Model-Driven Hybrid and Embedded Software for Automotive Applications

Anouck R. Girard, Adam S. Howell and J. Karl Hedrick

***Abstract* — Complex large-scale embedded systems arise in many applications, in particular in the design of automotive systems, controllers and networking protocols. In this paper, we attempt to present a review of salient results in modeling of complex large scale embedded systems, including hybrid systems, and review existing results for composition, analysis, model checking, and verification of safety properties. We then present a library of vehicle models designed for cruise control (and CACC) that attempt to cross the chasm between theory and practice by capturing real-world challenges faced by industry and making the library accessible in a public domain form.**

***Index Terms* — Automotive Control, Embedded Software, Hybrid Systems, Control Architecture, Model-Driven Software**

I. Introduction

Real-time, embedded systems have become prevalent in our everyday life. An embedded system is a special-purpose computer system built into a larger device [6]. Since many embedded systems are produced in the range of tens of thousands to millions of units, reducing cost is a major concern. Embedded systems often use a (relatively) slow processor clock speed and small memory size to cut costs. Programs on an embedded system often must run with real-time constraints; that is, a late answer is considered a wrong answer. Often there is no disk drive, operating system, keyboard or screen. Cell phones, PDA, televisions, washing machines, microwave ovens and calculators are all examples that contain embedded processors.

Demands placed on the functionality, complexity and critical nature of embedded systems are ever increasing. Modern-day automobiles now contain many different processors that perform functions such as engine control, ABS, vehicle stability and traction control, and electronic control of power windows, mirrors, and driver-seat settings. Aircraft control systems can be several orders of magnitude more complicated, due in part to greater need for system reconfiguration from mission to mission and fault tolerance requirements that include having triple redundant copies of critical sensing and actuation systems.

This work was supported by DARPA/ITO in the MoBIES project (Model-Based Integration of Embedded Systems) under Grant F33615-00-C-1698.

A. Girard and A. Howell are visiting post-doctoral researchers at the University of California at Berkeley, Berkeley, CA, 94720 USA (e-mail: anouck@eecs.berkeley.edu, ahowell@path.berkeley.edu).

J. K. Hedrick is a Professor of Mechanical Engineering at the University of California at Berkeley, Berkeley, CA, 94720 USA (e-mail: khedrick@me.berkeley.edu).

As expectations increase for more complicated embedded systems, the need for organized real-time, embedded software development processes becomes more pronounced. However, current industry standards fall short of producing high degree of confidence, reusable code that fulfills this need. A large pitfall of the current state of the art is that most bugs are caught in the final phases of the process, at system integration and testing time. Correcting problems at this stage often involves modifying the system requirements, specification or design, and such changes are costly as they imply significant rework of the system.

We begin by reviewing current approaches for the modeling, composition, analysis and model/property checking of complex systems. The use of well-understood mathematical modeling frameworks allows formal verification of the system. Concepts are presented using the simplest formalism possible to develop intuitions, and for extensions the reader is invited to consult the references. We then proceed to present a model-based process that places strong emphasis on performing as much testing and verification in “tight-loops” as possible. Thus we hope to catch bugs early on in the development process and minimize costs associated with fixing the problems. We choose to frame our models and controllers in the context of hybrid automata which allows formal verification of the controller using third party tools. Furthermore, timing properties of the software can be verified with additional information about the experimental platform. This gives us a high degree of confidence in the performance of the generated code. We also present a library of models that have increasing complexity and were developed in the context of intelligent cruise control applications. These models range from a linear double-integrator vehicle model to an eleven continuous state composed hybrid model that was used for simulations and implementation of an ACC/CACC system on experimental vehicles.

II. Modeling and Analysis of Complex Large-Scale Systems

A comprehensive review of all available methods for the modeling and analysis of complex systems is beyond the scope of this paper. Many techniques are available and can be broadly classified in a range of increasing complexity of system features to be modeled, starting from state machines, and going on to labeled state machines, I/O automata, composition, timed systems, hybrid systems and dynamic networks of hybrid automata. For a comprehensive review, the interested reader is referred to [3].

In a very general way, we consider systems that can be described as beginning in a "starting state" and progress from state to state in discrete jumps according to a set of specified rules. In general, systems are *nondeterministic*, that is, the next state might not always be determined by the previous state. There might be explicit choice points, for example in an algorithm; or there just might be different orders in which things can be done.

The basic mathematical model to describe complex systems is called a *state machine*. A state machine [2] is formed of a set of states Q, a set of allowable starting states Q_0, and a set of allowed transitions between states δ.

An *execution* of a state machine is a (possibly infinite) sequence of states such that the initial state q_0 is in the set of allowable starting states, and for each state q_i in the sequence, the transition from q_i to q_{i+1} is in δ.

One useful property to understand the behavior of a system is to study which states can be reached in its executions. A state is said to be *reachable* if it's the final state in some finite-length execution. The Mathworks' Stateflow toolbox [28] is a tool to visually model and simulate complex systems based on finite state machine theory.

Proving versus testing or simulation

Proving properties (such as correctness or safety properties) of a system is quite different from simple testing or simulation. Since most complex systems operate in the real world, they are faced with a very large (when not infinite) number of inputs; exhaustive testing is rarely possible, and partial testing does not guarantee proper behavior of the system for those inputs that were not tested. Proofs of complex system properties are playing a growing role in assuring quality, for critical or manned systems [5].

Proofs for state machines

There are a number of things that can be proven about systems that are modeled as state machines [1], such as:

- Invariant properties (some predicate of the state variables is true in all reachable states)
- Eventuality properties (eventually a = b)
- Time bound properties (after T steps, some predicate or property is true)

These properties are so important that people have developed languages for expressing them and computer programs to check for them.

Invariant properties can be used to describe properties that are always true, no matter how the system behaves. This can be useful to prove basic correctness properties for systems. Invariant proofs often use mathematical induction. Invariant and eventuality properties can be used to characterize *safety* (for example, the distance between two vehicles is never negative, or steady-state is reached in a controller). Safety properties are sometimes described as those which are finitely refutable; that is, if a behavior does not satisfy the property, then one can tell who took the step that violated it.

Other properties that one may wish to prove are true are that the executions terminate, or that they finish in some fixed amount of time. These properties are dubbed termination properties. Basic definitions of safety vs. liveness can be found in [4]. Several model checking tools can be used to prove liveness, invariant and eventuality properties [11-12].

Composition of systems modeled as state machines

Some systems are too big or complicated to model as a single state machine: one needs to break the description into pieces, using either abstraction (giving high-level description of systems, then separately, implementing the high-level description using low-level elements), or composition (building all the components separately out of individual specifications, then putting them all together) [13].

To decompose systems, we augment the state machine models considered previously with *labels* describing inputs, outputs, and internal parts of a system. Internal variables cannot be used by other components, and *externally visible behavior* is determined by the relationship between inputs and outputs. A slightly more general construct than labeled state machines is *I/O automata*, which are nondeterministic, infinite state machines whose inputs and outputs actions are labeled [1].

A component is said to *implement* another if their externally observable behavior is the same, so that one component can be substituted for another in a larger system. In addition, *composition* refers to the notion that two components can operate in parallel and interact. When two components interact, all that each "sees" about the other is its externally visible behavior. Composition allows one to understand the behavior of a large system once one understands the behavior of each of the individual components. A general principle for parallel composition appears in [14]. The Pi-calculus, which is an algebra that accommodates many kinds of combination operators, is described in [15].

Timed Systems

In the context of real-time, embedded systems, one needs to incorporate a notion of time in the modeling. Several modeling formalisms have been proposed, including timed I/O automata [1], reactive systems [16] (which can identify that one events occurs before another, but not by how much), time transition systems [17] (in which a time stamp is affixed to each state in a computation), and clocked transition systems [18] (timers increase uniformly when time progresses, and can be reset arbitrarily on transitions). Model checking tools are available for timed systems [18].

Hybrid Systems

A hybrid system allows the inclusion of continuous components in a timed system. Such continuous components may cause continuous changes in the values of some state variables according to some physical or control law.

Formally, a hybrid automaton consists of control locations with edges between them. The control locations are the vertices in a graph. A location is labeled with a differential inclusion, and every edge is labeled with a guard, a jump and a reset condition. A hybrid automaton is H = (L, D, E) where:

- L is a set of control locations
- D: L$\rightarrow$ Inclusions where D(l) is the differential inclusion at location l.

- $E \subseteq L \times Guards \times Jumps \times L$ are the edges – an edge $e = (l,g,j,m) \in E$ is and edge from location l to location m with guard g and jump relation j.

The state of a hybrid automaton is a pair (l,x) where l is the control location and $x \in \Re^n$ is the continuous state.

Modeling frameworks and verification tools for hybrid automata are available from [19-27]. Dynamic Networks of Hybrid Automata (DNHA) include the dynamic creation of hybrid automata, which then get composed with previously existing hybrid automata.

III. Real-World Challenges: Model-Driven Development Process

Our model-driven process, as shown in figure 1, places strong emphasis on performing as much testing and verification in "tight-loops" as possible. Thus we hope to catch bugs early on in the development process and minimize cost associated with fixing the problems. We choose to frame our models and controllers in the context of hybrid automata. This is the most useful modeling formalism for us, as we are modeling physical processes that are governed by differential equations, such as position and speed of the vehicles, in addition to modeling time. Simulation and real-time code generation are conducted using the TEJA software suite [29]. Safety properties are verified on simple models (including "the distance between the two vehicles is never strictly less than zero") [10], and timing properties of the code are analyzed using schedulability analysis [9].

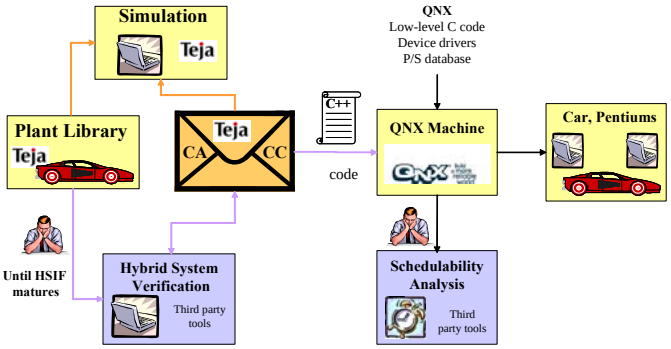

Figure 1. Intelligent cruise control software development process.

This development process was conceived in a joint effort between the University of California at Berkeley, Ford Scientific Research Laboratories and General Motors. The approach is applied to Adaptive Cruise Control (ACC) and Cooperative ACC systems.

IV. The V2V Libraries

We present a set of four levels of models for vehicle-to-vehicle (V2V) control (that is, for vehicle following applications and longitudinal control of vehicles, such as cruise control, ACC and CACC). The goal of this set of models is to present a range of models adequate for model configuration, composition, checking and analysis using a variety of tools, with relevance to V2V problems.

In the first three levels of modeling, we have two types of automata, one for the vehicle model, and one for the vehicle controller. We create two instances of each, to have two "full" vehicles (model + controller) in our scenario. Each vehicle is assumed to have an ideal forward looking sensor (FLS) that can detect a vehicle within a specified maximum range and measure both the range and range rate of the detected vehicle. An ideal communications channel with any surrounding vehicle is also assumed to allow knowledge of every vehicle's acceleration.

Linear models

Vehicle Model

The model of the vehicle dynamics is given by the following second order continuous dynamic system,

$$\dot{x} = v$$

$$\dot{v} = \frac{1}{\tau_{model}}(u - v)$$

where x and v are position and velocity of the vehicle, u is the control input given by the controller, and τ_{model} is the time constant of the vehicle's velocity dynamics.

Vehicle Controller

The controller is described by a hybrid automaton with two states: velocity following and distance-following. The initial state of the controller is the velocity following state, where the controller tracks a fixed desired velocity using the discrete time control law,

$$u[k] = v[k] + \frac{\tau_{model}}{\tau_{des}}(v_{des} - v[k])$$

where u[k] and v[k] are the control input and velocity measurement at the current time step, τ_{model} and τ_{des} are the known time constant of the model and the desired dynamics, and v_{des} is the desired velocity. Note that the controller runs at a fixed sample time, and the control is essentially passed through a zero-order hold to generate the continuous time control input used in the vehicle model. The controller will remain within this state until another vehicle is detected by the FLS, after which the controller will transition to the distance following state.

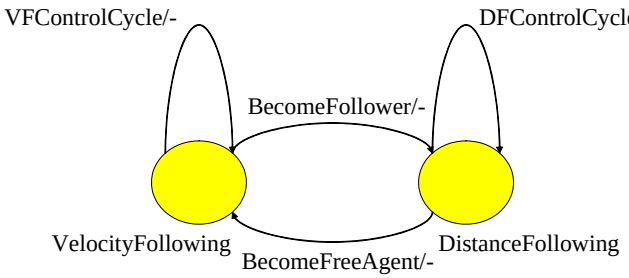

Figure 2: Simple CC/CACC controller.

In the distance following state, the control input is computed using the discrete time control law,

$$u[k] = v[k] + \tau_{model}(a_{prec}[k] + 2\zeta\omega_n \dot{\delta}[k] + \omega_n^2(\delta[k] - \delta_{des}))$$

where a_{prec} is the preceding vehicles acceleration known via communications, $\dot{\delta}[k]$ and δ are the range rate and range measured by the FLS, ζ and ω_n are controller parameters that determine the closed loop dynamics, and δ_{des} is the desired inter-vehicle spacing. Similar to above, the controller will remain in this state until there is no vehicle detected by the FLS, after which the controller will transition back to the velocity following state.

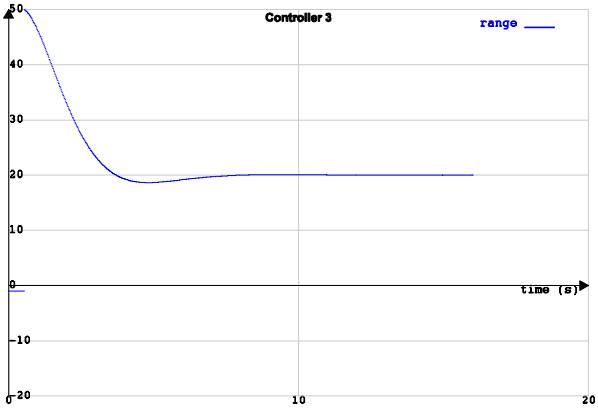

Figure 3: Range between vehicles 1 and 2, with τ_{model}=1s, τ_{des}=0.5s, v_{des}=20 and 21 m/s, ζ=1, ω_n=0.71, δ_{des}=40m, $\delta_{\max}$=120m, T_s=0.02s, and (x_0, v_0) being (0,19) and (60, 22).

Nonlinear models

Vehicle model

A model of vehicle powertrain dynamics was derived for this example and is given by the following second order continuous dynamic system,

$$\dot{x} = hR^* \omega_e$$
$$\dot{v} = hR^* \dot{\omega}_e$$

where:

$$\dot{\omega}_e = \frac{1}{J_{eff}}(k_u u - C_a h^3 R^{*3} \omega_e^2 - hR^* F_{roll})$$

x and v are position and velocity of the vehicle, u is the control input given by the controller, C_a is the vehicle drag coefficient, F_{roll} is the tire rolling resistance, R^* is the operating gear ratio, h is the wheel radius, ω_e is the engine speed, and k_u is the control coefficient .

The total moment of inertia is given by the following equation,

$$J_{eff} = I_e + R^{*2} I_\omega + h^2 R^{*2} M$$

where I_e and I_ω are the moment of inertias for the engine and wheel respectively, and M is the vehicle mass.

Vehicle Controller

The controller is described by a hybrid automaton with two states: velocity following and distance following. The initial state of the controller is the velocity following state, where the controller tracks a fixed desired velocity using the discrete time control law,

$$u[k] = \frac{1}{k_u}(C_a hR^* v[k]^2 + \frac{J_{eff} a_{syn}}{hR^*} + hR^* F_{roll})$$

where

$$a_{syn} = \frac{v_{des} - v[k]}{\tau_{des}}$$

u[k] and v[k] are the control input and velocity measurement at the current time step, and τ_{des} is the known time constant of the model and the desired dynamics, and v_{des} is the desired velocity. Note that the controller runs at a fixed sample time, and the control is essentially passed through a zero-order hold to generate the continuous time control input used in the vehicle model.

The controller will remain within this state until another vehicle is detected by the FLS, after which the controller will transition to the distance following state. In the distance following state, the control input is computed using the discrete time control law,

$$u[k] = \frac{1}{k_u}(C_a hR^* v[k]^2 + \frac{J_{eff}(a_{prec}[k] + 2\zeta\omega_n \dot{\delta}[k] + \omega_n^2(\delta[k] - \delta_{des}))}{hR^*} + hR^* F_{roll})$$

where a_{prec} is the preceding vehicles acceleration known via communications, $\dot{\delta}[k]$ and δ are the range rate and range measured by the FLS, ζ and ω_n are controller parameters that determine the closed loop dynamics, and δ_{des} is the desired inter-vehicle spacing. Similar to above, the controller will remain in this state until there is no vehicle detected by the FLS, after which the controller will transition back to the velocity following state.

Nonlinear models with look-up-tables

A look-up table was added to the vehicle models to accommodate for variable gearing based on speed.

Complex model

The vehicle model used for controller development is an complex model, which includes vehicle state dynamics, throttle and brake system dynamics, a two-state model for the spark-ignition engine as presented in [8], including external data maps which require interpolation, and models of the torque converter, transmission and wheel slip, as shown in figure 4.

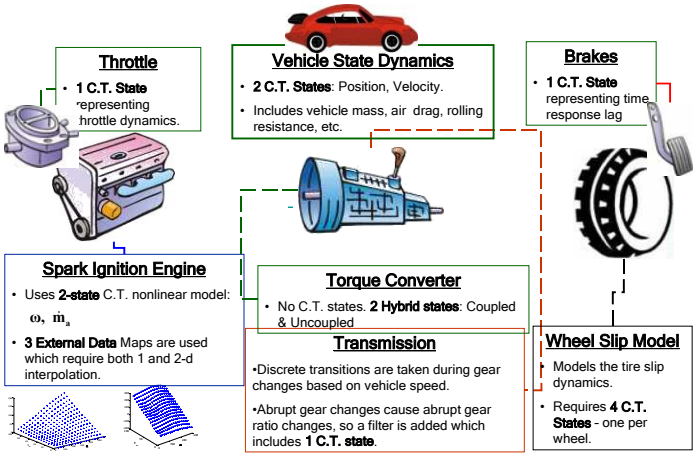

Figure 4. Complex vehicle model.

The vehicle state dynamics have two continuous states, vehicle position and velocity, and consider vehicle mass, air drag and rolling resistance. The throttle and brake dynamics are both first-order, with one continuous state for each representing actuator dynamics for the throttle and time response lag for the brakes. The model contains 11 continuous states, 3 external data look-up functions requiring interpolation, and several very nonlinear functions, including engine dynamics, the torque converter model and tire friction effects. Complete details of the model are available in [7].

The controller design process stems from system requirements. We consider only the longitudinal control of passenger vehicles (no automatic steering). Vehicles may be heterogeneous, that is of different types, makes and models. In our experiments, we limit ourselves to the utilization of two automated cars. This excludes cut-in scenarios for the experiments (they were considered in simulation). The desired behavior for the automated vehicle is to perform cruise control if the road is clear, otherwise follow the vehicle in front at a predetermined time gap, using communications if available.

The controller was split hierarchically between an upper level controller that has several modes, namely cruise control (CC), adaptive cruise control (ACC) and coordinated adaptive cruise control (CACC). In ACC mode we use only information from the host vehicle's forward-looking sensors, and in CACC mode we supplement this information with data from the wireless communication system.

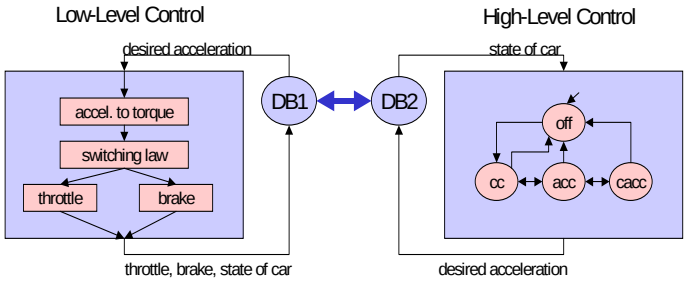

Figure 5. Decomposition of the vehicle control system in modes.

Experimental Results

The generated software for CC, ACC and CACC was run on experimental test vehicles operated by California PATH. The experimental vehicles are 1996 and 1997 model-year Buick LeSabres.

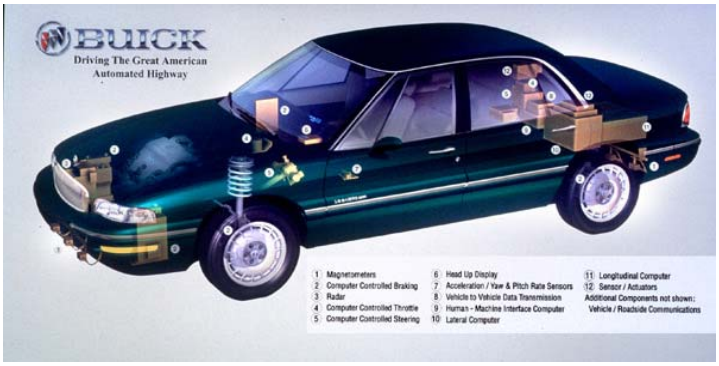

Figure 6. Experimental test vehicles.

They are equipped with throttle, brake and steering actuating systems, as well as with numerous sensors, including accelerometers, wheel speed sensors, engine speed and manifold pressure sensors, and magnetometers that are used as part of the lateral control. In addition, both an EVT-300 Doppler radar and a Mitsubishi lidar were mounted to the front bumper of the vehicles.

There are two control computers located in the trunk. Both run the QNX 4.25 real-time operating system and communicate over serial port connections. The computers run a host of tasks necessary for automated control of the vehicles, including reading sensor data and writing to actuators, control computations such as those described above for the ACC/CACC system and low-level controllers, and tasks pertaining to driver display information.

There are about 30 different tasks running on the most heavily loaded of the control computers, and timing is fairly critical as human test drivers are in the cars during runs and their safety is paramount. In consequence, we teamed up with another MoBIES team at Carnegie Mellon University to perform schedulability analysis of all tasks, using Rate Monotonic Scheduling algorithms [9]. Execution times for the different tasks were measured on the control computer, and a choice of priorities to set the tasks at in QNX was found that guarantees that timing properties are not violated.

Results for a CACC run on the Berkeley test track are presented below. The speed limit on the Berkeley track is 25mph, so the scenario presented is suitable for Stop-and-Go conditions.

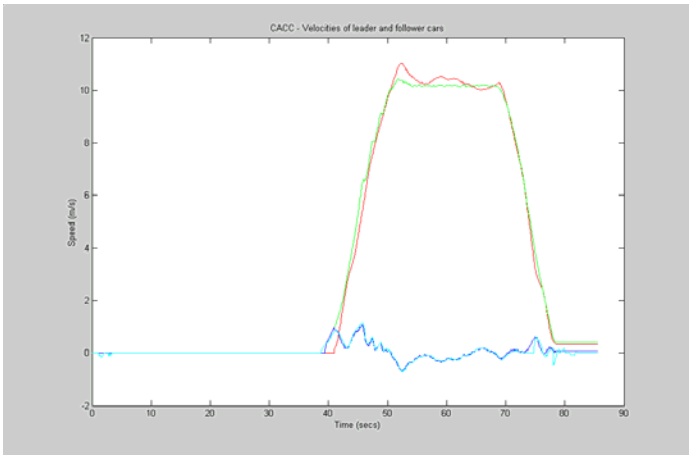

Figure 7. Results of CACC cruise control run, on test track. Green line indicates velocity of lead car, red velocity of follower car, blue lines indicate relative velocity as obtained from the communications and radar filtering.

The vehicle speeds match well, especially when the discontinuous nature of the speed profile is taken into consideration. A constant range policy was used for this particular low-speed test and the range between the vehicles was maintained at 15 meters throughout the test.

The V2V libraries are available from: http://robotics.eecs.berkeley.edu/~anouck/mobies.html

V. Conclusions

This paper presents a review of modeling, analysis and verification of complex systems and describes the use of a model-based approach to the development of real-time, embedded, hybrid control software for ACC applications. The models and controllers that have been developed are public-domain and have been used by several leading universities and research groups; hopefully, those models or their future generations will continue to help bridge the chasm between model-based embedded systems theory and practice.

References

[1] N. Lynch, "Distributed Algorithms", Morgan-Kaufman Publishers, Inc. San Mateo, CA, 1996.
[2] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, 8:231-274, 1987.
[3] N. Lynch, Class Notes, MIT, Computer Science Department class #6.879, 2001.
[4] B. Alpern and F.B. Schneider, "Recognizing Safety and Liveness", Distributed Computing, 2 (3):117-126, 1987.
[5] B. Powel-Douglass, "Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns", Addison-Wesley Publishing Company, 1999.
[6] http://www.wikipedia.org/wiki/Embedded_system, from Wikipedia, the free Encyclopedia
[7] M. Drew and J.K. Hedrick, "A Discussion of Vehicle Modeling for Control", Vehicle Dynamics Laboratory Technical Report, Mechanical Engineering Department, UC Berkeley.
[8] D. Cho and J.K. Hedrick, "Automotive Engine Modeling for Control", ASME Journal of Dynamic Systems, Measurement and Control, December 1989, Vol. 111, pp. 568-576.
[9] www.timesys.com
[10] Franjo Ivancic, "Report on Verification of the MoBIES Vehicle-Vehicle Automotive OEP Problem", Technical Report # MS-CIS-02-02, University of Pennsylvania, Philadelphia, PA, March 2002.
[11] N. Shankar, S. Owre and J. Rushby, "The PVS Proof-Checker: A Reference Manual", Technical Report, Computer Science Lab, SRI International, Menlo Park, CA 1993.
[12] S.J. Garland and J.V. Guttag, LP, The Larch Prover, http://www.sds.mit.edu/~garland/LP/overview.html
[13] E.M. Clarke, O. Grumberg and D.E. Long, "Model Checking and Abstraction", ACM Transactions on Programming Languages and Systems, 16 (5):1512-, September 1994.
[14] M. Abadi and L. Lamport, "Composing Specifications", ACM Transactions on Programming Languages and Systems, 15 (1):73-132, January 1993.
[15] R. Milner, "Communicating and Mobile Systems: The Pi-Calculus", Cambridge University Press, 1999.
[16] Z. Manna and A. Pnueli, "Temporal Verification of Reactive Systems: Safety", Springer-Verlag, New York, 1995.
[17] S. Yovine, "Model Checking Timed Automata", Lectures on Embedded Systems, LNCS Volume 1494, October 1998.
[18] Y. Kesten, Z. Manna and A. Pnueli, "Verification of Clocked and Hybrid Systems", Lectures on Embedded Systems, LNCS Volume 1494, October 1998.
[19] R. Alur and C. Coucourbetis, 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 (1):3-34, 1995.
[20] N. Lynch, R. Segala and F. Vaandraager, "Hybrid I/O Automata Revisited", Fourth International Workshop on Hybrid Systems, Computation and Control (HSCC), LNCS Volume 2034, Springer-Verlag, 2001.
[21] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee, "Modular Specifications of Hybrid Systems in CHARON", Hybrid Systems: Computation and Control, LNCS Volume 1790, pp. 6-19, 2000.
[22] A. Chutinan and B. H. Krogh, "Computational Techniques for Hybrid System Verification", IEEE Transactions on Automatic Control, 48 (1):64-75, 2003.
[23] Ashish Tiwari, "Approximate Reachability for Linear Systems", Proceedings of Hybrid Systems: Computation and Control (HSCC), LNCS Volume 2623, Springer-Verlag, 2003.
[24] A.B. Kurzhanski, P. Varaiya, "Ellipsoidal Techniques for Reachability Analysis", Proceedings of Hybrid Systems: Computation and Control (HSCC), LNCS Volume 1790, Springer-Verlag, 2000.
[25] T.A.Henzinger, P.H. Ho and H. Wong-Toi, "Hy-Tech: A Model Checker for Hybrid Systems", Software Tools for Technology Transfer, 1:110-122, 1997.
[26] M. Branicky, "Studies in Hybrid Systems: Modeling, Analysis and Control", Ph. D. Thesis, EECS, MIT, Cambridge, MA, 1995.
[27] John Lygeros, Claire Tomlin and Shankar Sastry, "Controllers for Reachability Specifications for Hybrid Systems", Automatica, Special Issue on Hybrid Systems, March 1999, pp. 349-370
[28] http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/stateflow.shtml
[29] http://www.teja.com

Acknowledgments

The authors would like to thank Anupam Pathak who implemented the simplified V2V libraries in TEJA as part of his work on the MoBIES project in the summer of 2002, and Stephen Spry for his assistance in controller development and data collection for figure 7. Figure 6 is courtesy of Gerald Stone and PATH publications.