Communicating Architectural Design Rules Using Models – An Action Case Study

Anders Mattsson
Lero—The Irish Software Engineering Research Centre, University of Limerick, Ireland, and Husqvarna AB, Huskvarna, Sweden

anders.mattsson@husqvarna.se

Björn Lundell
University of Skövde, Skövde, Sweden

bjorn.lundell@his.se

Brian Fitzgerald
Lero—The Irish Software Engineering Research Centre, University of Limerick, Ireland

brian.fitzgerald@ul.ie

Abstract

An important purpose of architectural design is to ensure that the system meets its quality requirements by defining a set of system wide design decisions. An important part of these design decisions is the set of architectural design rules that shall be followed by developers in the detailed design. The state of practice is to define these rules in natural language and to use manual reviews to enforce them. This way of transferring the rules to the developers is however error prone and requires a lot of effort from the architects since natural language is ambiguous and open for different interpretations and rule following have to be checked with manual reviews. This paper reports from an action case study where a novel approach for architectural modeling and automated conformance checking has been investigated regarding its ability to better communicate architectural design decisions to the developers. The findings indicate that the novel approach is significantly more effective than the state of practice. The findings also show that an important reason for this is that using a tool for conformance checking allows the developers to learn the rules by experimenting.

Keywords: Action Case study, Meta-modeling, Model-Driven Development, Software Architecture.

Introduction

An important design artifact in any software development project is the software architecture (Bass, Clements, & Kazman, 2003) consisting of system wide design decisions to be followed in the detailed design. The purpose of the architecture is to guide and control the design of the system so that it meets its

This paper was accepted through blinded executive review

quality requirements. Architectural design rules are defined by the architect as a part of the architecture and have to be followed by the developers in their detailed design of the system. A common way of capturing the architecture is to define a high level structure of the system as a set of subsystems with defined interfaces, together with a framework implementing a communication infrastructure used by the components of the subsystems (Mattsson, Lundell, Lings, & Fitzgerald, 2009). This is however not enough; you also need to specify rules as to what kinds of component to put in different subsystems and how the components are supposed to use the infrastructure. These rules cannot be handled by simply putting the components into the subsystems since the rules span more than the current version of the system, so more elements will be added later that also have to follow the rules. Examples of such rules, taken from the embedded software domain, are:

- *"A sensor shall be defined in the Sensors package."*
- *"A sensor may inherit Infrastructure::Observer to be able to react to changes in Data_Items, for instance to activate or deactivate itself."*
- *"A sensor may only have associations to In_Port_Ifc and Data_Items. These associations shall only be navigable from the sensor."*
- *"A sensor may not have any public operations or attributes."*

Architectural design rules have been recognized as an important part of the architecture from the early works on software architecture by Perry and Wolf (Perry & Wolf, 1992) to recent research in software architecture. In the work of Perry and Wolf architectural design rules are represented as form in the formula: Software Architecture = {Elements, Form, Rationale}, where form represents constraints on choice of elements and relationships between elements. In recent research focused on the treatment of architectural design decisions as first class entities (Jansen & Bosch, 2005; Jansen, van der Ven, Avgeriou, & Hammer, 2007; Kruchten, 2004a; Kruchten, Lago, & van Vliet, 2006; Tyree & Akerman, 2005), architectural design decisions impose rules and constraints on the design together with rationale. Identification of design rules are also typically an activity in methods for architectural design as elaborated below:

- In ADD (Bass et al., 2003; Wojcik et al., 2006) architectural design rules are represented as responsibilities and design constraints.
- In RUP 4+1 Views (Kruchten, 1995, 2004b) architectural design rules are represented as design guidelines.
- In QASAR (Bengtsson & Bosch, 1998; Bosch, 2000; Bosch & Molin, 1999) architectural design rules are represented bas rules and constraints.
- In S4V (Hofmeister, Nord, & Soni, 2000; Soni, Nord, & Hofmeister, 1995) architectural design rules are represented as design guidelines and design strategies.
- In ASC (Ran, 2000) architectural design rules are represented as design decisions.

However, neither of these research streams nor methods provide any suggestion on how to model architectural design rules, other than as informal text. There are also numerous languages intended for architectural descriptions, so called Architectural Description Languages (ADL) (Medvidovic, Dashofy, & Taylor, 2007; Medvidovic, Rosenblum, Redmiles, & Robbins, 2002; Medvidovic & Taylor, 2000) (e.g. ACME, Aesop, C2, MetaH, AADL, SysML and UML). However, none of these provide sufficient means to specify constraints or rules on groups of conceptual components only partly specified by the architect where the actual components are intended to be identified and designed by developers in later design phases. The main problem is that the ability to express these kinds of constraints is either missing, or the expressions to define even

quite simple rules become much too complicated to be usable in practice. An important factor to remember is that architectural design rules must be easily understandable by architects and developers; otherwise there is a risk that productivity will decrease instead of increase.

As a result of this lack of satisfactory support for modeling of architectural design rules the state of practice is to specify these kinds of rules in informal text. Informal text is ambiguous, open for different interpretations and impossible to automatically enforce. This makes transferring them to the developers an error-prone and time-consuming manual task, as reported in (Mattsson et al., 2009; Lang et al., 2005).

There is however a novel approach that promises to solve these problems by providing a method to model architectural design rules and automatically check that the detailed design conforms to the rules (Mattsson, Fitzgerald, Lundell, & Lings, 2012). An important feature of the approach is that the architectural rules are easily understandable by architects and developers, as claimed by the designers of the approach. This paper reports on the findings from an action case study using the approach in an industrial development project. The research objective was to study the effects on productivity and quality as well as the learnability of the approach in order to estimate its effectiveness in transferring architectural design decisions to the developers.

The rest of this paper is organized as follows. In the next section we introduce the novel approach for architectural modeling investigated in the action case study. Thereafter we present the research approach adopted for the study. Following that our findings are presented. Finally we discuss our conclusions.

On the Investigated Approach

In order to be successful in practice, it is essential that architectural design rules are modeled in such a way that they are both amenable to automatic enforcement and easy to understand and use by architects and developers. The latter is important in order to avoid increasing the work of developing the rules; otherwise there is a risk that the work burden is increased instead of decreased even though the enforcement is automated. Another important issue is that it should be possible to use current mainstream modeling tools to model both the architectural design rules and the system model so as to make it widely adoptable.

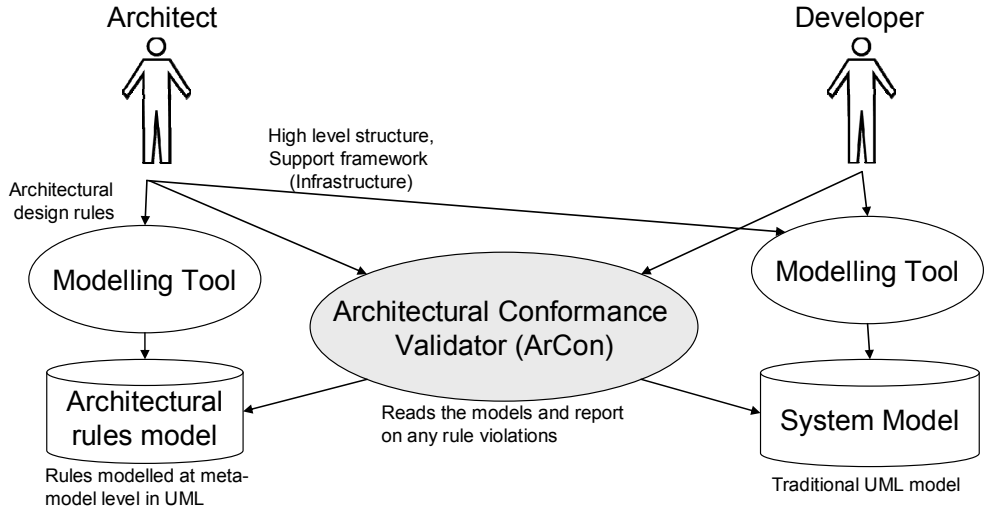


Figure 1: An approach for automatic enforcement of architectural design rules.

Such an approach is defined in (Mattsson et al., 2012) with tooling and modeling method available as open source at http://code.google.com/a/eclipselabs.org/p/arcon/. The approach, illustrated in Figure 1 is based on the idea to use UML (OMG, 2011) to define architectural design rules in a meta-model that constrains the use of UML in a system model.

Classes in the architectural rules model are transformed into UML stereotypes carrying constraints given by the constructs of the architectural rules model. The full mapping between UML constructs in the architectural rules model and the stereotypes used in the system model are provided in Appendix. Using this approach the constraint, *"A sensor may only have associations to In_Port_Ifc and Data_Items. These associations shall only be navigable from the sensor",* is modeled according to Figure 2.

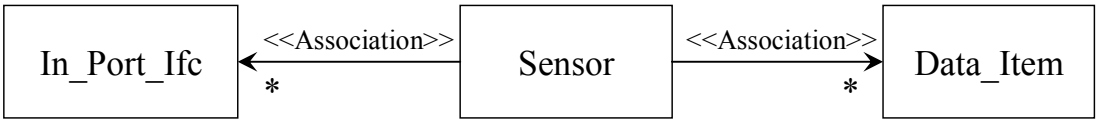


Figure 2: An architectural rule modeled according to the novel approach

Accompanying the method a tool called ArCon (Architectural Conformance Validator) has been built. ArCon is a stand–alone tool that reads the system model and the architectural rules model upon request, and reports on any violations in the system model against the rules defined in the architectural rules model. The idea is that developers should use the ArCon tool regularly to check that they are following the rules, thus eliminating the need for manual architectural reviews.

Research Approach

On the Research Approach Adopted

The objective of the research presented in this paper was to understand the effect of using the approach for automatic architectural enforcement illustrated in Figure 1 in an industrial development project. Of specific interest were effects on productivity and quality as well as the learnabilty of the method and tool in order to understand its effectiveness in communicating architectural design decisions.

Given this objective, a hybrid of the interventionist/change and interpretivist/understanding perspectives was appropriate. Braa and Vidgen (Braa & Vidgen, 1999) locate a number of hybrid research approaches where a mixture of perspectives is motivating the research, and in case of a mixture of interventionist/change and interpretivist/understanding perspectives, as in this study, the Action Case approach is deemed appropriate. 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 (Walsham, 1993). It is our belief that in this area where little exists by way of successful exemplars, then the appropriate approach is an in-depth study which a single case provides, what has been termed the "revelatory case" (Yin, 1994). 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?" (Mintzberg, 1979). 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 generalisation. However, Lee and Baskerville (Lee & Baskerville, 2003) identify a fundamental and long-standing problem with the type of generalization based on the type of statistical sampling fre-

quently 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 sample size or number of case study organizations, but Baskerville and Lee argue cogently for the ultimate futility of this flawed strategy. In terms of external validity, while a single case study cannot achieve statistical generalization(Runeson & Höst, 2009), but rather contribute to an analytical generalization in which the findings of the case study contribute more generally to developing a theory of the phenomenon being studied.

Action research originates from the work of Lewin (Lewin, 1948) and several 'flavours' 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 organisation, 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. (Baskerville, 1999; Kock, McQueen, & Scott, 1997; Lau, 1997).

The Action Case is a trade-off between being an observer who can make interpretations (understanding) and a researcher involved in creating change in practice. With case study methods the researcher aims to collect a rich set of data to provide insight into some situation, while in action research the aim is to support desirable change in an organizational setting. However, when doing case studies researchers contribute to change by questioning events and applying new concepts. On the other hand, full-scale action research projects are often not appropriate due to organizational constraints or the nature of the topic to be investigated. Small scale intervention with a deep contextual understanding is one way of balancing this dilemma (Boland et al., 2004).

These characteristics were very much present in this research: The intervention was done in a relatively small project over a limited time period, and where the researcher had deep knowledge of the problem domain and the organization. He had 20 years of experience from development of embedded real-time systems across a wide range of organizations in the automotive, defence, medical, telecom and automation industries. The last 15 of these years he had alternated between the roles of a software architect and a mentor in software architecture and Model-Driven Development in numerous projects, several of these in the organization where this action case study was performed. However, in this action case study the role of the researcher was purely that of a researcher, transferring the method to the architect and observing the use of it.

Data Collection and Analysis

Data was collected and analyzed following the suggestions of Seaman (Seaman, 1999). Building on constructs derived from earlier work an interview guide was created. Interviews were transcribed and items of relevance were coded. Archival data in the form of models and other development artifacts, as detailed below, was used as a second data source. In order to increase the validity of the results the two data sources were used for triangulation.

Semi-structured interviews were used to provide rich information on the interviewees' prior knowledge, values and expectations, all important for the estimation of the learnability of the method (Lings & Lundell, 2004). They were also used to provide information on perceived efficiency and quality of the work done.

Two interviews were done with each interviewee. The first, focusing on prior knowledge, values and expectations, was performed before interviewees started to work with the method. The second interview, focusing on usability, learnability of the method, perceived development effi-

ciency and product quality compared to traditional way of working, was performed when they had worked with the method for two to six months.

Four persons were interviewed - the architect and three developers working with the operating system kernel. Each interview started with a specific set of questions prepared in advance. Many of these were open-ended, intended to solicit information not foreseen by the interviewer and encouraging the interviewee to tell context-rich stories. The interviews were recorded and then transcribed. The transcribed interviews were sent to the interviewee who was invited to clarify, correct, and elaborate parts of the interview. All interviewees were also promised anonymity in any reports.

In addition to the interviews archival data was used in the data analysis as a tactic to further minimize the risk of bias from the researcher and improve the validity of the results. The following artifacts produced during development were used as data sources:

- The architectural rules model.
- The system model.
- A text document with the textual rules and additional rationale for the rules

Findings

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

Action Case Context

The action case study was performed in a development project at a company within the Swedish defense industry. The purpose of the development project was to develop a software platform, including a real-time operating system with special scheduling features, to be used in some of the system computers in a fighter aircraft. The project involved one project manager, one software architect and five developers during sixteen months until first customer delivery.

The project used MDD (Model Driven Development) with Rhapsody as modeling tool. Although there had been previous projects using UML modeling and code synchronization with the Rational Rose tool this was the first project using this tool chain and MDD with full code generation at the company.

The project followed RTCA/DO-178B for level A software, the highest safety level for avionics software. This meant that there were very strict and rigorous rules for how to develop and verify the software. Of specific interest for this study was the fact that although the architectural rules were modeled they still also had to be documented as text. The alternative would have been to qualify the architectural validation tool according to RTCA/DO-178B but that would have been far too costly for this project. The need for keeping the textual rules made it possible to count them and make comparisons between modeled versus non-modeled rules but it also made the benefits of reduced effort for defining the rules less than what could normally be expected.

The project started with an initial phase during one month where the initial architectural rules were modeled in collaboration between the researcher and the architect. Then this architecture model was evolved by the architect alone during four months. At that point a four-hour workshop was held where the rules were presented for the developers by the architect. The developers then started the detailed design according to the architecture. Three incremental iterations were done during twelve moths until first delivery.

Findings in Development Artifacts

There were 126 rules and 39 guidelines in the Architectural rules document. Rules had to be complied with always, whereas guidelines were recommendations that could be broken if there were good reasons. Guidelines were also often more vague than rules and might require the developer to exercise judgment in order to follow them. Typical examples of rules were: "*A Data_Class shall only have private attributes*" and "*A Data_Class is only allowed to have operations to read and write attributes*". Typical examples of guidelines were "*Keep the coupling between components as low as possible*" and "*The use of inheritance should be avoided*".

Of the 126 rules, 121 were modeled and automatically validated by the tool. The five additional rules could be modeled according to the method but not automatically validated since they were using OCL (OMG, 2003) expressions not supported by the tool. The rules were modeled in seven different class diagrams. The most complex one is shown in Figure 3.

Of the 39 guidelines 18 could have been modeled and automatically validated by the tool. However, the architect chose not to model guidelines at all. The motivation for this was that since following them was not mandatory, there was a risk of 'polluting' the validation reports with error messages that should have been treated as warnings. The remaining 11 guidelines that could not be modeled were all of the nature of requiring the developer to exercise judgment in following them, which made them inherently impossible to formalize.

Thus, in total 73% of the rules and guidelines could be automatically verified and potentially, with some additions to the tooling, 89% would be possible to automate. This would have a major impact both on reduced effort for manual reviews and on the quality of the system.

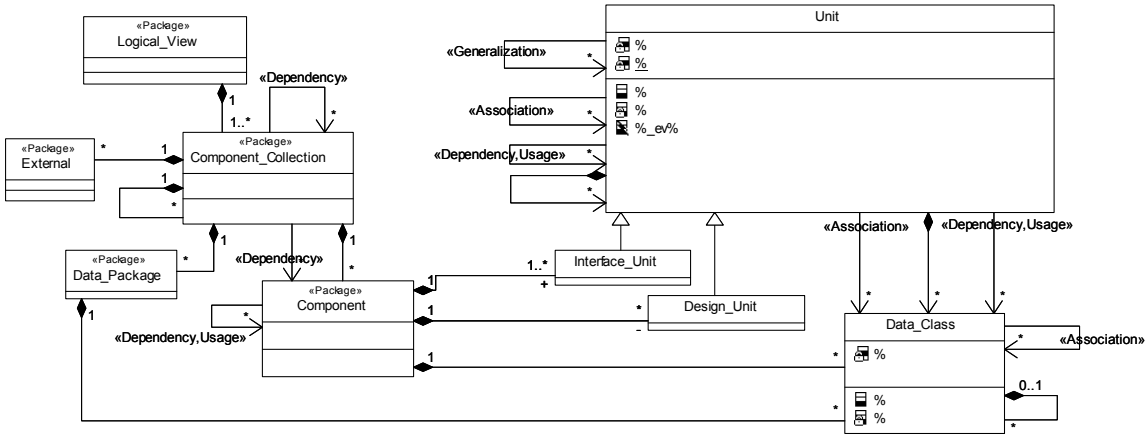


Figure 3: Model of rules applying to Components in the Logical View

Effects on Productivity

The architect stated that even though he was forced to develop both textual rules and the model, due to process requirements (see section "Action Case Context"), he gained efficiency in developing the architecture. This was due to the fact that the model was at a higher level of abstraction than the textual rules, making the complexity lower, and that the formalism of the models eliminated ambiguous and contradicting rules. It was easier, more efficient and gave better results to first model the rules and then manually extracts the textual rules from the model, than the traditional practice to work with rules only as text. He estimated that in a normal situation where the textual rules would not be needed he would gain between 10 to 50% depending on the complexity of the rules, in this phase. The greatest effect for his productivity though, he estimated, was that

he had to spend a lot less time communicating and reviewing the design than in a normal MDD project. Normally he would spend one to two days doing an architectural review; this time was now reduced to less than an hour checking the five rules not checked by the tool. He also used to spend a large part of his time explaining the meaning of the rules to the teams, with this new way of working there was almost no such questions. Spending a lot less time on reviews and communicating the rules enabled him to do optimizations to the architecture and supporting the teams in their design which, he stated, had made the system better and the teams more efficient.

All developers claimed that they found it faster to read and understand the modeled rules than the textual rules. The only difficulty they saw was that the error messages from the tool sometimes were difficult to understand, but as one of them pointed out:

"*This is the same problem you have with a new compiler; the error messages can be hard to understand at the beginning but you soon learn how to interpret them*".

All developers believed that after a few days they were already more efficient then they would have been if working the traditional way, and were sure that the quality, in respect to how they had followed the rules, was much higher than if they would have had to manually follow the textual rules. This, they estimated, had an even greater impact on their productivity since there was no longer any rework after the architectural reviews.

Effects on Quality

According to the architect, when working in the traditional way where the architectural rules are enforced by manual reviews, generally a few violations are not detected until very late in the process, typically in the design of a future version of the system, since many architectural rules address maintainability, scalability and portability requirements. With automatic detection of all violations there are no violations to the architectural design rules in the system. According to Boehm and Basili (Boehm & Basili, 2001):

"Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase."

This means that eliminating architectural violations should have a major impact on the quality of the system. The architect also claimed that the rules were of higher quality, compared to textual rules, since there were no ambiguous, redundant or contradicting rules due to the formal modeling.

Another effect was that both the review and the discussions about the architecture between the architect and the developers focused on how to package the functionality in different components and not on how to follow detailed design rules since this latter question was automatically handled. The architect claimed that in a "normal" project this would have been the opposite way around and that he believed that the increased focus on more difficult design decisions also led to a better system. The only apparent risk was that of relying too much on the tool and missing things that were not checked by the tool either by design or because of possible errors in the tool.

Learnability of the Method

The architect had eleven years of experience from UML modeling working as an architect in several organizations within the embedded software industry. He had also quite deep knowledge of the UML meta-model from building model-to-code transformations in MDWorkbench. The architect was positively disposed to using the approach since he was the one who had taken the initiative to use it after having seen a presentation of the method and tool at an internal workshop arranged for technology leaders within the company.

Developer A had a master degree in mechatronics and automation and two years of experience of design and implementation of embedded software. Developer B had a bachelor degree in electronics and three years of experience in design and implementation of embedded software. Developer C had a master degree in software engineering and electronics and two-and-a-half years of experience of design and implementation of embedded software. None of the developers had any knowledge of meta-modeling and all had only a one-week training course in UML modeling. All developers were positively disposed to using the method and tool based on expectations that it would reduce the effort needed to follow the architectural rules.

The architect and a researcher familiar with the method collaborated in developing an initial version of the architectural rules model with an effort of about two person-weeks during one month. After that the architectural rules model was evolved by the architect alone during four months with an effort of about three person-weeks. At that point the model was presented to the developers in a four-hour long workshop. After that there were only minor adjustments to the architectural rules model. The architect estimated that it took about one month to master most of the modeling concepts and an additional month to fully master the method.

Both the developers and the architect emphasized the benefits of getting immediate feedback from the tool, it made the learning of the rules much faster. All three developers claimed that they found it easy to understand the modeled rules right from the beginning with the primary reasons being that the rules were modeled in UML, which they all had some experience of, and that they got instant feedback from the validation tool, illustrated by these statements:

- *"If you understand UML then you understand the rules"*
- *"Instant feedback is the only efficient way to learn. If you give a banana to a monkey one hour after it has done something good nothing happens, if you do it instantly, it will learn to do it again"*

Conclusions

There is a lack of satisfactory solutions on how to model architectural design rules in the current body of literature. As a result the state of practice is to specify these kinds of rules in informal text. Informal text is ambiguous, open for different interpretations and impossible to automatically enforce. This makes transferring them to the developers an error-prone and time-consuming manual task. The objective of this study was to understand the effects of using a novel approach that promises to solve this problem in an industrial development project.

Several findings indicate that the approach provides a more efficient way of communicating architectural design decisions than the traditional way:

- Normally the architect would spend one to two days doing an architectural review; this time was now reduced to less than an hour checking the five rules not checked by the tool. The architect also used to spend a large part of his time explaining the meaning of the rules to the teams, with this new way of working there was almost no such questions.
- All developers claimed that they found it faster to read and understand the modeled rules than the textual rules.
- All developers claimed that they found it easy to understand the modeled rules right from the beginning.
- The architect claimed that the rules were of higher quality, compared to textual rules, since there were no ambiguous, redundant or contradicting rules due to the formal modeling.

- Both the review and the discussions about the architecture between the architect and the developers focused on how to package the functionality in different components and not on how to follow detailed design rules since this latter question was automatically handled. The architect claimed that in a "normal" project this would have been the opposite way around and that he believed that the increased focus on more difficult design decisions also led to a better system.

An important reason for the more efficient communication was that the rules were modeled in UML, which is a widely spread modeling language that all interviewees had experience in. Another important reason, as pointed out by several of the interviewees, was that that they got instant feedback from the validation tool whether they followed the rules or not.

Also the approach proved to be easy to learn, especially for the developers. The demands are of course much higher for the architect, who is tasked with constructing the rules, so it is not surprising it takes longer to master the method, still, two months must be considered to be a relatively short time to fully master a new modeling technique. It probably takes longer without any knowledge of meta-modeling but judging from how fast the developers, without any meta-modeling experience, understood the models this should not be a major hurdle. A lack of experience in UML modeling (or more significantly, any OO modeling language) would probably be a bigger problem, but UML modeling are today to be considered to be a basic required skill for a software architect

However, since change in work practice requires people to change, perhaps the most important finding was the enthusiasm expressed both by the architect and the developers, they all thought that they produced a better result with less effort and had more fun doing it. As expressed by one of the interviewees:

- *"I think you have come up with something really useful here."*

Although the action case study only covered one development project, two factors suggest that the results should, to a large extent, be transferable to other systems and organizations, at least in the embedded software domain:

1. The defined transformations are based on raising the general modeling constructs of UML to the meta-model level, not on the specific needs of the developed system.
2. The developed system is a real-world system.

Acknowledgement

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie)

References

Baskerville, R. L. (1999). Investigating information systems with action research. *Communications of the AIS, 2*(3), 4.

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice* (2nd ed.). Boston: Addison-Wesley.

Bengtsson, P., & Bosch, J. (1998). Scenario-based software architecture reengineering. *Fifth International Conference on Software Reuse* (pp. 308-317). Victoria, BC, Canada

Boehm, B., & Basili, V. R. (2001). Software defect reduction top 10 list *Computer, 34*(1), 135-137.

Boland, D., & Fitzgerald, B (2004). *Transitioning from a co-located to a globally-distributed software development team: A case study at Analog Devices Inc*. The 3rd International Workshop on Global Software Development, users.jyu,fi

Bosch, J. (2000). *Design and use of software architectures: Adopting and evolving a product-line approach*. Reading, MA: Addison-Wesley.

Bosch, J., & Molin, P. (1999). Software architecture design: Evaluation and transformation. *IEEE Conference and Workshop on Engineering of Computer-Based Systems, ECBS '99*. (pp. 4-10).

Braa, K., & Vidgen, R. (1999). Interpretation, intervention, and reduction in the organizational laboratory: A framework for in-context information system research. *Accounting, Management and Information Technologies, 9*(1), 25-47.

Hofmeister, C., Nord, R., & Soni, D. (2000). *Applied software architecture*. Reading, Mass.: Addison-Wesley.

Jansen, A., & Bosch, J. (2005). Software architecture as a set of architectural design decisions. *Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA 05)* (pp. 109-120).

Jansen, A., van der Ven, J., Avgeriou, P., & Hammer, D. K. (2007). Tool support for architectural decisions. *Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 07)* (pp. 44-53). Mumbay, India.

Kock, N. F., McQueen, R. J., & Scott, J. L. (1997). Can action research be made more rigorous in a positivist sense? The contribution of an iterative approach. *Journal of Systems and Information Technology, 1*(1), 1-24.

Kruchten, P. (1995). The 4+1 View Model of architecture. *IEEE Software, 12*(6), 42-50.

Kruchten, P. (2004a). An ontology of architectural design decisions in software intensive systems. *2nd Groningen Workshop on Software Variability* (pp. 54-61).

Kruchten, P. (2004b). *The rational unified process: An introduction* (3rd ed.). Boston: Addison-Wesley.

Kruchten, P., Lago, P., & van Vliet, H. (2006). Building up and reasoning about architectural knowledge. In *Quality of Software Architectures* (Vol. 4214, pp. 43-58): Springer Berlin / Heidelberg.

Lang, M., & Fitzgerald, B. (2005). Hypermedia systems development practice: A survey. *IEEE Software*, *20*(2), 68-75.

Lau, F. (1997). A review on the use of action research in information systems studies. In A. S. Lee, J. Liebenau & J. I. DeGross (Eds.), *Information systems and qualitative research* (pp. 31-68). London: Chapman and Hall.

Lee, A. S., & Baskerville, R. L. (2003). Generalizing generalizability in information systems research. *Information Systems Research, 14*(3), 221-243.

Lewin, K. (1948). *Resolving social conflicts: Selected papers on group dynamics*. New York: Harper & Row.

Lings, B., & Lundell, B. (2004). On transferring a method into a usage situation. In B. Kaplan, D. Truex, D. Wastell, T. Wood-Harper & J. DeGross (Eds.), *Information Systems Research* (pp. 535-553). Boston: Springer.

Mattsson, A., Fitzgerald, B., Lundell, B., & Lings, B. (2012). An approach for modelling architectural design rules in UML and its application to embedded software. *ACM Transactions on Software Engineering and Methodology, 21*(2).

Mattsson, A., Lundell, B., Lings, B., & Fitzgerald, B. (2009). Linking model-driven development and software architecture: A case study. *IEEE Transactions on Software Engineering, 35*(1), 83-93.

Medvidovic, N., Dashofy, E. M., & Taylor, R. N. (2007). Moving architectural description from under the technology lamppost. *Information and Software Technology, 49*(1), 12-31.

Medvidovic, N., Rosenblum, D., S., Redmiles, D., F., & Robbins, J., E. (2002). Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodologies, 11*(1), 2-57.

Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering, 26*(1), 70-93.

Mintzberg, H. (1979). An emerging strategy of 'direct' research. *Administrative Sciences Quarterly 24*(4), 582-589.

OMG. (2003). *UML 2.0 OCL Specification*.

OMG. (2011). *Unified Modeling Language: Superstructure* (No. formal/2011-08-06).

Perry, D. E., & Wolf, A. L. (1992). Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes, 17*(4), 40-52.

Ran, A. (2000). ARES conceptual framework for software architecture. In M. Jazayeri, Ran, A., van der Linden, F. (Ed.), *Software architecture for product families principles and practice.* (pp. 1-29). Boston: Addison-Wesley.

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering, 14*(2), 131-164.

Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering, 25*(4), 557-572.

Soni, D., Nord, R. L., & Hofmeister, C. (1995). Software architecture in industrial applications. *17th International Conference on Software Engineering IEEE Cat No 95CH35745*, 196-207.

Tyree, J., & Akerman, A. (2005). Architecture decisions: Demystifying architecture. *IEEE Software, 22*(2), 19-27.

Walsham, G. (1993). *Interpreting information systems in organizations*. Chichester: Wiley.

Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R. L., et al. (2006). *Attribute-Driven design (ADD), Version 2.0* (Technical Report No. CMU/SEI-2006-TR-023): Carnegie Mellon University/Software Engineering Institute.

Yin, R. K. (1994). *Case study research: Design and methods* (2nd ed.). Thousand Oaks, California: Sage Publications.

Appendix

In **Table 1** the definition on how to interpret the constructs in the architectural rules model as constraints on a system model is given. The definitions refer to an architectural rules model complying with the form of the generic model given in

Figure 4. References to terms defined in the generic model are written in italics.

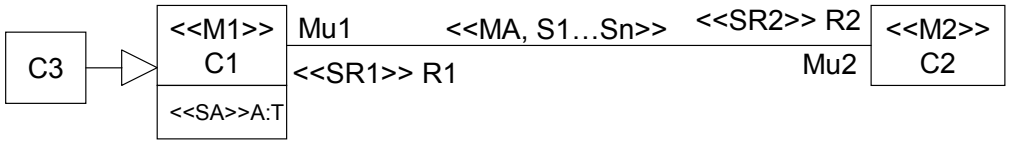
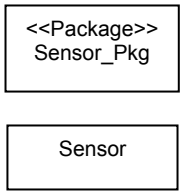
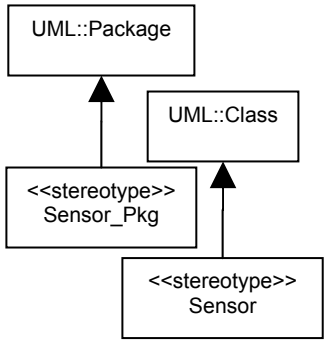
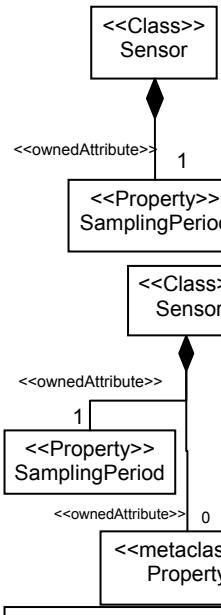
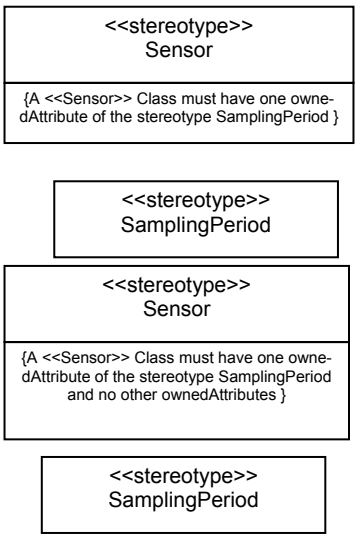
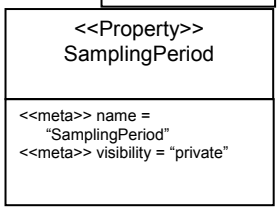
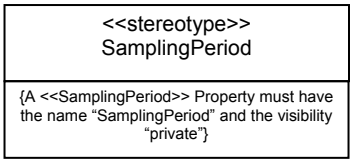
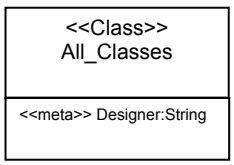
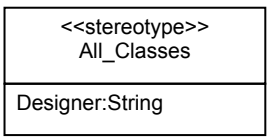
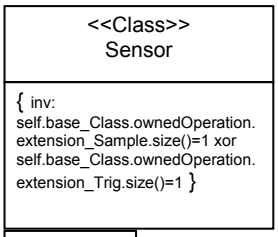
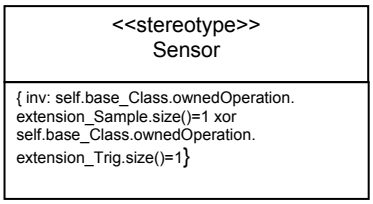
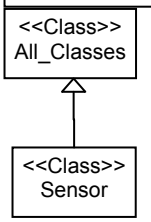
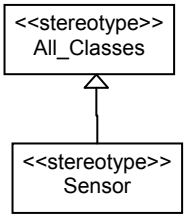
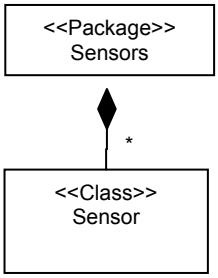
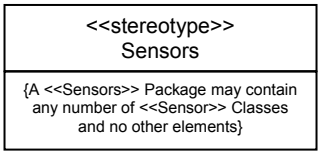
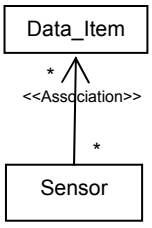
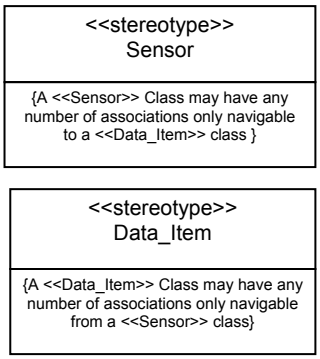


Figure 4: A generic architectural rules model used in the definition of the transformations

Table 1 Definition of transformations between constructs in the architectural rules model and constraints on stereotypes in the system model

Transformation	Example Architectural rules model	System model stereotypes
T1: A class named *C1* with the stereotype *M1* is transformed into a stereotype named *C1* extending the meta-class *M1* unless transformation number T3 below applies. If *M1* is undefined then "Class" is assumed		
I**T2**: If *SR2* is the role in the language meta-model on the far end of an association from the meta-class of *C1* to the meta-class of *C2* then the multiplicity of *R2* for a <<*C2*>> element shall be constrained to *Mu2* in stereotype <<*C1*>>		
T3: If *M1* equals "metaclass" then *C1* represents the class *C1* in the UML meta-model and is not transformed into anything in the system model. This can be used to specify constraints in other stereotypes in respect to these meta-classes		
T4: If *SA* equals "meta" far an attribute *A* and the name of *A* matches the name of an attribute of class *M1* in the meta-model then it is transformed into a constraint on that attribute on allowed values. The value of the attribute is constrained to match a regular expression specified as the default value of the attribute.		
T5: If no match is found for an *A* where SA equals "meta" (according to T4) then *A* is transformed into an attribute A of the stereotype (tag-definition), thus defining a tagged value to be set in the model elements where the stereotype is applied		

Transformation	Example Architectural rules model	System model stereotypes
T6: Any OCL constraint in the context of a class *C1* is copied exactly as it is into the stereotype *C1*		
T7: A generalisation relationship from a class *C3* to a class *C1* in the architectural rules model is transformed to a generalisation from stereotype <<C3>> to stereotype <<C1>>. The UML meaning of this is that all constraints and attributes of the stereotype <<C1>> are inherited by the stereotype <<C3>> and that any <<C3>> element is also an <<C1>> element.		
T8: If M1 equals "Package" and the aggregation of R1 is "composite": A <<C1>> Package is constrained to have Mu2 number of <<C2>> elements as packagedElements. The visibility of these elements shall be the visibility of Mu2. Also, a <<C1>> package is not allowed to have any packagedElements unless explicitly allowed iin the model.		
T9: *<<C1>>* elements are only allowed to have the associations, dependencies, generalizations and realizations explicitly allowed according to T10 and T11.	See examples for T10 and T11.	
T10: If *MA* equals "Association": A *<<C1>>* element shall be associated with *Mu2* number of *<<C2>>* elements. The association ends shall have the same navigability, aggregation (none, shared or composite) and visibility as *R1* and *R2*. The association ends shall also have qualifiers according to the qualifiers of *R1* and *R2*. The name and type of these shall be according to the transformations for attributes specified in T12. The association shall have the stereotypes *S1* to *Sn*.		

Transformation

T11: If *MA* equals “Dependency”, “Generalization” or “Realization” and the association is only navigable from *C1* to *C2*:

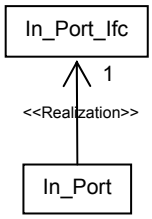
A <<*C1*>> element shall have a relationship according to *MA* to *Mu2* number of <<*C2*>> elements with stereotypes *S1* to *Sn*

T12: If there are attributes *A* of *C1* where SA is not equal to “meta”:

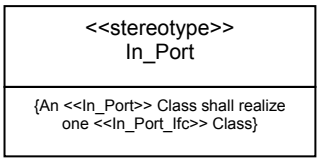
- All parts of the definition of an attribute of a <<*C1*>> class must match the corresponding part of an *A*, where the wild card characters “@” and “%” in any part of the definition of *A* can be replaced with any character sequence. Parts of *A* not specified (as for instance default value for Sampling_Period in the example to the right)
- All *A* must be matched by one attribute in a <<*C1*>> class. An exception to this is if the name of *A* contains the wild card character “%”; in this case any number of matches (including zero) is allowed.
- If the name of a type of *A* is identical to the name of a class C in the architectural rules model then the type of a matching attribute must be a <<C>> element.

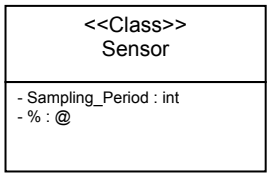
Example

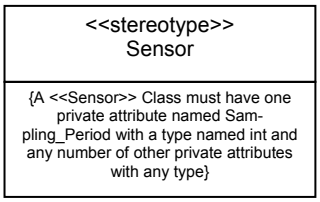
Architectural rules model



System model stereotypes







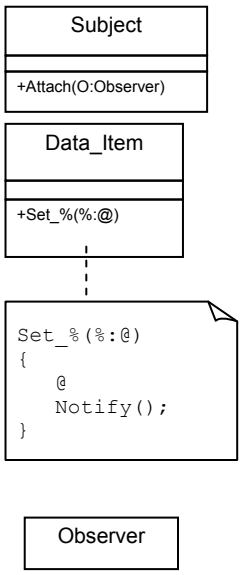
Transformation

T13: If there are operations O of *C1*:

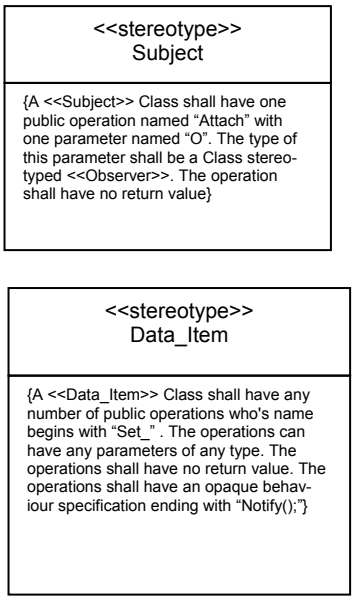
- All parts of the definition of an operation of a <<*C1*>> class must match the corresponding part of an O, where, for each part of the definition, the wild card characters "@" and "%" can be replaced with any character sequence. Properties of O not specified (as for instance parameter directions for operations in the example to the right) are unconstrained.
- This requirement holds for all parts of the definition of O defined in the UML meta-model, such as for instance opaque behaviour specified for the operation.
- The character "%" in a parameter name means that the definition of this parameter can be repeated any number of times, including zero. In these parameter definitions "%" can be replaced with any character sequence.
- If the name of the type of O or a parameter of O is identical to the name of a class B in the architectural rules model then the type of matching operations or parameters in the <<*C1*>> class must be of a <<B>> Class.
- All O must be matched by one operation in a <<*C1*>> class. An exception to this is if the name of O contains the wild card character "%"; in this case any number of matches (including zero) is allowed.

Example

Architectural rules model



System model stereotypes

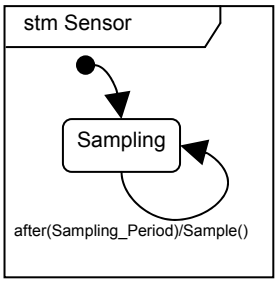


Transformation

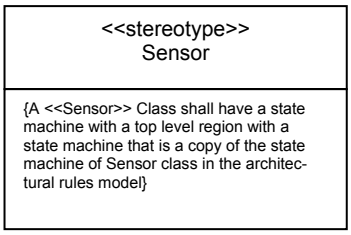
T14: If *C1* has a state machine then a <<*C1*>> class must have a state machine where there for each region in *C1* shall be an identical region in the <<*C1*>> class. The wild card character "@" may be used in the transition definitions in *C1* and shall then be matched with any text string in the corresponding transition in the state machine of a <<*C1*>> class. It is allowed to have additional regions in the state machine of a <<*C1*>> class.

Example

Architectural rules model



System model stereotypes



Biographies

Anders Mattsson received the MSc degree from Chalmers University of Technology, Sweden, in 1989. He then worked two years as a software designer at Volvo Data AB, followed by one year as a system designer at Saab Instruments AB. After that he worked 19 years as consultant, manager and Lead Engineer at Combitech AB with an increasing focus on software architecture and model-driven development. He is currently software design manager at Husqvarna AB. He is currently also pursuing a PhD at Lero – the Irish Software Engineering centre. His research interest include software architecture and model-driven development in the context of embedded real-time systems.

Brian Fitzgerald holds an endowed professorship, the Frederick A Krehbiel II Chair in Innovation in Global Business & Technology, at the University of Limerick, Ireland. He has previoysly served as Vice-President Research at the University of Limerick and as Research Leader for Global Software Development at Lero – the Irish Software Engineering Research Centre, where he was also Founding Director of the Lero Graduate School in Software Engineering. His publications include 12 books, and more than 130 papers in leading international journals and conferences. Prior to taking up an academic position, he worked for 12 years in the software industry in a vraiety of roles and sectors.

Björn Lundell has been a staff member at the University of Skövde since 1984, and he received his Ph.D. from the Universty of Exeter in 2001. 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 has been researching the Open Source phenomenon for a number of years. His research has also included fundamental research on evaluation, and associated method support. His research is reported in over 50 publications in a variety of international journals and conferences.