Code Generation for Hierarchical Systems

Gerd Behrmann, Kåre Kristoffersen, Kim G. Larsen

September 6, 1999

Abstract

We evaluate different approaches towards code generation for hierarchical state machines. Our goal is to create very compact, high performance code suitable for integration as the control program in large software systems or storage in ROM as the control program for a hardware device.

Hierarchical state machines, pioneered by David Harel in 1987, are widely used in specification, design, and modelling of software and hardware systems. An industry partner recently asked us how compact code for their state chart variant used in the visualSTATE integrated development environment for embedded software could be generated. The current version of the tool translates the hierarchical description to an equivalent, although much larger, flat system and uses the code generator used in previous versions of visualSTATE.

We will very briefly introduce a stylized variant of the visualSTATE modelling language called HSEM and describe our table driven interpreter for it. We will then summarize different variations currently under evaluation.

A hierarchical state event machine (HSEM) is a nested structure of states, providing parallel and sequential composition of sub-states. The model includes a history feature which remembers the last activated sub-state. Transitions (which include multi level transitions) between states are guarded by an event and a boolean condition over state synchronizations, and can be associated with a multiset of actions to be generated when the transition is executed. States are associated with entry and exit actions, which are generated on activation and deactivation, respectively.

When generating code for embedded systems, we have to consider both RAM, ROM and CPU usage. In general RAM is much more expensive than ROM and is a scarce resource. ROM on the other hand is slower than RAM.

Our code generation approach translates an HSEM into a number of tables describing the nesting structure and the transitions of the system. These tables are not modified at run-time and can be stored in ROM. The current configuration and a working copy of it are stored in RAM together with a handful of variables used by the interpreter.

The system is processed by an interpreter at runtime. The interpreter does not directly depend on the processed HSEM, although we choose from a number of different interpreter implementations, depending on the nature of the HSEM. For instance, we might choose different orderings of the elements in the tables to optimize speed. The basic approach is the following: The interpreter receives an event and identifies the activated transitions. The transitions are executed on the fly as they are encountered and the system configuration is updated. In the process, we need to take care of executing output actions associated with the transition and the states being (de)activated in the proper order.

One experiment is to change the representation of the system configuration. A straight forward approach is to represent it as a n-tuple, where n is the number of or-states (an or-state is a sequential composition of the sub-states in it; similar to a traditional finite state machine). But since the degree of concurrency in the system varies, it would be sufficient to represent the configuration as an m-tuble, where m is the largest possible number of concurrent or-states. This can be found using either a syntactic analysis or state space exploration techniques. Unfortunately, this makes evaluating the conditions on transitions more CPU intensive.

To address this we propose a technique that reuses (caches) the truth value of a condition. Briefly, we precompute the set of conditions that are influenced by an activation or deactivation of a state. This information is stored in ROM. This information is used to invalidate the stored thruth value of a condition and which is recomputed when needed. We believe that most transitions only slightly change the configuration of a system. If this is true, the number of invalidated transitions is small and hence the number of reused truth values large.