Chapter 20

REFINEMENT OF HYBRID SYSTEMS *

From Formal Models to Design Languages

Jan Romberg¹, Christoph Grimm²

¹ Systems & Software Engineering, TU München, Germany
² Technische Informatik, J. W. Goethe-Universität Frankfurt am Main, Germany

Abstract System-level design of discrete-continuous embedded systems is a complex and error-prone task. While existing design languages like SystemC and its extension to the mixed-signal domain, SystemC-AMS, are supported by tools and libraries, they lack both mathematical precision and intuitive, abstract design notations. Graphical design notations with formal foundations such as HyCharts suffer from the lack of tool support and acceptance in the developer community. To overcome the deficiencies of both approaches, we present a design flow from graphical HyCharts to SystemC-AMS designs. This design flow uses a formally founded refinement technique to ensure the overall consistency of the design.

1. Introduction

Embedded control systems are frequently characterized as a mixture of continuous and discrete behaviors. We call such systems discrete-continuous or hybrid systems. Because the discrete part of a hybrid system introduces discontinuities in the system’s evolution, the behavior of hybrid systems tends to be difficult to predict and analyze. For the task of high-level design of a control system, it is highly desirable to use representations that accurately reflect both continuous and discrete

*This work was supported with funds of the Deutsche Forschungsgemeinschaft under reference numbers Br 887/9 and Wa 357/14-2 within the priority program Design and design methodology of embedded systems.

behavior. In the early stages of a design, this frees the developer from considering implementation details like quantization and sampling, and allows designers to concentrate on the essential features of the design.

Formalisms like Hybrid Automata [1], or HyCharts [11] are well suited for precisely capturing the continuous/discrete behavior of a hybrid system. An advantage of these formalisms is that, being based on formal semantics, models are susceptible to automated analysis and formal verification. However, as most designs are implemented using digital hardware, there is currently a gap between the capturing and verification of an abstract design in mixed discrete-continuous time, and the discretized design and implementation of such systems. For the later design phases, discrete approximations of the hybrid model that explicitly consider quantization and sampling effects are more appropriate.

Hybrid systems combine continuous behavior specified by differential equations with discontinuities introduced by discrete switching logic. With discrete hardware being pervasive in embedded systems, the prevalent way of simulating and implementing hybrid systems is based on discrete-time or discrete-event algorithms. Numerical solvers are one example for such discrete algorithms; a variety of variable- and fixed-step algorithms are successfully used for continuous-time simulation, and their capabilities and limitations are well-understood [9]. For implementation or large simulations, simple fixed-step algorithms like Euler forward with quasi-constant performance are widespread.

The effects that introduce deviations to the ‘ideal’ behavior of a realization (or simulation) can be roughly characterized as:

- Quantization and limitation of the variable’s values.
- Quantization of the time. Modeling smooth changes of analog functions would require an infinite number of events/process activations. This is approximated by activation at a finite number of discrete time steps.

It has been recognized by numerous authors [9] that simulation or real-time computation of hybrid system across discontinuities may cause large errors when the design is discretized ad-hoc. For component-level simulation, variable-step algorithms offer good results; however, simulation performance is generally unacceptable for larger (system-level) designs.

SystemC-AMS [5] offers support for mixed solvers for continuous, hybrid, and discrete designs. Cyclic dependencies between different solver kernels are broken into acyclic structures by introducing a sufficiently small delay. The resulting acyclic structure is then ordered in the direction of the signal’s flow. After that, the outputs can be computed successively from already known inputs by executing a step in each solver