The Fujaba Real-Time Statechart PlugIn

Sven Burmester* and Holger Giese†
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[burmi|hg]@upb.de

ABSTRACT

Distributed embedded real-time systems are one of the most successful application areas of the UML. However, the UML techniques for behavior modeling such as Statecharts in their current form do not support real-time as required, because of the unrealistic underlying zero-time execution assumption for side-effects. With Real-Time Statecharts, a related extension has been developed for the Fujaba Tool Suite that overcomes these limitations by supporting a well-defined real-time semantics based on Timed Automata and code synthesis which guarantees the specified timing characteristics. Besides the Real-Time Statecharts the paper describes the currently available tool support and the underlying principles of the code generation for the currently supported platform, Real-time Java.

Keywords

Statecharts, Real-Time, Embedded Systems, UML, Fujaba.

1. INTRODUCTION

Today, the *Unified Modelling Language (UML)* [13] is successfully applied to model complex embedded systems. However, the standard UML behavior modeling techniques such as Statecharts are not appropriate in their current form. For distributed real-time systems, the underlying zero-time execution assumption for side-effects is often unrealistic and conflicts with a consistent implementation of the high level UML model on available hardware and software platforms.

With Real-Time Statecharts [4, 6], a related extension has been developed for the Fujaba Tool Suite that overcomes these limitations by supporting a well-defined real-time semantics based on Timed Automata and code synthesis guaranteeing the specified timing characteristics.

The tool support currently available for Real-Time Statecharts in form of a Fujaba PlugIn consists of an extended diagram notation, a time consistency checker, and code synthesis. Additionally, a function permits to check whether multiple Real-Time Statecharts can be scheduled on a single node using the timing information extracted from the real-time UML model.

*Supported by the International Graduate School of Dynamic Intelligent Systems. University of Paderborn

†This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and founded by the Deutsche Forschungsgemeinschaft.

In Section 2 we will first review the requirements for real-time modeling with UML and the shortcomings of the current UML support. Then, Real-Time Statecharts are described informally in Section 3. Section 4 guides how to use the Real-Time Statecharts PlugIn for the Fujaba Case-Tool. The features of Real-time Java are described in Section 5 before the mapping from the Real-Time Statecharts to Real-time Java code is shown in Section 6. Section 7 draws a conclusion and gives a perspective on future work.

2. REAL-TIME MODELING

Developing software is divided in a phase of design and a phase of implementation. During the design phase, the structure and the behavior of the software are specified by appropriate modeling languages such as UML, constituting a standard for modeling these different aspects. A developer can use Object- and Class-diagrams to model the structure and Statechart- and Activity-diagrams to specify the behavior.

For the specification of software for embedded real-time systems, a number of object-oriented approaches [14, 1, 5, 7] including ROOM [14] have been proposed. As most ROOM concepts have been integrated into the UML 2.0 proposal of the main tool vendors [13], they are likely to soon become a part of the standard UML and will therefore be widely available. However, these concepts do not address the temporal behavior of the operational model and therefore do not improve the situation when it comes to the automatic code generation.

Another thread of development is the *UML Profile for Schedulability, Performance, and Time* [12]. The profile defines general resource and time models which are used to describe the real-time specific attributes of the modeling elements such as schedulability parameters or quality of service (QoS) characteristics. Besides an abstract *logic model*, a more concrete *engineering model* can be specified by using these extensions. The engineering model is later used for the required model analysis and code generation. However, it remains an open question in the UML profile how all required details of the engineering model are determined.

Therefore, the current practice when building embedded software with hard real-time constraints is to specify the software on a high abstraction level, then to partition it to make it run on a real-time operating system in concurrent threads (usually without adequate analysis), to implement it, to test if the time restrictions hold and then usually to re-partition it. Repeating this cycle a number of times is usually very costly but mostly unavoidable.

Typically, an embedded software application consists of several concurrent running threads[1], often these threads are periodic. When the application is employed in the real-time domain, the time when each thread completes is of crucial importance. The longest acceptable duration until the completion of a thread is designated by the *deadline*. In order to accomplish a schedulability analysis, the so called *worst case execution time (WCET)* of every thread has to be known. This time characterizes the upper bound of the possible duration of the thread, if it is executed on the processor of the underlying computer system without preemption [10].

Unfortunately, UML Statecharts do not allow the integration of these important attributes. The only way to bring time into UML Statecharts is the use of the so called *After-* and *When-* constructs. These constructs can be used to model temporal behavior, but are not sufficient to specify real-time behavior [4, 6]. The underlying zero-time execution assumption cannot be fulfilled in a distributed setting, as the required side-effects as well as the emitting of messages always require some minimal amount of time. Another weakness is that a reasonable real-time semantics for Statecharts including the side-effects is therefore not possible.

Another approach for modeling temporal behavior are *Timed Automata* [8, 11]. This kind of automata can be used to specify real-time behavior in a well-defined manner, dependent on clocks, but is very restricted in the output.

Real-Time Statecharts combine the advantages of Statecharts with those of the Timed Automata and add some constructs and restrictions which allow the user to specify real-time behavior and generate proper real-time code that ensures the specified timing properties. This extension overcomes the limitation of Statecharts w.r.t. real-time systems by supporting a well-defined real-time semantics based on Timed Automata [6].

3. REAL-TIME STATECHARTS

In the first part of this section, the syntax and semantics of Real-Time Statecharts are explained informally. Section 3.2 introduces the problem of time conflicts.

3.1 Syntax and Semantics

Figure 1 depicts a Real-Time Statechart. It consists of states and transitions, like usual Statecharts. The states are extended –compared to usual Statecharts– with the following annotations: Time invariants, clock resets associated with `entry()`- and `exit()`-Methods, WCETs for the `entry()`-, `do()`- and `exit()`-Methods, and a period for the `do()`-Method.

Transitions are associated with events, guards, side-effects, time-guards, clock resets, priorities, deadlines, worst case execution times, and channels and events for synchronization.

Let the set of clocks be denoted by C, a clock by $t_i \in C$. A time invariant is of the form $\bigwedge_{t_i \in C} t_i \leq T_i$, $T_i \in N \cup \{\infty\}$ and delivers the point in time when the specific state has to be left via a transition.[2] If the invariant does not contain a clock $t_j \in C$, this part is assumed to be $t_j \leq \infty$ and is omitted in the graphical representation.

[1] *Threads* are often designated as *Tasks* or *Processes*

[2] For all time annotations the unit of time has to be the same. For reasons of readability, it is disregarded in the formulas.

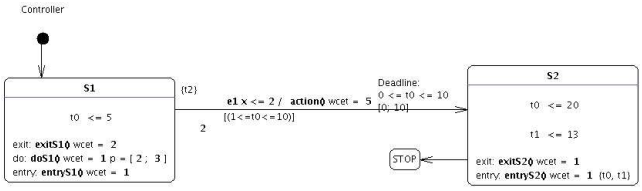


Figure 1: Real-Time Statechart

In Figure 1, the invariant of state S_1 is $t_0 \leq 5$ and the one of S_2 is $t_0 \leq 20 \wedge t_1 \leq 13$. The `entry()`-method is executed when a state is entered, the `exit()`-method before the state is exited. They have a WCET of 1 resp. 2 msec. Assigning clocks to these operations resets them at the moment of entrance resp. exit (t_0 and t_1 in state S_2). In order to perform analysises in the time domain (schedulability etc.), the WCET for each operation needs to be known (described by the annotation `wcet` = ...). As the `do()`-operation is executed periodically while the automaton stays in the specific state, it is reasonable to annotate a period with this method. Often the user wants to specify a range for the period instead of a certain value. Thus the period for the `do()`-operation is specified by an interval ($p \in [2; 3]$ in State S_1).

A transition is triggered if the associated event (e_1) is available, the guard ($[x \leq 2]$) and the timeguard ($[1 \leq t_0 \leq 10]$) evaluate to true. The timeguard is of the form $\bigwedge_{t_i \in C} a_i \leq t_i \leq b_i$, $a_i \in N$, $b_i \in N \cup \{\infty\}$, $a_i \leq b_i$. Similar to the invariants, a timeguard is assumed to contain the expression $0 \leq t_j \leq \infty$ if no interval is specified for $t_j \in C$ and is omitted in the graphical representation.

When the transition fires, all clocks denoted in the set of clock resets are reset to 0 (t_2 in the example) and the sideeffect is executed (`action()`). When a transition is triggered, it fires. The firing will not be delayed. This behavior is generally denoted as *urgent* behavior.

If multiple transitions are activated, the one being triggered first fires. For the case that multiple transitions are triggered at the same time and that they are mutual exclusive, priorities have been introduced (In the example the priority is 2, a priority of 1 will be omitted in the graphical representation). *Only if multiple transitions, being mutually exclusive, are triggered at the same time, the one with the highest priority of these transition fires.* If the Statechart is in a parallel AND-state, it is possible that multiple transitions that are not mutually exclusive are triggered at the same time.

In addition to that, a worst case execution time ($\mathsf{wcet} = 5$) and a deadline are associated to a transition. The deadline is split into the relative and the absolute part. The relative part is of the form $[d_{low}, d_{up}]$ and describes that the switching (the execution of the transition) has to be finished at least d_{up} and at the earliest d_{low} after being triggered. The absolute part is depicted by a term of the form $\bigwedge_{t_i \in C} t_i \in [d^i_{low}, d^i_{up}]$ and describes the lower and upper bounds dependant on the clocks ($[0, \infty]$ is the default interval for both parts). In the example, the deadline is $[0, 10] \wedge t_1 \in [3, 11]$.

There exist 3 different types of synchronization: *External* synchronization by enqueing events, *internal* synchronization, and synchronization via *shared resources*. Internal

synchronization makes sense when the Statechart is in a parallel AND-state. Transitions can be associated with a channel and one of the actions `sending` or `receiving` (e.g. `a?` denotes `receiving` through channel `a` and `a!` denotes the complementary action `sending` through `a`). A transition associated with such an internal synchronization fires just when a transition with a complementary action through the same channel is triggered, too. The third possibility is the synchronization via shared resources. A shared resource is, for example, a specific memory area, that is written by some operations and read by some others. When some concurrent firing transitions are accessing the resource at the same time, the effect of priority inversion [10, 3] can appear and may delay the execution of one transition due to blocking effects.

Thus, the access should be controlled by a so called *monitor*, reducing the blocking time. The user is demanded to create a monitor-class in whose methods the critical sections are rolled out. When the worst case execution times of these methods are known and the monitor is associated with the operations that use the methods of the monitor to access the shared resources, then the maximal possible blocking time is considered in a scheduling analysis (see Section 4).

3.2 Timing analysis

The possibility to specify behavior, based on clocks, leads to the problem of time inconsistencies. If the user adds time annotations without care, it may lead to non-realizable behavior. Imagine a state with an invariant of $t_0 \leq x$ and leaving transitions, having a timeguard of the form $t_0 \geq x + n$, $n > 0$. So, this state cannot be left before its invariant exceeds, which will result in a so called *time stopping deadlock*. Analyzing the structure and the annotations of the Real-Time Statechart, these and other inconsistencies can be found by an algorithm.

There exist 2 different kinds of inconsistencies: A Real-Time Statechart is *inconsistent* if the specified behavior will definitely lead to problems. It is called *insecure* if it is possible, but not certain to run into problems. An example for insecureness is a state with a time invariant (e.g. $t_0 \leq x$) and leaving transitions which are triggered by events. As it is not possible to predict before runtime –without the use of a Modelchecker– if the according event occurs, it is possible for this Statechart to run into a time stopping deadlock, but not certain. A Statechart that is neither inconsistent, nor insecure is called *timeconform*. The different forms of inconsistency and insecureness are described in [4]. Some inconsistencies can be removed automatically by an algorithm, some can only be eliminated manually [4]. The user can choose on his own if he wants to use the algorithm for automatic inconsistency elimination.

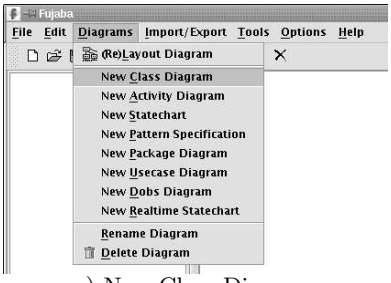
If the user intends to run multiple Real-Time Statecharts or different applications on the same target platform, he sometimes wants to specify a maximum processor load for every Statechart. This is done with the *utilization factor* (the utilization factor has a range from 0 to 1).

The sketched static analysis algorithm detects temporal inconsistencies at low costs. Some of the detected inconsistencies are removed automatically. Due to the incompleteness of the analysis, it is only a supplement to model checking but cannot, of course, replace it to detect all inconsistencies in the general case.

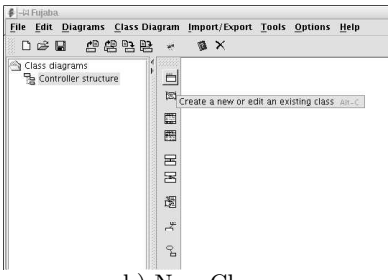
4. USER GUIDE

This section addresses users, new to Real-Time Statecharts. It gives a step-by-step instruction how to create a simple Real-Time Statechart, to check it for time consistency and to generate executable code.

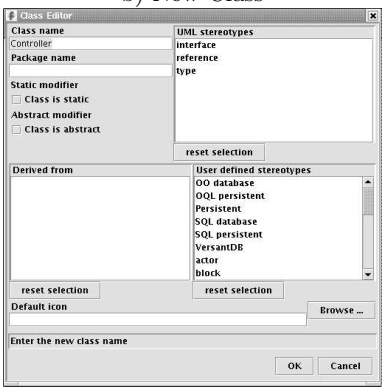
To use Real-Time Statecharts, the *Fujaba Tool Suite* and the Real-Time Statechart PlugIn are required.[3] After downloading, installing and starting Fujaba, Real-Time Statecharts can be used. As every Statechart is associated with a class, the first step before creating a Real-Time Statechart is creating a class diagram with at least one class (see Figure 2a) - c)). To do that, select the menu ,,Diagrams → New Class Diagram“ and enter a diagram name. After that choose the menu ,,Class Diagram → Create / Edit Class“ or press the first toolbar button as depicted in Figure 2 b). This will result in a dialog, where the name of the new class should be entered (*Controller* in the example). When choosing the menu ,,Diagrams → New Realtime Statechart“ (see Figure 3) the new class can be selected as the base class for the new Real-Time Statechart.



a) New Class Diagram



b) New Class



c) Edit the new class

Figure 2: Creating a Class

[3] www.fujaba.de

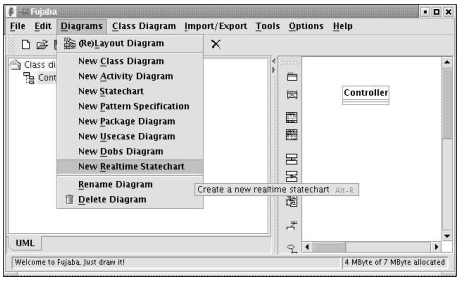


Figure 3: Creating a Real-Time Statechart

Figure 4 shows the diagram-specific menustructure and the toolbar, whoose actions can be used to edit the Real-Time Statechart.

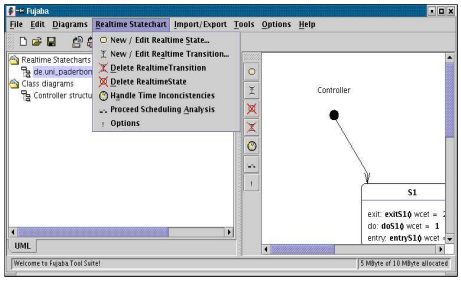


Figure 4: Menustructure and Toolbar

As a Statechart consists of states and transitions, the first two buttons on the toolbar represent the actions ,,New / Edit Realtime State...“ and ,,New / Edit Realtime Transition...“. There exist 3 different kinds of states: Start states, stop states and complex states. The start state should be unique for every Statechart. In the complex state, it is possible to specify the annotations described in Section 3.1 (see Figure 5).

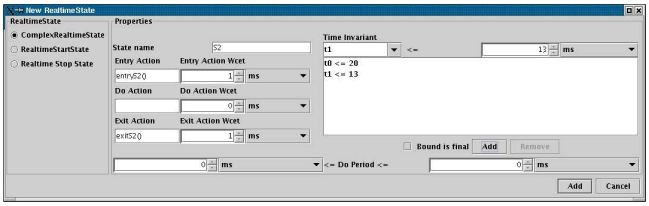


Figure 5: Dialog for States

The actions can be the call of methods, specified in the corresponding class diagram, or commands from the target language (e.g. Real-Time Java). Figure 6 depicts the dialog for transitions. For every timing attribute (deadline, invariant, timeguard), a final-flag can be set. When searching for time inconsistencies (see Sections 3.2) and an inconsistency that can be removed automatically by adjusting the specific attribute is found, this is only done if the flag is not set.

When creating new states or transitions, the user is asked to enter the worst case execution times for each operation. These declarations are used when searching for time inconsistencies (see 3.2) or performing a scheduling analysis. Additionally the WCETs of the ”simple, atomic operations”,

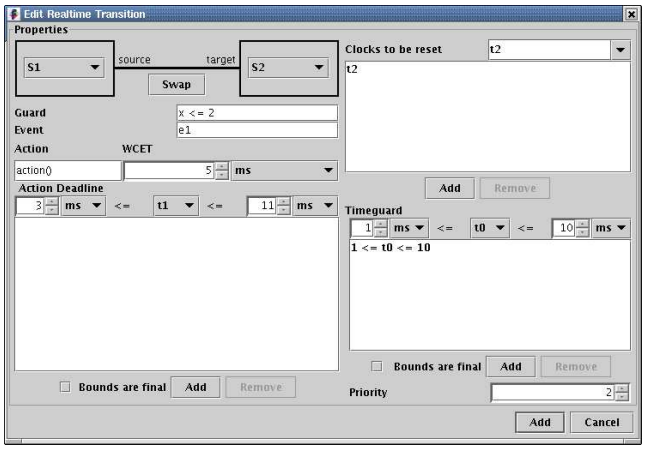


Figure 6: Dialog for Transitions

like the assignment of a long integer variable, the comparison of two integer variables and others, need to be specified in an xml-file (see Figure 7). Furthermore the WCETs of the actions can be specified in this doument, too. So in principle, it is possible to use external tools, determining the WCETs and providing this file.

```
<wcet>
  <action id="RealtimeStatechartactionTrans1"
     wcet="40" unit="ms" />

  <systemconstant name="INTEGER_ASSIGNMENT"
     wcet="1" unit="ms"/>
  <systemconstant name="LONG_INTEGER_ASSIGNMENT"
     wcet="1" unit="ms"/>
  <systemconstant name="INTEGER_ADDITION"
     wcet="2" unit="ms"/>
  <systemconstant name="INTEGER_COMPARISON"
     wcet="1" unit="ms"/>
  <systemconstant name="GET_METHOD_CALL"
     wcet="2" unit="ms"/>
  <systemconstant name="SET_OR_ADD_METHOD_CALL"
     wcet="3" unit="ms"/>
  <systemconstant name="OBJECT_ASSIGNMENT"
     wcet="3" unit="ms"/>
  <systemconstant name="TYPE_CAST"
     wcet="1" unit="ms"/>
  <systemconstant name="START_APERIODIC_THREAD"
     wcet="6" unit="ms"/>
  <systemconstant name="END_APERIODIC_THREAD"
     wcet="5" unit="ms"/>
  <systemconstant name="SLEEP_AFTER_THREAD_START"
     sleep="3" unit="ms"/>
  ...
</wcet>
```

Figure 7: wcet.xml

The path of the xml file can be set in the options dialog (see Figure 8). When generating code, a *schedule document* is generated (beside the source code files). This document contains information about all threads that are started, their WCETs and deadlines and which threads can run concurrently. These informations are needed for a scheduling analysis if multiple Real-Time Statecharts shall run in parallel on the same target platform. The destination path for creating the schedule document can be set in the options dialog as well.

In Section 3.2 it is described that a Real-Time Statechart can be timeconform, insecure or inconsistent. The radio buttons in the options panel are used to set the claimed security level. The last attribute that can be set is the utilization factor, described in section 3.2, either.

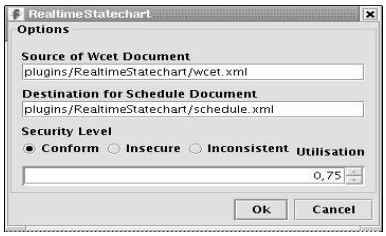


Figure 8: Options for Real-Time Statecharts

After specifying a Real-Time Statechart, the user should check for time conformity. This is done by calling the menu ,,Realtime Statechart → Handle Time Inconsistencies“ or pressing the equivalent button on the toolbar. Depending on the set security level and on the final-flags for the timing attributes (see Section 4), the inconsistencies are displayed and removed automatically.

When the desired degree of time consistency is achieved, it is possible to generate code. This is done by calling the menu ,,Import/Export → Export (All) Class(es) to Java“. Beside the source code file (that has the name of the class, associated with the Statechart), the schedule document is generated in the file specified by the options. This document is used for a scheduling analysis of multiple Real-Time Statecharts.

Even when there is just one Real-Time Statechart that should run on the target platform, the user should perform the scheduling analysis, as an additional class called *Main* is generated. The Main class contains code for assigning priorityes to the threads (important for scheduling) and starts all Real-Time Statecharts. As information about all involved Real-Time Statecharts is required, this class is not generated when calling ,,Export (All) Class(es) to Java“. For the same reason, the user is asked to add all relevant schedule documents after starting the Scheduling Analysis (see Figure 9).

After generating all target code, it can be compiled and started. For this, the `de.uni_paderborn.fujaba.umlrt.realtimestatechart.sdm`-package is required. This package is included in the `libs/sdm.jar`-file in Fujaba's PlugIn directory.

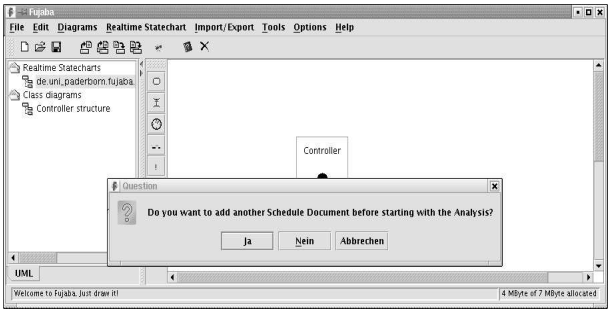


Figure 9: Declaration of Schedule Documents

5. REAL-TIME JAVA

The Real-Time Specification for Java (RTSJ) [2] provides an API, defining classes and methods, allowing the use of a real-time Scheduler and Memory Management in Java. In particular, it is possible to gain deterministic garbage collector behavior.

5.1 Scheduling

Figure 10 depicts the classes relevant for scheduling. As in every application exists at most one scheduler, the class `Scheduler` is implemented as a singleton. A Real-Time Operating System (RTOS) can provide plenty of different schedulers. One of the most common ones is the so called *Priority Scheduler*. To be conform to the specification, an implementation of the RTSJ has to deliver at least a Priority Scheduler. Of course it is possible to implement another scheduler (e.g. written in the programming language C) and use it in Java by accessing the routines of the scheduler via the Java Native Interface (JNI),[4] used in a subclass of `Scheduler`.

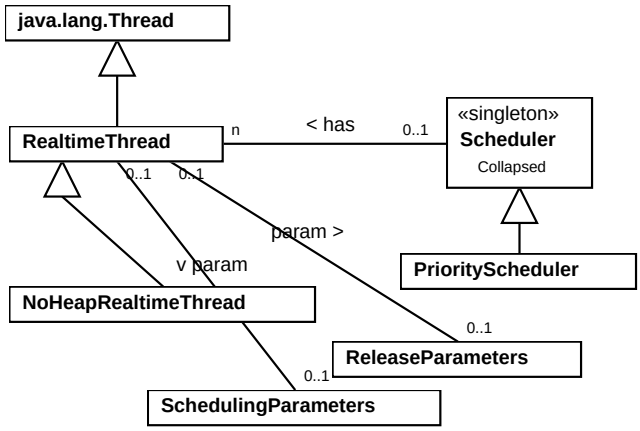


Figure 10: Scheduling

To assign the data relevant for scheduling like deadlines, periods, start time points etc. to a thread, the class `java.lang.Thread` is extended to `RealtimeThread`. Every Thread that shall be scheduled has to be registered with the instance of `Scheduler` via the `has`-association. Every `RealtimeThread` (or `NoHeapRealtimeThread`) can contain different parameters:

- `SchedulingParameters`

  This object contains just the priority of the associated thread, needed by the priority scheduler.

- `ReleaseParameters`

  The release parameters contain the WCET, deadline and the start time. If the thread is periodic, the object contains the period, too. If a deadline-miss-handler is specified, it is started at the point in time when the deadline is missed. Similarly, a cost-overrun-handler will be activated if the cost (WCET) is exhausted.

[4] java.sun.com

As it is not compulsory to provide the cost-overrun-feature, it is not available in all implementations of the RTSJ.

- `MemoryParameters and MemoryArea`

  When a `RealtimeThread` allocates new objects, they are allocated in the specified `MemoryArea` (see section 5.2). In the `MemoryParameter` object, a maximum amount of memory can be specified for the current and the immortal memory area (see section 5.2). Apart from this, it is possible to set an allocation rate in bytes per second that can be used for analysises.

- `ProcessingGroupParameters`

  On some real-time platforms, the operating system can guarantee that a thread never obtains more execution time than specified by the WCET. If the underlying operating system supports this capability and the real-time thread has a reference to a `ProcessingGroupParameter` object, the thread gets no more execution time than indicated by the cost attribute of this object.

- `java.lang.Runnable`

  If the real-time thread has a reference to a `runnable` object, the `run()`-method of this object is executed, after starting the thread, instead of the `run()`-method of the thread object.

```
public class PeriodicThread
 extends RealtimeThread
{
  public void run()
  {
    while (true)
    {
      ...
      waitForNextPeriod();
    }
  }
}
```

Figure 11: A periodic real-time thread

In Figure 11, it is shown how to implement a periodic thread. The method `waitForNextPeriod()` accesses the period attribut from the `ReleaseParameters` object and handles the coordination with the scheduler. The method is not exited before the scheduler allocates the processor for the specific thread, again.

5.2 Memory Management

Real-time Java divides the memory into different areas which are described in the following:

- Heap Memory

  The heap is the ,,traditional“ memory. Sun's virtual machines use nothing but the heap memory for dynamic memory allocation.

- Immortal Memory

  Instances allocated in the immortal memory area are never erased. It is usefull to place objects there existing during the whole time the virtual machine runs. Dynamic structures should not be allocated in this area.

- Scoped Memory

  Heap and immortal memory are implemented as singletons, but it is possible to create multiple scoped memory areas. A scope memory area has a so called *reference count*. This reference count is incremented every time a new instance is allocated and decremented when an instance is dereferenced. If a dereferenced object is placed in the heap memory, it will be freed and erased the next time the garbage collector runs. Being in a scoped memory, it will not be freed before the reference count for the whole scope drops to zero. At the moment when it becomes zero, the garbage collector is executed by the active real-time thread.

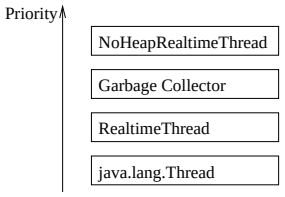


Figure 12: Priorities

In Figure 10 in Section 5.1, the `NoHeapRealtimeThread` is shown. Instances of this class have a higher priority than the garbage collector (see Figure 12). If the garbage collector runs and such a thread is started, it interrupts the garbage collector. Conversely, a `NoHeapRealtimeThread` will never be interrupted by the garbage collector. An ,,ordinary“ real-time thread has a lower priority than the garbage collector. So, real-time behavior can only be achieved with the use of these threads if the application abdicates on dynamic structures and the garbage collector is never started.

As mentioned above, every real-time thread is associated with a memory area. If a thread allocates new objects, they are allocated in its memory area.[5] Alternatively, objects can be placed in the different areas via the use of reflection. Therefore methods like `MemoryArea.newInstance(...)` or `MemoryArea.newArray(...)` exist.

Attention has to be paid when creating a `NoHeapRealtimeThread` object, as it is not possible to instantiate it on the Heap, and only within the context of a real-time thread.

This shall be clarified by the following example: In Figure 13, it is shown how to start a `NoHeapRealtimeThread`. At first a ,,help-thread“ `starter` is created. This object is allocated on the heap, but when it allocates objects on its own, they are placed in the scoped memory, represented by the object `scopedMemory`. This object `scopedMemory` *represents* a scoped memory area, but it itself is placed on the heap. The starter creates and starts a new `NoHeapRealtimeThread` (`nhrtt`). As the memory area of `starter` is `scopedMemory`, `nhrtt` is placed in `scopedMemory`. The RTSJ does not demand the `Thread.sleep(...)` instruction, but in the reference implementation[6] used, the virtual machine will hang up without this command. This is a bug, that probably is fixed in their new release.

[5] `java.lang.Thread` always allocates new instances in the heap memory

[6] We used the free available reference implementation from TimeSys (www.timesys.com)

```
public static void main(String[] args) {
  Starter starter = new Starter();
  ScopedMemory scopedMemory = new ...;
  starter.setMemoryArea(scopedMemory);
  starter.start();
}
public class Starter extends RealtimeThread {
  public void run() {
    NoHeapRealtimeThread nhrtt = new ...;
    nhrtt.start();
    try {
      Thread.sleep(3);
    } catch (InterruptedException e) {}
  }
}
```

Figure 13: Starting a `NoHeapRealtimeThread`

5.3 Monitors

Often concurrent running threads access the same memory. These *shared memory* areas have to be controlled by a so called *monitor*. Figure 14 gives an example for a monitor, implemented in Java. The key word `synchronized` avoids the parallel execution of the two methods.

```
public class Monitor {
  public synchronized int readDate()
  {
    ...
  }
  public synchronized writeData (int i)
  {
    ...
  }
}
```

Figure 14: Monitor

When a thread is inside a monitor (executes a `synchronized`-method of the object), it cannot be interrupted by another thread that wants to enter the monitor, too – even when this second thread has a higher priority. This leads to the well-known problem of priority inversion, described in [10, 3]. To deal with priority inversion, some useful protocolls have been developed. These protocols *do not* avoid the effect that a higher-priority thread is blocked by a lower-priority thread, but these protocols *minimize the blocking time*. Two well-known protocols are the *Priority Inheritance* and the *Priority Ceiling Protocols* [10, 3]. These protocols are represented in Real-time Java by two classes (see Figure 15). The priority inheritance protocol can be set when required.

The use of Priority Ceiling Emulation is also possible, but is only an optional element of any RTSJ implementation. It is important to mention that setting one of these protocols only has the expected effect if the underlying system provides the protocol.

6. CODE GENERATION

When generating source code from Real-Time Statecharts, at least one periodic thread –called *main thread*– is created. This main thread administrates the Statechart and has knowledge about the current state. In every period it checks periodically the outgoing transitions of the current state for being triggered and fires them if so.

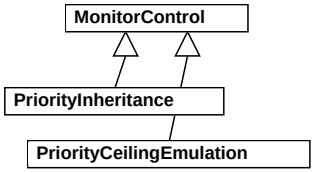


Figure 15: MonitorControl

There exist plenty of different patterns for implementing Statecharts. In order to satisfy real-time requirements, a variant without dynamic data structures has been choosen. In the target class, a constant is generated for every state (see Figure 16).

```
public static final int STATE_S1 = 0;
public static final int STATE_S2 = 1;
private int currentState = STATE_1;
public void handleTransitions() {
  switch (currentState) {
    case STATE_S1:
      if (/*check guard, timeguard and event*/) {
        action();
        t1 = 0;
        currentState = STATE_S2;
      }
      break;
    case STATE_S2:
    ...
  }
}
```

Figure 16: Implementation of a Real-Time Statechart

The attribute `currentState` indicates which state is valid so that the main thread just handles the (outgoing) transitions of the current state. As a Statechart can be in multiple states simultaneously because of hierarchy and parallelism, `currentState` is a more complex data structure than `int`, shown in Figure 16.

When executing the main thread, it runs on a real physical machine. Thus it cannot run infinitely fast, but with a minimal period. This results in the problem that a triggered transition is not recognized at the moment of activation, but with a delay, proportional –in the worst case– to the period. That's why the period should be as small as possible, but with respect to processor load and its own WCET, the period should be as long as possible. The deadline describes the point in time when a side-effect needs to be terminated, the WCET the time that is needed for execution and the period describes the delay between triggering and start of execution. Simplified, you can say: delay + WCET $\leq$ deadline must be satisfied. So the longest acceptable delay (and resulting, the period) can be determined, depending on the deadlines and WCETs. In general, shorter deadlines result in shorter periods.

Side-effects with a WCET greater than the period cannot be executed by the main thread. These actions are rolled out into aperiodic threads. The sequence diagram in Figure 17 depicts the application flow, when this is done. `FMainThread` is taken from the `sdm`-package, delivered with the Real-Time Statechart PlugIn. Its logic is the same for every Real-Time Statechart. The generated handler class implements the

interface `FRealtimStatechart` (defined in the `sdm`-package, too). It contains all the Real-Time Statechart specific information. First, the main thread calls `handleTransitions()` to determine the points of activation for all outgoing transitions of the current state. If the first activated transition has to be executed in an own aperiodic thread, the handler calls `executeAperiodicThread()` on the main thread to create and start a new aperiodic thread. The start is aperiodic, which results in concurrent running threads. As threads, that once terminated, cannot be reused in Java, the aperiodic thread is created in a scoped memory that frees the allocated memory after termination. After handling the transitions, the main thread waits for the next period.

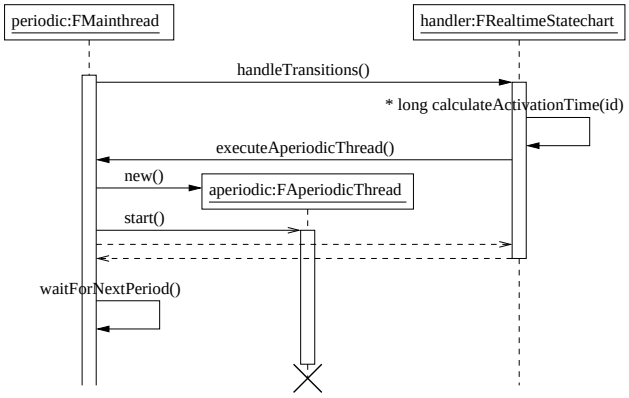


Figure 17: Logic of the main thread

The outlined scheme for the code generation permits to generate code for Real-time Java which ensures that the specified deadlines are always met when the WCETs are correct upper bounds.

7. CONCLUSION AND FUTURE WORK

Real-Time Statecharts provide a modeling technique, that allows to specify complex behavior on the one hand and real-time behavior on the other hand. Contrary to many other models, Real-Time Statecharts contain all information needed for code generation, which guarantees the specified timing characteristics [4]. The concepts for code generation are general, so that it is easy to adapt the generation algorithm to other target code. It is planned to provide a C++ generation in addition to the Real-time Java code generation soon.

Another capability of Real-Time Statecharts is the proof of temporal consistency with incomplete, but effective algorithms. In order to complete this verification, a model checker has been employed as reported in [9].

In addition, the visualization of recorded execution traces of Real-Time Statechart is planned [15].

Acknowledgements
We thank Florian Klein for his comments on earlier versions of the paper.

8. REFERENCES

[1] M. Awad, J. Kuusela, and J. Ziegler. *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion*. Prentice Hall, 1996.

[2] G. Bollella, B. Brosgol, S. Furr, S. Hardin, P. Dibble, J. Gosling, and M. Turnbull. *The Real-Time Specification for Java^{TM}*. Addison-Wesley, 2000.

[3] L. P. Briand and D. M. Roy. *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*. IEEE Press, 1999.

[4] S. Burmester. Generierung von Java Real-Time code für zeitbehaftete UML Modelle. Master's thesis, University of Paderborn, Software Engineering Group, 2002.

[5] B. P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. The Addison-Wesley Object Technology Series. Addison-Wesley, October 1999. Second Edition.

[6] H. Giese and S. Burmester. Real-Time Statechart Semantics. Technical Report tr-ri-03-239, Computer Science Department, University of Paderborn, June 2003.

[7] H. Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, January 2000.

[8] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *Proc. of IEEE Symposium on Logic in Computer Science*. IEEE Press, 1992.

[9] M. Hirsch and H. Giese. Towards the incremental model checking of complex real-time uml models. In *Proc. of Frist Fujaba Days*, Kassel, Germany, 2003.

[10] M. Joseph. *Real time systems : specification verification and analysis*. Prentice Hall international series in computer science. Prentice Hall, 1996.

[11] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Springer International Journal of Software Tools for Technology*, 1(1), 1997.

[12] OMG. UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02, September 2002.

[13] OMG. UML 2.0 Superstructure final adopted specification. Technical Report ptc/03-08-02, August 2003.

[14] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.

[15] M. Tichy and M. Kudak. Visualization of the execution of real-time statecharts. In *Proc. of Frist Fujaba Days*, Kassel, Germany, 2003.