Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 133 (2005) 101–117
www.elsevier.com/locate/entcs

Improving Software Quality in Safety-Critical Applications by Model-Driven Verification

Anders Henriksson, Uwe Aßman[1,2]

Linköpings Universitet
Linköping, Sweden

James Hunt[3]

Forschungszentrum Informatik
Karlsruhe, Germany

Abstract

We propose a new development scheme for quality-aware applications, *quality-driven development (QDD)*, based on the Model-Driven Architecture (MDA) of *Object Management Group* OMG. We argue that software development in areas, such as real-time systems, should not only rely on *code verification,* but also on *design verification*, and show that a slightly extended MDA process offers the opportunity to integrate system development together with design verification.
As an instance of the method, we present the MDA-based tool environment of the HIDOORS project [10]. In this environment, a real-time model checker is interpreted as a platform in the sense of MDA. UML designs can be annotated with verification markup, which is not only compiled to code, but also to a *design verification model* of the *verification platform*, the model-checker. In this way, model-checking for real-time designs is integrated into the model-driven development process and allows for *early verification*.
The approach can easily be transfered to other verification techniques. We give a preliminary classification of the possible verification platforms and analyse their interplay. The analysis reveals that for quality-aware application areas, the standard MDA approach should be extended by one or more MDA stacks for *model-driven verification (MDV)*. The resulting approach, *quality-driven development (QDD)*, is, to our knowledge, the first systematic approach to integrate code generation and verification in model-driven development.

Keywords: Model Driven Architetcture, Model Driven Verification

[1] Email: andhe@ida.liu.se
[2] Email: uweas@ida.liu.se
[3] Email: jjh@fzi.de

doi:10.1016/j.entcs.2004.08.060

1 Introduction

If in quality-aware application domains, such as real-time scenarios, software should be developed, appropriate static verification mechanisms are required. Trusting a system requires certification, and certification requires proofs about the functional and quality features of the software (conformance to specified functionality, resource restrictions, timing constraints, liveness or fairness of the software, etc.) Hence, for quality-aware domains, both functional and non-functional requirements play an important role and exert a strong pressure to prove the resulting product against its specification.

MDA is a novel, promising development architecture and process for multi-platform development [12]. It has a lot of advantages. For instance, it promises to reuse design models over many platforms. The design specifications (platform-independent models, PIM) are transformed towards a platform-specific implementation models (PSM), using parameterizations and other mapping technologies. MDA also tackles the *design aging problem*. In development often the relation between code and specifications is lost after some time because the specifications are no longer updated when the implementations evolve. Since MDA maintains *mappings* between design and implementation levels, implementation changes can better be tracked in the designs, and designs can be updated easier.

Unfortunately for verification-aware application domains, in particular, embedded and real-time software, little attention has been payed to integrate the required verification techniques into MDA. Since in these domains verification has to take place on the code level, two problems result which MDA could—but does not yet—solve. Firstly, design is often not verified at all. Because the systems must be certified on the code level anyway, it is not attractive to invest the effort to relate the code additionally to a design verification. Of course, this contributes to design aging. Secondly, even if a verification of the design is done, it ages rather quickly. Design models may be produced and checked, but since the proofs have to be repeated on the code level, the design verifications are thrown away after the transition to the code. In other words, in quality-aware software domains the problem is that the design verifications are not *integrated* with the design, nor the code, nor the code verifications.

This indicates that MDA, although being targeted at quality-aware application domains, such as real-time and embedded systems, does not yet offer the necessary integrated verification technology. This paper attempts to fill this gap and proposes an extension of MDA for *model-driven verification (MDV)*. For the integrated method, MDA+MDV, we use the notion of *quality-driven development (QDD)*.

The paper summarises observations from the **HIDOORS** project [10], in which a design verification of a real-time UML model has been envisaged, in addition to the usual code generation and code verification. In **HIDOORS**, the real-time UML designs are verified with a real-time model checker (Sec. 3). For the integrated development of verification and code generation we have developed a real-time UML profile [5], which refines the SPT profile of OMG [11]. The profile is used for code generation, as well as for the design verification with the real-time model checker.

In this framework we have discovered that MDA should be extended to support design verification. It is advantageous to introduce a second kind of platform stack for verification, in parallel to the traditional MDA stack (Sec. 3). This leads to the definition of new *verification platforms* for design and code verification, and platform stacks towards the final verification model of these platforms (Sec. 4). We show that in **HIDOORS**, two verification platforms are used: the model checker and a worst-case execution time analysis (WCETA), which analyses the generated Java system. We give an overview of the interplay of the traditional MDA stack and the verification stacks of the model checker and the WCETA (Sec. 4.2).

Hence, the basic observation of the paper is that MDA for quality-aware software needs to be extended by model-driven verification (MDV) to quality-driven development (QDD). To our knowledge, QDD is the first method for MDA in quality-aware application domains.

QDD has more advantages. Due to the tight integration with MDA, MDV supports *early* verification. The **HIDOORS** MDV supports model-checking for real-time designs early on in the development process, even before unit testing. If the model-checker verifies the design, the code has a much better chance to be correct, which implies that the code verification becomes simpler. Secondly, due to the benefits of model-driven development, the employed verification platform can be easily exchanged for others. In the **HIDOORS** framework, the employed model checker can be exchanged for another. Finally, there is a lot of cross-fertilisation of the MDA and the MDV stacks. Because the design verification aids the code verification, and the code verification supports the design verification, it pays off to keep and maintain the design. Because the design is not only used for code generation, but also for design verification, developers are encouraged to keep design and code consistent always. Hence, QDD reduces the effects of *design aging.*

In the following, we talk of *design verification* if a design model is verified against its requirements model. In particular, this include the case that timing requirements are proven for the design. In contrast, we speak of *code verification* if an implementation model is verified against its requirements

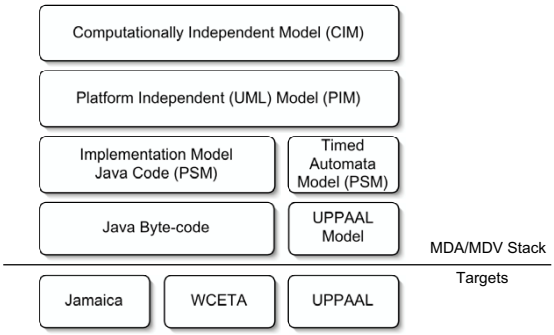

Figure 1. The HIDOORS model stacks: MDA and MDV.

model. (This implies that we identify the platform-specific implementation model with the code.) We summarize stereotypes and tagged values under the name *markup tags*, because they markup UML designs, i.e., assert non-standard domain-specific information. As usual, a *model stack* is a layered set of models on different abstraction levels.

2 Quality-Driven Development in HIDOORS

In **HIDOORS** an MDA stack is used for producing target implementations for a selected target platform. MDV is essentially additions to the existing MDA stack that support verification tasks adding additional targets for code analysis and model verification. Together the MDA stack with the MDV additions form an QDD-stack for **HIDOORS** Qualitiy-Driven Development as shown in figure 1.

2.1 The HIDOORS MDV stack for model verification

The **HIDOORS** MDV stack for model verification is for intended for verifying the constraints expressed in the UML model of a system. As opposed to MDA where the final target always is an implementation the final target for the MDV stack for model verification is a verification model intended as input for a model-checker. The two first levels in the MDA stack are the same for MDA and verification. Where the platform specific model for the MDA consists of Java code the platform specific model for the MDV is a Timed Automata [2,1]. The final level of the MDV model stack is the input model for the model-checker. The model verification works on an abstract model of the system. On this level not all details are available about the target implementation and runtime platform. To make model verification feasible some assumptions must be made. During model verification it is assumed that the time budgets

given in the UML model using annotations form the **HIDOORS** profile equals worst case execution time for the budgeted code and that these budgets hold. The model verification verifies the model constraints and the verification is valid only as long as the assumptions hold.

The WCETA utilizes the same stack as the MDA but for different purposes. Similar to the MDV stack for model verification the MDV stack for code analysis is intended for verification. In this case the target is a Worst-Case Execution Time Analysis (WCETA) engine. WCETA and model-verification works on models of different abstraction levels. While the model verification verifies constraints in the model it makes assumptions about the final implementation and runtime platform. It is up to the WCETA analysis to verify that these assumptions will hold for the final implementation on the selected runtime platform.

3 The HIDOORS Model Driven Verification Process

Model Driven Verification (MDV) in **HIDOORS** consists of two major steps. The first step is the verification of model constraints against the model, the second step is the verification of the model constraints against the implementation. Together with the **HIDOORS** MDA these two steps form the **HIDOORS** MDV process illustrated in figure 2.

In the **HIDOORS** development environment the user starts with requirements engineering, specifying the functionality and constraints of the system. After requirements engineering follows systems modeling where the user specifies a system design in UML using the **HIDOORS** profile [5]. When the system is modeled, model constraints are verified in the model verification step.

The model verification assumes that the modeled time-budgets will hold and assumes that time-budgets are equal to worst-case execution time, the timing constraints in the model are then verified with respect to these assumptions. Next follows a decision, if the model-verification step verifies that the modeled constraints will hold, then the process can continue, otherwise some actions must be taken to correct the model. When the model-verification step fails this means that at least one modeled constraints will not hold in the implemented system *on any platform*, given the modeled time-budgets the broken constraint will be impossible to achieve. The next question is wether the system can be remodeled to better meet the constraints. In those cases when the system cannot be remodeled to meet the necessary model constraints the next question is if the system requirements can be relaxed to make the system feasible. If this is not possible the required system is unfeasible, it

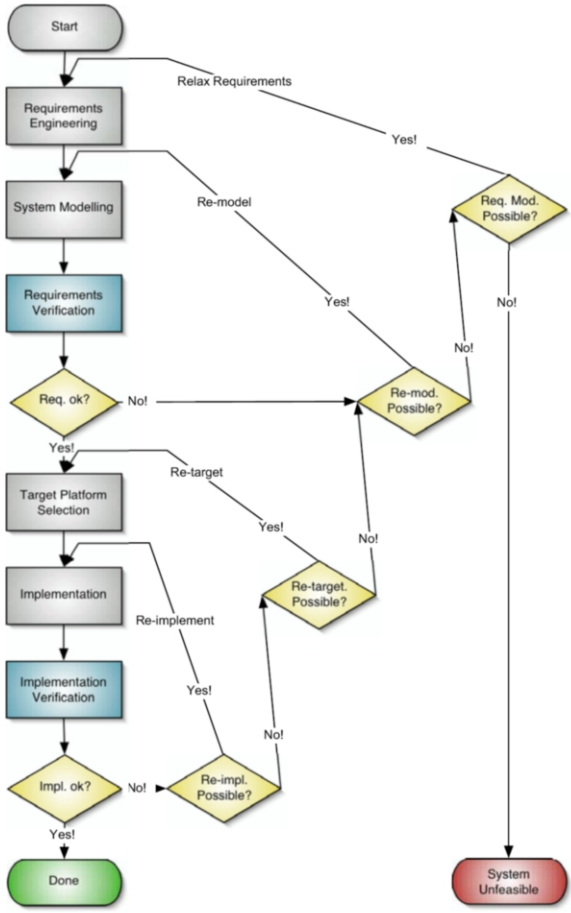

Figure 2. The HIDOORS Model Verification Process.

cannot be built.

When the systems model has passed the model verification step the model represents a feasible system. The next thing to do is to select a target platform for the system. Given a target platform and a systems model the system is implemented. Next follows the second major step in the MDV-process, verifying the implementation of the model. The model-verification makes certain assumptions about the implemented system. Now it is up to the implementation verification step to verify that the model is implemented in such a way that these assumptions hold. When the implementation verification fails this means that the implementation breaks one ore more assumptions made by the model-verification. This means that the implemented system will not full-fill the modeled constraints in some case.

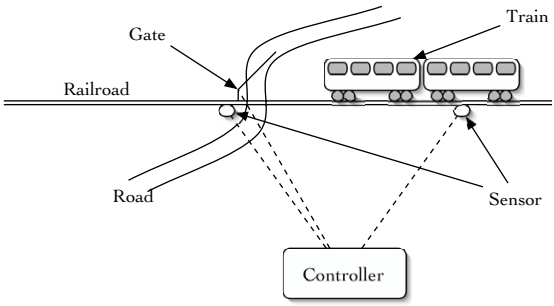

Figure 3. The Train-crossing example.

If the implementation verification fails the first question is wether the model can be re-implemented or refactored in such a way that the assumptions will hold. If the system cannot be re-implemented the next question is if the system can be retargeted to another platform with more resources. When no platform exists where the system can be implemented then the model cannot be implemented. In this case re-modeling is required. Finally when an implementation is found that can be verified on an available target platform system has been implemented that will meet the model constraints on that given platform.

3.1 An Example

To illustrate the MDV process, the traincrossing example will be used as a running example. Figure 3 shows an overview of the train-crossing scenario.

The main objective in this example is to design the controller that opens and closes the gate at a train-crossing when a train passes. There are four objects involved in the example, a sensor that detects the train as it approaches, a gate that should be closed, a sensor that detects that the train has left the train-crossing and a controller which acts upon the signal from the sensors and handles the gate. From a real-time perspective there is a constraint on the response time of the controller. The response time must be for the gate to be closed by the time the train arrives at the gate. The idea is that the train has a maximum allowed speed when it passes the gate, which together with a fixed distance between the sensor and the gate yields a time budget for closing the gate.

The system is modeled in UML using class-diagrams, message-sequence charts and state-carts. Figure 4 illustrates the UML model of the train-gate system.

The UML-model is annotated with stereotypes and tagged values form the **HIDOORS** profile [5]. The stereotypes and tagged values of the **HIDOORS**

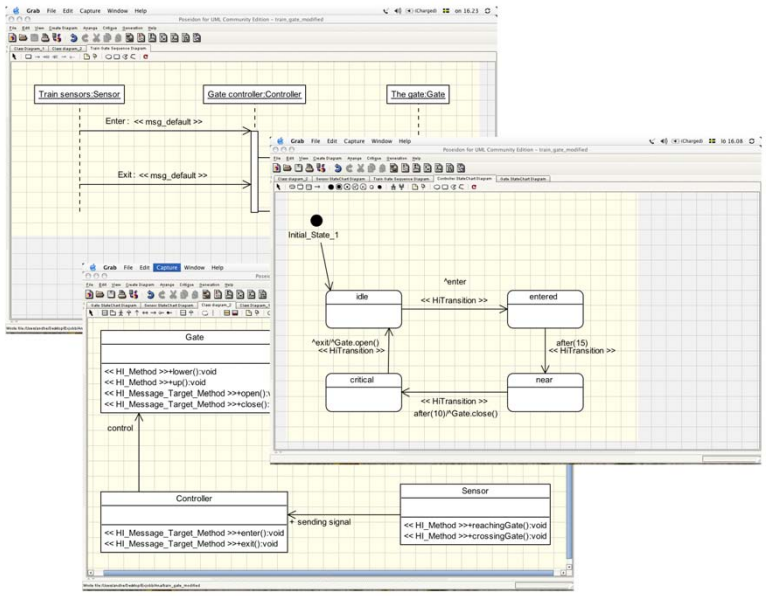

Figure 4. The Train-crossing example modeled in UML.

profile expresses relationships and constraints between UML elements of the model such as for example timing constraints and buffer sizes.

3.2 Verifying Model Constraints

The objective of the model verification step is to verify that the modeled constraints will hold assuming that the model is correct. The UML model is platform independent, that is, no information is available about the resources available on the target plaform. In order to be able to perform any useful verification task at this abstraction level some assumptions must be made about more detailed abstraction levels. It is assumed that the time-budgets associated with methods equals worst-case execution time of the implemented method on the target platform and that these time-budgets will not be broken.

Timing constraints in the model are expressed as relative deadlines in message-sequence-charts and state-charts. Buffer constraints are expressed as buffer sizes associated with the receiver. Constraints are checked using a symbolic model-checker. Model verification is performed through the model-

Figure 5. The **HIDOORS** Model-checking tool-chain.

checking tool-chain shown in figure 5.

The model-checking tool-chain consists of four major steps; class-diagram to state-chart transformer, message-sequence-chart to state-chart merger, timed automata generation and last model-checking. The first step in the model-checking tool-chain is the Class-Diagram (CD) to State-Chart (SC) transformer. The CD to SC transformer is a "lowering engine". This transformer analyzes the class-diagrams for constraints between classes representing communication patterns and instantiates the objects needed to implement this communication pattern in the form of state-charts. Next is the Message-Sequence-Chart (MSC) to SC merge. Using the **HIDOORS** profile the user may express timing constraints both in MSCs and SCs. The MSC to SC merges the timing constraints expressed in the MSCs into the corresponding SCs. The state-charts are then used as a source for generating a Timed-Automata [2,1] representation of the SCs suitable for model-checking.

The last step in the model-checking tool-chain is the model-checking itself. In the **HIDOORS** project the UPPAAL [3,8] model-checker is used for verifying timing constraints and buffer sizes. A model-checker essentially

checks for the reachabillity of a certain state in a timed automata. Thus the timed automata representing the system model is constructed in such a way that whenever a constraint is broken the corresponding timed automata moves into an error state. Checking of model constraints can then be performed by checking for reachabillity of the error sates in the timed automata. When a model-checker finds a state unreachable it simply returns stating that fact. If on the other hand the model-checker finds a state reachable it will return a "trace" showing how the state was reached. Whenever an error state is found to be reachable the constraint that was broken can be found trough analyzing the "trace" produced by the model-checker to find from which state the error-state was reached.

3.3 Verifying an Implementation of the Model

Verifying the correctness of a system's timing requires not just model checking: the final implementation must be verified as well. By the time analysis occurs at the implementation level, the overall timing and scheduling should have been verified by model checking. What needs to be done is to demonstrate, that the implementation is consistent with the model. The**HIDOORS** project uses worst case execution time analysis (WCETA) as a basis for this validation.

When moving from the Model to Implementation, it is important to clearly separate pure design information from the details of implementation. In theory, the timing information in the model should depend only on factors that are external to the control hardware and software. In practice, time budgets often need to consider the relative computational difficulty of different tasks that may compete for hardware resources. Still, one should be able to change the hardware used for implementation without changing the model or invalidating its model level verification results. The role of WCETA is then relegated to demonstrating that a particular implementation abides by the timing information given in the model.

Since the goal of MDA is to generate code automatically from the model, care must be taken in the interpretation and application of timing constraints in the model. The MDA code generation process can introduce new subclasses with system dependent implementations (in the following called *helper classes*). The methods in the new helper classes must meet the same timing constraints as those in the original class. This parallels the general constraints for proper subtyping.

A subclass must be usable wherever its superclass can be used. In the functional domain this means that each method in the subclass has preconditions that a equal to or weaker than those of the equivalent method in the superclass and postconditions that are equal to or stronger than those of the

overridden method. By analogy, all resource constraints are postconditions. In other words, a proper subclass must meet all resource constraints on each of the methods it overrides from its parent class.

Once this principle is established, validating the timing of the implementation can be reduced to demonstrating that the implementations of all methods execute within the timing constraints specified in the model. Since the overall timing relationships have been validated by model checking, WCETA can concentrate on validation the timing of individual method calls. These method calls need only fill the timing constraints given in the model.

Several steps are involved in moving from model to implementation. The UML model must first be converted to code. In the **HIDOORS** case, this means Java code. The Java code is then compiled to byte code. The platform independent nature of Java means that this step can also be implementation independent. Platform dependencies are introduced in the next step when Java byte code is compiled to machine code. WCETA is then used to analysis this machine code in conjunction with a detailed description of the processor and caches with their respective clock frequencies and the memory bus frequency.

One could eliminate the translation of byte code to machine code by using the Java virtual machine's interpreter. Though the code would remain platform independent, its execution would not. In fact, timing analysis would be more difficult, since the byte code would need to be analysed together with the interpreter. Analysing just-in-time compilation is simply intractable. Therefore, the **HIDOORS** tools are designed to compile all byte code to machine code in time critical tasks.

The combination of compiling byte code to machine code and the use of proper subtyping relationships for subclasses means that standard techniques for WCETA can be used. The key is to pass timing constraints from the UML model through to the resulting native code. The results are not part of the model, but rather constitute validation evidence of a particular implementation. Changing the target platform will require rerunning the WCETA, but the model checking results would still apply.

4 Verification and Runtime Platforms in QDD

After looking at the examples from **HIDOORS**, this section investigates the relationship of the verification and runtime platforms in QDD in general. We observe that the goal to verify introduces strong coupling between the verification models and improves the motivation for developers to keep all specifications consistent, also during software evolution.

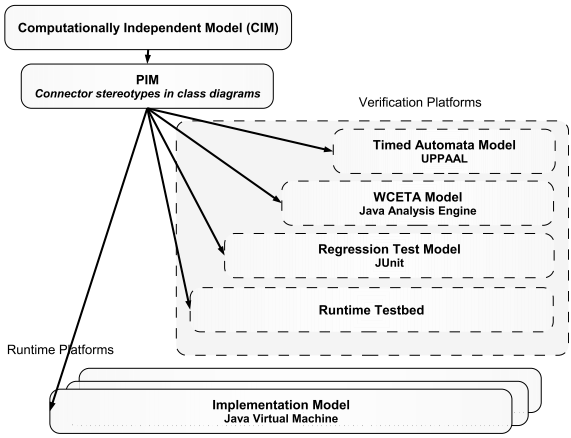

Figure 6. Verification and runtime platforms in QDD.

In the following, stereotypes and tagged values are called *markup tags*, because they markup UML designs, i.e., assert non-standard domain-specific information.

4.1 An analysis of the platform types

Standard MDA builds on *runtime platforms.* Since MDA is a process, which reuses design specifications, models, for several platforms, its focus is *reuse.* As such, MDA is not concerned with correctness of the designs, which is a major disadvantage for quality-aware applications. MDV, however, employs model-driven development for *verification* of models on *verification platforms.* Its focus is not on reuse, but on verification of the design models and the code. MDV intends to aid the final code verification, by transforming the designs into a form that they can be checked, validated, or proven.

Platforms fall into the following groups, forming the *platform classification of QDD* (see Fig. 6):

Runtime platforms. These are the standard platforms of MDA, such as component models, machines, operating systems, etc.

Verification platforms. These platforms execute design or models either symbolically, or in test environments.

- **Code validation platforms.** Such platforms are test environments for testing code correctness.
  - **Test platforms.** Test platforms execute the code in a test environment, in vitro. These can be unit test platform, regression test platforms, such as JUnit, etc.
  - **Runtime test-beds.** Runtime test-beds execute the system on the runtime

platform, but introduce additional runtime checking, e.g., dynamic type tests, exception tests, etc.

Design verification platforms. Verification platforms provide a formal language in which features of the system can be verified statically. Design verification platforms, however, verify *design models (design verification).* A design model is more abstract than the code. Examples are Petri Net models, CSP models, or real-time state-chart models.

Code verification platforms.

Analysis platforms. These platforms are tied to *abstract interpretation* [6], i.e., symbolic execution of the code to prove features of the system. The WCETA analysis of **HIDOORS** (Sec. 3.3) is an example.

Functional verification platforms. In these platforms, the code is fully verified with regard to functional and non-functional requirements.

In development of quality-aware software, all these verification and validation platforms play an important role because features of the system should be proven or checked before the system runs. Hence, system development in safety-critical domains requires a tight interplay between code and verification. MDA alone, reasoning only about *architecture,* is not enough for these applications.

Typical for this scenario is that a verification platform can be more *abstract* or *higher-level* than another. This means that the system model is represented in a more abstract form than the code. For instance, the model-checking verification model assumes certain features of the code, which have to be proven or tested later, by other verification platforms, for instance the WCETA. Hence, model-checking platform is more abstract, seeing less detail of the system. On the other hand, in a more abstract verification it might be easier to prove features of the abstract system. Model checking works because it assumes that the interactions correspond to regular language (which is usually an abstraction). And this is the reason why QDD is attractive: in order to prove system features, a set of abstract verification platforms is needed, which relate to each other and allow for conclusions about the quality of the system.

4.2 The Interplay of MDA and MDV in QDD

This section investigates the relationships of the verification platforms and the runtime platform. For verification and testing of a verification platform, information has to be added to the PIM. This can be done by adding markup specifications. In the following, we investigate for the example of the **HIDOORS** QDD, how the different types of markup for verification interplay

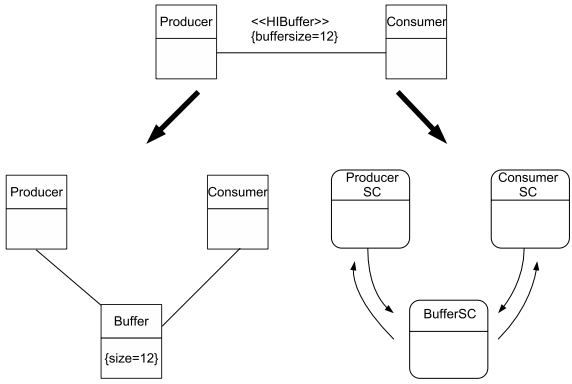

Figure 7. Translation of a PIM-related stereotype HIBuffer. Left: translation to Java class model (PSM), right: translation to state-charts

with the markup for the MDA.

Example 4.1 For the interplay analysis, let us present an example from the **HIDOORS** profile (Fig. 7). The profile contains an association stereotype *HIBuffer*, which can be annotated to binary associations, indicating that the two incident classes communicate with a producer/consumer protocol over a FIFO buffer. The stereotype can be enriched with a tagged value *size* that specifies the size of the buffer. Once annotated to an association of the PIM, the stereotype can be translated into a standard PSM. In that case, the code generation inserts a buffer helper class that maintains the buffer of length *size*. **HIDOORS** uses Java, so the buffer is a Java class. On the other hand, the stereotype implies a protocol, the FIFO protocol, which can be translated to a set of state-charts representing all involved classes.[4] The state-charts can be translated to a set of timed automata and fed into the model checker verification platform.

Suppose a change in the requirements occurs that reduces the available memory of the runtime platform. Then, the PIM should be changed such that the buffer is reduced in size. In QDD, not only the code can be regenerated easily, also the model checker platform can be used to verify the changed design.

Having this example in mind, we can observe that verification-related markup serves at least 3 purposes: *code generation*, *assertional assumption* to facilitate proofs on a higher-level verification platform, and *design knowledge assertions* for a lower-level verification platform. Firstly, markup serves

[4] This translation is ongoing work.

for code generation in the standard MDA stack (which is well-known). Then, the markup may verification knowledge for verification models. In the first place, the markup may *assert assumptions* about lower-level verification models and, finally, the code. These constrain the lower-level models, generating *proof obligations* for them. Assertional assumptions are an effect of abstraction. Since some details of the lower-level verification platform or the runtime platform are ignored, the proof on the higher level relies on assertions that must be proven in the later platform. Essentially, assertional assumptions define *lemmas* for proofs in higher-level models, which are proven in a lower-level model. Hence, the assertions provide an *interface* between proofs in higher-level models and proofs in lower-level models.

Essentially, the assertional assumptions bridge proof gaps in proofs of higher-level verification models, because the markup generates proof obligations for the lower-level verification platforms.

Finally, there is a kind of dual effect in QDD. In a PIM, *assertions about design knowledge* can be specified, encoding design knowledge that helps verification in later phases. Usually, a designer has more information about a system than a code analysis can find out, or a code verification can prove. In general, full verification may be rather hard for a system, however, it can be aided by additional assertions of design knowledge. Then, the code verification delivers a proof that is *relative* to the assertions on the design level. We call this effect a *design-supported code verification*.

Design-supported verification has been discovered also in verification of functional requirements. Its advantage is that it enables *relative verifications* in those cases when a full verification is too difficult [4,9].

4.3 How QDD Tackles Design Aging

We have shown that verification models are related to each other and to the code by two phenomena, namely *assertional assumptions* generating proof obligations for lower-level models, and *design knowledge assertions* for the proofs on a lower-level platform. This has several consequences.

Firstly, proofs on each abstraction level support proofs on other abstraction levels. In particular, design knowledge assertions support code verifications. Design verification can be done on the model, and assertional assumptions can simplify them. Hence, applications can be checked early on on errors, and the trust of the developer into the system is greatly improved, even before the code verification is performed.

Because the verification of the design model can be used for proving features of the system, it is attractive not to throw it away after code verification, but to use it to re-verify the system, whenever the system changes. Hence,

the two relationships between verification models create a strong tendency in software maintenance to keep PIM design models, verification models, and the PSM consistent. Hence, QDD makes it attractive for developers to foster their design models.

5 Related Work

The idea that proofs about a system can be done on different abstraction levels which are coupled via lemmata, or mappings, is a central idea in abstract interpretation [7]. However, in this context, abstract interpretations are created by hand and not derived from design models. QDD advances here, since it embeds the idea of several abstraction levels for verifications into model-driven development, in which the distinction of design and target models plays an important role. By identifying an abstract interpretation as a specific platform, MDA and abstract interpretation can be unified. Of course, this paper has only done a very first step in this direction, much more work is needed.

Heyer's work on user-aided verification delivers a framework for user assertions in functional verification [9]. He develops a technology to bridge proof gaps in Dijkstra's weakest precondition calculus by user assertions. If a fact cannot be proven automatically by the proof system, but is evident, the user can assert it, and the overall proof can continue. In essence, this method seems to reach more far than fully automatic verification of systems. Although few experiments have been performed with this technology, it seems to gain more importance in model-driven verifications.

6 Conclusions

We have shown that MDA for quality-aware software needs to be extended by model-driven verification (MDV) to quality-driven development (QDD). To our knowledge, QDD is the first method for MDA in quality-aware application domains, with the following advantages. With integrated *design verification* in model-driven development, we are able to detect errors of real-time PIMs early. Traditional methods, testing or code verification, discover mistakes very late. Since the later a mistake is found, the more expensive it is to correct, QDD should give a leading edge to increase developer productivity.

Next, verification should become easier, if done in parallel on several different abstraction levels. Developers should be able to detect errors that are very hard to find using traditional testing. On the one hand, assertional assumptions in a PIM facilitate the design verification and can be proven in the code verification later on. On the other hand, design knowledge asser-

tions simplify code verification, or enable it, when it was impossible otherwise. Also, these strong relationships between the verification models and the code (PSM) create an incentive to keep design and implementation consistent, thereby tackling design aging.

Lastly, QDD provides an elegant extension of MDA for quality-aware application domains. When regarding a verification tool as a platform, the verification engine becomes rather similar to a runtime platform. This gives QDD the flavor of unifying verification techniques and platform-oriented development into one uniform model-driven framework.

References

[1] Rajeev Alur. Timed automata. In *Computer Aided Verification*, pages 8–22, 1999.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL in 1995. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 431–434, 1996.

[4] S. Bonnier and T. Heyer. COMPASS: A comprehensible assertion method. *Lecture Notes in Computer Science*, 1214:803–??, 1997.

[5] The HIDOORS Consortium. The hidoors uml profile for high-integrity distributed object-oriented real-time systems. http://www.hidoors.org.

[6] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*. Springer, 1992.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[8] Alexandre David, M. Oliver Moller, and Wang Yi. Formal verification of UML statecharts with real-time extensions. In *Fundamental Approaches to Software Engineering*, pages 218–232, 2002.

[9] Tim Heyer. *Semantic Inspection of Software Artifacts From Theory to Practice*. PhD thesis, Linköping University, 2001.

[10] Karlsruhe James Hunt. Forschungszentrum Informatik. High-integrity distributed object-oriented real-time systems. http://www.hidoors.org.

[11] Object Management Group (OMG). *UML Profile for Schedulability, Performance, and Time Specification*, 2002. http://www.omg.org/uml.

[12] Object Management Group (OMG). *MDA Guide*, 2003. http://www.omg.org/mda.