Reactive Semantics for Distributed UML Activities

Frank Alexander Kraemer, Peter Herrmann

Reactive Semantics for Distributed UML Activities

Frank Alexander Kraemer and Peter Herrmann

Norwegian University of Science and Technology (NTNU),
Department of Telematics, N-7491 Trondheim, Norway
{kraemer, herrmann}@item.ntnu.no

Abstract. We define a *reactive* semantics for a subset of UML activities that is suitable as precise design language for reactive software systems. These semantics identify run-to-completion steps for execution on the level of UML activities as so-called activity steps. We show that activities adhering to these semantics and a set of rules lead to event-driven and bounded specifications that can be implemented automatically by model transformations and executed efficiently using runtime support systems.

Key words: UML Activities, UML Semantics, Reactive Systems

1 Introduction

UML 2.0 activities essentially denote in which order certain actions have to be executed to accomplish some task, and are therefore suitable for a wide range of applications. With their revision for the second major version of the UML standard [1], they were considerably enhanced with respect to supporting concurrent flows and hierarchically structured specifications. In the following, we focus on the application of UML activities on the domain of reactive systems. This class of systems, characterized by Pnueli as ones that *"maintain some interaction with their environment"* [2], is interesting, since with an increasing degree of connectivity of devices and the ubiquity of sensors that provide data, more and more applications fall into this category of systems. In general, these reactive applications and corresponding systems are characterized as follows:

- There is a high degree of concurrency in the applications, since typically several connections are simultaneously active and events from different sources can occur at any time.
- These applications are *event-driven*, that means they execute their behavior as reactions on events such as incoming signals from other devices, user interface interactions or updated sensor inputs.

These characteristics make the development of reactive systems quite demanding, especially with respect to concurrency. Achieving concurrency by processes executing in parallel is complicated, since synchronizations between them are difficult to understand and error-prone. In addition, the number of processes

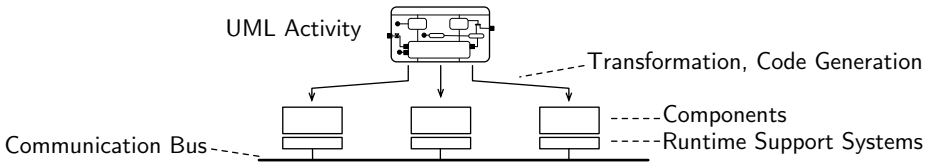

Fig. 1. Execution of components by runtime support systems

that can be executed efficiently in parallel is limited, in particular on mobile or embedded devices.

One way to deal with the complexity of concurrent executions is the introduction of runtime support systems [3], illustrated in the lower part of Fig. 1. These systems contain schedulers that control the execution of a component's behavior by dispatching events. Such events are the arrival of signals from other components, the expiration of timers, or internal signals that arise from local sensor data or interrupts. To be executable by a runtime support system, all application behavior must be expressed as transitions triggered by observable events as defined above. Moreover, communication must be implemented so that the sender is not blocked, for instance via an asynchronous message bus. These properties enable an execution with *run-to-completion* characteristics. The execution of a run-to-completion step is not preempted, and concludes with a stable state in which the runtime support system waits to dispatch the next observable event. This makes it relatively easy to execute also complex and highly concurrent behavior with very limited resources and a low number of system processes. Therefore, we are able to offer code generators different execution platforms, ranging from resource-constrained embedded systems on Sun SPOTs [4] to systems serving large numbers of users like Telenor's Connected Objects platform [5].

Our interest in activities is based on a number of characteristics they have that are relevant for the development of distributed, reactive systems: In comparison to state machines (that we have used before) they provide a high degree of concurrency by default, since activity flows execute independently from each other. We have also shown that activities can be grouped and abstracted in building blocks [6], which leads to system designs that are built to high proportions from reused solutions. Furthermore, since activities have the concept of partitions, they also describe collaborative behavior, and may encapsulate the behavior of several components that is necessary to fulfill a certain task, illustrated in Fig. 1. We therefore built a tool that takes UML activities and implements them automatically, using first a model transformation to UML components and state machines [7] and then a code generation step.

In order to use activities for reactive applications that are to be executed on an event-driven runtime support system as described above, their semantics need to address several issues:

– Activities should identify the run-to-completion steps executed by the runtime support system, as well as the observable event triggering a step.

– It must be clear on which location a run-to-completion step is performed, i.e., which component is responsible for executing it.
– Activities should make communication explicit, i.e., all necessary communication between components must be represented by activity flows.

One could argue, of course, that the abstraction level of activities does not need to deal with the same concepts as lower levels. For instance, non-local behavior of activities could be allowed, and communication patterns on the level of component execution could be synthesized automatically (as done for instance by Yamaguchi et al. [8]). However, we want the activities to express communication explicitly, for example to facilitate a security analysis as done in [9]. Furthermore, we want developers working on the level of activities to be still aware of costly operations such as sending to encourage efficient solutions. For the same reason we think it is beneficial to be aware of run-to-completion steps, since they form the temporal behavior of specifications.

In the following, we describe what we call *reactive* semantics for activities. Our intention is not to cover all details of the UML standard, but a subset of elements necessary to describe a wide range of reactive applications. We will present UML activities with an example and define some syntactic constraints in Sect. 3. Our semantics is based on run-to-completion steps, which on the activity level we call *activity steps*. These are defined in Sect. 4. The execution of activities based on activity steps is then described in Sect. 5. Section 6 discusses properties of activities and some additional constraints.

2 Related Work

There exists a variety of approaches that use different techniques to partially define and discuss semantics of UML 2.0 activities. Conrad Bock, one of the authors of the UML standard [1], covers the semantics of activities in a series of articles which informally clarify numerous semantic details. In particular, he describes the *traverse-to-completion* principle [10], according to which tokens pass only when all elements along a path accept the passing. This principle is implied by our semantics due to the definition of run-to-completion steps. Eshuis' work on model checking activity diagrams [11] defines their semantics by two mappings onto the model checker NuSMV, but in comparison to our work targets workflow systems, in which an activity is executed by a central workflow system that coordinates the execution of actions [12]. Störrle uses Coloured Petri Nets in [13] to define semantics of some UML activity elements. In [14], Störrle and Hausmann conclude that Petri Nets are probably not suitable to cover more advanced concepts, and point out the difficulty of combining all the various target domains. Barros and Gomez explain the semantics of actions with several input and output pins by "unfolding" these elements into activities with simpler control nodes [15]. Crane and Dingel [16] cover the detailed semantics of some specific UML action types, and describe in [17] a virtual machine for their interpretation. Engels et al. [18] define semantics of activities using Dynamic Meta Modeling (DMM), which is based on graph transformations. Sarstedt and

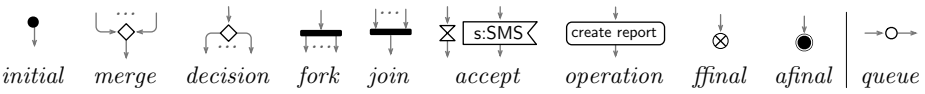

Fig. 2. Node kinds

Guttmann [19] use Abstract State Machines (ASMs) for the definitions of activity semantics. This work treats token passing on a very detailed level, removing some restrictions present in other approaches. To execute systems described by activities, they propose the interpretation of models [20].

In contrast to these approaches, we explicitly apply UML activities to the domain of reactive applications as characterized above, with an emphasis on execution mechanisms for runtime support systems. For this case, we found the semantics defined by the approaches mentioned above as not ideal, since they are either too general or target for example business processes, which pose different requirements on communication and synchronization. Therefore, we cannot use them as basis for the construction of executable components.

3 UML Activities

The UML standard uses simple text to explain activities, and characterizes their semantics as "Petri-like" [1, p. 324]. Some simple activity graphs can indeed easily mapped to Petri Nets. The mapping of more complex elements of UML activities, however, turns out to be difficult, especially for termination behavior, as pointed out by others [14, 21, 12]. In the following, we will therefore describe the reactive semantics of activities based on the intuitive token flows as found in Petri Nets, but use general state-transition systems in which also groups of tokens of several places can be removed simultaneously.

An activity is a directed graph A with a set of activity nodes V_A and a set of activity edges E_A. Function $kind_A \in [V_A \to K]$ assigns to each node a kind, with $K = \{$*initial*, *merge*, *decision*, *fork*, *join*, *accept*, *operation*, *afinal*, *ffinal*, *queue*$\}$, as illustrated in Fig. 2. Nodes of kind *accept* model accept event actions, which in our semantics represent either internal signal receptions or timer expirations. Internal signals are used to represent events from lower layers of a component, such as interrupts or events from user interfaces. Nodes of kind *operation* represent method calls. There are two kinds of final nodes, activity final nodes (*afinal*) that terminate an entire activity (hence removing tokens also from other places and preempting other behaviors), and flow final nodes (*ffinal*) that just consume tokens to conclude a flow.

An activity has a non-empty set of partitions P_A. In general, UML uses partitions to characterize commonalities among nodes. We use them strictly to denote different locations of executions. Thus, a flow crossing partition borders implies communication. For this communication, we assume an asynchronous message bus. This type of communication allows for an increased degree of con-

currency, but may introduce interleaving behavior that has to be taken care of by corresponding synchronization. Since this behavior is tightly interleaved with application logic, the delay of messages must be represented in the activity diagrams as well. For this reason, we introduce explicit queue places where activity flows cross partition borders. In Fig. 4, these are nodes q_1 to q_7.

Function $in_A \in [V_A \to 2^{E_A}]$ yields for each node its incoming edges, and function $out_A \in [V_A \to 2^{E_A}]$ its outgoing edges. Vice versa, we refer to $source_A(e)$ to get the source node of an edge, and $target_A(e)$ for its target. We assume that only merge and join nodes have more than one incoming edge, and only decision and fork nodes have more than one outgoing edge. All structural constraints are summarized by the rules in Fig. 3. Function $part_A \in [V_A \to P_A \cup (P_A \times P_A)]$ assigns partitions to nodes, so that each queue node is mapped to a pair of partitions modeling the transmission of signals between components (**PQ**). All other nodes are assigned to exactly one partition (**PN**). Rule **E1** ensures that edges do not have the same node as source and target, and **P1** ensures that there are no edges between two queues. These cases do not model useful behavior. Rules **P2** to **P4** ensure that edges do not cross partition borders without a queue

$$\textbf{IN1}\ \frac{v \in V_A \quad kind_A(v) \in \{initial\}}{|in_A(v)| = 0} \qquad \textbf{IN2}\ \frac{v \in V_A \quad kind_A(v) \in \{merge, join\}}{|in_A(v)| \geq 2}$$

$$\textbf{IN3}\ \frac{\begin{array}{c} v \in V_A \quad kind_A(v) \in \{decision, fork, \\ accept, operation, ffinal, afinal, queue\} \end{array}}{|in_A(v)| = 1} \qquad \textbf{OUT1}\ \frac{\begin{array}{c} v \in V_A \quad kind_A(v) \in \{initial, \\ merge, join, accept, operation\} \end{array}}{|out_A(v)| = 1}$$

$$\textbf{OUT2}\ \frac{\begin{array}{c} v \in V_A \\ kind_A(v) \in \{ffinal, afinal\} \end{array}}{|out_A(v)| = 0} \qquad \textbf{OUT3}\ \frac{\begin{array}{c} v \in V_A \\ kind_A(v) \in \{decision, fork\} \end{array}}{|out_A(v)| \geq 2}$$

$$\textbf{PN}\ \frac{v \in V_A \quad kind_A(v) \neq queue}{part_A(v) \in P_A} \qquad \textbf{PQ}\ \frac{q \in V_A \quad kind_A(q) = queue \quad p_1, p_2 \in P_A}{part_A(q) = \langle p_1, p_2 \rangle \quad p_1 \neq p_2}$$

$$\textbf{E1}\ \frac{e \in E_A}{source_A(e) \neq target_A(e)} \qquad \textbf{P1}\ \frac{e \in E_A \quad kind_A(source_A(e)) = queue}{kind_A(target_A(e)) \neq queue}$$

$$\textbf{P2}\ \frac{e \in E_A \quad kind_A(source_A(e)) \neq queue \quad kind_A(target_A(e)) \neq queue}{part_A(source_A(e)) = part_A(target_A(e))}$$

$$\textbf{P3}\ \frac{\begin{array}{c} e \in E_A \quad p, q \in P_A \\ kind_A(source_A(e)) \neq queue \\ kind_A(target_A(e)) = queue \\ part_A(target_A((e)) = \langle p, q \rangle \end{array}}{part_A(source(e)) = p} \qquad \textbf{P4}\ \frac{\begin{array}{c} e \in E_A \quad p, q \in P_A \\ kind_A(source_A(e)) = queue \\ kind_A(target_A(e)) \neq queue \\ part_A(source_A((e)) = \langle p, q \rangle \end{array}}{part_A(target(e)) = q}$$

Fig. 3. Rules for incoming and outgoing edges and partitions

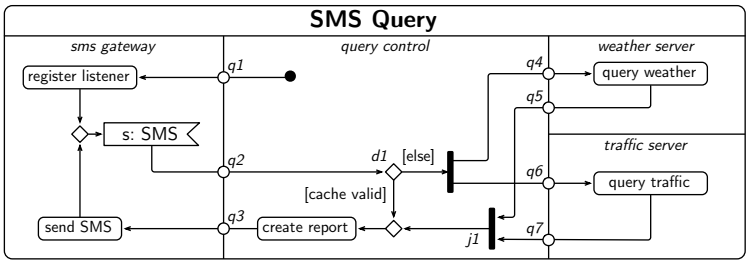

Fig. 4. Example activity for an SMS-based query system

node in between. (These rules are listed here for completeness. In practice, they are ensured by construction, since queue places are added automatically for edges crossing partitions.)

Figure 4 shows a simplified SMS-based query service, in which customers can request weather and traffic information by SMS. The system consists of four components with different tasks, represented by separate activity partitions. The behavior is started by the query control server, which activates the SMS gateway to listen for incoming SMS messages. These are received via accept node *s: SMS*, emitting one token for each SMS that is forwarded to the query control server. There, a decision is made in *d1* whether the cache holds valid traffic and weather data. If yes, a report is created immediately. If data is not available locally, queries are sent towards the weather and traffic servers concurrently. Join *j1* then collects their responses, which are used to create the report that is forwarded to the SMS gateway and sent out.

4 Run-to-Completion Steps in Activities: Activity Steps

As explained in the introduction, the execution of components by runtime support systems is based on run-to-completion steps that are triggered by discrete, observable events. On the level of activities, a run-to-completion step is called *activity step*. With respect to the formal representation of an activity diagram, an activity step is a subgraph a with $V_a \subseteq V_A$ and $E_a \subseteq E_A$. An activity step covers all nodes and edges that describe behavior executed within one run-to-completion step. An activity diagram describes with its graph a set of activity steps. The complete set of activity steps for a diagram can be obtained by considering all possible subgraphs a_i of A, for which the rules in Fig. 5 hold.

- Whenever a node is part of an activity step, then at least one of its incoming or outgoing edges is part of the step as well (rule **V**). Vice-versa, for each edge part of a step, also its source and target nodes are part of the step (**E**).
- All initial nodes within the same partition release their tokens simultaneously, so that also all other initial nodes within the same partition are part of an activity step (**I**).

– For merge nodes, only one incoming edge is part of an activity step (**M**). This rules out intricate behavior in which two or more tokens pass the same edge within the same step, as discussed later.
– For decision nodes, several activity steps are produced. Each contains the incoming edge, the decision node, and exactly one outgoing edge (**D**). This means that an activity step executes exactly one branch of a decision.
– When a fork is part of an activity step, so are its incoming edge and all outgoing edges, since they are executed in parallel (**F**).
– When a join is part of an activity step, then there is at least one incoming edge part of the step as well (**J**).
– Operations are executed within one run-to-completion step, and thus the incoming and outgoing edge must be part of the same activity step (**O**).
– The sending and the reception at partition borders are part of separate activity steps. Therefore, for each queue place, either the incoming or the outgoing edge is part of the same activity step (**Q**).
– When a node of type *initial*, *accept* or *queue* is part of a step and its outgoing edge as well, then it is triggering the step. Rule **T2** ensures that each activity step has at least one such trigger, and rule **T1** ensures that is at most one.
– Accept nodes are covered by the general rules **V** and **E**.

Figure 6 shows the complete set of activity steps that can be produced from the activity in Fig. 4. Steps a_1 to a_7 are executed within the query control, steps

$$\mathbf{V}\ \frac{v \in V_a}{in_A(v) \cap E_a \neq \emptyset\ \vee\ out_A(v) \cap E_a \neq \emptyset} \qquad \mathbf{E}\ \frac{e \in E_a}{source(e) \in V_a \quad target(e) \in V_a}$$

$$\mathbf{I}\ \frac{i \in V_a \quad part_A(i) = part_A(j) \quad kind_A(i) = kind_A(j) = initial}{j \in V_a}$$

$$\mathbf{M}\ \frac{m \in V_a \quad kind_A(m) = merge}{out_A(m) \subseteq E_a \quad |in_A(m) \cap E_a| = 1} \qquad \mathbf{D}\ \frac{d \in V_a \quad kind_A(d) = decision}{in_A(d) \subseteq E_a \quad |E_a \cap out_A(d)| = 1}$$

$$\mathbf{F}\ \frac{f \in V_a \quad kind_A(f) = fork}{in_A(f) \subseteq E_a \quad out_A(f) \subseteq E_a} \qquad \mathbf{J}\ \frac{j \in V_a \quad kind_A(j) = join}{in_A(j) \cap E_a \neq \emptyset}$$

$$\mathbf{O}\ \frac{o \in V_a \quad kind_A(o) = operation}{in_A(o) \subseteq E_a \quad out_A(o) \subseteq E_a} \qquad \mathbf{Q}\ \frac{q \in V_a \quad kind_A(q) = queue}{in_A(q) \cap E_a \neq \emptyset \Leftrightarrow out_A(q) \cap E_a = \emptyset}$$

$$\mathbf{T1}\ \frac{t, u \in V_a \quad out_A(t) \cap E_a \neq \emptyset \quad part_A(t) = part_A(u) \quad kind_A(t) \in \{initial, accept, queue\} \quad kind_A(u) \in \{accept, queue\}}{out_A(u) \cap E_a = \emptyset}$$

$$\mathbf{T2}\ \frac{\textsc{True}}{t \in V_a \quad out_A(t) \cap E_a \neq \emptyset \quad kind_A(t) \in \{initial, accept, queue\}}$$

Fig. 5. Rules for activity step subgraphs

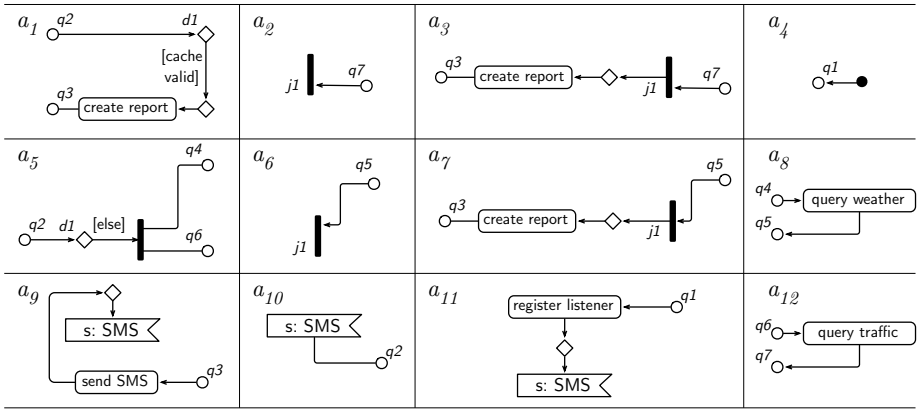

Fig. 6. Complete set of activity steps for the example

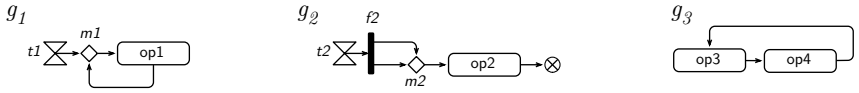

Fig. 7. Illegal subgraphs forbidden by rules

a_9 to a_{11} by the SMS gateway, and a_8 and a_{12} by the weather resp. the traffic server. Some steps are especially interesting: Steps a_1 and a_5 model the arrival of an SMS by the main server from the SMS gateway. Together they cover the alternative branches introduced by decision node d_1, which either creates the report instantly or starts the query. Steps a_2 and a_3 represent the arrival of the results from the traffic server (via q_7). Step a_2 covers the situation that the weather information did not yet arrive (and therefore stops at j_2), while step a_3 can fire through j_1 since the weather information already arrived.

Figure 7 shows examples of subgraphs that are not valid activity steps. Subgraph g_1 results in infinite executions of *op1* and is therefore not desired. Subgraph g_2 executes *op2* twice. While this is not necessarily wrong, we think that this is probably not obvious to and by no means intended by developers, and should be forbidden. Subgraph g_3 shows behavior that simply is never reachable. The existence of such invalid subgraphs in a diagram are detected by rule contradictions. The subgraph g_1 is illegal since rule **O** and **E** would force that both incoming edges of *m1* are part of the activity step, which is forbidden by rule **M**. Similarly, in g_2 rule **F** would contradict rule **M**. Subgraph g_3 has no trigger, contradicting rule **T2**. The existence of rule contradictions signify inconsistent diagrams that have to be changed.

5 Execution Steps and Execution Semantics

Formally, the behavior of a reactive system A expressed by a UML activity is as a state transition system in which the states are represented as a placement of tokens on the vertices and edges of the activity. For this reason, we define the type $tokens_A \triangleq 2^{[V_A \cup E_A \to \mathbb{N}]}$ of token mappings assigning each node and edge a natural number. Thus, a token mapping specifies a single system state.

A specific token mapping $init_A \in tokens_A$ refers to the token mapping of system A in its initial state. This initial mapping assigns one token to each of the vertices of kind *initial* while all the other nodes and edges are empty, as described by the following rules:

$$\textbf{INIT1}\ \frac{v \in V_A \quad kind_A(v) = initial}{init_A(v) = 1} \qquad \textbf{INIT2}\ \frac{v \in V_A \quad kind_A(v) \neq initial}{init_A(v) = 0}$$

$$\textbf{INIT3}\ \frac{e \in E_A}{init_A(e) = 0}$$

The state transitions are represented by execution steps which correspond to the activity steps and additional information describing the token settings before resp. after executing the step. Formally, we specify the execution step of an activity step represented by a subgraph $\langle V_a, E_a \rangle$ as the quadruple $ax_a = \langle V_a, E_a, pre_a, post_a \rangle$. $pre_a \in tokens_A$ describes the token setting of A before executing the activity step, and $post_a \in tokens_A$ denotes the token setting afterwards. For an ax_a to be valid, it must fulfill the following properties:

- In general, a token is added to accept and queue nodes when their incoming edge is part of an activity step (**IX**).
- In order to execute, the place representing a trigger must hold a token (**AX1**). If the trigger place is re-filled within the same step, its token count stays the same (**AX3**), otherwise it is reduced by one (**AX2**).
- For join nodes, two rules exist: Rule **JX1** models the firing of a join when tokens arrive at its incoming edges. All incoming edges not part of the activity step must already provide a token each. When the join fires, these tokens are deleted, and the step continues with the outgoing edge of the join. Rule **JX2** handles the arrival of tokens at a join when it is not yet complete. As consequence, it adds these arriving tokens but does not continue.
- Once a final node is part of an activity step, all tokens within the same partition are removed (see rule **AFX1** for tokens on edges and **AFX2** for tokens on nodes). The latter rule also removes all tokens within queues *towards* the partition containing the activity final node. This takes into account that the component implementing the terminated partition is switched off, and any remaining signals towards it are discarded. The additional precondition

  $$nofinal(n\colon 2^{N_A}, p\colon P_A) \triangleq \forall f\colon f \in n \wedge part_A(f) = p \Rightarrow kind_A(f) \neq afinal$$

  added to rules **IX**, **AX3**, **JX1** and **JX2** ensures that these rules only apply if no activity final node is part of the activity step, to give rules **AFX1** and **AFX2** priority.

$$\textbf{IX}\ \frac{v \in V_a \quad kind_A(v) \in \{accept, queue\} \quad in_A(v) \cap E_a \neq \emptyset \quad out_A(v) \cap E_a = \emptyset \quad nofinal(V_a, part(v))}{post_a(v) = pre_a(v) + 1}$$

$$\textbf{AX1}\ \frac{v \in V_a \quad kind_A(v) \in \{initial, accept, queue\} \quad out_A(v) \cap E_a \neq \emptyset}{pre_a(v) > 0} \qquad \textbf{AX2}\ \frac{v \in V_a \quad kind_A(v) \in \{initial, accept, queue\} \quad out_A(v) \cap E_a \neq \emptyset \quad in_A(v) \cap E_a = \emptyset}{post_a(v) = pre_a(v) - 1}$$

$$\textbf{AX3}\ \frac{v \in V_a \quad kind_A(v) = accept \quad in_A(v) \cap E_a \neq \emptyset \quad out_A(v) \cap E_a \neq \emptyset \quad nofinal(V_a, part(v))}{post_a(v) = pre_a(v)}$$

$$\textbf{JX1}\ \frac{v \in V_a \quad kind_A(v) = join \quad es = in_A(v) \cap E_a \quad \forall e \in in_A(v) \setminus es : pre_a(e) > 0 \quad nofinal(V_a, part(v))}{out_A(v) \subseteq E_a \quad \forall e \in in_A(v) \setminus es : post_a(e) = pre_a(e) - 1}$$

$$\textbf{JX2}\ \frac{v \in V_a \quad kind_A(v) = join \quad es = in_A(v) \cap E_a \quad \exists e \in in_A(v) \setminus es : pre_a(e) = 0 \quad nofinal(V_a, part(v))}{out_A(v) \cap E_a = \emptyset \quad \forall e \in es : post_a(e) = pre_a(e) + 1}$$

$$\textbf{AFX1}\ \frac{e \in E_A \quad f \in V_a \quad kind_A(f) = afinal \quad part(f) = part(target_A(e))}{post_a(e) = 0} \qquad \textbf{AFX2}\ \frac{f \in V_a \quad v \in V_A \quad kind_A(f) = afinal \quad part(v) = part(f) \quad \vee\ part(v) = \langle \_, part(f) \rangle}{post_a(v) = 0}$$

Fig. 8. Execution rules for activity steps

– All vertices and edges of A not mentioned by one of the rules explicitly do not change their token setting in the execution step ax_a.

We define the set AX_A as the set of execution steps ax_a each following the rules mentioned above. This set contains execution steps that are not reachable. We define the set of reachable execution steps $RAX_A \subseteq AX_A$ by means of the following two rules:

$$\textbf{R1}\ \frac{a \in AX_A \quad \forall v \in V_A : kind_A(v) = initial \Leftrightarrow pre_a(v) = 1 \quad \forall v \in V_A : kind_A(v) \neq initial \Leftrightarrow pre_a(v) = 0 \quad \forall e \in E_A : pre_a(e) = 0}{a \in RAX_A}$$

$$\textbf{R2}\ \frac{a, b \in AX_A \quad \forall v \in V_A : pre_a(v) = post_b(v) \quad \forall e \in E_A : pre_a(e) = post_b(e) \quad b \in RAX_A}{a \in RAX_A}$$

The rules define recursively that the reachable execution steps a are those containing a token setting pre_a reachable by a finite trace of execution steps from

the initial token setting. The behavior of system Sys_A is then defined by the state transition system expressed by the quadruple $Sys_A \triangleq \langle V_A, E_A, init_A, RAX_A \rangle$.

6 Properties of Activities with Reactive Semantics

In the following, we will discuss the properties of activities with reactive semantics. The properties in Sect. 6.2 and 6.3 require additional behavioral invariants, that is, they only hold for a subset of activities. We will assure these by additional rules that have to hold for any $a \in RAX_A$. In practice, these rules are verified by model checking, as discussed in [22].

6.1 Event-Driven Execution and Run-to-Completion

As described in the introduction, the events observable by a runtime support system are the expiration of timers, the arrival of internal signals, or the arrival of signals from other components. On the activity level, these events are modeled by accept event actions and activity flows that cross partition borders via queues. The initial startup of a component is also an event, modeled by initial nodes in activities. Due to rules **T1** and **T2**, we ensure that each execution step declares exactly one such trigger, meaning that activity steps started by an observable event. Furthermore, tokens may be only placed according to two properties:

(i) Tokens may rest on a vertex only if it is of kind *initial*, *accept*, or *queue*.
(ii) Tokens may rest on edges if they lead to a join node, but only if the join is not yet complete, i.e., there is one incoming edge that cannot offer a token.

Property (i) ensures that tokens only wait in places that imply waiting for an observable event. In initial nodes, this is the start of the system, in queues the arrival of the signal and for accept nodes the expiration of a timer or the arrival of an internal signal, resp. Property (ii) is more subtle. Tokens may wait on edges preceding join nodes. But since join nodes do not declare any observable events, tokens may only rest before a join if the join is not yet complete. The join fires through within the same step once the last missing token arrives. Hence, a token stored in an incomplete join implies waiting for another observable event. Formally, (i) and (ii) can be expressed as $P \triangleq E_1 \wedge E_2 \wedge E_3 \wedge E_4$, with

$$E_1 \triangleq \forall v \in V_A : kind(v) \in \{initial, accept, queue\} \vee init_A(v) = 0$$

$$E_2 \triangleq \forall e \in E_A : kind(target_A(e)) \neq join \wedge init_A(e) = 0 \\ \vee \ \exists f \in E_A : f \in in(target_A(e)) \wedge init_A(f) = 0$$

$$E_3 \triangleq \forall v \in V_A \forall a \in RAX_A : kind(v) \in \{initial, accept, queue\} \vee post_a(v) = 0$$

$$E_4 \triangleq \forall e \in E_A \forall a \in RAX_A : kind(target_A(e)) \neq join \wedge post_a(e) = 0 \\ \vee \ \exists f \in E_A : f \in in(target_A(e)) \wedge post_a(f) = 0$$

E_1 and E_2 describe that (i) and (ii) hold in the initial state of the system while E_3 and E_4 guarantee that all execution steps $a \in RAX_A$ preserve them as well. E_1 holds due to rules **INIT1** and **INIT2** and E_2 holds due to rule **INIT3**.

To prove E_3, we consider the rules in Fig. 8. The only rule describing that the token setting of a vertex v after executing an execution step a can be greater than before (i.e., $post_a(v) > pre_a(v)$) is **IX**. But according to the second premise of the rule, the token setting may only be increased for tokens of kind *accept* and *queue*. Thus, if a vertex of a type other than *initial*, *accept* or *queue* has no token placed on it before the execution of the execution step, it will carry none afterwards as well. In consequence, it will never carry a token at all.

The proof of E_4 is quite similar. The only rule in Fig. 8 modeling an increase of a token setting on an edge in an execution step is **JX2**. Yet it does not allow the placement of new tokens on edges leading to a vertex not being of kind *join* as expressed by the third premise. So, the first disjunct of E_4 holds and in all system states only edges into a join may have tokens at all. Further, the forth premise of the rule states that there is an edge leading into the join node to which no token is assigned. As this edge is not an element of the edges receiving new tokens (expressed by set es), it will not carry a token after firing the execution step. Thus, the second disjunct of E_4 holds as well and a join node will always have an incoming edge without a token placed on it.

6.2 Realizability and Distribution

To be executable as one run-to-completion step, an activity step must only have direct local effects, and only depend on data that is available locally. Rules **Q** and **P1** to **P4** split up activity steps at partition borders. Within one activity step, all nodes except queues are therefore part of the same partition. For flows crossing partition borders, we take the unavoidable communication delay into account by explicit queue nodes, so that they can be implemented using some communication middleware. For some nodes, the general semantics of UML describe also non-local dependencies, that motivate some further constraints:

- As an extension to standard UML, we assume that variables, like activity nodes, are assigned to partitions. Guards and actions are only allowed to access variables within their own partition.
- Initial nodes are started simultaneously, but only those within the same partition. This is already ensured by rule **I**.
- In standard UML, accept event actions without incoming edge denote that they are activated with the surrounding activity. Instead, we require these actions to have an incoming edge (rule **IN3**), to model activation explicitly.
- Activity final nodes terminate in standard UML the entire activity. With the reactive semantics, they only remove tokens within the *same* partition (rules **AFX1** and **AFX2**). Since messages towards a terminated partition are discarded, queues towards a terminated partition are emptied as well.

To comply with the general semantics of activity nodes in which an activity final node terminates the behavior in *all* partitions, the other partitions must not be able to execute any further activity steps. This is ensured by the additional rules **TRM1** and **TRM2**. The former states that whenever an activity final

$$\textbf{TRM1}\ \frac{f \in V_a \quad v \in V_A \quad kind_A(f) = afinal \quad kind_A(v) = accept \quad part_A(v) \neq part_A(f)}{pre_a(v) = post_a(v) = 0}$$

$$\textbf{TRM2}\ \frac{f \in V_a \quad q \in V_A \quad kind_A(f) = afinal \quad kind_A(q) = queue \quad part_A(q) \neq \langle \_, f \rangle}{pre_a(q) = post_a(q) = 0}$$

$$\textbf{AB}\ \frac{v \in V_a \quad kind_A(v) = accept}{post_a(v) \leq 1} \qquad \textbf{JB}\ \frac{v \in V_a \quad kind_A(v) = join \quad e \in in_A(v)}{post_a(e) \leq 1}$$

$$\textbf{QB}\ \frac{v \in V_a \quad kind_A(v) = queue}{post_a(v) \leq maxqueue}$$

Fig. 9. Additional rules for activities

node is reached, there must not be any token in accept nodes of other partitions. The latter states that all queues not targeting the terminated partition must be empty as well. Since tokens offered to join nodes cannot trigger any behavior on their own, they do not have to be removed upon termination.

6.3 Boundedness

Since the number of places needed to describe an activity is limited, the state space implied by a specification is finite if (and only if) each place only contains a bounded number of tokens, i.e., $|tokens_A(x) < N|$, with N as a finite boundary. Rules **IX** and **JX2** are the only rules increasing token places.

- Accept nodes are either enabled or disabled, represented by one or zero tokens on their corresponding place. We found that adding more than one token in an accept node is in most cases unintended and an indicator of a design flaw. We therefore rule out such behavior by rule **AB**.
- Places before join nodes can hold many tokens which implies buffering of data or control flow. We found that this makes activities harder to understand, without adding any expressiveness for reactive systems; we rather recommend to use explicit building blocks to describe buffering and rule out behaviors in which tokens accumulate before joins by rule **JB**.
- Queues between partitions need to be bounded as well. That means, they must not exceed a certain value $maxqueue \in \mathbb{N}$, expressed by rule **QB**.

If the boundedness rules hold, the set of reachable execution steps RAX_A will be finite as it is lower or equal to the product of run-to-completion steps times possible token markings following the boundedness constraints.

7 Concluding Remarks

We described a reactive semantics for UML activities, in which each execution step is triggered by an observable event. This is motivated by existing mechanisms present in runtime support systems for efficient but nevertheless simple

scheduling and execution of highly concurrent behavior. As a consequence of the reactive semantics, the components produced by our model transformation [7] from activities have the same efficiency as if they would have been produced manually by an experienced designer, i.e., do not contain any overhead for token control, and it is not visible that they were generated from UML activities.

We have implemented comprehensive tool support with Arctis [22], a set of Eclipse plug-ins. It enables editing, analyzing and automatically implementing activity-based specifications. The syntactical rules from Fig. 3 are ensured by the editor, highlighting erroneous parts of the graph. Similarly, the tool can produce all activity steps implied by a diagram, using the rules from Fig. 5. Rule contradictions that signify illegal diagrams are detected and explained to the user. Finally, the additional rules for sound behavior in Fig. 9 are verified by model checking [22]. Our tool also supports data in activities, which we did not treat here. Our experience from implementing data shows that their semantics can be covered by extending the semantics for control flows in a straight-forward way. Operations with more than one incoming flow, for instance, can be modeled similar to join nodes. More advanced nodes can be modeled by dedicated building blocks. Concerning the expressiveness of the chosen reactive semantics we point to numerous case studies (summarized in [6]) that exemplify their application in various domains. With an abstraction mechanism described in [6] such solutions can also be encapsulated by dedicated building blocks from which large system designs can be produced in a scalable manner.

References

1. Object Management Group: Unified Modeling Language: Superstructure, version 2.2, formal/2009-02-02 (2009)
2. Pnueli, A.: Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds.) Current Trends in Concurrency. Overviews and Tutorials. LNCS, vol. 224, pp. 510–584, Springer (1986)
3. Bræk, R., Haugen, Ø.: Engineering Real Time Systems: An Object-Oriented Methodology Using SDL. Prentice Hall (1993)
4. Kraemer, F.A., Slåtten, V., Herrmann, P.: Model-Driven Construction of Embedded Applications based on Reusable Building Blocks – An Example. In: Bilgic, A., Gotzhein, R., Reed, R. (eds.) SDL 2009, LNCS, vol. 5719, pp. 1–18, Springer (2009)
5. Herstad, A., Nersveen, E., Samset, H., Storsveen, A., Svaet, S., Husa, K.E.: Connected Objects: Building a Service Platform for M2M. In: Beyond the Bit Pipe. Proccedings of the 13th ICIN Conference (2009)
6. Kraemer, F.A., Herrmann, P.: Automated Encapsulation of UML Activities for Incremental Development and Verification. In: Schürr, A., Selic, B. (eds.) Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Proceedings. LNCS, vol. 5795, pp. 571–585, Springer (2009)
7. Kraemer, F.A., Herrmann, P.: Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In: Ehring, K., Giese, H. (eds.) 6th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT), Proceedings. Electronic Communications of the EASST, vol. 7, EASST (2007)

8. Yamaguchi, H., El-Fakih, K., von Bochmann, G., Higashino, T.: Protocol Synthesis and Re-Synthesis with Optimal Allocation of Resources based on Extended Petri Nets. In: Distrib. Comput., 16(1) pp. 21–35 (2003).
9. Gunawan, L.A., Herrmann, P., Kraemer, F.A.: Towards the Integration of Security Aspects into System Development using Collaboration-Oriented Models. In: Security Technology. International Conference on Security Technology (SecTech 2009), Proceedings. Communications in Computer and Information Science, vol. 58, pp. 72–85. Springer (2009)
10. Bock, C.: UML 2 Activity and Action Models, Part 4: Object Nodes. In: Journal of Object Technology, 3(1), pp. 27–41 (2004)
11. Eshuis, R.: Symbolic Model Checking of UML Activity Diagrams. In: ACM Transactions on Software Engineering and Methodology, 15(1), pp. 1–38 (2006)
12. Eshuis, R., Wieringa, R.: Comparing Petri Net and Activity Diagram Variants for Workflow Modelling - A Quest for Reactive Petri Nets. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems, LNCS, vol. 2472, pp. 321–351. Springer (2003)
13. Störrle, H.: Semantics and Verification of Data Flow in UML 2.0 Activities. In: Electronic Notes in Theoretical Computer Science, vol. 127, pp 35–52 (2005)
14. Störrle, H., Hausmann, J.H.: Towards a Formal Semantics of UML 2.0 Activities. In: Liggesmeyer, P., Pohl, K. Goedicke, M. (eds.) Software Engineering, Fachtagung des GI-Fachbereichs Softwaretechnik, LNI, vol. 64, pp. 117–128. GI (2005)
15. Barros, J.P., Gomes, L.: Actions as Activities and Activities as Petri Nets. In: Workshop on Critical Systems Development with UML, Proceedings. (2003)
16. Crane, M.L., Dingel, J.: Towards a Formal Account of a Foundational Subset for Executable UML Models. In: Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Vlter, M. (eds.) Model Driven Engineering Languages and Systems, 11th Int. Conference, MoDELS 2008, Proceedings. LNCS, vol. 5301, pp. 675–689. Springer (2008)
17. Crane, M.L., Dingel, J.: Towards a UML Virtual Machine: Implementing an Interpreter for UML 2 Actions and Activities. In: CASCON '08: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research, pp. 96–110, ACM (2008)
18. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML Activities Using Dynamic Meta Modeling. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS, LNCS, vol. 4468 pp. 76–90. Springer (2007)
19. Sarstedt, S., Guttmann, W.: An ASM Semantics of Token Flow in UML 2 Activity Diagrams. In: Virbitskaite, I., Voronkov, A. (eds.) Ershov Memorial Conference, LNCS, vol. 4378, pages 349–362, Springer (2006)
20. Sarstedt, S., Gessenharter, S., Kohlmeyer, J., Raschke, A., Schneiderhan, M.: ActiveChartsIDE: An Integrated Software Development Environment Comprising a Component for Simulating UML 2 Activity Charts. In: European Simulation and Modelling Conference (ESM'05), Proceedings, pp. 66–73 (2005)
21. van der Aalst, W., Hofstede, T.: Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages. In: Jensen, K. (ed.) Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002), Proceedings. DAIMI, vol. 560, pp. 1–20 (2002)
22. Kraemer, F.A., Slåtten, V., Herrmann, P.: Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. In: Journal of Systems and Software, 82(12), pp. 2068–2080 (2009)