Model-Based Development of Embedded Systems*

B. Schätz[1], A. Pretschner[1], F. Huber[2], and J. Philipps[2]

[1] Institut für Informatik, Technische Universität München
Arcisstr. 21, 80290 München, Germany
[2] Validas AG
Software-Campus, Hanauer Str. 14b, 80992 München, Germany

Abstract. Model-based development relies on the use of explicit models to describe development activities and products. Explicit process and product models allow the definition and use of complex development steps that are correct by design, the generation of proof obligations for a given transformation, requirements tracing, and documentation of the process. We argue that the concept of model-based development is orthogonal to a specific process, be it agile or rigorous.

1 Introduction

Intuitively, model-based development means to use diagrams instead of code: Class or ER diagrams are used for data modeling, Statecharts or SDL process diagrams abstractly specify behavior. CASE tool vendors often praise their tools to be model-based, by which they mean that their tools are equipped with graphical editors and with generators for code skeletons, for simulation code, or even for production code.

However, we do not believe that model-based development should be regarded as the application of "graphical domain-specific languages". Instead, we see model-based development as a paradigm for system development that besides the use of domain-specific languages includes *explicit* and *operational* descriptions of the relevant entities that occur during development in terms of both product and process. These descriptions are captured in dedicated models, *process models* and *product models*.

We believe that many important problems in industry like the coupling of different tools for different development aspects (e.g., data aspects, behavior aspects, scheduling and resource management aspects) are still unsolved because of a lack of an underlying coherent metaphor. We see explicit product and process models as a remedy to this problem. In this paper, we provide a rather abstract treatment of our understanding of model-based development. An extended version of this paper has been published as a technical report [11]. As application domain, we choose that of embedded systems, but the general ideas apply to other domains as well.

* This work was in part supported by the DFG (projects KONDISK/IMMA, InOpSys, and Inkrea under reference numbers Be 1055/7-3, Br 887/16-1, and Br 887/14-1) and the DLR (project MOBASIS).

2 Models

The shift from assembler towards higher languages like C or Ada essentially reduces to the incorporation of abstractions for control flow (like alternative, repetition, exceptions), data descriptions (record or variant types), and program structure (modules) into these higher languages. We consider model-based development to be a further step in this direction, using domain-specific abstractions. In the field of embedded controllers, the concepts of capsules and connectors of, e.g., the UML-RT are used as well as state machines to describe component behavior. That these abstractions have intuitive graphical descriptions is helpful for acceptance, but not essential for the model concept.

What are the advantages of model-based development? One advantage is independence of a target language: Models can be translated into different languages like C or Ada for implementation. The key advantage, however, is that the product model, which subsumes the abstract syntax of a modeling language, restricts the "degrees of freedom" of design in comparison with programming languages. This is akin to modern programming languages that restrict the degrees of freedom of assembler languages by enforcing standard schemes for procedure calls, procedure parameters and control flow. Ada subsets like Ravenscar or SPARK explicitly restrict the power of the language, e.g., in terms of tasks.

Model-based development incorporates the aspects of *abstraction* and *restriction* in high level languages. This happens not only at the level of the product but also at the level of the process. Working with possibly executable models not only aims at a better understanding and documentation of requirements, functionality, and design decisions. Models may also be used for generating simulation and production code as well as test cases. We consider the integration of different models at possibly different levels of abstraction as the key to higher quality and efficiency of the process we propose. In an incremental development process, increments (or parts of a product) have to be integrated over time (Figure 1), and this integration is concerned with both products and processes, on a horizontal as well as a vertical level.

Horizontally, different aspects have to be integrated. These aspects reflect a separation of concerns by means of abstractions. They deal with concepts like structure, functionality, communication, data types, time, and scheduling. Structural abstractions concern logical as well as technical architectures, and their relationship. Functional abstractions discard details of the actually desired behavior of the system. Communication abstractions allow the developer to postpone decisions for, e.g., hand-shaking and fire-and-forget communications. Data abstractions introduce data types at a level of granularity that increases over time. Timing and scheduling abstractions enable the developer to neglect the actual scheduling of components in early development phases.

Vertically, different levels of abstraction for each of the above aspects have to be brought together in a consistent manner. This applies to both integrating different structural abstractions and integrating structure with functionality

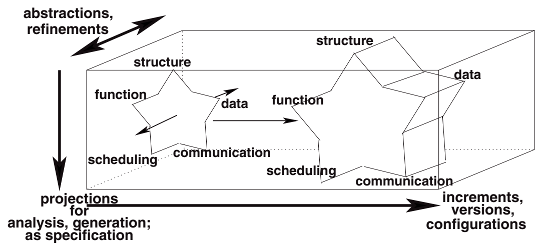


Fig. 1. Model-Based Development

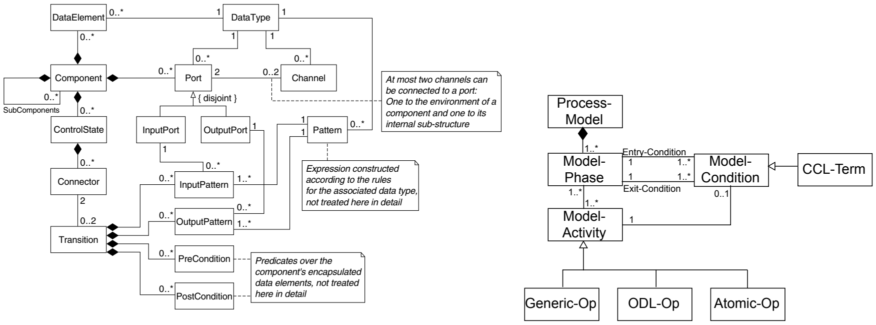
and communication. Furthermore, different levels of abstractions in all areas have to be interrelated: Refinements of the black box structure have to be documented and validated, and the same is obviously true for functional and data refinements.

Process and product models. In the UML, the notion of a model is used to describe the elements and concepts used during the development process, e.g., class, state, or event. Since, however, this distinction is too coarse for the description of the model-based approach, here more fine-grained notions of models will be used: *process, product, conceptual,* and *system* models. In the following, these models are explained in more detail and related to each other. We use the domain of embedded systems development and the CASE tool AutoFocus [7] with its UML-RT-like description techniques for illustration.

The first two models are used to describe the development process from the engineer's and thus the domain model point of view. Together process and product models form the *domain* model:

Process model: The process model consists of the description of *activities of a development process* and their relations. Because of the explicit description, activities are *repeatable*, *undoable* and *traceable*. In the domain of embedded reactive systems, e.g., the process model typically contains modeling activities ("define system interface", "refine behavior") as well as activities ("generate test cases" or "check refinement relation"). The activities are related by their dependency between them defining a possible course of activities throughout the development process.

Product model: The product model consists of the description of those aspects of a system under development *explicitly* dealt with during the development process and handled by the development tool. For embedded systems, a product model typically contains domain concepts like "component", "state" or "message", as well as relations between these concepts like "is a port of a component". In addition to these more conceptual elements, used for the description of the product, more semantically oriented concepts like "execution trace" are defined to support, for example, the simulation of specification during the development process. Finally, it contains process oriented product concepts like "scenario" or "test case", supporting the definition of



At most two channels can be connected to a port: One to the environment of a component and one to its internal sub-structure

Expression constructed according to the rules for the associated data type, not treated here in detail

Predicates over the component's encapsulated data elements, not treated here in detail

Fig. 2. Simplified Conceptual Product Model (left) and Process Model (right)

process activities. All activities in the process models are defined in terms of the entities in the product models.

3 Product

The product model describes the aspects, concepts and their relations needed to construct a product during the development process. Thus, it supplies the 'language' to describe a product. Usually, this language is represented using view based description techniques like structural, state oriented, interaction oriented, or data-oriented notations. The concrete product itself is an instance of the product model. Since process activities are defined as changes of such instances of the product model, a process model can only be defined on top of a product model. The granularity of a product model also defines the expressiveness and thus the quality of the process model. Using both models, detailed development processes can be described, accessible to CASE support.

3.1 Structure of the product model

While the 'abstract syntax' is sufficient to describe conceptual relations of abstract views or the functionality of modeling activities during the process, a semantical relation is needed to define or verify more complex semantical dependencies of views as well as properties of activities (like refining activities or activities not changing the behavior as, e.g., refactoring). Since the semantical and the conceptual part of the product model are used differently in the model-based approach, the product model is broken up into two sub-models:

Conceptual model: The conceptual model consists of the *modeling concepts* and their relations used by the engineer during the development process. The conceptual model is independent of its concrete syntactic representation used during the development process. Typical domain elements for embedded systems are concepts like "component", "port", "channel", "state", or "transition". Typical relations are "is_port_of", "is_behavior_of", etc. Figure 2 shows a simplified part of the AutoFocus conceptual model.

System model: The system, or semantical, model consists of *semantical concepts* needed to describe the system under development. A typical element is "execution sequence". Typical relations are "behavioral refinement" or "temporal refinement".

A semantic interpretation is assigned to the instances of the conceptual model by instantiating the system model according to its relation to the conceptual model. In this way, for instance, refinement relations can be proved, or the validity of timing constraints can be checked.

3.2 Application of models

The purpose of the product model is to support a more efficient and sound development process by providing a domain-specific level of development. For the engineering process, the model is used *transparently* through views of the model in form of description techniques, e.g., graphical diagrams, and interaction mechanisms supporting the development process. Two mechanisms can be used: *consistency conditions* and the *definition of process activities.* Consistency conditions exist at three different levels:

Invariant conceptual consistency conditions: They are expressible within the conceptual model, and hold invariantly throughout the development process. Therefore, they are enforced during construction of instances of the conceptual model. They can be defined, e.g., as multiplicities of relations of the conceptual model. Examples are syntactic consistency conditions as used in AutoFocus like "a port used during the interaction of a component is part of its interface".

Variant conceptual consistency conditions: Like the invariant conceptual conditions, these conditions can be expressed completely within the conceptual model. However, unlike those, they may be relaxed during certain steps of the development process and are enforced during others. Examples are methodical consistency conditions like "All transitions leaving a state have disjoint patterns thus ensuring deterministic behavior".

Semantic consistency conditions: These conditions are not expressible in the conceptual model. Since, generally, they cannot simply be enforced, the validity of these conditions is not guaranteed throughout the development process but must be checked at defined steps of the process. An examples is the semantic condition: "The glass box behavior of a component refines the black box behavior".

In general, the distinction between invariant and variant conceptual consistency conditions is a matter of flexibility and rigorousness of the development process supported by the underlying model. In AutoFocus we use CCL (Consistency Constraint Language, [7], [12]) to define conceptual consistency conditions. Similar to the OCL, it corresponds to a first order typed predicate calculus with the types (classes) and relations (associations) of the conceptual model. Expressions are evaluated using an instance of the conceptual model as universe, returning all counterexamples of the current instance of the conceptual model.

4 Process

As mentioned above, the justification of the product model is its application in the definition of a process model. By the use of a detailed product model we can (1) give a detailed definition of the notions of *phase* and *activity* in terms of how they interact with the conceptual model, (2) increase the soundness of the development process by introducing semantical consistency conditions or sound activities with respect to the system model, and (3) most importantly, add CASE support to the process to increase efficiency of development.

4.1 Structure of the Process Model

As shown in Figure 2, a simplified process model consists of

Phases: Phases define a coarse structure of a process. They can be associated to conditions that must be satisfied before or after the phase. Each phase has an associated set of activities that can be performed during this phase. A typical example is "Requirements analysis", and a simplified example for a corresponding condition is "Each requirement must be mapped to an element of the domain model" to hold at the end of requirements analysis. Using the consistency mechanism, development is guided by checking which conditions must be satisfied to move on in the process.

Activities: In contrast to the unstructured character of a phase, an activity is an operationally defined interaction with an instance of the conceptual model and thus executable. An activity can be extended with a condition stating its applicability at the current stage for user guidance. An activity is either a generic operation generated from the conceptual model, an atomic operation supplied by the system model, for example checking semantic consistency, or a complex operation constructed from the basic operations.

In more complex processes, phases and activities usually consist of sub-phases and sub-activities, respectively. Since phases and activities are defined in terms of the product model, their dependencies can be expressed in terms of the product rather than in generally unspecific ways as found in general process description languages [3].

Examples for basic operations include simple construction steps like "introduction of a new transition into the state-description of a component". Complex operations include refactoring steps like "pull up a subcomponent out of its super-component to become a component of the same level" (involving a change in the subcomponent relation and a relocation of ports and channels).

Process activities generally consist of a collection of simple and complex operations to be applied during an activity. Complex operations can be defined in the form of extended pre/post-conditions, describing a transformation of instances of the conceptual model. In the AutoFocus approach these operations can be defined in ODL (Operation Definition Language, [12]), an extension to the OCL-like CCL introduced above. ODL allows to precisely define the pre-

and postconditions (including user interaction) of an operation in terms of the instance of a conceptual model. ODL definitions are executable but come in a logical form that supports the verification of conceptual properties of the operation.

To illustrate how process, conceptual and system models interact, we use the example of the "elimination of dead states" refactoring step that reduces code size. In this example, a control state including all its adjacent transitions is removed from an automaton provided this state is marked as unreachable.

Process: On the level of the process model, an activity "Show unreachability of a state" is introduced. This may either be an atomic operation (carried out, e.g., by some model checking algorithm) or a complex operation requiring user interaction. Furthermore, the activity "Remove dead state" removes all transitions leading to a state marked unreachable as well as the state.

Conceptual: On the level of the conceptual model, the concept of "unreachability" of a state, e.g., as a state annotation, is introduced. If no user interaction is required for the proof of unreachability, this is sufficient. Otherwise conceptual elements of proof steps must be added.

System: On the level of the system model the semantics have a direct effect: in case of an atomic operation, there is an operational notion of the semantical predicate "unreachability", e.g., in form of a model-checking algorithm. In case of an operation requiring user interaction, the system model is used via atomic operations corresponding to operational relations of the system model (e.g., combining parts of a proof, applying modus ponens).

Since an activity as the atom of a process describes how a product is changed, an activity can be understood as a process pattern in the small. Additionally, each activity is described in an operational manner. Furthermore, by the use of the system model, properties like "soundness considering behavioral equivalence" or "executability of the specification" can be established for activities and phases. This combination of user guidance by consistency conditions, of executable activities, and of the possibility for both arbitrary and provably sound process activities and states of a product, is directed at improving the efficiency of the CASE-based development process.

4.2 Agility

Model-based development is orthogonal to the degree of rigorosity of a process. Without giving a clear definition of agile processes, we illustrate this claim by embedding aspects of Extreme Programming [1] into model-based development.

Language: While generally associated with Java, XP itself does not prescribe any particular language, and we might thus use any model-based formalism appropriate for a given application domain. XP clearly is code centered. However, since we advocate the use of operational models not only for documentation purposes, in this context there is no qualitative difference between using a low level and a high level language. One of our main points, the restriction of general purpose languages, is orthogonal to this aspect.

Testing: One of the core ideas of XP is to continually test the system under development. In fact, test cases are the only formal means of specification. Incorporating explicit process activities for testing and implementation steps does by no means contradict the principles of XP nor those of model-based development.

Refactoring: The activity of refactoring is a common technique in incremental processes like XP. Refactoring is the activity of restructuring an artifact (code) without altering its behavior. Within the model-based approach, these refactoring steps may well be incorporated in the process model as explicit process activities as well.

Augmenting agile approaches like XP by model-based constructs is likely to pay off. Firstly, if one accepts domain specific languages to be a good choice, then it is a good choice for agile methods as well. Secondly, every automatic approach to test case generation clearly requires a clear and explicit understanding of the semantics of the language that is used. Thirdly, refactoring relies on behavioral equivalence: An explicit semantics must thus complement the intuitive notion when building, for instance, safety critical systems. The definition or automatic generation of test cases for checking behavioral equivalence of the system with its refactored counterpart obviously requires this semantics as well. Assigning explicit meaning to the language the XP engineers employ would thus result in an explicit system model which could be linked to the refactoring patterns in the process model.

Intuitively, model-based development seems to imply rather rigorous processes. However, depending on the rigor of the defined process activities, it can support very flexible processes as well. Thus, applying the model-based approach to XP, this could result in a less code-centered but still flexible 'Extreme Modeling' approach.

5 Related work

Though widely used, we are not aware of explicit definitions of "model-based development". Especially, this term is often used in a more restricted sense, e.g., domain oriented software architectures.

The approach presented by Sgroi et el. [13] and Keutzer et al. [8] is similar to ours, in terms of the incorporation of different levels of abstraction, separation of concerns—in particular, computation and communication—, and the emphasis on explicit system models. The especially CASE relevant distinction of models within a layered approach as we propose it complements their line.

While clearly code centered, aspect oriented programming [9]—or, more generally, separation of concerns—is similar in its vision w.r.t. finding abstractions for aspects like concurrency, exception handling, etc. Differing from our approach, the idea is to incorporate these abstractions into unrestricted—and therefore difficult to analyze—general purpose languages like Java or C rather than to use dedicated domain specific languages. In its pure form, AOP does not require an explicit product nor process model.

The notion of explicit product and process models is also found in the area of Process Definition Languages [3], however, focusing on user participation and neglecting the importance of a domain-specific, detailed product model to define a process upon.

In the UML, a detailed model of the product is defined, integrating different views of a product. However, semantical relations exceeding the structural relations of the conceptual model are missing. The RUP defines a process on top of the UML. Activities of the process, their preconditions and results, however, are not defined in terms of the UML meta model; rather, the RUP only outlines the phases to be carried out and suggests useful notations for each phase.

Modeling approaches like MOF (Meta Object Facility) rather focus on technical aspects of how to implement and access models and meta models, but do not address their application in defining domain-specific development processes. The OMG's model driven architecture [14] aims at the definiton of platform-independent models in a platform-independent language (UML) that are later mapped to platform-specific models (CORBA, SOAP, etc.). It is thus concerned with the aspect of communication as well as structure as described in Section 2, however focusing on the architecture of a product.

Graphical editors in tools like Together or ArgoUML concentrate on an explicit (UML) meta model but do not take into account a process model.

6 Conclusion

Our vision of model-based development rests on two pillars: Explicit *product models*, which for the developer appear as domain-specific languages, and explicit *process models*, which define the developer's activities that transform early, abstract, partial products to the final, concrete and complete products that are ready to be delivered and deployed.

The *benefits of model-based development* come from the *interaction of process and product models* and their *realization in a CASE tool*: Firstly, *complex design steps* such as refactorings or the introduction of complex communication patterns [11] between components can be naturally defined and performed in a tool. Secondly, the application of such design steps naturally leads to a *development history* that can be recorded in the tool and used for a kind of high-level configuration and version management. Finally, the *requirements and design rationales* that influence design steps can be *traced and documented* throughout the complete development process.

The main goal of model-based development is to produce high-quality software at acceptable cost. While high software quality can be achieved even now—as demonstrated by avionics software—, cost and development time are usually forbidding. Model-based development aims at improving not only the *product*, but also the *process* that leads to the product, making high-quality software development more affordable.

However, model-based development is not without risk. It is not obviously clear whether a seamless development process from early design to final target code is feasible: Challenges, such as, e.g., knowledge of environment properties, which are difficult to formalize, or customer-specific coding and certification standards, can—with varying degrees of difficulty—be solved. However, the main problem is that in contrast with, for instance, compiler construction, they can be solved not by tool builders alone, but only in close cooperation with domain experts: A model-based development will necessarily be domain-specific. Finding common vocabularies and notations to define the conceptual, system and product models is a rather ambitious goal.

Still, in view of our experiences with the AutoFocus project we are optimistic that such tools can be built. Although we do not yet have enough experience with industrial-size projects, we obtained satisfying results with some core aspects of such systems (deployment of systems on 4-bit and 8-bit microprocessors, schematic introduction of security aspects, custom scheduling algorithms to distribute computation effort over time). That the close integration of domain properties into CASE tools is feasible has been demonstrated, for example, for simultaneous engineering in process automation [2].

References

1. K. Beck. *Extreme Programming Explained: Embrace Change.* Addison Wesley, 1999.
2. K. Bender, M. Broy, I. Péter, A. Pretschner, and T. Stauner. Model based development of hybrid systems: specification, simulation, test case generation. In *Modelling, Analysis and Design of Hybrid Systems*, LNCIS. Springer, 2002. To appear.
3. Jean-Claude Derniame, Badara Ali Kaba, and David Wastell, editors. *Software Process: Principles, Methodology and Technology.* Springer, 1999. LNCS 1500.
4. Franz Huber and Bernhard Schätz. Integrated Development of Embedded Systems with AutoFocus. Technical Report TUMI-0701, Fakultät für Informatik, TU München, 2001.
5. K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 19(12), December 2000.
6. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. European Conference on Object-Oriented Programming (ECOOP'97)*, Springer LNCS 1241, 1997.
7. B. Schätz, A. Pretschner, F. Huber, and J. Philipps. Model-based development. Technical Report TUM-I0204, Institut für Informatik, Technische Universität München, 2002.
8. Bernhard Schätz. The ODL Operation Definition Language and the AutoFocus/Quest Application Framework AQuA. Technical Report TUMI-1101, Fakultät für Informatik, TU München, 2001.
9. M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal Models for Embeded System Design. *IEEE Design& Test of Computers, Special Issue on System Design*, pages 2–15, June 2000.
10. R. Soley. Model Driven Architecture. OMG white paper, 2000.