An ASM Semantics of Token Flow in UML 2 Activity Diagrams

Stefan Sarstedt and Walter Guttmann

University of Ulm, 89069 Ulm, Germany
sarstedt@acm.org · walter.guttmann@uni-ulm.de

Abstract. The token flow semantics of UML 2 activity diagrams is formally defined using Abstract State Machines. Interruptible activity regions and multiplicity bounds for pins are considered for the first time in a comprehensive and rigorous way. The formalisation provides insight into problems with the UML specification, and their solutions. It also serves as a basis for an integrated environment supporting the simulation and debugging of activity diagrams.

1 Introduction

The Unified Modeling Language (UML) is widely used for specification and documentation purposes in the software development process. UML activity diagrams model behaviour aspects of software systems, particularly control and data flow. To provide tool support beyond drawing assistance, and to use activity diagrams effectively, it is necessary to exactly understand their meaning.

While the UML specification [1] is a step forward to define activity diagrams more precisely, it is insufficient for several reasons. First, it is vague, leaving much space for interpretation – as will become evident throughout this paper. Second, it is informal, thus a large gap has to be bridged until it can be usefully applied for model execution and automated reasoning. Third, it contains implausible requirements, e.g., for nested interruptible activity regions as discussed in Sect. 6, which reduces its usability.

We propose a solution to these shortcomings by defining the semantics of activity diagrams using Abstract State Machines [2]. The Abstract State Machine (ASM) specification is precise and therefore it enables to understand the meaning of a model to the utmost detail. It is formal and can therefore serve as a foundation for the implementation of tools. Finally, it helps to ensure that the specified behaviour meets the intuition of the modeller.

The state of the art in semantics for UML 2 activity diagrams covers three distinct approaches: mapping to Petri-nets, using graph transformation rules, or providing pseudo-code. A detailed discussion of related work is given in Sect. 8.

We improve on existing work by imposing less restrictions on activity diagrams, e.g., treating multiplicity bounds for pins and interruptible activity regions. The construction using ASMs leads to enhanced clarity and reveals problematic issues in the UML specification. Our solution also shows how to deal with several of these problems without inflicting any biased decision.

The scope of this paper is to describe the ASM semantics of token flow. This specifies the meaning of a transition from one state to another within an activity diagram. It is, however, only one part of a complete ASM semantics of activity diagrams. The remaining parts deal with events and multiple activity and action executions, and are elaborated in [3].

In Sect. 2 we introduce the basic concepts of activity diagrams and Abstract State Machines. The structure of the token flow semantics is presented in Sect. 3, and the following sections describe its aspects. Token offers are computed in Sect. 4 and selected in Sect. 5. Restrictions we have to impose on activity diagrams are also discussed there. Interruptible activity regions and configurable semantics are dealt with in Sect. 6. A brief summary of our techniques to solve problems we have encountered with the UML specification is given in Sect. 7.

2 Basics

In this section we detail the basic concepts of UML activity diagrams, also called "activities" in [1], and Abstract State Machines [2] to a level needed for the following development. In this paper, by writing UML we mean UML 2.0 unless stated otherwise.

2.1 Activity Diagrams

UML facilitates the modelling of control and object (or data) flow by means of activity diagrams, comprising a multitude of concepts. Several levels are defined that support different parts of these concepts. This paper mainly addresses the *intermediate level* that includes object nodes, concurrent flows with guards, and decisions. We additionally discuss *interruptible activity regions* as an example of a useful feature having a vague semantics.

The fundamental elements of activity diagrams are *actions* that are connected by edges to indicate control and data flow. Actions specify transformations on the state of the system that are not further decomposed within the given diagram. They are either implementation-dependent or more specific, e.g., used to send and receive signals or to invoke behaviour specified in other diagrams. Since this distinction is of no concern for the purposes of the paper at hand, the most general term "action" is used.

Edges connecting actions may pass through *control nodes* that coordinate the flows in an activity diagram. A *decision node* chooses between different outgoing edges and the corresponding *merge node* unites several independent flows. On the other hand, a *fork node* splits a flow into concurrent flows along all outgoing edges and the corresponding *join node* synchronises all incoming flows. Moreover, flows may originate in *initial nodes* and terminate in *final nodes*.

Object nodes allow for object flows in addition to control flows. They arise as *input pins* and *output pins* attached to actions, indicating the delivery of data. On the level of activities, objects can be passed through *activity parameter nodes*. Objects may also be buffered in *central buffer nodes*.

An interruptible activity region is a subset of nodes and edges supporting the termination of parts of an activity diagram. It is further examined in Sect. 6.

Example. Several of these building-blocks are illustrated in the activity diagram shown in Fig. 1 that acts as the running example throughout this paper. It contains actions A and B, activity parameter nodes C and D, central buffer node E, input pin F, control flows e_1–e_3, object flows e_4–e_8, the diamond-shaped decision and merge nodes, the bar-shaped join node, and the bullet-shaped initial node. The decision node's outgoing edges are decorated with guards that indicate the conditions for passing the edges.

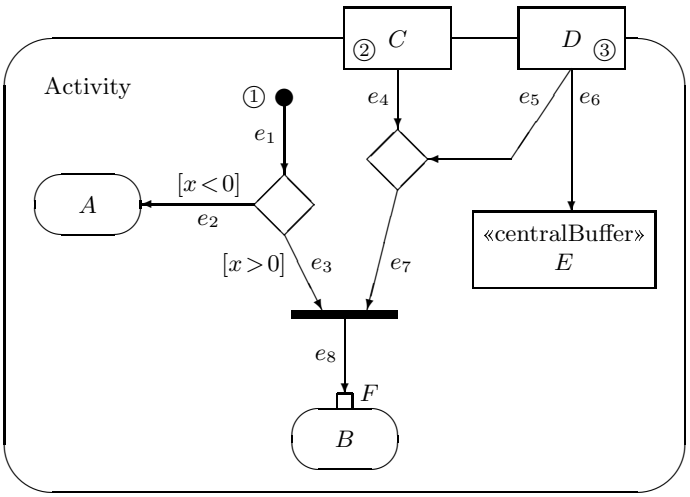

Fig. 1. Activity diagram used as a running example

While Petri-nets are not adequate to describe the semantics of activity diagrams (see Sect. 8), our example can be explained at least in terms of tokens. Upon start of the activity diagram, tokens are available on the nodes C and D, and on the edge e_1. There are three different situations, depending on the value of the attribute x:

- $x < 0$: Token ① may enable action A and token ③ may move to buffer E. The join node must not be traversed.
- $x = 0$: Only token ③ may move to E.
- $x > 0$: Token ② may move to input pin F, enabling action B. Independently, token ③ may move either to F or to E, but not to both nodes.

Note that in any of the three situations, any of the indicated flows may take place, but is not required to.

2.2 Abstract State Machines

Basic ASMs may be viewed as "pseudo-code over abstract data" [2], and indeed we use them as a convenient way to describe computations in this paper. We present a brief overview of the most important concepts and refer to [2] for details, including an operational semantics.

An ASM comprises transition rules that operate on a state composed of functions defined over a base set. The *update rule* $f(s_1, \ldots, s_n) := t$ modifies the value of f at $(s_1, \ldots, s_n)$ to t. In general, several transition rules execute in parallel, requiring that individual updates do not conflict each other. Such conflicts may be avoided by enforcing sequential execution with **seq**. Further constructs include the no-operation **skip**, abstractions using **let** ... **in**, the conditional, and rule calls with call-by-name semantics. The rule **forall** x **with** φ **do** R executes R in parallel for each x satisfying φ. The rule **choose** x **with** φ **do** R chooses some x satisfying φ and then executes R. The rule **iterate** R executes R until it provides no further or inconsistent updates. Borrowed from AsmL, the **add** ... **to** and **remove** ... **from** rules denote non-conflicting, partial updates to sets [4].

3 Flow Computation

In this section we present our main ASM rule for the computation of transitions, describing the structure of the token flow semantics. Relevant terms used by the UML specification are introduced as needed.

The semantics of activity diagrams is specified in terms of tokens [1]. Our transition rule is called whenever tokens are available at actions, initial nodes, or object nodes. According to the UML specification, these nodes *offer* the tokens on their outgoing edges. Tokens can be rejected by edges because their guards evaluate to false, or by nodes not accepting them.

Offered tokens may move, if they are accepted by all intermediate edges and control nodes, as well as their destination nodes. The latter comprise actions, final nodes, and object nodes. The *traverse-to-completion* principle [5] requires that the whole path from the original node to the destination is traversed at once. The definitive goal of our rule is to determine which tokens move, triggering which destinations, and to perform the entailed transition.

The exact working of the propagation of token offers and their selection at destination actions and object nodes, however, is neither formally defined, nor adequately discussed in the specification. Our proposal for transition computation and execution consists of the following main ASM rule that is executed repeatedly as long as control or data tokens are available:

```
    InitialiseFlowsForControlFlowSources
    InitialiseFlowsForObjectFlowSources (see Sect. 4.1)
seq PropagateFlowInformation (see Sect. 4.2)
seq SelectTokenOffers (see Sect. 5)
seq RemoveFlowsInInterruptedRegions (see Sect. 6)
seq ActivateAcceptEventActions
seq ExecuteTransition
```

The InitialiseFlows and PropagateFlowInformation macros spread token offers from the source nodes, where the actual control and data tokens rest, through the activity graph. These macros are described in the following section. After all possible offers have been computed, subsets are selected at destination actions, object and final nodes, preparing the traversal of the associated tokens. The selection mechanism is described in Sect. 5. Note that selecting token offers can invalidate other, conflicting token offers.

Since aborting interruptible activity regions can prevent tokens from traversal, the rule RemoveFlowsInInterruptedRegions removes those selections. Problematic cases unconsidered by the UML specification, including the handling of nested interruptible activity regions, are discussed in Sect. 6.

Accept event action nodes without incoming edges, contained in interruptible activity regions, are initialised by ActivateAcceptEventActions. The actual execution of the token traversal and the termination of actions in interrupted regions is performed by ExecuteTransition. For want of space, both rules are not detailed in this paper but presented in [3] that also deals with event handling and multiple activity and action executions.

4 Computation of Token Offers

The distribution of token offers is performed in two steps. First, new token offers are created for tokens resting at outgoing edges of actions or initial nodes (being sources of control flows), or at object nodes (being sources of object flows). Second, the token offers are propagated through the activity graph towards the consuming destination nodes, namely actions, object nodes, and final nodes.

4.1 Creation of Token Offers

Offers are created by the InitialiseFlows macros. We show the rule for object flow sources, and omit the similar one for control flow sources. The latter joins control tokens offered by the same edge by creating only one token offer.

We use static ASM functions to model the activity diagram being worked on, according to the UML meta model [1]. The domain *ObjectFlowSource* subsumes output pins, central buffer nodes, and incoming activity parameter nodes.

```
InitialiseFlowsForObjectFlowSources ≡
  forall n with n ∈ ObjectFlowSource ∧ |dataTokens(n)| > 0 do
    let m = |outgoing(n)| in
      forall i with 1 ≤ i ≤ m do t(i) := new(TokenOffer)
      seq
      forall i with 1 ≤ i ≤ m do
        let e = outgoing(n, i) in
          if IsGuardTrue(e, Self.context, head(dataTokens(n))) then
            t(i).offeredToken := head(dataTokens(n))
            t(i).paths := {[e]}
            t(i).exclude := {t(j) | 1 ≤ j ≤ m ∧ i ≠ j}
            t(i).include := ∅
            offers(e) := {t(i)}
```

The function *dataTokens* yields the data tokens currently available at a node. If there is more than one, only the first is considered, assuming a FIFO-ordering [1, p. 380]. For each outgoing edge, if its guard evaluates to true for that token, a new instance of the *TokenOffer* structure is initialised. Since the syntax and the implementation of guards are left open by the UML specification, the evaluation is performed by the monitored ASM function *IsGuardTrue*.

The *TokenOffer* structure consists of the following components:

$$\textbf{domain}\ \mathit{TokenOffer} =_{def} \{\mathit{offeredToken} : \mathit{Token};\ \mathit{paths} : \mathcal{P}(\mathit{ActivityEdge}^*);\ \mathit{exclude} : \mathcal{P}(\mathit{TokenOffer});\ \mathit{include} : \mathcal{P}(\mathit{TokenOffer})\}$$

The component *offeredToken* contains the actual data token, whose possible traversal is represented by this offer. The component *paths* represents the path beginning from the source node of the token to the current position of the offer. Actually, a *set* of paths is needed, since control flows must be included when combined with object flows by a join node, as described in Sect. 4.2.

Furthermore, according to the specification [1, p. 381], "a token in an object node can traverse only one of the outgoing edges". Our algorithm must therefore ensure that the offers on these edges exclude each other, as is the case in Sect. 4.2 for decision nodes with non-exclusive guards. Tool implementation requires efficiency, and we therefore avoid to try out all possible combinations for several nodes with competing edges. An appropriate backtracking mechanism is also ruled out, although for other reasons, by [6]. Note that lifting this problem to the interpreting level, e.g., to the ASM choice construct, does not solve it.

Therefore, the component *exclude* of *TokenOffer* collects all conflicting offers. It is initialised to contain the offers on all edges except the current, since all outgoing edges of object nodes compete with each other. By the way, this is not the case for control flow sources, where they are initialised as empty.

The component *include*, finally, contains those offers that have contributed to the current offer. Being initial offers, their *include* set is empty. Both the *exclude* and *include* sets are used for selecting token offers at destination nodes in Sect. 5.

The ASM function $\mathit{offers} : \mathit{ActivityEdge} \to \mathcal{P}(\mathit{TokenOffer})$ stores the new token offers. While it initially maps to singleton sets, multiple offers may exist on a single edge at later stages, as stated explicitly in [1, p. 369]. All offers stored on all incoming edges of a node n are returned by $\mathit{offersForNode}(n)$.

4.2 Propagation of Token Offers

After all initial offers have been created, PropagateFlowInformation distributes them by iteratively calling rules for the join, decision, merge, and fork nodes.

Join. We first deal with join nodes, being the most complex kind. The following ASM rule processes all join nodes of the current activity once, ensured by the predicate *visited*. All previously computed offers on the outgoing edge are cleared and, if there are offers on all incoming edges, we differentiate the two cases specified by [1, p. 369].

```
PropagateFlowForJoinNode ≡
  forall n with n ∈ JoinNode ∧ AreAllPredecessorsVisited(n) ∧ ¬visited(n) do
    let o = outgoing(n) in
      offers(o) := ∅
      seq
      visited(n) := true
      if ∀e ∈ incoming(n) : offers(e) ≠ ∅ then
        if IsControlFlow(o) then PropagateControlFlowForJoinNode(n, o)
                            else PropagateObjectFlowForJoinNode(n, o)
```

If some (or all) incoming edges are object flows, all offers from these flows have to be forwarded. By joining all incoming control flows to the *paths* set of each transmitted token offer, they can be removed if the actual transition of the data token takes place. The *include* and *exclude* sets of each offer contain all control and object flows to prevent conflicting offers to flow that might invalidate the join condition.

```
PropagateObjectFlowForJoinNode(n, o) ≡
  forall e with e ∈ incoming(n) ∧ IsObjectFlow(e) do
    forall t with t ∈ offers(e) do
      if IsGuardTrue(o, Self.context, t.offeredToken) then
        let t' = new(TokenOffer) in
          t'.offeredToken := t.offeredToken
          t'.paths := {p +- o | p ∈ ⋃{s.paths | s ∈ controlFlowOffers(n)} ∪ t.paths}
          t'.exclude := ⋃{s.exclude | s ∈ offersForNode(n)}
          t'.include := ⋃{s.include ∪ {s} | s ∈ offersForNode(n)}
          add t' to offers(o)
```

For this procedure to work we have to impose a restriction: We assume that the incoming token offers are *consistent*, as discussed in Sect. 5. Otherwise, sets of token offers would have to be considered to handle the additional dependences.

A similar rule emits only one token offer, if only control flows are joined.

Decision. The specification of decision nodes requires that each incoming token is offered to those outgoing edges whose guards are satisfied. The modeller must ensure that only one outgoing edge is actually traversed. Additionally, [1, p. 349] states that "if multiple edges accept the token and have approval from their targets for traversal at the same time, then the semantics is not defined".

It is, however, left unspecified what the "approval" proviso means. We propose that any selection of token offers may be chosen as long as no two outgoing edges are traversed simultaneously by the same token. The following fragment of our algorithm shows the use of the *exclude* sets to implement this:

```
forall i with 1 ≤ i ≤ |acceptingEdges| do t(i) := new(TokenOffer)
seq
forall i with 1 ≤ i ≤ |acceptingEdges| do
  t(i).offeredToken := t.offeredToken
  t(i).paths := {p +- elementAt(acceptingEdges, i) | p ∈ t.paths}
  t(i).exclude := t.exclude ∪ {t(j) | 1 ≤ j ≤ |acceptingEdges| ∧ i ≠ j}
  t(i).include := t.include ∪ {t}
  add t(i) to offers(elementAt(acceptingEdges, i))
```

The set *acceptingEdges* contains all edges with true guards. Extending the guard mechanism of edges, "a predefined guard 'else' may be defined for at most one outgoing edge" of a decision node [1, p. 349]. As expected, this guard succeeds only if all other guards fail. The else-guard is easily incorporated into the transition rule as a special case.

Merge. The propagation for the *merge nodes* is considerably simpler, since "all tokens offered on incoming edges are offered to the outgoing edge" [1, p. 374]. We immediately check if the token satisfies the guard of the outgoing edge, and calculate the new token offer as follows:

$$
\begin{array}{l}
\textbf{let } t' = \mathit{new}(\mathit{TokenOffer}) \textbf{ in} \\
\quad t'.\mathit{offeredToken} := t.\mathit{offeredToken} \\
\quad t'.\mathit{paths} := \{p +\!\!+ \mathit{outgoing}(n,1) \mid p \in t.\mathit{paths}\} \\
\quad t'.\mathit{exclude} := t.\mathit{exclude} \\
\quad t'.\mathit{include} := t.\mathit{include} \cup \{t\} \\
\quad \textbf{add } t' \textbf{ to } \mathit{offers}(\mathit{outgoing}(n,1))
\end{array}
$$

Fork. The calculation for the *fork nodes* is almost identical, except that token offers are made at each outgoing edge with a true guard. Our algorithm can be extended to deal with the buffering of tokens at fork nodes [1, p. 363]. Note that, when used in combination with guards, fork buffering leads to unexpected behaviour. The extension is presented in [3], along with a discussion of the problems caused by the UML specification.

4.3 Example Computation

Figure 2 contains the computed token offers for our example shown in Fig. 1, assuming $x > 0$. The offers 1, 3, 4 and 5 are computed by the InitialiseFlows rules, whereas the remaining offers are added by the Propagate rules. The next section explains how the propagated offers are selected at destination nodes.

Edge	Offers, $id : (\mathit{offeredToken}, \mathit{paths}, \mathit{exclude}, \mathit{include})$
e_1	$1 : (-, \{[e_1]\}, \emptyset, \emptyset)$
e_2	no offers
e_3	$2 : (-, \{[e_1, e_3]\}, \emptyset, \{1\})$
e_4	$3 : (②, \{[e_4]\}, \emptyset, \emptyset)$
e_5	$4 : (③, \{[e_5]\}, \{5\}, \emptyset)$
e_6	$5 : (③, \{[e_6]\}, \{4\}, \emptyset)$
e_7	$6 : (②, \{[e_4, e_7]\}, \emptyset, \{3\})$ and $7 : (③, \{[e_5, e_7]\}, \{5\}, \{4\})$
e_8	$8 : (②, \{[e_4, e_7, e_8], [e_1, e_3, e_8]\}, \{5\}, \{1, 2, 3, 4, 6, 7\})$ and $9 : (③, \{[e_5, e_7, e_8], [e_1, e_3, e_8]\}, \{5\}, \{1, 2, 3, 4, 6, 7\})$

Fig. 2. Computed offers for the example in Fig. 1

5 Selection of Token Offers

Once the token offers are computed, we select subsets of them to participate in the planned transition. The transition may lead, e.g., to the start of a new action as described by the specification [1, p. 302]. Other possibilities are moving tokens to central buffer nodes, outgoing activity parameter nodes, and final nodes. The UML specification, however, does not indicate what to perform if there are enough token offers to conduct several of these operations.

We use the ASM iteration and non-deterministic choice constructs to adhere to the specification. The iteration may be stopped by choosing $n = \mathsf{skipSelection}$ at any stage. Otherwise, we select token offers depending on the kind of node.

```
SelectTokenOffers ≡
  iterate
    choose n with n ∈ {skipSelection} ∪ Action ∪ FinalNode
          ∪ CentralBufferNode ∪ OutgoingActivityParameterNode do
      if n ∈ Action then SelectTokenOffersForAction(n)
      if n ∈ FinalNode then SelectTokenOffersForFinalNode(n)
      if n ∈ CentralBufferNode ∪ OutgoingActivityParameterNode then
        SelectTokenOffersForCentralBufferAndParameterNode(n)
```

In the following, we focus on action nodes. The selection for the other kinds of nodes is dealt with by similar, even simpler rules.

For action nodes, we select a subset of token offers S_i for each input pin i of the action. Conditions for the acceptance of tokens by input pins are that the number of selected tokens is between *lower* and *upper* [1, p. 249], and that the total number of tokens resting on each pin does not exceed its upper bound [1, p. 380]. If an appropriate selection has been found, we commit to it and update the remaining offers according to the specification.

The selected subsets are accumulated in *tokenSelections*. The function *taken* keeps track of the selections for each object node to make sure they do not overflow. The input pins of action n are provided by $input(n)$ according to the UML meta model.

```
SelectTokenOffersForAction(n) ≡
  if ∀e ∈ incoming(n) : offers(e) ≠ ∅ then
    let p = |input(n)| in
      choose S_1, ..., S_p with ∀1 ≤ i ≤ p : ∃j = input(n, i) : S_i ⊆ offersForNode(j)
            ∧ lower(j) ≤ |S_i| ≤ upper(j)
            ∧ |S_i| + taken(j) + |dataTokens(j)| ≤ upperBound(j) do
        let selection = ⋃{S_i | 1 ≤ i ≤ p} ∪ offersForNode(n) in
          UpdateOffers(selection)
          tokenSelections := tokenSelections ∪ {(n, selection)}
          forall i with 1 ≤ i ≤ p do taken(input(n, i)) := taken(input(n, i)) + |S_i|
```

The UpdateOffers rule removes the selection of token offers and all offers inconsistent to it. If any inconsistent offers are removed from a *join* node, we re-propagate this information using the rules introduced in Sect. 4, since the removal may affect other token offers originating at that node.

UpdateOffers($selection$) $\equiv$
forall n **with** $n \in JoinNode$
$\quad \wedge \exists e \in incoming(n) : \exists t \in offers(e) : \exists t' \in selection : \neg AreConsistent(t, t')$ **do**
forall n' **with** $n' \in AllControlNodeSuccessors(n) \cup \{n\}$ **do** $visited(n') := false$
forall e **with** $e \in ActivityEdge$ **do**
$offers(e) := \{t \in offers(e) \mid t \notin selection \wedge \forall t' \in selection : AreConsistent(t, t')\}$
seq
PropagateFlowInformation

The symmetric predicate *AreConsistent* is used to calculate which token offers must be removed according to the specification.

$$\begin{aligned} &AreConsistent : TokenOffer \times TokenOffer \to Boolean \\ &AreConsistent(t_1, t_2) =_{def} (t_1.exclude \cap (t_2.include \cup \{t_2\}) = \emptyset) \\ &\qquad\qquad \wedge (t_2.exclude \cap (t_1.include \cup \{t_1\}) = \emptyset) \end{aligned}$$

To avoid an inefficient search at each destination node, we assume that the incoming token offers are consistent. The modeller can ensure this, and also the consistency required for join nodes in Sect. 4, e.g., by placing appropriate guards on competing edges leading to the same destination nodes. A stronger, syntactic condition is the absence of two paths from the same decision or object node to the same action, final, join, or object node. Here, an action together with its input pins is considered as one node.

Example Selection. Let us discuss the case $x > 0$ of our running example shown in Fig. 1, assuming further that the action node B is chosen by SelectTokenOffers. Then, either of the offers 8 and 9 shown in Fig. 2 may be selected, or both of them, by SelectTokenOffersForAction. In any case, the *exclude* set of the selection contains the offer 5 that is therefore removed by UpdateOffers. Since all offers into the join node are consistent with the selection, no further propagation is performed.

If, on the other hand, central buffer E was chosen by SelectTokenOffers, and offer 5 selected, the offers 4, 7, 8, and 9 would be removed. By re-propagating from the join node, offer $8 : (②, \{[e_4, e_7, e_8], [e_1, e_3, e_8]\}, \emptyset, \{1, 2, 3, 6\})$ is reconstructed and may be chosen in the next iteration.

6 Interruptible Activity Regions

After token offers have been selected for destination nodes, we determine which interruptible activity regions are aborted and eliminate flows that are in conflict with those regions.

If one of the selected offers passes an interrupting edge [1, p. 366], all tokens in the interrupted region must be removed, and therefore all offers of these tokens have to be removed as well. Note that it is permitted to have concurrent, non-interrupting flows out of aborted regions.

The specification, however, gives no information regarding concurrent flows leading *into* aborted regions. Figure 3 shows the offer ① that originates from a

node outside of the region, and the offer ② that re-enters the region after having left it. Since either keeping or destroying such tokens can be useful, our algorithm can be adapted to both alternatives. To this end, we introduce the configuration of semantics by using UML tags, a standard extension mechanism of the UML. This mechanism has already been applied successfully to the configuration of signal handling [7]. In Fig. 3, the ignoreFlowIntoInterruptedRegion tag is used that is queried in the following ASM rule.

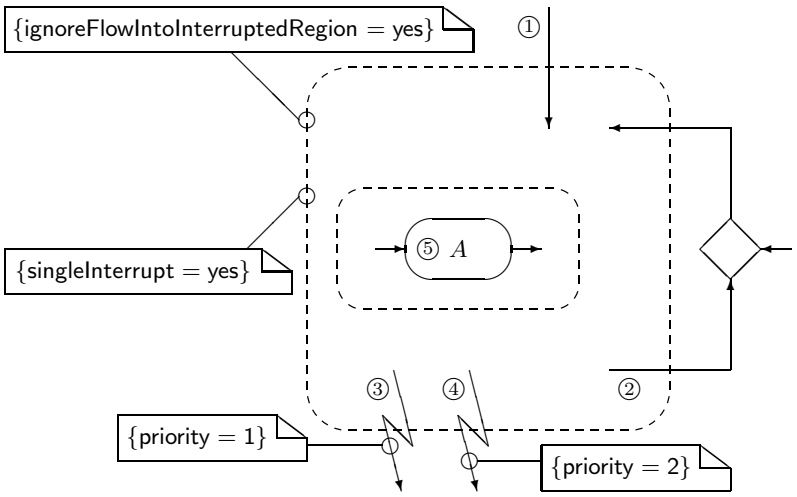

Fig. 3. Issues with interruptible activity regions

In general, multiple interrupting edges can be passed at the same time (see offers ③ and ④ in Fig. 3), leading to another scenario where configuration is useful. To enable the user to specify that only a single edge may be passed, the singleInterrupt and priority tags are defined. If singleInterrupt is enabled for a region, our algorithm selects the edge with the highest annotated priority, implemented by ChooseIntEdge. Offer selections for the other interrupting edges are then discarded.

Interruptible activity regions, being activity groups, are also allowed to be nested. A major deficiency of the UML specification is missing information about how to deal with them. According to the specification [1, p. 323], "no node or edge in a group may be contained by its subgroups or its containing groups, transitively". This means that, when a region is aborted, its nested regions are not. Instead of this unexpected behaviour we propose to interrupt all nested regions. Token ⑤ would thus be removed in Fig. 3 if offer ③ was selected for traversal.

The following ASM rule implements the discussion just carried out. For all regions that are to be aborted, we remove all selected offers that do not leave the

region, as checked by *HasInnerFlow*. We furthermore eliminate flows according to the specified configuration tags. Finally, all nested regions are marked for termination.

RemoveFlowsInInterruptedRegions ≡
  forall r **with** $r \in \mathit{InterruptibleActivityRegion} \wedge \mathit{IsInterrupted}(r)$
      $\wedge \nexists r' \in \mathit{parents}(r) : \mathit{IsInterrupted}(r')$ **do**
    remove $\{s \in \mathit{tokenSelections} \mid \mathit{HasInnerFlow}(s, r)\}$ **from** *tokenSelections*
    if $\mathit{tagValue}(r, \mathsf{singleInterrupt}) = \mathsf{yes}$ **then**
      let $e = \textsc{ChooseIntEdge}(\bigcup\{\mathit{interruptingEdge}(s, r) \mid s \in \mathit{tokenSelections}\})$ **in**
        remove $\{s \in \mathit{tokenSelections} \mid \mathit{Interrupts}(s, r) \wedge e \notin \mathit{interruptingEdge}(s, r)\}$
          from *tokenSelections*
    if $\mathit{tagValue}(r, \mathsf{ignoreFlowIntoInterruptedRegion}) = \mathsf{yes}$ **then**
      remove $\{s \in \mathit{tokenSelections} \mid \mathit{HasFlowInto}(s, r)\}$ **from** *tokenSelections*
    add $\{r\} \cup \mathit{children}(r)$ **to** *regionsToInterrupt*

7 Solving Problems of UML

Let us finally compare the problems with the UML specification we have encountered and the ways we solve them. Deliberate under-specifications are modelled by the non-deterministic choice of ASMs (e.g., see Sect. 5). To deal with open issues that can be decided by the modeller we introduce UML tags that are queried from the ASM rules. If the specification should have decided, e.g., concerning non-exclusive guards on competing edges, we propose a decision. Unintuitive consequences of requirements in the specification (e.g., of fork buffering) are discussed in detail, also by providing alternative implementations [3]. Obvious errors, e.g., for nested interruptible activity regions, are corrected.

8 Related Work

Existing work covers the semantics for UML 1.∗, including an ASM semantics for activity diagrams, excerpts of which are presented in [8]. For historical reasons, however, UML 1.∗ activity diagrams are special kinds of state charts. In UML 2.0 they have been completely redefined. We therefore discuss only UML 2.0 related work in the following.

Since the UML specification envisions a "Petri-like semantics" for activity diagrams [1, p. 314], it is manifest to propose a mapping between the two. Störrle [9, 10] uses different variants of Petri-nets, e.g., coloured Petri-nets for data flow, and procedural Petri-nets for activities. The treatment of join nodes having mixed object and control flows is, however, neither discussed nor obvious. The development culminates in [11] concluding that Petri-nets might, after all, not be appropriate for formalising activity diagrams. Especially mapping advanced concepts, such as interruptible activity regions, is found not to be intuitive. Moreover, the lack of a unified Petri-net formalism, integrating the different variants used to map different concepts, is observed. Assuring the traverse-to-completion semantics is identified as another problem. The related paper [12] also translates to Petri-nets, but focuses on the parameters of actions.

Vitolins and Kalnins [13] present an algorithm for computing the token flow, proposing a forward and backward search by using so-called "push" and "pull" engines. Several far reaching restrictions are, however, imposed on activity diagrams. Decision nodes must have mutually exclusive guards, and object nodes must have no outgoing concurrent edges. This simplifies their algorithm, since they do not have to pull all input tokens in one atomic step – traverse-to-completion is thus not observed. Fork and join nodes must not be on the same path between two actions. Tokens resulting from join nodes are grouped, which is neither prohibited nor stated in the specification.

Hausmann [6] formalises activity diagrams using "Dynamic Meta Modeling", where graph transformation rules operate on an instance of the UML meta model. The transformation engine is responsible to resolve the non-determinism occurring at competing edges of object and decision nodes. This renders the approach too inefficient to serve as a basis for tool support. The semantics of a large part of activity diagrams is described very detailed and problems of the UML specification are discussed. Apart from this, several restrictions apply also to this work. Only one offer is allowed per edge, and – as a consequence – when different data tokens are offered to a join node, only one of them is forwarded. Guards and interruptible activity regions are not supported.

The ongoing UML Semantics Project [14] aims at formalising a subset of UML by providing "a strong foundation for the definition of a UML virtual machine that is capable of executing UML 2.0 models". The Modelware Project [15] implements a tool capable of simulating basic activity diagrams, but only with control flows. Currently, no formalisation of the algorithms behind their execution engine is available.

Our paper shows how to deal with the restrictions mentioned before. Moreover, none of the works discussed so far, and none that we know of, handles the problems presented in Sect. 6 related to interruptible activity regions, including incoming flows, multiple interrupting edges, and nested regions. The useful feature of *lower* and *upper* multiplicity bounds on pins is also not treated elsewhere.

9 Conclusion

We formalise the semantics of token flow in UML 2 activity diagrams in terms of ASM rules. The resulting rules can be traced back to requirements present in or absent from the UML specification. Our contribution deals with several features neglected elsewhere, such as interruptible activity regions and multiplicity bounds for pins. The part presented in this paper is embedded into rules for asynchronous multi-agent ASMs specifying signal handling and activity and action executions [3].

The formalisation is high-level enough to reveal problematic issues with the UML specification. On the other hand, it can be directly executed using the AsmL compiler. Furthermore, it is suitable to serve as a basis for tool support, e.g., for model checking [16] and verification [17]. An integrated environment

has been implemented [18], supporting the simulation and debugging of activity diagrams.

References

1. Object Management Group: UML 2.0 Superstructure Specification. (2005)
2. Börger, E., Stärk, R.: Abstract State Machines. Springer-Verlag (2003)
3. Sarstedt, S.: Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams. PhD thesis, Universität Ulm (2006)
4. Gurevich, Y., Tillmann, N.: Partial updates: Exploration. Journal of Universal Computer Science **7**(11) (2001) 917–951
5. Bock, C.: UML 2 activity and action models part 4: Object nodes. Journal of Object Technology **3**(1) (2004) 27–41
6. Hausmann, J.: Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages. PhD thesis, Universität Paderborn (2005)
7. Sarstedt, S.: Overcoming the limitations of signal handling when simulating UML 2 activity charts. In Feliz-Teixeira, J., Carvalho Brito, A., eds.: Proceedings of the 2005 European Simulation and Modelling Conference (ESM'05). (2005) 61–65
8. Börger, E., Cavarra, A., Riccobene, E.: An ASM semantics for UML activity diagrams. In Rus, T., ed.: Algebraic Methodology and Software Technology. Volume 1816 of Lecture Notes in Computer Science, Springer-Verlag (2000) 293–308
9. Störrle, H.: Semantics of control-flow in UML 2.0 activities. In: Symposium On Visual Language And Human Centric Computing. IEEE (2004) 235–242
10. Störrle, H.: Semantics and verification of data flow in UML 2.0 activities. In Minas, M., ed.: Workshop on Visual Languages and Formal Methods. Volume 127, Issue 4 of Electronic Notes in Theoretical Computer Science, Elsevier (2005) 35–52
11. Störrle, H., Hausmann, J.: Towards a formal semantics of UML 2.0 activities. In Liggesmeyer, P., Pohl, K., Goedicke, M., eds.: Software Engineering 2005. Volume P-64 of Lecture Notes in Informatics, Gesellschaft für Informatik (2005) 117–128
12. Barros, J., Gomes, L.: Actions as activities and activities as Petri nets. In Jürjens, J., Rumpe, B., France, R., Fernandez, E., eds.: Critical Systems Development with UML: Proceedings of the UML'03 workshop. TUM-I0317 (2003) 129–135
13. Vitolins, V., Kalnins, A.: Semantics of UML 2.0 activity diagram for business modeling by means of virtual machine. In: Ninth International EDOC Enterprise Computing Conference. IEEE (2005) 181–192
14. UML 2.0 Semantics Project: Web page (2006) http://www.cs.queensu.ca/~stl/internal/uml2/
15. Modelware Project, WP1 Modelling Techniques: D1.3 Model Simulation Scheme: Definition (2005) available from http://www.modelware-ist.org/
16. Winter, K.: Model Checking Abstract State Machines. PhD thesis, Technische Universität Berlin (2001)
17. Gargantini, A., Riccobene, E.: Encoding abstract state machines in PVS. In Gurevich, Y., Kutter, P., Odersky, M., Thiele, L., eds.: Abstract State Machines: Theory and Applications. Volume 1912 of Lecture Notes in Computer Science, Springer-Verlag (2000) 303–322
18. Sarstedt, S., Gessenharter, D., Kohlmeyer, J., Raschke, A., Schneiderhan, M.: ActiveChartsIDE: An integrated software development environment comprising a component for simulating UML 2 activity charts. In Feliz-Teixeira, J., Carvalho Brito, A., eds.: Proceedings of the 2005 European Simulation and Modelling Conference (ESM'05). (2005) 66–73