

# From Hybrid Systems to Embedded Software

Rajeev Alur  
University of Pennsylvania

## Motivation

Embedded processors are everywhere in today's society spanning a vast array of engineered products across diverse industries including automotives, avionics, consumer electronics, medical devices, robotics, and telecommunications. Embedded processors continue to become smaller, cheaper, faster, and networked, and can be the key enablers for sustaining the ongoing technological revolution [1, 17]. Realizing this potential, however, will be predicated upon our ability to produce *embedded software* that can effectively harness the functionality of sensors and processors. Embedded software is different, and more demanding, than the typical programming applications in many ways. First, in many embedded applications, embedded software is constantly interacting with continuously evolving environment. In this context, the dynamics of the environment and the timely reaction by the software is of utmost importance for correct functionality. Second, resources such as memory, time, power, devices, and communication bandwidth are limited and cannot be disregarded in embedded programming. Since resource properties are notoriously non-compositional—two components working correctly in isolation may not form a working system, integration of system components is a significant bottle-neck in embedded systems development. Third, many embedded applications are safety critical demanding higher reliability. Absence of formal guarantees with respect to correctness requirements, while acceptable for word processing software, is an impediment in domains such as avionics and medical devices. Modern programming languages abstract away from real time and resources, and do not provide adequate support for embedded applications. Consequently, current development of embedded software requires significant low-level manual effort for scheduling and component assembly. This is inherently error-prone, time-consuming and platform-dependent. Consequently, developing novel programming and implementation methodology for synthesizing portable, predictable embedded software is an important challenge for software science.

## Hybrid Systems Modeling

A model-based design paradigm, with its promise for greater design automation and formal guarantees of reliability, is particularly attractive for embedded software. The appropriate mathematical model for embedded software systems is *hybrid systems* that combines the traditional state-machine based models for discrete control with classical differential- and algebraic-equations based models for continuously evolving physical activities [2]. For example, a high-level model of a cruise controller describes the many possible discrete modes of operation (such as manual and automatic) along with the laws for computing the control output (such as throttle pressure) as a function of inputs (such as speed) and the discrete switches among modes. There are two trends which support the argument for using hybrid systems paradigm for embedded software design. First, contemporary industrial control design already relies heavily on tools for mathematical modeling and simulation. Tools such as Simulink (see <http://www.mathworks.com>) are popular in industrial design, and the models built by such tools can appropriately be formalized using hybrid systems theory. Second, hybrid systems is gelling as an academic discipline that brings together computer scientists and control theorists (for example, see the proceedings of the annual workshop on *Hybrid Systems: Computation and Control* [4, 16, 10]). Topics such as safety verification of hybrid systems, compositional modeling of hybrid systems, and design of hybrid controllers,

are being investigated by many research groups leading to tools for modeling and analysis using hybrid systems. For example, in the CHARON project at the University of Pennsylvania (see <http://www.cis.upenn.edu/mobies/charon/>) [3], we have developed a hierarchical modeling language that allows modular description of interacting hybrid systems, and tools for simulation and safety verification using reachability analysis augmented with abstraction techniques.

## From Models to Software

Generating embedded software directly from high-level models, such as hybrid systems, is appealing, but challenging due to the wide gap between the two. In current practice, this gap is bridged with significant manual effort by exploiting the run-time support offered by operating systems for managing tasks and interrupts [14, 8]. Even Real-time Java, a modern language for real-time applications, gives more control to the programmer by exposing the scheduling interface, but requires fairly low-level programming with threads, interrupts, and synchronized methods (see [www.rtj.org](http://www.rtj.org)). There is no programmer-level abstraction of the real-time instances at which the environment is sampled, the computation is performed, and outputs for the actuators are produced. As a result, the precise behavior of the program is platform dependent rendering the design process time-consuming and error-prone (c.f. [15, 17]).

Even though modeling tools such as Simulink support automatic code generation from the model, the emphasis has been performance-related optimizations, and many issues relevant to correctness are not satisfactorily addressed. First, the precise relationship between the model and the generated code is rarely specified or formalized. Second, the continuous blocks are either ignored, or discretized before code generation. Finally, code generation typically means generation of tasks, and does not incorporate scheduling. Consequently, the correspondence between the model and the code is lost, and analysis results established for the model are not meaningful for the code. It should be emphasized that this is not a theoretical problem for believers in formal approaches, but a practical issue causing unexpected failures (for example, a correctly functioning system may fail simply because of increase in the speed of the next-generation embedded processor).

We predict that in the near future a significant amount of research will be devoted to the problem of generating software automatically from hybrid models. There is a great industrial demand for tools for design automation of embedded software, and there are many new research problems that need to be articulated and solved to achieve this. We conclude this position statement by discussing two research issues.

## Programming Abstractions

One promising research trend is to develop programming abstractions where time is a central concept. For instance, an elegant higher level abstraction is promoted by the family of languages for synchronous reactive programming such as ESTEREL and LUSTRE [7, 11]. An ESTEREL programmer assumes that computation as well as communication takes zero time: at every logical event, the input stimuli arrive from the environment, the program computes the response in zero time, and waits for the next logical event. In this synchronous abstraction, the programmer is oblivious to scheduling concerns, and the compiler makes sure that the computation required to produce the reaction can be performed fast enough before the next input arrives. This makes compilation difficult, and researchers have investigated “asynchronous” extensions (for example, Globally Asynchronous Locally Synchronous Systems) which inherit the problems of programming with threads and interrupts. An alternative abstraction is the *Fixed Logical Execution Time* (FLET) assumption used in the Giotto project [12, 13]. In this model, the programmer assigns fixed execution times

to blocks of code. If a block of code has period  $\Delta$ , then every  $\Delta$  seconds the inputs are read, the code executes sometime during the period (based on scheduling algorithm), and outputs the values only at the end of the period using explicit buffering. This time-aware programming allows predictable execution that is independent of the scheduling and platform characteristics. Both these approaches, however, do not support continuous dynamics, and the traditional separation between the continuous modeling necessary to derive the control laws and sampling rates, and the discrete expression of the controller in a programming language, still exists. We have been exploring using the continuous-time semantics as a programming model. In particular, our recent case study involves compiling a model for coordinated motion of legs for walking onto Sony’s AIBO robot platform [5].

## From Model Properties to Software Properties

A key challenge to systematic software synthesis from hybrid models is to ensure that one can infer properties of the software from the properties of the model. This is inherently difficult due to the semantic mismatch between the continuous-time models and discrete-time software. However, the problem is not hopeless, and by defining a series of abstractions, one can quantify the possible discrepancies. We conclude with a brief discussion of the issues involved.

Traditionally, control theory and related engineering disciplines have addressed the problem of designing robust control laws to ensure optimal performance of systems with continuous dynamics. For example, given system dynamics  $\dot{x} = f(x, u)$ , where  $x$  represents the system state and  $u$  represents the control input, one can design a control law  $u = g(x)$  with respect to the given specification (c.f. [6, 9]). To implement this control law, one must first determine the sampling period  $\Delta$ . The software, then, is a task that consists of “sense  $x$ ; compute  $u = g(x)$ ; send  $u$  to actuators,” which must be scheduled every  $\Delta$  time units. Compared to the mathematical model  $\dot{x} = f(x, u); u = g(x)$ , the behavior of the generated code may be different for many reasons: the system state  $x$  may not follow the model  $\dot{x} = f(x, u)$  precisely, sampling introduces discretization errors, and there may be numerical errors in computing  $g$ . However, the discrepancy can be bounded if we assume that the system state follows the model closely, there is a bound on numerical errors, and the control law is robust.

Now consider a system with two modes. Initially the system is in mode  $M_1$  with dynamics  $\dot{x} = f_1(x, u)$ . It can stay in the mode  $M_1$  as long the invariant  $a(x)$  holds, and switches to mode  $M_2$  if the condition  $p(x)$  holds. The dynamics is  $\dot{x} = f_2(x, u)$  in mode  $M_2$ . Suppose we design the controllers  $u = g_1(x)$  and  $u = g_2(x)$  for the two modes separately. The software corresponding to this controller samples the system state  $x$  every  $\Delta$  time units. It has a mode variable which is initially  $M_1$ , and updated to  $M_2$  if  $p(x)$  evaluates to true. The control output  $u$  is computed based on the value of the mode variable. In terms of the discrepancy between the high-level model and the code, there are many sources of errors. Switching errors, in particular, can cause significant problems, as there is no general theory of approximation and robustness of controllers in presence of switching, and if a switch is missed, the resulting trajectory can be entirely different. If the invariant  $a(x)$  of a mode and the guard  $p(x)$  of a switch out of this mode overlap for a duration greater than the sampling period, then the code will not miss the switching event. It is worth noting that this requirement implies that the model is inherently non-deterministic: semantically, the switch may happen at any time in the duration for which the invariant and the guard overlap. This is in contrast with the hypothesis that modeling languages for reactive systems should have deterministic reactions to external inputs to be implementable (c.f. [7, 11, 15]).

## References

- [1] Embedded everywhere: A research agenda for networked systems of embedded computers, 2001.
- [2] R. Alur, C. Courcoubetis, T.A. Henzinger, and P. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, volume LNCS 736, pages 209–229. Springer-Verlag, 1993.
- [3] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1), 2003.
- [4] R. Alur, T.A. Henzinger, and E.D. Sontag, editors. *Hybrid Systems III: Verification and Control*. LNCS 1066. Springer-Verlag, 1996. Proceedings of the Third International Workshop.
- [5] R. Alur, F. Ivancic, J. Kim, I. Lee, and O. Sokolsky. Generating embedded software from hierarchical hybrid models. In *Proceedings of the ACM Symposium on Languages, Compilers, and Tools for Embedded Systems*, 2003.
- [6] P. Antsaklis and A. Michel. *Linear Systems*. McGraw Hill, 1997.
- [7] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: design, semantics, implementation. Technical Report 842, INRIA, 1988.
- [8] G.C. Buttazo. *Hard real-time computing systems: Predictable scheduling algorithms and applications*. Kluwer Academic Publishers, 1997.
- [9] C.T. Chen. *Linear System Theory and Design*. Oxford University Press, 1999. 3rd Edition.
- [10] M. Greenstreet and C. Tomlin, editors. *Hybrid Systems: Computation and Control*. LNCS 2289. Springer, 2002.
- [11] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.
- [12] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.
- [13] T.A. Henzinger and C.M. Kirsch. The embedded machine: Predictable, portable, real-time code. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 315–326, 2002.
- [14] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 2000.
- [15] E.A. Lee. What’s ahead for embedded software. *IEEE Computer*, pages 18–26, September 2000.
- [16] N. Lynch and B.H. Krogh, editors. *Hybrid Systems: Computation and Control*. LNCS 1790. Springer, 2000.
- [17] S. Sastry, J. Sztipanovits, R. Bajcsy, and H. Gill. Modeling and design of embedded software, 2003.