Model based development of hybrid systems: specification, simulation, test case generation*

K. Bender[1], M. Broy[2], I. Péter[1], A. Pretschner[2], and T. Stauner[3]

[1] Lehrstuhl für Informationstechnik im Maschinenwesen
Technische Universität München, Boltzmannstr. 15, 85748 Garching, Germany
[2] Software&Systems Engineering, Institut für Informatik
Technische Universität München, Arcisstr. 21, 80290 München, Germany
[3] BMW Car IT, Petuelring 116, 80809 München, Germany

Abstract. This paper gives an overview of our approach to the development of discrete-continuous systems in a general model based setting. This includes formalized description techniques, CASE support for modeling and simulation, and test harness as well as test case generation. HyROOM is presented, a formally founded notation for the integration of continuous activities into MaSiEd, a CASE tool prototype based on the ROOM methodology. In addition, an approach to the automated generation of test cases for discrete and also discretized hybrid systems specified within a second CASE tool, AutoFocus, is presented.

1 Introduction

The development of hybrid systems, which operate mixed discrete and continuous data streams, is an interdisciplinary task. Engineers from different disciplines are involved in their designs. On a conceptual level, the artifacts in question are operational abstractions of aspects such as functionality, structure, logical and technical (deployment) architecture, data, communication, scheduling, fault tolerance, and quality-of-service related issues.

Integration, one key aspect of model based development, is needed: it is desirable (1) for integrating these not entirely orthogonal aspects, concerning (2) the process and its different created artifacts over time, and (3) for different levels of abstractions. While not in general true, graphical description techniques in the domain of hybrid systems turn out to ease communication between engineers from different disciplines. The descriptions are representations of *models* that form the essence of the system under development during its stages of increasing precision that eventually lead to possibly generated production code.

It is difficult to envision model based development without machine support. Complex systems require sophisticated management and design techniques for consistent models and their relationship. Simulation and code gen-

* This work was supported with funds of the Deutsche Forschunsgemeinschaft under reference numbers Be 1055/7-1–7-3 and Br 887/9 within the priority programs *KONDISK* and *Design and design methodology of embedded systems.*

eration facilities are desirable in integrated model based development. Requirements tracing is impossible without tool support. Test case generation should be automated and therefore a feature of an integrated tool.

Just as for safety critical discrete systems, it is desirable to apply a high degree of mathematical rigor in the development of safety critical hybrid systems, provided it does not lay too much burden on the engineer, and it is simple. Formalism itself certainly does not solve any problems. Applications that require a (transparent) formalization include semantics-preserving design steps like refactoring, refinements from continuous to discrete time [Sta02,Sta01], and testing [PLP01]. Consequently, the two tools presented in this paper have been given a formal semantics.

This paper gives an overview of our activities in IMMA (Integrated Mathematical Machine Modeling), a project within the DFG priority program KONDISK. We cover model based development in general, modeling and description techniques, semantics, tool support, and test case generation. The nature of an overview article implies a high degree of abstraction. Technical details have been previously published: [SPP01,PPS00] define the formal semantics and present the case study. For refinements of HyCharts, e.g., the transition from continuous to discrete time, see [Sta02,Sta01]; automatic test case generation is treated in, e.g., [LP00,Pre01,PLP01,PSS00,SP02].

The paper kicks off with a brief overview of model based development and its relationship with approaches like HW/SW codesign or Simultaneous Engineering [BK95]. Being aware of the fact that model based development with integration along the dimensions of time (process) and content (product) is a rather ambitious undertaking, we then present partial approaches to implementing this paradigm. Two CASE tools are presented, MaSiEd and AutoFocus. MaSiEd basically integrates the Real time Object Oriented Modeling methodology (ROOM) description techniques with continuous activities as specified by Matlab block diagrams. The simplicity of the execution semantics of AutoFocus, on the other hand, is the basis for effectively and efficiently applying verification and validation techniques like model checking, theorem proving, or test case generation.

Following a glimpse of related work, we summarize the basic ideas behind model based development in Sec. 2. MaSiEd is described along the lines of a wire stretching plant (Sec. 3). With the application of generating test harnesses, this also includes the automated translation of hybrid scenarios into hybrid state machines. In Sec. 4, a compositional and incremental approach to the automated generation of test sequences for hybrid systems in AutoFocus is sketched. Sec. 5 concludes.

1.1 Related work

[Mos99] contains an overview of simulation packages for hybrid systems. The reason for presenting yet another description technique for hybrid systems is that in popular tools like Matrix_X or Matlab/Simulink/Stateflow systems

components are either discrete or continuous, but not both. Often these packages offer convenient, sometimes application specific, graphical description techniques, but, with the exception of Charon [AGH+00], a formal semantics is usually not defined for them. There are also simulation tools with a strong formal background, see [FvBR98]. Their focus is not on visual specification.

A central issue of our work is the research for a convenient modeling methodology for hybrid systems which is suitable for practice and can be put on a formal basis (see [Sta01] for a detailed overview). Therefore, most of the work cited above is complementary to our approach, either dealing with the modeling and simulation of hybrid systems, or with formal models for them. A notable exception is the work in the context of [FNW98] where UML's class diagrams are extended for hybrid systems and coupled with Z specifications.

There is a large body of literature on testing labeled transition systems (see [SP02] for an overview). Lack of space prohibits a description of all the available frameworks, tools and techniques (e.g., Lurette [RWNH98] or TorX [VTB+00]). The main difference with our approach is that we do not explicitly construct labeled transition systems but rather work on composed finite state machines that describe the behavior, which enables us (1) to compute with sets of values by means of symbolic execution, and (2) to easily incorporate heuristic search strategies.

2 Model Based Development

Even though the notion of model based development was coined a decade or so ago, the search for clear definitions of this concept results in a hard time. We give a brief overview of our understanding of this idea [SP02].

2.1 Models

Mastering the development of complex systems requires the use of suitably chosen abstractions for describing the essence of the system under development. This essence may differ for the points of view an engineer takes: it may be concerned with the above mentioned aspects of structuring the system, or with documentation, code generation, or analysis. This necessitates projections of integrated models.

For a particular purpose, abstractions discard details that are not relevant. Since they are simplifications, the artifacts under development become manageable. Clearly, for development, simplifications cannot go too far—remember that complexity is an essential rather than an accidental property of software. Embedding models, or rather code that is generated from them, into their target context (legacy systems, operating systems, sensors and actuators, different technical deployment architectures) obviously requires suitable concretizations. By now, we are only able to cope with them in an ad-hoc

manner. Automatization of this task is the subject of current work since we consider bridging the gap between the modeling and implementation levels to be the key challenge in model based development. However, models as simulations of actual programmable logic controllers (PLCs) allow for simultaneous engineering [BK95] of hardware and control systems which was one of the driving forces behind the development of the MaSiEd tool.

Undoubtedly, the development from Assembler to higher-level programming languages like Ada or C has caused an enormous increase in productivity. The essence of this transition lies in abstractions of control, data and program structures. In terms of control flow, constructs for procedures (no explicit call stack), repetition, sequence, alternative, and, more recently, exceptions have been incorporated into such higher languages. Structured data types with dedicated access mechanisms exempt engineers from treating data on a memory cell level. Many languages are equipped with abstractions for inter-process communication—just consider monitors or Linda as an implementation of the communication paradigm in tuple spaces. Concepts like modules allow for structuring and mastering larger projects. Abstractions are ubiquitous: some object-oriented and declarative language implementations provide automatic garbage collectors, and window toolkit APIs like Swing are readily available. Java's comprehensive libraries and the buzzword of componentware are further developments in this direction. Well understood, this list is far from being complete.

The vision of model based development is to take these ideas a step further. Necessarily domain-specific essential entities and their relationship are encoded in the (syntactical) meta model. In the case of embedded systems, these may be components, ports, connectors, etc. Concepts for describing behavior (functions, statecharts, Mealy machines, Petri nets) are also part of the meta model. For some application domains, e.g., time triggered bus architectures, synchronous Mealy machines may turn out to be a good choice. For others, like dedicated smart card operating systems with a focus on cryptography, Petri nets with their possibility of implicitly encoding command interleavings, may be a better choice.

The kind of properties, refinements, and semantics needed to describe a system are encoded in the system model. Meta and system model together form the product model. As stated above, in model based development, the product-oriented point of view has to be complemented by a process-centered perspective. Interrelated with the product model, the process model defines the different incremental development steps (add functionality, perform a refinement in the mathematical sense, etc.). This also includes coping with variants and versions of a system under development [SP02].

2.2 Process

The systematic use of models does not prescribe any particular process. In fact, processes like the Rational Unified Process or Cleanroom operate

with models as the basic entities. Languages/methodologies like the Spark or Ravenscar subsets of Ada encourage the use of abstract design but, like the RUP and Cleanroom, do not emphasize the dependency on a particular domain.[1] In fact, model based development should not be seen as the philosopher's stone for every single problem. It might turn out that it is beneficiary only in dedicated parts of an agile family of processes. We are concentrating on iterative processes (grow, not build software [PLP01]) with executable artifacts right from the beginning. Briefly, due to the possibility of frequently checking back with a customer, the key advantage of this kind of process is intellectual control over the process. Increments occur along the three aforementioned dimensions of level of abstraction, development over time—versions, variants, configurations, elaboration of aspects like function, data, etc.—, and projections for the purpose of analysis or generation. Due to the complexity of the involved systems, CASE support for development, requirements tracing, validation, and ensuring integrity is mandatory for model based processes.

In principle, the ideas of model based development also carry over to code-centered processes like Extreme Programming since the essence of a model is clearly independent of its representation. However, languages used in model based tools like AutoFocus deliberately restrict the power of general-purpose languages, as do Spark and Ravenscar (tasking). The reason is that this facilitates design steps that are correct by definition or that can be validated by machines (the generation of proof obligations would be a first step), which is in general impossible for full-fledged languages like Ada or C++. As a side benefit, using more abstract model-based, possibly graphical, notations renders systems development language-independent.

3 MaSiEd (Machine Simulator/Editor)

MaSiEd is a CASE tool for modeling, simulating and analyzing the I/O behavior of general discrete, continuous, and hybrid systems. It has been tailored to the needs of field bus based manufacturing systems with the aim of testing the associated PLC software. The possibility to create virtual machine models of manufacturing plants is a prerequisite for PLC tests.

3.1 Modeling discrete systems

The I/O behavior of modern manufacturing systems can be characterized as a mainly event driven discrete behavior (with incorporated continuous behaviors; the focus, however, is on discrete systems which decreases the adequacy of tools such as MatrixX that focus mainly on continuous parts). The MaSiEd CASE tool enables one to model reactive systems using the real time object

[1] Unless safety critical software is considered a domain—Spark allows explicit code annotations for verification purposes.

oriented modeling methodology (ROOM [SGW94], now substantial part of the UML-RT). ROOM's emphasis is on the seamless use of models from the requirement/high-level design phase down to the low-level design and testing stages. The primary concepts of the ROOM modeling language are actors, protocols, ports, bindings, and ROOMcharts, and they are used to model architectures consisting of hierarchies of communicating concurrent components.

An actor is a concurrent active object that hides its implementation from other actors in its environment. Fig. 2, left, shows an architecture diagram where actors are depicted as boxes. The behavior of actors is specified by a variant of the statechart formalism called ROOMcharts. ROOMcharts basically are extended state machines with hierarchic states, but unlike statecharts without parallel composition of states: parallel composition is defined using architecture diagrams like in Fig. 2, left. This formalism can model asynchronous event driven real-time systems.

3.2 Modeling continuous and hybrid systems

Even though the I/O behavior of most modern manufacturing systems can be mainly characterized as an event driven discrete behavior, there are, in addition, parts that have to be modeled in a continuous/hybrid manner.

The primary concepts added to ROOM in order to obtain the hybrid ROOM (HyROOM) modeling language are block diagrams, stores, and state activities. These concepts can be used to model hierarchies of communicating concurrent hybrid components. In order to support the modeling task of continuous subsystems we adopted the block diagram notation (Fig. 2, bottom right) as used in control theory. The block diagram notation is a widely used formalism for modeling, simulating, and analyzing dynamic systems. Block diagrams basically represent sets of differential equations. Note that block diagrams are, among other things, a means for architectural specifications of continuous systems.

For modeling hybrid systems, we extended ROOMcharts with the concept of continuous activities. Fig. 2, right, shows such an extended automaton. An ad hoc way of enabling control-loop behavior modeling is to specify a state's activity in the form of block diagrams. Variables assigned to connectors in the block diagram associated to the activity can be evaluated in the transition conditions belonging to the respective state. Numerical algorithms associated with the block diagram stop execution upon exiting from the state. Different actors in a model may be multi rate and thus updated at different rates.

The newly introduced concept of a *store* enables the transfer of real valued message data from state machines to block diagrams. The last message arriving in a store can serve as input to a block diagram. Stores may be connected to other actors with input for continuous or hybrid behavior or analog outputs to external hardware.

3.3 Modeling and simulation infrastructure

MaSiEd provides a user-friendly graphical design interface where hierarchical block diagrams and ROOM models with inheritance can be edited in the same environment. Inheritance on both the structural and behavioral levels provides a basis for reuse. In the same modeling environment, it is also possible to capture the system requirements using HySCs (hybrid Sequence Charts, e.g., Fig. 1, right) and later to use the captured requirements for validating the model.

MaSiEd includes an incremental model compiler to translate HyROOM models into C^{++} source code programs that are then compiled to run on a ROOM virtual machine. A DDE (dynamic data exchange) interface to Matlab/Simulink enables the use of an automatic C program segment generation based on Matlab Real-Time Workshop and the evaluation of continuous models in early stages of the development. The C code corresponding to the block diagrams translated by the Matlab Real-Time Workshop and the C^{++} code generated from the rest of the model are combined automatically. The generated model-specific code is linked with pre-compiled run time system libraries (MicroRTS, developed by ObjecTime Ltd.). Once the model compiled, it can be downloaded from the developing environment to a target computer running the VxWorks or RTLinux real-time operating system.

3.4 Example: Wire stretching plant

We chose to include a sketch of this industrial case study, previously described in [SPP01,PPS00], in order to show how different description elements—architecture diagrams, extended state machines, continuous block diagrams, and hybrid Sequence Charts—can be connected within the MaSiEd tool. The system's purpose is to wind wire of different thicknesses on reels. The case study was done in order to test the discrete process control; the actual PLC has been connected to MaSiEd for this purpose. The system's structure is as follows. The environment produces wire that enters the system at a variable speed. This wire has to be wound up on a reel. The turning reel's velocity has to be almost equal to the incoming wire's velocity in order to guarantee a homogeneously wound wire. It's velocity is controlled by a device between reel and environment, called the *dancer*, that consists of a set of pulleys the wire runs over (Fig. 1, left).

Not all of the pulleys are fixed so that the wire's velocity is dependent on the vertical position of the loose pulleys in this device. Once a reel is totally wound up it has to exit the system. This is achieved by a table that brings a new (empty) reel in position after the full one has been put on a belt. This is a complex, mostly discrete process that involves moving the table, fixing the new wheel on the motor's axis, cutting the wire, and making the new reel turn. There are two main conveyor belts involved in the system, one for empty, and one for wound up reels. This part of the system is omitted here for

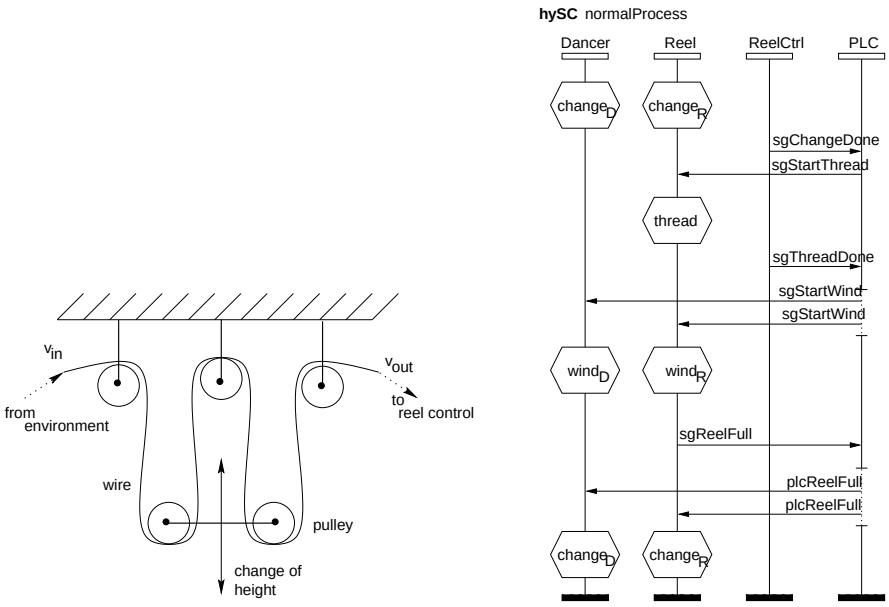

Fig. 1. Dancer (left), HySC: normal operation (right)

brevity's sake. In addition to hydraulic aggregates that guarantee the fixed position of a (turning) reel on the axis of the associated motor—the motor that interacts with the dancer via a controller for the turning speed—, the last main component of this system is the PLC part with roughly 180 I/O ports. The MaSiEd model consists of roughly 100 discrete actors, 20 block diagrams, and about 10 hybrid actors.

3.5 Hybrid subsystem

The hybrid subsystem that consists of the dancer, the DC motor for driving the reel, and the controller connecting the DC motor with the dancer is used for demonstrating the different description techniques. Its basic structure is depicted in Fig. 2, left, where continuous ports are marked with a semi-circle around a box. The systems input is the wire's continuously changing input velocity, v_{in}. The system communicates discretely with the PLC via port *pPLC*, and with the reel control via port *pReelCtrl*. The reel control takes care of exchanging a full reel in the system by an empty one.

Fig. 1, right, contains a hybrid Sequence Chart (HySC [GKS00]) depicting a typical use case for this system. HySCs are a variant of UML's Sequence Charts [Rat97] and use the standard Message Sequence Charts (MSCs [ITU99]) notation. Unlike MSCs, HySCs employ a synchronous time model. They use the MSC condition boxes (depicted as hexagons) to refer to the (qualitative) state of one or more components. Dotted parts of an axis in-

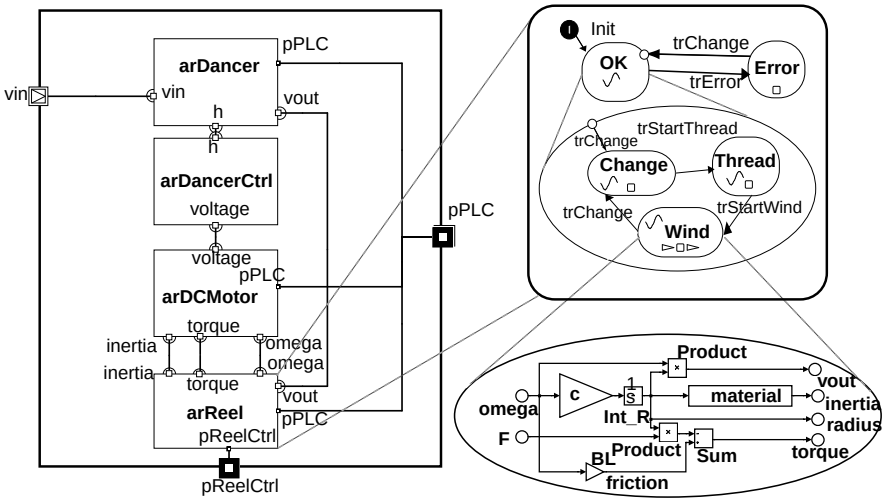

Fig. 2. Hybrid subsystem's architecture and reel's behavior

dicate that the associated signals occur simultaneously. In Fig. 1, right, a use case for the normal operation mode is specified: First, an empty reel has to be inserted in the system (states *change*). Once the change is done, the *threading* process starts; the wire is put onto the new reel, and it is cut from the old one. If this process successfully completes, the actual *winding* process is initiated; compared with the *change* state, its main characteristic is a relatively high velocity of the reel. When the reel is full, the PLC re-initiates the process of *changing* the reel by moving the full one out of the system and bringing an empty one in position. For the sake of brevity, we omit the predicates that describe the states as well as the differential equations describing the different continuous behaviors. Fig. 2, bottom right, exemplifies the use of block diagrams. The DC motor is a standard PID controlled motor with its own controller. Its inputs are a voltage (which is proportional to the PID controlled dancer's height) that directly controls the motor's angular velocity as well as the reel's torque and inertia. We omit the (standard) details for brevity's sake. The *motor* component consists of just two states, *on*, and *off*. The third hybrid component of interest is the reel itself. Given the wire's input velocity, it keeps track of the reel's inertia, its torque, and its continuously growing radius (wire is being wound up; e.g., Fig. 2, bottom right).

When state *change* is reached, actor *reel* is reset: the reel's radius is set to zero When the new reel has been fixed to the motor, the wire then needs to be threaded in state *thread*.

3.6 From HySCs to state machines

HySCs also support test case implementation in MaSiEd. Traditional testing of simulation models is manual, time consuming and error prone. In order to facilitate model testing, MaSiEd supports the automatic generation of complete unit and integration test harnesses directly from HySC specifications. In contrast to the generation of test cases that we describe in the next chapter, scenarios have to be fully described.

The algorithm used in MaSiEd for the automated synthesis of complete unit and integration harnesses directly from HySC test case specifications is based on the maximum progress algorithm [LMR98]. Test case specifications in form of HySC are analyzed with respect to their software architectural content, including structure and behavior, and are represented in terms of HyROOM. Every concurrent instance (axis) in the HySC specification is represented by exactly one concurrent HyROOM actor. The motivation for the maximum progress algorithm is to determine maximum progress transitions in the HySC specification and to map these onto HyROOM behavior descriptions. This means that synthesized HyROOMchart transitions can span events originating from more than one HySC. Since we do not use a hierarchical state machine structure, the synthesized ROOMcharts will be flat. One HyROOMchart per instance in the HySC specification is generated.

3.7 Semantics

MaSiEd has been given a precise formal semantics [SPP01] on the grounds of HyCharts [GSB98]. Roughly speaking, HyCharts provide means of specifying both structure (HyACharts) and behavior (HySCharts, DiCharts) of hybrid systems. The semantics is given by stream processing functions [Bro01]: (infinite) input trajectories are mapped to output trajectories. While developed independently, it turns out that there is a natural semantic mapping from HyROOM into HyCharts. We omit any technicalities for the sake of brevity; the semantics is defined in [SPP01]; applications like program transformations are treated in [Sta02,Sta01].

4 Model Based Testing with AutoFocus

Formal methods like model checking and theorem proving are concerned with properties of a *model* that provides an abstraction. Proving or approximating properties of the actual *implementation* is the mandatory second step. Model based testing includes generating test cases from models and executing them. These test cases are used for testing different iterations (and/or projections) of the current stage of the product. Besides disambiguating requirements, the aim is to reach a valid model of a system. Generating test cases is thus a part of the requirements capture as well as implementation or design activities.

Models are used for hardware-in-the-loop simulations, for generating production code, or for validating existing systems. In the latter case, the idea

is to perform conformance tests of a system with its model. This may require suitable concretizations of the respective test cases. Clearly, an automatic assignment of verdicts for functional test cases only makes sense if the same model is not used for generating production code and test cases. Otherwise, the system would be tested against itself (in this case, test cases may not be suitable for establishing functional conformance, but they might help in verifying environmental assumptions or the correctness of code generators). Using models for verification is a natural choice if for organizational reasons, quality assurance and implementation departments are to be separated, if efficient code generators for a particular target language do not exist, or if the system contains large legacy parts.

Currently, we are unable to generate test cases from MaSiEd models. The main reason is the use of C++ as transition annotation language which in general eludes automated formal analysis. This is why we implemented a test sequence generator for the CASE tool AutoFocus [HSE97] which uses a functional language for guard specifications instead. The remainder of this section briefly describes AutoFocus, and explains how test case generation with Constraint Logic Programming (CLP) is performed. Note that what we do here is different from the generation of test harnesses from HySCs as described above where full discrete control and continuous signal information is to be provided. The technique described here aims at computing this information.

4.1 AutoFocus

Similar to MaSiEd, the main description elements of AutoFocus are concerned with structure, behavior, data, and interaction specifications as encoded in the meta model. Hierarchic system structure diagrams depict components (actors, capsules). They encapsulate data and behavior, and they thus provide a means of functionally decomposing a system. Bottom level components are assigned a behavior in terms of a Mealy-like state machine. Transitions consist of statements that read input channels, of a guard for establishing whether or not a transition may fire, assignments that update local variables, and of statements that compute outputs.

Guards and assignments are specified in a Haskell-like functional language. Components communicate over typed channels. The rationale for using a functional language for typing is that in embedded systems, data modeling with elaborate constructs like class diagrams is rarely necessary. Simple sum and product types turn out to be sufficient.

Similar to clock-synchronous hardware circuits, all components perform their computations simultaneously: they read values from their input channels, compute updates for local variables and output channels, and write these updates so that at the next clock tick, the values are available. This results in a time-synchronous communication scheme with buffer size one—staying with the analogy of clocked hardware, each channel contains an implicit latch, or shift register, respectively. The rationale behind choosing this admittedly restricted semantics is that it is exactly this simplicity that allows AutoFocus

models to be formally analyzed, e.g., model checked, or used for test case generation. By using recursive list types, it is also possible to implement asynchronous communication. This semantics is inherently discrete. Continuous system parts are coped with by discretization [PSS00]. Matlab block diagrams are automatically translated.

4.2 Test case generation

It turns out that the simple clock-synchronous semantics is naturally encoded by Horn clauses with axiomatizations of natural numbers or reals [LP00]. The resulting CLP code may be used for simulation by giving inputs to the system for each step, similar to what is done with other simulation code generators as well. It is also possible to partially specify inputs, outputs, or constraints over them—for instance, a maximum number of signals to occur, or temporal dependencies—without specifying their exact timing. By enumerating all traces of a bounded length, the LP engine then computes those traces that satisfy the constraints imposed on inputs, outputs, states, transitions, or local variables.

Conceptually, the generation of test sequences is hence achieved by formalizing the test purpose by means of existential specifications of the kind "given a set of constraints, make the system reach state q_1, q_2, etc." where each q_i specifies a desired constrained value of the variables for control states, data states, inputs, or outputs. The resulting I/O traces are the test sequences we are interested in. These existential test case specifications are sufficient for covering use cases from requirements capture activities or finding test sequences that satisfy a given coverage criterion. These can be reduced to a set of test case specifications, each of which makes the system reach a certain state or condition.

Computationally, this would be too simple to work efficiently. In fact, our approach is akin to bounded explicit model checking or other state space exploration techniques. State space explosion is the commonly accepted hindrance of these approaches for acceptance in the industrial practice. We use dedicated heuristic A*-like search algorithms in order to find those q_i we are interested in [Pre01]. Furthermore, our system allows for explicitly specifying environmental and efficiency constraints for manually pruning the search tree. In terms of continuous or hybrid subsystems, environmental and efficiency constraints may include gradients of the respective curves, or restrict certain values to given intervals. This kind of constraints is taken care of by predefined constraint solvers connected to typical available CLP systems.

This not only reflects the need for manual intervention; experiences with industrial partners have shown that test engineers are in fact capable of identifying those parts of a system that may be sliced away. Constraints are used for taking care of temporal dependencies, numerical properties, excluded or enforced occurrences of certain signals. Furthermore, they allow to compute with and efficiently store sets of states [Pre01].

Test case specifications may also include restrictions of the search space. They are provided directly as constraints with temporal operators, as sequence diagrams, or as finite state machines. In this latter case, the test case specification often is a combination of a partial environment model and the formalized test purpose. When testing protocols, for instance, the test case specification, given as an automaton, specifies certain typical runs or threat scenarios. It is also possible to define transition probabilities. As in the case of general models, the essence of a test case specification clearly is independent of its representation, be it a formula, a sequence diagram, or a state machine.

Specifications do not contain only existential properties. Universal properties like invariance, safety, or liveness are also specified. Since testing is, by definition, a finite activity, these properties cannot be tested exhaustively. We thus approximate the universal property by a set of existential properties. Justified by the success of limit testing in the setting of testing transformative systems, we compute traces that come as close as possible to a state that violates the invariance. This is done on the grounds of the same A* like heuristics used for finding particular elements in the state space [Pre01].

4.3 Process: procedure, regression tests, compositionality

Test sequence generation proceeds as follows. The automatically translated AutoFocus model is conjoined with the (existential) test case specification, environmental and efficiency constraints. The resulting test sequences are used for debugging the model itself. This is done by (manually) comparing every I/O sequence to what one would have expected—at this stage, there usually is no formal operational specification to compare with. Instead, the model itself is the executable specification.

In an incremental setting, models are developed iteratively. For the sake of brevity, we only consider increments that add functionality to a model. If feedback from the customer suggests changes in increment $\mathcal{I}_n$, it becomes a modified part of increment $\mathcal{I}_{n+1}$. $\mathcal{I}_n$ might also remain unchanged in $\mathcal{I}_{n+1}$. For each $\mathcal{I}_j$, we consider functional and structural[2] test case specifications to be given by the engineer. The test case specifications are then used for computing actual test sequences.

These traces can be computed separately for each of the increments. Validity of the traces has to be checked manually. It is, however, possible, to use test sequences for increments $\mathcal{I}_j$, $\mathcal{T}(\mathcal{I}_j)$, with $j \leq n$ for regression testing increments $\mathcal{I}_k$ for $k > n$. We simply feed the test sequences $\bigcup_{j=1}^{n} \mathcal{T}(\mathcal{I}_j)$ into $\mathcal{I}_{n+1}$, and are hence able to automatically assign verdicts to these tests. $\mathcal{I}_{n+1}$ is checked for conformance with $\mathcal{I}_j$ for $j \leq n$.

These verdicts have to be taken with caution. The problem is that adding functionality may actually restrict the behavior of a system; false negatives

[2] When adequately modeled, structural coverage criteria like state coverage may well be considered as functional tests. This is because each control state encodes a certain functional unit.

are the result. This is, for instance, the case if a timer that periodically emits a timeout is composed to a system $\mathcal{I}_n$. The test sequences for $\mathcal{I}_n$ may consist of traces that respond to timeouts that occur erratically.

By inverting the above idea, we get a compositional approach to generating test cases. Consider a system $\mathcal{I}_{n+1}$ consisting of increment $\mathcal{I}_n$ that is composed with a component k such that there is a channel between the two in each direction. It is then possible to use $\mathcal{T}(\mathcal{I}_n)$ for generating test cases for k and for $\mathcal{I}_{n+1}$. We can use the outputs of $\mathcal{T}(\mathcal{I}_n)$ as a driver for k, and thus get new test sequences for k, and, consequently, for $\mathcal{I}_{n+1}$. Conversely, we can use the inputs of $\mathcal{T}(k)$ as putative outputs of $\mathcal{I}_n$. Remember that using CLP allows us to partially specify outputs and make the system compute those fully instantiated I/O traces that eventually result in the specified output. Ignoring the problem of running into the same problem as with regression testing, we directly get new test sequences for $\mathcal{I}_n$, and for $\mathcal{I}_{n+1}$.

4.4 Example

We do not give the AutoFocus diagrams of our case study here since, apart from block diagrams, they are almost identical to the MaSiEd specification. Neither do we provide any actual test case specifications or computed test sequences for this system since this would require a rather deep level of technicality. We do, however, give some informal test purposes that readily translate into formalized test case specifications and that we have used for test sequence generation. Among others, test purposes include the following. For each of the discrete PLC, environment, and other components coverage on states, transitions, or guards is a test case specification. Reflecting the composition of components, these unit test sequences are combined in order to derive new test sequences for the connected components, as described above. Furthermore, for the dancer, there are HySCs from the requirements capture activities. We easily translate these into automata and use them as test case specification such the diagram depicted in Fig. 1, right. As a last example, in terms of universal properties, we compute a test suite for the property *whenever state Error is reached, we can escape from it.* Clearly, many more test case specifications are conceivable. For the sake of brevity, we omit the discussion of assessing the quality of a test suite.

5 Conclusion

Major advances in software and systems engineering seem to be bound to the use of abstractions as the key metaphor. Artifacts at increasing levels of abstraction enable intellectual control over highly complex systems. Integrated tool support, ranging from specification, implementation, verification to requirements tracing and documentation is desirable for an efficient workflow.

We have presented our approach to model based development which relies on suitably chosen abstractions for the essential constructs in a particular

domain. Tool support for modeling, simulation, code generation, and test case generation for two CASE tools, MaSiEd and AutoFocus, has been presented.

Whether or not CASE support with graphical description techniques rather than using dedicated IDEs like Forte or Eclipse is the right choice, is not obvious. In a model based setting, IDEs for language subsets like Spark or Ravenscar in addition to test tools may turn out to be the more practicable approach. The arguments that graphical description techniques facilitate the understanding of a system loose validity with increasing complexity of the system under development. In fact, misuse of hierarchic statecharts makes system designs foggy, as does misuse of inheritance in class diagrams.

The step from models to implementations may involve adding technical details that are not relevant in early development phases. Real time issues demonstrate, however, that low level technical details may have to be considered right from the beginning. We are convinced that in many areas, it is possible to achieve a seamless integration of abstract models and low level technical issues (for instance, this is certainly true for PLCs as considered in the case study of this paper, or for smart cards). If, in general, this turns out be an illusion, then model based development boils down to a philosophy of the activities of requirements engineering, and clearly remains most valuable in that it allows for intellectually mastering the complexity of large systems.

We are convinced of the necessity of a transparent, precise semantics. However, simplicity should be a key factor when formalizing it—otherwise, there is a formal semantics, but engineers will not have the time to deeply understand it. A clear understanding of the meaning of an artifact is the prerequisite for transformations, be they refinements [Sta01,Sta02] or refactorings. They are also necessary for code generation and validation techniques like test case generation and execution. Formal semantics for the two tools have been defined but are not part of this paper.

MaSiEd was presented, a tool for modeling and simulation of hybrid systems specifically targeting at the application field of process automation. Roughly, MaSiEd integrates the ROOM virtual machine with Matlab block diagrams. The modeling concepts, an extension of ROOM, have been described and demonstrated along the lines of an example system taken from an industrial case study.

In terms of ROOM based modeling, [PSS00] as well as the case study in this paper showed that the clear distinction between structure and behavior results in the need of copying the same set of states from one component to another in the same subsystem. This problem is alleviated by the use of MaSiEd's inheritance mechanism, but the general problem still persists (it does not in statecharts for there is no clear differentiation between structure and behavior as well as no concept of interfaces).

Finally, AutoFocus was presented. Due to the simplicity of its semantics, it is possible to derive test sequences for discrete or discretized systems. The idea is to use a combination of symbolic execution and state space exploration

with heuristic search on the grounds of Constraint Logic Programming. The embedding of this approach into an incremental model based development process was described. This technique is a complement to the generation of HyROOMCharts from HySCs (i.e., test harnesses from scenarios) since in this latter case, complete information about signals and their temporal dependencies have to be provided. The AutoFocus based approach aims at computing this complete information.

In industrial practice, test cases are seldom developed systematically. If they are, engineers often use coarse discrete abstractions (e.g., "quickly accelerate" or "slowly accelerate") of a system in order to identify interesting scenarios. Clearly, (mis)using condition or state boxes of HySCs to this end directly lends itself to the specification of test cases with HySCs. The test case generation procedure profits from this abstractions since the model becomes less complicated.

Future work includes machine support for sound refinements and refactorings of hybrid systems. The integration of hybrid class diagrams into MaSiEd is the subject of current work. In terms of the test case generator, we currently assess its applicability in various industrial projects. The question of how to automatically extract "good" test suites is yet unsolved; we consider the analysis of error classes in a particular domain a first step in the right direction.

Acknowledgment. We would like to thank Lingxiang Xu for providing the original discrete case study. In numerous discussions, J. Philipps, B. Schätz, F. Huber, W. Schwerin, and P. Braun provided valuable insights into the nature of model based development.

References

[AGH+00] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in Charon. In *Proc. HSCC'00*, Springer LNCS 1790, 2000.

[BK95] K. Bender and O. Kaiser. Simultaneous Engineering durch Maschinenemulation. *CIM Management*, 11(4):14–18, 1995.

[Bro01] M. Broy. Refinement of time. *Theoretical Computer Science*, 253(1):3–26, 2001.

[FNW98] V. Friesen, A. Nordwig, and M. Weber. Object-oriented specification of hybrid systems using UML^{h} and ZimOO. In *Proc. 11th Int. Conf. on the Z Formal Method (ZUM)*, LNCS 1493. Springer, 1998.

[FvBR98] G. Fábián, D. A. van Beek, and J. E. Rooda. Integration of the discrete and the continuous behaviour in the hybrid chi simulator. In *1998 European Simulation Multiconference, Manchester*, pages 207–257, 1998.

[GKS00] R. Grosu, I. Krüger, and T. Stauner. Hybrid Sequence Charts. In *Proc. of ISORC 2000*. IEEE, 2000.

[GSB98] R. Grosu, T. Stauner, and M. Broy. A modular visual model for hybrid systems. In *Proc. of FTRTFT'98*, LNCS 1486. Springer-Verlag, 1998.

[HSE97] F. Huber, B. Schätz, and G. Einert. Consistent graphical specification of distributed systems. In *FME '97: 4th International Symposium of Formal Methods Europe, LNCS 1313*, pages 122 – 141, 1997.

[ITU99] ITU. ITU-T Recommendation Z.120: Message Sequence Charts (MSC), 1999.

[LMR98] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing ROOM Models from MSC Specifications. Technical Report TR-98-06, University of Waterloo, 1998.

[LP00] H. Lötzbeyer and A. Pretschner. AutoFocus on Constraint Logic Programming. In *Proc. (Constraint) Logic Programming and Software Engineering*, 2000.

[Mos99] P. J. Mosterman. An overview of hcbrid simulation phenomena and their support by simulation packages. In *Hybrid Systems Computation and Control (HSCC'99)*, LNCS 1569. Springer-Verlag, 1999.

[PLP01] 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.

[PPS00] I. Péter, A. Pretschner, and T. Stauner. Heterogeneous development of hybrid systems. In *Proc. GI workshop Rigorose Entwicklung softwareintensiver Systeme*, pages 83–93, 2000.

[Pre01] 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.

[PSS00] A. Pretschner, O. Slotosch, and T. Stauner. Developing Correct Safety Critical, Hybrid, Embedded Systems. In *Proc. New Information Processing Techniques for Military Systems, NATO Research*, 2000.

[Rat97] Unified modeling language, version 1.1. Rational Software Corporation, 1997.

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *Proc. 19th IEEE Real-Time Systems Symposium*, 1998.

[SGW94] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons Ltd, Chichester, 1994.

[SP02] B. Schätz and A. Pretschner. Model based development of embedded systems. Submitted to Model-Driven Approaches to Software Development, OOIS'02, 2002.

[SPP01] T. Stauner, A. Pretschner, and I. Péter. Approaching a Discrete-Continuous UML: Tool Support and Formalization. In *Proc. UML'2001 workshop on Practical UML-Based Rigorous Development Methods*, pages 242–257, 2001.

[Sta01] T. Stauner. *Systematic development of hybrid systems*. PhD thesis, Technische Universität München, 2001.

[Sta02] T. Stauner. Discrete-Time Refinement of Hybrid Automata. In *Proc. HSCC'02*, 2002. To be published.

[VTB+00] R. de Vries, J. Tretmans, A. Belinfante, J. Feenstra, L. Feijs, S. Mauw, N. Goga, L. Heerink, and A. de Heer. Côte de Resyste in Progress. In *Progress 2000 – Workshop on Embedded Systems*, pages 141–148, October 2000.