Journal of Software Engineering for Robotics 1(3), January 2012, 28-56
ISSN: 2035-3928

Coordinating Robotic Tasks and Systems with rFSM Statecharts

Markus Klotzbücher Herman Bruyninckx
Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium.

Abstract—Coordination is a system-level concern defining execution and interaction semantics of functional computations. Separating coordination from functional computations is a key principle for building complex, robust and reusable robotic systems. This work introduces a minimal variant of Harel statecharts called rFSM designed to model coordination of robotic tasks and systems with a minimal number of semantic primitives. Firstly, the semantics of the rFSM language are derived by analyzing state-of-the-art discrete event models and implementations and extracting a motivated and semantically well-defined subset that is considered best practice for the domain of robotic coordination. Secondly, a real-time capable reference implementation of rFSM is presented, which has been loosely integrated into the OROCOS/RTT framework. The application of rFSM is illustrated using a detailed description of a dual robot coordination problem. Lastly, several best practices and patterns are presented with the goal of i) supporting development of robust Coordination models, ii) illustrating how limitations of the statechart model can be overcome by extending the execution semantics, and iii) offering guidance in designing pure coordination components that optimize reusability.

Index Terms—Coordination, Statecharts, Domain Specific Language, Control, Component-based system design

1 Introduction

The design of todays complex robot systems is driven by multiple and often partically conflicting requirements. Apart from the primary functionality, these requirements commonly include the need to reuse existing parts, for the system to behave robust or safe in the presence of errors or to facilitate reconfiguration of the system due to changed requirements.

The extent to which these goals can be fulfilled is strongly influenced by the approach used to divide the system into parts. This work builds on the suggestion of Radestock and Eisenbach [1] to separate the systems according to the four concerns of Communication, Computation, Configuration and Coordination: Communication defines how entities communicate. Computation defines the functionality and hence what is communicated. Configuration defines how computations are configured and which computations communicate with each other. Lastly, Coordination is responsible for managing the individual entities such that the system as a whole behaves as specified.

Most of today's robotic software frameworks support the separation of Communication, Computation and Configuration. For example, the Orocos RTT framework [2] permits to separately specify communication connections between components and the respective parameters such as buffering policies. Component configurations can be defined separately and need only be applied to the respective component instances at runtime. Similarly, the ROS framework [3] permits separate storage of node parameters using the parameter service.

In contrast, the concern of Coordination is not yet recognized as a first-class design aspect in many of today's complex robotic systems. Instead, Coordination is often implicitly incorporated into Computation and Communication.

For example, in the Orocos RTT framework prior to version 2, coordination was often realized using the RTT::Event mechanism. Using this, a computational component declares the ability to raise an event represented as a parametrized function. Other computational components register handlers matching the event interface, which are then called to be notified of the event occurrence. As a result of this tight coupling, the event recipient is polluted with system-level coordination that

Regular paper – *Manuscript received July 17, 2012; revised November 15, 2012*

- *The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-231940-BRICS (*Best Practice in Robotics*) and no. FP7-ICT-230902-ROSETTA (*Robot control for skilled execution of tasks in natural interaction with humans; based on autonomy, cumulative knowledge and learning*), and from the KU Leuven Geconcerteerde Onderzoeks-Acties* Model based intelligent robot systems *and* Global real-time optimal control of autonomous robots and mechatronic systems.

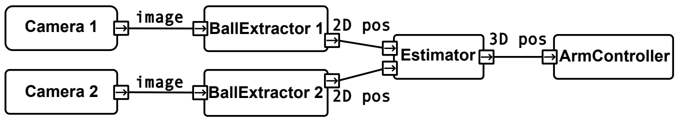

Fig. 1. Data-flow architecture of Ball tracking application.

limits its reusability. Moreoever, such implicit coordination can lead to reduced robustness by requiring to compromise between reusability and robustness. The following example further elaborates these issues.

Although the idea of explicit coordination is not limited to component based systems, we assume this context throughout the paper, since the majority of modern robotics software frameworks [2], [4], [5], [6] are component oriented. When using the term coordination or computation, we are referring to the concern, unless explicitely stated otherwise as in *coordination component*.

1.1 Motivating example

The following example introduces the the concern of coordination. A ball swinging on a string is observed using two cameras and shall be followed by a robot manipulator. The data-flow component diagram in Figure 1 shows the involved computational components and the communicated data. 2D ball positions are extracted from the camera images by BallExtractor components and passed to an estimation component. The estimated 3D position is then sent to the RobotController actuating the robot arm. To avoid confusion with state machine diagrams, we utilize the SysML flowport notation [7], a small box with an arrow pointing in or out to describe component input or output data-flow ports respectively.

This system behaves as intended for the nominal case of the ball being visible to the cameras. However, if the ball swings out of the observed camera range the behavior is not well-defined, since different estimators will produce different results; for instance an estimator based on a constant velocity model will predict the ball motion to continue with the last estimated velocity, while one based on a constant position model will continuously predict the last estimated position.

This example constitutes a typical coordination problem, characterized by undesirable behavior at the system level even though the individual parts are functioning correctly.

Assume the requirement that the robot arm shall stop close to the last observed ball position. A naive solution to this problem would be to extend the estimator to stop the robot controller once the ball left the field of view of the camera. This solution is suboptimal for several reasons: firstly, the reusability of the estimator is severely reduced by adding this application-specific feature. Secondly, the exchange of this component with one not making this application-dependent assumption is prevented. Lastly, the solution provides limited robustness in case of communication failures between BallExtractor and Estimators, even though short interruptions may well be dealt with gracefully.

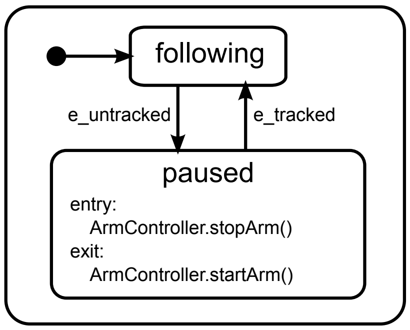

Fig. 2. Ball-tracking coordination state machine.

The latter shortcoming illustrates the previously mentioned compromise between robustness and reusability: the developer is forced to choose between either reduced reusability or to accept undefined behavior in corner cases or in the event of errors. The proposed solution to avoid these shortcomings is described by the following steps:

1) **Making relevant internal state visible**: the estimation component is extended to raise two events: `e_untracked` and `e_tracked`[1] when the ball becomes invisible and visible respectively (i.e. when the confidence in the ball position drops beneath/rises above a configurable threshold). This extension is both generic and non-intrusive to the estimator, as it only makes its internal state explicit but does not alter its behaviour.
2) **Introducing explicit coordination**: The dependency between Estimator and ArmController is encapsulated by introducing a separate Coordinator entity (typically, but not necessarily, deployed in a component itself) that reacts to the estimator's events. The state machine in Figure 2 models the desired behavior: the events `e_untracked` and `e_tracked` trigger the transitions from the nominal `following` state to `paused` and back respectively. The robot arm is stopped when entering and restarted when exiting the `paused` state.

This way, the reusability of the estimator is preserved while the desired behavior is specified in an explicit manner. Moreover, the proposed Coordination will deal gracefully with communciation failures between BallExtractor and Estimator; a sufficiently long interruption will result in the raising of the `e_untracked` event that stops the robot, while a short interruption will be deal gracefully. If a distinct reaction to this condition is required, it can be easily specified by extending the Coordination FSM to react to an event that represents the loss of communication.

1. To improve readability events are prefixed with e_ throughout this paper.

1.2 Contributions

This paper advocates the introduction of explicit Coordination components into every robotics software system and offers guidelines to do so. Its contributions can be divided into four parts. Firstly, it presents a study of task and system coordination mechanisms with respect to their suitability for the robotics domain. Secondly, based on the identified shortcomings of existing models a minimal variant of Harel statecharts addressing these issues is proposed, in form of the restricted Finite State Machine (rFSM) model. The term minimal is to be understood in terms of *the smallest number of primitives necessary for humans to construct practical coordination statecharts*, and not as minimal, yet with equal computational expressiveness. Thirdly, a real-time safe reference implementation of rFSM as an executable Domain Specific Language (DSL) is presented and the integration into robotic frameworks described. Lastly, to facilitate adoption of the described approach, a set of best practice patterns and guidelines are provided, to help system and component developers with the design of efficient, robust and reusable coordination.

1.3 Related Work

This Section summarizes previous work on analysis, formalization and classification of finite state machine semantics. A detailed review of state-of-the art coordination models themselves is presented in Section 2.

Von der Beeck [8] compares twenty different statechart variants according to a set of distinctive features. Some suggestions are given about which features should be included in a statechart formalism, however it is not clear what the motivation for inclusion or not is.

Eshuis [9] identifies a set of constraints for which subsets of the three common statechart semantics—STATEMATE, Fixpoint and UML—behave identically. Besides suggesting improvements to statechart semantics, this work differs significantly from ours that is focussed on using statecharts for the particular purpose of coordination.

Breen [10] identifies several shortcomings of Harel statecharts for the purpose of system specification. Most issues are related to parallel states (modeling concurrency), and hence do not affect our rFSM model that excludes this model element. Breen observes that the difficulty of understanding the complex relationship between parallel states (which is a consequence of broadcast event communication and the inevitable non-determinism of communication delays) can be avoided by simply using separate statecharts instead of parallel states. This observation, albeit for different reasons discussed in Section 7, confirms our approach to exclude the parallel state element. Furthermore, Breen describes the following shortcoming of the statechart model that is related to hierarchy and also affects the rFSM model: by observing just the graphical model of a state machine, the designers' motivation for using hierarchy is not always obvious. This is due to multiple use-cases for using hierarchical states. To this end, the following use-cases are identified: clustering, abstraction, transition dependence, state variable instantiation and expression of constraints. Breen concludes by pointing out that the described problems will not be significant in relatively simple models.

Simons [11] examines the semantics of UML State machines and suggests a revised interpretation. Extensions are identified that can be considered either redundant or harmful to the compositional properties of the model. The latter class include the inversion of conflict resolution priorities in UML with respect to Harel statecharts and the special semantics of completion transitions. Moreover, the revised semantics attempt to improve the differentiation between statecharts and flowcharts. The rFSM semantics described later follow several, but not all, of these suggestions.

1.4 Outline

The rest of this work is structured as follows. Section 2 reviews existing coordination models and implementations. After briefly introducing the rFSM model in Section 3, the insights gained from these reviews are used in the discussion of structural, execution and event semantics in Sections 4–6, that ultimately lead to the derivation of rFSM. Section 7 is dedicated to motivate the exclusion of the widely used parallel state element from the rFSM model. Section 8 describes the reference implementation that was developed. Best practices and patterns are described in Section 9. Section 10 presents a detailed example that illustrates step by step the process of developing the coordination for a dual robot haptic coupling system. We provide a discussion of our contributions in Section 11, and conclude in Section 12.

2 Review of Coordination Models

2.1 Classical Finite State Automatons

Finite state machines (FSM), sometimes called finite state automata, are mathematical models of behaviour. FSM are widely used, ranging from application in computational linguistics to language parsing and artificial intelligence. A FSM consists of a finite set of states connected by transitions.

Commonly, two types of FSM are distinguished: *acceptors* and *transducers*. Acceptors, which are also known as recognisers, return a binary result of whether or not a certain input was recognized, and are mostly used in language recognition. Transducers can generate output with the help of *actions*, and can be divided into two classes: *Moore* and *Mealy* state machines. In the Moore model, the output is a function only of the current state, while in the Mealy model the output is a function of the input and the current state. While both models are equally expressive, in practice often formalism as UML 2 state machines are used, where the output can be a function only of state or of state and input.

2.2 Harel Statecharts

The *Harel statechart* [12], [13] is the first state machine based formalism to extend classical flat automatons with hierarchy, parallelism and broadcast events. Hierarchical states may contain states themselves, thereby facilitating abstraction and modularization of larger models. Additionally, hierarchy enables transition prioritisation using a strategy named structural priority: in case of two or more conflicting transitions the one with the least nested source state (within the global hierarchy) takes priority. Parallelism permits substates contained in a so called AND state to be simultaneously active. Broadcast events are a more controversial feature [10] causing that events raised in one sub-statechart (such as a parallel substate) are observed and hence can trigger transitions throughout the entire statechart, thus violating the scope introduced by each level of hierarchy. The aforementioned controversy arises from the fact that the behaviour of a large statechart can become considerably intricate, especially in combination with statechart parallelism.

The quasi-standard statechart semantics are documented and implemented in the STATEMATE model and tool [13], [14]. Apart from STATEMATE, numerous other variations exist [8], which is to some extent a consequence of the fact that a detailed description of statechart semantics was only provided years after the initial proposal. Today, the most widely used variants are *UML 2* [15] state machines, the W3C *State Chart XML* (SCXML) [16] model and the commercial *Simulink Stateflow* tool of the MathWorks [17].

2.3 OMG UML State Machines

UML state machines [18] are derived from Harel statecharts and introduce various syntactic and semantic extensions with the goal to enable application in an object oriented context. One limitation of UML in general are the so-called *semantic variation points* that intentionally leave the precise semantics of model entities open; for example, the order in which events are dequeued from the internal event queue is left undefined. This is to give tool-implementers of the standard more liberty to define these semantics themselves. As a consequence, practically all implementations of UML state machines yield incompatible execution behaviour, *and* the precise semantics often remain hidden in the *implementation* of the tool.

A second issue with UML statecharts arises from semantic deviations made from Harel statecharts with the goal to make UML statecharts behave more like objects as in the object-oriented programming paradigm; the consequences are discussed in Section 5.

In spite of these limitations, UML state machines are widely used and supported by many modeling tools. Hence, the statecharts proposed in this work follow these semantics wherever appropriate, in order to facilitate automatic transformation of UML state machines to rFSM models.

2.4 Simulink Stateflow

Stateflow [17] is a statechart extension to the Matlab/Simulink environment into which it is tightly integrated. Stateflow offers an even larger set of primitives than statecharts: *Condition actions* are executed if a guard is true, even if the transition itself is not taken, *inner transitions* emanate from the inner boundary of a composite state and are similar to self-transitions but do not result in exiting and entering the source state. Apart from these convenience features, major differences to Harel statecharts are that Stateflow only supports processing of one event at a time and does not transition based on more than one event. Stateflow models can be configured in great detail, for instance to define transition priorities on the chart or state level or to define if supersteps may occur or not. This permits fine grained configuration and optimisation, though at the price of compromising compatibility with other implementations, and reuse in other application contexts.

2.5 Statecharts in Robotics

Statecharts have been previously used in robotic systems. Merz et. al. [19] use an augmented flavor of Statecharts called Extended State Machines (ESM) to specify control and data flow in a robotic control framework. In contrast to our approach that advocates the separation of the concerns of computation and coordination, ESM states tightly couple control and data-flow by permitting these to also output data via data ports. While this combination might be convenient, reusability is reduced by tightly coupling application coordination with functional computations. Billington et. al. [20] propose an approach to requirement engineering using UML Statemachines that are extended with non-monotonic logic for describing domain knowledge. This logic is used to describe the behavior of single FSM and the corresponding predicates to be used as guard conditions. This formulation then allows to validate certain properties such as the exclusivity of guards. Similarly to our approach, the behavior of the system is defined by multiple interacting state machines, though the focus is on modeling complex high-level behavioral protocols. In contrast, our work is concerned with preserving reusability of computational components.

2.6 IEC 61131-3 Sequential Function Charts

Sequential function charts (SFC) are a PLC programming language specified in the IEC 61131-3 [21] standard. The SFC language is based on the Grafcet language [22], which in turn is derived from Petri nets. This heritage reflects in the SFC semantics that generally impose less constraints with respect to hierarchies and permitted transitions.

SFCs are constructed by linking *steps* with *transitions*. Each transition has a *condition* that defines when control passes to the next step. *Simultaneous divergence* permits to activate two successor states simultaneously while *simultaneous convergence* permits to synchronize the parallel execution again. In order to specify the in-step behaviour, steps can be associated with *Action blocks*. *Action qualifiers* define the exact manner in which the action is executed. Example qualifiers are `N` (non-stored) for executing an action as long as the state is active, `S` (stored) for executing an action permanently until disabled by the `R` (reset) qualifier, `P` (pulse) executing an action only when entering and/or leaving a state, or `L` and `D` for limiting or delaying the execution of the action to/for a certain time period. SFCs support hierarchical decomposition by recursive specification of an action with a SFC (or any other IEC diagram).

Bauer and Engell [23] compare SFC to statechart semantics. Summarised, the major differences are:

- Although SFC support hierarchical steps by recursively specifying an action with a SFC, none of the constraints of statecharts are enforced. This is because there is no transitively upwards closed active state list. In contrast, if a hierarchically nested state is active in a statechart, then so is the parent state. For SFC it is possible that a parent step is deactivated but the child step remains active.
- In contrast to statecharts, SFC do not permit to specify inter-level transitions.
- A statechart will behave non-deterministically if *conflicting transitions* are unable to be resolved by means of the conflict resolution mechanisms. If conflicting transitions are found in a SFC then *all* transitions in question will be enabled.[2]
- IEC 61131-3 does not define the order of action execution and transition evaluation for SFCs, however this ordering may influence the overall behaviour.
- In statecharts, higher priority transitions can prevent lower-level transitions from triggering. Whether this is possible in SFC depends on the respective implementation of hierarchy.

Bauer and Engell conclude by suggesting that a combination of statecharts (for specifying the high level operational modes and safety aspects) and Sequential Function Charts (for defining the sequences of lower-level computations), would yield the benefits of both models. A possible execution semantics for the combined model is proposed.

2.7 Behavior Trees

Behaviour trees (BT) [24] are a graphical language to support the process of behavior engineering. The BT language consists of different types of states that are organized in a tree form to express behavior. For example, states can represent conditions (`IF`, `WHEN`, dataflow `Data-out` or state (`System`, `Internal`. The behavior engineering process consists of several steps, each producing a new behavioral tree. Firstly, individual functional requirements are modelled using requirements behavioral trees (RBT). Next, these requirements are integrated into the design behavioral tree (DBT) that composes all requirements. Using the DBT, the component interaction network (CIN) and the component's behavior tree (CBT) are derived. The first describes which components interact and the seconds models each components behavior.

2. According to IEC61131-3, pg. 100, this is an error.

2.8 The Task Description Language

The Task Description Language (TDL), introduced by [25], is a language to describe Robot Tasks. TDL is based on the task tree datastructure, whos nodes can contain commands, goals, monitors or exceptions. Goals describe higher level tasks whos children may be goals or commands. Monitors are invoked repeatedly to validate certain conditions; exceptions can be used to signal and handle erronous conditions. When a goal node is expanded, new children (goals or commands) are added to the goal itself, thus defining what needs to be handled before the goal completes. This way, the tree datastructure implies an ordering between execution of parent and child nodes. In contrast, sibling nodes are executed concurrently unless ordering constraints are imposed using one of the synchronization mechanisms. A TDL program is a C++ program with additional syntax that is transformed to pure C++ by a dedicated compiler.

2.9 The Urbiscript Language

The *urbiscript* language [26] by GOSTAI is a multi-paradigm scripting language targeted towards the domain of robotics. To that end, it provides syntax to specify concurrent execution of statements and primitives to supporting event driven programming. The GOSTAI Studio IDE supports development of hierarchical state machines that can be transformed to urbiscript. However, as of today no information about the semantics of this state machine formalism is publicly available; with the exception that, according to GOSTAI, the semantics are close to a subset of UML 2.

2.10 ROS SMACH

ROS SMACH [27], [3] is a Python library for specifying models of robotic behaviour. States encapsulate computations that are executed while a state is active. States are exited via a so called *state outcomes*, that essentially represents different accept conditions. Additionally, states may specify *userdata* that is either required by a state to do processing or provided as an output of the computation of the state. When specifying a SMACH state machine the connections between input and output userdata can be defined. SMACH supports parallel execution of state behaviours as well as controlled preemption.

Nevertheless, in contrast to its name the SMACH library has actually more in common with a flowchart than with a classical state machine: (i) states do not represent conditions of the system but rather processing steps, and (ii) SMACH does not foresee a mechanism to *react* to external events.

2.11 Conclusion of Literature Review

Based on the previous analysis, we have selected Statecharts as a starting point for deriving a minimal coordination mechanism suitable for the robotics domain. This choice was based on the following requirements. To support reuse of coordination, the model must be *composable* and permit recombination of existing models within each other. Since the Statechart model allows states to contain states this is easily possible. It is worth noting that the reuse potential of statecharts has also been recognized by other domains such as Game AI development [28] or for building reusable webservices [29].

Moreover, compositionality is of primary concern, since the goal is to support construction of complex models by combining simple ones while maintaining predictability of the global system behaviour. *Compositional robustness* is closely related to compositionality; we define this property as follows: a system which is constructed from elementary parts is compositionally robust if it behaves robustly as a whole under changes or errors in individual parts. This characteristic is given particular attention in the following, because robotic coordination can often be conveniently modelled as a composition of multiple, heterogeneous layers of abstraction.

The ability to satisfy real-time constraints is a fundamental requirement for robot coordination related to aspects such as motion control or safety mechanisms. This does not only involve guaranteeing deterministic timing behaviour but also the ability to provide introspective information about the temporal behaviour. On the other hand, building complex, multi-robot systems requires distribution of such local real-time safe coordination over unreliable networks. As a consequence, coordination must be robust under (event) communication failures or varying latency. For example, reordering or loss of events may never lead to a coordination dead- or live-lock. Structural priorities and time-events facilitate this.

Lastly, statecharts and its variants are widely known and many developers are familiar with these. As this paper aims at providing a minimal subset of the statechart model, the reuse of existing tools becomes possible.

For our purposes, the main alternatives to statecharts are behavioral trees or hierarchical petri-nets. Behavior trees can be composed in a similar way as statecharts, but offer a richer model and require more rigorous commitment to the approach, compared to this work's approach of superposing a statecharts to coordinate computational components. Moreoever, statecharts are better suited for modeling reactive and preemptable behavior by using non-local state transitions (i.e. a top level transition causing the deactivation of an entire sub-branch) taking into account the proper invocation of associated exit and entry functions. Similar limitations apply to hierarchical petri-nets (HPN) or variations of these as SFCs. Moreoever, the implementation of concurrency in Petri-nets requires making similar assumptions as those which we choose to avoid by excluding parallel states in rFSM. Furthermore, petri-nets abstract away communication (e.g. for synchronization) between concurrent threads of execution. Thus, especially distributed Petri-Nets would require making additional assumptions on communication properties, which we can avoid for rFSM. Lastly, statecharts are, to some extent, also applicable to modeling of sequences (such as assembly tasks) that are traditionally the domain of petri-nets.

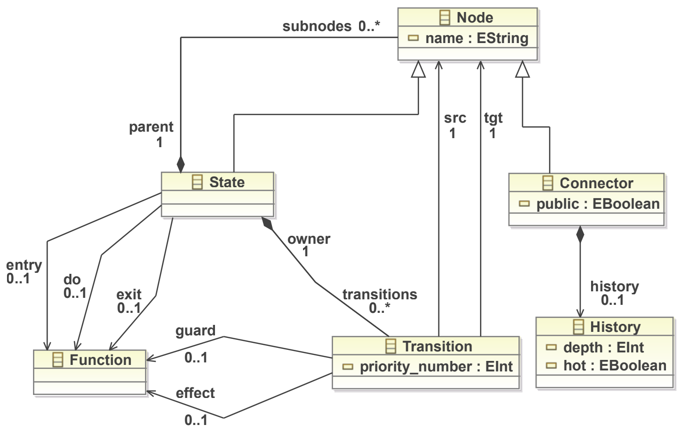

Fig. 3. Complete rFSM Ecore model.

3 The rFSM Model

The rFSM model is a minimal subset of UML 2 and Harel statecharts consisting of only three model elements: **states**, **transitions** and **connectors**. Additionally, a virtual model element named *node* is introduced to simplify explanations concerning both states and connectors. Figure 3 shows the structural model as an Ecore diagram.

States can be composite or leaf states, depending on whether they contain child nodes or not. For example in Figure 4, the outer rounded rectangle (labelled `root`) is a composite state; the rounded rectangles `opening`, `grasping`, and `closing` are leaf states of the tree formed by the hierarchical composition of states; the arrows are transitions; and the filled black circle is an initial connector. Distinguishing between leaf and composite states is important for modelling additional constraints that must be satisfied by valid rFSM models. For example, an important constraint that simplifies validation and execution semantics is, that, for an active, well-formed rFSM statechart, exactly one leaf state must be active (unless during transitioning, where it may be zero). Moreover, this constraint permits representing the entire active state of a rFSM statechart by means of a single leaf state. This is owned to the constraint of the statechart semantics, requiring that for any active state all parent states are active too.

At the top-level, a rFSM model is always contained within a state; that way any state machine model is inherently composable within other state machines. This composability is essential to enable the reuse of Coordination functionality represented by rFSM models.

Transitions connect nodes in a directed way, and may define a side-effect free, Boolean-valued *guard* function (whose result determines whether the transition is enabled or not), as well as an *effect* function (invoked when transitioning). Note that defining a specific *trigger* language in the core model is avoided by deferring the responsibility of determining the triggering of a transition to the guard function. The latter can then be extended (in an automated way, e.g., by a plugin) to validate any event specification specified on a transition against the current set of events.[3]

Transitions are owned by composite states and *not*, as often assumed, by the states from which they originate (more explanation is given in Section 4.2.1). As this constraint is not expressed by the Ecore model, the OCL constraint shown in Listing 1 is added. The formulation assumes a function `LCA` to compute the least common ancestor and a predicate `ancestor` to check whether the first argument is an ancestor of the second.

```
context: Transition inv:
   let lca : State = LCA(self.src, self.tgt) in
      self.owner=lca or ancestor(self.owner, lca)
```

Listing 1. OCL constraint on transition ownership.

Following UML, rFSM permits states to be associated with behaviours that are executed at different points in time. These behaviours are the `entry` and `exit` actions that are executed upon entering and leaving a state respectively, and the in-state `do` activity. While actions are generally short, activities are composed of atomic actions themselves and can run over longer periods, during which they are interruptible at the granularity of actions. Hence, to simplify execution semantics, the `do` activity is restricted to leaf states only. Moreover, rFSM states may define *internal transitions*, that permit reacting to events by executing an effect, though in contrast to regular transitions *without* leaving the state. In the rFSM model, all behavior is modelled as opaque functions.

Connectors can be used for constructing composite transitions by interconnecting two or more elementary ones. When such a transition is taken, the scope of connectors is honoured, permitting to define exactly which states are exited and entered throughout the composite transition. Connectors have multiple uses: for example they can be used to define the interface of a composite state by providing different entry or exit points. The initial connector has special semantics: when a transition ending on the boundary of a composite state is executed, the execution will continue with the transition emanating from the initial connector. To avoid stuck transitions, a constraint is introduced to enforce that each composite state that is target of a transition also defines an initial connector. This is expressed by the first OCL constraint in Listing 2; the second enforces that a composite state may define at most one initial connector.

```
context:
   State inv: self.subnodes.size>0 and
      Transition.allInstances()->select(t | t.tgt=self)
           implies
         (self.subnodes()->select(c | c.isTypeOf(Connector)
              and c.name="initial"))->size() = 1

   State inv: self.subnodes.size>0 implies
      (self.subnodes()->select(c | c.isTypeOf(Connector)
           and c.name="initial"))->size() <= 1
```

Listing 2. OCL constraint on existence of initial states.

As with states, multiple transitions may emanate from a connector. This permits to implement *dispatching* transitions that are triggered by two or more events and that dispatch to different states (see Section 4.3 for an example). The rFSM connector model element unifies the four very similar UML model elements junction, initial, entry- and exit pseudostates.

A state-machine is entered for the first time by transitioning via the transition emanating from the initial connector of the root state, resulting in the target state of this transition to be entered.

The elementary way to advance a rFSM state machine is to call the *step* function on it. This function retrieves *all* events that accumulated since the last step and attempts to find an enabled transition. This process starts top-down, from the root composite state down to the active leaf state. The rFSM semantics require that, as soon as an enabled transition is found, the searching terminates and the transition is executed.

This approach of identifying the next transition has the advantage that it assigns explicit priorities to transitions, so-called *structural priorities* [13]. The higher the source state of a transition is located in the state machine's tree, the higher the priority of the transition. The priority is visible in the state graph: given a set of events, the current active states, and the value of the guard predicates, it is immediately visible which transition will be selected. This follows the approach chosen for the STATEMATE semantics. Furthermore, through structural priority conflicts between transitions are largely avoided, leaving only the possibility of local conflicts among transitions exiting the same state. These conflicts can be eliminated either by additional guard conditions, or by a mechanism such as priority numbers

The minimum requirement for events is to carry identity and hence to be comparable between (possibly distributed) connected state machines. The simplest approach that remains comprehensible to humans and that is real-time safe[4] is to use string events.

The rFSM model does *not* include a parallel state element. The reason for this is that this element requires making a large number of fundamental, platform-specific assumptions. These are, for instance, the *order* in which parallel states are entered and exited, which underlying *concurrency mechanisms* such as threads are being used to execute the state machine instances

3. Throughout the paper we assume a basic trigger language that permits specifying a list of events of which each triggers the respective transition.

4. Obviously assuming that memory is appropriately pre-allocated for the longest possible event.

or what the *priorities* of different parallel regions are. Instead of parallelism, a loosely coupled approach of distributed state machines is used, that permits multiple rFSM instances to interact by means of the available communication middleware (Sec. 7).

Moreover, rFSM does not adopt the STATEMATE *execution time* requirement which enforces that changes occurring in step n can only be sensed in step $n + 1$, because of the complexity involved to manage such delayed processing of effected changes. But also, more importantly, because of the impossibility of delaying changes in open environments, as it is inherently the case in robotics. rFSM also differs with respect to the *greediness property* of transition selection, by choosing a simpler *take first* approach to computing the set of transitions to execute. In contrast to STATEMATE, external and internal events are not distinguished.

The ability to simulate a statechart before execution is a useful feature than can facilitate early detection of errors. However, often subtle corner cases remain in which simulation and real system behaviour differ, as is the case for STATEMATE. The rFSM reference implementation avoids this problem by unifying simulation and real system in form of an *executable model*. Obviously, this is possible only because the rFSM model targets the domain of coordination (in software) and need not be synthesised to targets such as VHDL (Very High Speed Integrated Circuit Hardware Description Language).

For convenience, we use the graphical notation of UML to visualise rFSM statecharts throughout the paper. The only deviation is necessary to distinguish between initial and non-initial connectors: the first are (as in UML) shown as filled circles while the latter are depicted as empty circles. Public connectors must be drawn on the border of composite states.

4 Structural Semantics

The following sections discuss in detail the implications of different semantics of finite state machine model elements. Since the rFSM model is derived from Harel statecharts and UML state machines, the discussion is based upon these standards. Other models are included in the discussion where appropriate.

4.1 Fundamental State Machine Elements

The fundamental elements of UML state machines are introduced using an example of a gripper coordination FSM shown in Figure 4. The required behavior is the following. Initially the gripper shall be opened. After receiving a request `e_close` the gripper shall start to close. The closing will be aborted and the gripper reopened if the request `e_open` is received. If a successfull grasp is detected, the grasp control loop shall be activated and remain active until either the object is dropped (represented by `e_tactile_lost`) or releasing the object is requested (`e_release`). If the grasp fails, the gripper shall be reopened in preparation of a new attempt.

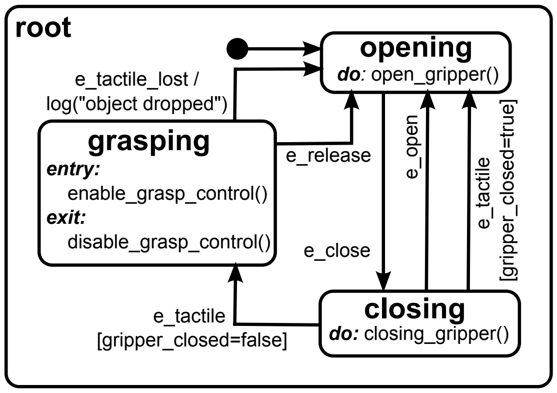

Fig. 4. Coordinating a gripper.

The required behavior is realized as follows. Three states `opening`, `closing` and `grasping` are introduced, each representing a distinguishable and exclusive state of the gripper. The filled circle is an UML the *initial pseudo-state* and together with the transition to `opening` ensures that the latter will be the first state entered. Transitions connect states in a directed manner and carry a label of the following form:

```
trigger [guard condition] / transition effect
```

A trigger (typically one or more events) enables a transition between a source and a target state if and only if the optional *guard condition* evaluates to true.

An example of a guard condition can be found in Figure 4 on the transition from state `closing` to `grasping` labelled with `e_tactile [gripper_closed = false]`. This expression defines a successful grasp as an event from the tactile sensor (`e_tactile)` *and* the condition that the gripper is not fully closed. Likewise, a failed grasp is detected by a tactile event and a fully closed gripper; this triggers the transition back to `opening`. Guard conditions constrain transitions and often permit reducing the amount of states necessary. On the other hand, guards also hide state (hidden by not being represented by a state model element) and can thus lead to conflicting transitions if two guards are simultaneously true. Therefore, it is considered a best practice to define these such that the exclusive disjunction of all guard conditions is true at any time.

A *transition effect* is an action that is executed during a transition between states. In Figure 4 the transition triggered by event `e_tactile_lost` from state `grasping` to `opening` uses the effect for logging a message that the grasped object has been dropped. Effects are necessary to define actions that are specific to transitions and not to source or target states. For the grasping example, the log message is only required when the transition to `opening` takes place as a result of unintentionally dropping the grasped object (triggered by `e_tactile_lost`), and not when the object is deliberately released (`e_release`).

The Matlab Stateflow model [17] includes a *condition action*, that is already executed when the guard condition evaluates to true, but before the transition as a whole is enabled. That way, condition actions behave similar to transitions connected by UML Choice pseudo-states (describes in Section 4.3.2), and are not included in the rFSM model for the same reason.

Actions can be associated with states too. The *entry* and *exit* actions are executed when a state is entered or exited respectively. For instance, when the state `grasping` is entered, force control is enabled and disabled when the state is left. Like transition effects, entry and exit actions are always executed atomically as part of a transition and are never interrupted. In contrast, a state's *do* activity is executed as long as the state is active. In contrast to entry and exit actions, UML defines that (but not how!) the `do` activity can be interrupted if an event occurs that triggers an outgoing transition. For the example given in Figure 4, in state `closing` the gripper motors are continuously stepped by the `closing_gripper()` activity until the gripper is fully closed. Executing this operation in the do-activity permits it to be interrupted by a request to re-open the gripper or an event from the tactile sensors.

It is important to note that for robotic coordination this interruptibility can not be achieved by forced preemption, as for instance used to preempt operating system threads. This is because in systems interacting with real hardware some code paths must be treated as atomic in order to avoid non-deterministic and potentially dangerous behaviour.

For achieving safe preemption of the `do` activity, rFSM adopts the idea of GenoM *codels* [30]. A codel is the smallest non-interruptible unit of execution. Codels can be composed into larger computations that are interruptible at the granularity of the individual codel. By defining the `do` activity in terms of codels, its execution can be safely interrupted. The worst case latency for exiting the `do` program is defined by the execution time of the longest codel. The rFSM reference implementation realises this behaviour by implementing `do` using coroutines [31].

The last fundamental primitive is the *internal transition*. The behaviour of this transition type is best explained by contrasting it to a self-transition, which is a regular transition starting and ending on the same state. When the latter transitions, the state is exited and entered, including invocation of the respective `exit` and `entry` actions. In contrast, when an internal transition is taken, the defining state is *not* exited. Consequently, the only action invoked is the internal transitions effect. Syntactically, internal transitions are drawn within a state and otherwise have the same from
`trigger [guard] / effect` as regular ones.

Interestingly, this seemingly simple model element already illustrates the issue of implicit semantic variation points in UML. For instance, it is undefined whether, in case of conflicts, internal or external transitions take priority. Furthermore, it is unclear if and how an internal transition affects a running `do` activity. Does it cause the `do` activity to restart or does the internal transition effect run interleaved?

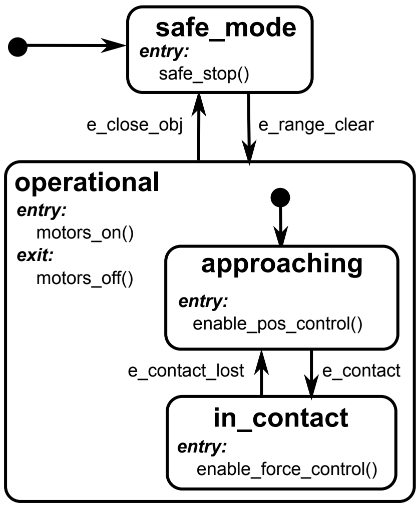

Fig. 5. Hierarchical state machine.

According to Simons [11], internal transitions are redundant since all behaviour associated with a state can be modelled using a substate. This is however, only true for an internal transition defined within a leaf-state. Internal transitions defined within non-leaf composite states can not be converted to self-transitions, since the execution of the effect action implies exiting and entering of all child states. In practice, a typical use-case for internal transitions is to model *secondary mode switches*. For the grasping FSM of Figure 4, this could be to permit runtime switching between different controller configurations, but without otherwise interfering with the nominal grasping task. Such mode switches can be conveniently modelled using internal transitions defined at the root level.

With respect to variation points, the rFSM model assumes the following: regarding priorities, internal transitions are given equal priority to external ones unless made explicit by priority numbers. As to interference with the `do` activity, it is assumed that the internal transition execution takes place interleaved, though in a safe way honouring the atomicity of codels.

In summary, the rFSM model adopts the following fundamental state machine elements from the UML 2 standard: state, transition, internal transitions, event, entry, exit and transition effect actions, do activity and guard condition.

4.2 Hierarchical State Machines

Hierarchical state machines were first introduced by Harel [12]. The key idea is to permit nesting of states within states. This is illustrated by Figure 5 showing a model of a robot performing a force-controlled operation on a work piece. The actual operation is modelled using a nested state machine within the `operational` state. Such a state machine containing other states is also called a *composite state*, since it can be understood as the composition of multiple state machines. When the `operational` state is entered via the transition triggered by `e_range_clear`, it immediately continues entering the `approaching` state, since this is connected from the initial connector. As a consequence the work piece is approached until the event `e_contact` is received, that triggers the transition to the `in_contact` state, in which force control is enabled.

At the top level, the two states `safe_mode` and `operational` describe the basic behaviour of the system. The `safe_mode` state stops the robot immediately and is entered when a sensor such as a laser scanner reports an object within the safety range. Note that the execution of this transition will result in exiting the `operational` state, no matter which substate is active. The event `e_range_clear` signals that the safety range is clear again and triggers the transition back to the `operational` state.

A fundamental property of hierarchical state machines is that multiple states can be active at the same time. For example, if the state `approaching` is active, then so is the parent state `operational`. In statecharts, the set of active states, also called the *active state configuration*, is always transitively closed with respect to the parent relation.

It is often helpful to visualise hierarchical state machines as a rooted tree. The root is formed by the top level composite state with nested states as children. This way, the active state configuration of any active hierarchical state machine can be represented by a single state (assuming the transitivity relationship from above).

Moreover, the tree structure facilitates unique identification of states within a hierarchical state machine by making use of the concept of a *fully qualified state name*. For instance `in_contact` becomes `root.operational.in_contact`. This scheme solves the problem of naming conflicts between identically named states in different composite states.

Hierarchical state machines serve multiple purposes. Firstly, composite states can be used as an abstraction mechanism and to modularise systems. For the sample model shown in Figure 5, the `operational` composite state can be developed and tested independently from the rest of the system.

Secondly, hierarchical states serve to express constraints in an efficient way. For the example above, one of the constraints is the following: the motors are only enabled while in the operational state. This is enforced by the entry and exit actions of the `operational` composite state. Of course a non-hierarchical flat state machine can model the same behaviour. However, in that case it would be necessary to add transitions from all substates of `operational` to the state `safe_mode`, thereby preventing the internals of the operational state to be modelled separately from the safety measures.

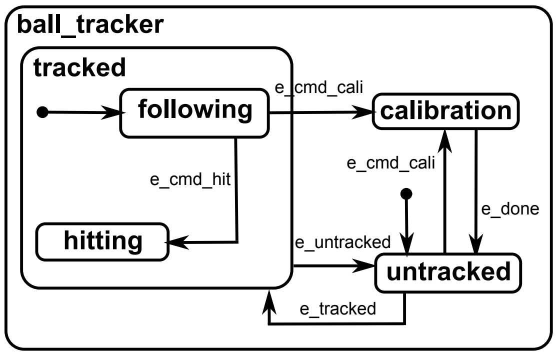

Fig. 6. Boundary crossing.

Lastly, hierarchy is also used to define priority among groups of transitions. This is discussed in Section 5.3.

The rFSM model supports hierarchical state machines and requires that these can be composed within each other.

4.2.1 Boundary Crossing Transitions

A boundary crossing transition is defined as a transition crossing a composite state's boundary. An example is shown in Figure 6: the transition from state `following` to `calibration` crosses the boundary of the `tracked` state. There exists some controversy in literature about this type of transition. Simons [11] strongly suggest to prohibit boundary crossing for the reason that such transitions violate encapsulation of the nested state. On the contrary, UML 2 and Harel statecharts permit and encourage boundary crossing. According to Simons, one motivation for permitting this type of transition in UML was to provide a way to signal multiple accept conditions of a composite state by directly adding transitions from within the composite state to the respective target states. Meanwhile, this issue has been addressed in UML version 2.1 by introducing exit points. The latter model element provides an explicit way to specify multiple accept conditions of a composite state. The rFSM model supports exit points by means of the generic connector, as described in the following Section.

Nevertheless, an important use of boundary crossing is to support *structural priority*. This concept resolves many cases of conflicting transitions that would otherwise result in non-deterministic behaviour. Structural priority is discussed in more detail in Section 5.

Furthermore, taking a closer look at the ownership relation between states and transitions reveals that boundary crossing does not necessarily violate encapsulation as described by Simons. The frequently made, wrong assumption is that transitions are owned by their source state. Under that assumption the boundary crossing transition of the example indeed violates encapsulation of the `tracked` and `following` states, because it introduces a dependency on a state (`calibration`) not part of the composite. However, if this transition is instead owned by the state `ball_tracker`, which is parent of both

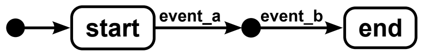

Fig. 7. UML Junction.

tracked and calibration, the problem vanishes. From this perspective transitions are viewed as layers added by surrounding composite states, thereby permitting to extend the enclosed behaviours without breaking their encapsulation. The recent version 2.3 of UML adds a paragraph making a similar observation [18, p. 583]:

> The owner of a transition is not explicitly constrained [. . .]. A suggested owner of a transition is the LCA of the source and target vertices.

The least-common ancestor (LCA) of a transition is the deepest nested state which is a parent of both transition source and target. To ensure construction of modular and reusable statecharts the rFSM semantics impose a stronger constraint than UML: it is *required* that transitions are owned by no state less nested than the transition LCA. This ensures that no state can contain a transition with an unresolved reference to a state.

The LCA concept also plays an important role for describing the execution semantics of hierarchical state machines (Sec. 5).

4.3 UML Pseudo-States

UML state machines introduce several types of so called Pseudo-States with special semantics.

4.3.1 Initial State

The UML 2 initial pseudo-state defines the sub-state of a composite state that is entered by default when a transition ending on the composite state is executed. UML permits only one transition to emerge from an initial state, that furthermore may not define a guard condition. Apart from these constraints, the initial state is semantically equivalent to the junction model element described below.

4.3.2 Junction and Choice

The junction (available both in UML and STATEMATE) is used to create *composite transitions* by chaining together multiple elementary transitions, as shown in Figure 7.

Composite transitions formed by junctions are evaluated statically, meaning that all elementary transitions up to the next state are checked *before* the composite transition is enabled. Only when the conjunction of all transition elements is enabled[5] the composite transition is too. Junctions permit to create splits and merges, that for instance are used in the *dispatcher* pattern illustrated in Figure 8.

This pattern employs a junction as the entry point of a composite state. Internal to the error state, further dispatching to specific handler states takes place using multiple outgoing

5. A transition is enabled when triggered by the current events and the guard condition evaluates to true.

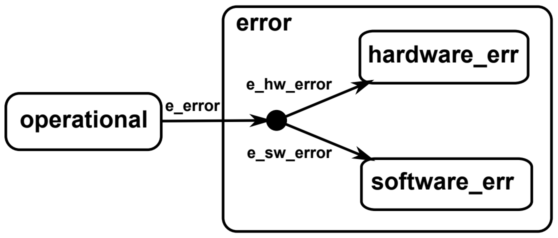

Fig. 8. UML Junction with split.

transitions. It is the responsibility of the state machine designer to ensure that only one enabled path is found. Otherwise the state machine will make an arbitrary choice and thus become non-deterministic. Mechanisms to avoid conflicts are discussed in Section 5.3.

UML Choices are similar to junctions, differing only by being *dynamically* evaluated by the transition logic while the transition is already being executed. Thus, choices have the dangerous property that composite transitions can get stuck during execution if at some point none of the outgoing transitions are enabled. To avoid this, the UML standard advises to always include an *else* transition, that is automatically enabled if no other transition is true.

4.3.3 Entry and Exit Points

Entry and exit points were introduced in version 2.1 of the UML standard with the goal of permitting to define multiple ways to enter a composite state, or to exit it with different outcomes, respectively.

Examining the semantics of entry and exit points reveals that these are semantically almost identical to the junction element described above. According to [18, p. 551], only one outgoing transition is permitted from the entry point to a state, while a junction permits multiple. In our opinion, there is no need for this constraint. Hence the only remaining difference is concerned with the graphical representation of whether the circle is drawn on the border of the enclosing composite state or not.

4.3.4 The rFSM Connector: unifying Initial, Junction, Entry and Exit Pseudostates

Given the large semantical similarities between the UML 2.1 initial, junction, entry and exit pseudo-states, the rFSM model unifies these under the name *Connector*. This unification significantly reduces the amount of concepts, while still providing the same semantic expressivity. The name Connector follows the suggestion of Simons [11], who pointed out that the term state (even with the "pseudo" prefix) is misleading. This is because the element in question does not represent a state (defined as a distinguishable condition of the system) but rather constitutes part of a transition.

As entry and exit points represent public interfaces to a state machine, the rFSM adds a Boolean-valued attribute public to the connector element. This serves two purposes: (i) it can be used by a model checker to issue warnings if a state machine designer adds transitions to non-public connectors from outside of a composite state, and (ii) this attribute may be used by graphical visualisation tools to draw public connectors according to the UML visual guidelines for entry and exit points, which is on the border of a composite state. In the following we represent connectors as small circles: initial connectors are filled in black while all others are not filled. This avoids the ambiguity in UML resulting from both initial and junction pseudo-states being graphically depicted as black, filled circles. If this notation were used instead of UML, the middle junction in UML diagram 7 (connecting `start` with `end` would be not filled, since it is not an initial connector. The junction in diagram 8 would move to the border of `error` state and be likewise not filled, unless it actually is the initial state.

Because of the dangerous property of permitting *stuck* transitions, the UML choice pseudo-states is not included in the rFSM model.

4.3.5 Final State

Transitioning to a UML final state means that the enclosing state has completed. In contrast to the initial pseudo-state, the final state is not a pseudo-state at all, but a specialised regular state. The reason for this is not explained in the standard, however it can be assumed that this is because a well-formed UML state machine can actually *be* in the final state. In contrast, it is not possible to be in the initial state, since this is part of a transition. When a final state is entered, UML requires a *completion event* to be raised that may trigger transitions emanating from a parent state. If no transitions are enabled, the final state remains active.

The rFSM model does not introduce a special final state. Instead an empty, user-defined regular state named *final* can serve this purpose. When no `do` function is defined, the completion event is raised immediately after entry and can hence be used to trigger transitions emanating from parent states. Major semantic differences exist between UML and rFSM with respect to the *completion event* itself; this is discussed in Section 6.4.

4.3.6 Terminate

The UML *Terminate* pseudo-state offers a mechanism to terminate the execution of the entire state machine instance by transitioning to this connector. No actions are executed except those associated with the transition. In our opinion, there is no benefit of using a Terminate model element instead of a regular state called `terminate` that executes the required shutdown procedure. Hence the rFSM model does not include a terminate pseudo-state.

4.3.7 History States

The *History state*, graphically denoted by a circle around a capital **H**, is an initial connector with special semantics.

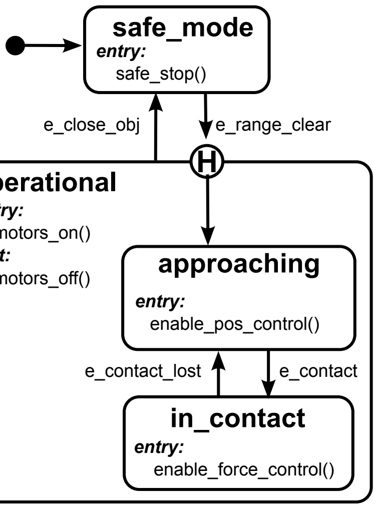

Fig. 9. UML History states.

Consider the example of Figure 5. After returning from the state `safe_mode` (due to an interrupt caused by receiving the `e_close_obj` event) the `operational` state is reentered. In this model the execution of `operational` will be restarted, by re-executing the transition from the initial to `approaching`. However, in some applications it is preferable to *resume* the execution instead of restarting it. For instance this could be the case if the force-controlled operation is to be performed only once and the robot was paused in a position ready to resume. To achieve this behaviour, the initial connector is replaced by a history connector, as illustrated in Figure 9.

The target of the `e_range_clear` transition is now a history connector of the `operational` state, from which a second transition is defined to the `approaching` state. The latter defines the *default* state to enter when a composite state is entered via a history connector for the first time. So far, the state machines shown in Figure 5 and 9 behave identically. However, on subsequent entries of operational through the history connector, the last active state configuration (i.e., the one active at the time when `operational` was exited) will be restored. This effectively results in resuming the previously interrupted task.

UML 2 defines two types of history states: *shallow* and *deep*. Transitions to the former result only in restoring the active configuration for states at the same level as the history connector within the composite state. In contrast, transitions to the latter type result in restoring the complete state configuration including all substates.

History states permit (possibly recursively) resuming previously preempted activities. This pattern is very common in robotics, both at the lower, hard real-time level (such as for implementing safety mechanisms) and at higher levels to model reactive agent architectures.

The rFSM model includes support for both shallow and deep types of history connectors by including a generalised version offering a configurable depth parameter, that defines up to which substates the active configuration is restored. Additionally, taking advantage of the *Codel* based model of computation described in Section 4.1, a Boolean-valued `hot` attribute is introduced. A hot history connector will not only restore the active state configuration but also resume the execution of the `do` codels at the point where it was preempted. In contrast to UML, where a history connector always functions as an initial connector, rFSM permits defining these attributes separately.

4.3.8 Fork and Join

The UML fork and join pseudo-states are used to create and merge concurrent transitions in the context of parallel states: implicitly when entering and exiting parallel regions, and explicitly to synchronise different parallel regions. As discussed in detail in Section 7, the rFSM model does not include parallel states, and hence also does not require fork and join model elements.

4.4 State Machine Extension

UML 2 supports the concept of inheritance of state machines. A derived state machine may override different aspects of its super state machine. This way, inheritance permits defining *abstract* state machines[6] that function as a common interface for specialised versions. Inheritance may also improve reusability by facilitating redefinition of existing models to support new use-cases that otherwise would have required developing a new model.

In contrast to the previously discussed model elements, extension is a meta-level feature that provides an alternate way to specify state machines. It does not influence the semantics of concrete models. Therefore, rFSM does not include primitives to support the definition of derived state machines. This is because adding inheritance would pollute the core model with a feature that can be easily added as an extension, for instance, as a *development tool* that generates a derived model from a super state machine and a list of overridden properties. Since the rFSM reference implementation uses the fundamental Lua table data-structure for specifying models, redefining, removing or adding elements is easily possible.

5 Execution Semantics

While the previous Section predominantly deals with *structural* aspects of state machine semantics, this Section focuses on the *behavioural* aspects, i.e., the execution semantics. To this end, some terminology is introduced. As explained in Section 4.2, the *active configuration* is the set of states that is active at given point in time. A *full transition* [13, p. 302], is a transition starting and ending on a state. A full transition

6. Not to be confused the Abstract State Machine (ASM) formal method.

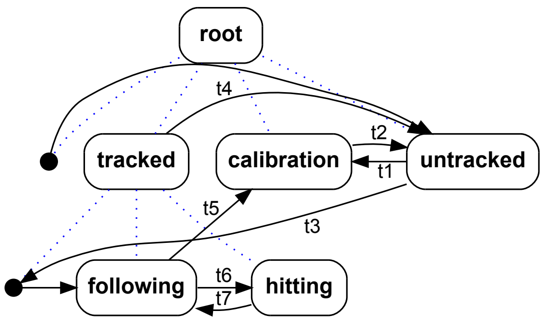

Fig. 10. Scope of transitions.

can be either *simple* or *composite*, the latter consisting of a sequence of simple transitions joined together by connectors. The *scope of a transition* [13, p. 309] is defined as the lowest common ancestor of source and target vertices of a transition.[7] This is best illustrated using a state machine visualised as a tree as exemplified in Figure 10.

This Figure shows a similar FSM as in Figure 6. The dashed lines represent the containment relationship between composite and contained states, the arrows represent regular transitions. For instance the scope of the transition `t4` from `tracked` to `untracked` is `root`, as both source and target states are direct children. The scope of transition `t5` is also `root`, because the least common ancestor state is root.

The scope of a transition is essential for statechart execution semantics because it determines how the active configuration changes when a transition is taken.

5.1 Fundamental Execution Semantics

The basic mode of operation of a state machine is to transition in response to events received. UML 2 describes the *run-to-completion* assumption that defines that a new event will only be processed *after* the processing of the current one has completed. This conforms to the STATEMATE semantics that require the following: *"Reactions to external and internal events [. . .] can be sensed only after completion of the step"* [13, p. 298]. For both semantics, this implies that in order to react to new events, the state machine must be in a stable state; a stable state being defined as the execution of exit action, transition effect and entry action having completed.[8]

Beside these similarities, major differences exist with respect to how and which events are selected for computing a transition, how conflicts between simultaneously enabled transitions are avoided or resolved, and what the semantics of events are.

7. For state machines with parallel states the definition is extended to: lowest common non-parallel ancestor.

8. It is theoretically impossible to encounter a STATEMATE statechart during transition, since transitions are assumed to take zero time. This is the so-called *perfect synchronicity hypothesis*, which seldom holds in modern robotic systems.

5.2 Event Selection

In UML 2 events are stored in a global queue from which they are removed one at a time. The standard does not specify the exact procedure [18, p. 574], in order to permit modelling of different priority-based schemes. After an event has been selected, it is used to compute the set of enabled transitions. If none are triggered, the event is discarded.

The classical statechart semantics differ from this by taking *all* events that occurred since the last transition into account for computing the set of enabled transitions. These events are discarded after the step.

5.3 Computing the Enabled Transition Set

Next, the selected event(s) are used to compute the set of enabled transitions given all transitions emanating from the current active configuration. With both UML and Harel semantics it is possible that the set of enabled transitions contains more than one transition. Apart from certain special cases,[9] this condition signifies a conflict: executing both transitions would result in an invalid configuration of the state machine as multiple non-parallel states would become active simultaneously.

Because conflicts are likely to happen, it is necessary to resolve these by applying rules. The concept of *structural priority*, introduced by Harel [13, p. 328] largely resolves conflicts by assigning priorities to transitions based on their location in the state machine tree. The initial approach was to assign higher priorities the higher the position of the transition LCA was in the hierarchy of states. Later versions of STATEMATE simplified this by using the transition source state instead of the LCA. This priority rule solves most conflicts; the remaining can be explicitly solved by assigning so-called priority numbers to transitions or by using a domain specific mechanism realized as a guard condition.

UML state machines adopt Harel's approach of structural priority although the order of priorities is reversed. This means that transitions with deeper nested source states are assigned higher priority than less nested states. The reason for this deviation is not explained in the UML standard but presumably has its cause in the inherent object oriented focus of UML. Assigning higher priority to deeper nested states permits substates to be interpreted as specialisations that refine the behaviour of parent states.

5.4 Discussion

The different focus of UML state machines and Harel statecharts become apparent by examining both execution semantics. The UML semantics, being part of an Object Oriented modelling standard are more focused on introducing object oriented concepts in state machines. In contrast, the original Harel semantics are targeted towards building complex, reactive systems. Hence, it comes as no surprise that the classical STATEMATE semantics are often better suited to model the behavior of complex, multi-robot systems, as is explained in the following.

Firstly, the behaviour of a classical statechart is largely predictable from the graphical model since the next transition to be executed depends only on the active configuration, the set of input events and the state of the guard conditions. This supports developers that can rely on the graphical model to understand and predict the statechart behaviour. This is not as simple for an UML model, that requires a developer to have additional knowledge about the particular dequeuing implementation and about the currently deferred events (discussed in Section 6.7).

Secondly, taking all events into account for determining which transitions to execute has the advantage that the highest-priority events take effect while lower-priority events are ignored. Moreover, the queue of events is less likely to overflow by being emptied on each step. This is important for complex systems for which floods of events are possible.

Thirdly, the classical statechart approach of conflict resolution supports *compositional robustness*. The example in Figure 5 illustrates this. Assume the developer of the nested state machine `in_contact` adds a self-transition for the event `e_close_obj`. This transition conflicts with the top-level transition from `operational` to `safe_mode`. In the Harel semantics this does not affect the behaviour of the state machine as a whole, since the (safety relevant) transition from `operational` to `safe_mode` has higher structural priority. The system is said to behave *compositionally robust* to a minor change (*minor* being defined in terms of depth of the changed state) because a small change results in little or no change of the system as a whole. In contrast, in the UML semantics this change will alter the behaviour of the entire state machine because the new transition will take precedence and prevent transitioning to `safe_mode`. Hence, a minor local change results in a large change at system level. For modelling complex and modular systems this is undesirable.

The rFSM model adopts the STATEMATE semantics of both event selection and computation of the enabled event set. Moreoever, the approach of using the transition source depth to define priorities is adopted as well as the conflic resolution mechanism of priority numbers.

5.5 Evaluating Composite Transitions

One important issue that is scarcely discussed for UML and STATEMATE semantics is the *extent* to which composite transitions are checked prior to being executed. Starting from the example show in Figure 5 and assuming that `safe_mode` is active and the event `e_range_clear` is in the queue, the transition from `safe_mode` to `operational` is enabled. The question posed by this scenario is whether the state

9. Multiple transitions entering a parallel state are simultaneously enabled *without* being in conflict.

machine logic may already start executing the transition at this point, or if further checking is necessary. The imminent danger of starting execution is that after the first part of the transition has been executed, the second part from the initial connector to the `approaching` state is *not* enabled, hence resulting in the FSM getting stuck. UML 2.1 circumvents this problem by prohibiting initial transitions to define guard conditions, and by introducing a semantic variation point [18, p. 560]. The latter leaves open how to interpret transitions to composite states without initial connectors. It is then up to tool-implementers to decide if this is to be treated as an ill-formed state machine or as the intention to enter a composite state but none of its substates.

Regarding this issue, there are differences in STATEMATE between simulated and generated code [13, p. 303]: while the simulator will not begin executing the transition, the generated code will, and hence get stuck when the continuation transition is not enabled.

The rFSM model deals with the above-mentioned UML variation point by defining a transition to a composite state without an initial connector as ill-formed. Secondly, *deep transition checking* is required, meaning that for a transition to be enabled the complete path until reaching a leaf state must be enabled, thus including zero to many transitions from initial connectors to states. This eliminates the possibility of transitions getting stuck during execution. Because the rFSM model treats initial connectors and junctions identically, no special logic is required. Moreover, the UML constraint that forbids initial transitions to define guard conditions becomes unnecessary and can be dropped.

5.6 Transition Execution

Once a transition is enabled, it is executed in the following sequence. Firstly, all source states up to, but excluding, the LCA are exited by invoking their exit functions. For each composite state of these states, the last active sub-state is stored in preparation of a potential re-entry via a history connector. As an example, for a transition between two states within the same composite state only the source state is exited, as the LCA *is* the composite state. In contrast, for transitions contained by different composite states, multiple states are exited.

Secondly, the transition effect is executed, followed by the third part of the transition execution in which the transition target state including its parent states are entered. If this target is a leaf state, the transition execution is completed. Otherwise, the transition execution is continued at the initial connector of the composite state until a leaf state is reached. In case of composite transitions this procedure is executed for each individual transition.

As an example, assume that the active state of the statechart in Figure 6 is `following`, and the event `e_cmd_cali` occurs. The LCA of this transition is the root state (`ball_tracker`), hence first `following` and `tracked` will be exited. Next, the transition effect (not used in the example) will be executed. Lastly, the target states—for this example, only `calibration`—are entered.

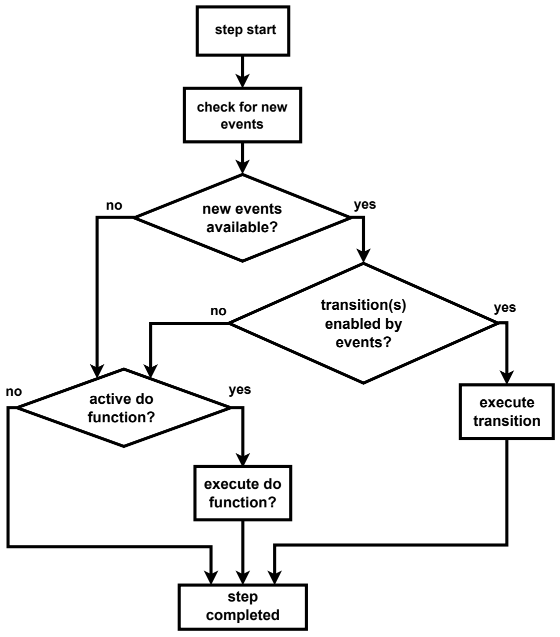

Fig. 11. Flowchart of the step procedure.

5.7 rFSM Transition Semantics

Figure 11 summarises the process of executing a rFSM step. Executing a step will advance the state machine in an atomic way, leading to, at most, the execution of one composite transition.

Each step begins with the retrieval of all events that have accumulated since the last step. Next, if new events are available, it is checked whether these trigger any transitions. If yes, the one with the highest structural priority is selected and executed. If no events exist, the rFSM core checks if the currently active state has an enabled `do` function to execute. If yes, it is run. Otherwise the step is completed.

To complement `step`, a mechanism called `run` is provided: when a state machine is *run*, the step procedure is invoked either until no new events are available and the active state has no enabled `do` function, *or* a maximum number of steps (given as an argument to run) have been executed. The practical application of step and run is described in detail in Section 9.

An active `do` function can be configured to be in one of two modes: idle and non-idle. Which one is appropriate is strongly influenced by the model of state machine progression (Sec. 9.1). If a `do` function is non-idle, the state machine engine will recall it immediately, provided that no events enabling outgoing transitions were received. Conversely, an idle `do` function is not recalled immediately, but only during the next `step`. In general, the choice of mode should depend on whether the `do` activity is used to complete a task as fast as possible, or instead is just used to periodically perform some task.

6 Event Semantics

Previous Sections described how events enable transitions and what are the ramifications thereof. This Section examines the semantics of events themselves, their structure and specialisations.

In classical statecharts, such as the STATEMATE semantics, events can be understood as messages which represent the occurrence of something. Events can be compared to events specified on transitions and are hence required to have identity. UML includes this basic event type using the name *SignalEvent*. However, there exist other, more specialised types of events which are discussed below.

6.1 UML ChangeEvent

A *ChangeEvent* is raised when a Boolean-valued condition evaluates to true. For this, the syntax `when <condition>` on a transition is used. In principle, such an event type seems useful for robotics coordination. The problem however, is that the UML specification leaves open when and how the condition is evaluated [18, p. 452]. Obviously, for a hard real-time robotic system this is not acceptable, as the condition checking will directly influence the worst-case latency of the event generation and hence the transition execution. This situation can be remedied in two ways. Either the ChangeEvent is extended to take into account the additional semantics of evaluation frequency, timing precision etc., or this event type is excluded from the core semantics and realized outside of the statechart. In the latter approach the validation is carried out by a separate component that raises the respective events. For rFSM the latter approach was chosen, as it does not require extensions to the core semantics and makes the involved assumptions explicit. Moreover, many computational components such as robot driver components, control components or estimators can easily provide such validation of internal conditions and the respective event generation with little computational overhead. Therefore, it is considered a best practice to extend computational components to raise events based on configurable constraints on their internal state.

6.2 UML Time Event

UML defines a *TimeEvent* as an event which is raised at a relative or absolute point in time. The syntax used is `after <duration>` or `at <time>`. For robotic applications TimeEvents can be useful in various circumstances, yet they suffer from a similar limitation as ChangeEvents, as no assumptions can be made on the qualitative properties of the time used. Hence, the variation point can be resolved likewise, by either extending the TimeEvent or by implementing it outside of the scope of the core semantics. rFSM takes the latter approach and does not include TimeEvents in the core semantics, but instead as a configurable plugin.

It is worth pointing out the danger of misusing time events for modeling flowchart-like execution flow [11]. A state should reflect a distinguished condition of the system or a part thereof, not a time-bounded computation. The need for the latter is an indication to use a Flowchart formalism such as UML Activity Diagrams. Conversely, TimeEvents have important use-cases, most notably to model timeouts that trigger transitions to states for dealing with the absence of the nominally expected event.

6.3 UML Call Event

A CallEvent is an event defined for state machines used in the context of Object Oriented software systems. This event represents a request to invoke a certain method on an object. The CallEvent is generated after the operation has been invoked, thereby permitting the FSM to track the methods that are invoked. As the rFSM semantics do not make any assumptions about the underlying programming paradigm, no CallEvent is included.

6.4 UML Completion Event and Final State

According to UML, a *completion event* is raised either when the `do` behaviour of a simple state completes or a final state is entered. Graphically it is depicted by an *unlabelled transition* (called completion transition) emanating from a simple state or from a composite state that contains a final state.

The UML completion event has special semantics and is *implicitly* assigned highest priority among all events. For the UML semantics this is necessary, as a CompletionEvent does not carry any information about which state completed. Therefore, this event is only valid for the active configuration of the FSM at its time of generation. Queuing a completion event would permit the triggering of different unlabelled completion transitions from other states at a later time, which would be undesirable.

The rFSM model avoids this problem by adding identity in form of the fully qualified state name (Sec. 4.2) to the completion event. For instance, the completion event of the `in_contact` state of Figure 5 would be `e_done@root.operational.in_contact`. This way no special treatment of the completion event is necessary. Moreover, by *not* assigning special priority to this event, priorities are respected as a completion event must compete with all other events stored in the event queue. Finally, the readability of completion transitions is improved by labelling these explicitly. The semantics of unlabelled transitions are discussed below.

As in UML, the rFSM model requires that the completion event is raised either when the `do` function of a leaf state has completed, or immediately if no `do` is defined. The use-case of UML final states is covered in rFSM by using an empty, regular leaf state named *final*. Once this state is entered, the completion event `e_done@final` will be raised. Transitions can then be defined to react to this event.

6.5 UML AnyReceiveEvent and unlabeled Transitions

As the rFSM model (unlike UML) chooses to label completion transitions explicitly, the question of the semantics of unlabelled transition arises. There exist at least two natural interpretations of such a transition. The first is a transition that is never enabled, because no events are specified that could trigger it. The second, more useful interpretation is a transition that is enabled by any event. Harel statecharts opt for this interpretation because such *true-transitions* can be useful: when chaining multiple transitions by means of Connectors, true-transitions permit to avoid repeating the enumeration of the events specified on the previous transitions for all subsequent transitions, thereby reducing redundancy. A second use-case is to avoid having to exhaustively enumerate all possible events for a transition that is only constrained by a guard condition. For UML state machines this interpretation can be realised by specifying the *AnyReceiveEvent* which is denoted by the keyword `any`. Because simpler and arguably more intuitive, the rFSM model adopts the Harel interpretation of the unlabelled transition as a transition being triggered by any event.

6.6 Edge- and Level-triggered Events

In the simplest case, an event represents the one-time occurrence of something. Based on all events that occurred since the last step, the FSM core reasons to find and execute a transition and afterwards drops these events. However, in some cases events are not only valid at a particular time instant, but persist during a period of time and/or until some action takes place. These two types of events are called edge and level triggered events, respectively.[10] The first are only valid at one point in time (on the rising or falling edge) while the second persist for some time, i.e., as long as the level remains *high*.

Level triggered events are often useful to signal a condition requiring some form of response. For example this could be a slight over-temperature condition of a motor during a manipulation task. Eventually, it is necessary to react to this event and reduce the overall speed, however this can take place after the current manipulation has completed. Moreoever, level-triggered events permit expressing the use case of deferred events, which are discussed below.

Level triggered events are quite similar to STATEMATE conditions [13]. The main difference is that in STATEMATE a condition value is cached before each step while the rFSM semantics make no assumptions in this regard. The recommended way to realize a level triggered event in rFSM is by means of a guard function checking the condition in question. The details of how this condition is realized strongly depends on the underlying software framework in use.

10. In analogy to edge-triggered interrupts in operating systems.

6.7 Deferred Events

The UML standard permits a state to define a list of *deferred events*. When such a state is active, receiving a deferred event will not trigger any outgoing transitions. Instead this event remains in the queue until a state not deferring it becomes active. Since deeper nested states deferring events take priority over less nested states [18, p. 576], deferred events can be interpreted as an object-oriented specialisation mechanism which permits refining the behaviour of less nested states. Because rFSM adopts STATEMATE's priority approach to resolve conflicts, a deferred event mechanism would not violate the principle of compositional robustness. Nevertheless, the rFSM model does not include deferred events because (i) this primitive obscures the hierarchical conflict resolution, and (ii) the use-case of deferred events can be accommodated by a *level triggered* event. This is because a deferred event is effectively a level triggered event in disguise; it remains active (stored in the queue) until it is convenient to be processed. By using level triggered events, no special defer primitive is required and no additional, hidden state is introduced in form of a queue of deferred events.

7 Concurrency Semantics

State machine parallelism was introduced by Harel and is currently supported by most specifications, including UML, SCXML and Simulink Stateflow. This feature permits more than one substate or state machine contained in so called orthogonal regions to be active, and therefore the concurrent execution of their associated behaviors.

As robotic applications are inherently concurrent, the concept of parallel states is generally suitable for modelling these systems. Nevertheless, the weakly specified semantics of the parallel state element introduce several ambiguities. For instance, according to the UML standard [18, p. 575], no assumptions are made about the underlying thread environment, even though this greatly influences both performance and real-time properties of the executed model. Furthermore, the level at which concurrency is implemented is left undefined. In the simplest case, parallelism may only mean simultaneous execution of `do` behaviours, whereas more elaborate implementations might choose to implement concurrent entry of individual orthogonal regions. The latter would require introducing more assumptions: for instance it has to be decided which of the different regions is entered first and whether the entry is executed interleaved or sequentially.[11]

11. The latter example only holds true if multiple composite transitions emanate from the same fork.

The analysis of existing coordination state machines shows that robotic coordination, unlike computation, rarely requires such tightly coupled concurrency. This is due to the fact that coordinative actions generally consist of issuing commands to lower level, concurrently running computation components. Therefore, coordination does not benefit from parallelization in terms of performance as intensive computations do.

On the other hand, supporting distributed state machines is more important to robotics than the previously mentioned internal concurrency. This is particularly true for multi-robot applications. Distributed state machine instances must be able to observe each others' state and share some events. This approach has several advantages compared to parallel state machines: essentially, the complexity of parallel states and the associated assumptions on the threading environment can be avoided. Furthermore, the aspect of communication is moved out of the implementation and thereby made explicit, which is a best practice advocated in the 4Cs design paradigm. If, for instance, the communication between two FSM breaks, an explicit event can be raised. The system architect can select a communication middleware suitable for the particular purpose. The downside is that additional effort is required to deploy such a distributed state machine. However, this process is generally suited to be automated by deployment *tools*.

A generic pattern of distributed sub-states is described in Section 9.5.

8 Reference Implementation

To illustrate the applicability and the exact semantics of the described model, a complete reference implementation of rFSM statecharts has been developed[32]. As of today, this execution engine provides support for all mechanisms described in this paper, apart from internal events and history connectors, which are expected to be added soon.

This rFSM engine is implemented as an *internal domain specific language* (DSL) [33] in the Lua programming language [34]. Internal DSL are built on top of an existing programming language, while external DSL are developed from scratch. By reusing existing infrastructure, internal DSLs are significantly easier to create and maintain than external ones. Moreover, internal DSLs can very easily be combined with programs of the host language or even with other DSLs. This extensibility is an important requirement for a state machine implementation, whose major purpose is to execute user defined actions according to the specified state machine model. The only disadvantage of an internal DSL is that its syntax is constrained by the host language. In practice this limitation is often acceptable given the reduction in development time and the simplicity gained for combining DSLs together.

Besides being suitable for building DSLs, the Lua language was chosen for the following reasons. Firstly, the language is designed to be both embeddable and extensible, which is reflected by the small memory footprint and straightforward foreign function interface. These properties are important since distributed coordination implies multiple Lua instances executing state machine instances embedded within components of a robotic software framework. Secondly, Lua offers a simple syntax that facilitates less experienced programmers not familiar with C++ or Java to build state machines; nevertheless, the language is mature and semantically well grounded by being strongly influenced by the Scheme language [35]. Moreover, the use of a scripting language contributes to the robustness of a system, because scripts, in contrast to C/C++, can not easily crash a process and thereby bring down unrelated computations executed in sibling threads. This property is essential for the aspect of coordination, that, as a system level concern, has higher robustness requirements than regular functional computations.

The following Listing shows the textual input model corresponding to the model of Figure 2.

```
return state{
   following = state{},

   paused = state{
      entry=function() ArmController:stopArm() end,
      exit=function() ArmController:startArm() end
   },

   initial = connector{},

   transition{ src='initial', tgt='following' },

   transition{ src='following', tgt='paused',
               events={ 'e_untracked' } },

   transition{ src='paused', tgt='following',
               events={ 'e_tracked' } },
}
```

Listing 3. Textual model of Ball-tracking FSM.

This model is a valid Lua program representing a tree of states constructed using the `table` data type. When loading this file, the Lua interpreter performs basic syntax checking and instantiates the data structures. Note that this requires no effort by the DSL developer; it comes for free for an internal DSL. Next, the rFSM engine carries out basic validation and transformation of the tree structure to a graph ready for efficient execution. The validation is very limited and only concerned with detecting common structural errors. Formal verification of Statecharts has been treated in literature [36], [37], [38] and outside the scope of this work.

Composition of states is supported through the `rfsm.load` primitive. For instance, to reuse an existing `following` state machine defined in a file, line 2 of Listing 3 could be defined as follows:

```
following = rfsm.load("following_fsm.rfsm")
```

As illustrated by example 3, a state machine developer will use Lua functions to implement behaviour in form of `entry`, `do`,[12] `exit` and `effect` programs. Calls to low-level C/C++

12. In the reference implementation `do` is renamed to `doo` to avoid conflicts with the homonymous Lua keyword.

are easily integrated by means of the foreign function interface, as described below for the OROCOS/RTT framework.

Hooks are the key mechanism to extend and embed rFSM statecharts by means of custom, user defined functions. The most important hook is `getevents`, that allows customizing where events are retrieved from. A `getevents` function is expected to return a list of events that occurred since the last invocation and will be called by the rFSM engine one or more times during the execution of a step. The example in Listing 4 shows a sample getevents hook to retrieve all new events from a port `events_in` in the context of the Orocos RTT framework. The second part illustrates how the FSM is customized with this hook.

```
function rtt_getevents()
   local ret = {}
   while true do
      local fs, event = events_in:read()
      if fs ~= 'NewData' then break end
      ret[#ret+1] = event
   end
   return ret
end

return state {
   getevents = rtt_getevents,
   following = state{},
   ...
}
```

Listing 4. Sample getevents hook.

Since retrieving events from ports is very common when using rFSM with Orocos RTT, the auxillary function `gen_read_events(port1, port2,...` is provided in the module `rfsm_rtt` to automatically generate a getevents hook to read all events from the given ports.

Other hooks include `pre_step_hook` and `post_step_hook` that are called before and after a step is executed respectively. These are lowlevel hooks mainly used by extensions, such the timeevent plugin that checks for expired timers or the event memory plugin described in Section 9.4 to keep track of occurred events. The rfsm module-level `preproc` hook allows registering functions that will be called at initalization time and can be used to preprocess or validate the rFSM model prior to execution. An example use is for transforming platform independent task models to rFSM hooks, as explained in Section 9.2.

8.1 Software Framework Integration

The reference implementation is implemented in pure Lua and has no dependencies whatsoever. This permits standalone use which is convenient for testing and debugging of statecharts. To use the implementation in the context of robotic software frameworks requires a plugin to make the primitives of these frameworks available within Lua. We have developed such bindings for the OROCOS Real Time Toolkit (RTT) [2] that permit interacting with Components, Services, Operations, Input- and Output Ports and data types. A RTT LuaComponent is an initially empty container into which Lua programs can be loaded. This approach permits treating Coordination components just as regular computational components that are configured to load Coordination statecharts. Event driven input ports can be used to trigger dormant components upon receiving events and output ports are used to emit events. Using the standard communication primitives of a framework avoids duplicating these mechanisms for the sake of Coordination.

8.2 Considerations for Hard Real-Time Execution

The RTT framework provides a hard real-time safe execution environment for components. Naturally, real-time requirements exists also at the Coordination level. Satisfying hard real-time constraints for interpreted, garbage collected languages poses several challenges. Firstly, the allocation of memory must take place in a temporally deterministic way, which is typically not guaranteed to be the case for the default memory allocators of general purpose operating systems such as Linux. Secondly, the recuperation of unused memory must take place in a way that does not interfere with the nominal execution.

Our approach achieves deterministic allocation by extending the Lua interpreter to use an O(1) memory allocator [39]. To achieve deterministic recuperation in critical real-time paths, the Lua garbage collector is stopped and manually controlled. Further details can be found in [40] [41].

8.3 Representing Events

In the majority of examples and real applications we have represented events using strings. While the rFSM implementation permits any comparable type to be used, the string representation is convenient for humans and still reasonably fast. Provided that memory is preallocated correctly, string events can even be used in hard real-time. Nevertheless, depending on the application, significant overhead could be avoided by denoting events using numbers. To take advantage of the performance of the numeric representation and the readability of string events, the latter could be transformed to the former at load time using an rFSM pre-processing hook defined as a rFSM plugin.

9 Patterns and Best Practices

This Section discusses reoccurring patterns and best practices in robotic coordination and their realization using rFSM.

9.1 Models of State Machine Progression

An important decision a statechart designer needs to take is to define how and when a statechart is advanced. Harel [13] describes two basic approaches: the asynchronous and synchronous model (see Section 9, “Two models of time”). The synchronous model assumes that one step is executed every time step, thereby causing the state machine to react to events and changes that occurred since completion of the previous step. In contrast, the asynchronous model advances the state machine only upon receiving events and is typically configured to execute as long as events are available. These two models correspond to the paradigms of event-triggered (ET) and time-triggered (TT) systems [42], [43] and have received thorough treatment in literature. In summary, the TT architecture offers several advantages over the ET architecture, including exact predictability of temporal behavior and allowing for systematic formal verification of temporal properties. In contrast, ET-systems generally require substantial testing to ensure that deadlines are met. Unfortunately, open and uncertain environments common in robotics require significant effort to determine the necessary granulation of observation lattice and maximum execution times [42].

While the rFSM model and reference implementation support both models, each model has different use-cases for which it is appropriate. The asynchronous model is best suited for coordination scenarios in which multiple components, possibly running at different frequencies, are coordinated. To this end, a dedicated Coordination component with its own activity is introduced. When new events are received, the state machine component is woken up and run (via the `run` function) and permitted to execute until it goes idle (which happens if there are no events in the queue and there is no active `do` function). Input events can originate from various sources: user commands, error events from computational components or filtered and processed raw events. Outputs may consist of performing actions such as starting or stopping components, creating or destroying connections, invoking component services, or configuration of parameters.

The synchronous execution model is suitable for coordination that takes place at a fixed frequency. The typical use case is coordination of a single component. For each cycle of the computation, the `step` function of the FSM is invoked to advance the state machine. This results in at most one transition being executed, taking into account all events that occurred since the last step. In contrast to the asynchronous model, the coordination can be executed conveniently within the activity of the coordinated host component. For instance, a PID controller component can be decorated with a discrete task-aware coordinator that monitors and adjusts control parameters according to the current task state.

In practice, hybrid models combining both asynchronous and synchronous advancing of state machines have often proven to be useful. This behaviour can be achieved by connecting a periodic timer component in addition to other asynchronous event sources to the incoming event port of a coordination component. This way, the coordinator is guaranteed to wake up for processing at a minimum rate defined by the timer component. However, if events arrive in between timer events, the coordinator will react instantly.

Note that such timer events should be understood in the broadest possible sense of time: the events do not have to be emitted at fixed intervals of real-time, but may be raised according to a virtual, task-specific clock.

9.2 Platform and Robot independent Coordination

For reusing coordination models on different robots and with different software frameworks, it is necessary to avoid introducing dependencies on these aspects. A simple way to achieve this is to encapsulate the platform specific functions used by the FSM in modules. A supported platform must then provide implementations satisfying the required FSM API.

The downside of this approach is that platform independence is only achieved for the FSM, but not for the behavior hidden in the opaque functions. Platform independence *including* this aspect becomes possible if the behavior of states can be formally modelled in a platform independent way. To this end, the statechart model is specified such that states *reference* platform independent task models. By means of a plugin realized using a rFSM pre-processing hook, the platform independent task models are dynamically transformed to platform specific rFSM hook functions at initalization time. The following example illustrates this transformation for a statechart coordinating end-effector motions. The motion specifications are expressed in the task frame formalism, a hybrid force-velocity control robot programming formalism [44], [45].

Listing 5 shows a rFSM statechart modeling a robot arm moving down using velocity control and aligning upon entering in contact. Motion models are specified in an external module `tff_motions` and not further described here. Examples of the TFF motion DSL can be found here [46]. States reference task models using a keyword `task`. Introducing this is legal since rFSM is implemented as an *open model*, signifying that keywords not part of the rFSM model are ignored instead of treated as errors.

```
require "rfsm_tff"
require "tff_motions"

return state {
  move_down = state {
    task = tff_motions["move_down"],
  },
  push_down = state {
    task = tff_motions["push_down"],
  },
  transition{ src="initial", tgt="move_down" },
  transition{ src="move_down", tgt="push_down",
              events={"e_contact"} }
}
```

Listing 5. rFSM model referencing TFF motions models.

Prior to executing the statechart model, the referenced task models need to be transformed to standard rFSM hook functions. This is achieved using the `rfsm_tff` plugin, shown in Listing 6. This plugin installs `transform_tff` as a rfsm pre-processing hook for carrying out the transformation at initalization time (line 13). This function in turn uses the rFSM `mapfsm` higher-order function to invoke `tff2hooks` on all states of the FSM (line 10). For each state defining a task, `tff2hooks` generates a function (using `rtt_gen_apply` in line 5), that when called commands the TFF controller to apply the given TFF motion using the respective platform specific mechanism. This function is set as an entry function such that this motion will be executed upon entering the state. In this way, the abstract task model rFSM statechart is transformed to an executable, platform specific rFSM instance. Though functional, the example is intended to be illustrative; for instance the Orocos RTT specific `rtt_gen_apply` function should not be hardcoded but become a parameter of the `rfsm_tff` module.

```
 1  module("rfsm_tff")
 2
 3  function tff2hooks(s)
 4     if tff.is_TFFMotion(s.task) then
 5        s.entry  = tff_rtt.gen_apply(s.task)
 6     end
 7  end
 8
 9  function transform_tff(fsm)
10     rfsm.mapfsm(tff2hooks, fsm, rfsm.is_state)
11  end
12
13  rfsm.preproc[#rfsm.preproc+1] = transform_tff
```

Listing 6. Dynamical transformation of TFF model to rFSM hook functions using plugin.

The plugin approach permits easily adding further preprocessing or validation steps. For instance, an additional robot specific plugin could be used to ensure that the used forces and velocities are within the limits of the actual capabilities of the robot.

9.3 Best practice *Pure Coordination*

Most coordination models do not restrict the primitives that can be used in actions, as for instance rFSM does not constrain the `entry`, `do`, `exit`, and `effect` functions in any way. In contrast, a pure coordinator limits its side-effects to exclusively raising events and has the following advantages. Firstly, the reusability of coordination models is increased by drastically limiting dependencies on platform specific actions. Secondly, the blocking invocation of operations on functional computations is avoided, thereby improving the the determinism of the Coordinator. Lastly, Coordinator robustness is increased by avoiding operations that might block indefinitely or crash, either of which may effectively render the coordinator inoperative.

To that end we propose splitting the *rich* coordinator component that executes actions itself into a *Pure Coordinator* and a *Configurator*. Although the coordinator remains in charge of commanding and reacting, the execution of actions is deferred to the Configurator. The Configurator is configured with a set of configurations that it will apply upon receiving the corresponding event. This pattern, called *Coordinator–Configurator*, has been implemented as a Configurator domain specific language for the Orocos RTT framework. More detail on pattern and DSL can be found here [47].

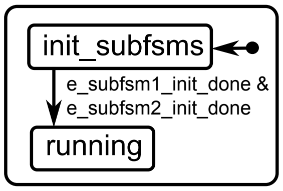

Fig. 12. Dealing with inter-step history.

A pure coordinator requires to be informed via events about relevant changes in system state. This can be achieved in two ways: on the one hand by introducing an explicit monitor component that is configured with constraint-event pairs. When a constraint is violated, the corresponding event is raised to inform the coordinator. The other approach is to extend computational components themselves to raise events when (configurable) constraints on their internal state are violated. In general, the latter approach should be preferred if the constraint is specific to the computation, as it avoids the need to communicate state to a monitor. However, if the constraint is application specific, the monitor component is preferrable, since it preserves reusability of the computational component. This trade-off is further elaborated in the step by step example given in Section 10.

9.4 Event Memory

The default behaviour of statecharts is to avoid any state apart from the currently active configuration; all events are discarded after the execution of a step. One the one hand, this improves the deterministic nature of statechart execution by avoiding hidden state (e.g. in the form of deferred events). Yet, on the other hand, event-only coordination is complicated, in particular for distributed statecharts. Consider the example shown in Figure 12. This coordinator must wait in state `init_subfsms` until it receives an event from each sub-FSM, signalling that initialisation has completed. The problem with this implementation is that the transition will not be enabled, unless both events are received at exactly the same step.

The proposed solution to this problem is to extend each state with memory of the events that were received while this state was active. This permits to check if (and how often) an event has occurred while the transition source state is active. The rFSM reference implementation provides an *event memory* extension in form of a plugin, that when loaded keeps track of this information. That way, the problem can be solved by reformulating the transition using a guard as show in Listing 7. Using the recorded event history, the guard condition will inhibit the transition until both required sub-FSM events have been observed *while* residing in the source state of the transition.

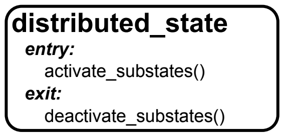

Fig. 13. Distributed state.

```
guard=function (tr)
        return tr.src.emem.e_subfsm1_init_done > 1 and
               tr.src.emem.e_subfsm2_init_done > 1
      end
```

Listing 7. Event memory based guard condition.

Moreover, event memory can support detection of erroneous FSM behaviour, such as reception of too many of a particular event within a time step. A trigger to detect such conditions can check whether the number of a certain error event divided by the number of virtual time-events (Sec. 9.1) does not exceed a threshold.

9.5 Distributed Substates

The following paragraphs describe a generic pattern for implementing distributed statecharts. The approach is generic since it permits expressing concurrency at different levels of distribution, ranging from states distributed over a network to the traditional, closely coupled, thread-level parallelism.

The pattern is illustrated by the state machines in Figures 13 and 14, which are subsequently called the top and bottom half of the generic distribution mechanism.[13]

The first Figure shows a *container*, the top half of a distributed state. Its purpose is to activate and deactivate the (semantically) contained, concurrent substates on entry and exit respectively. Depending on the type of parallel state the entry function will carry out different actions: for a local thread-level distributed state it might spawn a new thread for each substate. For a parallel substate distributed over the network it might connect to a running and waiting instance. Substate activation and deactivation is achieved simply by sending the corresponding events to the bottom half.

Each distributed substate, the bottom half, is formed by a generic substate as shown in Figure 14. This state contains two states `active` and `inactive` that are connected by transitions. The active state *includes* the application specific state machine, that is entered per default via its initial connector once the `active` state is entered. This way, the top half can control the execution of the bottom half by sending it the events `e_sub_activate` and `e_sub_inactivate`. Likewise, by observing the active state of the bottom half the top half can determine when all substates have successfully been entered or exited.

13. Apart from the name, the concept of top and bottom half has no relationship to the (obsolete) interrupt handling mechanism of the Linux Kernel.

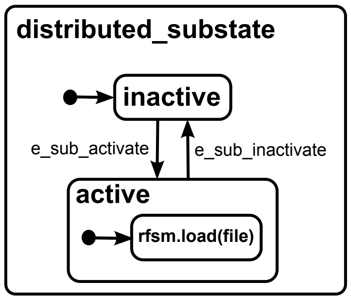

Fig. 14. Distributed substate.

Using this pattern formed by two FSM, any hierarchical state machine can be distributed independently of the form of distribution and obeying the rules of hierarchical statecharts. The only requirement is that events can be communicated between distributed instances.

Communication and Deployment: By distributing state machines as described above, the communication between these instances is made explicit. To implement this, an asynchronous message passing mechanism (to communicate events) is most suitable. Communication is required for notifying a distributed state to activate or deactivate its sub-state machines, for propagating events between the two halves and for notifying the top half about state changes in the bottom half. The latter is necessary, for example, in the exit function of the top half, which must wait for all substates to enter the `inactive` state after sending `e_sub_inactivate` event. This mechanism is similar to the `in(<state>)` *conditions* available in STATEMATE.

An important questions is *which* events are communicated between event sources and sinks (statecharts, but also computational components). Generally, statecharts deal robustly with unused events by simply dropping these. Nevertheless, it is a good practice to limit the communicated events to the set of events that can trigger a peer statechart. Besides reducing the communication load, this avoids accidentally triggering unlabelled transitions and reduces the likeliness of event buffer overflows. In practice, this can be straightforwardly achieved by introducing two (or more) functions `raise_local(eventB)` and `raise(eventA)` that raise events locally or globally to a statechart, respectively.

Moreover, depending on the system it might be necessary to introduce buffers for storing events until they are retrieved by the statechart engine. Whether this is necessary or not depends on the worst-case number of events that can be raised simultaneously by all peer event sources, and the model of state machine progression (see Section 9.1). Complementary to this, *Complex Event Processing* techniques can help to reduce the number of events prior to being fed to a statechart.

A further, important issue concerns the reliability of the communication channel. In case of unreliable communication (such as UDP), event messages might get lost or be duplicated.

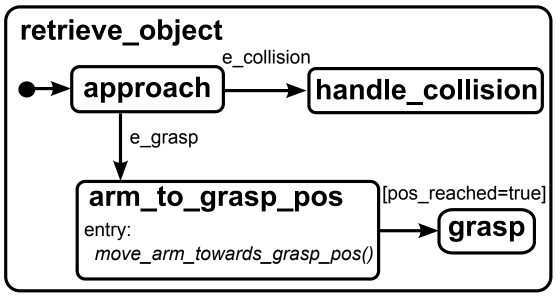

Fig. 15. Object retrieval without Preview Coordination.

Robust statecharts *can* deal with these issues by different means: for instance time triggered `timeout` transitions can re-raise events if the expected conditions are not met; likewise the for the receiving statechart non-nominal behaviour should be dealt with by *explicitly* defining transitions that trigger when expectations are violated.

9.6 Serialised Locally Distributed States

The most common motivation to use parallel (and hence also distributed) sub-states is to improve performance by parallelization. A less common use-case, nevertheless more relevant to the domain of coordination, is for combining behaviours that must be executed concurrently but that shall be kept separated for reasons of reusability (of each of the behaviours individually).

An example for this is the iTaSC-level coordination in the iTaSC framework [48], that requires composition of a generic (to the iTaSC framework) state machine with another state machine defined by the application developer. The generic state machine first triggers all task-level FSMs to run at the right time. The second, user-defined FSM implements the overall task execution. Although both parts must always be executed after each other (first the generic, then the user defined) it is desirable to keep them specified separately to be able to reuse the generic part. To achieve this composition, a state named *serialised-locally-distributed state* (SLDS) was introduced. This name originates from the fact that the substates are advanced one by one in a serialised way in the same activity (thread) as the parent state machine. More concretely, a SLDS state can conveniently be realised as a leaf state that triggers all substates in its `do` function. This way, both FSMs can be composed from separate models at a late stage, during the loading of the SLDS state.

To concisely specify the behaviour of this type of state, additional properties must be defined: in which (partial) order shall the substates be advanced, shall this take place by invoking `step` (and how often?) or `run`? Shall non-idle substates result in the `do` of the SLDS to become non-idle too? As a side note, the considerable number of parameters required by the implementation of this most simple form of parallel state illustrates the complexity introduced by this family of model elements and confirms our approach to exclude these as primitives.

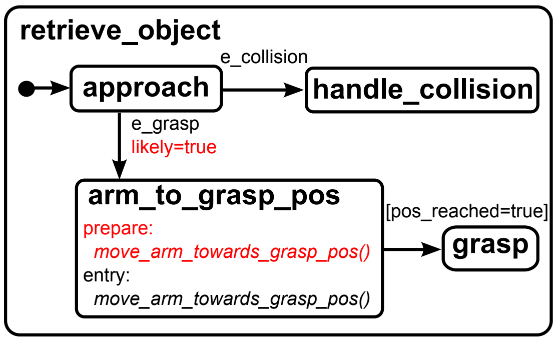

Fig. 16. Object retrieval with Preview Coordination.

9.7 Discrete Preview Coordination

Discrete Preview Coordination is inspired by the concept of *preview control* [49], but also by compiler branch-prediction techniques such as gcc's (GNU Compiler Collection) `__builtin_expect`. The basic idea is to exploit knowledge about the future behaviour of a system to optimise the current actions. In order to apply preview techniques to discrete coordination, we extend the core rFSM model as follows: transitions are extended with an optional Boolean `likely` attribute and states with an additional `prepare` action. While checking for enabled transitions from the currently active states, the preview mechanism will additionally check for transitions that are likely. If such a transition is found, the transition target state's prepare action is executed. This way, the prepare action can be used to prepare the activity of the respective state in expectation that it might be entered next.

The use of preview coordination is illustrated by the example statechart shown in Figure 15. This statechart models a mobile robot skill for retrieving an object. Object retrieval consists of three nominal substates of first approaching the target location, then moving the arm close to the object to be grasped and lastly grasping the object. Additionally, the situation of unexpected collisions is dealt with. For simplicity, details on the realization of `approach` and `handle_collision` are omitted.

The key observations are that firstly, the nominal sequence from `approach` to `arm_to_grasp_pos` will take place most of the time, and secondly, that the behaviors of `approach` and `arm_to_grasp_pos` are neither logically nor kinematically in conflict and can hence be executed in parallel. To exploit this knowledge using preview coordination, the statechart from Figure 15 is extended as shown in Figure 16.

Firstly, the transition from `approach` to `arm_to_grasp_pos` is marked as likely (`likely=true`). Secondly, a `prepare` function is added to `arm_to_grasp_pos` for preparing the states behavior.

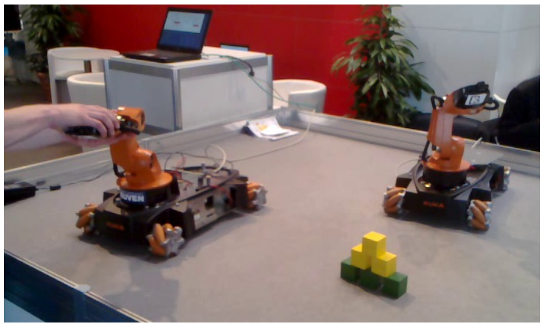

Fig. 17. The dual youbot coupling demo at Automatica.

In this case, the prepare function is the same as the entry function `move_arm_towards_grasp_pos()`. This assumes that this function can be invoked more than once and will immediately return if the arm is already at the desired position.

As a consequence of these extensions, the `prepare` function of the `arm_to_grasp_pos` state is invoked even though the robot is still approaching the grasp position. This in turn causes the robot arm to be moved to a suitable grasp position, thereby reducing or even eliminating the time spent in `arm_to_grasp_pos` after the approaching phase has completed.

It should be noted that the described Discrete Preview Coordination mechanism does not improve the expressiveness of the rFSM model; the same result could be obtained by calling `move_arm_towards_grasp_pos()` from within the `approach` state. The main advantage is, however, that the prepare action can be placed in the context of the semantically related state, namely `arm_to_grasp_pos`. Only this way, both `approach` and `arm_to_grasp_pos` states can be reused independently of each other.

The described preview coordination mechanism has been implemented as a rFSM plugin that extends the core execution semantics. This preview implementation permits the `likely` attribute to be defined as a function that returns true or false. This way, the likelihood of transitions can change over time and is not limited to static load time specification.

The described approach currently only considers a *preview horizon* of one state; however the rFSM preview plugin could be easily extended to two or more. The major challenge for applying such multi-state horizons in practice will be to detect potentially conflicting `prepare` actions. Such reasoning will require more formal representation of actions (and thus robot tasks) and is outside the scope of this paper.

10 Step by step example: Coordination for a Dual-Robot Haptic Coupling

The following describes approach and methodology to construct the coordination of an dual robot haptic coupling.[14] Two KUKA youbots are to be coupled in a bidirectional manner in cartesian space using an impedance controller. That way, either youbot can be used to move the other and forces applied on one side can be felt on the other (see figure 17).

14. This demo was shown at the Automatica 2012 tradefair in Munich.

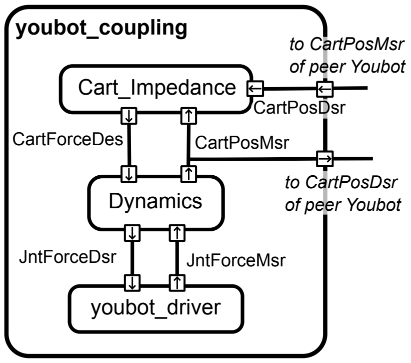

Fig. 18. Computational component architecture of the dual youbot haptic coupling.

Furthermore, the coupling shall satisfy the following requirements. Initially, both arms are decoupled and compensate for gravity, thus they can be moved freely around by operators. The force coupling between the two robots is only established once two constraints are satisfied. Firstly, the communication quality between the two robots must be sufficiently good (here defined in terms of round-trip latency). Secondly, the end-effector forces that would result from the coupling must not exceed a certain threshold. In other words, if the end-effectors of the robots are too far apart (relative to their bases), then the resulting force that would pull them together would be too high too, thus preventing the coupling.

Moreover, it shall be possible to manually switch between a five and eight degrees of freedom mode in which either only the arm or arm and omnidirectional base are used. This switching shall only be possible when the coupling is established.

The first step to model coordination is generally to examine the architecture of computational components. In most cases this architecture is more or less fixed as a consequence of reusing existing components. Figure 18 depicts the component architecture to control each robot. The coupling is achieved by connecting each impedance controllers desired position to the peer robots measured position. The resulting cartesian space force is locally communicated to the dynamics component that computes the inverse dynamics; the resulting desired joint-space forces are then sent to the driver for execution.

How many statecharts shall be introduced to coordinate this system? A best practice facilitating this decision is the following: any subsystem connected via unreliable or temporally non-deterministic communication should be coordinated by a separate, loosely connected coordinator. This rules out the approach of making one robot the master that supervises the slave robot.

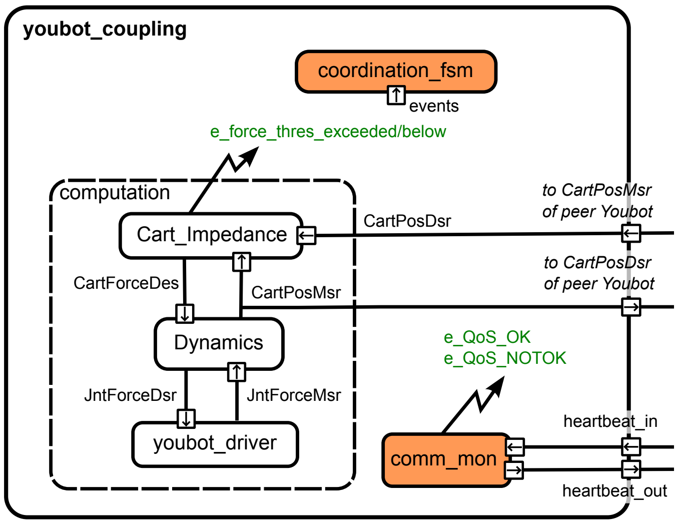

Fig. 19. Component architecture with extensions to support coordination.

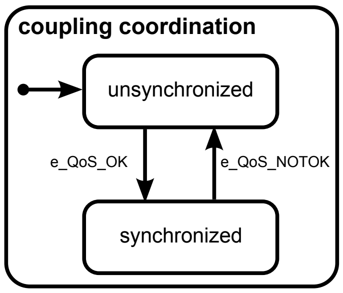

Fig. 20. Constructing the coordinator, step 1: modeling the communication quality constraint.

A further motivation for introducing two coordinatiors is reuse. Since the same computational architecture is used on each robot, it should be feasible to achieve the same symmetry and hence reuse for coordination.

Next it must be considered how an individual coordination statechart can obtain the necessary state information from the system. For reactive models like statecharts, this information is usually best represented by events. For this application we are interested in the communication quality and the impedance controller force output `CartForceDes`.

Yet, which component shall raise these events? Generally, there are two fundamental strategies for this: *embedding* the logic of event raising within a computational component or by introducing a *separate monitor* component for that purpose. By avoiding additional communication, the embedding strategy allows for lower latencies than with the external monitor. On the other hand, the external monitor component avoids polluting the computational component with application specific details.

For the coupling application one could consider extending the impedance controller component to raise communication quality events, since it is receiving the cartesian position from the peer (for instance by determining the communication latency using the creation timestamp of the received measured position).

This is obviously a bad choice, as it would clutter the controller component with application specific information. Thus, we introduce a separate, external communication monitor that determines the communication latency based on timestamped heartbeat messages exchanged with the remote side. Based on a configurable quality level, the events `e_QoS_OK` and `e_QoS_NOTOK` are emitted.

In contrast, for raising the force threshold events, the embedding approach was chosen and the impedance controller extended to raise two events `e_force_thres_exceeded` `e_force_thres_below` upon exceeding respectively falling below a configurable threshold. Unlike with the communication quality, the desired output is an essential quantity of a controller and hence this event is likely to be useful in other circumstances. Secondly, to permit simple switching between gravity compensation and coupling mode, the impedance controller was extended with a Boolean mode *external reference mode* (`ext_ref_mode`). When true (the nominal case), the external desired position (`CartPosDsr`) is taken into account to compute the output force. If false, the external input is ignore and instead zero forces are output. A functionally equivalent and less intrusive solution to achieve emitting a zero force would have been to (on-the-fly) remove the external `CartPosDsr` connection and instead connect the measured positions `CartPosMsr`. However, since rewiring connections is not a real-time safe operation in Orocos, we opted for the former.

Figure 19 shows the extended architecture including a coordination component containing a (yet to be defined) rFSM statechart, the external monitor component raising the QoS events and the extended impedance controller component.

Defining coordination statecharts is best carried out by top-down refinement. Since statechart priorities are decreasing with depth, this corresponds to starting with the highest priority states and transitions. For this application this is the requirement that the communication is established and sufficiently good. Without communication coupling the robots is impossible. Thus, we introduce two states `synchronized` and `unsynchronized` that model this requirement (Figure 20).

Next we refine the systems behavior by extending the `synchronized` state (entered when the communication is good enough) by adding two substates `gravity_comp` and `copying` (Figure 21). `gravity_comp` represents the state in which the impedance controller output force `CartForceDes` is too high and hence gravity compensation is enabled. In `copying` the actual coupling is established by enabling external reference mode.

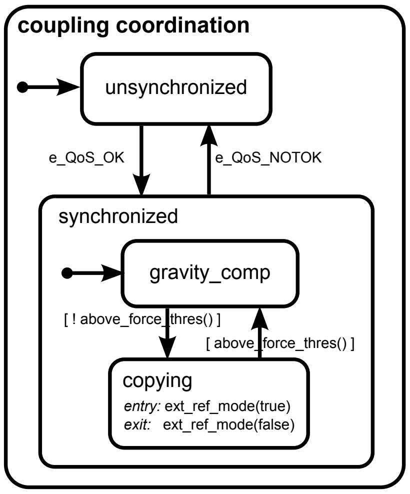

Fig. 21. Constructing the coordinator, step 2: modeling the force threshold constraint.

Conversely, external reference mode is disabled upon exiting the `copying` state, since only that way it is guaranteed that this mode is disabled no matter how `copying` is exited. For instance, if external reference mode were only disabled in the entry function of `gravity_comp`, the transition to `unsynchronized` triggered by an `e_QoS_NOTOK` when in `copying` would result in entering `unsynchronized` with external reference mode still enabled.

Note that in contrast to the toplevel transitions between `unsynchronized` and `synchronized`, the transitions between `gravity_comp` and `copying` make use of a guard condition `above_force_thres()` instead of being triggered by events. The reason for this is that upon entering the `synchronized` state the impedance controller forces will already be either too high or not and the corresponding events already raised. Thus, if these transitions were triggered only by events, the `gravity_comp` state might erronously remain active, unless by chance this condition just changes after entering `gravity_comp`.

```
function above_force_thres()
   local flow_status, value = force_thres_ex:read()
   if flow_status == 'NoData' then return false end
   else return value end
end
```

Listing 8. Guard condition `above_force_thres`.

To realize this guard, the edge triggered events emitted by the impedance controller can easily be transformed into the level triggered events required by the guard condition. Listing 8 shows how this can be achieved using the Orocos RTT framework. The latest received event is cached in the coordination component using a regular data-port (`force_thres_ex`), that then can be read and evaluated by the guard condition.

What remains now is only the functionality to switch between five and eight DOF mode. Once again this is achieved by top-down state refinement, in this case by extending `copying` with the two modes, as shown in figure 22. The operator can switch between both modes by sending the `e_8DOF` and `e_5DOF` events respectively. As an optimization, a history connector could be used instead of a plain initial connector. That way the previously selected mode would be resumed after an interruption of the coupling.

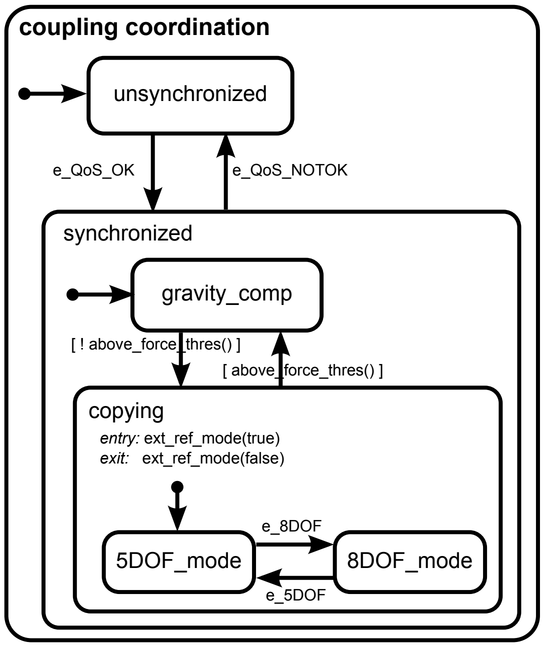

Fig. 22. Constructing the coordinator, step 3: adding modes

Using states to model modes works nicely when modes are mutually exclusive. However consider the requirement to additionally support two switchable modes *low-* and *high force threshold* in which the force threshold is reconfigured accordingly. Extending copying to support these modes would result in a combinatorial explosion of states. A good way to solve this is to use an SLDS state (discussed in Section 9.6) to compose the orthogonal modes as separate, concurrent substates.

11 Discussion

This paper describes a statechart model consisting of a *minimal* number of semantic primitives necessary for constructing *practical* robotic coordination. This minimality approach has the advantage of reducing implementation complexity, avoiding introducing unnecessary assumptions (as for instance the aforementioned communication properties) and being simple to understand and use.

Nonetheless, a minimality approach also has some disadvantages. For one it places a higher burden on developers to construct the required *composites*.

We intend to address this by introducing a *standard library* of coordination extensions. First mechanisms included are event memory, the serialised locally-distributed state and the preview coordination extension.

Additional effort is also required for deploying a distributed statechart, as connections and communication must be configured between the different FSM. This involves considering communication reliability, defining buffer sizes, triggering and buffering policies. Little if any tooling exists to support this process; tools for deploying *computations* might be of some help, yet the different characteristics of *coordination* will most likely require dedicated tools. As an example, having multiple simultaneously active writers communicating data to a single reader is often an erroneous situation for Computations (e.g. only one controller may command a robot at a time). Conversely, it is common to connect multiple, active event sources to one Coordination statechart.

For using the rFSM model in the context of complex distributed robotic systems, we propose to make exclusive use of *pure coordination*. Pure coordination can be achieved by rigorously separating the concerns of Computations from Coordination; more concretely by limiting Coordination actions to exclusively raising events.

The benefits of adhering to this best-practice are increased robustness and reusability of Coordination. The drawback, however is that as of today few components provide sufficiently expressive constraint configuration interfaces; one reason for this being presumably the lack of a generic mechanism to specify condition-event pairs on internal state of computations.

One motivation for choosing graphical models (as statecharts) to model complex systems is their apparent ease to be understood by humans. Yet surprisingly few tools exist to visualise graphical models (online or offline) by performing automatic layout and rendering of the textual representation. The tool most widely used for this purpose is probably graphviz [50], that is also used by the rFSM reference implementation for visualising statechart models. Unfortunately, the generated representation, especially when involving hierarchical states, is often suboptimal[15]; significant fine tuning of layout parameters only resulted in a marginal improvement. Nevertheless, this simple tool has proven invaluable to validate that a model specified in textual form does indeed correspond to the graphical model the developer has in mind. We intend to address this topic in future work.

12 Conclusion and Future Work

We have presented a lightweight Coordination statechart model that is derived by analysing and extracting a minimal subset of model elements from existing formalism.

15. Optimality defined by how a human would draw a statechart, as is the case with all figures (apart from Figure 10) in this paper.

The proposed rFSM model is graphically a subset of UML with simplified execution semantics derived from the well known STATEMATE statecharts. By selective inclusion of model elements, many of the corner cases and additional rules required by existing formalism can be avoided, thus simplifying both implementation and coordination models themselves.

Instead of providing a rich set of built-in features for all possible use-cases, the rFSM model advocates dealing with complexity by composition. Composition means both local hierarchical composition of Statecharts or distributed composition as described in Section 9.5 as well as composition of core execution semantics with run-time extensions such as event memory.

To back up these claims we implemented an extensible, framework independent, real-time safe reference implementation in the Lua scripting language. The featurewise almost complete rFSM core engine currently amounts to less than 830 lines of code[16]. This reference implementation has successfully been integrated into the OROCOS RTT software framework and applied to a wide range of use-cases.

Moreover, we describe several best-practice patterns of robotic task and system coordination that were discovered during our work. These patterns serve to highlight frequent coordination design issues together with best practice solutions, with the goal of fostering adoption of the rFSM model.

Up to now, the described rFSM model has been sufficient to describe all robotic task and system coordination we encountered, hence there currently seems no need for extensions to the core model. On the other hand we observe a severe lack of tools to support creating, deploying and visualising rFSM statecharts and Coordination models in general. We intend to address this in future work.

Acknowledgment

We would like to thank the anonymous reviewers for their concise and valuable comments to improve the quality of this paper.

References

[1] M. Radestock and S. Eisenbach, "Coordination in evolving systems," in *Trends in Distributed Systems. CORBA and Beyond*. Springer-Verlag, 1996, pp. 162–176. 1

[2] P. Soetens, "A software framework for real-time and distributed robot and machine control," Ph.D. dissertation, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2006, http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf. 1, 8.1

[3] Willow Garage, "Robot Operating System (ROS)," http://www.ros.org, 2008, last visited 2012. 1, 2.10

[4] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based RT-system development: OpenRTM-Aist," in *Conf. Simulation, Modeling, and Programming of Autonomous Robots*, Venice, Italia, 2008, pp. 87–98. 1

[5] Korean Institute for Advanced Intelligent Systems, "OPRoS," http://opros.or.kr/, last visited November 2010. 1

16. Calculated using the cloc(1) tool.

[6] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009. 1

[7] OMG, "OMG Systems Modeling Language(SysML)," http://www.sysml.org/. 1.1

[8] M. von der Beeck, "A comparison of statecharts variants," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, ser. LNCS, H. Langmaack, W.-P. de Roever, and J. Vytopil, Eds. Springer Berlin / Heidelberg, 1994, vol. 863, pp. 128–148. 1.3, 2.2

[9] R. Eshuis, "Reconciling statechart semantics," *Sci. Comput. Program.*, vol. 74, no. 3, pp. 65–99, 2009. 1.3

[10] M. Breen, "Statecharts: Some critical observations," 2004. 1.3, 2.2

[11] A. J. H. Simons, "On the compositional properties of UML statechart diagrams," in *Rigorous Object-Oriented Methods*, ser. Workshops in Computing. BCS, 2000. 1.3, 4.1, 4.2.1, 4.3.4, 6.2

[12] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Comp. Prog.*, vol. 8, pp. 231–274, 1987. 2.2, 4.2

[13] D. Harel and A. Naamad, "The STATEMATE semantics of statecharts," *ACM Trans. on Software Engineering Methodolody*, vol. 5, no. 4, pp. 293–333, 1996. 2.2, 3, 5, 5.1, 5.3, 5.5, 6.6, 9.1

[14] D. Harel, H. Lanchover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A working environment for the development of complex reactive system," *IEEE Transactions on Software Engineering*, vol. 16, pp. 403–414, 1990. 2.2

[15] OMG, "UML specification," http://www.omg.org/spec/UML, 2009. 2.2

[16] W3C, "State Chart XML (SCXML): State machine notation for control abstraction," 2010, http://www.w3.org/TR/scxml/. 2.2

[17] MathWorks, "Design and simulate state charts by The Mathworks," www.mathworks.de/products/stateflow/. 2.2, 2.4, 4.1

[18] OMG, "Unified Modeling Language (UML) superstructure specification, version 2.4.1," http://www.uml.org/, 2011. 2.3, 4.2.1, 4.3.3, 5.2, 5.5, 6.1, 6.7, 7

[19] T. Merz, P. Rudol, and M. Wzorek, "Control system framework for autonomous robots based on extended state machines," in *International Conference on Autonomic and Autonomous Systems, 2006. ICAS '06. 2006*, july 2006, p. 14. 2.5

[20] D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock, "Modelling behaviour requirements for automatic interpretation, simulation and deployment," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. LNCS, N. Ando, S. Balakirsky, T. Hemker, M. Reggiani, and O. von Stryk, Eds. Springer Berlin / Heidelberg, 2010, vol. 6472, pp. 204–216. 2.5

[21] T. C. . International Electrotechnical Commission, *IEC 61131-3 Ed. 2: Programmable controllers — Part 3: Programming Languages*. IEC, 2003. 2.6

[22] R. David and H. Alla, *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992. 2.6

[23] N. Bauer and S. Engell, "A comparison of sequential function charts and statecharts and an approach towards integration," in *Proceedings of the Workshop INT'02, Grenoble 2002*, 2002, pp. 58–69. 2.6

[24] R. Dromey, "From requirements to design: formalizing the key steps," in *First International Conference on Software Engineering and Formal Methods.*, sept. 2003, pp. 2 –11. 2.7

[25] R. Simmons and D. Apfelbaum, "A task description language for robot control," in *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, vol. 3, oct 1998, pp. 1931 –1937 vol.3. 2.8

[26] J.-C. Baillie, "Urbi: A universal language for robotic control," *International journal of Humanoid Robotics*, 2004. 2.9

[27] J. Bohren and S. Cousins, "The smach high-level executive [ros news]," *Robotics Automation Magazine, IEEE*, vol. 17, no. 4, pp. 18 –20, dec. 2010. 2.10

[28] C. Dragert, J. Kienzle, and C. Verbrugge, "Toward high-level reuse of statechart-based ai in computer games," in *Proceedings of the 1st International Workshop on Games and Software Engineering*, ser. GAS '11. New York, NY, USA: ACM, 2011, pp. 25–28. 2.11

[29] Y. Gamha, N. Bennacer, L. Ben Romdhane, G. Vidal-Naquet, and B. Ayeb, "A statechart-based model for the semantic composition of web services," in *2007 IEEE Congress on Services*, july 2007, pp. 49–56. 2.11

[30] S. Fleury, M. Herrb, and R. Chatila, "GenoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture," in *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Grenoble, France: IROS97, 1997, pp. 842–848. 4.1

[31] M. E. Conway, "Design of a separable transition-diagram compiler," *Commun. ACM*, vol. 6, pp. 396–408, July 1963. 4.1

[32] M. Klotzbuecher, "rFSM statecharts," http://www.orocos.org/rFSM, 2011, last visited January 2012. 8

[33] M. Fowler, "Language workbenches: The killer-app for domain specific languages?" Juni 2005. [Online]. Available: http://martinfowler.com/articles/languageWorkbench.html 8

[34] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, "Lua—an extensible extension language," *Softw. Pract. Exper.*, vol. 26, no. 6, pp. 635–652, 1996. 8

[35] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "The evolution of lua," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 2–1–2–26. 8

[36] S. Borland, "Transforming statechart models to DEVS," Master's thesis, Montreal, Canada, 2003. 8

[37] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann, "Implementing statecharts in PROMELA/SPIN," in *Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, ser. WIFT '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 90–101. 8

[38] Q. Zhao and B. Krogh, "Formal verification of statecharts using finite-state model checkers," *Control Systems Technology, IEEE Transactions on*, vol. 14, no. 5, pp. 943 –950, sept. 2006. 8

[39] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo, "A constant-time dynamic storage allocator for real-time systems," *Real-Time Syst.*, vol. 40, no. 2, pp. 149–179, 2008. 8.2

[40] M. Klotzbuecher, P. Soetens, and H. Bruyninckx, "OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages," in *International Workshop on Dynamic languages for RObotic and Sensors*, 2010, pp. 284–289. 8.2

[41] M. Klotzbuecher and H. Bruyninckx, "Hard real-time control and coordination of robot tasks using Lua," in *Proceedings of the Thirteenth Real-Time Linux Workshop*, October 2011. 8.2

[42] H. Kopetz, "Should responsive systems be event-triggered or time-triggered ?" *Institute of Electronics, Information, and Communications Engineers Transactions on Information and Systems*, vol. E76-D, no. 11, pp. 1325–1332, 1993. 9.1

[43] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, jan 2003. 9.1

[44] M. T. Mason, "Compliance and force control for computer controlled manipulators," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-11, no. 6, pp. 418–432, 1981. 9.2

[45] H. Bruyninckx and J. De Schutter, "Specification of force-controlled actions in the "Task Frame Formalism": A survey," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 5, pp. 581–589, 1996. 9.2

[46] M. Klotzbuecher and H. Bruyninckx, "A lightweight, composable metamodelling language for specification and validation of internal domain specific languages," in *Proceedings of the 8th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, September 2012. 9.2

[47] M. Klotzbuecher, G. Biggs, and H. Bruyninckx, "Pure coordination using the coordinator–configurator pattern," in *Proceedings of the 3rd International Workshop on Domain-Specific Languages and models for ROBotic systems*, November 2012. 9.3

[48] J. De Schutter, T. De Laet, J. Rutgeerts, W. Decré, R. Smits, E. Aertbeliën, K. Claes, and H. Bruyninckx, "Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty," *The International Journal of Robotics Research*, vol. 26, no. 5, pp. 433–455, 2007. 9.6

[49] T. B. Sheridan, *Telerobotics, Automation, and Human Supervisory Control*. Cambridge, MA, USA: MIT Press, 1992. 9.7

[50] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *SOFTWARE - PRACTICE AND EXPERIENCE*, vol. 30, no. 11, pp. 1203–1233, 2000. 11

Markus Klotzbücher In 2004, Markus Klotzbücher received a Dipl.-Inf (FH) degree from the University of Applied Sciences in Konstanz, Germany. After that he worked for several years as an independent consultant in the area of embedded and realtime systems, and also held seminars on embedded Linux system design and Linux device driver development. In 2008 he returned to University to pursue a PhD in the field of Robotics at the Katholieke Universiteit in Leuven, Belgium. His research focuses on domain specific languages to support constructing complex and reusable robot systems that can operate under hard real-time constraints.

Herman Bruyninckx Dr. Bruyninckx obtained the Masters degrees in Mathematics (Licentiate, 1984), Computer Science (Burgerlijk Ingenieur, 1987) and Mechatronics (1988), all from the Katholieke Universiteit Leuven, Belgium. In 1995 he obtained his Doctoral Degree in Engineering from the same university, with a thesis entitled "Kinematic Models for Robot Compliant Motion with Identification of Uncertainties." He is full-time Professor at the K.U.Leuven, and held visiting research positions at the Grasp Lab of the University of Pennsylvania, Philadelphia (1996), the Robotics Lab of Stanford University (1999), and the Kungl Tekniska Hogskolan, Stockholm (2002). Since 2007, he is Coordinator of the European Robotics Research Network EURON (http://www.euron.org). His current research interests are on-line Bayesian estimation of model uncertainties in sensor-based robot tasks, kinematics and dynamics of robots and humans, and the software engineering of large-scale robot control systems. In 2001, he started the Free Software ("open source") project Orocos (http://www.orocos.org), to support his research interests, and to facilitate their industrial exploitation.