To appear in Proc. New Information Processing Techniques for Military Systems, NATO Research and Technology Organization, Istanbul, October 2000.

Developing Correct Safety Critical, Hybrid, Embedded Systems*

Alexander Pretschner, Oscar Slotosch, Thomas Stauner
Institut für Informatik, Technische Universität München
Arcisstraße 21, 80290 München, Germany
{pretschn,slotosch,stauner}@in.tum.de

Abstract

Several aspects of the development process of correct safety critical discrete and hybrid embedded systems are discussed. The general process and its support by the CASE tool AutoFocus is outlined. This is illustrated along the lines of a simplified version of NASA's Mars Polar Lander. It is argued that specific aspects of hybrid systems do require the modification of classical theories on software development, and these modifications are discussed. The paper concludes by focusing on one part of the development process, namely testing. A novel approach to the automated generation of test cases for discrete as well as hybrid systems is presented. The Mars lander's crash serves as an example for the derivation of meaningful test cases.

Keywords. Reactive Systems, Validation, Development Process, Automatic Test Case Generation, CASE

1 Introduction

Safety critical systems. Developing correct safety critical software for hybrid, embedded systems is a difficult and error prone task. The functional reliability of the resulting systems is at least as important as security aspects. High quality of the resulting systems can only be achieved using a well structured development process. We present a development process that integrates many methods for quality assurance for discrete systems. In this paper, *discrete* systems refer to discrete event systems. However, those discrete event specification techniques we consider also have a discrete-time execution model. For mixed discrete-continuous systems (or "hybrid systems") we discuss elements necessary to obtain a similar integrated process, taking into account discrete as well as continuous aspects. The process for discrete systems is an extension of the V-model [22]. It is based on system models that are validated by means of formal techniques.

AutoFocus. Discrete systems are modeled using graphical description techniques for structure, behavior, and interaction. In this paper we shortly present AutoFocus, a tool prototype for modeling discrete embedded systems that we will use to model a hybrid system, namely a simplified version of NASA's crashed Mars Polar Lander. The models are based on a common formal semantics and can therefore be used to support the development process from the requirements engineering phase throughout the test phase. The existing features of AutoFocus suffice to model discretizations of hybrid systems. These discretizations, however, usually alter the model which reduces the set of properties that can be derived from a system model.

Hybrid systems. The development of hybrid systems is an interdisciplinary task. Usually engineers from different disciplines are involved and must discuss their designs. Graphical description techniques are one element very useful to support this communication. Just as for safety critical discrete systems, it is furthermore desirable to apply a high degree of mathematical rigor in the development of safety critical hybrid systems. Today formal methods for hybrid systems are still an active area of research, and there are hardly any tools available which could yet be used in industrial practice. In this paper we therefore outline how formal tools for discrete systems (such as AutoFocus) can be extended with aspects for continuous systems in a development process for hybrid systems that is feasible today. We discuss shortcomings of this method and outline how our work on hybrid modeling and validation leads to an *integrated* development process for hybrid systems which is close to the current AutoFocus approach for discrete systems.

Testing. Much work has been devoted to checking validity and consistency of a specification. By now, testing is the only practicable, scalable means of validating the conformance of an implementation w.r.t. its specification, even though Dijkstra's popular remark that testing can only reveal the presence but never the absence of errors undoubtedly holds true. We discuss the role of testing as a complement to formal methods and present a novel approach

*This work was supported with funds of the Deutsche Forschungsgemeinschaft under reference numbers Br 887/9 and Be 1055/7-2 within the priority programs *Design and design methodology of embedded systems* and *KONDISK* [10], and by the DASA.

based on Constraint Logic Programming to automatically generating test cases for discrete as well as hybrid systems. Experimental results from a case study of a safety critical system, the Mars Polar Lander, are discussed. We also take into account the integration of testing processes into the development process of safety critical systems and our method's benefits and shortcomings.

Overview. The remainder of this paper is organized as follows. In section 2, we briefly discuss principles of a software development process for reliable discrete systems. Section 3 describes shortcomings of this classical approach w.r.t. hybrid systems and suggests modifications that remedy these shortcomings. Specifically, we argue for integrated description techniques right from the beginning. Section 4 then describes the CASE tool AutoFocus and its description elements. Since so far there is no tool support for the integrated development process of Sec. 3, section 5 exemplifies the use of a discrete modeling tool for a hybrid system, the Mars Lander. Section 6 discusses the generation of test cases for discrete as well as hybrid systems along the lines of our example. Section 7 concludes this paper. Related work is cited in the respective sections.

2 Notes on the Development Process

For safety critical systems, we advocate a development process that heavily relies on *models*. In this process, models are developed in phases known from conventional software development processes. These models are then validated, and the last step is to generate code from them in order to get an executable implementation. A last validation step consists of testing the developed systems in interaction with their environment, for example together with other components or hardware. Model validation is the key factor for producing highly reliable programs for safety critical systems. Useful models should describe the developed system from different views by means of various hierarchical diagrams. The diagrams can be used to capture requirements, architecture, design and implementation decisions, and to represent test sequences.

Model validation may be seen as checking consistency. It can be applied at several levels: syntactic consistency (checking names), completeness consistency (checking references and types), semantic consistency (checking refinement relations), and adequacy [28, 4], i.e., conformance of a model with possibly informal requirements. Many available tools perform a syntax check with fixed built-in routines. For this purpose, modern tools use the object constraint language OCL [38] of UML which, in principle, could also be used to check completeness of the models. At present, CASE tools with useful semantic checks are not available. Recently, model checking tools have been connected to tools based on statecharts or SDL, but due to the complex semantics of these languages without practical relevance. A systematic generation of test sequences is also not available. Checking adequacy is reduced to simulation facilities.

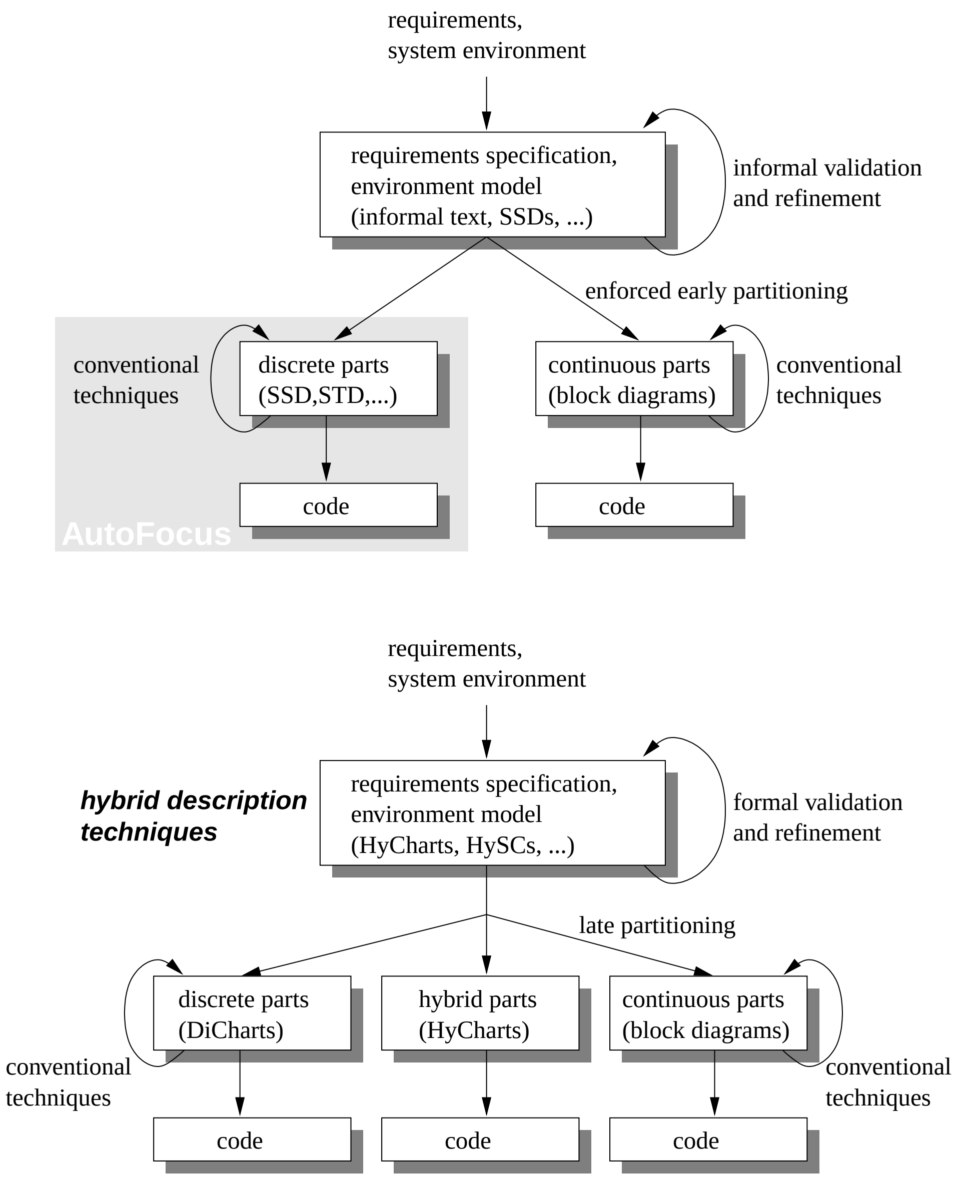
AutoFocus, on the other hand, also allows for semantic validation. These validation techniques are based on the simple and intuitive semantics of AutoFocus [21], and can be used to support model-based development steps according to the V-model within all phases; in particular, testing is supported at the design, implementation, integration and system requirements levels.

Modeling hybrid systems with AutoFocus is done by a simple discretization of the continuous behavior, and this approach allows to integrate discrete and continuous parts in a single model (as will be shown in the example in Section 5).

3 Hybrid Systems

In this section we outline how a conventional formal tool for discrete systems, such as AutoFocus, can be used within the development of hybrid systems. We discuss advantages and drawbacks of such an approach, and present a more visionary approach not yet supported by tools. The new approach is supposed to prevent these drawbacks. It results from carrying over ideas like graphical specification with different systems views and model based validation based on formal methods to hybrid systems. A central characteristic of the proposed approach is that it is based on notations that have a clearly defined semantics.

A conventional development process. A conventional development process for hybrid systems builds upon isolated description techniques for purely discrete and purely continuous components. Popular in industry are tool couplings such as using Statemate together with Matrix$_X$ or the MATLAB/Simulink/StateFlow environment [11, 9]. For the development of safety critical systems we advocate the use of formal methods and notations wherever possible. This hinders the use of current commercial tools like Statemate, ObjectGeode, Rational RoseRT or Stateflow. Their notations only have a formal syntax, but *the semantics remains imprecise and ambiguous, or very complex.* A semantics for the coupling with continuous tools in not defined anyway. However, tools like AutoFocus are available which support the design of discrete systems based on formal notations such as architecture descriptions, extended automata and MSC dialects [23]. For continuous systems there also are analysis and simulation tools based on block diagram notations, e.g. MATLAB [36]. Note that we regard block di-



requirements, system environment
requirements specification, environment model (informal text, SSDs, ...)
informal validation and refinement
enforced early partitioning
conventional techniques
discrete parts (SSD,STD,...)
continuous parts (block diagrams)
conventional techniques
code
code
AutoFocus
requirements, system environment
hybrid description techniques
requirements specification, environment model (HyCharts, HySCs, ...)
formal validation and refinement
late partitioning
discrete parts (DiCharts)
hybrid parts (HyCharts)
continuous parts (block diagrams)
conventional techniques
conventional techniques
code
code
code

Figure 1: A conventional development process (top) and an integrated development process with hybrid description techniques (bottom).

agram descriptions of continuous systems as formal here, because a mathematical model can be associated with individual blocks and their interconnection in a straightforward manner. (Nevertheless, the user has to keep in mind that the selection of integration algorithms for simulation can have a great impact on simulation results and can cause them to differ strongly from the mathematical model.) As soon as the system under development is partitioned into discrete and continuous parts, a formal specification can be written down using these existing tools, see Figure 1, top. Well-known techniques from the discrete and continuous world can then be applied to the respective parts of the model. For instance, model checking and automatic test case generation may be used for the discrete part and analysis of eigenvalues for the continuous part.

Drawbacks. So far, the only currently available technique for examining properties of the mixed system is simulation. There are hardly any analytical methods regarding the mixed model and there are no techniques which support design modifications that affect both parts of the model. In fact such modifications could necessitate a redesign of the whole model.

Furthermore, in such a development process a designer has to perform a number of development steps informally, i.e., without documenting them with clearly defined notations, before a clearly documented process can start, i.e., a process relying on formal description techniques. In particular, these steps include partitioning the design into discrete and continuous parts which may involve an (implicit) discretization of some parts. This is unsatisfactory since the partitioning decisions may be difficult to alter later on. Apart from that, the resulting coupled discrete continuous model often is not natural for some components of a hybrid system. For example, analog to digital (AD) and digital to analog (DA) converters, and in some systems the environment, are inherently hybrid components.

Recommendation. As no formal hybrid notations with tool support that is suitable in practice are available today, there currently is no real alternative to the outlined conventional development process. We therefore propose to use informal text coupled with formal descriptions where possible in this process: Writing down mathematical formulae expressing hybrid behavior directly is hardly reasonable for bigger systems. In the context of AutoFocus we recommend using architecture descriptions (SSDs, see Sec. 4) already during the requirements capture phase and to describe the behavior of system components informally with text or, where practicable, with mathematical formulae, until the partitioning into discrete and continuous components has been performed. Figure 1, top, outlines the conventional process and its support by AutoFocus. For the discrete part the model checking and testing techniques already implemented in the tool can be employed. They enable a more rigorous analysis of the discrete part than what is possible with other tools for discrete systems that do not have a formal semantics.

Outlook: An integrated development process. In a development process with hybrid description techniques, such as the one depicted in Figure 1, bottom, the designer is able to formally specify mixed discrete continuous models at early stages of the development process. In the context of formal methods, we refer to "refinement" as altering (or augmenting) a system's functionality without violating properties that have already been established. If validation and transformation techniques, such as simulation and refinement, are available for these description techniques, the model can be *systematically* designed to meet those system requirements which affect its discrete as well as its continuous aspects. Rudimentary versions of such techniques already exist and are an area of current research (e.g. [3, 10]). In later steps the model can be refined into discrete, continuous, and possibly some remaining hybrid submodels. For the discrete and continuous submodels conventional techniques can then be used to realize those properties which only affect the respective part. Thus, the availability of formal hybrid description techniques and supporting methods for them pushes the point at which systematic development, i.e. development with formal description techniques, can begin towards the beginning of the analysis phase. A partitioning into discrete and continuous submodels can be postponed towards subsequent development phases. Such a development process with hybrid description techniques allows to obtain greater confidence in the model before a partitioning. Namely, testing and model checking techniques can be used to analyze requirements and refinement techniques can be used to guarantee some requirements by construction. By postponing implementation related questions changing requirements can more easily be taken into account. Thus, errors made in the initial development phases can be found earlier and are therefore cheaper to correct.

The development process we propose in Figure 1, bottom, is based on description techniques developed within our group in the last years. For requirements specification and environment modeling it uses the MSC-like notation *HySC* [15], and the combination of architecture diagrams and a hybrid automata variant which is subsumed in *HyCharts* [16]. A methodological transition from HySCs to HyCharts is ongoing work (for similar work on discrete systems see [24]). Succeeding steps in the figure refer to HyCharts rather than to HySCs. As notations for the discrete and the continuous part we propose DiCharts [17], a discrete-time variant of HyCharts, and (continuous time) block diagrams, repectively, taht can be integrated easily into the HyChart notation.

Note that the aspect of postponing the partitioning of a system into discrete and continuous parts is related to the area of hardware/software codesign [6]. There, the decision on which parts of a system are implemented in hardware and software is postponed to later phases. However, unlike hardware/software codesign the partitioning into discrete and continuous components proposed here does not yet imply how the components are implemented. The discrete part could be implemented in software or on digital hardware, the continuous part can be turned into a discrete-time model and implemented in software (or digital hardware), or it could be implemented in analog hardware.

While there is hardly any tool support for the integrated process today, a close coupling of discrete and continuous notations in the HyChart style is implemented in the *MaSiEd* tool [1], which also allows simulation. The *HyTech* tool [18] (or other tools, e.g., Uppaal or Kronos) which offers model checking of hybrid models is another element needed as support for an integrated development process. Presently, however, its application is limited due to scalability problems and deficits of the underlying hybrid automata model [29]. Promising tool approaches for the future should couple analysis algorithms like those implemented in *HyTech* with modular graphical description techniques, e.g. HyCharts, in comprehensive tool frameworks, such as accomplished with AutoFocus for discrete systems.

Note that the development process for hybrid system proposed in [9] can be regarded as an intermediary between the two processes outlined here. There, the authors propose to complement block diagrams and automata-based notations with formal specifications using Z [34].

4 AutoFocus

AutoFocus [19, 20] is a tool for graphically specifying embedded systems. It supports different views on the system model: structure, behavior, interaction, and data type view. Each view concentrates on a fixed part of the specified model.

Structural view: SSDs. In AutoFocus, a distributed system is a network of components, possibly connected one to another, and communicating via so-called channels. The partners of all interactions are components which are specified in *System Structure Diagrams* (SSDs). Figure 4 shows a typical SSD. In this static view of the system and its environment, rectangles represent components, and directed lines visualize channels between them. Both are labeled with a name. Channels are typed and directed, and they are connected to components at special entry and exit points, so called *ports*. Ports are visualized by filled and empty circles drawn on the outline (the *interface*) of a component. As SSDs can be hierarchically refined, ports may be connected to the inside of a component. Accordingly, ports which are not related to a component are meant to be part of unspecified components which define the *outside world* and thus the component's interface to its environment. Components can have local variables to store values; these variables can be used to describe the behavior and the interaction of components.

Behavioral view: STDs. The *behavior* of an AutoFocus component is described by a *State Transition Diagram* (STD). Figures 5 and 6 show typical STDs. Initial states are marked with a black dot. An STD consists of a set of *control states*, *transitions* and *local variables*. The set of local variables builds the automaton's *data state*. Hence, the internal state of a component consists of the automaton's control as well as its data state. A transition can be complemented with several annotations: a label, a precondition, input statements, output statements, and a postcondition, separated by colons. The precondition is a boolean expression that can refer to local variables and transition variables. Transition variables are bound by input statements, and their life-cycle is restricted to one execution of the transition. Input statements consist of a channel name followed by a question mark and a pattern. An output statement is a channel name and an expression separated by an exclamation mark. The expression on the output statement can refer to both local and transition variables. A transition can *fire* if the precondition holds and the patterns on the input statements match the values read from the input. After execution of the transition the values in the output statements are copied to the appropriate ports and the local variables are set according to the postcondition. Actually the postcondition consists of a set of actions that assign new values to local variables, i.e., the assignments set the automaton's new data state.

Communication semantics. AutoFocus components have a common global clock, i.e., they all perform their computations simultaneously. The cycle of a composed system consists of two steps: First each component reads the values on its input ports and computes new values for local variables and output ports. After the clock tick, the new values are copied to the output ports where they can be accessed immediately via the input ports of connected components and the cycle is repeated. This results in a *time-synchronous* communication scheme with buffer size 1. Values on the output ports are copied over the channels to the appropriate input ports and the cycle is repeated. This results in a non blocking synchronous communication.

Interaction view: MSCs. Message Sequence Charts (MSCs) are used to describe the interaction of components. In contrast to Message Sequence Charts as defined in [23], AutoFocus MSCs refer to time-synchronous systems. In the following, the term MSC always denotes these time-synchronous sequence charts. Progress of time is explicitly modeled by ticks which are represented by dashed lines. All actions between two successive ticks are considered to occur simultaneously, i.e., the order of these actions is meaningless. An action in an MSC describes a message that is sent via a channel from one component to another.

MSCs can be used to describe requirements, simulation traces, counter examples (from model checking), and test sequences. Figures 7 and 8 show a typical test case specification as well as a satisfying test case that satisfies it.

Datatype view: DTDs. For the specification of user defined data types and functions AutoFocus provides Data Type Definitions (DTDs). Definitions in DTDs are written in a functional style. For hybrid systems functions with continuous ranges can be defined, for instance:

```
const GMars = 3.73;
fun speed(last:Float,dt:Float)
        = last+GMars*dt.
```

Features of AutoFocus. In addition to its modeling capabilities, AutoFocus allows for checking consistency between views as well as simulating models (using OCL). For the German BSI (Federal Agency for Security in Information Technology) several validation techniques have been integrated into AutoFocus [33]. The result is a model validation framework that supports

- model checking and bounded model checking to check temporal properties (invariants) [31],

- abstraction techniques to *safely* reduce complex models to simpler ones,
- interactive theorem proving techniques to verify arbitrary security requirements, and
- systematic generation of test sequences and test cases [39, 26, 27].

All these validation techniques are based on the simple and intuitive semantics of AutoFocus [21], and can be used to support model-based development steps. This framework also allows to generate Java and C code from the validated models in order to get executable implementations.

5 The Mars Lander

This section describes our example, a spaceship similar to the doomed Mars Polar Lander that allegedly did not complete its mission due to a faulty design [8]. The discretization technique we apply is common in the design of discrete-time control systems [30]. The next section then shows how automated test generation could have helped in avoiding this problem.

Behavior. The hybrid system and its environment may be described by four main distinct states: The lander may be orbiting (`orbiting`) or falling freely after leaving its orbit (`Rockets Off`). Furthermore, the lander may have ignited its retro rockets (`Rockets On`) in order to slow down its vertical movement, and ant it may have landed (`Landed`). While orbiting, the lander's vertical speed, $v_\ell(t)$, is

$$v_\ell(t) \approx 0.0 \qquad (1)$$

$$v_\ell(t) - v_\ell(t_0) \approx \int_{t_0}^{t} g_{mars}\,\mathrm{d}t = g_{mars} \cdot (t - t_0) \qquad (2)$$

$$v_\ell(t) \approx \int_{t_0}^{t} \big(g_{mars} - \frac{\dot{m}_\ell}{m_\ell(t)} \cdot v_f\big)\mathrm{d}t + v_\ell(t_0) \qquad (3)$$

$$\dot{m}_\ell \approx \int_{t_0}^{t} (c_1 \cdot (v_\ell(t) - v_{req}) + c_2 \cdot \dot{v}_\ell)\mathrm{d}t \qquad (4)$$

Figure 2: Mars Lander orbiting/landed (1), falling without (2) and with (3,4) rockets.

zero (Fig. 2-1). This behavior is also exhibited when the lander has landed. Once it has left its orbit, we assume it is freely falling without friction. Its behavior can thus be described by Fig. 2-2 where the planet's gravity, g_{mars}, is assumed to be constant. Strongly simplifying, the system's dynamic behavior in state `RocketsOn` may be modeled as follows. Let $F_w(t) \approx g_{mars} \cdot m_\ell(t)$ be the lander's weight force with $m_\ell(t)$ being the lander's mass. Furthermore, let $F_{thr}(t) = \dot{m}_\ell \cdot v_f$ be the rocket's thrust force where $\dot{m}_\ell$ is the lander's (negative) change of mass and v_f the (negative, approximately constant) rocket's exhaust speed. By ignoring friction effects and letting $h(t)$ denote the lander's height one derives $\ddot{h} \approx g_{mars} - \frac{\dot{m}_\ell}{m_\ell(t)} \cdot v_f$ and thus the lander's vertical speed (Fig. 2-3) from $F(t) \approx m_\ell(t) \cdot \ddot{h} = F_w(t) - F_{thr}(t)$. For simplicity's sake, horizontal forces have been ignored. Oversimplifying again, we furthermore assume a simple PD controller for the lander, modeled by $\ddot{m}_\ell \approx c_1 \cdot (v_\ell(t) - v_{req}) + c_2 \cdot \dot{v}_\ell$ for adequate gains c_1, c_2 and the lander's required speed, v_{req}. This yields the system's second descriptive equation (Fig. 2-4) for this state. The control variable is thus $\dot{m}_\ell$ (or $\ddot{m}_\ell$, respectively) which reflects an increase or decrease in fuel to be burnt.

In order to model the system with AutoFocus, the above equations have to be discretized (i.e., linearized). Being the results of two AutoFocus simulations, the curves in Fig. 3 have been obtained after discretization with step size $\Delta t = .01$. It shows height and velocity for two behaviors of the spaceship. After a certain time, it is caused to start its landing procedure by leaving the orbit (event "enter"; events are symbolized by long vertical arrows). When a maximum speed (45 m/s) is reached, the rockets are ignited ("rocketsOn"). Some time later, the legs are caused to open. From now on, the behaviors differ. The intended behavior is that when the spaceship actually lands, it should turn off its rockets (trajectories with annotation "rockets on"). *Ground contact is inferred from a shock in the legs.*

If, on the other hand, opening the legs causes the rockets to be switched off, velocity immediately increases which results in a crash (trajectories annotated with "rockets off"). This is what allegedly happened to the real spaceship: Opening and adjusting the legs caused some sensors in the lander to believe the spaceship had landed *for the legs sensed a shock.* (According to [8], engineers were well aware of this problem. When testing the system, they encountered a wiring problem, fixed it, and did not re-run their tests. Nonetheless, we will use this example as a motivation for a semi-automated generation of test cases in Sec. 6.)

Structure. The above equations show that two main variables are involved, namely the change of mass, $\dot{m}_\ell$, and the lander's vertical speed, $v_\ell(t)$. This motivates the systems top level structure as described by the SSD in Fig. 4 that consists of three components: a `Lander`, a `Physics`, and a `Controller` component. All components receive the current time via channel `T` from the environment. The value of `T` is assumed to be present throughout every time slice, and to be increased by a constant value. The controller sends control commands to the other components, in order to switch the lander's rockets on and off, to enter the landing phase, and to open the legs for landing. It has a local variable `CState` to

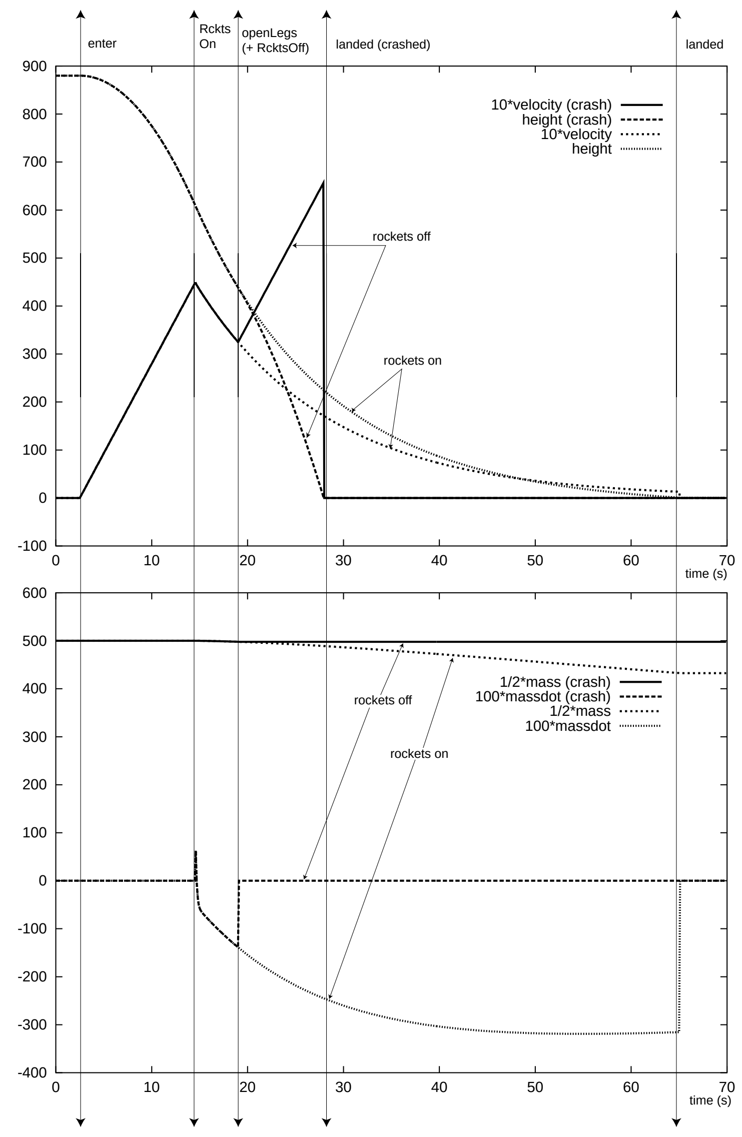


Figure 3: Spaceship crashes and lands.

record the spaceship's state since it needs to know when to issue which command. The initial value of this variable is `Waiting`; the type of this variable is a DTD `data CState = Waiting | Ron | Roff`. When port `Sensor` receives `True`, this should (!) indicate that the lander has sensed ground contact, and in turn its rockets will be switched off.

The differential equations of Fig. 2-3 and 2-4 are mutually dependent. In order to compute the values independently, the computation has to be separated into two subcomponents, namely `Lander` and `Physics`. The main interaction is between `Lander`, and `Physics`: the environment sends the current velocity to the lander (channel `V`), and the lander, in turn, sends its change in mass to the environment (channel `Mdot`). Channel `Speed` is only necessary for the initial value of the control process. `Lander` and `Physics` could have been grouped together into one hierarchic component. This is advisable if systems become more complex. The behavior of component `Physics` is separated into two states (see Fig. 5). State `Control Off` just outputs the current speed (and does not react to changes of mass), whereas in state `Control On`, the new speed is computed from the last speed and the actual change in mass (Eq. 3). Component `Physics` has several local variables to store past values, used for integration, and differentiation, `LastT:Float`, for instance, to compute time differences. The denotation of the transition labels in Fig. 5 consists of functional terms computing new values according to the discretized equations. The behavior of component `Lander` is separated into four states (see Fig. 6), each representing a differential equation or the respective computation of new values (mainly for the local variables: `LastT`, `LastV`, `LastM`, `LastH`, and `LegsOut`) and for the output `Mdot`. In the initial state `Orbiting` the lander waits for the command `enter` from the controller (received at port

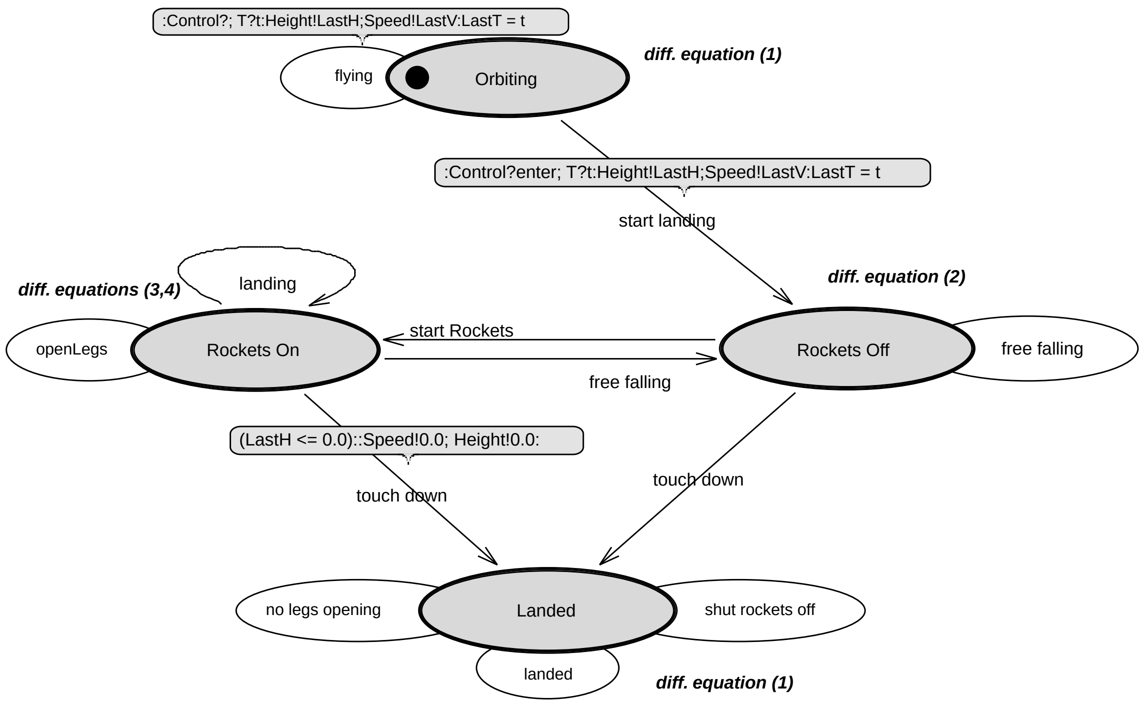


Figure 6: Behavior of Component Lander

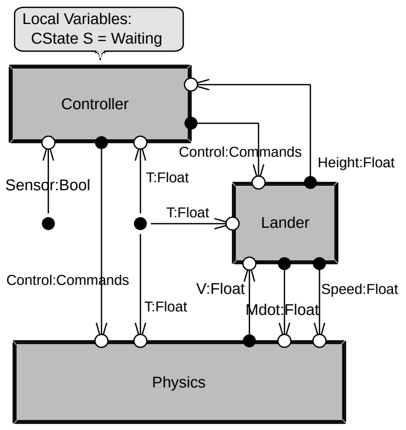


Figure 4: System Structure Diagram

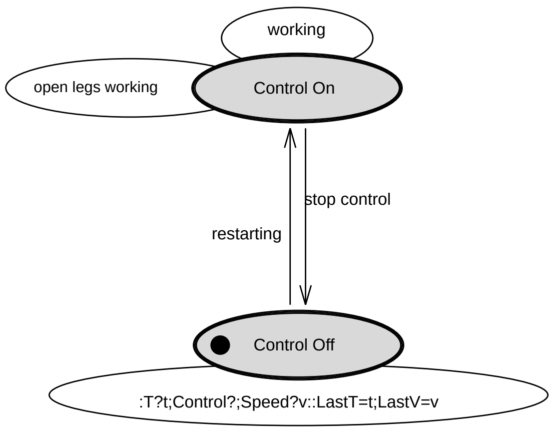


Figure 5: Behavior of Component Physics

Control). Unless no enter command is present, initial values are sent to the environment. This is done within the transition labeled flying and the following semantics:

- input pattern Control? denotes no input on port Control,
- input pattern T?t denotes that the input on port T is bound to transition variable t,
- output patterns Height!LastH, and Speed!LastV denote the sending of the current values of the variables to the output ports,
- action LastT = t stores the value t into the local variable LastT.

This last transition has no precondition (since t>LastT is a general assumption on time).

The states Rockets On and Rockets Off control the lander during the landing phase (with and without boosters); control commands RocketsOn and RocketsOff from port Control can be used to switch between the two respective equations. If the height is less or equal to zero in one of the states, the lander reaches the final state landed.

Shortcomings. Suitably discretizing a continuous model is a difficult problem. We chose a simple piecewise linearization with trapezoidal approximation. Problems with this approach include a conservative determination of Δt as well as meaningful error estimations. Thus far, we use the same Δt for all components (in accordance with user-defined step sizes for each component). This approach may result in efficiency problems, but it solves the problem mentioned below for communicating components integrating over a same variable in the case this variable is t. The automatic generation of discretized systems from continuous equations is subject of ongoing work. Especially methods are developed to break systems of differential equations into single components, to determine appropriate discretization methods, and to to find good timing rates for the components.

6 Testing

Approaches to ensure a system's reliability include validating a model w.r.t. its specification as well as checking an implementation's conformance w.r.t. the specification. Formal methods, such as model checking, allow for determining a system's correctness in terms of user defined properties usually formulated in an (unintuitive) logic, e.g., the Linear time Temporal Logic LTL. Without suitable (and usually hard to determine) abstractions, model checking is restricted to finite state spaces which, for instance, typically grow exponentially with the number of variables involved. Not surprisingly, industrial applicability has not yet been achieved. In the following, we describe how a classical approach to quality assurance, namely testing, is supported by AutoFocus. We advocate an integration of mathematically complete techniques (model checking) with testing. In addition to specifying test cases during the design phase, testing should also be done interactively, for certain errors can only be revealed by "playing around" with the model. This kind of testing may thus be seen as a debugging aid. This is, in fact, the case for most of the spectacular software faults the model checking/theorem proving communities use as a motivation for their work. The discussion of test management strategies and particular techniques such as mutation analysis and fault injection is beyond the scope of this article and thus omitted.

Applicability and Terminology. We distinguish between possibly informal requirements, a specification which is called a *model* if it is written down formally (e.g., in AutoFocus), and an implementation. Testing an implementation is usually done w.r.t. its specification, e.g., [32, 26, 27]; the specification is thus considered to be correct. Obviously, this is a strong and usually unrealistic assumption. However, we think it is *one* necessary step. The techniques sketched below and explained in more detail in [26, 27, 39] allow for the determination of test sequences on the grounds of a test case specification. A test case specification is the formalization of some test purpose, i.e., reach a particular state or cause the system to throw a particular exception. Test case specifications can, for instance, be written down as mathematical formulas [12], formal specifications [37, 5, 32], as MSCs [13, 26, 27, 39], as partial I/O traces or constraints over them [26, 27, 7]. A test case is an artifact that satisfies a given test case specification and may be formulated in the same forms as test case specifications. A test sequence, finally, is an executable test case, e.g., an I/O trace. [27] discusses this terminological framework.

Our work aims at (semi-) automatically deriving test cases from test case specifications that may be used for both, interactively white box testing a specification and (semi-)automatically black box testing an implementation. As indicated above, the interactive part plays an important role in the development process. Even though it is undoubtedly true that a system should be thoroughly thought over before it is implemented or modeled, we believe that simulation and interactive testing help in understanding a model. This is related to the rapid prototyping approach in software engineering; we even see simulation as a specialization of the testing process [26, 27].

However, it is worth emphasizing that most likely none of the commonly used approaches to quality assurance will do it alone. In contrast to formal methods testing is an inherently incomplete process. As formal methods yet do not scale to real size applications, this deficit has to be accepted but borne in mind. Dijkstra's popular remark that testing can only reveal the presence but never the absence of errors also applies to formal methods: One can only check properties that have been formulated by a human. This process, however, obviously is also necessarily incomplete.

Test case specification. The specification of test cases or properties to be checked requires intuitive and, if possible, graphical description techniques. One problem with formal techniques surely lies in the fact that without an intense formal education properties are hard to express in formalisms such as LTL or the Temporal Logic of Actions TLA [25]. We hence advocate the use of a variant of Message Sequence Charts [23] for the specification of test cases [13, 39, 26, 27]. MSCs (HySCs) are augmented with elements for talking about states in condition boxes [15] as well as constructs for expressing iteration and the necessity of certain transitions to fire. The identification of typical test purposes, e.g., causing the system to output certain values, reaching states, executing transition sequences [39], led to the incorporation of these language constructs.

An important concept is that of negation (negating transitions, the reachability of states, or forbidding certain inputs or outputs). However, a suitable semantics for MSCs in the context of test cases seems to be incomplete in the sense that between two elements in an MSC, arbitrarily many others may be present. Apparently the formal definition of a semantics for negation in this context is not obvious [24] and subject of ongoing work.

In AUTOFOCUS, test cases may be specified by both LTL formulas and MSCs. In the following, we focus on the derivation of test cases from system and test case specifications. In the remainder of this section, the system specification should be thought of as an AUTOFOCUS model, and the test case specification is formulated using MSCs. Computed test cases (I/O sequences) are displayed in the form of MSCs themselves for inspection by a human (or comparison with expected test results, i.e., correspondence of the model's output with the output as described in the test case specification). Note that in this paper we concentrate on testing a specification and do not take into account testing implementations even though computed test sequences can be fed into an implementation for conformance testing with the specification.

Testing discrete systems. This paragraph briefly describes the generation of test cases from test case specifications by means of Constraint Logic Programming (CLP) as well as of propositional logic. These methods *automatically* derive test sequences from system and test case specifications.

CLP is the result of integrating two declarative programming paradigms, namely logic and constraint programming. Distinctive features include invertability of functions, the use of free (logical) variables that may be bound during program execution, built-in search mechanisms – backtracking –, and a semantics based not only on terms but rather on arbitrary domains. It turned out that AUTOFOCUS models can very naturally be translated into CLP languages. The idea is to feed the executable model with partial I/O traces and make the test case generation system create actual test cases (possibly partial I/O sequences subject to certain constraints, e.g. ranges for variables) by relying on the above mentioned built-in search mechanism and by using logical variables. By imposing constraints (e.g., in the forms of MSCs) on the set of all possible system execution, the search space can significantly be reduced. Further analyses such as automated interval analyses or (manually derived) classification trees [14] for variables then allow for the determination of meaningful test sequences (taking into account, for instance, range boundaries that yield equivalence classes to be tested). [26, 27] contain a more detailed description of this approach.

Testing based on propositional logic is suitable only for small finite systems (in particular, for systems with small, finite variable ranges). AUTOFOCUS models as well as test case specifications are translated into propositional logic and combined into a single formula which is fed into a propositional solver. The results (binding of free variables in traces) are translated back into MSCs. A detailed description of this approach which is related to bounded model checking [2] can be found in [39].

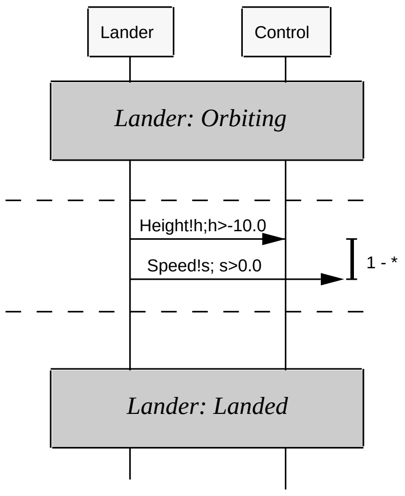


Figure 7: Test case spec.: Reach state *landed.*

Testing hybrid systems. In principle, the above automatic CLP based generation of test sequences is also applicable to mixed discrete-continuous systems, for numerical or algebraic solvers can easily be connected to the CLP system. Yet, assuming that continuous activities take place within particular states of the system and that there is a *continuous* data flow between components, a number of problems arise. First of all, it is not clear how a continuous data flow can be simulated on ordinary computers (in control systems, however, there indeed is a continuous flow of data). Secondly, numerical solvers also discretize differential equations and solve these equations with different, possibly even dynamic, integration step sizes. It is not clear how to handle the situation where one component triggers a transition dependent on, e.g., the global time. Assume that two components run at different speeds, i.e., with different integration step sizes. If one component integrates over a common variable and meanwhile receives a value for exactly this variable that has been determined according to an earlier time, it has to stop its integration process and to step back. This results in severe methodical as well as efficiency problems, both of which are subject of ongoing work, based on (1) the semantics for hybrid systems as defined in [35] and (2) a modification of the AUTOFOCUS semantics where continuous activities do not take place on transitions but rather within states. This applies only to hybrid testing since real time simulation forbids re-calculating certain variable boundaries.

Example: Testing the lander. In accordance with these considerations, so far the methods for deriving test cases described above have only been implemented for discrete systems. As AUTOFOCUS is based on an inherently time-discrete semantics, this paragraph illustrates the derivation of test sequences for the discretized model of the Mars lander. Due

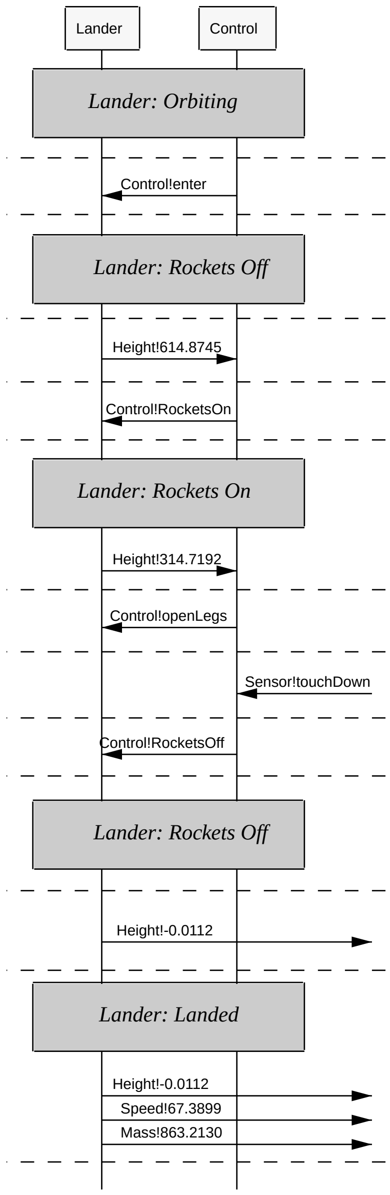


Figure 8: Test case: Lander crashes.

to space limitations, we concentrate on just one test case specification which is, however, sufficient to convey the principal idea. Figure 7 shows the graphical specification for the test case "find a system run that makes component *lander* reach state *landed*". A derived corresponding test case specifying this specification is depicted in Fig. 8. Note the close relationship with the State Transition Diagram of Fig. 5. This system run makes the lander crash for its final velocity when touching the ground is much too high (approximately 67 m/s). Obviously, this is just *one* test case for the given specification. Another successful run leaves the rockets ignited until the spaceship actually has had ground contact. Both possible runs are depicted in Fig. 3 in forms of the respective variables' trajectories. Note that in case of the crash there is no automated means for assessing the outcome of a test case, nor some help in order to detect the fault. Above, we described two possible test scenarios, one of which consisted of test case specifications with verdicts and has been created independently of the modeling process. The second scenario is closer to the area of rapid prototyping, where testing is seen as a debugging aid. In the case of Fig. 8, both scenarios may apply. However, there obviously is need for an engineer who derives from the test sequence that using a shock in the legs as ground detection mechanism is a bad idea!

7 Conclusion

A central aim of our work is the support of a systematic design of correct safety-critical hybrid embedded systems. For discrete systems we think that a number of effective validation and verification techniques has been integrated within the AutoFocus framework. In this paper we presented an example of an ad-hoc discretization of a hybrid system, using discrete formal models. The model allowed an improved validation; in particular, important test scenarios have been derived. Secondly, precise requirements for dealing with hybrid systems in the context of discrete CASE tools have been obtained: (1) systematic discretization support, and (2) extending formal modeling and validation methods with continuous features to hybrid methods. Thirdly, a new development process for hybrid systems has been proposed and discussed.

In the future we will further evaluate how AutoFocus can be applied in the development of safety critical avionic systems and what is necessary to make it compatible with the certification process required for such systems. Furthermore, a testing medhodology (which test cases to choose, how many, etc.) is the subject if future work.

Acknowledgment. We would like to thank Michael van der Beeck for helpful comments on this paper.

References

[1] J. Albert and J. Tomaszunas. Komponentenbasierte Modellbildung und Echtzeitsimulation kontinuierlich-diskreter Prozesse. In *Proc. of VDI/VDE GMA Kongreß Meß- und Automatisierungstechnik*, 1998.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In W. Cleaveland, editor, *Proc. TACAS/ETAPS'99*, LNAI 1249, pages 193–207, 1999.

[3] M. S. Branicky. Stability of switched and hybrid systems. In *Proc. 33rd IEEE Conf. Decision and Control*, 1994.

[4] P. Braun, H. Lötzbeyer, B. Schätz, and O. Slotosch. Consistent integration of formal methods. In *Proc. 6th Intl. Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, 2000.

[5] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Proc. 8th Intl. Conf. on Protocol Specification, Testing, and Verification*, pages 63–74, 1988.

[6] K. Buchenrieder and J. Rozenblit. Codesign: An overview. In *Codesign – Computer-aided HW/SW Engineering*. IEEE Press, 1995.

[7] A. Ciarlini and T. Frühwirth. Using Constraint Logic Programming for Software Validation. In *5th workshop on the German-Brazilian Bilateral Programme for Scientific and Technological Cooperation*, Königswinter, Germany, March 1999.

[8] CNN News. NASA: Premature engine shutdown likely doomed Mars lander. 28.3.00, www.cnn.com/2000/TECH/space/03/28/lander.report.02/.

[9] M. Conrad, M. Weber, and O. Müller. Towards a methodology for the design of hybrid systems in automotive electronics. In *Proc. of ISATA'98*, 1998.

[10] DFG. Priority program KONDISK (analysis und synthesis of continuous-discrete systems). www.ifra.ing.tu-bs.de/kondisk/, 2000.

[11] M. Fuchs, M. Eckrich, O. Müller, J. Philipps, and P. Scholz. Advanced design and validation techniques for electronic control units. In *Proc. of the International Congress of the Society of Automotive Engineers*. SAE International, 1998.

[12] M. Gaudel. Testing can be formal, too. In P. Mosses, M. Nielsen, and M. Schwartzbach, editors, *Proc. Intl. Conf. on Theory and Practice of Software Development (TAPSOFT'95)*, LNCS 915, pages 82–96, Aarhus, Denmark, May 1995.

[13] J. Grabowski. *Test Case Generation and Test Case Specification with Message Sequence Charts*. PhD thesis, Universität Bern, 1994.

[14] M. Grochtmann and K.Grimm. Classification trees for partition testing. *Software Testing, Verification, and Reliability*, 3:63–82, 1993.

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

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

[17] R. Grosu, G. Ştefănescu, and M. Broy. Visual formalisms revisited. In *Proc. International Conference on Application of Concurrency to System Design (CSD'98)*, 1998.

[18] T. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HyTech. In *TACAS 95: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1019. Springer-Verlag, 1995.

[19] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch. Tool supported specification and simulation of distributed systems. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Proc. Intl. Symp. on Software Engineering for Parallel and Distributed Systems*, pages 155–164. IEEE, 1998.

[20] F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic Lights - An AutoFocus Case Study. In *1998 International Conference on Application of Concurrency to System Design*, pages 282–294. IEEE Computer Society, 1998.

[21] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313, pages 122–141. Springer Verlag, 1997.

[22] IABG. Das V-Modell. www.v-modell.iabg.de, (documents also available in English), 2000.

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

[24] I. Krüger. *Using MSCs for design and validation of distributed software components*. PhD thesis, Technische Universität München, 2000.

[25] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

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

[27] H. Lötzbeyer and A. Pretschner. Testing Reactive Systems with Constraint Logic Programming. In *Proc. 2nd workshop on Rule-Based Constraint Reasoning and Programming*, Singapore, September 2000. To appear.

[28] B. Müller. Unterstütung von Entwicklungsschritten auf Objekten mit unterschiedlichen OCL-Konsistenzanforderungen. Master's thesis, Institut für Informatik, TU München, 2000.

[29] O. Müller and T. Stauner. Modelling and verification using linear hybrid automata - a case study. *Mathematical and Computer Modelling of Dynamical Systems*, 6(1):71–89, 2000.

[30] K. Ogata. *Discrete-Time Control Systems*. Prentice Hall, 1987.

[31] J. Philipps and O. Slotosch. The quest for correct systems: Model checking of diagrams and datatypes. In *Proc. IEEE Asian Pacific Software Engineering Conference (APSEC'99)*, pages 449–458, 1999.

[32] S. Sadeghipour. *Testing Cyclic Software Components of Reactive Systems on the Basis of Formal Specifications*. PhD thesis, TU Berlin, 1998.

[33] O. Slotosch. Overview over the project Quest. In *Proc. of FM Trends 98*, LNCS 1641. Springer-Verlag, 1998.

[34] J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[35] T. Stauner and G. Grimm. Prototyping of hybrid systems - from HyCharts to Hybrid Data-Flow Graphs. In *Proc. of WDS'99 (satellite workshop to the 12th International Symposium on Fundamentals of Computation Theory, FCT'99)*, Electronic Notes in Theoretical Computer Science 28. Elsevier Science, 1999.

[36] The MathWorks Inc. MATLAB. www.mathworks.com/products/matlab/, 2000.

[37] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software–Concepts and Tools*, 17(3):103–120, 1996.

[38] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 1998.

[39] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic, December 2000. J. Software Testing, Verification & Reliability (STVR): Special Issue on Specification Based Testing. To appear.