

# WIP: Towards an UML 2.0 Conforming State Machine Implementation for the OROCOS Real-Time Framework

Markus Klotzbücher

March 13, 2009

This project investigates on how to integrate the concept of UML 2.0 finite state machines into the Open Robot Control Software (OROCOS) framework by attempting to answer the following questions: i) how can the semantics of UML 2.0 state machines be mapped to OROCOS ii) what changes would be required to OROCOS to support this and iii) how can this be implemented in an efficient and hard real-time compatible way. These efforts can also be understood as a case study for integrating a behavioral UML model in a real world software framework.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	UML 2.0 State Machines . . . . .	2
1.2	OROCOS . . . . .	2
<b>2</b>	<b>Problem Statement</b>	<b>3</b>
2.1	Non Standardized Behavior . . . . .	3
2.2	Scripting . . . . .	3
2.3	Meta State Machines . . . . .	4
<b>3</b>	<b>Proposed Solutions</b>	<b>5</b>
3.1	Unifying Events . . . . .	5
3.2	Dealing with Variation Points . . . . .	5
3.3	Scripting . . . . .	6
<b>4</b>	<b>Conclusion</b>	<b>7</b>

# 1 Introduction

## 1.1 UML 2.0 State Machines

The UML 2.0 standard largely adopts the semantics of Harel statecharts [2] which extend classical finite state machines (FSM) with concepts for modeling nested states (hierarchical composition) and concurrency (AND-states). In addition to these *behavioural finite state machines* UML 2.0 adds an specialized FSM variant called *protocol state machines*. Protocol state machines are always defined in the context of a classifier such as a class and allow to specify which methods are allowed to be called and which conditions must hold before and after the invocation.

The UML 2.0 standard *incompletely* defines the semantics of its entities. This is necessary for covering the wide range of systems which are to be modeled with UML. Such incomplete semantics are called variation points and any implementation will have to decide on how to fill these in. For system designers it is extremely important to know the precise semantics of variation points and to have the possibility to override them.

## 1.2 OROCOS

The Open Robot Control Software Project (OROCOS) aims at building a framework for construction advanced distributed machine control and robotics applications. It consists of the Kinematics and Dynamics (KDL) library, the Bayesian Filtering (BFL) library and the OROCOS Real Time Toolkit (RTT). The latter provides a component based framework for building distributed and hard real-time machine control and robotics applications.

Some important features of the RTT are support for mixing real-time and non real-time components, real-time state machines and scripting and advanced run-time deployment. The RTT state machine implementation and scripting are subject of this work-in-progress summary.

**Components** RTT applications are composed of individual components, which interact with each other by using different primitives. These are `RTT1::Ports` (data exchange), `RTT::Events` (asynchronous messages), `RTT::Methods` (synchronous object oriented style invocations) and `RTT::Commands` (asynchronous request of a service). The actual *computation* is performed in a components state machine, script of C/C++ hook. However this does not define *when* a computation takes place, which is specified by attaching an *Activity* to a component. Depending on the type of Activity chosen the computation will either be executed periodically (it gives up control and returns after a certain time) or permanently (it does not give up control).

---

<sup>1</sup>RTT is also the C++ namespace for the Real Time Toolkit libraries. The prefix is added here in order to avoid confusion with the often identical but semantically different UML terminology.

## 2 Problem Statement

The current RTT FSM implementation is showing some weaknesses which are summarized below.

### 2.1 Non Standardized Behavior

RTT finite state machines does not conform to any standard. This makes it more time costly to learn and complicates the integration with graphical modeling tools, which typically adhere to the UML 2.0 standard.

**Composite States** Composite states are used to model hierarchies of state machines and provide a powerful abstraction mechanism. The RTT provides a similar mechanism called submachine. Submachines are state machines that can be reused and contained within a parent state machine. However RTT submachines have different semantics in that a submachine is an independent state machine which is not part of the parent machine and executes asynchronously to it.

**UML::Event Types** *UML::Event* is an abstract meta class, which is sub-classed by a family of concrete event types such as *CallEvent* (an operation is being invoked), *TimeEvent* (a timer has expired), *SignalEvent* (an asynchronous signal was received) or *ChangeEvent* (a condition has become true) (see [1], pg. 440).

The RTT currently has no such unified event handling and supports only an equivalent to the UML::SignalEvent type in form of the RTT::Event.

### 2.2 Scripting

It is important to point out the close relation between state machines and scripts. By scripts is meant executable statements which are evaluated at run time. UML 2.0 state machines consist to a large part of such statements: entry, do and exit programs, guard conditions and transitions effects are essentially scripts which are tied together by some underlying execution logic and thereby form the overall behavior.

The current OROCOS scripting language has evolved from a simple language into a more general purpose language. Naturally some weak points have surfaced:

- non standardized syntax
- little debugging support
- no performance profiling features
- limited expressiveness (e.g. recursion is not allowed)

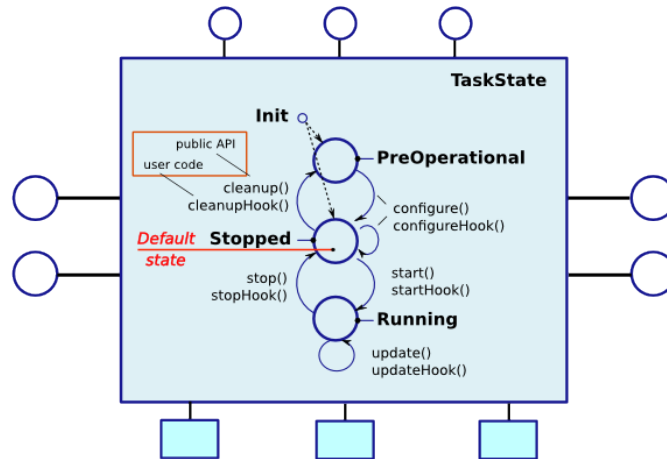


Figure 1: Component State Diagram (source: [5])

## 2.3 Meta State Machines

Besides the obvious OROCOS FSM as a mechanism for implementing intra-component behavior there are other more subtle and implicit state machines contained in RTT.

**Component States** An RTT component can be in different states as described by figure 2.3.

Simplified, a state machine starts its life in the *pre-operational* state. When the `configure` operation is invoked the component is configured and transitions to the *stopped* state. Calling `start` will cause a transition to the *running* state, in which the component actually does its work.

This component behavior is obviously defined by a state machine, although it is hard coded and therefore can not be extended easily by a user. However the possibility to override this default state machine would be a very useful feature.

**Component Execution** Figure 2.3 describes how the different parts of a component are executed sequentially by an attached `RTT::Activity`. First the `ProgramProcessor` executes RTT program scripts followed by the `StateMachineProcessor` which advances state machines. The `CommandProcessor` and `EventProcessor` process the `Command` and `Event` operations respectively. At last the user implemented C/C++ code in is executed.

Again this sequence of steps can very naturally be expressed with a state machine. Allowing the possibility to dynamically override this currently hard coded sequence with a state machine would allow a much higher level of flexibility for defining component behavior.

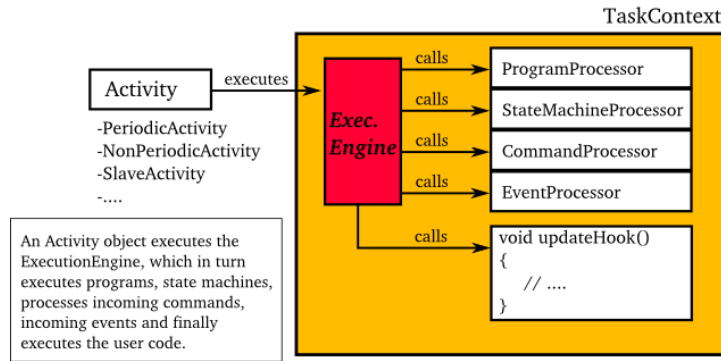


Figure 2: Component Execution (source: [5])

## 3 Proposed Solutions

### 3.1 Unifying Events

The first step for seamlessly supporting UML 2.0 state machines is to unify the notion of events as in UML (Event in the sense of an *occurrence of something*). In OROCOS this concept is currently scattered across different entities, e.g. UML::TimeEvents are realized by RTT::Activitiy, UML::CallEvents are implicitly realized by RTT:Methods etc. But ultimately these are only different ways of specifying which behavior is executed on reception of which event.

We propose *one event queue* per event type for each component. To these queues all UML::Events are delivered.

A (default) meta state machine which describes the component execution (see 2.3) is responsible for invoking subordinate computations such as UML::CallEvent processing, user supplied state machines or scripts and user C/C++ code which *consume* these events from the queues.

By moving towards the more general UML understanding of events component behavior can seamlessly be defined by an user overridable default meta FSM.

### 3.2 Dealing with Variation Points

There are several UML variation points which affect FSM, however for the purpose of brevity we only describe the *Time of ChangeEvent evaluation*, which in our opinion is the most difficult to implement.

An UML::ChangeEvent is described in the UML superstructure specification [1], pg. 435 as follows:

A change event occurs when a Boolean-valued expression becomes true.  
 [...] A change event is raised implicitly and is not the result of an explicit action.

The standard does not specify when the condition of a `ChangeEvent` is evaluated. Indeed this is a highly application specific property and choosing anything but a very conservative default could easily degrade overall performance.

Because of this the proposed default is to evaluate this condition only *before* the do behavior is executed. The rationale behind this is to allow the `ChangeEvent` to trigger a transition before the do behavior is executed (and thereby preventing its execution) while at the same time creating as little overhead as possible.

For conditions that require more frequent evaluation this event type must be realized in terms of an individual component to which an periodic activity with the desired frequency and priority can be attached.

### 3.3 Scripting

As explained in section 2.2 state machines and scripting are intimately related to each other. The quality of state machine implementation therefore depends very much on the quality of the scripting language used.

**Hard Real-Time Scripting** Given that some sort of scripting language will be required for interpreting expressions during execution, it is worth considering to implement the underlying state machine logic itself in the scripting language. The main advantage of this is a high level of flexibility for overriding and configuring semantic variation points. The main drawback is that such a light-weight, hard real-time scripting language expressive enough to make this feasible currently does not exist. The main reason for this is the inherent difficulty of hard real-time garbage collection. We intended to further explore this topic, as such a hard real-time domain specific language would be very valuable for prototyping hard real-time applications.

**Soft Real-Time Scripting with Lua** As hard real-time scripting will require long term investigation, a intermediate solution is required. One approach is the use of a soft real-time scripting language. Lua [3] is a mature and lightweight programming language especially targeted to be embedded. A key feature of Lua is the recently added soft-real time incremental garbage collector that allows fine grained control of the collection process. Tuning these parameters makes it possible to balance temporal behavior against overall performance.

Adopting Lua as the next generation scripting language for would have several benefits. First of all, it provides a non-intrusive migration path towards a more powerful extension language. Secondly state machines could be initially implemented in Lua, profiled<sup>2</sup> and then optimized by moving parts of the implementation to C/C++. The foreign function interface for this is very efficient and simple to use.

The downside of this approach would be a initial decrease in hard real time capabilities. However we believe this would be acceptable because

---

<sup>2</sup>Lua has support for profiling

1. an application that has critical real-time requirements will implement this particular part in C/C++ anyway.
2. the amount of such critical code is relatively small compared to the amount of code for which slightly softer real time constraints would suffice
3. there is more optimization possible for periodically executing scripts and state machines. Adaptive garbage collection algorithms [4] could be developed for learning to precisely schedule garbage collection within the idle period.

## 4 Conclusion

This analysis of the RTT finite state machine implementation has pointed out several shortcomings of the current implementation and described possible approaches for moving towards an UML 2.0 conforming implementation. It has become clear that this can only be done effectively by refactoring other core parts such as the component execution and the event handling. The most important insight so far is that all component behavior can effectively be expressed by one or more state machines. Secondly the need for a unifying events as in UML::Event has become apparent for allowing the meta level behavior of components to be expressed with state machines.

Of course as in any Open Source Project these issues need to be discussed with the community which has been done since the beginning. Further work will include the actual implementation, benchmarking and optimization of the real-time capabilities of this approach.

## References

- [1] Object Management Group. Uml 2.0 superstructure, ver. 2.1.2, formal/07-11-02, 2007.
- [2] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
- [3] Roberto Ierusalimschy, Luiz Henrique, Figueiredo Waldemar, and Celes Filho. Lua - an extensible extension language. *Software: Practice and Experience*, 26:635–652, 1996.
- [4] Sven Gestegard Robertz and Roger Henriksson. Time-triggered garbage collection: robust and adaptive real-time gc scheduling for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 93–102, New York, NY, USA, 2003. ACM.
- [5] Peter Soetens. *The Orocos Component Builder's Manual 1.6.1*. FMTC, 2009.