
Abstract
ET: Enhanced Tool
The effectiveness of state/mode diagrams in supporting the design of embedded systems has been demonstrated. An enhanced tool is described for the design of embedded systems based on mapping each nested state/mode to a C++ class with methods corresponding to discrete approximations to analogue control equations. Examples are provided confirming the effectiveness of its support for embedded systems in design.

Why ET?
This is not the first compiler tool ever, nor is it the first place where hybrid modes were proposed. However, the authors have created a compiler that generates C++ code from a high-level language that is very convenient for representing the architectural and behavioral hierarchy of a control system. The notation (Charon) already existed, as well as the notion of approximating an analogue function with discrete steps, as well as the model of a system moving through sets of feasible states (governed by constraints and invariants). So the main contribution of this paper seems to be the code generator that takes this high-level language and generates C++ classes.

Question - [Method/means of development]
How can we automate the production of systems embodying hybrid modes?

Results - [Tool / notation]
The authors’ new compiler for the Charon language/environment represents each mode as a C++ class. The class has a method for each of the mode’s entry points, as well as a method for each of the mode’s exit points. These methods can access variables located in the architectural hierarchy representing the agents comprising the overall thing to be controlled. (The methods are actually objects with read/write methods so that the backing store can be something besides just memory… such as an IOCTL interface, etc.)

The compiler also generates a loop, similar to the event loop in a traditional C Windows program, which continually checks to see what guards are satisfied on transitions out of active modes. When permitted by invariant conditions, it also advances logical time in steps and updates variables using constraints, but mode switches are permitted to occur in zero logical time. In this way, the generated program maintains the set of currently active states as well as the values of variables in all the agents.

The authors provide a formal definition of each agent and mode, which implicitly specifies the space of programs that can be specified using the language. The compiler essentially maps elements of this space to sets of C++ classes as noted above. The authors do not actually prove that this mapping produces a program with behavior semantically equivalent to the corresponding program specification, but such a proof wouldn’t really be any different than the proof for any other language compiler. The authors state a theorem essentially saying that if there is a feasible path for the specified program (in a mathematical sense), then the generated C++ simulator will find a path, provided that the time step is small enough that the discrete approximation wouldn’t cause the system to overshoot the next feasible state from the current state.

One way to visualize this is to see the feasible states as points in a space, and these points are joined into curvy (analogue) pipes of feasible volume. The current state is a point in a pipe. The discrete approximation means that the current state moves (nondeterministically) in choppy steps through the pipes, roughly parallel to the pipe. But if the steps are too big, the current state will penetrate the wall of the pipe and go out into the infeasible region outside the pipes. The solution is to take small enough steps that the current state never unnecessarily penetrates the pipe wall.
Validation – [Experience]
The authors try out the notation and compiler on a real problem, the specification of leg movements for an Aibo robodog. They never say it explicitly, but it appears that the dog is successfully able to walk.