Concise Semantics of Poka Yoke Activities

January 11, 2024

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, actions are executed atomically and have guards and updates that specify how action execution influences state.

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

The reason for such a translation is two-fold. Firstly, even though 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 fall into two groups, here called and-type and or-type nodes.

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 abstract 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 Abstract Activities

This section defines the (static) structure and (dynamic) behavior of abstract 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 abstract 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 (non-deterministic) updates 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*. Let State be the set of all states and $\sigma \in \mathsf{State}$ be a typical state. State is an abstract notion, which could for example be instantiated to be the valuation of some set of variables.

Guards and updates. Guards can be thought of as predicates over states, while updates are functions that map states to any number of (i.e., zero or more) 'successor states', to be able to model non-deterministic action behavior.

Let $\mathsf{Guard} = \mathsf{State} \to \mathbb{B}$ be the set of all *guards* and $\mathsf{Update} = \mathsf{State} \to 2^{\mathsf{State}}$ be the set of all *updates* over states (with 2^{State} the powerset of State). We use $g \in \mathsf{Guard}$ and $u \in \mathsf{Update}$ to denote a typical guard and update, 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 update 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 update (i.e., the update 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 non-deterministically. (Further execution semantics details are given in Section 2.2.)

More formally, let $\mathsf{Action} = \mathsf{Label} \times \mathsf{Guard} \times \mathsf{Update}$ 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 update u.

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

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

Abstract nodes. Now that actions have been defined, we can define the *nodes* of abstract activities, which carry out these actions during executions of the activity. Abstract activity nodes consist of three parts: a node identifier, a node type, and 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 abstract 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 $\mathsf{AbstrNode}$ of all possible *abstract activity nodes* is defined as follows:

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

Abstract 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 $\eta \in \mathsf{AbstrNode}$ to denote a typical node. Let $\mathsf{action} : \mathsf{AbstrNode} \to \mathsf{Action}$ be a projection function for obtaining the actions of abstract nodes, so that:

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

Abstract activities. Let us now define *abstract activities*, which are essentially graphs consisting of abstract nodes, as defined in the previous paragraph, that are connected via guarded edges. The edge guards are (later) used to restrict when an edge can receive a token. During activity execution, any edge can only receive a token in a state where the edge guard holds. This is 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.

More formally, abstract activities $\mathcal{A} = (\mathcal{N}, \mathcal{E})$ are graphs with nodes $\mathcal{N} \subseteq \mathsf{AbstrNode}$ and edges $\mathcal{E} \subseteq \mathcal{N} \times \mathsf{Guard} \times \mathcal{N}$. Any edge $\varepsilon = (\eta, g, \eta') \in \mathcal{E}$ consists of a *source node* $\eta \in \mathcal{N}$, a *target node* $\eta' \in \mathcal{N}$ and a *guard* $g \in \mathsf{Guard}$. Let $\mathsf{source} : \mathcal{E} \to \mathcal{N}$, $\mathsf{target} : \mathcal{E} \to \mathcal{N}$ and $\mathsf{guard} : \mathcal{E} \to \mathsf{Guard}$ be projection functions to obtain the source, target and guard of any edge, respectively. (The definitions of these projection functions are likewise to the definitions of earlier defined projection functions, and are therefore omitted here.)

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.
- $\mathsf{in}(\eta, \sigma) = \{\varepsilon \in \mathsf{in}(\eta) \mid \mathsf{guard}(\varepsilon)(\sigma)\}$ be all incoming edges of η whose guard holds with respect to σ.

- $\mathsf{out}(\eta, \sigma) = \{\varepsilon \in \mathsf{out}(\eta) \mid \mathsf{guard}(\varepsilon)(\sigma)\}$ be all outgoing edges of η whose guard holds with respect to σ.

2.2 Dynamics

This section defines the dynamic behavior of abstract 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 containing a token together with a (current) state. There are two reduction rules, for and-style and or-style action execution, as explained earlier. Finally, we introduce initial configurations and execution traces.

Configurations. The semantics of abstract 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 have 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 state. We write $c \in \mathsf{Config}$ to denote a typical configuration, such that $c = (\Sigma, \sigma)$.

When some edge has a token with respect to some configuration c, we sometimes say it is *enabled in* c, and when c is clear from the context, we may just say it is *enabled*. Note that an edge can be enabled, i.e., hold a token, even when its guards does not hold with respect to the current state. This is because an edge guard only restricts when the edge can receive a token (rather than when an edge can hold a token).

Reductions. Let us now define the operational semantics of abstract 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 already, these rules only differ in their requirement on, and distribution of, edge tokens.

More formally, the operational semantics of abstract 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:

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

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

The AND rule defines the execution of and-type nodes η, for which $\mathsf{action}(\eta) = \mathsf{and}\langle\alpha\rangle$. AND has five premises. Firstly, it requires all incoming edges into η to have a token. Secondly, it requires none of the outgoing edges $\mathsf{out}(\eta)$ to have a token (otherwise action execution might attempt to put more than one token on a single edge, which is not allowed since edges have at most one token). Thirdly, AND requires that the guards of all outgoing edges hold with respect to the successor state σ'. Recall that edge guards are conditions for edges to receive tokens, and since AND aims to put a token on all outgoing edges of η, all their guards have to hold. Fourthly, it requires the action guard $\mathsf{guard}(\alpha)$ to hold with respect to the current state σ. And finally, it requires a successor state σ' to be available from the update $\mathsf{update}(\alpha)$ of α. If all five conditions are met, AND removes all tokens from $\mathsf{in}(\eta)$ and puts tokens on $\mathsf{out}(\eta)$, so that the arrangement of edge tokens becomes $(\Sigma \setminus \mathsf{in}(\eta)) \cup \mathsf{out}(\eta)$.

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 which are much like the premises of AND, except that they concern single incoming/outgoing edges of η, rather than all of them. Firstly, OR requires the existance of a single incoming edge ε of η that holds a token. Secondly, a single outgoing edge ε' of η must exist that does not yet have a token (recall that edges cannot have more than one token) and is allowed to receive a token (i.e., its guard must hold with respect to the successor state). The third and fourth premise are the same as the last two premises of AND. If all four conditions are met, OR removes the token from ε and puts it on ε', so that the arrangement of tokens becomes $(\Sigma \setminus \{\varepsilon\}) \cup \{\varepsilon'\}$.

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

3 Concrete Activities

This section defines concrete activities (Section 3.1) that more closely resemble UML activities than abstract activities, and defines their operational semantics in terms of translations to abstract activities (Section 3.2). Some similarities and differences of concrete activities with respect to fUML semantics are highlighted (Section 3.3). Finally, known open issues are briefly discussed (Section 3.4).

3.1 Statics

First we define concrete activities, which more closely resemble UML activities than the abstract activities defined earlier.

Nodes. In contrast to abstract 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. Let us now define a more concrete set of activity nodes that more closely resemble (f)UML activity nodes. While doing so, let us focus on a basic subset of UML, leaving out more advanced concepts like signals, object flows, exception handlers, interruptable 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 $n \in \mathsf{ConcrNode}$. The node types are the typical types of UML activity nodes: initial and flow final nodes (init and final), fork and join nodes (fork and join), decision and merge nodes ($\mathsf{decision}$ and merge), and action nodes (act) which are closely related to opaque actions in UML.

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

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 all edges have guards, rather than just the ones coming out of decision nodes. This is consistent with UML activities (except that, in UML, guards are ignored unless the edge comes out of a decision node).

3.2 Dynamics

We now define the operational semantics of concrete activities, by translating concrete activities to abstract 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 $[\![\, \cdot \,]\!]_{\mathsf{n}} : \mathsf{ConcrNode} \to \mathsf{AbstrNode}$ that translates concrete nodes to abstract 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 update 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 abstract 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).

Flow 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 flow final node, all incoming tokens are consumed ([1], page 218). Note that executing a flow final node does not mean that all edge tokens in an activity will disappear. 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 (decision) are translated as or-typed nodes, as their execution requires a single incoming edge (usually there is only one) to be enabled and non-deterministically chooses a single outgoing edge whose guard holds to receive a token. In fUML semantics, when firing a decision node, a token is offered to all outgoing edges whose guard holds[2] ([1], page 213).

[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.

Merge nodes (merge) are translated in a similar manner: their execution requires a single incoming edge (usually there are multiple) to be enabled, and puts a token on one outgoing edge (usually there is 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).

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

$$[\![(n, g, n')]\!]_{\mathsf{e}} = ([\![n]\!]_{\mathsf{n}}, g, [\![n']\!]_{\mathsf{n}})$$

Activity translation. Now that we can translate the nodes and edges of abstract activities, let us combine these translations to translate activities.

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

Appendix A of [2] additionally shows how more complex behavior of UML activities can be defined in terms of and and or nodes, covering also other types of nodes, like 'send' and 'accept' nodes.

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 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 concrete values[3]. The formalization could later be instantiated with such details.

3.3 Similarities and differerences 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. This is something we added to α in this definition of activities.

- This definition of activities imposes a *waiting semantics*, meaning that execution of some action α 'waits' until $\mathsf{guard}(\alpha)$ holds, and likewise for edge guards. In fUML, action nodes are not guarded and do not have a waiting semantics. Instead of waiting, in fUML, typically the activity terminates whenever a condition 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 $\mathsf{update}(\alpha)$ is executed without interference from updates 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 non-deterministic 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.

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

[4] I 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, I assume that this statement is true in fUML.

3.4 Open issues

Below is a list of open issues for which a solution is still needed:

- In the semantics of Poka Yoke activities as presented in this document, actions are atomic, which may be problematic. The consequence of actions being atomic is that nothing else can occur in parallel. In practice, an action could be a movement of a robot, which might take a long time. In that time, other things could happen. Therefore, actions being strictly atomic will pose issues in practice. Especially since implementing a code generator for this formalization as we have it now means that everything becomes single threaded, which will not fly.

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).

Appendix A: List of Symbols

$\mathcal{A}$	An abstract activity
A	A concrete (UML-like) activity
a	An action label
α	An action, consisting of an action label, a guard and an update
$\mathsf{and}\langle\alpha\rangle$	An abstract node type describing an and-style execution of α
c	A configuration, consisting of a set of edges holding a token, and a state
$[\![\cdot]\!]_{\mathsf{e}}$	Operation for translating concrete activity edges to abstract ones
$[\![\cdot]\!]_{\mathsf{n}}$	Operation for translating concrete activity nodes to abstract ones
$\mathsf{decision}$	The decision node type of concrete activities
$\mathcal{E}$	A set of abstract edges of an abstract activity
E	A set of concrete edges of a concrete (UML-like) activity
ε	An abstract edge of an abstract 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
g	A guard, which is a state predicate
ℓ	A node identifier, used to give identity to abstract and concrete activity nodes
init	The initial node type of concrete activities
join	The join node type of concrete activities
merge	The merge node type of concrete activities
$\mathcal{N}$	A set of abstract nodes of an abstract activity
N	A set of concrete nodes of a concrete (UML-like) activity
η	An abstract node of an abstract activity
n	A concrete node of a concrete activity
$\mathsf{or}\langle\alpha\rangle$	An abstract node type describing an or-style execution of α
π	A trace, i.e., a sequence of actions
Σ	A set of abstract edges holding a token, as part of a configuration
σ	An (execution) state, e.g., the valuation of some set of variables
u	An update, mapping a state to zero or more successor states
$\rightarrow$	Reduction relation defining the semantics of abstract activities
$\xrightarrow{\alpha}$	Reduction relation for an abstract node with action α
$\xrightarrow{\eta}$	Reduction relation for an abstract 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