

# Simplification Rules for Poka Yoke Activities

April 24, 2025

## 1 Introduction

This document defines simplification rules for (simple UML-like) activities. These activities and their semantics are slightly different from other UML semantics, e.g. fUML, activities. For example, every action has an associated guard and effects that specify how action execution influences the system state. Edges have two guards, named incoming and outgoing guard, that define when a token can be placed on an edge, and when it can be removed, respectively. This document aims at finding patterns of action, nodes, edges that can be transformed into simpler patterns, while maintaining the same operational semantic of the activity. We call these transformations *simplifications*.

## 2 Preliminaries and Notation

We largely refer to the Operation Semantics of Poka Yoke Activities document for the static structure and dynamics behaviors of activities. We recall here only the elements that are useful for this document.

### 2.1 Statics

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

**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 our semantics document 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*.

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

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

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

$$\text{action}((\ell, \text{and}\langle\alpha\rangle)) = \alpha \quad \text{action}((\ell, \text{or}\langle\alpha\rangle)) = \alpha$$

## 2.2 Dynamics

This section defines the dynamic behavior of core activities, by means of a token passing style operational semantics. Its 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 token-passing rules: one for **and**-style and one for **or**-style action execution.

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

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

## 2.3 Additional Notation

A core activity  $\mathcal{A} = (\mathcal{N}, \mathcal{E})$  is fundamentally a graph with nodes  $\mathcal{N} \subseteq \mathbf{CoreNode}$  and edges  $\mathcal{E} \subseteq \mathcal{N} \times \mathbf{Guard} \times \mathbf{Guard} \times \mathcal{N}$ . Let us define  $F$  as the *flow relation* of  $\mathcal{A}$ , defined as  $F \subseteq (\mathcal{N} \times \mathcal{E}) \cup (\mathcal{E} \times \mathcal{N})$ . We use the following symbols and notation for the set of predecessors and successors of a node  $n \in \mathcal{N}$  and edge  $e \in \mathcal{E}$ :

- $\bullet e = \{n \mid (n, e) \in F\}$  = the set of input nodes of  $e$
- $e \bullet = \{n \mid (e, n) \in F\}$  = the set of output nodes of  $e$
- $\bullet n = \{e \mid (e, n) \in F\}$  = the set of input edges of  $n$
- $n \bullet = \{e \mid (n, e) \in F\}$  = the set of output edges of  $n$

With slight abuse of notation, we extend this notation to sets of nodes and edges.

A core activity  $\mathcal{A}$  is *safe* if all configurations reachable by legal sequences of edge firings from the initial configuration have either zero or one tokens in every edge. We assume that all considered core activities are safe.

For every core activity  $\mathcal{A}$ , the language of  $\mathcal{A}$ , denoted  $L(\mathcal{A})$ , is the set of all legal sequences  $\omega \in T^*$  of edges firings starting from the initial configuration of tokens  $c_0$ . Let  $X$  be a set of nodes of core activity  $\mathcal{A}$ . We use  $\mathcal{A}|X$  for the core activity obtained by restricting the underlying graph to  $X$ , and  $\mathcal{A} - X$  to denote  $\mathcal{A}|\bar{X}$ , where  $\bar{X}$  denotes the complement of  $X$ . This notation is also used for core activity states and firing schedules. For example,  $\mathcal{A} - \{e_1, e_2\}$  refers to what is left of activity  $\mathcal{A}$  once one removes the edges  $e_1$  and  $e_2$ . We draw inspiration from [4] for the following definition of equivalence of core activities.

**Equivalence of Core Activities** The goal of this manuscript is to take a core activity  $\mathcal{A}$  and to reduce it to a smaller activity  $\mathcal{A}'$  that is more amenable to analysis. Of course, we want the activity  $\mathcal{A}'$  to be in some sense the “same” as the original activity  $\mathcal{A}$ . We will focus on language preservation (among unmodified edges).

**Definition 2.1.** *Fix some rule for transforming a core activity  $\mathcal{A}$  to a core activity  $\mathcal{A}'$ . Let  $U$  be the set of edges of  $\mathcal{A}$  which are left completely unmodified by this transformation. We say that the (simplified) core activity  $\mathcal{A}'$  is equivalent to activity  $\mathcal{A}$  after the transformation if*

$$L(\mathcal{A})|U = L(\mathcal{A}')|U.$$

### 3 Simplifications

#### 3.1 Serial Fusion

**Simplification Rule 3.1.** *If there exist two edges  $e_1$  with  $\text{guard}_{in}(e_1) = g_1$ ,  $\text{guard}_{out}(e_1) = g_2$ , and  $e_2$  with  $\text{guard}_{in}(e_2) = g_3$ ,  $\text{guard}_{out}(e_2) = g_4$ , and a node  $n$  satisfying:*

1. *The initial configuration  $c_0$  does not place any token on  $e_2$*
2.  *$e_1 \bullet = \{n\} = \bullet e_2$ , i.e. node  $n$  is the unique output node of  $e_1$  and the unique input node of  $e_2$*
3. *Node  $n$  is disconnected from all other edges, and has no effects*
4.  *$(\bullet e_1) \bullet = \{e_1\}$ , i.e. edge  $e_1$  does not share any of its input place with any other edges*

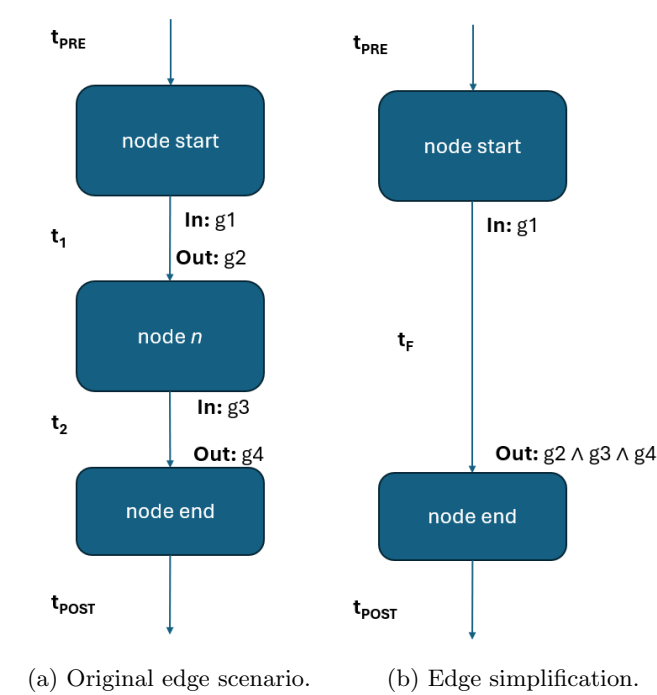


Figure 1: The series edge merging scenario. [Pictures to be improved.](#)

then edges  $e_1$  and  $e_2$  can be replaced by a single edge  $e_F$  with input set  $\bullet e_F = \bullet e_1$ , with output set  $e_F \bullet = e_2 \bullet$  and guards  $\text{guard}_{in}(e_F) = g_1$ ,  $\text{guard}_{out}(e_F) = g_2 \wedge g_3 \wedge g_4$ .

*Proof.* We must show that  $L(\mathcal{A}) - \{e_1, e_2\} \subseteq L(\mathcal{A}') - \{e_F\}$  and that  $L(\mathcal{A}) - \{e_1, e_2\} \supseteq L(\mathcal{A}') - \{e_F\}$ . For simplicity, let us assume that node  $n$  does not have a guard. In presence of a guard, it is sufficient to conjunct it with either the outgoing guard  $g_2$  or the incoming guard  $g_3$ .

Let us first show that  $L(\mathcal{A}) - \{e_1, e_2\} \subseteq L(\mathcal{A}') - \{e_F\}$ ; we do so by induction on the number of firings of  $e_2$ . Note that we are interested in *non-blocking* sequences, so we should always be able to take edge  $e_2$  and execute its outgoing node (which might be a final node or not). First, note that  $\mathcal{A} - \{e_1, e_2\} = \mathcal{A}' - \{e_F\}$ . For the base case, let  $\varsigma$  be a schedule that contains no firing of  $e_2$ . If also  $e_1 \notin \varsigma$ , then  $\varsigma \in L(\mathcal{A} - \{e_1, e_2, n\})$ , and we can use  $\varsigma' = \varsigma$  in  $\mathcal{A}'$  to satisfy our claim. If instead  $e_1 \in \varsigma$ , note that there must be exactly one firing of  $e_1$  in  $\varsigma$  since  $\mathcal{A}$  is safe. Let  $\varsigma = \varsigma_1(e_1)\varsigma_2$  where neither  $e_1$  or  $e_2$  appear in  $\varsigma_1, \varsigma_2$ . By definition,  $g_1$  must be true at some

point of  $\varsigma_1$  in order to place a token in  $e_1$ . Similarly, both guards  $g_2$  and  $g_3$  must be true at the end of  $\varsigma_1$  for the firing of  $e_1$ . Note that  $e_1$  cannot be enabled at the end of  $\varsigma_2$  – since  $e_2$  has not fired yet – because the activity is safe and node  $n$  has no other outgoing edges. Then, we can use  $\varsigma' = \varsigma_1(\cdot)\varsigma_2$  as a schedule for  $\mathcal{A}'$  – note that we can put a token on  $e_F$  if and only if  $g_1$  holds at some point, but we cannot remove it until also  $g_4$  holds – and our claim is satisfied.

Let us now assume that the claim is true until  $k$  firings of  $e_2$ , and consider a schedule  $\varsigma$  of  $\mathcal{A}$  where there are  $k+1$  firings of  $e_2$ . Since  $e_1 \bullet = \{n\} = \bullet e_2$ , activity  $\mathcal{A}$  is safe, and  $e_2$  is not in the initial configuration,  $e_1$  must fire exactly once before each firing of  $e_2$ . Hence,  $\varsigma$  contains either  $k+1$  or  $k+2$  firings of  $e_1$ . Let consider a schedule with  $k+1$  firings of  $e_1$ : we can write

$$\varsigma = \varsigma_1(e_1)v_1(e_2)\varsigma_2(e_1)v_2 \dots \varsigma_{k+1}(e_1)v_{k+1}(e_2)\varsigma_{k+2},$$

where neither  $e_1$  or  $e_2$  appears in  $\varsigma_i$  nor  $v_i$ . By inductive hypothesis, we can set  $\varsigma' = \varsigma_1(e_1)v_1(e_2)\varsigma_2(e_1)v_2 \dots \varsigma_{k+1}$  such that the two languages are equivalent (until step  $k$ ). If edge  $e_1$  does not share any input nodes, then then final firing in  $\varsigma$  must happen when  $g_2 \wedge g_3$  holds. Let us now analyze  $v_{k+1}$ : if during this sequence of firings the values of  $g_2, g_3, g_4$ , do not change – and in particular must all be true, since we have a firing of  $e_2$  afterwards – we can simply adapt the final part of the schedule as  $\varsigma_{k+1}(e_F)v_{k+1}\varsigma_{k+2}$  or  $\varsigma_{k+1}v_{k+1}(e_F)\varsigma_{k+2}$  without loss of generality, and again our claim is satisfied. In general however, sequence  $v_{k+1}$  can change the truth values of  $g_2$  or  $g_3$  (Note that at the end of  $v_{k+1}$  guard  $g_4$  must hold, since we have a firing of  $e_2$ ). If that happens, we have a sequence in  $\mathcal{A}$  that cannot be reproduced in  $\mathcal{A}'$ : to prevent this, we impose that if  $g_4$  is true, then also  $g_2 \wedge g_3$  must hold. In this way, we ensure that at the end of  $v_{k+1}$  all guards  $g_2, g_3, g_4$  must hold, and we can use the adapted version of  $\varsigma$  (see previous discussion) in  $\mathcal{A}'$ , and this concludes the proof.

Let us now show that  $L(\mathcal{A}) - \{e_1, e_2\} \supseteq L(\mathcal{A}') - \{e_F\}$ . Similarly to the previous case, we consider schedules  $\varsigma'$  in  $\mathcal{A}'$  and prove that there exists a schedule  $\varsigma$  in  $\mathcal{A}$  that ensure language equivalence. We proceed by induction in the number of steps of  $e_F$ . If there are no firings of  $e_F$ , the proof follows analogously to the previous case scenario. Let us now take the inductive hypothesis, and consider a schedule

$$\varsigma' = \varsigma'_1(e_F)\varsigma'_2(e_F) \dots \varsigma'_{k+1}(e_F)\varsigma'_{k+2},$$

that contains  $k+1$  firings of  $e_F$ , and the  $\varsigma_i$  do not contain any firing of  $e_F$ . Note that  $\varsigma_{k+1}$  enables the firing of  $e_F$  by definition: this means that, by

inductive hypothesis, there exists a schedule  $\varsigma$  in  $\mathcal{A}$  where edge  $e_1$  is enabled. The guards as well simply mirror this, as if  $g_2 \wedge g_3 \wedge g_4$  holds, so does  $g_2 \wedge g_3$  (the condition to enable  $e_1$ ). Thus, edge  $e_1$  can freely fire: we extend the schedule  $\varsigma$  for  $\mathcal{A}$  by appending the firing of  $e_1$  at the end of  $\varsigma$ . To that, we append also  $(e_2)\varsigma'_{k+2}$ : since  $\bullet_{e_F} = \bullet_{e_1}$ , if the last edge of  $\varsigma'_{k+1}$  enables  $e_F$ , then it also enables  $e_1$ .  $\square$

**Simplification Rule 3.2.** *If there exist two edges  $e_1$  with  $\text{guard}_{in}(e_1) = g_1$ ,  $\text{guard}_{out}(e_1) = g_2$ , and  $e_2$  with  $\text{guard}_{in}(e_2) = g_3$ ,  $\text{guard}_{out}(e_2) = g_4$ , and a node  $n$  satisfying:*

1. *The initial configuration  $c_0$  does not place any token on  $e_1$*
2.  *$e_1 \bullet = \{n\} = \bullet_{e_2}$ , i.e. node  $n$  is the unique output node of  $e_1$  and the unique input node of  $e_2$*
3. *Node  $n$  is disconnected from all other edges, and has no effects*
4.  *$(\bullet_{e_1}) \bullet = \{e_1\}$ , i.e. edge  $e_1$  does not share any of its input place with any other edges*

*then edges  $e_1$  and  $e_2$  can be replaced by a single edge  $e_F$  with input set  $\bullet_{e_F} = \bullet_{e_1}$ , with output set  $e_F \bullet = e_2 \bullet$  and guards  $\text{guard}_{in}(e_F) = g_1 \wedge g_2 \wedge g_3$ ,  $\text{guard}_{out}(e_F) = g_4$ .*

*Proof.* The proof \*should\* follow similarly, perhaps with induction on  $e_1$  for the first part. [#TODO](#)  $\square$

## References

- [1] Shatz, Sol M., et al. "An application of Petri net reduction for Ada tasking deadlock analysis." *IEEE Transactions on Parallel and Distributed Systems* 7.12 (1996): 1307-1322.
- [2] Ehrig, Hartmut, and Julia Padberg. "Graph grammars and Petri net transformations." *Advanced Course on Petri Nets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.
- [3] Evangelista, Sami, Serge Haddad, and J-F. Pradat-Peyre. "Syntactical colored petri nets reductions." *International Symposium on Automated Technology for Verification and Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.
- [4] Sloan, Robert H., and Ugo Buy. "Reduction rules for time Petri nets." *Acta Informatica* 33 (1996): 687-706.