T U M

I N S T I T U T F Ü R I N F O R M A T I K

Model-Based Development

Bernhard Schaetz, Alexander Pretschner, Franz Huber, Jan Philipps

TUM-I0204
Mai 02

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-05-I0204-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Model-Based Development*

B. Schätz, A. Pretschner, F. Huber, J. Philipps
Institut für Informatik, Technische Universität München
www4.in.tum.de/~{schaetz,pretschn,huberf,philipps}

Abstract

Model-based development relies on the use of explicit models to describe the development process including its activities and products. Among other things, the explicit existence of process and product models allows for the definition and use of complex development steps that are correct by design (refactorings), for generating proof obligations after a given transformation (run a certain automatically generated test suite), for requirements tracing, and for documenting the process. Our understanding of model-based development in the context of embedded systems is exposed. We discuss domain-specific modeling languages, and argue for machine support in model-based development.

1 Introduction

Intuitively, model-based development means to use diagrams instead of code: Class or ER diagrams lend themselves to 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 mere 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

*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).

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 allow the description of design activities. Because of the explicit description, activities are *repeatable*, *undoable* and *traceable*. Activities include low-level tasks like renamings and refactorings, but also higher-level domain-specific tasks like the deployment of abstract controller functionalities on a concrete target platform.

Product models contain the entities that are used for the description of the artifact under development and the necessary parts of its environment, as well as the relations between these entities.

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

Overview. In this paper, we provide a rather abstract treatment of our understanding of model-based development. As application domain, we choose that of embedded systems, but the general ideas apply to other domains as well. The article's remainder is organized as follows. We kick off with the basic idea of explicit process and product models in Section 2. The essence of product and process models is described in Sections 3 and 4, respectively. In Section 3, we explain in more detail how model-based development naturally lends itself to consistency-by-design and supports the development process.

The incorporation of this paper's ideas in a CASE tool is sketched in Section 6. Related work is presented in Section 7, and Section 8 concludes. We assume some familiarity with the basic concepts of description techniques like the UML, and the UML-RT, or ROOM, respectively, as well as some basic knowledge of SCR, the RUP, and AOP. Knowledge of these processes or paradigms is, however, not necessary for understanding the fundamental ideas.

2 Models

This section gives a first overview of our understanding of model-based development. We briefly discuss (a) restriction of the power of general purpose

languages as a key to intellectual mastery of the artifacts under development (Sec. 2.1), (b) separation of concerns (Sec. 2.2), (c) the necessity of explicit process and product models (Sec. 2.3), and (d) the different layers of these models (Sec. 2.4).

2.1 Abstraction and restriction

The shift from assembler towards higher languages like C or Ada essentially reduces to the incorporation of abstractions for control flow (sequence, alternative, repetition and—in modern languages—exceptions), data descriptions (record and variant types), and program structure (modules) into these higher languages [23, 41]. Garbage collectors or middleware like CORBA or .NET are further examples of increasingly abstract development. We consider model-based development to be a further step in this direction. It aims at higher levels of domain-specific abstractions as seen, at a low level, in the abstraction step that was performed in `lex`. In the field of embedded controllers, the concepts of capsules, ports, and connectors of, for instance, the UML-RT are used as well as state machines for the description of component behavior. That these abstractions have intuitive graphical descriptions is helpful for acceptance, but not essential for the model concept. Furthermore, in model-based development there is no need to exclusively rely on one particular description technique, or rather the underlying 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. For graphical simulation, other languages (Java or dedicated multimedia languages) are likely better suited. Again, this is in analogy with the abstraction step, or, inversely, compilation of programming languages: C code can be translated into a number of different assembler languages.

The key advantage, however, is that the product model, i.e., for now roughly 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 design freedom of assembler languages by enforcing standard schemes for procedure calls, procedure parameters and control flow. In a similar sense, Java restricts C^{++} by disallowing, among other things, explicit pointers and multiple inheritance. Ada subsets like Ravenscar or SPARK [1] explicitly restrict the power of the language, e.g., in terms of tasks. The reason is that these concepts have proved to yield artifacts that are difficult to master.

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.

2.2 Separation of concerns

The essence of model-based development is an integrated development with models, i.e., abstractions. Working with possibly executable models not only aims at understanding requirements and better 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. 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 (deployment) architectures, and their relationship. Functional abstractions may discard details of the actually desired behavior of the system. Communication abstractions allow the developer to postpone decisions for synchronous and asynchronous, or hand-shaking and fire-and-forget communications. Data abstractions allow the engineer to work with data types at a level of granularity that increases over time and that helps in building functional, communication, and structural abstractions. Finally, timing and scheduling abstractions enable the developer to neglect the actual scheduling of components—or even abstract away from timing by relying solely on causality—in early development phases. Other aspects like security, fault tolerance, or quality-of-service must, in certain applications, be considered as well. We are aware that these aspects are not entirely orthogonal one from another. However, thinking in these terms allows for better structuring systems. We will get back to this issue in Section 5.1 with an example.

Vertically, different levels of abstraction[1] 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

[1]Note that the term *abstraction* is used in an ambiguous manner: abstractions in the mathematical sense (i.e., formally, implications) and abstractions on a conceptual level where constructs for describing one view of a system are considered (i.e., ontological entities).

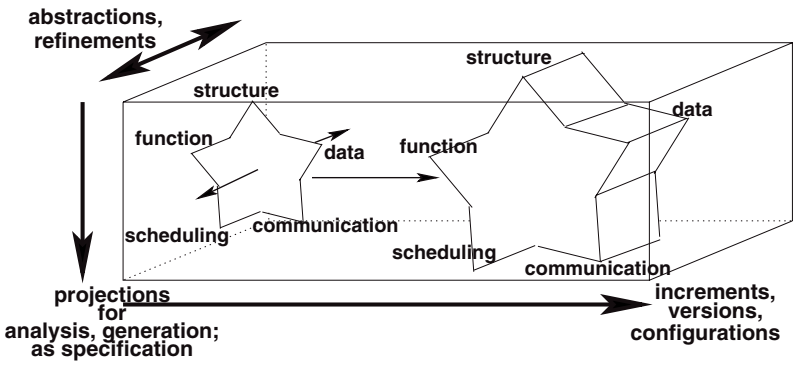


Figure 1: Model-based development

with functionality 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. Since in a sense, possibly informal requirements also constitute abstractions, tool supported requirements tracing is a must for such a model-based process.

In an incremental development process, increments (or parts of a product) have to be integrated over time (Figure 1). While this figure suggests that the concepts of level of abstraction and increments are orthogonal, one might well argue that a refinement step does constitute an increment. The reason for the distinction is that abstractions and refinements form special increments the correctness of which might, in a few cases, be proved or automatically tested.

2.3 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 [16] with its UML-RT-like description techniques for illustration. Figure 2 gives an overview of the models under consideration and their relation.

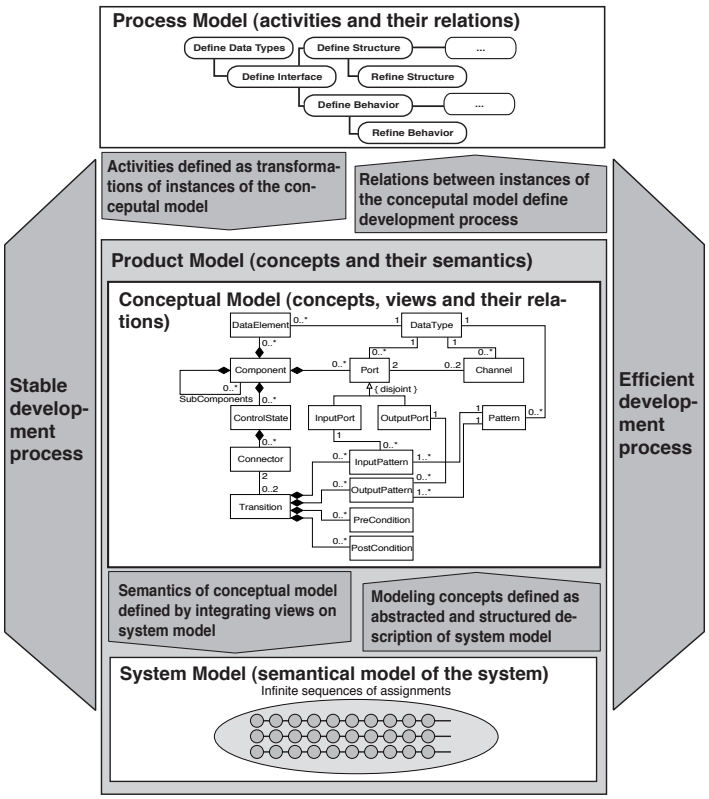


Figure 2: Models in the CASE development process

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

Process model: The process model consists of the description of *activities of a development process* and their relations. In the domain of embedded reactive systems, for instance, the process model typically contains modeling activities (e.g., "define system interface", "define behavior", "refine behavior") as well as activities like "generate scenarios or test cases", "check refinement relation", or "compute upper bound for worst case execution time". The activities can be related using a dependency relation between activities and thus a possible course of activities throughout the development process. By relating them to a product model, process patterns can be formalized as activities and thus integrated in the process model.

Product model: The product model consists of the description of those aspects of a system under development *explicitly* dealt with during the

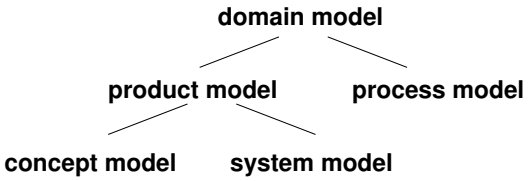


Figure 3: Models

development process and handled by the development tool. For embedded systems, a product model typically contains domain concepts like "component", "state", "variable", or "message", as well as relations between these concepts like "is a port of a component", etc. In addition to these more conceptual elements, used for the description of the product, more semantically oriented concepts like "assignment of variables" or "execution trace" are defined to support, for example, the simulation of specifications during the development process. Finally, it contains process oriented product concepts like "scenario" or "test case", supporting the definition of process activities.

Process and product models together form the domain model. Figure 3 shows the relationship between the different kinds of models.

2.4 Model levels

As mentioned above, models are used in a model-based development

- to structure and restrict the development process and the products under development,
- and to support the generation of the instances of models (the products of the process).

To decide what models (and instances) are needed in a model-based approach, two different points of views an be taken: the *methodical*, and the *CASE-oriented* points of view. As mentioned above, from a methodical point of view, models are useful at three levels:

Process level: The entities and relations on the process level describe activities of the development process and their relations. They are used to guide the process. Their definition is done on top of the elements of the conceptual model. Section 4 describes the process model in more detail.

Conceptual level: The entities and relations on the conceptual level describe the elements used to construct products. They are used to describe the product, and they are visualized by means of description techniques. Section 3 contains examples of these entities and relations.

System level: The entities and relations on the system level describe the meaning of the conceptual model. The system model is used to define process activities or to describe and validate their properties. Section 3 treats this in more detail.

From a CASE-oriented point of view, models (and instances) span the following three categories:

Meta level: The meta model is defined at (or before) 'build time' of a CASE tool to describe how domain specific models can be built. It is needed for a generic approach, i.e., if different domain-specific instances of the CASE tool are built. Models of the meta model level are generally domain-independent. For the process as used in the AutoFocus approach, a process meta model is shown in Figure 7.

Domain level: The domain level is defined at 'build time' of a CASE tool to describe the process activities supported by the tool, its modeling concepts, and their meaning. To support a reasonable development process, domain-specific restrictions are necessary. Thus, all models of this level are—more or less—domain-specific. The models of the domain level (i.e., domain process model, domain concept model, and domain system model) are instances of corresponding meta models. A simplified version of a conceptual model as used in AutoFocus is shown in Figure 5.

Instance level: The instance level is defined at 'run time' of a CASE tool to describe the actual activities that are are performed as well as the products that are built. From the point of the engineer who uses the CASE tool, it is the level of the product under development. The instances at this level (a specific development, a product description, and the semantics of this description) are instances of models of the domain level. Figure 6 illustrates how instances of the conceptual model are related to the conceptual model.

Figure 4 shows examples of models and instances according to these two different dimensions.

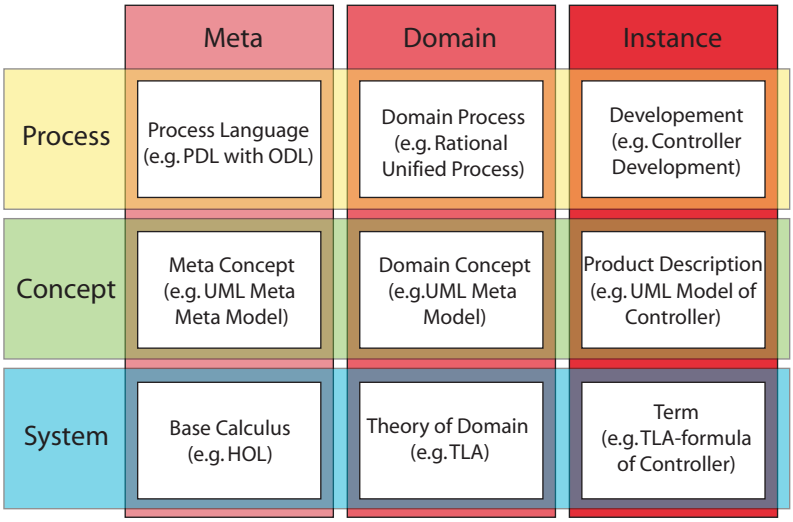


Figure 4: Models and instances

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, for example represented by system structure diagrams, state transition diagrams, or MSC-like event traces.

Since process activities are defined as changes of 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).

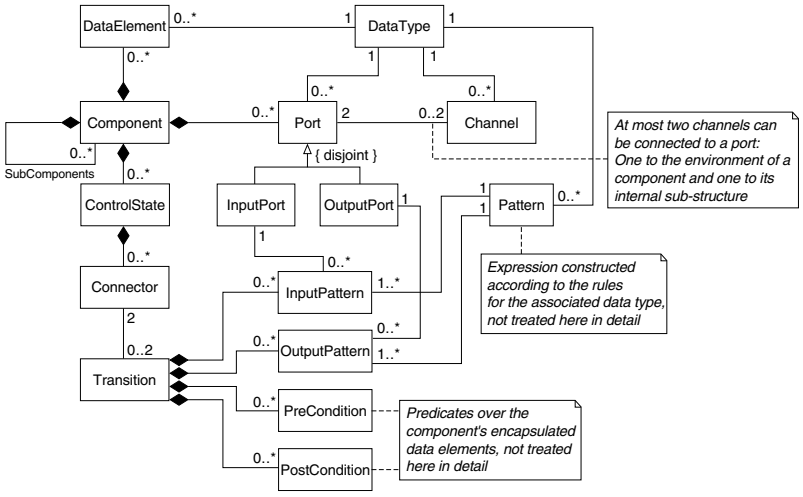


Figure 5: Simplified version of a conceptual product model

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 instances of the conceptual model are the descriptions of the system under development constructed by the engineer. Thus, from a CASE-oriented point of view, the conceptual model is the 'data model' of products explicitly handled by the tool. Therefore, the conceptual model can be described as a class diagram (see 5 for a simplified version) extended with conceptual consistency conditions as described below.

The conceptual model is independent of its concrete syntactic representation used during the development process. Typical domain elements of a conceptual model for embedded systems are concepts like 'component', 'port', 'channel', 'state', 'transition', etc. Typical relations between concepts are 'is_port_of', 'is_behavior_of', 'is_substate_of', etc. Figure 5 shows a simplified part of the AutoFocus conceptual model. Besides these low-level concepts, concepts like 'requirement' or 'test case' including relations like 'discharged_by' or 'is_test_case_of' as well as counterparts of semantical relations like 'is_refinement_of' are included.

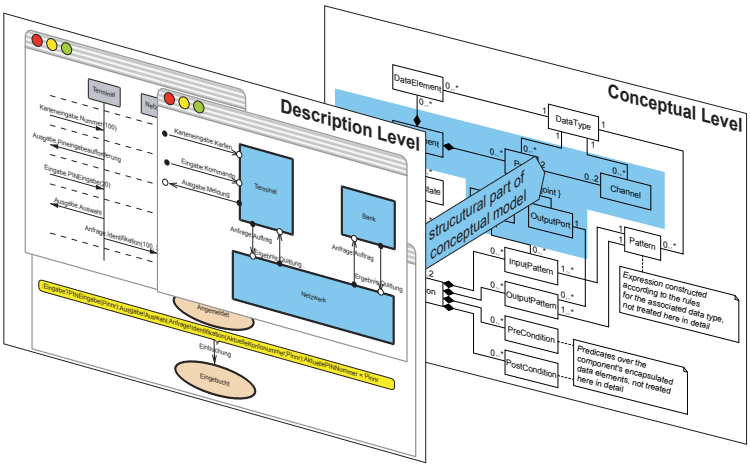


Figure 6: Models and Views

System model: The system, or semantical, model consists of *semantical concepts* needed to describe the system under development. These semantical concepts are given in form of a theory, e.g. an extension of Higher Order Logic. By mapping instances of the conceptual model (i.e., system descriptions) to instances of the system model (i.e., terms of the theory), a semantics of the descriptions is defined. The system model is outside of the CASE tool and used either at 'build time' or at 'run-time' through atomic operations (like in Subsection 5.2).
Typical elements expressed in this theory are 'variable assignment' or 'execution sequence'. Typical relations are 'behavioral refinement' or 'temporal refinement'.

As shown in Figure 6, the notion of the conceptual model is closely related to the notion of views and description techniques. Views of a product correspond to abstractions of an instance of the conceptual model (e.g., horizontally: structure, communication; vertically: component, subcomponent) and are represented using description techniques.

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 as described above 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: These conditions are expressible within the conceptual model. They hold invariantly throughout the development process. Therefore, they are enforced by construction of instances of the conceptual model. Since generally only simple consistency conditions are enforced as invariant consistency conditions, they can be defined 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 the interface of the component' or 'a channel and its adjacent ports have the same types'.

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 'the dependency graphs of variable assignments are non-circular' in SCR, or 'all transitions leaving a state have disjoint patterns thus ensuring deterministic behavior' in AutoFocus.

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. Examples are semantic conditions as in AutoFocus: 'the behavior of an event trace of a component is a refinement of the behavior of the component', or 'the timing behavior of a component respects its worst case time bounds'.

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 the AutoFocus approach we use CCL (Consistency Constraint Language, [31]) 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. AutoFocus offers an evaluation mechanism for CCL expressions returning all counterexamples of the current instance of the conceptual model.

Variant conceptual consistency conditions as well as generic primitive operations of the conceptual model (introducing and removing instances of elements and relations) provide the base operations needed to access the conceptual model from the process point of view. Together with the semantic operations like checking of semantical consistency conditions they form the basic activities of a development process as explained in more detail in Section 4.

Variant conceptual consistency conditions as well as generic primitive operations of the conceptual model (introducing/removing instances of elements and relations) provide the base operations needed to access the conceptual model from the process point of view. Together with the semantic operations like checking of semantical consistency conditions they form the basic activities of a development process as explained in section 4.

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

- give a detailed definition of the notions of *phase* and *activity* in terms of how they interact with the conceptual model,
- increase the soundness of the development process by introducing semantical consistency conditions or sound activities with respect to the system model, and
- most importantly, add CASE support to the process to increase efficiency of development.

4.1 Structure of the process model

As shown in Figure 7, 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. Typical examples are phases like "requirements analysis" or "module implementation". Simplified examples for corresponding conditions are "each requirement must be mapped to an element of the domain model" to hold at the end of requirements analysis or "each component has an implementable time-triggered behavior" to hold at

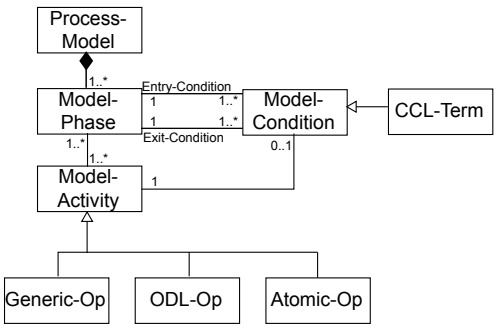


Figure 7: AutoFocus meta process model

the end of the implementation phase. 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 [10].

Examples for basic operations include simple construction steps like "generation of a new state of a component" or "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)" or application of pattern or specification modules as described by Huber and Schätz [32].

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

those operations can be defined in ODL (Operation Definition Language, [31]), 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 like stability w.r.t. consistency conditions or semantical properties like behavioral refinement. Examples of how the process, conceptual, and system levels interact are provided in Section 5.

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 matter. Furthermore, by the use of the system model, properties like "soundness considering behavioral equivalence" or "executability of the specification" can be established for activities or products. This combination of user guidance by consistency conditions, executable activities, and the possibility for both arbitrary as well as sound process activities and states of a product (i.e., activities and states with guaranteed properties), is directed at improving the efficiency of the CASE based development process.

5 Process examples

In this chapter, we illustrate the use of explicit process descriptions. This includes two of the aspects of Sec. 2, namely communication and structure. In Sec. 5.1, we substitute perfect communication by a simple protocol that deals with lossy channels. The operation in question is a refinement step. We then give a second example in Sec. 5.2 that deals with structure. It is shown how a behavior preserving refactoring step may be defined on the grounds of an explicit conceptual model.

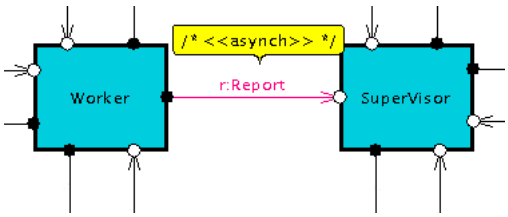
5.1 Communication refinement

Much of the complexity in the design of distributed systems lies in the communication mechanisms between components. Communication paths are often lossy (i.e., not all messages sent do indeed appear at the receiver) or insecure (malicious parties can listen to the communication or even change it); moreover, there is usually a certain time delay between sending and reading of messages.

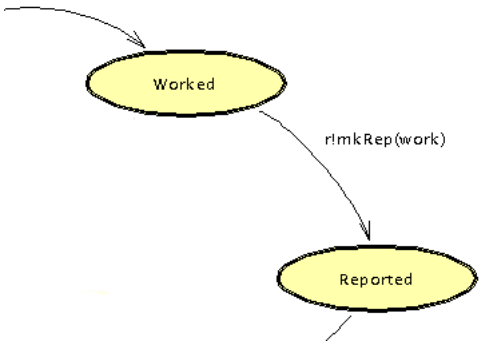
It is methodologically desirable to abstract from these matters in the early system design phases. The logical functionality of a system can be derived first using a simplified communication model (for instance, synchronous

communication based on shared variables as in the current AUTOFOCUS semantics, or directed asynchronous message transfer over buffered channels). Imperfections in the communication media and the means to work around these imperfections are then added in later design steps.

As an example, Figure 8 shows part of a distributed systems with a worker component and a supervisor component. Whenever the worker finishes a subtask, it transmits a report to the supervisor over the (asynchronous and buffered) channel.



(a) System Structure



(b) Fragment of Worker Behavior

Figure 8: Simple Communication System

Is the abstraction of reliable communication justified? From the point of view of the system functionality, yes. All reports from the worker should arrive at the supervisor, no reports may be lost. Is this abstraction justified for the implementation? This depends on the infrastructure of the target platform of the system. For example, the report communication may be mapped to operating systems that employ the popular TCP/IP protocols; in this case, the assumption of guaranteed delivery is justified.

On other platforms, the reports may need to be transmitted over less reliable mechanisms, for instance using infrared communication without the benefit of an elaborate protocol stack that guarantees message delivery. In this case, reliable communication must be ensured by the system itself. A simple and popular protocol that ensures reliable delivery is the alternating bit protocol (ABP). Figure 9 shows the system extended by sender and receiver components for the ABP. Note that the channels between the worker and sender and between the receiver and supervisor components are modeled as reliable channels, while the channels between sender and receiver are lossy. Obviously, later deployment steps have to ensure that the boundary between processing nodes is between sender and receiver, and not between the other components.

In addition to the new channels and communication ports, a new message type has to be introduced; it holds the reports by the worker as well as the message bit for the protocol; bits are assumed to be synonymous with the Boolean type defined by the two constants `True` and `False`:

```
data BitReport = BR(Report, Bit)
```

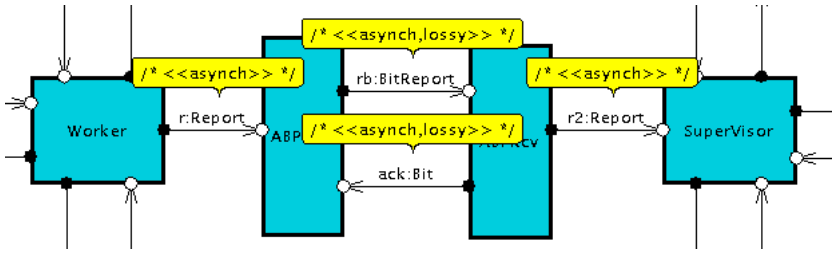
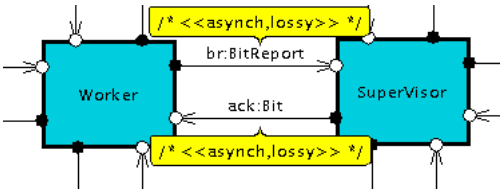


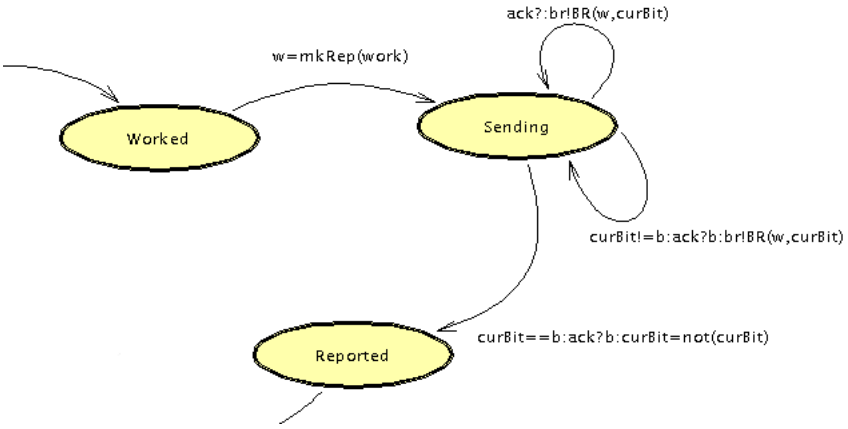
Figure 9: Communication system with ABP components

Explicit sender and receiver components clutter up the system description. A better approach might be to directly change the behavior of worker and supervisor so that they implement the protocol themselves, as shown in Figure 10.

Note that while these transformations might seem simple (partly because of the simplicity of the ABP), they are not trivial since the model transformations are nonlocal. To obtain the system shown in Figure 10, the system structure has to be modified (replacing one report channel by a message and an acknowledgment channel; new communication ports have to be added, the old report ports have to be deleted), a new data type has to be introduced



(a) System Structure



(b) Fragment of Worker Behavior

Figure 10: Communication system with internal ABP

for the pairing of report and the message bit, the state transition diagrams of the worker and supervisor components have to be modified, and both worker and supervisor components have to be augmented with new state attributes for the current bit. All these transformations have to be done consistently.

Based on simple operations on the conceptual model it is rather straightforward to define a compound process activity as an ODL expression that replaces single asynchronous communication channels by lossy channels and the corresponding ABP functionality.

Interestingly, in spite of its simplicity, it is not obvious that this system transformation is correct. A suitable mathematical system model for asynchronous communication is FOCUS [8]; in this model, we expect that the set of possible communication histories over the channels that connect the system to its environment remains unchanged by the transformation. Based on this model, the correctness can be proved, but the proof requires certain assumptions about the properties of lossy channels: In particular, they must be fair (when infinitely many messages are transmitted, then infinitely many messages must arrive at the other end), and they must not reorder message elements (sliding window protocols can be used if a bounded number of message reorderings must be tolerated). Proof techniques for these and similar problems on data flow in distributed systems are presented by Breitling and Philipps [6, 5] as well as by Broy [7].

Other transformation patterns can be imagined that encapsulate cryptographic protocols to replace secure channels by insecure ones. Wimmel and Wißpeintner [40] present some security patterns that are based on such an approach.

Both for guaranteed and for secure message delivery, the separation of functionality and communication aspects helps to keep models concise and leads to a natural notion of incremental system design.

5.2 Refactoring

By a *refactoring* [11] we mean model transformations that leave the behavior of the model essentially unchanged (execution times, for example, might be changed). One example for a refactoring is a change in the hierarchy of the model: While hierarchical descriptions are obviously desirable for complexity reasons, in any incremental design process this hierarchy may need adaption. A process activity should, for example, allow components to be "pulled up" from within somewhere within the hierarchy and to be "pushed down" elsewhere. This activity must ensure that all relevant communication channels to and from this component are also moved; new ports and channels may need to be introduced, named (nontrivial, if done reasonably!) and typed.

Refactoring 1: pull-up component. To illustrate the different levels of models, a very simple form of refactoring is used. This transformation does not change the behavior of the system. We describe how the 'Pull-Up'-refactoring step is treated in our framework. During such a refactoring step a sub-component is removed from a component and adjacently placed on the same level in the structural hierarchy as the component. Note that according to the AUTOFOCUS semantics this transformation leads to no change of the behavior of system.

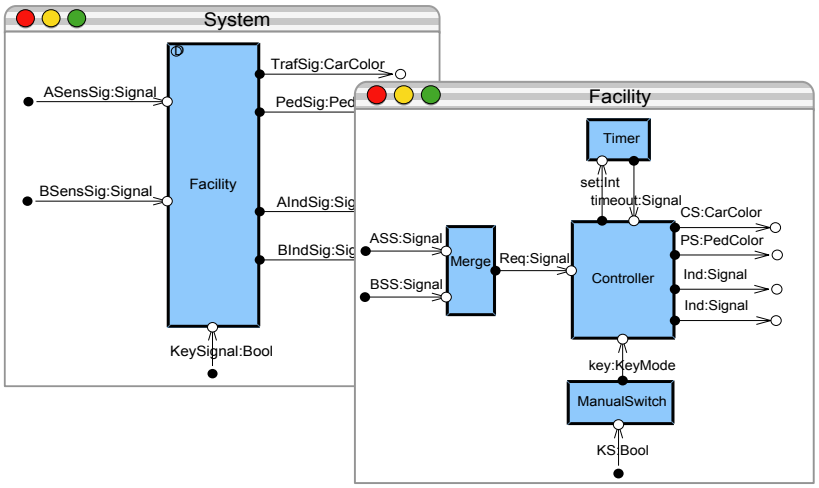


Figure 11: System structure before refactoring

Figure 11 shows a system before the refactoring step. Component ManualSwitch is part of component Facility. During the refactoring it is moved out of Facility and placed next to it. On the instance level of the conceptual model, this requires

- a change of the *component-subcomponent* relation of ManualSwitch (from Facility to System),
- the deletion of a Boolean port in Facility and the channels to and from ManualSwitch as well as the KeySignal-channel in System, and
- the creation of a new KeyMode port in Facility as well as new channels to and from ManualSwitch.

The outcome of the transformation is shown in Figure 12.

Using this example, we will illustrate the differences between the *domain* and the *instance* levels as introduced in Subsection 2.4. Since we do not focus

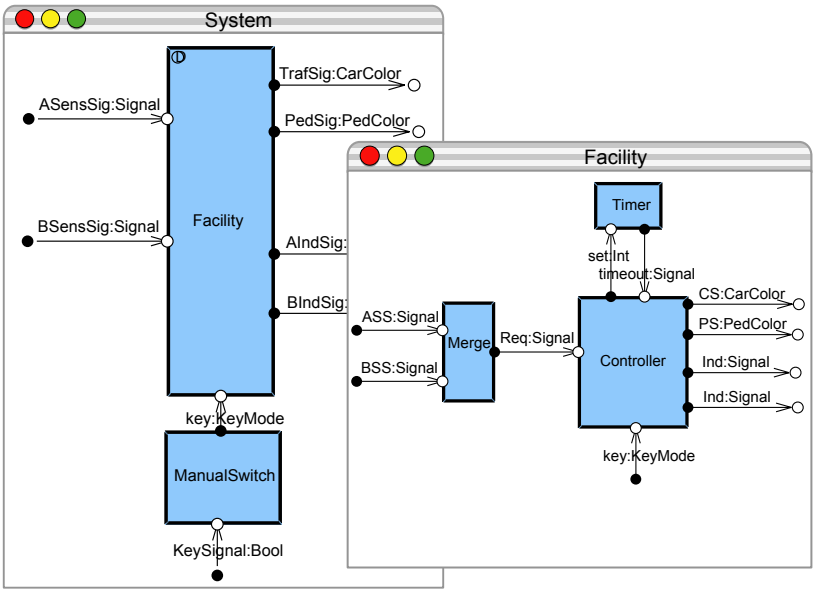


Figure 12: System structure after refactoring

on the definition of a product or a process, the *meta* level will not be treated here:

Domain level: On the domain level, we have all the informations about process and product that is common for the application domain, i.e., is independent of the run of the corresponding CASE tool . This information is used at construction time of the tool:

Process: A domain-specific process model supporting 'Pull-Up'-refactoring offers an appropriate activity for this step. This activity is, e.g., defined in form of an ODL-operation. It can be applied to arbitrary component/sub-component instances by assigning them to the operation's parameters.

Concept: Since the refactoring activity transforms structural hierarchies, the domain-specific concept model contains the corresponding concepts. In case of AutoFocus, these are the elements and relations of the System Structure Diagrams, i.e., components, ports, and channels as well as the sub-component relation.

System: The domain-specific system model defines a semantical model tailored towards the specific needs of the domain. The AutoFocus system model defines how the (infinite) set of (infinite)

execution traces describing the interface behavior is associated with each component. More formally, execution traces are expressed in terms of (infinite) sequences of assignments of values to ports of components formalized using a base theory. Examples for base theories include the Temporal Logic of Actions (TLA, [22]) or Higher Order Logic of Computable Functions (HOLCF, [29]). According to the AutoFocus semantics the behavior of a component is defined compositionally, i.e., associative w.r.t. subcomponent-clustering. This implies that the correctness of the ODL-operation can be proved statically (i.e., at 'build time' of the CASE tool). Such a static proof is, for instance, carried out using a theorem prover like *Isabelle* with the AutoFocus semantics as extension of the HOLCF-theory introduced by Schätz and Spies [34].

Instance level: On the instance level, we have all the information about process and product that is used applying the CASE-tool constructing a product, and thus occurs at run-time of the CASE tool:

Process: On the instance level, a process instance consists of a sequence of applied activities. One of these activities is the application of the 'Pull-Up'-refactoring step to an instance of the conceptual model. Application of this step leads to another instance. Basically, this corresponds to binding elements of the instance of the conceptual model to the argument variables of the ODL-operation. Here, e.g., it corresponds to instantiating the operation with ManualSwitch as the component to be pulled up out of the Facility component, or KeySignal as a channel to be redirected.

Concept: The instances of the conceptual model are the products that are constructed during the development process. Considering the refactoring step, the relevant instances are the products immediately before and after the refactoring. They contain the system structures shown in Figures 11 and 12. The instance prior to the refactoring step contains the sub component to be pulled up (ManualSwitch) and its super component (Facility) including the explicit representation of their component-subcomponent relation. Its counterpart after the application contains both the components and their modified relations. It includes necessary changes like introducing a key-port and key-channel connected to Facility, or deleting the keySignal-port and channel connected to it.

System: Instances of the system model, e.g. HOLCF-terms describ-

ing the behavior of Facility, ManualSwitch, etc., are not needed directly here. This is because the proof of the correctness of the refactoring can be performed statically for the ODL-operation, independently of the actual instance of the conceptual model. Thus, in other words, the proof is given for all possible instances of system models related to the conceptual models before and after the operation.

Refactoring 2: eliminate dead states. To illustrate how process, conceptual, and system models interact when the system model is used at 'run time' of the CASE tool, we use the example of the 'elimination of dead states'-refactoring step reducing code size. In this example, a state (i.e. a control state as well as all posssible data states associated with it) of a state-based behavioral description together with all its adjacent transitions is removed from an automaton if this state is marked as unreachable. Besides changes at the domain level for process and conceptual model, the system model is used at instance level in form of an explicit representation of the semantics of components:

Process: With the domain specific process model, an activity "Show unreachability of state" must be introduced. This step may either be a single atomic operation (and can be carried out, e.g. by some form of model checking algorithm), or a more complex operation requiring user interaction. These operations have counterparts in dedicated relations of the system model. Furthermore, the activity "Remove dead state" removes all transition leading to a state that is marked as unreachable as well as the state itself. This operation can be expressed at the conceptual level, e.g., by using ODL.

Conceptual: On the level of the conceptual model, the concept of 'unreachability' of a state, e.g., as a state annotation, must be introduced. If no user interaction is required for the proof of unreachability, this extension is sufficient. Otherwise, conceptual elements of proof steps must be added, as well as additional concepts like state/assignment of a variable, or the precondition of a transition.

System: On the level of the system model, in contrast to the 'pull-Up'-refactoring step the semantics have a direct effect. To check the unreachability of a state of a component, the semantics of this component (e.g., in form of a temporal logic term) have to be constructed. In case of an atomic operation, there is an operationalized notion of the semantical predicate 'unreachability', for example in form of a model-checking

algorithm. This operation, applied at the instance level, can add the conceptual annotation 'unreachable' to a state that is unreachable according to the semantics of the system model. In case of a more complex operation requiring user interaction, the system model is accessible via atomic operations corresponding to operational relations of the system model. Examples include combining parts of a proof, applying a modus ponens, etc. Besides this application of the system model 'at run-time of the CASE tool', the system model is also applied 'at build-time of the CASE tool' to prove the correctness of this refactoring step.

6 Towards a model-based CASE tool

AutoFocus [2] [17] is a tool for developing graphical specifications of embedded systems based on a simple, formally defined semantics. Embedded systems in AutoFocus are networks of components that communicate via directed channels. AutoFocus supports four different views on the system model to describe different system aspects: structure, behavior, interaction, and data and function definitions. A simplified version of the conceptual product model of AutoFocus is shown in Figure 5.

Early versions of AutoFocus were mainly used for graphical specifications. Supported by the Bundesamt für Sicherheit in der Informationstechnik, a number of verification and validation tools were connected to AutoFocus [38, 37]. Moreover, the tool was extended with code generators for C, Java and Ada and by a test case generator that takes as input an AutoFocus model and a (structural or functional) test case specification, and automatically produces a set of test cases (I/O sequences) [27, 26].

Currently, AutoFocus is a rather generic tool and is mainly used as a high-level design language. We have successfully modeled various embedded controllers, smart cards, and security applications with it, and used to models for code generation, test case generation or formal verification with model checkers.

6.1 AutoFocus II

Based on the experience we gained with AutoFocus, we have started a project to develop AutoFocus II as a true model-based CASE tool which also offers process support. While from AutoFocus we learned—somewhat to our surprise—that a single set of description techniques can be used in

[2] http://autofocus.in.tum.de

widely varying application domains, we believe that AutoFocus II will be more restricted in its application domain.

There are two main reasons for this restriction: Firstly, for fully seamless development, the conceptual model must also include descriptions of the deployment target and its environment. These descriptions are used mainly in order to generate correct bindings to operating systems, device drivers and hardware interfaces, but also as assumptions in quality assurance steps. Obviously, smart cards, phone exchanges and automotive controllers are quite different in this respect.

Secondly, while some process activities like refactoring are generic and useful for any given conceptual model and set of description techniques, other activities will likely depend on established engineering practices in the application domain, or—like the conceptual model—on peculiarities of target platforms and environment.

The application domain of AutoFocus II is not yet fixed, but it will be in the field of distributed automotive electronics. It is easy to imagine a number of relevant process activities, such as refinements or refactorings (see Section 5).

Other process activities affect certain aspects in a system: In the field of automotive electronics, an example would be the distribution of two components onto different processing nodes. This step would, for example, require the modification of the physical communication between the two components and changes in the task scheduling for each node.

Note that for such process activities, their pre- and postconditions must be defined precisely. For his refactorings, Fowler [11] takes a pragmatic viewpoint and ignores semantic issues; the semantics is assumed to be common sense, or trivial.

Given precise conceptual and system models, are higher degree of rigor can and should be achieved. Nevertheless, because of the effort required to check pre- and postconditions in practice, sometimes a more pragmatic approach must be followed. For refactorings, which should not change the essential model behavior, a test suite should be generated which checks the preconditions of a refactoring step, or possibly the equivalence of the original and the modified model.

6.2 Tools for model-based testing

In general, our view of model-based development aims at a seamless development process with production code as the final product: Note that on a fine grained level, code itself is a model.

Besides code generation, also *testing* is an activity that is more and more frequently associated with models (Binder's book [4] is a rather recent and comprehensive example). The traditional idea of model-based testing is to build a model as an abstraction of existing code, an approach also sometimes taken in software model checking [15, 9] in the context of model checking. This model is then used to derive test cases. Most often, this is done manually. If the difference of the abstraction levels of model and code allows it, these test cases are then fed into the implementation. Not only functional test cases can be derived: In Cleanroom [28], for example, stochastic usage models are the core of the testing activities.

We might thus use all the ideas of model-based development in order to get models for test case generation. There are two basic scenarios:

- In the first scenario, models are used only for test case generation, while the actual system has been created independently of the model.

  This is the case if, for organizational reasons, the quality assurance and development departments are separated, if the system is a legacy system that is to be tested in new environments, or if sufficiently efficient and reliable code generators for a particular target language simply do not exist. Clearly, test cases have to be concretized from the level of models to the level of actual code. This concretization is in general not a simple task: On the one hand, we want models as simplifications, but on the other hand, the system is at the implementation level usually much more complicated and complex than the model.

- In the second scenario, models are used both for code generation and test case generation. This might seem to defy the whole purpose of testing, as a system would then be tested against itself. There are, however, a number of arguments for test case generation even if the model is also used for code generation: For example, test cases can be used to test environmental assumptions or part of the environment itself, be it legacy code, operating systems, or hardware. If the outcomes of all test cases are checked manually, then the generated test cases can be useful, since checking may be less labor-intensive than conceiving test cases. Finally, for structural model transformations like refactorings which do not aim at changing the behavior, test suites can be generated automatically for regression tests. There are refactoring steps which require knowledge of a system invariant [24, 25]; in view of the difficulty of establishing such an invariant, a testing approach might be more practical. This case extends to the situation where the code generator or compiler cannot be trusted.

The implications of using model-based testing in an iterative development process are discussed by Pretschner et al. [27]; for model-based test case generation within AutoFocus, see Pretschner [26].

Note that testing is but one quality assurance technique: Reviews and inspections are obviously indispensable activities in any development; while formal verification based on deduction and model checking has so far not been very successful in software design, model-based development with explicit product models offers some promise for these fields, too.

7 Related work

While now ubiquitously used and at least a decade old, 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 as discussed by Withey [42]. Harel [12] is concerned with using statecharts for behavior specification. The same idea of separating the model from its views is also used in the model-view-controller paradigm [20].

The approach presented by Sgroi et el. [36] and Keutzer et al. [18] 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 is not contradictory to their line. The whole school of mathematical program specification as advocated by Dijkstra, Bauer, Hehner, Morgan, Hoare, and others, is clearly related to model-based development, but without the explicit concepts of process and product models. In terms of CASE tools, ArgoUML [30], for instance, does rely on the UML meta model. As far as we know, there is no explicit process model.

In the UML approach, a fine grained, explicit model of the product is defined, including different views of such a product (use case view, object/class view, interaction view, state-based view, etc.). These views are integrated into the UML meta model—corresponding to what we call the conceptual model. However, the establishment of semantical relations exceeding those structural relations of the conceptual model is missing. Since UML is focused on defining the conceptual product model, this model must be related to development activities. While the RUP [21] defines a process on top of the UML description techniques, it does not make use of the fine grained meta model underlying those description techniques. Activities of the process, their preconditions and results are not defined in terms of the UML meta model; rather, the RUP outlines the phases to be carried out and suggests

the description techniques that are useful for each phase.

The OMG's model driven architecture [39] aims at the definition 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 aspects of communication as well as structure as described in Section 2.

Software Cost Reduction [13] uses the notion of an abstract state machine and pre/post-condition style of specification at a mathematical level. SCR supports an explicit conceptual model including the notion of environment variables, system interface, states, and transitions. Based on this model, it supports consistency conditions that ensure well definedness (completeness, non-circularity). A process model is not explicitly defined.

Schätz et al. [33] provide a discussion of model-based development in the context of agile vs. rigorous processes.

While clearly code centered, aspect oriented programming [19]—or, more generally, separation of concerns—is similar in its vision w.r.t. finding ontological entities, i.e., abstractions, for aspects like concurrency, exception handling, etc. Differing from our approach, the idea is to incorporate these abstractions into general purpose languages like Java or C rather than to use dedicated domain specific languages. While there are static analysis tools for these languages, the power of general purpose languages renders these analyses most difficult—this is one reason why we emphasize the *restriction* of existing 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 [10], however, focusing on user participation and neglecting the importance of a domain-specific, fine-grained product model to define a process upon. The more structured approaches to process definitions, as found in the process pattern area [35], are also missing this detailed model as well as a tight coupling of product and process. 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.

Graphical editors in tools like Together concentrate on an explicit (UML) meta model but do not take into account a process model. Development platforms like Eclipse[3], the latest in IDE development, define process patterns (e.g., refactorings) but do not do this in an explicit manner. As far as we know, there is no explicit product model, either. While we do not know how simple the definition of new patterns in Eclipse would be, we were surprised

[3] http://www.eclipse.org

how simple it is to do so with an explicit product model in AutoFocus. Describing the transformation of replacing a set of channels by one tuple channel, for instance, turns out to be a simple exercise.

8 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 stem 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 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.

While these benefits do not necessarily improve the quality of the final *product*, they help to improve the *process* that lead to the product. In particular, our hope is to increase the efficiency of the development not only of single products but of related product families.

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: Some design steps might demand knowledge of environment properties which are difficult to formalize. Design steps in the later phases will require precise knowledge of the target platform, for instance to access device drivers or in order to estimate the worst case execution times which are needed as input for scheduling algorithms. Even if this knowledge is formalized and incorporated into the product model—as, for example, partly done in the Giotto language [14]—, more pragmatic problems, like the integration of legacy code, tailoring to customer-specific coding and certification standards or possibly just idiosyncrasies in compiler or operating system technologies can hamper our ideal of a seamless process.

These problems can—with varying degrees of difficulty—be solved. 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 [3, 2].

Goal of the AutoFocus II project is to realize a model-based CASE tool for the application domain of distributed automotive electronics. Crucial first steps in this project are to identify core concepts of this domain that lead to a suitable conceptual model for such a tool. Compared to this step, the development of the system model is quite well understood; the definition of activities for the process model will also be comparatively straightforward if it can be based on an adequate and stable conceptual model.

Of course, the acceptance of languages and tools depends not only on technical or even on methodological factors. We are closely working together with our industrial partners in automotive electronics and avionics, who provide us with valuable feedback on their application domains and development processes.

Acknowledgments. Our understanding of model based development was influenced by numerous discussions with P. Braun, W. Schwerin, T. Stauner, M. von der Beeck, and B. Rumpe. W. Prenninger, A. Wißpeintner, J. Jürjens, G. Wimmel, and J. Romberg provided valuable comments on a draft version of this paper.

References

[1] J. Barnes. *High Integrity Ada: The Spark Approach.* Addison Wesley, 1997.

[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] K. Bender and O. Kaiser. Simultaneous Engineering durch Maschinenemulation. *CIM Management*, 11(4):14–18, 1995.

[4] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison Wesley, 2001.

[5] M. Breitling and J. Philipps. Black Box Views of State Machines. Technical Report TUM-I9916, Institut für Informatik, Technische Universität München, 1999.

[6] M. Breitling and J. Philipps. Step by step to histories. In *AMAST 2000, LNCS 1816*, 2000.

[7] M. Broy. From states to histories. Lecture Notes of the Marktoberdorf Summer School on Engineering Theories of Software Construction, 2000.

[8] M. Broy and K. Stølen. *Specification and Development of Interactive Systems.* Springer, 2001.

[9] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proc. ICSE'02*, pages 439–448, 2000.

[10] J.-C. Derniame, B. A. Kaba, and D. Wastell, editors. *Software Process: Principles, Methodology and Technology.* Springer, 1999. LNCS 1500.

[11] M. Fowler. *Refactoring - Improving the Design of Existing Code.* Addison Wesley, 1999.

[12] D. Harel. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer*, 25(1), January 1992.

[13] C. Heitmeyer. SCR: A Practical Method for Requirements Specification. In *Proc., 17th AIAA/IEEE/SAE Digital Avionics System Conference (DASC)*, 1998.

[14] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of EMSOFT 2001, LNCS 2211*, 2001.

[15] G. Holzmann. From Code to Models. In *Proc. 2nd Intl. Conf. on Applications of Concurrency to System Design*, pages 3–10, 2001.

[16] F. Huber and B. Schätz. Integrated Development of Embedded Systems with AutoFocus. Technical Report TUMI-0701, Fakultät für Informatik, TU München, 2001.

[17] F. Huber, B. Schätz, A. Schmidt, and K. Spies. Autofocus—a tool for distributed systems specification. In *FTRTFT'96, LNCS 1135*, 1996.

[18] 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.

[19] 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.

[20] G. Krasner and S. Pope. A cookbook for using the model-view controler user interface paradigm in Smalltalk-80. *J. of Object Oriented Programming*, 1(3):26–49, August/September 1988.

[21] P. Kruchten. *The Rational Unified Process - An Introduction.* Addison Wesley, 2000.

[22] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[23] D. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2(4):1–9, March 1976.

[24] J. Philipps and B. Rumpe. Refinement of pipe and filter architectures. In *FM'99, LNCS 1708*, pages 96–115, 1999.

[25] J. Philipps and B. Rumpe. Roots of refactoring. In *Proc. 10th OOPSLA Workshop on Behavioral Semantics: Back to Basics*, pages 187–199, October 2001.

[26] A. Pretschner. Classical search strategies for test case generation with Constraint Logic Programming. In *Proc. Formal Approaches to Testing of Software*, pages 47–60, 2001.

[27] A. Pretschner, H. Lötzbeyer, and J. Philipps. Model Based Testing in Evolutionary Software Development. In *Proc. 11th IEEE Intl. Workshop on Rapid System Prototyping*, pages 155–160, 2001.

[28] S. Prowell, C. Trammell, R. Linger, and J. Poore. *Cleanroom Software Engineering*. Addison Wesley, 1999.

[29] F. Regensburger. HOLCF: Higher Order Logic of Computable Functions. In E. Schubert, P. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 293–307. Springer-Verlag, 1995.

[30] J. Robbins. *Cognitive Support Features for Software Development*. PhD thesis, University of California, Irvine, 1999.

[31] B. Schätz. The ODL Operation Definition Language and the AutoFocus/Quest Application Framework AQuA. Technical Report TUMI-1101, Fakultät für Informatik, TU München, 2001.

[32] B. Schätz and F. Huber. Integrating formal description techniques. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods, Proceedings of the World Congress on Formal Methods in the Development of Computing Systems, LNCS 1709*, pages 1206–1225. Springer Verlag, 1999.

[33] B. Schätz, A. Pretschner, F. Huber, and J. Philipps. Model-Based Development of Embedded Systems. In *Proc. Workshop Model-Driven Approaches to Software Development*, 2002. To appear.

[34] B. Schätz and K. Spies. *Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik*. 1995. Technischer Bericht, TU München, Institut für Informatik, SFB-Bericht Nr. 342/16/95 A.

[35] Scott W. Ambler. *Process Patterns –Building Large-Scale Systems Using Object Technology*. Cambridge University Press/SIGS Books, 1998.

[36] 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.

[37] O. Slotosch. Quest: Overview over the Project. In *Applied Formal Methods - FM-Trends 98*, pages 346–350. Springer LNCS 1641, 1998.

[38] O. Slotosch. Modelling and validation: AUTOFOCUS and Quest. *Formal Aspects of Computing*, 12(4):225–227, 2000.

[39] R. Soley. Model Driven Architecture. OMG white paper, 2000.

[40] G. Wimmel and A. Wißpeintner. Extended description techniques for security engineering. In *Proceedings of the 2001 International Conference on Information Security (IFIP/SEC)*, 2001.

[41] N. Wirth. Program Development by Stepwise Refinement. *CACM*, 14(4):221–271, April 1971.

[42] J. Withey. Implementing model based software engineering in your organization: An approach to domain engineering. Technical Report CMU/SEI-94-TR-0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1994.