From Formal Semantics to Executable Models: A Pragmatic Approach to Model-Driven Development

Helmuth Partsch, Marcel Dausend, Dominik Gessenharter, Jens Kohlmeyer, and Alexander Raschke
(Institute of Software Engineering and Compiler Construction, Ulm University, Germany)

Abstract With respect to contents, Software Engineering deals with the mutual combination of formalisms, methods, and tools that are integrated by a common theoretical basis, as well as applications that use all these aspects. The model-driven development (MDD) of software is a typical instance of this view, as it exhibits all the facets mentioned above. In this paper, we identify some major unsolved problems of MDD concerning the contemplated areas and postulate a pragmatically motivated couple of hypotheses defining our constraints for solving these problems. Our particular view of MDD uses the Unified Modeling Language (UML) which we substantiate by defining a formal semantics for wide parts of it. On this basis, we suggest a process which incorporates model elaboration combined with aspects of quality assurance. Prototypical tools to support our approach are presented and their applicability is shown by means of two examples.

Key words: model-driven development (MDD); formal semantics; Abstract State Machines (ASM); Unified Modeling Language (UML); high quality models; code generation


1 Introduction

1.1 Central issues in software engineering

In a traditional view, Software Engineering comprises the fields technical aspects of software development, quality assurance, and project management. From a different perspective emphasising its contents, Software Engineering deals with various, strongly coupled aspects which can be depicted as in Fig.1.

Formalisms are needed to describe the various artefacts produced by different activities during software development. Among others, such formalisms comprise graphical models, logical calculi as well as all kinds of specialised languages.

Methods are necessary to guide the proper usage of formalisms and to put them into context. On the other hand, methods as such are usually abstract and need descriptive means to be made concrete.
Tools are usually intended to support methodological aspects and the systematic usage of the various formalisms. They are indispensable in Software Engineering simply due to the huge quantities of information one has to deal with.

Theory provides the semantics of formalisms, or calculi for reasoning about important properties such as correctness, consistency, or completeness. Also, theory establishes the foundation for methods and methodological aspects such as hierarchical decomposition, refinement, compositionality, or model transformation. And last but not least, theory is the basis for architectural aspects of tools, their complexity, or the correctness of their functionality. Thus, theory can be viewed as the “glue” that keeps the other parts together.

In the context of applications, particular formalisms, methods, and tools, that should share a common semantic basis, are used in an integrated fashion, but usually without direct reference to theory.

Within Software Engineering, all these parts are equally important and none of them alone really makes sense without the other ones.

1.2 Model-Driven development

Since long, it is commonly agreed that any approach to software development should proceed continuously from requirements through specification to implementation, using formalisms, methods and tool sets on a common theoretical basis in an integrated fashion (as depicted in Fig. 1). This is also the dedicated aim of Model-Driven Development (MDD), which — usually combined with classical development processes — tries to raise the level of abstraction by using (mainly graphical) models instead of natural language specifications and conventional programming languages, and to bridge the gap between specification and implementation by appropriate model transformations or by direct generation of code out of the models.

Although, a step into the right direction, MDD still has to face quite a number of obstacles and to solve some open problems. Out of these — without any claim for completeness — we identified the following major difficulties.

P1: Adequate and Integrated Formalism. There is a huge amount of different formalisms (cf. Ref. [33]) that are useful for problem specification. In particular, the Unified Modeling Language (UML) can be profitably used to model requirements and specifications by means of suitable diagrams. In addition, further
formalisms may appear during development, as well as in the context of quality assurance and project management. For all these formalisms expressiveness, integration, and a formal semantic foundation are the central issues.

**P2: Precise Semantics.** Many of the formalisms used in modelling only have a semantics specified as prose (if at all) which leads to inconsistencies between different implementations of the formalism in different tools and to ambiguities in the interpretation of the models (cf. Ref. [3]). Accordingly, one of the major shortcomings of the UML is the lack of a precise formal semantics with the consequence of weak tool support (cf. P6) and restricted possibilities to do analysis (cf. P4) with models (cf. Ref. [5]). This problem has been addressed in recent years by research on formalising the semantics of several kinds of diagrams\textsuperscript{17,25,29,30,44} or approaches aiming at an integrated formal semantics\textsuperscript{2,9}.

**P3: Systematic Development of High Quality Models.** Compared to programming, modelling addresses a higher level of abstraction, but is by no means easier. E.g., how to develop models that appropriately represent natural language requirements, is still a topic of ongoing research. Also, it is known that two totally different models may reflect the same given situation, and it is neither clear nor really well understood, how to relate or compare these models, since appropriate quality criteria do not yet exist. And, of course, modelling too, requires appropriate means to prevent or remove errors (e.g. by simulation, tests or static checks).

**P4: Model Consistency.** Today it is widely agreed upon that a complete model of a system comprises different views (e.g. at least structure and behaviour), which are usually expressed by means of different formalisms (e.g. class diagrams for static structure and state diagrams for behaviour). This fact obviously causes a problem of consistency, the solution to which not only requires precise (integrated) semantics of the formalisms, but also appropriate consistency rules to be checked (within as well as between the different formalisms).

**P5: Correctness of Implementations.** A related problem concerns the consistency (or correctness) between the models used during development and the resulting code. Here too, it has to be guaranteed that model transformations, refinement to code, or code generation are correct w.r.t. the underlying semantics.

**P6: Tool Support.** There is a huge variety of tools to support MDD, in particular tools to support the UML (cf. Ref. [15]). However, all these tools have substantial drawbacks (cf. Section 4), in particular poor coverage of the existing possibilities for modelling and a missing formal semantic foundation.

### 1.3 Working hypotheses and our view of MDD

Obviously, any attempt to solve (some of) the problems in Section 1.2 needs a reasonable basis to start from. Therefore, our approach to MDD is based on several hypotheses most of which are motivated by pragmatic considerations — in particular, to build upon existing good ideas and approaches rather than to start from scratch and invent everything anew. In the following we will introduce these hypotheses and explain the reasons for our decisions.

**H1: Formal Semantics.** We agree with many other researchers (e.g. Refs. [5, 12, 36]) that for MDD to be successful, a formalism with a precisely defined semantics is indispensable. A MDD approach with imprecisely or weakly defined modelling
concepts has the disadvantages already sketched above and may be considered even
worse than the traditional use of programming languages. In our view (cf. Section
2.3 for details) Abstract State Machines (ASMs) are a suitable mechanism to provide
a solid (integrated) semantic basis for MDD, in particular, in order to solve P2, P4
and P5.

H2: Established Formalisms. For pragmatic reasons, we do not want to
invent a new modelling formalism, but rather improve one that is already widely
used in practice, by formalising its semantics. In principle, there are two ways of
arriving at suitable modelling formalisms with precise semantics. The first possibility
is “semantics first”, i.e. to define semantics according to an available theoretical
basis and adapt/restrict the available modelling concepts to this semantics. In this
way a nice, clean, and comprehensive semantics is given to the modelling concepts,
but maybe at the price of reduced expressiveness and weak acceptance. The second
possibility is to start from a widely used, highly expressive formalism, and give formal
semantics to the existing informally defined modelling concepts. In this way some of
the semantic definitions may become somewhat more complicated, but the original
expressiveness is maintained and acceptance is not an issue at all. We decided to follow
the second possibility and based our approach on the UML as a (partial) solution to
P1.

H3: High Quality Models. We are convinced that MDD can only be suc-
cessful, if quality aspects are considered as early as possible in software development.
Only if it is guaranteed that the models the development starts with adequately re-
fect the customers’ original requirements and that these models are correct w.r.t.
characteristic properties to be achieved, a subsequent development into executable
code is economically feasible and profitable. Therefore, as part of a solution to P3,
in addition to classical approaches to quality assurance such as static checks (in the
form of reviews or inspections) or various forms of tests, we strongly believe in early
model simulation as a means to validate the models’ adequacy, as well as in model
checking to assure correctness.

H4: Tool Support. We agree with Ref. [5], that an integrated tool chain based
on a common semantic basis has to be the ultimate goal where each of the tools in
the chain should be “robust and powerful enough for industrial use” (cf. Ref. [3]).
Also, again in accordance with Ref. [3], we believe that a “high degree of automa-
tion” is necessary which addresses “the capturing and elaboration of models, the
analysis of models with respect to their consistency and important properties as well
as techniques for generating further development artefacts from models”. Therefore,
“maximal” code generation from models is for us as important as an open tool archi-
tecture (on a common semantics basis) that is easily extensible and supports smooth
integration of further tools to be developed — and, thus, to provide a solution to P6.

H5: Adaptability to Specific Domains. With respect to applications, we are
convinced that the adaptation of UML to specific application domains by means of
sterotypes and profiles will lead to practical results — provided a precise semantics
of these mechanisms can be given — in the sense of P1 and P2.

Our vision is an integrated, tool-supported, MDD approach (cf. Fig. 2) to high
quality software that solves at least the problems mentioned above. This approach
should comprise effective elicitation and documentation of requirements (if possible,
by appropriate models), a methodically backed transition from requirements to high quality design models as well as an implementation by automatic code generation, possibly combined with certain model transformations — all on the basis of an integrated formal semantics and accompanied by appropriate means for quality assurance.

Figure 2. Vision of our integrated, tool-supported, MDD approach

Not all aspects w.r.t. this vision have been successfully completed by now — in particular the steps necessary to arrive at an initial model. However, substantial progress was made for some other important aspects. These are an integrated semantic foundation (cf. H1, P2, and P4) on the basis of ASMs, UML (with precise formal semantics) as an established formalism (cf. H2 and P1) and a basis for domain-specific adaptations (cf. H5), different kinds of support for quality assurance (cf. H3 and P3) and comprehensive tool support (cf. H4 and P6), in particular extensive, automatic code generation (according to the formal semantics, cf. P5).

1.4 Structure and overview of the remainder of this paper

This paper treats our pragmatic approach to MDD in detail and presents our main contributions along the lines sketched in Section 1.2.

Section 2 gives a short introduction to some UML diagrams and our formalisations of their semantics by means of ASMs. Some methodological aspects are dealt with in Section 3, in particular the usefulness of adapting model checking mechanisms to UML activity diagrams. The content of Section 4 are various tool aspects, in particular, extensive code generation (inclusive of “difficult” UML concepts). Section 5 then is devoted to applications. In addition to reporting on applications we did so far, the aspect of domain-specific languages (DSLs) based on UML will be discussed in the context of developing user interfaces (UIs), in particular multimodal ones. The conclusion in Section 6 summarises current activities and gives an outlook to future work.

2 Formalisms

In this section we briefly introduce UML[31], with a focus on those diagrams that are used in the applications we deal with. Furthermore, we introduce mechanisms of extending UML towards DSLs as well as our formalisation of the UML semantics by means of ASMs.

2.1 UML modelling at a glance

UML is a meanwhile established standard formalism (cf. H2) for graphical modelling that offers different kinds of diagrams to describe structure and behaviour of a system.
The static structure of a system can be specified with structure diagrams, in particular class diagrams that show the system’s classes, their attributes, and the relationships (e.g. generalisations or associations) between classes.

Behavioural aspects of software systems are represented by several UML language packages defining State machines, Activities and Interactions. Each type of diagram is useful by itself, offering different facilities to exhibit different properties of a system: State machines emphasise the changes of a system’s state due to the occurrence of events by specifying states of a system and transitions between them. Activities focus on control and data flow and coordinate lower-level behaviours by specifying their dependencies and the allowed execution sequences. Interactions show the communication between objects by describing the sequence of messages exchanged.

In the following we describe class diagrams and activity diagrams in a little more detail by means of an example which is part of a sample project designed to demonstrate the feasibility of our approach on the basis of a self-explanatory system. This system models a TravelPlaner that holds the timetables of public service vehicles of an imaginary city. Due to space limitations, we only touch the most important requirements of the system. For a detailed description we refer to Ref. [21]. Figure 3 shows an excerpt of the static structure of the TravelPlaner. The entire class diagram is contained in Ref. [20].

![Figure 3. Static structure of the sample project TravelPlaner](image)

A location where Vehicles (e.g. busses) stop to allow for boarding and deboarding is called a Stop. Stops are accessed by TravelPlaner and connected by Segments. The fact that a vehicle rides along a segment is represented by the AssociationClass Ride. A Schedule holds the time table for the associated Stop.

The fundamental elements of activity diagrams are Actions that are connected by edges to indicate control or data flow. Actions specify transformations on the state of the system that are not further decomposed within the given diagram. They are either implementation-dependent (CallOperationAction) or more specific, e.g., used to send and receive signals (AcceptEventAction) or to invoke behaviour specified in other diagrams (CallBehaviorAction).

Edges connecting Actions may pass through ControlNodes that coordinate the flows in an activity diagram. A DecisionNode chooses between different outgoing edges and the corresponding MergeNode unites alternate, independent flows. A ForkNode splits a flow into concurrent flows along all outgoing edges and the corresponding JoinNode synchronises all incoming flows. Furthermore, flows may originate in InitialNodes and terminate in FinalNodes. InputPins and OutputPins (as special forms
of **ObjectNodes**) attached to actions indicate object flows. An **InterruptibleActivityRegion** contains a subset of nodes and edges which characterises a part of an activity diagram that can be separately terminated.

The UML specification proposes a petri net-like semantics for activity diagrams: Tokens determine the current state of execution by flowing along edges and causing actions to execute.

Figure 4 shows a simplified activity diagram for an excerpt of the **TravelPlaner-Behaviour**. The user concurrently selects a departure and a destination stop as well as the time of the earliest departure and latest arrival. When both actions are completed, the system searches for connections meeting the specified constraints in the action **FindConnections** and returns connections represented as a list of list of **Rides** (**ShowConnections**). If this list is empty, the user may modify his input. All actions are contained in an **InterruptibleActivityRegion** and therefore are aborted if a signal **Abort** is received by the **AcceptEventAction**.

2.2 Domain specific languages on the basis of UML

Reference [45] argues to use UML as a general purpose language for domain specific applications. In order to use a formalism such as UML as a DSL, extensibility is a crucial issue for its acceptance. In principle there are two possibilities for extensions: **Extending the metamodel** or **using profiles and stereotypes**.

Metamodel extension, e.g. by using inheritance and adding new classes, in fact means to create a new language. Opposite to this, “A profile defines limited extensions to a reference metamodel with the purpose of adapting the metamodel to a specific platform or domain.”[31, p. 679]. Therefore, our interest w.r.t. DSLs concentrates on profile extension.

A profile can be composed of **Stereotypes** (to define an extension referring to an element of the metamodel), **Extensions** (to enable different kinds of relations between elements, e.g. stereotypes and metamodel elements), **Constraints** (to define formal
constraints on elements of a profile), or Textual annotations (to describe the semantics of a profile and to contain further annotations as prose text). A brief introduction to UML profiles and an explanation of how they can be created step by step is given by Ref. [16]. Each profile provides an independent extension to UML that can dynamically be exchanged with other profiles or even reused for a different application. Syntax and semantics of a profile must not conflict with UML semantics.

Reference [32] gives a systematic review of approaches which use UML profiles. It includes 63 approaches from the most relevant conferences in an 11-year period (1999–2009). The abstract definition of profiles and quality of presentation of 39 of these approaches are analysed considering 26 variables of seven categories. Reference [32] states that “the definition of the profile formal semantics is very rare”, and that “behavioural modelling needs to be supported by their formal semantics to be useful”.

An application of using UML profiles for Human-Computer Interaction (HCI) is dealt with in Section 5.2. There, a pragmatic way of integrating profile semantics with the UML semantics has been chosen.

2.3 Abstract state machines

For the following reasons we use ASMs to formally define the UML semantics (cf. H1): ASMs have successfully been used to formalise the semantics of several programming and specification languages. ASM rules are precise, comprehensible, and executable. Thus, they allow to reveal problematic issues concerning the UML specification.

ASMs were introduced by Gurevich[24]. An ASM is a transition system which can be read as “pseudo-code over abstract data”[6]. It consists of rules of the form if condition then updates, where updates is a set of assignments, which are simultaneously executed when the guard condition is true. An update assignment $f(s_1, \ldots, s_n) := t$ modifies the value of $f$ at $(s_1, \ldots, s_n)$ to $t$.

Further constructs include abstractions using let...in, multiway conditionals, macro definitions (for grouping rules), and macro calls with call-by-name semantics. The rule forall $x$ with $\varphi$ do $R$ executes $R$ in parallel for each $x$ satisfying $\varphi$. Updates are performed in parallel unless sequentialised by seq.

In our work we use asynchronous multi-agent ASMs, allowing the concurrent execution of several ASM agents. Further details of ASMs, including an operational semantics, are provided by Ref. [6] and Ref. [37].

2.4 Formalising UML semantics

We briefly describe our general approach to formalise the semantics of UML using ASMs. The ASM rules need to access the concrete diagrams of the model whose semantics they define. To this end we translate the UML metamodel to static ASM domains (for classes) and functions (for attributes and associations). They are all initialised to yield the particular values corresponding to the concrete model. Special, so called monitored ASM functions are used for information determined by the environment. Individual executions of behaviour are represented by different agents. To model their interaction and signal handling, we use shared ASM domains and functions. The UML semantics is then specified by ASM rules acting on these functions. Full details of this approach are provided in Ref. [38] and Ref. [37].
In the following we illustrate our formalisation of the UML semantics by outlining the ASM semantics for the computation of the token flow in activity diagrams.

According to the UML specification\cite{31}, nodes offer tokens on outgoing edges. A token can only traverse an edge if its guard evaluates to true and the destination node accepts it. Since control nodes act solely as “traffic switches”, tokens can only rest on the outgoing edges of Actions and InitialNodes, as well as on ObjectNodes. The traverse-to-completion principle\cite{4} requires that, if a token is accepted by all intermediate edges and ControlNodes as well as the destination, the whole path from the source node to the destination is traversed at once.

The exact mechanism of the propagation of token offers and their selection at destination nodes, however, is neither formally defined, nor adequately discussed in the specification\cite{39}. Our proposal for transition computation and execution consists of the main ASM rule in Fig. 5.

\begin{verbatim}
INITIATEFLOWCOMPUTATION
  seq PROPAGATEFLOWINFORMATION
  seq SELECTTOKENOFFERS
  seq INTERRUPTIBLEREGIONCHOOSEFLows
  seq ACTIVATEACCEPTEVENTACTIONS
  seq EXECUTETRANSITION
\end{verbatim}

Figure 5. Main ASM rule for transition computation and selection

The INITIATEFLOWCOMPUTATION and PROPAGATEFLOWINFORMATION macros spread token offers from the source nodes, where the actual control and data tokens rest, through the activity. After all possible offers have been computed, subsets are selected at destination nodes, preparing the traversal of the associated tokens. Note that selecting token offers can invalidate other, conflicting token offers. Since aborting InterruptibleActivityRegions can prevent tokens from traversal, the rule INTERRUPTIBLEREGIONCHOOSEFLows removes selections containing those tokens. AcceptEventActions without incoming edges contained in InterruptibleActivityRegions are initialised by ACTIVATEACCEPTEVENTACTIONS. The actual execution of the token traversal is performed by EXECUTETRANSITION. The main rule is executed repeatedly as long as control or data tokens are available. A comprehensive set of rules dealing with the transition computation and selection is presented in Ref.\cite{38}.

The propagation of token offers is performed in two steps. First, new token offers are created for tokens resting at outgoing edges of actions or initial nodes. Second, the token offers are propagated through the activity towards the consuming destination nodes, namely actions, object nodes, and final nodes. Within this propagation, several rules for the different kinds of control nodes are involved. As an example, we present the ASM rule PROPAGATEFLOWFORMERGENODE in detail.

The propagation for a MergeNode (see Fig. 4) is described by the specification Ref.\cite[p.399]{31} as follows: “All tokens offered on incoming edges are offered to the outgoing edge. There is no synchronisation of flows or joining of tokens.”

In Fig. 6, at the merge node \(n\) all incoming edges \(e\) have to be checked for offered tokens \(t\). If the guard of the outgoing edge holds, the new token offer \(t(out)\) is calculated as defined in the let-expression.
PropagateFlowForMergeNode ≡

... for all e with e ∈ n.incoming do
for all t with t ∈ offers(e) do
  if IsGuardTrue(outgoing(n, 1).Self.context, t.offeredToken) then
    let t(out) = new(TokenOffer) in
    t(out).offeredToken := t.offeredToken
    t(out).paths := \{ p ⊕ outgoing(n, 1) | p ∈ t.paths\}
    t(out).exclude := t.exclude
    t(out).include := t.include ∪ \{t\}
    ... add t(out) to offers(outgoing(n, 1))

Figure 6. Excerpt of ASM macro for flow computation at a MergeNode.

The function paths represents the path beginning at the source node of the token to the current position of the offer. According to the specification Ref. [31, p.406], “a token in an object node can traverse only one of the outgoing edges”. Our formalisation must therefore ensure that these offers exclude each other, as it is the case for DecisionNodes with overlapping guards. To this end, the component exclude of TokenOffer collects all conflicting offers. It is initialised to contain all edges except the current, since all outgoing edges of control nodes compete with each other. The function include contains those offers that have contributed to the current offer.

Both the exclude and include functions are used for selecting token offers at destination nodes. All paths from the function paths are extended by the current outgoing edge, and the base token offer is added to the include set. Finally, the new offers are added to the set of offers of the outgoing edge.

Analogously, we defined a formal semantics of the UML language units Common Behaviors, Actions, Activities, State Machines, Interactions, and Use Cases using ASMs[27] as well as the semantics of their combined use[28]. The behaviour of software models can henceforth be specified by composing activity, state and sequence diagrams, choosing the most adequate formalism at each level of abstraction. The resulting rules also allow for the integration of further formalisms or even code to implement low-level behaviour. In Ref. [28] we also give a formal semantics of communication between these diagrams and thus achieve an integrated semantics of UML behaviour.

3 Methods

Our vision of an integrated MDD approach is sketched in Section 1.3. Here, we give more details of integrating different formalisms on the basis of a common formal semantics. This formal basis allows for a convenient application of different (well-known) methods for model elaboration and quality assurance.

Besides the general description, we illustrate a concrete already implemented instance of this overall approach (cf. Section 3.1). In particular, the interaction of UML class diagrams and activity diagrams in combination with hand-written code is shown.

Currently, we provide simulation and model checking for early development phases to ensure high quality of models (cf. Section 3.2). An inevitable prerequi-
site for the application of these methods on models is that all models rely on an integrated formal semantics (as mentioned in Section 2.4).

3.1 Model-Driven development methods

Figure 7 gives a more detailed view on typical steps which are part of our approach and are to be performed before a final product can be released. First, the informal (usually textual) requirements have to be transformed into a requirements model, which may be on an abstract level, but is already integrated into a formal basis. This allows for the use of known methods like hierarchical decomposition, model transformation and model refinement in order to gain a more concrete design model by iteratively taking into account more detailed requirements.

Since graphical information on its own is usually not sufficient to describe the functionality of a system properly, our approach includes the integration of additional artefacts (e.g. UI-description, formal specification, hand-written code, etc.), provided that a formal semantics of these artefacts is given. These additional artefacts are not only “add-ons” to the existing diagrams, but can be tightly integrated into the diagrams. For example, in our implementation so far, it is possible to define the behaviour of UML actions in Java code.

The formal semantics allows for simulation of these models (including additional artefacts) in early development phases. Thus, it is possible to validate the model against the requirements before a final, runnable product is available.

Iterating the refinement and quality assurance steps several times leads to a complete model, from which the final software product can be obtained by model transformation and/or code generation. For this last step, we deliberately leave it open, whether code generation has to be preceded by some model transformation, since this depends on the diagram types and additional artefacts used.

In the following, these ideas are substantiated by the TravelPlaner-example introduced in Section 2.1. Development is started by firstly modelling the system’s “Static Structure” (see Fig. 8) by means of a class diagram. The static structure is then supplemented by modelling behavioural aspects (cf. Section 2.1). In our example, application control and data flow is modelled using activity diagrams (see “Dynamics” in Fig. 8). Each class that should have its own behaviour has an associated activity, describing its functionality, e.g. the behaviour of the class TravelPlaner is specified in the activity TravelPlanerBehaviour.
UML actions — more precisely, UML CallOperationActions — are used to invoke custom Java code written by the developer (as additional artefact). In Fig. 8 this is indicated by a CallOperationAction named FindConnections and its associated method declaration List<List<Ride>> FindConnections (Stop from, Time depart, Stop to, Time arrive) in the lower right. For further (also technical) details and more examples we refer to Refs. [43, 38, 42, 21].

By allowing to use regular code within UML activity diagrams for the implementation of actions, we tightly integrate modellng and coding tasks while deliberately leaving the respective balancing between code and model to the developer.

Figure 8. Illustration of an instance of the proposed MDD methodology

3.2 Quality assurance

One main issue of MDD is to improve the benefit of modelling w.r.t. quality. One aspect is a sophisticated code generator (for static structure as well as for dynamic aspects), which leads to a reduction of the amount of hand-written code with all its advantages like less individual bugs, faster development, and better maintenance. Other aspects, w.r.t. quality assurance, are simulation and model checking of the created models, which can be applied besides traditional methods like reviews and tests (based on simulation). A further aspect that has to be dealt with is the internal and external consistency between all diagrams and artefacts used. The overall formal semantics facilitates automating these checks. The combination of these different quality assurance methods supports the creation of high quality models in early development phases, which is a crucial point for MDD.

Not all of these quality aspects have been realised in our concrete implementation up to now. In the next two sections, we focus on simulation, which supports checking
the adequacy of behavioural diagrams (validation) and model checking, which allows for proving desired properties (verification). In both cases, an underlying formal semantics is indispensable.

3.2.1 Simulation

According to hypothesis H3, the adequacy and correctness of models are important factors for successful MDD. Since models can be at least as complex as source code, the quality of a model cannot be assured by static analysis (e.g., review) alone. Especially conformance of the model with the desired requirements can be validated much better by simulation. As it is known that wrong or missing requirements are among the most common sources of errors in modern software development, early simulation helps to improve this situation, too.

Since the control flow can be executed independently from the concrete implementation of the actions, it is already possible to test the behaviour (resp. control flow) of an application without implementing any action.

3.2.2 Model checking

Model checking[8] allows for automatic verification of mandatory properties of a modelled system. It is very successfully used to check e.g. safety and liveness properties, especially for embedded systems and circuit design.

But model checking is not restricted to (models of) critical systems only. Also for information systems automatically checking important aspects may be required. For a more complex version of the TravelPlaner, checking whether the activities include an execution path which makes it possible to get a ticket without paying for it might be indispensable.

Since the revision of UML that resulted in UML 2, activity diagrams are no longer a special kind of state diagrams, but rely on a newly defined “token flow semantics” (cf. Section 2.4) and offer completely new concepts such as signal handling and interruptible regions. Due to this, it is no longer possible to directly apply model checking.

To solve this problem, we defined a transformation of activity diagrams into a state transition system (STS) as an intermediate representation that preserves the original semantics[34]. The main advantage of this approach is the reuse of existing model checking techniques and tools. Furthermore, this STS is essential to handle the combination and interaction of different diagram types as described in Section 2.4.

The above mentioned transformation is defined abstractly and leaves open which specific model checking tool is used. Based on the STS, the integration of well-known model checkers like NuSMV[7], or SPIN[26], is straightforward.

There are only a few approaches for model checking UML 2 activity diagrams, and all of them concentrate only on simple aspects of the control flow. Our work, however, covers also more difficult concepts of activity diagrams like signal handling, object flows and interruptible regions.

The most challenging part of our work is the correct consideration of the traverse-to-completion semantics of activity diagrams (cf. Section 2.4). Other approaches identified this as a central problem, e.g.: “We could thus discard the simple mapping presented [...] and devise a more complicated compilation approach which evaluates
the different possible control flows through a control structure of an activity diagram \[.\]^{[41]}. This idea is picked up and realised in Ref. [35], where the transformation of activity diagrams into an STS is formalised with ASMs, our common formal basis, according to the formal semantics (cf. Section 2.4). Some prototypical case studies in Ref. [35] show that even in small examples, model checking can help to find subtle errors in published activity diagrams.

4 Tools

It is impossible to completely survey all existing UML tools which are available from academia or from commercial vendors — both their number and their constantly progressing advancements are an unmanageable area. But a study of UML tools\[^{[15]}\] gives a compact overview of the current situation to which we can refer to.

Academic tools often are designed to be a proof of concept and therefore are not robust enough for industrial use. Tools of commercial vendors, however, often lack the semantic foundation of the supported modelling languages. Rather than offering extensive capabilities for further model processing like code generation, these tools usually only provide interfaces where extensions made by the user may be plugged in.

E. g. Sparx Enterprise Architect\[^{1}\] is a tool providing very low-level templates for code generation which at most can serve as examples for possible user extensions, IBM RationalSoftwareArchitect\[^{2}\] comprises an extensible code generator which initially generates only rudimentary code. However, both tools are classified as being able to generate code from class diagrams\[^{[15]}\].

Regarding activity diagrams, the situation is even worse. By only delivering very basic implementations for features which can be adapted and extended by the user — who most likely has not the necessary know-how to extract as much information of a model as possible — tool vendors avoid dealing with the shortcomings of UML which is felt to be skating on thin ice.

4.1 ActiveCharts

As we address a formal specification of the token flow semantics of activities or static aspects like associations in our work, we are interested in a tool able to support this features. For lack of an appropriate tool\[^{[15]}\], we implemented an eclipse\[^{3}\]-based integrated development tool based on our ActiveChartsIDE\[^{[40]}\].

Using eclipse is a rather pragmatic decision: this IDE provides facilities for project and version management, team support, graphical modelling editors and much more. In particular, it includes a well structured plug-in concept and is widely used.

The basic elements that are currently implemented in our new prototype are an activity interpreter, a model transformation and a code generator. Like its predecessor ActiveChartsIDE, the interpreter plug-in is based on the formal semantics of the UML token flow\[^{[38]}\] but underwent a reconstruction with regard to the interpreting process. Instead of step computations at runtime, lookup tables are created while starting an activity execution leading to a noticeably faster step execution\[^{[22]}\]. This enhanced interpreter is used for step by step execution of activities when simulating

\[^{1}\] http://www.sparxsystems.com/
\[^{2}\] http://www.ibm.com/software/awdtools/architect/swarchitect/
\[^{3}\] http://eclipse.org/
behaviours during early development phases.

For reasons of efficiency, a code generator is currently under development and near completion. It covers the static structure as well as behaviours. Like supporting good model quality by simulation and model checking, generating high quality code is the main objective of our code generator.

The generated code implements all classes with attributes and associations contained in a class diagram, including code to handle modifications (i.e. addition and removal of objects) of those relationships. In particular, the semantics of “difficult concepts” like correct handling of multiplicity constraints, navigability and visibility of association ends\cite{18}, composite aggregations, higher order associations and association classes is defined and proper code is generated\cite{19}. With regard to behaviour, code is generated for control and object flows of activities covering highly sophisticated modelling concepts like concurrency and interruptible activity regions — a feature not supported by any of the tools examined in Ref. [15].

In order to keep the code generation process straightforward, a model transformation running prior to code generation is implemented. Complex modelling concepts like associations, in particular higher order associations or composite aggregations, are mapped to implementation patterns resulting in a design model from which code is generated\cite{19}. This transformation is not defined by using a model transformation language but is directly implemented as an eclipse plug-in in Java code. Yet, it has not been examined which transformation language — if any — is powerful enough for describing the desired transformation. However, being eclipse based, a better support for or more extensive use of transformations — possibly with other implementation strategies — can easily be added to our tool.

All parts of the prototype are implemented according to the formal semantics and have been systematically tested. However, the correctness of the implementations has not yet been proved. As ASMs are a formalism with an execution semantics, the direct execution of the ASM rules is a thinkable strategy. Unfortunately, currently none of the available tools like CoreASM\cite{4} or ASML\cite{5} supports all features which we used.

Our prototype is extensible by the eclipse plug-in mechanism. Extension points are defined for a model validation implementation, a model transformation and code generator. Being based on the eclipse UML2\cite{6} implementation, every editor can be used for designing a diagram if it is capable to produce a XMI (cf. Ref. [31]) representation of the model. For visualising the token flow during the step-by-step execution of an activity, an editor with special capabilities is needed.

4.2 Extensions

Based on the formal semantics, we extended our approach for state diagram support. Furthermore we introduced HCI modelling and simulation support to this extension (cf. Section 5.2). We provide nearly all the features offered for class and activity diagrams for state diagrams. However, we do not offer code generation yet, because our primary interest is on investigation of extensibility and applicability of our approach. Moreover, code generation for state diagrams is already well known.

\begin{itemize}
  \item \cite{4} http://www.coreasm.org/
  \item \cite{5} http://asml.codeplex.com/
  \item \cite{6} http://www.eclipse.org/uml2/
\end{itemize}
Our open tool architecture enabled the integration and coupling of the commercial XML- and script-based UI development tool FlexBuilder\(^7\) and ActiveCharts inside the same eclipse environment. This combination provides a solid basis for model checking, simulation and code generation.

5 Applications

Our approach has been evaluated in two dimensions: It has been used in the context of a couple of sample applications and it has been extended and applied to HCI modelling.

5.1 A sample application

Using our approach and tool in the context of sample applications aims to evaluate both, the applicability and simplification of the software development process.

In addition to examples of embedded systems like a simple alarm device controller, an elevator controller or a molding press which all share the symptom of complex behaviours but very simple static structures, we developed the TravelPlaner (cf. Section 2 and Ref. [21], for the complete example), an information system with a very complex static structure by using our prototype.

For the static structure consisting of 17 classes, 14 associations and 3 association classes, 1700 lines of code are generated of which 1300 lines are non trivial code. This code covers the entire implementation of associations like providing methods for connecting or disconnecting objects. In addition, approx. 700 lines of code are generated to implement the behaviours.

In total, 75–80\% of the sample application’s code is generated, whereas the basic implementations which are available for commercial tools generate only about 150 to 500 lines of code (5–20\%). For clarity, it should be noted that the percentage of generated code when using our prototype may vary as the developer freely chooses the balance of modelling and coding (cf. Section 3.1).

Currently, the implementation of a code generator for activities is already useable but is not yet in its final state. Like the interpreter, it will support control and data flows, signal handling, CallBehaviorActions and CallOperationActions as well as InterruptibleActivityRegions — a feature the runtime semantics of which is not supported by any of the tools examined in Ref. [15]. First benchmark tests giving an idea of the higher performance of generated code are published in Ref. [22].

5.2 Multimodal user interfaces as domain specific application

In cooperation with Daimler Research our ideas on an extension of the UML (mainly by profiles) towards a DSL have been applied to HCI in the field of automotive in-car infotainment environments. This HCI is multimodal i.e., it includes graphic-haptic operation as well as speech operation and interaction between both. We found the application domain very challenging to prove our MDD approach in general and analyse its extensibility in particular.

The decision for applying HCI modelling implied that not only formalisms, but also methods and tools (cf. Fig. 1) had to be taken into account. Therefore, during

\(^7\) [http://labs.adobe.com/technologies/flex/](http://labs.adobe.com/technologies/flex/)
development of our UI profile (see below), we considered the extension of our formalism (UML and its ASM semantics), the validation of the semantics of the HCI models by simulation and case studies, as well as extending the available tools (cf. Section 4) to support state diagrams, profiles, and application of our MDD method\[11\].

Modelling HCI systems has been a topic for many years. An analysis and comparison of the most common and generally accepted approaches can be found in Ref. [11]. Among these, UML, promoted in the field of HCI modelling by a small community, can be found, too. Although it is not a classical approach to the modelling of interactive systems, it meets most requirements better than other approaches, particularly in view of acceptance and dissemination\[13\].

We decided to concentrate on UML state diagrams as a basis for our multimodal HCI profile. An important argument for this decision is the characteristic behaviour of UIs which is perfectly reflected by state diagrams: A dialogue consists of several steps between dialogue states. Each dialogue state in fact describes an invariant for the HCI which holds, until a user or system interaction occurs. The appropriateness of creating speech dialogues with state diagrams is already demonstrated in Ref. [23].

![Figure 9. Our metamodel profile definition](image)

In our UI profile to implement HCI modelling, we extend UML state diagrams without changing their semantics. This profile (cf. Fig. 9) contains three stereotypes \(<\text{GUI}>\), \(<\text{Grammar}>\) and \(<\text{Prompt}>\) to implement graphic-haptic, as well as speech dialogues. Each of these stereotypes extends the class \textit{State} from the UML metamodel by introducing attributes which enable referencing modality-specific content. By means of appropriate attributes the profile describes static aspects, like graphical elements and appearance (in \(<\text{GUI}>\) ), state related speech grammars (in \(<\text{Grammar}>\) ) and contents of system prompts (in \(<\text{Prompt}>\) ).

A major problem was the formal definition of our profile’s operational semantics which was necessary, since we intended to use the profile for the simulation of our models. We defined the profile’s semantics using ASMs based on the semantics for UML state diagrams in Ref. [28]. In this context, three influencing factors were taken into account: Temporal order of actions during state transitions, enter, exit and do actions, duration and termination of UML behaviour and behaviours of stereotypes, as well as the kind of each behaviour, namely synchronous versus asynchronous. To this end we introduced several ASM-macros and embedded them directly into the existing macros describing the behaviour of state diagrams. As already mentioned in Section 2.2, this was a temporary, pragmatic solution, a more generic approach for coupling UML and profile’s semantics is currently being investigated\[10\].

We have conducted several case studies to validate our profile’s semantics, our
HCI modelling approach in general, and we applied our MDD approach to HCI in order to get more experience in modelling complex systems.

Among others, we modelled and validated by simulation comprehensive parts of the telephone application of the recent Mercedes Benz C-Klasse. The complete model consists of 39 state diagrams: 16 for graphic-haptic operation, 4 for modelling of UI relevant system parts and 19 for speech operation. In total, the model contains 206 states (without pseudo states) and 546 transitions.

Our simulation consists of the model simulation including the complete semantics for state diagrams as well as the UI profile, and our UI simulation. It offers manual input by haptics using original in-car components, and composes and shows the currently active speech grammars as well as the active screen. This way we got a good impression of the multimodal in-car phone application HCI and a good basis for the comparison to the real system to be developed.

We also performed an expert evaluation to get qualitative feedback on our modelling approach from both the Software Engineering point-of-view and the HCI-domain one\cite{14}. None of the experts had any doubts about the general suitability of the approach. All of them rated the modelling approach as positive and gave suggestions for mostly minor further improvements.

We conclude, that our modelling approach for HCI (cf. Ref. [14]) is feasible for modelling multimodality except combined multimodal interaction, like speech and gestures, which has still to be examined. In particular this application also confirmed our strong belief that a formal semantics is a crucial point of every UML extension.

6 Conclusion

6.1 Summary

We address a very general difficulty with formalisms (cf. P1): expressiveness, integration and a formal semantic foundation somehow influence each other. Formalisms of great expressiveness like e.g. UML lack a formal semantics whereas specialised formalisms like state machines do not integrate different aspects such as static structures and dynamic behaviour.

Many formalisms lack a precise semantics (cf. P2), which — in our view — is indispensable for MDD to be successful (cf. H1). As we do not invent a new formalism (cf. H2), an obvious solution to P1 and P2 is using UML — which in fact is a well-known, highly expressive, and widely-used formalism — and defining its semantics formally. Such a definition is given in Section 2 for those parts of UML relevant to our work. The problem of model consistency (cf. P3) is also addressed there by combining different language units of UML.

We claim that a modelling language should be extensible in order to elaborate the adequacy of a formalism in a special context (cf. H5). Mechanisms for extending UML and defining a semantics for extensions are discussed in Section 2.2.

A problem which we have not yet solved is the question how to create high quality models on the basis of requirements (cf. P3). A central issue of our work is to make known techniques for quality assurance applicable to all behavioral models of UML (cf. H3). The incorporation of simulation and model checking in our approach is discussed in Section 3.2.
High quality models provide a basis for high quality implementations. Errors occurring during the process of implementing a model in code are a threat to the correctness of a model implementation (cf. P5). Hence, a code generator able to translate a complete model is a central issue for tools.

Our prototype is based on a formal semantics, supports quality assurance methods, comprehensively generates code, and is extensible. These features support overcoming the drawbacks of state-of-the-art tools (cf. P5) by incorporating an integrated tool chain (cf. H4).

Of course, not all aspects of our work are completely new, but we do not know of any integration of all presented issues into a single approach to MDD.

6.2 Next steps

Currently we focus on the extension, integration, and consolidation of our implementations. In particular, the extensions of the code generator for advanced concepts of class diagrams like composite aggregations and multiple inheritance need final polishing. Another issue is the completion of the code generator for activities.

Other issues which are considered to be future work are given in the following list, which is neither complete nor exclusive:

- method and tool support for development of initial models from requirements in natural language
- model transformations for optimisation and simplification
- quality issues like consistency checking, measuring of model quality and integration of comments and explanations to model elements
- generation of views (e.g. the generation of interaction diagrams from activities, state machines and class models)
- development and integration of an ASM-interpreter for supporting on-the-fly changes to the underlying semantics.

References


Raschke A. Zur automatischen Verifikation von UML 2 Aktivitätsdiagrammen [Ph.D. Thesis].
Addendum: Impressions of Manfred Broy

I know Manfred since more than half of his (and my) life. Based on fragments of memories, in the following I try to sketch my (obviously subjective) view on some aspects of his personality. Already shortly after he had joined our (CIP-)group in 1976, I could observe some of his characteristics, in particular the following ones:

- He has a very strong self-confidence. From the beginning, no problem seemed too hard for him. His first attempts in transformational programming dealing with Steinhaus-type permutations and the Ackermann-function were very ambitious, but nevertheless successful. As an immediate reaction to a scientific challenge posed to him by Albert Endres, he even had a bet on his success — and, of course, won. During his first participation in a Marktoberdorf summer school he, the scientific nobody then, consciously provoked a scientific disputation with the famous Edsger Dijkstra, and he had the better arguments.

- He has a very sharp mind and the remarkable ability to pinpoint weaknesses at a first glance. In fact, the confrontation with Dijkstra mentioned above, was caused by a very subtle error in Dijkstra's presentation which only Manfred recognised.

During the approximately six years of working together in the CIP-group under F. L. Bauer’s guidance, we had lots of technical discussions most of which finally resulted in some kind of common publication. During this time — but also later — more of Manfred’s virtues and abilities could be observed:

- He is a real hard worker that only stops working when the problem is solved. He characterises himself to be a workaholic and, indeed, he is among the extremely small number of colleagues who can be reached in their office already very early in the morning.

- He works extremely efficiently. I remember him drafting a nearly complete conference paper within only three hours during the surveillance of a written exam.

There are at least two more aspects characterising Manfred:

- He enjoys social life, good food and wine. Out of the many occasions to witness this, the following ones immediately came to my mind: We had several remarkable carnival parties in the institute in Munich, he organized a huge farewell-party when leaving to Passau, we
enjoyed nice (and wet) evenings during the first ADT-workshop when we all stayed at his and Martin’s flat in Passau, and I remember long evenings during the Ferienakademie in Sarntal, as well as a very nice dinner evening when he visited Nijmegen.

- He would not miss a good joke. Former colleagues from the CIP-group certainly remember a mysterious brick and empty bottles of champaign that appeared from time to time on the desk of a certain colleague (who was always puzzled, because he had no explanation for these mysteries).

It is the sound mixture of such characteristics that makes up a strong and successful personality — congratulations, Manfred.

H. Partsch