Prototype-Based Tests for Hybrid Reactive Systems*

G. Hahn, J. Philipps
Validas Model Validation AG
gate
Lichtenbergstr. 8, 85748 Garching
Germany

A. Pretschner
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching
Germany

T. Stauner
BMW Car IT
Petuelring 116
80809 München
Germany

Abstract

Model-based testing relies on the use of behavior models to automatically generate sequences of inputs and expected outputs. These sequences can be used as test cases to the end of both validating the model and also verifying an actual system. In the automotive domain many systems are reactive and exhibit continuous as well as discrete dynamics. This leads to an explosion of the model state space, which makes automated test case generation difficult, and, because of imprecisions in the continuous parts, requires an adequate treatment of fuzziness both in the dimensions of time and values. We report on experiments with model-based testing in the automotive domain. Roughly, the idea is to use two separate models, a discrete model as an abstract description of relevant scenarios, and a discrete-continuous model to produce reference outputs for the actual system. As an application example we use a fictitious autonomous cruise control system (ACC). We argue that rapid prototyping approaches fit well with the use of models that serve as specifications, as basis for test case generation, or as basis for production code generation.

1 Introduction

In many areas of software and systems engineering, the use of models enjoys an increasing popularity. In the domain of business information systems, data models as given by ER or class diagrams have served as specifications for many years. Platform independent models that abstract from a concrete communication infrastructure are at the heart of the OMG's model driven architecture. Both kinds of models exhibit the advantage of abstracting from details of a concrete implementation which turns out to be advantageous for (1) they enable systems engineers to intellectually master their artifacts even with regard to an ever increasing complexity, and (2) they allow for generating code or code skeletons which reduces the necessary coding efforts.

One characteristics of a second important domain, that of reactive embedded systems, is that the artifacts in question exhibit a comparably complicated control flow with comparably simple data structures. It is a seductive idea to also abstract from behavior details of a concrete implementation in order to work with artifacts that, due to their nature of being simplifications, are easier to understand and thus better manageable. The problem is that people want behavior models to be both abstract and concrete. Models are required to be abstract so that their intellectual mastery becomes realistic, and they are required to be concrete such that code or test cases can be automatically generated. The latter requires bridging the gap between different levels of abstraction which, quite naturally, turns out to be a difficult problem. Since models are domain- and application-specific, there is no general design methodology for this problem. In the automotive and avionics domains, however, modeling languages and CASE support in tools like Matlab or ASCET-SD, have reached a level of maturity that not only simulation code, but also production code for deployment in actual devices, can be generated. Similarly, behavior models have proved to be useful for fully automatic test case generation, as shown in [3] in the domain of chip cards.

We deem models useful in terms of both the process and the product. Incremental development of models fits well with processes that rely on rapid, or evolutionary, prototyping. The comparatively high description level of models eases interaction with customers, and since there is a considerably smaller body of code, i.e., modeled behavior, changes turn out to be easier to implement. This is discussed in detail in [6]. Models play the role of prototypes, of specifications, and that of mock-ups of the environment in order to make a model of the system under consideration executable. For instance, this applies to models of machines in Simultaneous Engineering when programmable logic con-

*Work in part supported by the DFG (project KONDISK/IMMA) and and the BMBF (project Embedded Quality).

trols are to be developed, or to models of devices when the network master of, say, a multimedia bus in modern vehicles is to be developed and validated. That is to say, in general models are needed for both the system under consideration and its environment.

The focus of this paper is on testing, and we hence do not carefully touch the problem of production code generation. The general idea is that a behavior model provides the reference behavior of an implementation. Roughly, we use the model to generate test cases that, after suitable concretizations, are fed into the system to be tested. Again, models turn out to be out necessary for both the implementation under test and the environment. Models of the environment are needed to restrict the otherwise intractable multitude of its behaviors. For instance, acceleration in a car obeys certain criteria: not all possible changes in speed can actually be exerted by a driver.

Now, it is arduous to check whether or not the system conforms to its specification. The reason is that full conformance cannot be established because of the usually infinite nature of the system's state space. Instead, one needs to approximate this conformance. This means that a reasonably small set of finite test traces must be selected that increases confidence in the system's correctness. Unfortunately, there is no commonly accepted notion for what constitutes a "good" test case. In turn, this means that for a given problem, test engineers have to rely on their intuition and experience to build test suites of sufficient quality (the metrics for which, again, remains vague and implicit). Hence, this process is bound to the ingenuity of single test engineers, it is often irreproducible and not systematic.

Testers quickly discovered that coverage criteria are one means to define the quality of a test suite. It also became clear that these criteria could also serve as test case specifications. While coverage-based specifications are not adequate in themselves, they turned out to be a useful complement to functional test case specifications.

As it turns out, the generation of test suites that satisfy a given coverage criterion reduces to the problem of finding elements of the system's state space (e.g., program counters have to reach each possible statement, or each control state in a state machine is to be reached). While in general this is difficult for general programming languages, it is much easier for restricted modeling languages. In the automotive and avionics domain embedded systems often exhibit a behavior that is both (event) discrete and continuous, i.e., hybrid. Because of the high time resolution and the continuous values in hybrid systems, the search space (and the test sequences) are much larger and automatic search becomes intractable.

In this paper, we report on a method for generating test cases from mixed discrete-continuous models specified in Matlab Simulink/Stateflow.[1] Roughly, the idea is to use two models for test case generation. A discrete model describes common usage scenarios and control phases of the system; it is on this model that test suites satisfying coverage criteria are generated. After (application-specific) concretization of these test sequences, they are fed into a mixed discrete-continuous model to obtain reference outputs. Test execution then feeds the concretized inputs in the actual system, and compares the system's response with the reference outputs. Of course, this step must allow for some tolerance both of the values and of the time points these values are observed.

Contribution. This paper presents an approach to generating test cases—trajectories—for hybrid systems in a structured and automatic manner. These test cases can be used both for validating models and verifying the respective systems. The ideas are discussed along the lines of a case study, an automatic cruise control. We are not aware of any published work that explicitly targets at generating test cases for mixed discrete-continuous systems and that does not rely on pure time discretizations of the overall system.

Outline. The remainder of the paper is organized as follows. In Sec. 2 the case study, an automatic cruise control, is described together with its model. Sec. 3 develops the general approach for testing of mixed discrete-continuous systems. Sec. 4 describes test sequences for the ACC, and Sec. 5 concludes. Related work is cited in its context.

2 Automatic Cruise Control

In this section we briefly explain the automatic cruise control system (ACC) and our model of it. The model is designed as a typical evaluation system for tests of mixed discrete-continuous systems. The ACC is a driver assistance system that controls a car's speed and the distance to the car in front (if any). Thus, it extends classic cruise control systems by also considering distance and not only speed.

The main requirements of the system are (1) to adjust the car's speed to the desired speed *Vset*, as set by the driver, if there is no slower car in front, and (2) to adjust the distance between the car and a preceding car which is going with a speed less than *Vset* to the set value for the distance, *Dset*. These adjustments have to made in a manner which is comfortable for the car's occupants. In particular this means that sudden, strong accelerations and decelerations must be avoided. This comfort requirement is highly important. It motivates that test cases which do not consider continuous dynamics do not suffice for testing the system. This is because they can hardly reflect the magnitude of the car's ac-

[1] Stateflow is the state machine tool in the MATLAB/Simulink product family [9]. MATLAB/Simulink is widely used in industry for control system design.

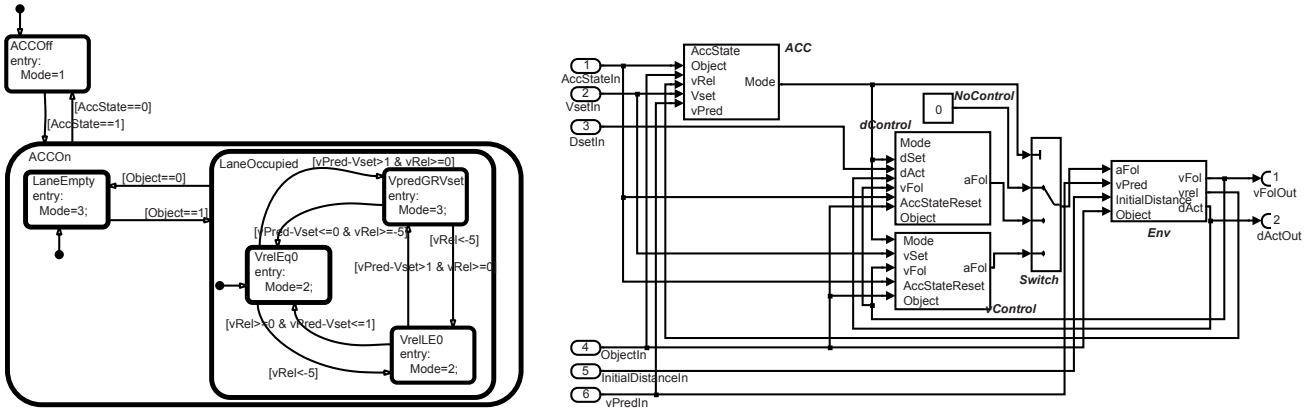

Figure 1. Models of the control logic and the controller with the physical dynamics

celeration and its changes. At least in part, the satisfaction of this requirement must be verified by real drivers.

The left side of Fig. 1 depicts the main control logic for the ACC system as a Stateflow Statechart. Since the system is activated and deactivated by the driver, the control logic has two top-level states *ACCOff* and *ACCOn*. Switching between these states is controlled by input variable *AccState*. State *ACCOn* is refined into the substates *LaneEmpty* and *LaneOccupied*. If there is another car (which we also refer to as the *predecessor* in the following) in the own car's driving lane, control is in *LaneOccupied*. Otherwise, it is in *LaneEmpty*. The presence/absence of a predecessor is signaled by input *Object*. If the lane is occupied the control logic distinguishes between three situations. Either (1) the predecessor (speed *vPred*) is faster than the own desired speed *Vset* (state *VpredGRVset*), or (2) the predecessor is much slower than the follower with current speed *vFol* (state *VrelLE0*; *vRel* in the diagram denotes the current relative speed $vPred - vFol$), or (3) the current relative speed is close to 0 or the predecessor is faster than the follower but still slower than *Vset* (state *VrelEq0*).

In states *LaneEmpty* and *VpredGRVset*, the control logic implements a control law for speed control. In states *VrelEq0* and *VrelLE0*, distance control is used. This is signaled by the output variable *Mode* which is used by the underlying Simulink model containing the control laws. In state *ACCOff* the system does not influence the cars lateral dynamics. The distinction between states *VrelEq0* and *VrelLE0* allows us to apply a faster, less comfortable control law in emergency situations with a new slow predecessor in front (not currently used). Note that switching between different control laws is typical for mixed discrete-continuous systems. The right side of Fig. 1 depicts the Simulink diagram containing the control logic (top left), the control laws for distance and speed control (middle) and a model of the physical dynamics (right).

3 Hybrid System Tests

Discrete systems. Before we turn to mixed discrete-continuous systems, let us briefly look at model-based test case generation for purely discrete systems. The general approach is sketched in Fig. 2; see [5] for details.

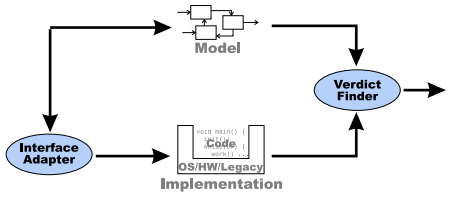

Figure 2. Testing for discrete systems

Assume that we are given a system to be tested (the "implementation"), and a model of those parts of the system behavior that we are interested in. We can then use the model to find test sequences that satisfy formal test specifications or coverage criteria over the model description. As described in [4], this step can be regarded as a search problem in the computation tree of the model. The input part of the test cases is fed into a model of the system in order to produce reference outputs. Then, during test execution itself, the input part is adapted to the interface of the implementation. The adapted inputs are fed into the implementation, and by comparing model and implementation outputs, a test verdict is formed. Obviously, the verdict finder must also bridge the difference in interfaces and interface abstraction levels between model and implementation.

Open-loop systems. The first idea that comes to mind when considering test cases for mixed discrete-continuous systems is that of using a time discretization (roughly, substituting differential by difference equations) and applying the procedure outlined above. This approach is pursued in [7, 1]. It quickly turns out, however, that the test sequences are too long and the search space is too large for systematic exploration, a result both of the small steps (milliseconds) in the dimension of time and of the continuous values. Discrete-event abstractions might appear as the natural solution to the problem. These abstractions are too coarse, however, to be used for generating test cases that are applied to an actual system.

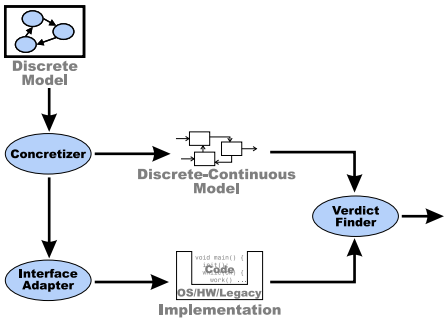

Figure 3. Testing: Open-loop Hybrid Systems

For open-loop-controlled systems, we suggest a different approach (Fig. 3). Instead of a single model of the system, which also serves for test case generation, we assume that we are given both a mixed discrete-continuous model of the system and a purely discrete abstraction of the model, which is tailored for common usage scenarios or control phases. Furthermore, we assume that a test suite has been generated for the discrete model. This test suite might cover all transitions of the discrete model, or all pairs of transitions, or it might be based on a completely different coverage criterion.

Since the discrete abstraction was chosen with some abstraction criterion in mind, it is possible to choose a concretization mapping that is dual to this abstraction. Here, the degrees of freedom include the duration of what has been abstracted by one single signal and the signal's evolution in this time slot.

In general, the output cannot be concretized in the same manner because the relationship between input and output cannot be reconstructed. This is because the abstraction in the discrete model usually is too coarse. We can, however, use the concretized input and feed it into the continuous model to obtain reference output sequences. These concretized inputs and the generated outputs can then be used just as test sequences for the purely discrete situation described above. Of course, inputs still have to be further adapted for the implementation, and the verdict finder must allow small derivations for the output values; we also need to allow small derivations for the times of the discrete mode changes.

Closed-loop systems. As experience shows, the situation is more complicated for closed-loop systems, where we consider not only the controller, but also the environment (the plant). Environment models are desirable in order to reduce the complexity of the discrete model, the concretizer and the verdict finder, and thus to indirectly also reduce the search space for test cases. For instance, the ACC introduced in Sec. 2 bases its decision partly on the current vehicle speed; it influences this speed indirectly through vehicle acceleration or deceleration. The dependency between acceleration and speed is trivial, but its exclusion by considering only open-loop systems would immensely increase the search space, thus requiring more elaborate discrete models to restrict the search.

However, now test case generation requires a feedback construction as shown in Fig. 4, in a way that the discrete model enforces a new control law only after the mixed model has reached a certain state. This state information is abstracted from the mixed model outputs. Typically, the abstractions used are simple partitions of the output value space.

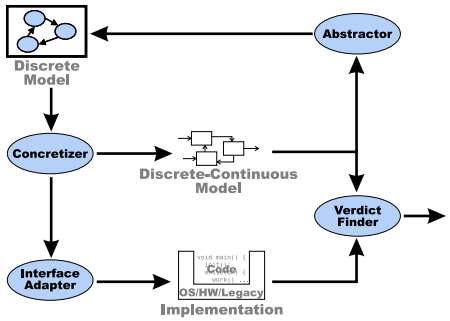

Figure 4. Closed-loop hybrid systems

Test case generation. For the test scenarios shown in Figs. 3 and 4, model-based test case generation techniques can be used to generate sets of input traces that satisfy coverage metrics over the discrete model.

For instance, if transition coverage is to be achieved, then a heuristics could be implemented as follows: for each discrete state (mode), it is recorded which transitions have already been taken. If a particular state is reentered, then a transition is chosen that has not been chosen before. If all transitions have been chosen before, then one can compute the transition that is most likely to lead to a transition that

has not been taken before. This involves the definition of proximity metrics on the state space, or fitness functions that compute the "distance" to all the transitions that have not fired before. The transition that is "closest" to one that did not fire before is chosen [4].

Test case execution. In the three situations mentioned above, we only hinted at the critical steps of the adaption of the test case inputs for the implementation and the comparison of model and implementation output (verdicts).

Input adaption is comparatively straightforward, but it is highly dependent on the used test bed. The comparison of model and implementation outputs is less trivial, however. Obviously there will always be some mismatch between both the values and the timing of the two outputs, since the implementation will suffer from some effects (e.g. friction), which can only roughly be described in the model. The solution here is to add "tolerance tubes" around the model output, and to accept an implementation output if its time/value combination falls within the tube, as shown in Fig. 5. Note that the tubes define a tolerance both for value and for time (note that the tubes extend beyond the boundaries of the different phases). While this approach is conceptually simple, the definition of suitable tubes is surprisingly intricate, it is described in more detail in [2, 8].

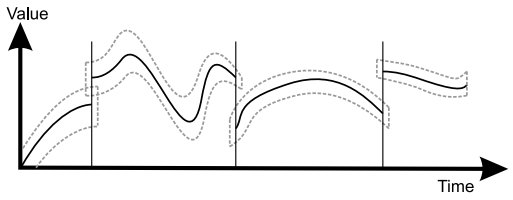

Figure 5. Tolerance tubes around a reference signal

Of course, there are some subtle deviations which are not tolerated by the tube construction—although they arguably should be. These deviations occur when the implementation roughly conforms to the mixed model, but is consistently faster (or slower) than the model. In this case, corresponding time/value points will fall outside the tolerance tube. We assume that such deviations should be handled in the model, and our experiments have not led us to believe otherwise. It is possible, of course, to augment the approach by retiming relations, which allow a certain, bounded, speed-up or slow-down of the implementation.

One may wonder whether the mixed model is indeed necessary—certainly the implementation could be directly driven by the discrete model. The discrete model, however, is too abstract. In particular, even if it were extended to produce reference outputs, finding a verdict would be much more difficult; a suitable verdict finder would have to contain part of the mixed model, which only shifts the problem.

4 ACC Test Cases

A natural abstraction of the ACC that reflects the qualitative states of the overall system, which includes ACC control logic, control laws, lateral dynamics and the behavior of predecessor cars, is as follows. In the overall system we have the qualitative states *Off*, where the ACC is switched off, and *OnEmpty* with the ACC switched on and the driving lane empty and some further states when the lane is occupied and the ACC is on. These further states reflect the values of the predecessor's speed, the follower's speed and the set value for the follower's speed relative to each other. Not all combinations of these speeds result in qualitatively different states. For instance, speed control in the ACC system is active if there is no car in front regardless of whether *vFol* is less than or greater than *Vset*. The interesting combinations are: $vPred \leq Vset + 1\frac{m}{s} \wedge vPred < vFol - 5\frac{m}{s}$ (state *PredMin*), $vFol - 5\frac{m}{s} \leq vPred \leq Vset + 1\frac{m}{s}$ (state *FolPredSet*), $Vset + 1\frac{m}{s} < vPred < vFol - 5\frac{m}{s}$ (state *SetPredFol*) and $Vset + 1\frac{m}{s} < vPred \wedge vFol - 5\frac{m}{s} \leq vPred$ (state *PredMax*).

These combinations are relevant, because they correspond to transition guards in the ACC control logic (Fig. 1, left side) which enforce that states *VrelLE0* and *VpredGRVset*, respectively, are entered. State *VrelEq0* is more or less "in between" those other two states when the values for the velocities are considered. The abstract states *SetPredFol* and *PredMin* can be unified to a single state, *PredFol*, since whenever one of the two predicates is true, the concrete ACC logic is in state *VrelLE0* without distinguishing further.

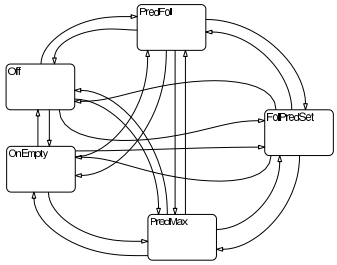

Figure 6. State machine of the discrete model

The state machine for the discrete abstract driver for test case generation is shown in Fig. 6. It consists of these five states (with the unified one). They are strongly connected by transitions that reflect the corresponding conditions which have to be satisfied when the state is entered. As described above, to the end of test case generation, we

can compute (1) sequences that cover all (pairs of) states, and (2) sequences that cover all (pairs of) transitions.

Abstraction and concretization. The abstraction mapping from the detailed discrete-continuous model to the abstract driver is simple. All the above states correspond to predicates over the overall system's state space. The abstraction mapping evaluates these predicates and makes the result available to the state machine of the abstract driver whose transitions are triggered by them.

The concretization is a lot more difficult, since there are whole ranges of legal values for the speeds in the states. Velocities can even change with the qualitative state remaining the same. For instance, in state *FolPredSet*, *vPred* can increase until *Vset* $+ 1\frac{m}{s}$ without a change in the qualitative state. In our case we used the following more or less arbitrary concretization: Since the main application area for the ACC system is highway traffic, we focus on ranges for the speeds between 30 and $45\frac{m}{s}$. Furthermore, accelerations and decelerations are limited by physics. We therefore consider accelerations/decelerations up to $\pm 3\frac{m}{s^2}$. With these limitations one sensible way of concretization is to randomly select linear trajectories for the velocities which are within these bounds. Operationally, this means that when entering a state, we use heuristics to determine which state or transition is desired to be visited/executed next. Based on the corresponding transition guard we can randomly select values which make the guards true for those continuous variables which are input to the concrete model. Based on the allowed accelerations/decelerations we can furthermore determine when the selected values can and should be reached. Then we linearly interpolate between the present values and the desired future values and provide the resulting trajectory as input for the concrete model.

In the ACC model *vFol* need not be concretized, because it is an output of the discrete-continuous system and input to the abstract driver. For the events of switching the system on and off and newly occurring predecessors a stochastic is used. A further stochastic model gives concrete values for the initial distance in which a predecessor appears in the driving lane.

Example test sequence. Fig. 7 shows an output of the Matlab model for a test case derived using the ideas described above. We describe a test case that was generated according to a specification that required all five states to be covered. Variable *dAct* denotes the actual distance between predecessor and follower, *vFol* and *vPred* are as above. The vertical lines in the plots define segments of $50s$, the vertical lines in the plot for *dAct* define segments of $50m$ and those in the plots for *vFol* and *vPred* define segments of $5\frac{m}{s}$. In the test case, the ACC is first switched off. At time $t = 10s$ the system is switched on and the car accelerates to *Vset* $= 30\frac{m}{s}$. After approximately $80s$ from start the set value is increased to $35\frac{m}{s}$ and the car accelerates further. Between time $125s$ and $160s$ the ACC is switched off and on again. This is not visible in the trajectory for *vFol* because the used model of the cars lateral dynamics does not include loss of energy by aerodynamics and friction. After approximately $160s$ a new car with *vPred* $= 33\frac{m}{s}$ appears in the driving lane at an initial distance of $100m$. This causes the ACC control logic to change to state *VrelEq0* and activate distance control. Thus the car first accelerates to decrease the distance until time $165s$ and then decelerates in order to obtain the desired distance.[2] Before this distance is reached the predecessor starts to steadily accelerate to a speed higher than *Vset*, namely to $40\frac{m}{s}$. This causes the ACC control logic to switch back to speed control after approximately $250s$. After $300s$ the predecessor disappears which is not visible in the trajectory for *vFol*, because the follower still pursues speed control. The next event occurs at about $340s$. A new predecessor appears in a distance of $30m$ with *vPred* being $6\frac{m}{s}$ smaller then *vFol* at that point in time. Due to the much smaller speed, the ACC control logic enters state *VrelLE0* and the follower decelerates. Some time later state *VrelEq0* is entered and the desired distance of $50m$ is obtained.

5 Conclusions

Short development cycles and frequently changing requirements render development of embedded automotive systems a particularly demanding area of software and systems engineering. It is our firm belief that the use of models in conjunction with processes that are based on rapid, or evolutionary, prototyping is a promising approach to coping with the intricacies of such processes. As long as sufficiently efficient production code generators—that, at the same time, do not put too many restrictions on the use of the respective modeling language—do not exist, we consider the following scenario as appalling. In close interaction with all involved parties, behavior models of a system under development are incrementally built (cf. the coarse treatment of this issue in Sec. 1 w.r.t. the necessity of behavior models of the environment). The resulting model not only serves as specification, but also as the basis for test cases. Serving as specifications, behavior models in the form of state machines are advantageous when compared to scenario-based formalisms such as Message Sequence Charts because they encode *all* behaviors. Since car manufacturers and manufactures of electronic control units often are distinct parties, the further clearly have an interest in checking whether or not the latter have built an ECU that

[2] A real ACC system would have more control states and would thereby be able to realize that acceleration is undesirable in this situation. For the purpose of this paper, however, we use the simple example system.

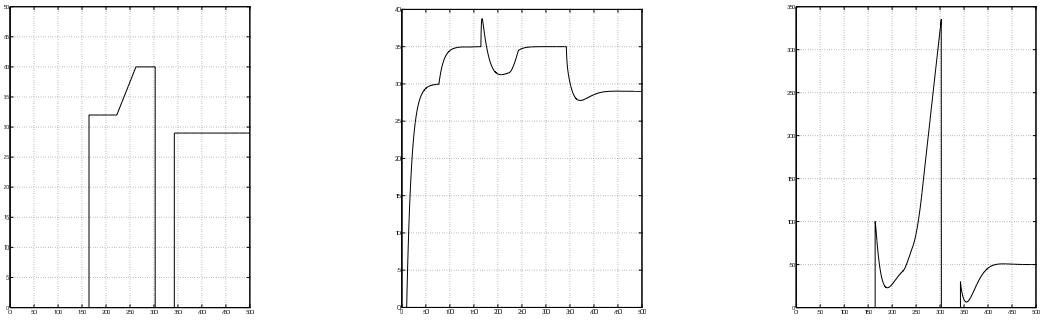

Figure 7. Trajectories of *vPred*, *vFol*, *dAct* (input to the concrete model)

conforms to its specification. It is the process of checking this conformance that we address in this paper, and we approximate a full conformance check by means of automatically generated test cases. Generating test cases for purely discrete systems is a difficult undertaking, and the sheer size of the state space makes mixed discrete-continuous systems even more demanding w.r.t. quality assurance.

The main benefit of the method we reported on is to have a systematic and highly automatic means for deriving test for mixed discrete-continuous systems. If test cases are sought manually the danger to forget an important case is high. In particular we encountered that it is indeed very likely to forget to test one of the qualitative states of the system. Possibly this is due to focusing too strongly on the continuous aspects—the details—while forgetting about the discrete states—the big picture—of the system.

Test case generation is based on a two-tiered modeling approach: A mixed discrete-continuous model serves as the reference for verdicts, while a purely discrete model describes usage scenarios and serves as a source for relevant test sequences. In this paper, we demonstrated our approach with Matlab models. The approach itself, however, is independent of the modeling language. The essence is the notion of "model-in-the-loop" simulation in order to resolve the nondeterminism of the environment of the system.

Our current work focuses on finding verdicts when the physical system is triggered with the generated sequences, in particular on finding suitable "tolerance tubes" around the reference sequences. If the tolerance is too high, inacceptable system behavior might go undetected; if the tolerance is too low, small perturbations in the environment will lead to false test failures. While our approach is usable as described in this paper, solving the tolerance problem is essential to making our approach cost-effective: The main cost occurs in the construction of the mixed model. Reuse of model building blocks will reduce these costs, but the risk of behavior mismatches between the model and the implementation system is higher than for a custom-built model.

References

[1] A. Ciarlini and T. Frühwirth. Automatic derivation of meaningful experiments for hybrid systems. In *Proc. ACM SIGSIM Conf. on Artificial Intelligence, Simulation, and Planning (AIS'00)*, Tucson, AZ, March 2000.

[2] V. Gupta, T. Henzinger, and R. Jagadeesan. Robust timed automata. In O. Maler, editor, *HART 97: Hybrid and Real-Time Systems, LNCS 1201*, pages 331–345. 1997.

[3] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model-based test case generation for smart cards. Submitted to FMICS'03, 2003.

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

[5] 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.

[6] A. Pretschner, H. Lötzbeyer, and J. Philipps. Model Based Testing in Incremental System Development. *The Journal of Systems and Software*, 2003. To appear.

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

[8] T. Stauner. *Systematic Development of Hybrid Systems*. PhD thesis, Technische Universität München, 2001.

[9] The MathWorks Inc. MATLAB Product Family. http://www.mathworks.com/products/ matlab/, 2000.