

Real-Time Statechart Implementation Notes

Sven Burmester

Introduction This document gives some notes about the implementation of Real-Time Statecharts. Here some answers to questions are given, which have not been described in [1] (german) and [2].

Representation of current state Each state is represented as an integer constant. The current state is implemented by the attribute `currentState`. Due to hierarchy and parallelism¹ multiple states can be the current state. Therefore `currentState` is not an integer attribute, but a matrix. The matrix consists of as many columns, as parallel states are active (so the number of columns changes at run-time). The rows represent the hierarchy in each parallel state.

Determining and selecting activated transitions In each period, the runtime framework calls the method `handleTransitions()` of the class, implementing the Real-Time Statechart. This method calls the method `addActivatedTransitions()` in order to add all activated Transitions to the Matrix `enabledTransition`. This is done within the method `calculateActivationTime`, which is called for each transition of the current states. Note that due to hierarchy and parallelism multiple states can be active.

The next step within the method `handleTransitions()` is to determine the transition or the transitions resp., which have been activated first. These transitions are fired – either by calling the `run` method directly or by starting an aperiodic thread. In both cases the `run(int transition)` method is called. The parameter indicates which transition has to be fired.

Firing transitions, respecting history, hierarchy, and parallelism Due to hierarchy and parallelism a state `s` can contain zero, one or multiple embedded statecharts. The start states indicates which state or –in case this statechart contains further hierarchic or parallel states– which states become valid if `s` is entered. Therefore an attribute matrix `init` is synthesized which describes for each statechart the initial state(s). In case of deep or shallow history, the initial states change. Therefore the attribute matrix `history` is synthesized. Contrary to `init` the valuation of the elements of `history` change at run-time.

When a transition is fired by calling the method `run(int transition)`, first the highest hierarchic state, that is exited is determined. Due to parallelism this state can be member of multiple columns. All of these columns are blocked, which means that no further transitions from these columns can fire. These blocked columns are *noted* in `notedCols`.

Then all `exit` methods of the exited states are called. The exited states are saved in the history matrix. The next step is to remove all exited states from the `currentState` matrix. After raising the events and executing the side-effects, the target states are entered. Again, the reason for entering multiple states are hierarchy and parallelism. Similar to the exit of states, first the highest state, which is entered is determined.

Then the local attribute matrix `newStates` is filled with all states which are entered. In case of history in the embedded statecharts, this is based on the history matrix attribute. Finally all `entry` methods of the states to be entered are called and the according states are added to `currentState`. Finally the clocks are reset and the point in time when the target location(s) are reached is saved.

References

- [1] S. Burmester. Generierung von Java Real-Time Code für zeitbehaftete UML Modelle. Master's thesis, Universität Paderborn, Paderborn, Deutschland, September 2002.
- [2] S. Burmester and H. Giese. The Fujaba Real-Time Statechart PlugIn. In *Proc. of the Fujaba Days 2003, Kassel, Germany*, October 2003.

¹Parallelism means *orthogonality* in the UML 2.0 terminology