware platforms.
- The requirements were a moving target where the initial requirement specifications would be overridden by acceptance tests delivered late in the project.
- The maintenance phase would be lengthy and had to be very cost-effective.
- There was a requirement to be able to differentiate the product, releasing different variants for different markets. New variants had to be developed and maintained efficiently.
- Products would be competing on performance and quality; the product with the best performance and quality would win the final contract.

### 4.3 Rationale for Choosing MDD

Combitech had experience from maintenance on the previous generation of the product, which had been developed by another company. The product existed in many different variants for different markets, so Combitech was confident that this would be the fact also for the new product. Combitech saw a potential to make the maintenance of the product a lot more efficient by building it according to a product-line approach.

Within Combitech, there was also extensive experience of working with models in UML and preceding modeling languages such as OMT, Booch, Coad-Yourdon, and Objectory, for both analysis and design models. Combitech also had experience of using rule-based transformation from design models directly into code that executed on a platform. However, in real projects, Combitech had so far only executed the transformations manually, although experimentation with automatic code generation had been done to a degree where the company felt ready to apply it in a real project. Given this experience, there was conviction among the project management team that MDD would help address the challenges of the project by making the team more efficient, agile, and flexible regarding the hardware platform:

- **Efficiency.** MDD would eliminate the manual and error-prone step of implementing the UML models.
- **Agility.** The approach would make it possible to work in an agile way where one could quickly go from requirements to tested implementation without having to skip documentation, something very important for the maintainability of the product.
- **Flexibility in HW platforms.** MDD would make it possible to test most of the code without access to the actual hardware by simply generating code for different platforms as the project gained access to hardware that became increasingly similar to the final target.

### 4.4 Tool Selection

Given the tight deadline, an out-of-the-box tool solution was required that would give the following:

- modeling in standard UML, to minimize the need for training,
- generation of code with good performance on the target platform, the host platform, and the platform for the previous generation of set-top boxes since this would be used as an intermediate test platform,
- 100 percent of the developed code generated from the model (to avoid synchronization problems with code and model) having at the same time the ability to use pure C++ code where there was a need (to eliminate a potential risk of not being able to do everything possible in the traditional way),
- the ability to debug at the model level,
- support for distributed team working, and
- a high probability that the vendor would continuously improve the tool toward the requirements of embedded real-time system development.

After a brief evaluation, Rhapsody from Ilogix was selected as the only tool that seemed to satisfy all these requirements.

The selection of Rhapsody meant that the system was designed as a UML model with action code in C++. This model (the system model) was then automatically converted to full production code in C++ by the tool. The generated code uses an execution framework (OXF), provided by the tool, to abstract out the target execution platform. In terms of MDA [40], the model built in Rhapsody corresponds most closely to the PIM, where OXF and the generated code corresponds to the PSM.

### 4.5 The Process

The project followed a phased process similar to the RUP model according to Fig. 4.

The main architectural work was done during the Elaboration phase for six months by the two architects. The Construction phase was started with a workshop where the architecture was presented for the design teams. The Construction phase was then performed by approximately 50 developers divided into seven teams for 12 months. To ensure that the design corresponded to the architecture, the overall design of each component was reviewed by the architecture team before detailed design of the component was allowed to start. Since there were 166 components, this work occupied the architects almost full time during this phase.

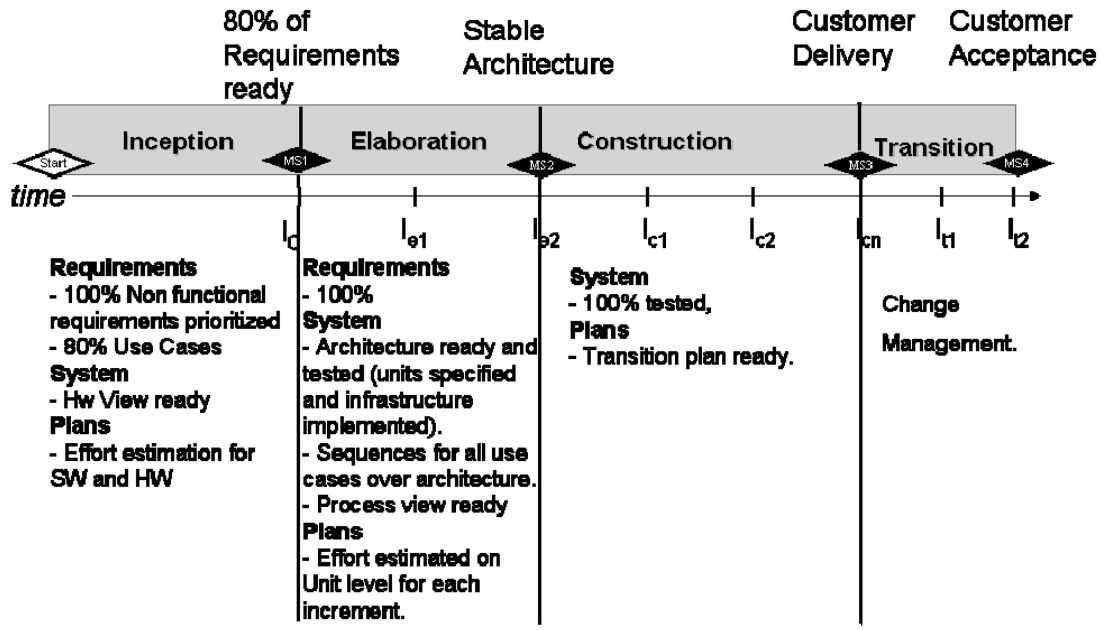



Fig. 4. The phases of the project.

### 4.6 Capturing the Architecture

To be able to meet the deadline, about 50 developers were assigned to the project after six months of architectural work undertaken by the architecture team. To be efficient, they had to be able to work as independently of each other as possible. This required a stable architecture to be developed during these first six months before the project scaled up. A product-line architecture approach [9] was selected to address the requirements for efficient development and maintenance of product variants. In addition to this, there were other important quality requirements such as portability (it was anticipated that the software would outlive the hardware) and overall performance, which had to be handled by the architecture. So, an appropriate architecture was fundamental to the success of the project.

One of the first problems to face was how to represent the architecture when working in an MDD context. A basic idea in MDD is to use models instead of documents to represent the requirements and design of a system and to generate the implementation code from these models. The traditional way of representing the architecture is in a document that guides and constrains the detailed design. In model-oriented processes like RUP [25], where models have replaced requirement specifications and design descriptions, one still represents the architecture in a document. The aim of the project was, insofar as possible, to automatically connect the architecture to the design, thereby minimizing both the maintenance problem and the effort to enforce the architecture in the design. In the end, the project management team settled for the following approach:

- The high-level structure was to be captured in the system model as UML packages.
- Architectural design rules were to be captured in natural language in a text document supported by a UML-class framework in the system model.
- Example components would be designed in a package in the system model illustrating how to follow the architectural rules.

#### 4.6.1 High-Level Structure

The high-level partitioning of the system, down to a level at which individual components were to be developed by individual developers, was captured in a package hierarchy populated with classes acting as facades [15] for the actual components. The system was divided into a number of layers according to Fig. 5:

- Hardware Abstraction Layer (HAL), delivered by the hardware manufacturer,
- Board Support Package (BSP), delivered by the RTOS vendor,
- Real-Time Operating System (RTOS), which was the commercially available RTOS VxWorks,

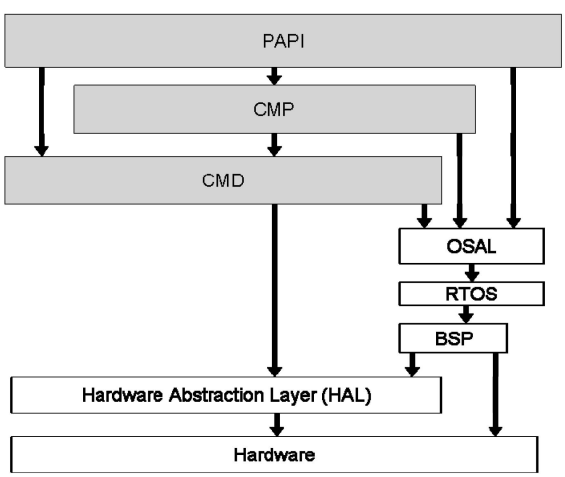



Fig. 5. The layers of the system.

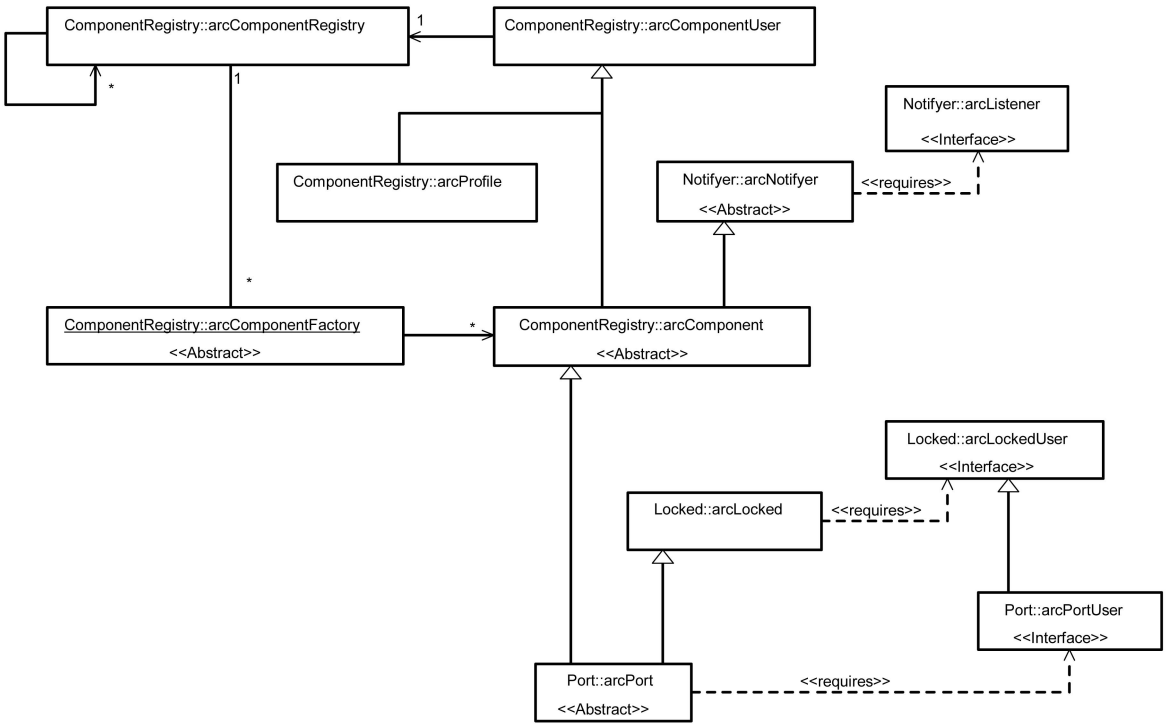



Fig. 6. Executable framework of the architecture.

- RTOS Abstraction (OSAL), a wrapper for the RTOS delivered together with the modeling tool,
- Common Media Drivers (CMDs), a set of hardware driver components delivered by the project,
- Common Media Platform (CMP), a set of components reusable on several hardware platforms and for several DVB standards (this was developed in the project), and
- Proprietary API (PAPI), the adaptation layer for a specific DVB standard (these components were developed in the project).

Each component was modeled as a package that was to contain the classes that realized the component. Each of these packages was placed in a package representing one of the layers PAPI, CMP, or CMD. The architects modeled the system down to the component packages; then, it was the job of the component designer to define the classes inside the component package.

### 4.6.2 Architectural Design Rules

A number of patterns and rules were to be followed when the components in the model were designed. To support these patterns and rules, an architecture package with an executable framework was developed as a part of the system model. Fig. 6 shows a UML class diagram, taken from the system model, showing the classes in this framework. The classes are described with the following text in the architecture document:

- **arcComponentRegistry.** The main task for the arcComponentRegistry is to be the resource provider for and of all arcComponents in a layer. The system's layer dependencies are maintained via its access to other arcComponentRegistrys. This is basically the dispatcher in a *Client-Dispatcher-Server* pattern; see [13].
- **arcComponentUser.** Being an arcComponentUser is the only way of retrieving another component. This is basically the client in a *Client-Dispatcher-Server* pattern; see [13].
- **arcComponent.** This is the basic logical architectural building block. This is basically the server in a *Client-Dispatcher-Server* pattern; see [13].
- **arcComponentFactory.** The arcComponentFactory implements the instantiation rules of arcComponents. There is at least one of these for each concrete specialization of arcComponent in the system. All arcComponentFactories are encapsulated by an arcComponentRegistry.
- **arcProfile.** arcProfile is the "root" of the system configuration and could be viewed as an instance of the reference architecture.
- **arcNotifyer.** The arcNotifyer notifies its attached arcListeners of important events. This is basically the publisher in a *Publisher-Subscriber* pattern; see [13].
- **arcListener.** The arcListener registers to an arcNotifyer to become notified of important events. This is basically the subscriber in a *Publisher-Subscriber* pattern; see [13].
- **arcLocked.** This is a basic resource locking mechanism (semaphore type). It is primarily intended for transaction locks on shared resources.
- **arcLockedUser.** This is the interface required by an arcLocked component.
- **arcPort.** An arcPort is a system resource locked to one user at a time. The main purpose of a port is to transmit data to and/or from its user.

- **arcPortUser.** This is the interface required by an arcPort component.

In principle, the framework contained abstract base classes representing the core abstractions of the system, relations between them, and operations that were to be overridden in specializations of the base classes. The framework also contained full implementations of some basic mechanisms that operated solely on the level of the abstract base classes, such as interprocess resource locking and component registry handling (registration, allocation, and deallocation).

Unfortunately, the project could not fully capture the architecture in a formal model. It was necessary to use natural language to express the rules on how to use the architectural framework to design the components in the architecture. There were more than 60 rules that had to be followed. Below is a small representative subset of these rules taken directly from the architecture rule document:

- "Any *arcComponent* with behavior similar to an *arcPort* should be a specialization of *arcPort*."
- "All specializations of *arcPort* may have one overloaded method for each of the methods Open(), Close(), and Write()."
- "All specializations of *arcPort* may have several methods for the method Ctrl(). These methods shall be named as ctrl_<specific_name> and may not change the parameter list of the base class, except for specializations of the parameter classes given for the base class. However, a method may omit the second (parameters) parameter."
- "*arcPort*::Write() shall be used to stream data to a port's data output stream."
- "*arcPort*::Ctrl() shall be used to control and manipulate a port's properties."
- "All specializations of *arcPort* must use its parent's implementation of the base-class method for their respective purposes. "
- "All specializations of *arcPort* require a specialization of the *arcPortUser* interface."
- "All specializations of the *arcPortUser* interface base class may have one overloaded method for each of the methods RxReady(), TxDone(), and GetMem()."
- "All specializations of the *arcPortUser* interface base class shall have one overloaded CtrlAck() method for each of the asynchronous ctrl_<name> methods."

#### 4.6.3 Providing Example Components

To guide the developers in how to develop the components using the architectural framework, a few example components were also developed by the architects as a part of the model:

- a component showing how to realize the "pipes and filter" pattern,
- a component showing how to use interrupts, and
- a component showing how to design a "port" component, which is a specialization of the arcPort component (this example was, however, never completed).

In addition to showing the design, the examples also showed how to use different diagrams to capture the design.

### 4.7 Manual Reviews of Architectural Conformance

Using natural language to describe architectural design rules meant that the project had to rely heavily on manual reviews to enforce conformance with the architecture. Performing these reviews was a heavy burden for the architects; it took almost all of their time during the first 12 months of development after the first release of the architecture.

The rules proved to be ambiguous and hard to comprehend and thus prone to different interpretations. Several developers reported having a hard time trying to understand and follow all of the detailed rules. This was manifested by the fact that major corrections were frequently needed after reviews, as shown by the review protocols. Sometimes, this meant that a lot of reworking had to be done since reviews were often held when design had continued too long. This was caused by work overload on the architects, which in turn was caused by a lot of effort expended on reviewing the designs generated by the different teams. The progress reports show that it was common that architectural reviews were actually done after a component was completed and tested, which was in violation of the rules that stated that the component had to pass the review before the detailed design was done.

## 5 Conclusions

Architectural design rules are an important part of the architecture, and there are no suggestions on how to model them in the current body of literature. The inability to formalize the architectural design rules leads to a need for manual enforcement of them. The research presented here shows that this is an error-prone and time-consuming task that takes most of the effort of the architects during the construction-intensive phases of a project. This problem exists in traditional document-based development, as well as in MDD, but it is more apparent and acute in MDD. This is because MDD has been able to automate the step from detailed design to implementation, eliminating time-consuming coding and code reviews. However, MDD has not been able to automate enforcement of the architecture on the detailed design due to the inability to model architectural design rules. The presented case shows that the inability to model architectural design rules makes architectural enforcement a bottleneck in MDD projects. This leads to a plethora of problems, including the following:

- **Stalled detailed design.** The design teams have to wait for the architects to review their overall design before they can dig deeper into the design.
- **Premature detailed design.** Design teams start detailing their design before their overall design is approved by the architect, with the risk that they will have to redo much work after the review.
- **Low review quality.** The reviews are of low quality, leading to problems later in the project.

- **Poor communication of the architecture.** The architects have no time to handle the communication with the design teams regarding architectural interpretations or problems; problems are "swept under the carpet."

The implications for theory are that there is a need for further research to find ways of modeling architectural design rules in such a way that they are both amenable to automatic enforcement on the detailed design and easy to understand and use by both architects and developers. The implications for practice are that until there is support for automatic enforcement of architectural design rules, extra resources are needed for architectural reviews. This has to be taken into account in the planning and in the architectural work practices. Although the architectural design is inherently a task for a relatively small group, it should be possible to delegate the architectural reviews to a larger group. This would give time for the architects to concentrate on the core architectural tasks: designing, communicating, and maintaining the architecture.

## ACKNOWLEDGMENTS

This research has been financially supported by the ITEA project Co-development using inner and Open source in Software Intensive products (COSI) (www.itea-cosi.org) through Vinnova (www.vinnova.se) and also by funding from the Science Foundation Ireland (www.sfi.ie).

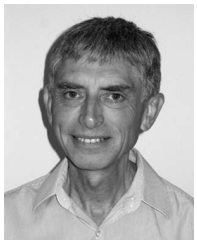
## REFERENCES

[1] P. America, E. Rommes, and H. Obbink, *Multi-View Variation Modeling for Scenario Analysis,* 2004.
[2] C. Atkinson and T. Kuhne, "Model-Driven Development: A Metamodeling Foundation," *IEEE Software,* vol. 20, no. 5, pp. 36-41, Sept.-Oct. 2003.
[3] P. Baker, L. Shiou, and F. Weil, "Model-Driven Engineering in a Large Industrial Context—Motorola Case Study," *Proc. Eighth Int'l Conf. Model Driven Eng. Languages and Systems,* pp. 476-491, 2005.
[4] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema, "Developing Applications Using Model-Driven Design Environments," *Computer,* vol. 39, no. 2, pp. 33-40, Feb. 2006.
[5] R.L. Baskerville, "Investigating Information Systems with Action Research," *Comm. AIS,* vol. 2, p. 4, 1999.
[6] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice,* second ed. Addison-Wesley, 2003.
[7] I. Bayley, "Formalising Design Patterns in Predicate Logic," *Proc. Fifth IEEE Int'l Conf. Software Eng. and Formal Methods,* pp. 25-36, 2007.
[8] P. Bengtsson and J. Bosch, "Scenario-Based Software Architecture Reengineering," *Proc. Fifth Int'l Conf. Software Reuse,* pp. 308-317, 1998.
[9] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach.* Addison-Wesley, 2000.
[10] J. Bosch and P. Molin, "Software Architecture Design: Evaluation and Transformation," *Proc. IEEE Conf. and Workshop Eng. of Computer-Based Systems,* pp. 4-10, 1999.
[11] K. Braa and R. Vidgen, "Interpretation, Intervention, and Reduction in the Organizational Laboratory: A Framework for In-Context Information System Research," *Accounting, Management and Information Technologies,* vol. 9, pp. 25-47, Jan. 1999.
[12] K. Braa and R. Vidgen, "Research: From Observation to Intervention," *Planet Internet,* K. Braa, C. Sørenssen, and B. Dahlbom, eds., pp. 251-276, Studentlitteratur, 2000.
[13] F. Buschmann, *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.
[14] A.H. Eden, "A Theory of Object-Oriented Design," *Information Systems Frontiers,* vol. 4, pp. 379-391, 2002.
[15] E. Gamma, *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[16] J. Greenfield and K. Short, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* John Wiley & Sons, 2004.
[17] B. Hailpern and P. Tarr, "Model-Driven Development: The Good, the Bad, and the Ugly," *IBM Systems J.,* vol. 45, pp. 451-461, 2006.
[18] C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture.* Addison-Wesley, 2000.
[19] *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems.* IEEE, 2000.
[20] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," *Proc. Fifth Working IEEE/IFIP Conf. Software Architecture,* pp. 109-120, 2005.
[21] A. Jansen, J. van der Ven, P. Avgeriou, and D.K. Hammer, "Tool Support for Architectural Decisions," *Proc. Sixth Working IEEE/IFIP Conf. Software Architecture,* pp. 44-53, 2007.
[22] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-Integrated Development of Embedded Software," *Proc. IEEE,* vol. 91, pp. 145-164, 2003.
[23] N.F. Kock, R.J. McQueen, and J.L. Scott, "Can Action Research Be Made More Rigorous in a Positivist Sense? The Contribution of an Iterative Approach," *J. Systems and Information Technology,* vol. 1, pp. 1-24, 1997.
[24] P. Kruchten, "An Ontology of Architectural Design Decisions in Software Intensive Systems," *Proc. Second Groningen Workshop Software Variability,* pp. 54-61, 2004.
[25] P. Kruchten, *The Rational Unified Process: An Introduction,* third ed. Addison-Wesley, 2004.
[26] P. Kruchten, P. Lago, and H. van Vliet, "Building Up and Reasoning about Architectural Knowledge," *Proc. Second Int'l Conf. Quality of Software Architectures,* pp. 43-58, 2006.
[27] P.B. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software,* vol. 12, no. 6, pp. 42-50, Nov. 1995.
[28] F. Lau, "A Review on the Use of Action Research in Information Systems Studies," *Proc. IFIP TC8 WG 8.2 Int'l Conf. Information Systems and Qualitative Research,* A.S. Lee, J. Liebenau, and J.I. DeGross, eds., pp. 31-68, 1997.
[29] A. Lauder and S. Kent, "Precise Visual Specification of Design Patterns," *Proc. 12th European Conf. Object-Oriented Programming,* 1998.
[30] A.S. Lee and R.L. Baskerville, "Generalizing Generalizability in Information Systems Research," *Information Systems Research,* vol. 14, pp. 221-243, Sept. 2003.
[31] K. Lewin, *Resolving Social Conflicts: Selected Papers on Group Dynamics.* Harper & Row, 1948.
[32] Y. Levy and T.J. Ellis, "A Systems Approach to Conduct an Effective Literature Review in Support of Information Systems Research," *Informing Science J.,* vol. 9, pp. 181-212, 2006.
[33] J.K.H. Mak, C.S.T. Choy, and D.P.K. Lun, "Precise Modeling of Design Patterns in UML," *Proc.. 26th Int'l Conf. Software Eng.,* pp. 252-261, 2004.
[34] N. Medvidovic, E.M. Dashofy, and R.N. Taylor, "Moving Architectural Description from under the Technology Lamppost," *Information and Software Technology,* vol. 49, pp. 12-31, 2007.
[35] N. Medvidovic, D.S. Rosenblum, D.F. Redmiles, and J.E. Robbins, "Modeling Software Architectures in the Unified Modeling Language," *ACM Trans. Software Eng. and Methodologies,* vol. 11, pp. 2-57, 2002.
[36] N. Medvidovic and R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Software Eng.,* vol. 26, no. 1, pp. 70-93, Jan. 2000.
[37] T. Mikkonen, "Formalizing Design Patterns," *Proc. 20th Int'l Conf. Software Eng.,* pp. 115-124, 1998.
[38] T. Mikkonen, R. Pitkanen, and M. Pussinen, "On the Role of Architectural Style in Model Driven Development," *Proc. First European Workshop Software Architecture,* pp. 74-87, 2004.
[39] H. Mintzberg, "An Emerging Strategy of "Direct" Research," *Administrative Sciences Quarterly,* vol. 24, pp. 582-589, Dec. 1979.
[40] *MDA Guide Version 1.0.1,* OMG, omg/2003-01-06, 2003.
[41] *Unified Modeling Language: Superstructure,* OMG, formal/2007-02-05, 2007.
[42] D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Eng. Notes,* vol. 17, pp. 40-52, 1992.

[43] A. Ran, "ARES Conceptual Framework for Software Architecture," *Software Architecture for Product Families: Principles and Practice,* M. Jazayeri, A. Ran, and F. van der Linden, eds., pp. 1-29, Addison-Wesley, 2000.
[44] D.C. Schmidt, "Model-Driven Engineering," *Computer,* vol. 39, no. 2, pp. 25-31, Feb. 2006.
[45] B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Software,* vol. 20, no. 5, pp. 19-25, Sept./Oct. 2003.
[46] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. Software Eng.,* vol. 21, pp. 314-335, 1995.
[47] D. Soni, R.L. Nord, and C. Hofmeister, "Software Architecture in Industrial Applications," *Proc. 17th Int'l Conf. Software Eng.,* pp. 196-207, 1995.
[48] M. Staron, "Adopting Model Driven Software Development in Industry—A Case Study at Two Companies," *Proc. Ninth Int'l Conf. Model Driven Eng. Languages and Systems,* pp. 57-72, 2006.
[49] J.P. Tolvanen and S. Kelly, "Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences," *Proc. Ninth Int'l Conf. Software Product Lines,* pp. 198-209, 2005.
[50] J. Tyree and A. Akerman, "Architecture Decisions: Demystifying Architecture," *IEEE Software,* vol. 22, pp. 19-27, 2005.
[51] G. Walsham, *Interpreting Information Systems in Organizations.* John Wiley & Sons, 1993.
[52] F. van der Linden, J. Bosch, E. Kamsties, K. Kansala, and H. Obbink, "Software Product Family Evaluation," *Proc. Third Int'l Conf. Software Product Lines,* pp. 110-129, 2004.
[53] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R.L. Nord, and B. Wood, "Attribute-Driven Design (ADD), Version 2.0," Technical Report CMU/SEI-2006-TR-023, Software Eng. Inst., Carnegie Mellon Univ., Nov. 2006.
[54] R.K. Yin, *Case Study Research: Design and Methods,* second ed. Sage Publications, 1994.
[55] U. Zdun and P. Avgeriou, "Modeling Architectural Patterns Using Architectural Primitives," *Proc. 20th Ann. ACM SIGPLAN Conf. Object Oriented Programming, Systems, Languages, and Applications,* 2005.

**Anders Mattsson** received the MSc degree from the Chalmers University of Technology, Sweden, in 1989. He then worked for two years as a software designer at Volvo Data AB, followed by one year as a system designer at Saab Instruments AB. Since then, he has been with Combitech AB, where he currently serves as the lead engineer in software architecture and model-driven development. His research interests include software architecture and model-driven development in the context of embedded real-time systems. He is a member of the IEEE and the IEEE Computer Society.

**Björn Lundell** received the PhD degree from the University of Exeter in 2001. He has been a staff member at the University of Skövde since 1984. He has been researching the open source phenomenon for a number of years. He co-led a work package in the EU FP6 CALIBRE project from 2004 to 2006. He is currently the technical manager in the industrial (ITEA) research project COSI (2005-2008), involving heterogeneous distributed development and analysis of the adoption of open source practices within companies. His research is reported in more than 50 publications in a variety of international journals and conference proceedings. He is a founding member of the IFIP Working Group 2.13 on Open Source Software and the founding chair of Open Source Sweden, an industry association established by Swedish open source companies. He is the organizer of the Fifth International Conference of Open Source Systems (OSS 2009), which is to be held in Skövde, Sweden. In addition, his research has also included fundamental research on evaluation and associated method support. He is a member of the IEEE and the IEEE Computer Society.



**Brian Lings** was with the University of Queensland, Australia, for a number of years. He then joined the Department of Computer Science at the University of Exeter, becoming its first elected head of department. He is now a member of the academic staff of the University of Skövde, Sweden, and the information systems director at Certus Technology Associates, United Kingdom. Certus Technology Associates uses an open source tool chain for the model-driven development of software product lines. He chaired the steering group of the British National Conference on Databases for three years, from 2004 until 2006.

**Brian Fitzgerald** holds an endowed professorship, the Frederick A Krehbiel II Chair in Innovation in Global Business and Technology, at the University of Limerick, Ireland, where he is also the research leader for global software development at Lero—the Irish Software Engineering Research Centre. His publications include 10 books and more than 100 papers in leading international journals and conference proceedings. Having worked in industry prior to taking up an academic position, he has more than 20 years experience in the software field.