

# Hybrid System Verification Is Not a Sinecure

## The Electronic Throttle Control Case Study

Ansgar Fehnker and Bruce H. Krogh

Dept. of Electrical and Computer Engineering,  
Carnegie Mellon University,  
Pittsburgh PA 15213, USA  
{ansgar, krogh}@ece.cmu.edu

**Abstract.** Though model checking itself is a fully automated process, verifying correctness of a hybrid system design using model checking is not. This paper describes the necessary steps, and choices to be made, to go from an informal description of the problem to the final verification result for a formal model and requirement. It uses an automotive control system for illustration.

## 1 Introduction

Hybrid systems are characterized by a non-trivial interaction between discrete and continuous subsystems. A typical setting is a digital controller in an analog environment. This interaction makes formal verification of hybrid systems not just tedious, but intrinsically difficult. In recent years the field of hybrid systems has seen significant advances. The hybrid automaton model has been widely adopted as standard for describing hybrid systems [1], and model checking has been shown to be decidable for important classes of hybrid systems and undecidable in general [11,17]. This research has resulted in a number of tools for model checking of hybrid systems such as Hytech [9], Verishift [16], d/dt [5] and CheckMate [3].

These tools and other techniques have been applied to a number of case studies in the domain of automotive control, robotics, avionics or process control. Examples can be found in the proceedings of the Workshop *Hybrid Systems: Computation and Control* (HSCC) [2] and in the proceedings of its predecessors. Despite successful applications of verification tools, it has been questioned if these techniques scale to *real life problems*, i.e. problems with a complexity that can be encountered in industry. The DARPA project Model Based Integration of Embedded Software (MoBIES) includes Open Experimental Platforms (OEPs) to assess the limits of current technology for hybrid systems verification. This paper considers the Electronic Throttle Control (ETC) problem of the automotive OEP. Given a Simulink/Stateflow model and an informal description of system and requirements, the task is to take the model, translate it if necessary, and to verify the system requirements.

This paper uses the ETC case study to illustrate the process that leads from the informal specification to verification. In [13] Henzinger et al. review several case studies performed with the tool HyTech. It presents criteria to decide when model checking with HyTech is promising, discusses shortcomings of HyTech, and future directions for

tool development. The authors conclude, for example, that the number of continuous variables is a limiting factor. In [19] Rushby describes the use of verification in the design process of critical systems, provides an introduction to different formal methods and their main concepts, and identifies steps in the design process where formal methods could contribute to the quality of the design. In contrast with this related work we assume the known limitations of hybrid verification, and assume that the decision to use formal verification has been made.

## 2 From Simulation to Verification

The ideal situation for hybrid verification would be to start from a formal model that either has the desired input format or can be translated automatically into it. In addition, the complexity of the verification model should be within the capabilities of the model checker. The property to be checked should be given as a proper temporal logic formula, or whatever formalism required. Given these preconditions verification would indeed be a *push button* technology.

It is, however, more likely that the starting point for formal verification is an informal system description. The description may be accompanied by a simulation model or implemented code. This information is then used to build the verification model manually. A first step in this process is to understand the mathematical relationships that govern the system. Examining the simulation model and studying several simulation runs can be very useful in this process.

A mathematical model alone is not sufficient. Model checking makes only sense if there are properties to be checked. As with modelling, there is often more to deriving verification requirements than simply translating given informal requirements. Some requirements might be implementation details, such as the target platform to be used. For other requirements it may be sufficient to simulate the model to check it. Only few requirements need to be verified formally.

Given the requirements and mathematical system model one can start building a verification model. Having both, properties and mathematical model, is important for determining what aspect of the system has to be included in the verification model.

A necessary next step, to deal with complexity, might be to divide the verification problem into sub-problems. A well known approach is assume-guarantee reasoning that introduces a number of tractable verification problems, that, when verified individually, imply the correctness of the requirement [10].

A last step in this verification process is to set up the model checking algorithm. This might include choosing a proper size for a hash table, defining a proper exploration order, or, since many hybrid system tools rely on numerical routines, to choose suitable numeric tolerances.

To verify the ETC requirements we will take the following steps:

1. Discover the mathematical model
2. Obtain the formal requirements
3. Build a verification model
4. Set up the verification problem
5. Set up the model checking algorithm

These steps will be part of any verification. In practice there is often not clear distinction between the separate steps, and steps 3 to 5 may be iterated a few times.

The next section describes the hybrid systems model checker CheckMate. The remaining sections will then discuss each of the above steps in more detail, and use the ETC case study for illustration.

### 3 A Brief Introduction to CheckMate

CheckMate is a model-checker for polyhedral invariant hybrid automata (PIHA) [3], a slightly restricted class of hybrid automata. As hybrid automata, PIHAs have a finite number of control locations. In each location a set of differential equations governs the continuous evolution of the continuous state variables. Locations are switched as soon as switching conditions become true. These switching conditions are defined as conjunctions of linear inequalities. Transitions can also reset the continuous state vector by applying an affine mapping .

The model-checking algorithm of Checkmate partitions the state space, and over-approximates the transition relation using flowpipe approximations. CheckMate then model checks the obtained abstraction against an ACTL specification. ACTL is a subset of CTL (computation tree logic) that states universal properties, that is, properties that are true for all trajectories of the system [4].

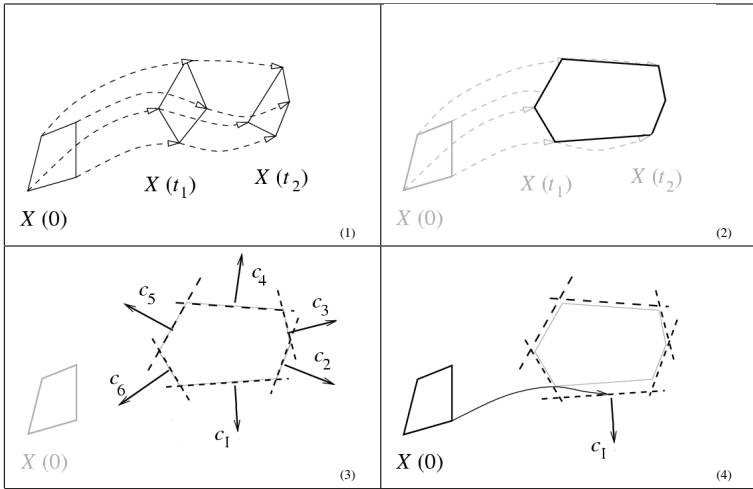
A flowpipe is the set of all states that are reachable from a given initial set by continuous evolution. A flowpipe can be viewed as a bundle of trajectories. Checkmate uses polyhedra to over-approximate flowpipes. This has the advantage that intersections of approximations with switching conditions and invariants, yield again polyhedra. The basic steps of the model checker are manipulations of polyhedra, computing flowpipe approximations, and model checking the resulting finite-state abstraction.

For a differential equation  $\dot{x} = f(x)$ , with  $x \in \mathbb{R}^n$ , let  $\varphi(x_0, t)$  be the solution at time  $t$  with initial point  $x_0$ . Given an initial set  $X(0) \subset \mathbb{R}^n$ , we define a flowpipe segment from  $t_1$  to  $t_2$  as the set  $\{x | \exists x_0 \in X(0), t \in [t_1, t_2]. x = \varphi(x_0, t)\}$ . The over-approximation of this segment is computed as follows (illustrated in Figure 1):

1. For the vertices  $x_{0_1}, \dots, x_{0_m}$  of  $X(0)$  compute  $\varphi(x_{0_i}, t_1)$  and  $\varphi(x_{0_i}, t_2)$ . CheckMate uses numerical integration to compute these points.
2. Compute a polyhedron that encloses these points. CheckMate computes either convex hulls or oriented hyper-rectangles [21], depending on an option set by user. Later in this paper we discuss implications of this choice. This polyhedron is an initial guess, and does not need to include the complete flowpipe segment.
3. Determine the linear inequalities  $c_i x \leq d_i$ , with  $c_i \in \mathbb{R}^{1 \times n}$  and  $d_i \in \mathbb{R}$ , that define the initial polyhedron.
4. Solve for each face of the polyhedron the optimization problem

$$\hat{d}_i = \max_{\substack{x_0 \in X(0) \\ t \in [t_1, t_2]}} c_i \varphi(x_0, t) \quad (1)$$

The conjunction of the inequalities  $c_i x \leq \hat{d}_i$  then defines an over-approximation of the flowpipe segment, i.e. of all points that are reachable from  $X(0)$  within interval  $[t_1, t_2]$  time.



**Fig. 1.** Steps in the flowpipe approximation are (1) simulating the vertices, (2) enclosing the simulation points in a polyhedron, (3) determining the normals and (4) increasing the size of the polyhedron, until it contains the complete segment.

Extending the flowpipe approximation to PIHAs with parametric differential equations  $\dot{x} = f(x, p)$ , where  $p$  is an unspecified constant parameter, is straightforward. We assume that  $p$  is an element of a bounded polyhedron  $P$  in  $\mathbb{R}^m$ . In the first step, we simulate all vertices of  $X(0)$  for all vertices of  $P$ . In the next two steps, we compute the enclosing polyhedron of the simulation points, as before. The last step includes the parameter in the optimization problem (1):

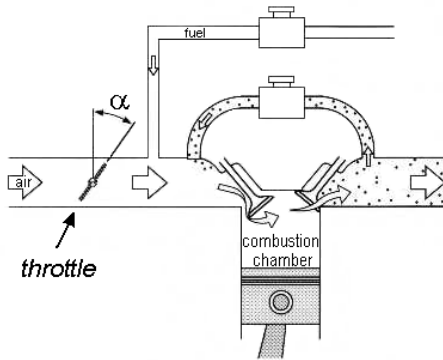
$$\hat{d}_i = \max_{\substack{x_0 \in X(0) \\ t \in [t_1, t_2] \\ p \in P}} c_i \varphi(x_0, t)$$

This defines a polyhedron that includes all states that are reachable from  $X(0)$  with parameter values in  $P$  and within interval  $[t_1, t_2]$  time. Note that while the parameter is assumed constant during continuous evolution, it may change non-deterministically when the analysis evaluates a discrete transition. This, however, includes also the case that the parameter remains constant, and the analysis is therefore conservative.

### 4 Discovering the Mathematical Model

The first step towards verification is to get an understanding of the system behavior. The essential components of the system, the control structure, and the physical laws that govern the behavior must be identified. Information from an informal description may be supplemented by a simulation model.

The mathematical model captures different system characteristics and should reflect the following aspects:



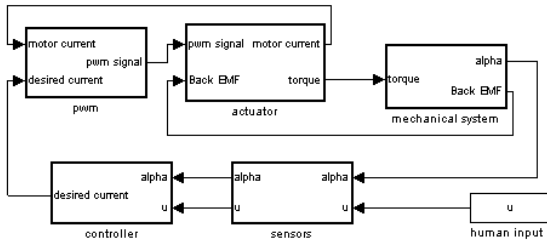
**Fig. 2.** The angle of the throttle plate determines the air flow, and indirectly the engine speed.

- Physical and mechanical laws, or chemical, biological processes. These are typically described as differential equations or inequalities, partial differential equations, and algebraic constraints.
- Switching conditions. Even if the state of a component evolves continuously, it may switch autonomously, for example in an elastic collision.
- Time scale. For example determined by the poles of a linear time-invariant system, or by the sampling rate of a sensor.
- Switching logic. May be modelled as state chart or as finite state machine. Control logic might also be given as a program, e.g. as relay ladder logic or sequential function chart for PLCs
- Control laws. Often defined by continuous-time feedback controllers or discrete time difference equations; in other applications completely encoded in the switching logic.
- Communication among components. Can be synchronous or asynchronous, with shared events or variables, using message buffers, channels, broadcasting, interrupts, or a combination of those.

If we are given a formal model, rather than an informal description, then the semantics define the mathematical model. In this case this model may be suitable directly for verification. The ETC system, however, was presented as MatLab/Simulink model, accompanied by an informal description [7,6]. We use information from simulation and informal description to develop the mathematical model manually.

The ETC system replaces the mechanical link between pedal and throttle plate. Figure 2 depicts the throttle plate as part of the powertrain. The throttle plate angle determines the airflow to the combustion chamber, and controls thus (along with the amount of injected fuel) the engine torque. The task of the ETC is to control the throttle angle, based on current control mode and human input.

The ETC system comprises a pulse-width modulation (PWM) driver, an actuator (a DC motor), the mechanical system (the throttle and spring), sensors and a controller (Fig. 3). The plant dynamics, i.e. the DC motor and the throttle behavior, are modelled as non-linear dynamic systems in Simulink. The PWM driver, the switching logic in the ETC



**Fig. 3.** An automotive throttle control system.

controller, and the scheduler are modelled as Stateflow diagrams. Control laws, such as a sliding mode controller, are modelled as difference equation. Connection between blocks are continuous-time real-valued signals. Switching in the simulation model occurs either by triggered transitions, or by discontinuous Simulink blocks, such as the sign-function block. The remainder of this section describes the mechanical system and the ETC controller in more detail.

*The Mechanical System.* The behavior of the throttle plate is governed by spring dynamics, Coulomb friction, viscous friction (airflow) and input torque. A changing current in the mechanical system induces an electro-magnetic force (back EMF) that opposes the change. The back EMF is a feedback to the actuator. The mechanical system is a second-order nonlinear continuous time-invariant system; the Coulomb friction accounts for the non-linearity.

*The ETC controller.* The ETC controller has several levels of hierarchy. The top level is a Stateflow diagram, with four normal modes, two failure modes and a startup mode. The human control mode uses a sliding-mode controller. All other modes are merely placeholders for undefined implementation details. The controller delivers in those modes just a constant and meaningless output.

The ETC controller uses a fifth-order filter (with poles -80, -80, -90, -90, -100) to smooth the input from the human driver (the sensor output). The performance of this filter determines in part whether the controller meets its performance requirements.

Sliding-mode controllers are commonly used in control applications, since they are very adaptable and versatile [22]. Sliding-mode controllers are designed as follows: First, define a surface in the state space, and show that states on the surface behave as desired. Next, design for each side of the sliding surface a control law that drives the system to the sliding surface, as illustrated in Figure 4 (a). As soon as the system hits the sliding surface, close enough to an equilibrium point, it stays on the surface and converges to the equilibrium point.

The model of the controller contains, in addition to the Stateflow model, the sliding-mode controller, the place holders for the other control modes, the fifth-order filter, blocks that model sampling of input and output, fault detection, delays, a scheduler, and finally signals that interconnect all components.

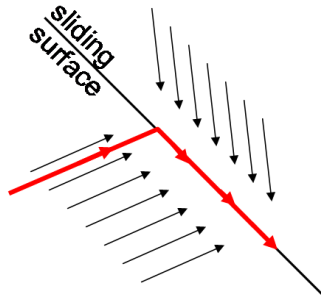


Fig. 4. Illustration of the concept of sliding-mode control.

## 5 Obtaining the Formal Requirements

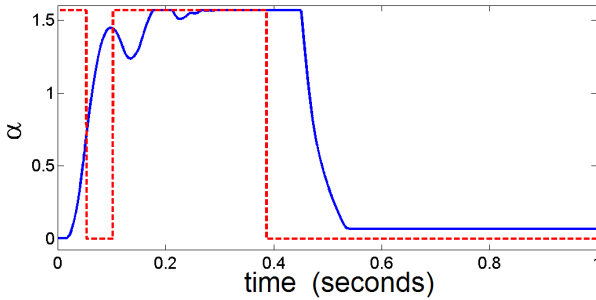
When handed an informal description of the system requirements, only some will be suitable for verification. We can distinguish three types of requirements.

- Implementation requirements. They impose certain implementation details that can be checked statically. They vary from requirements on floating point precision, platform, controller, programming language, clock-speed, scheduling policy or input range of sensors. There is no need to examine the dynamic system behavior to check these requirements.
- Requirements on representative behaviors. Those requirements define an acceptance criterion for a single execution of the system. Satisfaction of the requirement can be established by a single run of the simulation model; we will refer to them as *simulation requirements*. These requirements often serve as testing scenarios.
- Requirements on classes of behaviors. These requirements define a possibly infinite set of acceptable behaviors, of possibly infinite length. A typical example would be a liveness property such as "Each request is always eventually granted", which is defined for runs of infinite length. We refer to these requirement as *verification requirements*, since they require a formal proof.

The informal description of the ETC lists seven requirements. They include implementation requirements as well as simulation and verification requirements. We will focus on a few of these requirements for illustration.

The requirement that the nominal battery voltage should be 12 VA is a typical implementation requirement. There is no need to use simulation or formal verification, and correctness can be proven by inspection of corresponding parameter.

The rise-time requirement for the ETC is a requirement on representative behaviors. The rise time is defined as "the time required for the throttle plate angle response to a step change in pedal position to rise from 10% of the steady-state value to 90% of the steady-state value". The informal description continues, "The rise time for step changes from closed to fully open is 100ms (...)". The requirements thus put bounds on these times, given a particular change in the input signal. Whether this requirement holds can



**Fig. 5.** The throttle angle (solid) in response to an input (dashed) that changes between a wide open throttle, and a fully closed throttle.

be answered by a single simulation with the test input. The simulation test shows that the rise time requirement is indeed satisfied.

Another requirement was informally expressed as: "[The] throttle plate shall never hit the stops" [7, p. 10]. This requirement has to hold, no matter the input or operation mode (failure modes are excluded). This requirement is a candidate for formal verification. However, running just a few simulations shows that it is possible to reach the upper bound with a positive velocity, as can be seen in Figure 5, at approximately time 0.2 seconds. Formal verification is therefore not necessary. The counterexample proves that the requirement is not satisfied.

A verification requirement for the ETC is: The system will, after a step input, always eventually enter a certain neighborhood of the steady-state, and remain there forever. This property has an unlimited time horizon, unlike simulation requirements. In addition we assume that spring constant and spring equilibrium may deviate from their nominal values by up to 20%. Rather than defining infinitely many behaviors for a single system, this requirement defines a single behavior of infinite length for a infinite class of systems. Simulations in contrast require the parameters to be known exactly. In the verification model, these parameters are only known within bounds, and the verification covers all parameters values within these bounds.

This verification requirement was not part of the informal description for the ETC. We introduced it, since all requirements in the informal description could be checked easily by inspection or simulation. As a side note, in the cases where counterexamples were found, there was a subjective acceptance criterion, that considered these violations as not significant. The requirements are not just true or false, but certain violations are acceptable within a certain, albeit subjective range.

## 6 Obtaining the Verification Model

A limiting factor in hybrid system verification is the number of continuous variables and the number of control locations [13]. Verifying a model with a fifth-order system just to filter the input is already challenging. The ETC system has in addition a few character-

istics that make verification of the full model impossible. As in other applications, we do not verify the implementation model but a scaled-down version [19].

A well known technique for scaling down hybrid system models is abstraction [10]. An abstraction preserves the essential behavior of the original system. It is guaranteed that if a safety property holds for the abstraction, it also holds for the original system. But techniques from system and control theory, such as order-reduction and linearization, can also be useful to obtain proper approximations of the original system.

Considerations when building a verification model are the following.

- First, the verification model must be in the class of systems that the model checker can handle. Or vice versa, one has to choose a tool that can handle the class of systems.
- The kind of dynamic behavior. Models with non-linear dynamics are harder to analyze than models with linear dynamics, which are harder to analyze than multi-rate problems.
- The number of continuous variables. Even if the problem has simple dynamics, additional continuous variables add complexity.
- The switching behavior of the system. Even systems with few locations can have undesirable switching behavior. A particular example is Zeno-behavior, i.e. a behavior that exhibits an infinite number of discrete transitions in finite time.

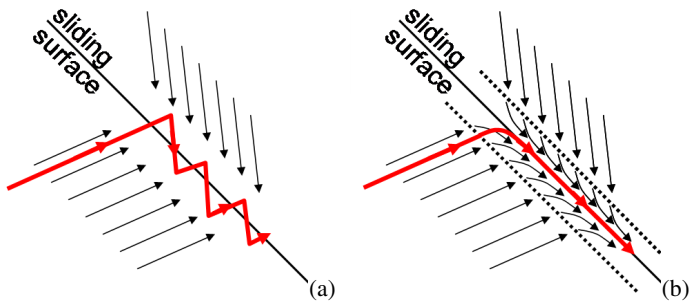
We illustrate the process of obtaining a CheckMate model for the ETC case study. The starting point is the OEP model. The OEP model serves two purposes: It is used for simulation studies, and it is used as a blueprint for implementation. There is limited incentive to be concerned about complexity, since it is used for simulation rather than verification. As blueprint for implementation, the OEP model contains details such as what task has to run on what platform and under which scheduling policy. On the other hand, when implementation details are unknown it contains empty subsystems that serve as placeholders for future implementation.

PIHAs are continuous-time models<sup>1</sup> and can include non-linear dynamics. However, some non-linearities can cause numerical problems, and a linear abstraction might be easier to analyze. The number of variables is a concern, too; models with more than 5 continuous variables are typically hard to analyze. Verification for more than 10 continuous variables is in most cases impossible. Furthermore, CheckMate assumes no two transitions can happen in zero time, which in particular excludes certain Zeno-behavior.

*Obtaining a Continuous-Time Model.* Since CheckMate models are continuous-time, the discrete-time components of the OEP model have to be replaced by appropriate continuous-time components. Discrete-time components in the OEP model are PWM driver, sensors and ETC controller.

The Simulink model of the ETC controller has only one mode with meaningful dynamics, the human control mode. We omit in the verification model all other modes, and can omit also the control logic. Filter and sliding-mode controller were designed as continuous-time models, but then discretized to become part of the ETC controller. Hence, we replace them by their continuous-time equivalent. The sampling times of

<sup>1</sup> There are extensions of CheckMate that allow discrete time and sample-data models [20,15].



**Fig. 6.** The left figure illustrates chattering at the sliding surface. The right figure illustrates the effect of a boundary layer.

PWM and sensors are a few orders smaller than the time scale of interest (100ms) and can replace them by gains.

*Resolving Zenoness.* CheckMate assumes, as most hybrid model checkers do, that all behaviors are non-Zeno. The sliding-mode controller, in contrast, intentionally drives the system to a surface where infinite, and even uncountable switching occurs (in the continuous-time realization). If we use a fixed step integration routine the solution will start chattering, which may lead to unreliable results. If we use a variable-step integration routine the procedure tends to get stuck on the sliding surface.

To resolve this problem, we define a *boundary layer* (or  $\epsilon$ -neighborhood) around the sliding surface. We apply the sliding-mode controller outside of this  $\epsilon$ -neighborhood, and replace it inside by a control law that is continuous and drives the system to the surface. The controller is thus equivalent to the original controller outside the boundary layer, and there is a steep but continuous transition from one sliding mode to the other. On the sliding surface the control law is equal to the so-called equivalent controller. Figure 6 (b) depicts the basic idea of a boundary layer. The boundary layer leads to a numerically well-conditioned, non-Zeno, and close approximation of the ideal sliding-mode behavior. Boundary layers are a very common approach used in physical systems to mitigate the physical stress by chattering that can lead to mechanical damage. A more thorough discussion of reachability analysis for sliding-mode controllers can be found in [14].

*Modelling Non-Linearities.* The mechanical system describes the dynamics of the throttle plate. This second-order system is nonlinear due to coulomb friction. We have to decide whether to include this non-linearity and non-linearities caused by saturation (actuator) and sliding-mode control as different modes, or as non-linearities. One extreme choice would be to model the ETC as non-linear hybrid system with a single mode. The other extreme choice would be to model it with linear dynamics, which results in 18 modes for the ETC problem.

If we put the complete behavior in a single non-linear differential equation, the flowpipe-approximation gets worse and computationally more expensive when the vector field changes abruptly, e.g. when the system changes the sliding modes.

If we model the system with many modes but with linear dynamics, it will result in a lot of switching. Each time the analysis algorithm encounters switching between modes it uses over-approximations of previous steps, and over-approximation errors may proliferate. Many modes lead thus to an increased over-approximation error. Based on above consideration we have chosen to model the Coulomb friction and saturation as nonlinearities, to reduce the number of modes, and to model the sliding-mode controller as different modes, to avoid over-approximation errors due to sudden changes of the vector field. This decision was made after running a number of experiments with different models.

*Reducing the Order.* The ETC uses a fifth-order filter to smooth the input from the human driver. This means that the filter alone has more than twice as many state variables as the rest of the system. Since verification of hybrid systems becomes more difficult with each additional dimension, we reduce the order of the filter. We obtain a reduced-order filter using the model-reduction capabilities of MATLAB's system identification toolbox. The combined dynamics of plant and reduced filter result in a fourth-order system with nonlinear dynamics. For a more thorough discussion of order-reduction for hybrid system verification see [8].

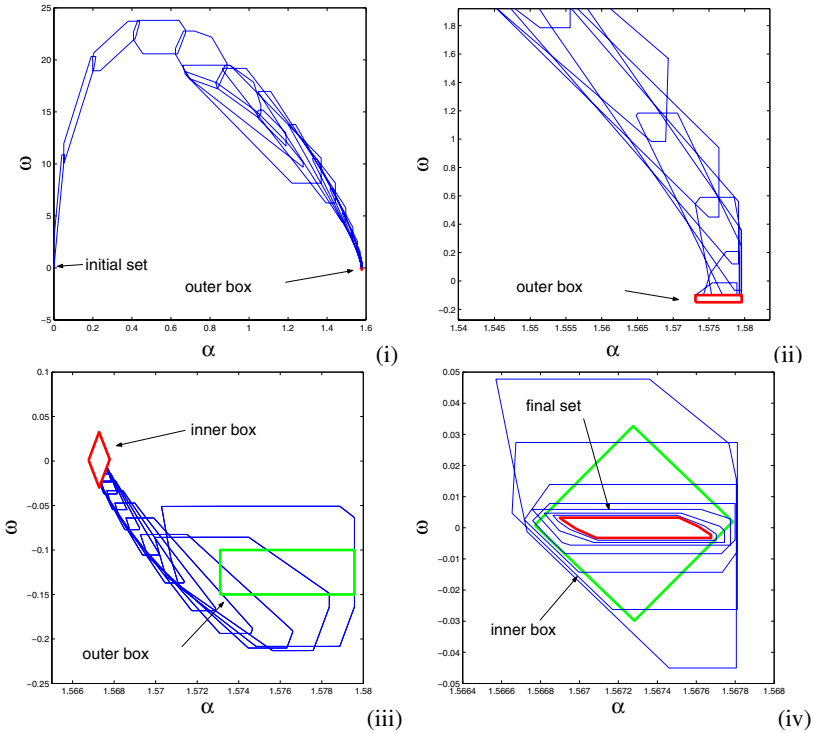
## 7 Setting Up the Problem

This section addresses the problem the requirement cannot be verified directly, due to the size of the verification problem. A common approach is to decompose system and property into smaller problems [12,18]. We illustrate this idea, by decomposing the liveness property for the ETC into a series of properties, which can all be verified with CheckMate.

The property that we verify is the following. Given that the system is in steady-state with throttle angle  $\alpha = 0$ , assume a step change in the desired angle to 89.8 degrees (which is the maximal input; the input has a safety margin of 0.2 degrees) at time 0. Verify that the angle will always eventually reach a 2% neighborhood of the desired angle, and remain there forever. We furthermore assume that the spring constant and spring equilibrium may deviate from their nominal values by 20%. This means that they may take any value in this range.

We define a cascade of subproblems to show that the system behaves as desired. For each of the stages we use a variant of the basic CheckMate model. For each stage we define a different initial and target set. The target set of one stage is then the initial set of the next stage.

*Transient phase.* The first stage of the cascade deals with the transient phase when the throttle angle changes quickly in response to the step input. We show that all trajectories that start from the initial set – in this case the origin – hit the first target set, the so called *outer box*. Figure 7 (i) and (ii) depict projections of the flowpipe approximations that show that all trajectories do indeed reach the outer box. The model checker verifies furthermore that the system will always reach this set eventually.



**Fig. 7.** Figure (i) and (ii) depict the flowpipe approximation for the transient phase. Figure (ii) is a close up of Figure (i). Figure (ii) depicts the flowpipe approximation in the neighborhood of the steady-state value. Finally, Figure (iv) shows that the states that start in the inner box will eventually return to this set. Note that these figures show only projections of the flowpipe segments onto throttle angle  $\alpha$  and angular velocity  $\omega$ .

*Regulation phase.* The next stage is to show that all trajectories that start in the *outer box* will eventually reach the *inner box*. We use the same model as for the transient phase, but of course with the outer box of the transient phase as initial set, and the inner box as target set. Figure 7(iii) show that the system starts in the outer box and all trajectories converge quickly to a neighborhood of the steady-state. No segment of the flowpipe approximation violates the 2% bound. This guarantees that once the system enters the outer box, the inner box will be reached without violating the 2% bound.

*Asymptotic behavior.* As the last step we show that the *inner box*, a neighborhood of the steady-state value, maps onto itself in a finite number of steps. CheckMate finds a flowpipe segment that is completely contained in the inner box, i.e. the initial set of this stage. This means all trajectories that start in the inner box, return to this set. None of the computed flowpipe segments of the over-approximation violates the 2% threshold.

Figure 7(iv) depicts the result. Note that it is not sufficient to show that some flow pipe segment is contained in another, since they are over-approximations. We cannot assume that all states in a segment are actually reachable. But if some segment is inside

the initial set we know that this set is recurrent. All states that can be reached in a certain time interval from the initial set will be contained in this segment, and thus also in the initial set. This completes the verification.

## 8 Setting Up the Verification Algorithm

The previous section presented the verification results. Getting the verification results is not only a matter of defining subproblems and corresponding models – which is some work by itself – getting the verification to run requires also a fair amount of tweaking the model checker. For finite-state model checkers this might entail choosing a proper order of the variables, for other model checkers it might entail to find a proper size for the hash table. To give an impression of the kind of choices that have to be made for CheckMate, we elaborate on the choice when to use convex hulls and when to use oriented rectangular hulls in the approximation.

This choice makes a difference in two different steps of the algorithm. As mentioned before, CheckMate obtains an initial approximation of the flowpipe segment by computing a polyhedron that encloses the simulation points (step 2, page 265). The convex hull of these points is by definition the smallest polyhedron that contains all points. Using the convex hull in this step has the advantage that the over-approximation error is small. A drawback is that the convex hull will also yield polyhedra with a lot of faces. Each additional face leads to one additional optimization problem in the last step of the flowpipe approximation procedure.

CheckMate offers as an alternative to use the so called oriented rectangular hall (ORH) routine [21]. The ORH routine chooses an oriented hyper-rectangle that keeps the over-approximation error small and limits at the same time the number of faces. For the ETC model with four state variables, the ORH will result in a polyhedron with exactly eight faces. If we use the convex hull approximation instead, CheckMate computes polyhedra with up to 119 faces before it gets stuck. Using the ORH solves this problem, and all segments of the approximation can be computed.

Another point where the choice between the convex hull and the ORH routine matters, is when CheckMate computes states reached by a discrete transition. To do so, CheckMate intersects each segment of the flowpipe approximation with the switching condition. If more than one segment intersects with a switching condition, the verification algorithm proceeds with an over-approximation of the union of these intersections. This over-approximation can either be the convex hull or the ORH of those sets. For this case study we found that the results of the ORH are too conservative. The over-approximation error soon becomes too large.

To summarize. To get the verification to run requires a proper setup of the verification algorithm. We use, for example, the ORH routine to compute the polyhedra of the flowpipe approximation, and the less conservative convex hull routine to compute the over-approximation of the intersections with switching conditions. Similar choices had to be made to find the proper integration routine, and to chose parameters for numerical integration and optimization routines.

## 9 Discussion

The starting point for the work presented in this paper was the OEP model and an informal description of the ETC system. The first step towards verification was to discover the underlying mathematical model. The OEP model was useful since it already provided information about the main components. When a such a model is not present in the beginning, building a simulation model can help significantly to understand the problem.

The second step towards the verification was to formulate the requirements of the system. In our case we had an informal description to start from. We found that none of the given requirements was suitable for verification. Most of the requirements were simulation requirements that could be checked by running a simulation, or implementation requirements that could be checked by inspection. For others we easily found counterexamples, and verification was not necessary either.

There is no need to use verification to check simulation and implementation requirements. Verification methods should not be used for the sake of verification. Verification should be used if the requirements are formulated for parametric models, uncertain initial conditions, or non-deterministic models. In that case formal verification can be valuable and complementary to simulation-based methods. We defined a liveness property for the system that captures the spirit of the simulation scenarios, but that also illustrates the added value of verification.

The mathematical model and the liveness requirement were the basis of the verification model. Building the verification model involved simulation of various models. We took into account that we needed a continuous-time model, with a small number of continuous variables, and dynamics that are numerically well-conditioned. The final result was a fourth-order hybrid system with non-linear dynamics.

Given the verification model we could not verify the requirement directly, but decomposed the problem into smaller problems. For the ETC case study it was sufficient to define a series of three problems that were solved by CheckMate. Finally, we had to find suitable verification parameters, which required several experiments with different settings.

Given our experience from the ETC case study, future research should focus on supporting the process described in this paper. Hybrid systems verification will in the foreseeable future not become a completely automated process. There is a lot of work currently focussing on automating and supporting particular steps, but little that aims to support the complete process. Tool support can be useful to guide and assist the designer throughout the process that leads from informal description to verification result. At the same time it can help to make this process transparent, such that the steps and choices can be re-evaluated at a later stage.

## References

1. R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

2. R. Alur and G. Pappas, editors. *Hybrid Systems: Computation and Control, 7th International Workshop, HSCC 2004 Prague, Philadelphia, PA, USA, March 25-27, 2004*, LNCS 2993. Springer, 2004.
3. A. Chutinan and B.H. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In *HSCC 99*, LNCS 1569. Springer Verlag, 1999.
4. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.
5. Thao Dang. *Vérification et synthèse des systèmes hybrides*. PhD thesis, Verimag, Grenoble, 1999.
6. P. Griffiths. Embedded software control design for an electronic throttle body. Master's thesis, UC Berkeley, 2002.
7. Automotive OEP Group. Electronic throttle control – end-to-end challenge problem for automotive mid-term experiment. Available at <http://vehicle.me.berkeley.edu/mobies/papers/>, August 2001.
8. Z. Han and B. H. Krogh. Using reduced-order models in reachability analysis of hybrid systems. In *IEEE Proc of American Control Conference*, 2004, to appear.
9. T.A. Henzinger, P. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.
10. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):540–554, 1998.
11. T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Proc. of the 27th Annual Symposium on Theory of Computing*. ACM Press, 1995.
12. T.A. Henzinger, M. Minea, and V. Prabhu. Assume-guarantee reasoning for hierarchical hybrid systems. In *HSCC 01*, LNCS 2034. Springer-Verlag, 2001.
13. T.A. Henzinger, J. Preussig, and H. Wong-Toi. Some lessons from the hytech experience. In *Proceedings of the 40th Annual Conference on Decision and Control*, pages 2887–2892. IEEE Press, 2001.
14. J. Kapinski and B. H. Krogh. Verifying asymptotic bounds for discrete-time sliding mode systems with disturbance inputs. In *IEEE Proc of American Control Conference*, 2004, to appear.
15. J. Kapinski and B. H. Krogh. A new tool for verifying computer controlled systems. In *IEEE Conference on Computer-Aided Control System Design*, pages 98–103, Sept 2002.
16. A. Kurzhanski and P. Varaiya. Ellipsoidal techniques for reachability analysis. In N. Lynch and B. Krogh, editors, *HSCC*, LNCS 1790, pages 203–213. Springer, 2000.
17. G. Lafferriere, G.J.. Pappas, and S. Yovine. A new class of decidable hybrid systems. In F.W. Vaandrager and J. van Schuppen, editors, *Hybrid Systems: Computation and Control*, LNCS 1569, pages 103–116. Springer, 1999.
18. N.A. Lynch, R. Segala, and F.W. Vaandrager. Hybrid I/O automata revisited. In *HSCC 01*, LNCS 2034. Springer-Verlag, 2001.
19. J. Rushby. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI, Menlo Park, CA, 1995.
20. B. I. Silva and B. H. Krogh. Modeling and verification of hybrid system with clocked and unclocked events. In *Proc. of the 40th Conference on Decision and Control.*, 2001.
21. O. Strusberg and B. Krogh. On efficient representation and computation of reachable sets for hybrid systems. In *HSCC'2003*, LNCS 2289. Springer, 2003.
22. V. Utkin. *Variable structure systems with sliding modes*. AC-22(2):212-222. IEEE Transactions on Automatic Control, 1977.