Operational Semantics of Poka Yoke Activities

February 14, 2025

1 Introduction

This document defines a concise token passing style operational semantics of (simple UML-like) activities. These activities and their semantics are slightly different from fUML activities. For example, every action has an associated guard and effects that specify how action execution influences the system state.

The strategy for defining the operational semantics of activities is to first define a notion of *core activities* (Section 2) with an intuitive and concise semantics, and then define the semantics of concrete UML activities by means of translating them to core activities (Section 3).

The reason for such a translation is two-fold. Firstly, even though UML activities have various sorts of nodes ('fork', 'join', 'decision', etc.), their semantics can fundamentally be divided into two categories: and-type nodes and or-type nodes, which are explained later. If one would specify separate execution rules for all the various sorts of UML activity nodes, then one would find out they are all much the same and would fall into these two categories.

Secondly, the semantics of and-type and or-type nodes is intuitive as well as easy to define. Instead of having to think about all kinds of corner cases with UML activities[1], the semantics of core activities requires no structural well-formedness conditions. When thinking of the semantics of some node, ideally one would only need to think about: is it an and-type or an or-type node?

2 Core Activities

This section defines the (static) structure and (dynamic) behavior of core activities, in Section 2.1 and Section 2.2, respectively.

[1]For example: what would happen if an action node has multiple incoming/outgoing edges? What would happen if a fork node has a self-loop? What if a final node has an outgoing edge, or an initial node has an incoming edge? What if a merge node has no outgoing edges? What if the activity actually contains disjoint disconnected parts? What if there is a non-trivial guard on an edge not coming out of a decision node? Do I need a legion of well-formedness conditions in order to be able to still make sense of activity diagrams at all?

2.1 Statics

This section defines the static structure of core activities, consisting of nodes and (guarded) edges. Nodes contain an action to be executed, and describe how the action should be executed. Actions contain guards, which are predicates over a global state, as well as (nondeterministic) effects that determine potential successor states. The concept of state is left abstract in this document, and could later be instantiated in any way you like (e.g., by taking CIF expressions and updates).

State. Activities are defined in the presence of *state*, for example a set of variables and their current valuation. We keep the notion of state more abstract for the purpose of defining the semantics, and let State be the set of all states. Users of this semantics could later instantiate State as desired, e.g., as variable valuation mappings. Let $\sigma \in \mathsf{State}$ be a typical state.

Guards and effects. Guards are defined to be state predicates, while effects are defined to be state transformers, i.e., functions that map states to any number of 'successor states', to be able to model nondeterministic action behavior.

Let $\mathsf{Guard} = \mathsf{State} \to \mathbb{B}$ be the set of all *guards* and $\mathsf{Effects} = \mathsf{State} \to 2^{\mathsf{State}}$ be the set of all *effects functions* over states, with 2^{State} the powerset of State. We use $g \in \mathsf{Guard}$ and $u \in \mathsf{Effects}$ to denote a typical guard and effects function, respectively.

Actions. The main building blocks of activities are the action nodes, which are nodes that carry out a certain *action*. Any action consists of three parts: an action label to identify it, a guard that must hold in order for the action to be performed, and an effects function that determines possible successor states after having performed the action. For any action it holds that, if its guard does not hold, or if there is no successor state determined by the effects (i.e., the effects function maps to an empty set), then the action cannot be performed. In case action execution leads to multiple possible successor states, one of them will be selected nondeterministically, which is further explained in Section 2.2.

More formally, let $\mathsf{Action} = \mathsf{Label} \times \mathsf{Guard} \times \mathsf{Effects}$ be the set of all *actions*, where Label is the set of all *action labels*. We use $a \in \mathsf{Label}$ to denote a typical label, and $\alpha \in \mathsf{Action}$ to denote a typical action, such that $\alpha = (a, g, u)$. So actions are triples consisting of an action label a, a guard g, and effects function u. Instead of effects function, we may say that u are the *effects of* α.

Let $\mathsf{label} : \mathsf{Action} \to \mathsf{Label}$, $\mathsf{guard} : \mathsf{Action} \to \mathsf{Guard}$, and $\mathsf{effects} : \mathsf{Action} \to \mathsf{Effects}$ be projection functions that give the label, guard, and effects function of any action, respectively, so that:

$$\mathsf{label}((a, g, u)) = a \qquad \mathsf{guard}((a, g, u)) = g \qquad \mathsf{effects}((a, g, u)) = u$$

Core nodes. Now that actions have been defined, we can define the *nodes* of core activities, which carry out these actions during executions of the activity.

Core activity nodes consist of two parts: a node identifier, and a node type that also contains (wraps) the action to perform. The node identifiers are used to give identity to nodes, which are needed to allow having multiple nodes in an activity that perform the same action. In core activities, actions can be executed in one of two ways: in and-style or in or-style, as indicated by the node type. An and-style execution of an action node requires that *all* its incoming edges must have a token, and after having performed the action, *all* outgoing edges will get a token (under some additional conditions that are explained later). In contrast, an or-style execution of an action node requires just *a single* incoming edge to have a token, and after having performed the action, *a single* outgoing edge will receive a token (again, under some extra conditions). This distinction between and-type and or-type nodes is also made in [2] with respect to a token firing semantics, explaining that most UML node types can be divided into these two groups. For example, UML nodes like 'fork' and 'join' can be represented as and-nodes, while 'decision' and 'merge' can be represented in terms of or-nodes. Section 3 further details such representations of concrete UML nodes.

More formally, let NodeID be a set of all *node identifiers*, used to give identity to activity nodes, and let $\ell \in \mathsf{NodeID}$ be a typical node identifier. Then the set $\mathsf{CoreNode}$ of all possible *core activity nodes* is defined as follows:

$$\mathsf{CoreNode} = \mathsf{NodeID} \times \mathsf{CoreNodeType} \qquad \mathsf{CoreNodeType} ::= \mathsf{and}\langle\alpha\rangle \mid \mathsf{or}\langle\alpha\rangle$$

Core activity nodes are pairs consisting of a node identifier ℓ and an action α to execute, either in and-style or in or-style. We write $\eta \in \mathsf{CoreNode}$ to denote a typical node. Let $\mathsf{action} : \mathsf{CoreNode} \to \mathsf{Action}$ be a projection function for obtaining the actions of core nodes, so that:

$$\mathsf{action}((\ell, \mathsf{and}\langle\alpha\rangle)) = \alpha \qquad \mathsf{action}((\ell, \mathsf{or}\langle\alpha\rangle)) = \alpha$$

Core activities. Let us now define *core activities*, which are essentially graphs consisting of core nodes, as defined in the previous paragraph, that are connected via guarded edges. The edges of core activities have two guards, namely an *incoming guard* and an *outgoing guard*. The incoming guards are (later) used to restrict when an edge can *receive* a token, while outgoing guards restrict when a token can be *taken* from an edge. That is, during activity execution, any edge can only receive a token whenever the activity is in a state where the incoming edge guard holds, and this token can only be taken from the edge whenever the outgoing edge guard holds. Incoming edge guards are for example needed to define the semantics of UML decision nodes, of which an outgoing edge can only receive a token if its guard holds. Outgoing edge guards are for example needed for defining extra (waiting) conditions for an action to be performed.

More formally, core activities $\mathcal{A} = (\mathcal{N}, \mathcal{E})$ are graphs with nodes $\mathcal{N} \subseteq \mathsf{CoreNode}$ and edges $\mathcal{E} \subseteq \mathcal{N} \times \mathsf{Guard} \times \mathsf{Guard} \times \mathcal{N}$. Any edge $\varepsilon \in \mathcal{E}$ so that $\varepsilon = (\eta, g_{in}, g_{out}, \eta')$ consists of a source node $\eta \in \mathcal{N}$, an incoming guard $g_{in} \in \mathsf{Guard}$, an outgoing guard $g_{out} \in \mathsf{Guard}$, and a target node $\eta' \in \mathcal{N}$. Let us define the following projection functions for edges:

- $\mathsf{source} : \mathcal{E} \to \mathcal{N}$ such that $\mathsf{source}((\eta, g_{in}, g_{out}, \eta')) = \eta$.
- $\mathsf{guard_{in}} : \mathcal{E} \to \mathsf{Guard}$ such that $\mathsf{guard_{in}}((\eta, g_{in}, g_{out}, \eta')) = g_{in}$.
- $\mathsf{guard_{out}} : \mathcal{E} \to \mathsf{Guard}$ such that $\mathsf{guard_{out}}((\eta, g_{in}, g_{out}, \eta')) = g_{out}$.
- $\mathsf{target} : \mathcal{E} \to \mathcal{N}$ such that $\mathsf{target}((\eta, g_{in}, g_{out}, \eta')) = \eta'$.

We define the following shorthand notations for any $\eta \in \mathcal{N}$ and $\sigma \in \mathsf{State}$:

- $\mathsf{in}(\eta) = \{\varepsilon \in \mathcal{E} \mid \mathsf{target}(\varepsilon) = \eta\}$ be the set of all incoming edges of η.
- $\mathsf{out}(\eta) = \{\varepsilon \in \mathcal{E} \mid \mathsf{source}(\varepsilon) = \eta\}$ be the set of all outgoing edges of η.
- $\mathsf{in}(\eta, \sigma) = \{\varepsilon \in \mathsf{in}(\eta) \mid \mathsf{guard_{out}}(\varepsilon)(\sigma)\}$ be all incoming edges of η whose outgoing guard holds with respect to σ.
- $\mathsf{out}(\eta, \sigma) = \{\varepsilon \in \mathsf{out}(\eta) \mid \mathsf{guard_{in}}(\varepsilon)(\sigma)\}$ be all outgoing edges of η whose incoming guard holds with respect to σ.

2.2 Dynamics

This section defines the dynamic behavior of core activities, by means of a token passing style operational semantics. Its reduction rules describe how to go from one configuration to another, where a configuration is essentially a set of edges holding a token together with a ‘current’ state. There are two reduction rules: one for and-style and one for or-style action execution, as explained earlier. Finally, we introduce initial configurations and execution traces.

Configurations. The semantics of core activities is essentially defined as a relation between *configurations*, in the sense that an execution step in an activity gets you from one configuration to another configuration. A configuration describes which edges currently hold a token, and what the current state is.

More formally, let $\mathsf{Config} = 2^{\mathcal{E}} \times \mathsf{State}$ be the set of all configurations. A configuration $(\Sigma, \sigma) \in \mathsf{Config}$ is a pair with $\Sigma \subseteq \mathcal{E}$ a set of edges—the ones currently holding a token—and σ a ‘current’ state. We write $c \in \mathsf{Config}$ to denote a typical configuration, such that $c = (\Sigma, \sigma)$.

Any edge ε is said to be *enabled in* c if $\varepsilon \in \Sigma$ and $\mathsf{guard_{out}}(\varepsilon)(\sigma)$ holds, i.e., if ε has a token in c and if the outgoing guard of ε holds in c. If c is clear from the context, we may just say that ε is *enabled*.

Reductions. Let us now define the operational semantics of core activities, in token passing style. The reduction rules of the operational semantics describe how edge tokens are distributed, and how state changes, while executing the nodes of the activity. There are two reduction rules: one for and-style execution and one for or-style execution of actions. As explained earlier, these rules only differ in their requirement on, and distribution of, edge tokens.

More formally, the operational semantics of core activities $\mathcal{A} = (\mathcal{N}, \mathcal{E})$ is defined in terms of labeled reduction rules $\rightarrow \subseteq \mathsf{Config} \times \mathcal{N} \times \mathsf{Config}$. The notation $c \xrightarrow{\eta} c'$ is used as shorthand for $(c, \eta, c') \in \rightarrow$. The two rules are:

$$\textsc{And}\quad \frac{\begin{array}{ccc} \mathsf{in}(\eta) = \mathsf{in}(\eta, \sigma) & \mathsf{in}(\eta) \subseteq \Sigma & (\mathsf{out}(\eta) \setminus \mathsf{in}(\eta)) \cap \Sigma = \emptyset \\ \mathsf{guard}(\alpha)(\sigma) & \sigma' \in \mathsf{effects}(\alpha)(\sigma) & \mathsf{out}(\eta) = \mathsf{out}(\eta, \sigma') \end{array}}{(\Sigma, \sigma) \xrightarrow{\eta} ((\Sigma \setminus \mathsf{in}(\eta)) \cup \mathsf{out}(\eta), \sigma')} \text{ for } \eta = (\ell, \mathsf{and}\langle\alpha\rangle)$$

$$\textsc{Or}\quad \frac{\begin{array}{c} \varepsilon \in \mathsf{in}(\eta, \sigma) \cap \Sigma \\ \mathsf{guard}(\alpha)(\sigma) \qquad \sigma' \in \mathsf{effects}(\alpha)(\sigma) \qquad \varepsilon' \in \mathsf{out}(\eta, \sigma') \setminus (\Sigma \backslash \{\varepsilon\}) \end{array}}{(\Sigma, \sigma) \xrightarrow{\eta} ((\Sigma \setminus \{\varepsilon\}) \cup \{\varepsilon'\}, \sigma')} \text{ for } \eta = (\ell, \mathsf{or}\langle\alpha\rangle)$$

The AND rule defines the execution of and-type nodes η, for which $\mathsf{action}(\eta) = \mathsf{and}\langle\alpha\rangle$. AND has six premises. It requires: (1) the outgoing guards of all incoming edges into η to hold with respect to the current state σ; (2) all incoming edges to have a token; (3) none of the outgoing edges of η except self-loops to have a token, otherwise action execution might attempt to put more than one token on a single edge, which is not allowed; (4) the action guard $\mathsf{guard}(\alpha)$ to hold with respect to σ; (5) a successor state σ' to be available from the effects $\mathsf{effects}(\alpha)$ of α; and (6) the incoming guards of all outgoing edges to hold with respect to σ'. Note that premises (1) and (2) together mean that all incoming edges into η must be enabled in the current configuration. If all six premises are met, AND removes all tokens from $\mathsf{in}(\eta)$ and puts tokens on $\mathsf{out}(\eta)$, making $(\Sigma \setminus \mathsf{in}(\eta)) \cup \mathsf{out}(\eta)$ the new token arrangement.

The OR rule defines the execution of or-type nodes η, for which $\mathsf{action}(\eta) = \mathsf{or}\langle\alpha\rangle$. OR has four premises. It requires: (1) the existence of an incoming edge ε of η that is enabled, i.e., that has a token and whose outgoing guard holds; (2) the guard of α to hold with respect to the current state σ; (3) a successor state σ' to be available from α's effects; and (4) the existence of an outgoing edge ε' of η which does not hold a token unless it equals ε, and whose incoming guard holds with respect to σ'. The OR rule allows multiple input edges to be enabled, but only one of them will participate per application of the OR rule. Similarly, if multiple outgoing edges could potentially participate, one of them actually participates. If all four premises are met, OR removes the token from ε and puts it on ε', making $(\Sigma \setminus \{\varepsilon\}) \cup \{\varepsilon'\}$ the new arrangement of tokens.

Finally, as a shorthand notation, we may write $c \xrightarrow{\alpha} c'$ if there exists an $\eta \in \mathcal{N}$ such that $c \xrightarrow{\eta} c'$ and $\mathsf{action}(\eta) = \alpha$.

Traces. AND and OR describe single-step executions. However, when executing or simulating an activity, typically multiple steps are performed, perhaps even infinitely many, leading to the concept of traces, i.e., sequences of steps.

Given any finite action sequence $\pi = \alpha_0\alpha_1 \ldots \alpha_m$ we write $c \xrightarrow{\pi}_* c'$ if there exists a reduction sequence $c_0 \xrightarrow{\alpha_0} c_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_m} c_{m+1}$ such that $c = c_0$ and

$c' = c_{m+1}$. Moreover, given any infinite action sequence $\pi = \alpha_0\alpha_1 \ldots$ we write $c \xrightarrow{\pi}_{\omega}$ if there exists a reduction sequence $c_0 \xrightarrow{\alpha_0} c_1 \xrightarrow{\alpha_1} \ldots$ where $c = c_0$.

3 Concrete Activities

This section defines concrete activities (Section 3.1) that more closely resemble UML activities than core activities, and defines their operational semantics in terms of translations to core activities (Section 3.2). Section 3.3 discusses how to deal with several extensions to concrete activities, like hierarchy and nonatomic actions. Finally, some similarities and differences of concrete activities with respect to fUML semantics are highlighted (Section 3.4).

3.1 Statics

First we define concrete activities, as a subset of UML activities.

Nodes. In contrast to core nodes as defined earlier in Section 2.1, which come only in two flavours (and and or), UML activities have many more flavours of activity nodes, like initial nodes, fork nodes, etc. We focus on a basic subset of UML, leaving out more advanced concepts like signals, object flows, exception handlers, interruptible regions, etc.

Let $\mathsf{ConcrNode} = \mathsf{NodeID} \times \mathsf{ConcrNodeType}$ be the set of all *concrete activity nodes*, consisting of an identifier and a (concrete) node type that is defined as:

$$\mathsf{ConcrNodeType} ::= \mathsf{init} \mid \mathsf{final} \mid \mathsf{fork} \mid \mathsf{join} \mid \mathsf{decision} \mid \mathsf{merge} \mid \mathsf{act}(\alpha)$$

We denote a typical activity node by $n \in \mathsf{ConcrNode}$. Action-typed nodes $\mathsf{act}(\alpha)$ closely relate to opaque actions in UML in the sense that their execution updates the current state according to α (possibly nondeterministically so in case α's effects has multiple successor states to choose from). The other node types are standard in UML: initial and final nodes (init and final), fork and join nodes (fork and join), and decision and merge nodes ($\mathsf{decision}$ and merge)

Activities. Concrete activities $A = (N, E)$ are graphs with $N \subseteq \mathsf{ConcrNode}$ a set of concrete activity nodes and $E \subseteq N \times \mathsf{Guard} \times \mathsf{Guard} \times N$ a set of edges that have an incoming and outgoing guard.

Any number of additional well-formedness conditions may be imposed on activities, e.g., action nodes having exactly one incoming and outgoing edge, or final nodes not being allowed to have outgoing edges. However, the semantics of activities, as defined in Section 3.2, is not dependent on any such additional structural well-formedness conditions, making the semantics easier to define.

Moreover, note that every edge has an incoming and outgoing guard, rather than, e.g., just the ones coming out of decision nodes. This is slightly different from UML activities, since in UML, any edge can only have a single guard, and this guard is ignored unless the edge comes out of a decision node. Any edge guards that are not ignored are then considered to be incoming guards in UML. Our definition of concrete activities is therefore more general than the standard UML activities, since we also support outgoing guards, in addition to incoming ones. Outgoing edge guards can be used to impose extra (waiting) conditions in order for some action node to be performed, in addition to the action guard.

3.2 Dynamics

We now define the operational semantics of concrete activities, by translating concrete activities to core activities, and using the reduction rules defined earlier. The advantage is that the semantics of the various concrete activity nodes can be defined by means of just two reduction rules: And and Or.

Node translation. First we translate activity nodes. We do this by defining a translation function $[\![\, \cdot \,]\!]_{\mathsf{n}} : \mathsf{ConcrNode} \to \mathsf{CoreNode}$ that translates concrete nodes to core nodes. To be able to define this translation function, we assume that the set Label of action labels is chosen in such a way to contain 'reserved labels' for all concrete node types, so that $\mathsf{ConcrNodeType} \subseteq \mathsf{Label}$.

Let $[\![\, \cdot \,]\!]_{\mathsf{n}}$ be defined as follows, where $\lambda\sigma.\mathsf{true}$ is the constant guard that is always true, and $\lambda\sigma.\sigma$ is the identity effects function that does not change the state:

$$[\![(\ell, \mathsf{init})]\!]_{\mathsf{n}} = (\ell, \mathsf{or}\langle(\mathsf{init}, \lambda\sigma.\mathsf{true}, \lambda\sigma.\sigma)\rangle)$$

$$[\![(\ell, \mathsf{final})]\!]_{\mathsf{n}} = (\ell, \mathsf{or}\langle(\mathsf{final}, \lambda\sigma.\mathsf{true}, \lambda\sigma.\sigma)\rangle)$$

$$[\![(\ell, \mathsf{fork})]\!]_{\mathsf{n}} = (\ell, \mathsf{and}\langle(\mathsf{fork}, \lambda\sigma.\mathsf{true}, \lambda\sigma.\sigma)\rangle)$$

$$[\![(\ell, \mathsf{join})]\!]_{\mathsf{n}} = (\ell, \mathsf{and}\langle(\mathsf{join}, \lambda\sigma.\mathsf{true}, \lambda\sigma.\sigma)\rangle)$$

$$[\![(\ell, \mathsf{decision})]\!]_{\mathsf{n}} = (\ell, \mathsf{or}\langle(\mathsf{decision}, \lambda\sigma.\mathsf{true}, \lambda\sigma.\sigma)\rangle)$$

$$[\![(\ell, \mathsf{merge})]\!]_{\mathsf{n}} = (\ell, \mathsf{or}\langle(\mathsf{merge}, \lambda\sigma.\mathsf{true}, \lambda\sigma.\sigma)\rangle)$$

$$[\![(\ell, \mathsf{act}(\alpha))]\!]_{\mathsf{n}} = (\ell, \mathsf{and}\langle\alpha\rangle)$$

This translation function maps concrete nodes to either and-typed or or-typed core nodes, where the type determines the execution semantics.

Initial nodes (init) are translated as or-type nodes, since they typically do not have incoming edges, and in case they have multiple outgoing edges, one of them is followed. In fUML semantics, when firing an initial node, a single token is created and offered to all outgoing edges, and exactly one of those edges then gets this token ([1], page 221). Final nodes (final) are translated likewise to initial nodes: they typically do not have outgoing edges, and in case they have multiple incoming edges, it is sufficient that one of them is enabled. In fUML semantics, when firing a final node, all incoming tokens are consumed ([1], page 218). Note that executing a final node does not mean that all edge tokens in an activity will disappear. For example, it may be possible that a final node is being executed while there are still tokens at other (forked) parts of the activity. In such cases, the execution of the final node will not mean that these other tokens dissapear. In other words, the execution of a final node does not necessarily mean that the activity is done executing. In case there are still tokens left, execution may resume.

Fork nodes (fork) are translated to and-typed nodes: their execution requires all incoming edges (usually just one) to be enabled, and puts a token on all outgoing edges (usually more than one). In fUML semantics, when firing a fork node, tokens are consumed from all incoming edges and offered to all outgoing edges ([1], page 218–220). Join nodes (join) are executed likewise to fork nodes: their executions requires all incoming edges (usually multiple) to be enabled and puts a token on all outgoing edges (usually just one). In fUML semantics, join nodes can only fire when all incoming edges have tokens ([1], page 222).

Decision nodes ($\mathsf{decision}$) are translated as or-typed nodes, as their execution requires a single incoming edge (usually just one) to be enabled and non-deterministically chooses a single outgoing edge whose incoming guard holds to receive a token. In fUML semantics, when firing a decision node, a token is offered to all outgoing edges whose (incoming) guard holds[2] ([1], page 213). Merge nodes (merge) are translated similarly: their execution requires a single incoming edge (usually multiple) to be enabled, and puts a token on one outgoing edge (usually just one). The description of the semantics of merge nodes in [1] is not super clear.

Finally, action nodes are translated as and-typed nodes. This is consistent with fUML semantics which describes that, once action execution has completed, "a control token is offered on all control flows outgoing from the action" ([1], page 267). Moreover, in case of fUML action nodes "the semantics of the offering of a token on control flows outgoing from an action are those of an implicit fork" ([1], page 267).

Appendix A of [2] shows how more complex behavior of UML activities can be defined in terms of and and or nodes, covering also other node types like 'send' and 'accept' nodes, which are out of scope for the purpose of this document.

Edge translation. The translation of edges is straightforward, and simply amounts to translating the source and target node. Let $[\![\cdot]\!]_{\mathsf{e}} : \mathsf{ConcrNode} \times \mathsf{Guard} \times \mathsf{Guard} \times \mathsf{ConcrNode} \to \mathsf{CoreNode} \times \mathsf{Guard} \times \mathsf{Guard} \times \mathsf{CoreNode}$ be the translation function for edges, defined such that

$$[\![(n, g_{in}, g_{out}, n')]\!]_{\mathsf{e}} = ([\![n]\!]_{\mathsf{n}}, g_{in}, g_{out}, [\![n']\!]_{\mathsf{n}}).$$

Activity semantics. Now that we can translate the nodes and edges of concrete activities, let us combine these translations. Let the translation of any con-

[2]The actual description as given in [1] is slightly more complex, since it involves decision input flows that supply extra values to decision nodes (e.g., object flows), which somehow are used as tokens. However, this is the gist of it.

crete activity $A = (N, E)$ be defined as $[\![A]\!] = (\{[\![n]\!]_{\mathsf{n}} \mid n \in N\}, \{[\![e]\!]_{\mathsf{e}} \mid e \in E\})$. Then the semantics of A is defined to be the semantics of $[\![A]\!]$.

Initial configuration. Obviously, any execution of an activity requires some starting point. In other words, since activity execution describes how to go from one configuration to another, there has to be a notion of an initial configuration.

The starting point of executing concrete activities are their init nodes. Since initial nodes are translated to or-type nodes, the execution of an activity is started along exactly one outgoing edge of a (single) init node. In case an activity has multiple initial nodes, or in case there are initial nodes with multiple outgoing edges, the activity has multiple potential initial configurations. In esoteric cases, an activity might have no initial configurations in case there are no initial nodes, or in case there are only initial nodes without outgoing edges.

Let $A = (N, E)$ be an activity. Then $(\{\varepsilon\}, \sigma_I) \in \mathsf{Config}$ is defined to be an *initial configuration of* A for any choice of initial state $\sigma_I \in \mathsf{State}$, if there exists an initial node $(\ell, \mathsf{init}) \in N$ such that $\varepsilon \in \mathsf{out}([\![(\ell, \mathsf{init})]\!]_{\mathsf{n}}, \sigma_I)$. In other words, any outgoing edge of any initial node can form an initial configuration together with some initial state σ_I, given that this edge is allowed by its guard to receive an (initial) token in the initial state σ_I. This initial state σ_I could for example be chosen to be the initial valuation of all variables defined in the context of A. Recall that the current definitions abstract over details like variables and their values[3]. The formalization could later be instantiated with such details.

3.3 Extensions

Let us now discuss several extensions to the activities as defined so far.

Hierarchy. We may want to use activities in a *hierarchical* manner, to allow larger activity specifications to be composed out of smaller ones. With hierarchy we then mean allowing activities to call other activities by means of *call actions*.

To support call actions, we first have to take into account that activities are defined and executed in a broader context, namely together with a number of contextual activities, $A_0, \ldots, A_n$, that may be called. Then we would have to extend activities by an additional type, $\mathsf{ConcrNodeType} ::= \cdots \mid \mathsf{call}(\imath)$, where $\imath \in \{0, \ldots, n\}$ is a reference to one of those contextual activities to call.

We disallow (in)direct recursion of action calls, i.e., we disallow the underlying call graph to contain cycles, so that hierarchy is maintained. By doing so, the extended activities can be *flattened* into single activities without any call actions by inlining any activity being called. The semantics of these extended hierarhical activities is then defined to be the semantics of the flattened activity.

Nonatomic actions. The execution semantics of activities is defined as a single-step reduction relation, meaning that actions are executed atomically.

[3]We are not going to define a custom expression/effects language and its semantics. There are plenty of good existing languages that could be picked off-the-shelf, like for example CIF.

Nevertheless, in reality, actions may not always be atomic. For example, an action could be the movement of a robot. The implication of considering this to be an atomic action, is that nothing can happen while the robot is moving. Hence, considering actions to always be atomic may sometimes be too strict.

We can support *nonatomic actions* by splitting them into two actions which are atomic: one for starting the nonatomic action and one for ending it. Elaborating further on that, let $\alpha \in \mathsf{Action}$ be an action that we want to execute nonatomically, so that $\alpha = (a, g, u)$. Then we can 'split' α and a into two actions: $\alpha_{start} = (a_{start}, g, \lambda\sigma.\sigma)$ and $\alpha_{end} = (a_{end}, \lambda\sigma.\mathsf{true}, u)$. Finally, any activity node for executing α can be replaced by a node that executes α_{start}, followed by a node that executes α_{end} with an edge between them. Alternatively, one might also exploit hierarchy (as described above) by modeling a nonatomic action as a separate activity that starts by doing α_{start}, and then ends by doing α_{end}. If needed, it is possible to extend $\mathsf{ConcrNodeType}$ by an additional type for nonatomic action nodes, like done for call actions as described above. In that case, one could eliminate such extensions like done for call actions.

3.4 Similarities and differences with fUML

Below is a list of similarities and differences with respect to fUML semantics:

- Nodes of type $\mathsf{act}(\alpha)$ roughly correspond to opaque actions in UML. However, opaque actions in UML do not have guards nor effects. This is something we added to α in our definition of activities.

- In fUML, every edge can only have a single guard, which only has actual meaning in case the edge goes out of a decision node. In that case, the edge guard is considered to be the incoming guard of that edge. Our definition of activities supports two types of edge guards: incoming and outgoing ones. Moreover, with our definition, every edge guard has a meaning, as opposed to just the ones that are related to decision nodes.

- Our definition of activities imposes a *waiting semantics*, meaning that execution of some action α 'waits' until $\mathsf{guard}(\alpha)$ holds, and likewise for incoming and outgoing edge guards. In fUML, action nodes are not guarded and do not have a waiting semantics. Instead of waiting, in fUML, the activity typically terminates whenever a state is encountered in which no immediate progress can be made, e.g., in case of decision nodes where none of the outgoing edges can be taken.

- Nodes of type $\mathsf{act}(\alpha)$ execute *atomically.* In particular, this means that $\mathsf{effects}(\alpha)$ is executed without interference from effects of other nodes. In fUML atomicity is not guaranteed. Instead, fUML semantics only guarantees that nodes in-between a fork and a join are executed after forking and before joining.

- In fUML the edges out of decision nodes have a declaration order, and the first enabled edge in the declaration order is taken[4]. Moreover, if none of the edges can be taken, the activity terminates (or rather crashes). Instead, in our version of activities, a nondeterministic choice is made among all enabled edges out of decision nodes. In case none of the edges are enabled, a waiting semantics is imposed, as described earlier.

- The execution of final nodes as defined in this document is different from fUML. As explained in Section 3.2, final nodes are translated to or-type activity nodes, which means that only a single incoming edge should be enabled for the node to fire, and when this happens, only a single token will be consumed. In fUML however, all tokens on enabled incoming edges will be consumed. The implication is that, in our semantics, final nodes may have to fire multiple times in case multiple incoming edges are enabled, to consume all their tokens. One might consider whether it is not better to translate final nodes to and-type nodes instead. However, such a translation would mean that all incoming edges into final must be enabled, which is in a way even more different from fUML semantics.

References

[1] Semantics of a Foundational Subset for Executable UML Models (fUML), v1.1. 2012. https://www.omg.org/spec/FUML/1.1/Beta1/PDF.

[2] Zamira Daw and Rance Cleaveland. Comparing model checkers for timed UML activity diagrams. *Science of Computer Programming*, 111:277–299, 2015. Special Issue on Automated Verification of Critical Systems (AVoCS 2013).

[4]We could not directly find this clearly stated in [1]. However, this is how the Cameo simulator handles decision nodes, and it claims to implement fUML semantics. Therefore, we assume that this statement is true in fUML.

Appendix A: List of Symbols

action	projection function for getting the action of an activity node
$\mathcal{A}$	A core activity
A	A concrete (UML-like) activity
a	An action label
α	An action, consisting of an action label, a guard and an effects function
$\mathsf{and}\langle\alpha\rangle$	A core node type describing an and-style execution of α
call	A call action node (extension)
Config	The set of all configurations
$\mathsf{ConcrNode}$	The set of all concrete activity nodes
$\mathsf{ConcrNodeType}$	The set of all concrete activity node types
c	A configuration, consisting of a set of edges holding a token, and a state
$\mathsf{CoreNode}$	The set of all core activity nodes
$\mathsf{CoreNodeType}$	The set of all core activity node types
$[\![\ \cdot\ ]\!]_{\mathsf{e}}$	Operation for translating concrete activity edges to core ones
$[\![\ \cdot\ ]\!]_{\mathsf{n}}$	Operation for translating concrete activity nodes to core ones
$\mathsf{decision}$	The decision node type of concrete activities
$\mathsf{Effects}$	The set of all effects functions
$\mathsf{effects}$	Projection function for obtaining the effects of an action
$\mathcal{E}$	A set of edges of a core activity
E	A set of concrete edges of a concrete (UML-like) activity
ε	An edge of a core activity
e	A concrete edge of a concrete activity
final	The final node type of concrete activities
fork	The fork node type of concrete activities
Guard	The set of all guards
guard	Projection function for obtaining the guard of an action
g	A guard, which is a state predicate
g_{in}	The incoming guard of an edge
g_{out}	The outgoing guard of an edge
ℓ	A node identifier, used to give identity to core and concrete activity nodes
in	Function for getting all incoming edges of a node
$\mathsf{guard}_{\mathsf{in}}$	Projection function for obtaining the incoming guard of an edge
init	The initial node type of concrete activities
Label	The set of all action labels
label	Projection function for obtaining the action label of an action
join	The join node type of concrete activities
merge	The merge node type of concrete activities
NodeID	The set of all node identifiers
$\mathcal{N}$	A set of nodes of a core activity

N	A set of concrete nodes of a concrete (UML-like) activity
η	A node of a core activity
n	A concrete node of a concrete activity
$\mathsf{or}\langle\alpha\rangle$	A core node type describing an or-style execution of α
π	A trace, i.e., a sequence of actions
Σ	A set of core edges holding a token, as part of a configuration
σ	An (execution) state, e.g., the valuation of some set of variables
source	Projection function for getting the source node of an edge
State	The set of all states
target	Projection function for getting the target node of an edge
out	Function for getting all outgoing edges of a node
$\mathsf{guard}_{\mathsf{out}}$	Projection function for obtaining the outgoing guard of an edge
u	An effects function, mapping a state to zero or more successor states
$\rightarrow$	Reduction relation defining the semantics of core activities
$\xrightarrow{\alpha}$	Reduction relation for a core node with action α
$\xrightarrow{\eta}$	Reduction relation for a core node η
$\xrightarrow{\pi}_*$	Multi-step reduction relation for the finite action sequence π
$\xrightarrow{\pi}_\omega$	Multi-step reduction relation for the infinite action sequence π