An Approach to Semantics for UML Activities

Dariusz Gall(✉) and Anita Walkowiak

Wroclaw University of Science and Technology, Wybrzeże Wyspiańskiego 27,
50-370 Wrocław, Poland
{dariusz.gall,anita.walkowiak}@pwr.edu.pl

Abstract. The precise semantics for UML Activities is must-have in automated applications of them. The paper proposes an approach to a definition of a semantics of a set of UML Activities in the form of LTS (*Labelled Transition System*). The set of activities is transformed to a graph of possible execution traces, using OCL (*Object Constraint Language*) operations. So far, it covers basic elements of an activity, i.e. sequential control and object flows. Moreover, an Activity Semantics Metamodel is introduced. It specifies concepts, informally described in the UML, used for Activity semantics definition.

Keywords: UML Activity · OCL · LTS · Formal semantics · Reachability graph

1 Introduction

An activity is well-known behavior representation commonly called a "control and data flow" model [8, 12, 15]. There are intensively used for software requirements specification [3, p. 92, p. 477, 9, 10, 17, p. 105]. However, the main obstacle in using activity is lack of precise semantics. The definition of UML presented in the OMG standard [8] is semi-formal, i.e. the semantics of elements is expressed in natural language and has variation points. Additional challenges are the extensibility of the UML and (deliberate or accidental) under-specifications.

The lack of formal semantics brings ambiguity problems [5, 8, 11, 13, 16], especially in case of automation of system development process – i.e. transformation of PIM and PSM models and code generation in MDA approach; and design of tools supporting the process [3]. Furthermore, the possibility of UML model-checking is limited to syntax verification [1].

There are many works on the UML Activities semantics, e.g., [2, 3, 5, 12–16, 4]. These approaches specify the UML Activities semantics in some scope, omitting certain aspects of the standard and consider only specific usages of the UML Activities [2, 3, 5, 8] and/or are just outdated [4], since refer to previous versions of the UML [8]. Some of these works provide translational semantics, by expressing the UML Activities in other formalisms, e.g., Petri-Nets, Abstract State Machines. Nevertheless, it is challenging to formulate the Activity semantics in these formalism, especially when more UML Activities constructions are considered [14, 16]. Moreover, the authors investigate an execution of a single Activity and do not discuss an execution of a set of Activities.

J. Świątek et al. (eds.), *Information Systems Architecture and Technology: Proceedings of 38th International Conference on Information Systems Architecture and Technology – ISAT 2017*,
Advances in Intelligent Systems and Computing 656, DOI 10.1007/978-3-319-67229-8_23

The main goal of the paper is to give an approach to a precise interpretation of a set of activities, consistent with the UML standard. The interpretation is expressed in terms of all possible execution traces that may be derived from the set. We propose a transformation of a set of activities into the Labelled Transition System [6], which describes the modeled behavior in the form of a graph of possible execution traces.

The proposed semantics will support automation of system development process. It will check compliance with end-users expectations, e.g. by model simulations and model execution. Next, it will support model checking, especially by using existing tools, like Construction and Analysis of Distributed Processes tool (CADP). Moreover, it is a base for test-scenarios generation. And last but not least, it is necessary to build and validate model transformations between PIM and PSM models, and code generation.

In Sect. 2, we discuss the syntax and semantics of the UML Activities presented in the UML specification [8]. We define the LTS-based semantics of the UML Activities in Sect. 3. It gives definitions that are necessary to introduce the LTS for the UML Activities, and finally we introduce the algorithm to construct a graph of possible execution traces. The paper is concluded in Sect. 4.

2 Syntax and Semantics of Activities

The activity defined by UML specification [8] is a graph. It is represented by the Activity metaclass which consists of ActivityNodes connected by ActivityEdges, stored in the *node* and the *edge* metaattributes of the Activity. The ActivityEdge connects two ActivityNodes via the *source* and the *target* metaattributes, respectively. Subclasses of the ActivityNode are the ExecutableNode, the ObjectNode, and the ControlNode. There are two types of the ActivityEdge: the ControlFlow and the ObjectFlow.

An Activity is a kind of a Behavior. The Behavior describes a set of possible executions, or more precisely it is "*a specification of events that may occur dynamically over time*" [13, p. 284]. The UML specification defines an Event as "*a set of possible occurrences*", whereas an occurrence is "*something that happens that has some consequence with regard to the system*" [13, p. 12]. An invocation of a Behavior creates its instance, known as a behavior execution. Each behavior execution is related to a specific execution trace, which is "*actual sequence of event occurrences due to the invocation [of the Behavior], consistent with the specification of the Behavior*" [13, p. 284].

An Activity appoints a partial order of behavior steps that may be executed and data flows corresponding to the steps. A behavior step lasts non-zero time, is expressed by ExecutableNode and represents, for example, an arithmetic computation, a call to an operation, or manipulation of object contents. The partial order of behavior steps execution is specified in terms of tokens flow rules inspired by Petri Nets formalism [14] – "*The effect of one ActivityNode on another is specified by the flow of tokens over the ActivityEdges between ActivityNodes*" [13, p. 372], whereas it is assumed that the flow, if it occurs, is immediate. There are two kind of tokens: a control and an object token. A control token flow specifies an effect of execution one ActivityNode on

another – set an order of the execution, and flows over ControlFlow edges. An object token is a container for a value that flows over ObjectFlow edges.

A behavior step is enabled to be executed when tokens are offered to it on incoming edges and specified conditions are met. When an execution of a behavioral step is started, "*tokens are accepted from some or all of its incoming* ActivityEdges *and a token is placed on the node*" [13, p. 373]. When a behavioral step completes an execution, "*a token is removed from the node and tokens are offered to some or all of its outgoing* ActivityEdges" [13, p. 373]. Thereby, we may distinguish following types of events occurring during the Activity execution:

- tokens acceptation by a node – e.g. a start of an ExecutableNode execution,
- tokens offering by a node – e.g. a finish of an ExecutableNode execution.

At the given point of an activity's execution, event types are disjoint for an activity node, which means that only one of them can be enabled.

An execution trace of an Activity is an actual sequence of tokens acceptances and/or offerings occurrences due to the invocation, consistent with the specification of the Activity. Set of all possible execution traces of the Activity is the semantic of the Activity.

In our approach we consider the semantics of a set of Activities with one indicated initial activity. The initial activity refers directly or indirectly to activities in the set by calling other activities (CallBehaviorAction), signal sending (SendSignalAction), etc.

3 Semantics of Activities

We introduce the metamodel representing execution traces of a set of activities (hereinafter referred to as "ASmetamodel"), (Fig. 1). We define the ASmetamodel in the context of UML Activity specification [8], thereby some of classes introduced to the ASmetamodel refer to UML Activity metaclasses, for example to the ActivityNode, the ActivityEdge. The ASmetaclasses are marked by «ASmetaclass» keyword.

The set of activities (instances of UML Activity metaclass) are represented by a set of Act activities. When an Act is invoked a new instance of this is created. The instance is represented by the Execution metaclass.

An execution trace of an Act activity is defined by indicating consecutive places of tokens within the considered Act. A place of a token in the Act is represented by the Location metaclass (Fig. 2). A token generally is represented by Token metaclass, which is specialized by the ControlToken and the ObjectToken metaclasses modelling a control token and object token, respectively. For the sake of the simplicity, an ObjectToken does not contain a value, however it states that a value is present at all. In the context of the Location, a token (referenced by *token* metaattribute) may be situated either at the *node* or the *edge* of the Act.

The Snapshot metaclass represents locations of all tokens at some point (snapshot) of an execution trace of an Act activity (Fig. 2). It corresponds to progress in the Act's behavior execution. A Snapshot refers to the instance of the Act by the *execution*, and to a set of tokens places by the *locations*.

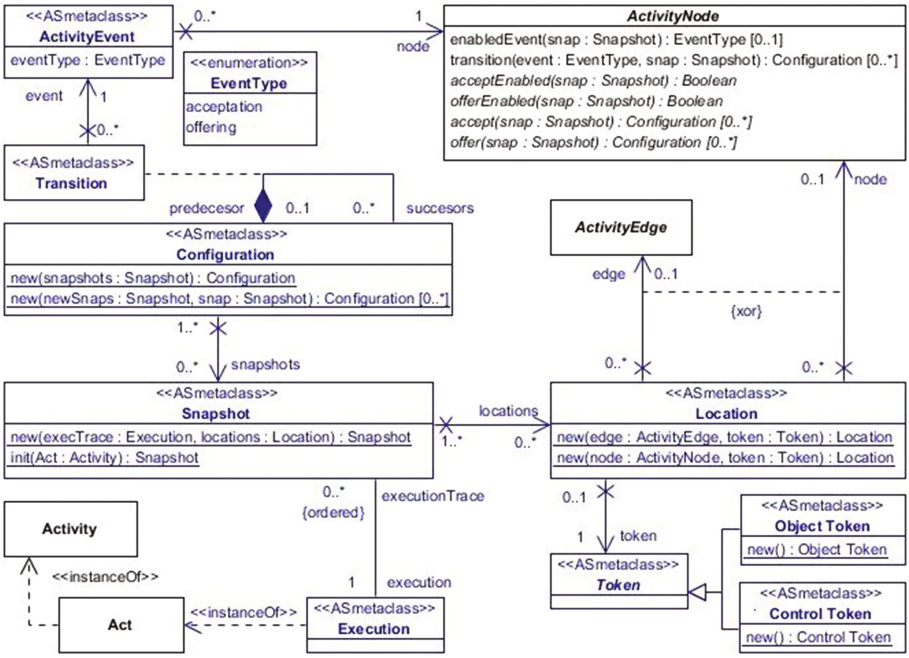

Fig. 1. Activity semantics metamodel.

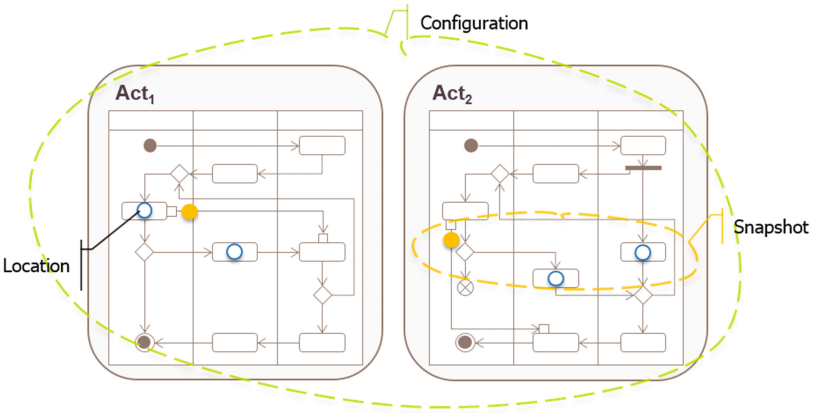

Fig. 2. Example configuration of a set of activities.

The Configuration metaclass represents places of tokens (indirectly via *snapshots*) after an event occurrence and before the very next event occurrence, for all instances of activities (Fig. 2). Or, if it is the Activities' invocation, it specifies initial places of tokens within the instances. The Configuration refers to progress in an execution of the set of Activities.

An event occurrence causing a transition from one Configuration to another is modelled by the Transition association metaclass. The event is specified by an ActivityEvent, which has an event type (*eventType*) and an activity node (*node*) which it refers to. An event type is one of the EventType values, i.e. either the acceptation value, for acceptation events, or the offering value, for offering events.

A Configuration refers to a *predecessor* configuration, unless it is the very first configuration, i.e. one from which it was directly transited.

There may be many successor configurations for a Configuration. Firstly, if there are many enabled events occurrences for the configuration, e.g., many ExecutableNodes in the activities are enabled to accept tokens (e.g., because of concurrent tokens flows), a new Transition is added for each event occurrence. Secondly, there may be many successor configurations, for a single event occurrence. For example, an ExecutableNode having optional outputs (OutputPin) may offer tokens at every subset of the outputs. Thereby, an event occurrence – tokens offering by the ExecutableNode – results in a set of possible successor configurations for each subset of the outputs. Similarly, there might be many possible ways of tokens acceptation, in example presented in Fig 3. The action within the snapshot (a.) can accept a control token from two alternative locations, and similarly can accept an object token (via input pin) from two alternative locations. There are four possible cases (two of them are presented in the Fig. 3 by snapshots (b.) and (c.)).

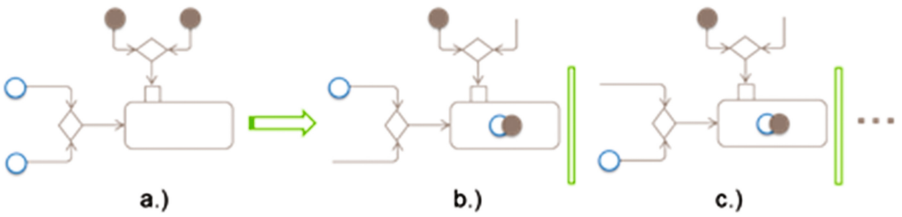

Fig. 3. Possible ways of tokens acceptation.

Metaclasses of the ASmetamodel have operations responsible for creation and expanding a model (a graph) representing semantic of the set of activities – all possible execution traces of the set, Fig. 2. The operations are defined using OCL [7].

For the sake of convenience, we defined operations named *new* supporting instances creations of all ASmetaclasses. In the Snapshot ASmetaclass we defined operation *init*, which sets up tokens initial placement (at the very beginning of an Activity invocation) for a given Snapshot.

We define operations within the UML metaclasses, mainly in ActivityNode. These operations are used to define transitions rules of the Labelled Transition System, introduced in the Subsect. 3.2.

Some of the operations (*acceptEnabled*, *offerEnabled*, *accept*, *offer*) are abstract, because there are specific to particular types of ActivityNode. The *acceptEnabled* (*offerEnabled*) determines whether *acceptation* (*offering*) event is enabled in the context of a node and for a given snapshot. The *accept* (*offer*) operation,

in the context of a node and for a given snapshot, results in a set of successor configurations (transits to new configurations), when the node accept (offer) tokens.

Additionally, there are auxiliary operations supporting LTS definition *enabledEvent* and *transition* common for all ActivityNodes. The *enabledEvent* operation (calling *acceptEnabled* and *offerEnabled* operations) provides available event type for a node within a given snapshot if any. The operation can be seen as a premise for running a transition on the node within the snapshot. The *transition* operation (calling *accept* and *offer*) does enabled transition for a node due to a given event type and snapshot, which results in a set of succeeding configurations.

Because of the space limitation, we put definitions of these operations in the appendix [17].

For the sake of simplicity we write thereafter $c \in Configuration$ meaning that c is an instance of the metaclass Configuration. Similarly, we write thereafter $C \subseteq Configuration$ meaning that C is a set of instances of the metaclass Configuration.

3.1 The Labelled Transition System

The semantics of an activity act_0 is defined in the context of a set of activities (including act_0) $\mathcal{A}$ being used by act_0. We introduce the semantics by means of the Labelled Transition System [6]:

$$LTS(\langle act_0, \mathcal{A} \rangle) = \langle \mathcal{C}, \rightarrow, \mathcal{E} \rangle \tag{1}$$

where:

$\mathcal{C} \subseteq Configuration$ is the set of possible configurations of the $\mathcal{A}$ activities,
$\mathcal{E} \subseteq ActivityEvent$ is the set of the $\mathcal{A}$ activities events,
$\rightarrow \subseteq \mathcal{C} \times \mathcal{E} \times \mathcal{C}$ is the set of allowed transitions between configurations.

We describe transitions $\rightarrow$ as a set of triples $\langle c, e, c' \rangle$, where a triple (a transition) define the very next configuration c', after a configuration c (i.e., to which c is transited), when an event e occurs. The result of applying $\rightarrow$ on c' is a set of new configurations $C' \subseteq \mathcal{C}$. The semantics imposes no order on the particular event to be processed and is nondeterministic.

A single transition: $\langle c, e, c' \in \rightarrow \rangle$ we will note thereafter in following way: $c \xrightarrow{e} c'$.

We distinguish two types of transitions (for each event type: $\rightarrow = \rightarrow_{accept} \cup \rightarrow_{offer}$).

- Transitions when the acceptation type events occur – if an acceptation event is enabled for an activity node within given snapshot (operation *node.acceptEnabled* returns true), then the activity node accepts tokens offered to it from some or all of its incoming edges and starts processing, i.e. for all possible cases of acceptation, new configurations are created (a result of *node.accept* call in stored in C').

$$\rightarrow_{accept} = \left\{ c \xrightarrow{e} c' \middle| c \in \mathcal{C} \wedge e \in \mathcal{E} \wedge e.eventType = acceptation \right.$$
$$\left. \wedge \exists_{s \in c.snapshots} \cdot \text{e}.node.acceptEnabled(s) \wedge c' \in e.node.accept(s) \right\} \quad (2)$$

– Rules for the offering event type – if an offer event is enabled for an activity node within given snapshot (operation *node.offerEnabled* returns true), then the activity node completes execution and token is removed from the node and tokens are offered to some or all of its outgoing edges, i.e. for all possible cases new configurations are created (a result of *node.offer* call in stored in C').

$$\rightarrow_{offer} = \left\{ c \xrightarrow{e} c' \middle| c \in \mathcal{C} \wedge e \in \mathcal{E} \wedge e.eventType = offering \right.$$
$$\left. \wedge \exists_{s \in c.snapshots} \cdot \text{e}.node.offerEnabled(s) \wedge c' \in e.node.offer(s) \right\} \quad (3)$$

We specify transitions for a set of selected kinds of activity nodes. For a given kind of node, a transition is specified for an event if it stems from UML specification. For example the acceptation event is applicable for the FinalNodes, and the ExecutableNodes, however the offering only for the ExecutableNodes [8]. We concretize the *acceptEnabled* and the *accept* operations, if the acceptation event is applicable, and accordingly the *offerEnabled* and the *offer* operations for the offering. Below we discuss transitions for the OpaqueAction, the FlowFinalNode, and the ActivityFinalNode, however for the sake of brevity we present detailed operations only for OpaqueAction and acceptation event.

OpaqueAction – accept

Within a given snapshot, an opaque action can accept tokens, if they are offered to all incoming control flows, and all incoming object flows of the action's mandatory input pins [13, p. 401]. It is defined in the *acceptEnabled* as follows:

$$context\ \mathbf{OpaqueAction} :: \mathbf{acceptEnabled}(snap: Snapshot): Boolean$$
$$body: controlLocationsCases(snap) \rightarrow notEmpty()\ and$$
$$mandatoryInputPins \rightarrow notEmpty()\ implies$$
$$objectLocationsCases(mandatoryInputPins, snap) \rightarrow notEmpty()$$

Tokens are offered to an edge by the source node of the edge. Offers propagate through edges and control nodes until they reach an action [13, p. 374]. As it is discussed in Subsect. 3.1, there are many ways in which tokens can be offered to the action.

We consider variations without repetition of sources of tokens. The *controlLocationCases* operation (see the appendix [17]) returns all mutually exclusive sets of control token locations from which tokens can be accepted by all control incoming edges of the opaque action. Similarly, we do for object token locations by invoking *objectLocationsCases* operation [17], however, ensuring that only mandatory input pins are taken into account. Tokens can be accepted if the control token location set is not empty and the object token location set is not empty, providing that there are any mandatory pins.

Within the given snapshot, when the opaque action begins execution, tokens are accepted from some or all of its incoming edges and a token is placed on the node [13, p. 373]. If there are many ways of tokens acceptation, a way of the acceptance is chosen randomly. Thereby, a new configuration is generated for each possible way, in the following manner:

$$
\begin{aligned}
&context\ \textbf{\textit{OpaqueAction}} :: \textbf{\textit{accept}}(snap: Snapshot): Set(Configuration)\\
&post:\\
&let\ loc: Location = result.snapshots.locations \rightarrow flatten()\\
&\qquad \rightarrow any(l | l.oclIsNew()\ and\ l.node\\
&\qquad = self\ and\ l.token.oclIsNew()\ and\ l.token.oclIsTypeOf(ControlToken))\ in\\
&let\ allInputPinsObjectLocationsCases: Set(Set(Location)) = inputPinCases\\
&\qquad \rightarrow collect(pinCase: Set(InputPin) | objectLocationsCases(pinCase, snap))\\
&\qquad \rightarrow collect\ (case: Set(Location) | case\\
&\qquad \rightarrow reject(l: Location |\ l.node <> null\ and\ l.oclIsType(DataStoreNode)))\\
&\qquad \rightarrow asSet()\ in\\
&mutualExclusiveCases(Set\{controlLocationsCases(snap), allInputPinsObjectLocationsCases\})\\
&\rightarrow forAll(case: Set(Location) | result \rightarrow one(c: Configuration | c.oclIsNew()\ and\ c.snapshots\\
&\qquad \rightarrow excluds(snap)\ and\ c.snapshots\\
&\qquad \rightarrow one(s: Snapshot | s.oclIsNew()\ and\ s.location = snap.locations - case\\
&\qquad \rightarrow including(loc)))\ )
\end{aligned}
$$

A new location *loc* with a new control token is created for the action.

A set of alternative locations for all subsets of input pins, for which tokens can be accepted, is computed. The set is obtained using the *inputPinCases* operation [17] which returns a power set of the input pins. Next, the *objectLocationsCases* operation [17] is applied to each element (a set of input pins) of the power set, and results are collected.

Last but not least, because of the semantics of the DataStoreNode, an object token taken from the data store has to be copied and restored [13, p. 397]. It is imitated by removing the object token from the set of alternative locations.

The operation *mutualExclusiveCases* [17] is invoked to calculate all mutually exclusive locations, i.e. combination of control token locations (result of the *controlLocationsCases* operation call [17]) and object token locations (the *allInputPinsObjectLocationsCases* value).

New snapshots for the alternative locations are calculated and in the result, a new configuration is created for each new snapshot. Finally, the configurations are returned by the *accept* operation.

OpaqueAction – offer

When the opaque action completes an execution a token is removed from the action and tokens are offered to some or all of its outgoing edges, [13, p. 373]. Within a given snapshot, the opaque action completes execution if there is a control token location related to the action. It is defined in the *OpaqueAction::offerEnabled* in the appendix [17].

Control tokens are offered on all outgoing control flows of the opaque action. Object tokens are placed in all mandatory output pins in order to be offered to outgoing object flows of the pins. Depending on a result of the opaque action execution, object tokens may be placed within a subset of optional output pins. Moreover, if object tokens are effectively offered to the ActivityParamterNode or the DataStoreNode, they are immediately propagated to these nodes [13, pp. 396–397]. If there are many ways of tokens offering, a way of the offering is chosen randomly. In such situation, the offer operation generates a set of configurations for the all possible ways. It is defined in the *OpaqueAction::offer* in the appendix [17].

ActivityFinalNode – accept
Within a given snapshot, an activity final node can accept tokens, if they are offered to any incoming control flow. When the activity final node accepts tokens, an activity instance (an execution) referred by the snapshot is finished. It is defined in the *ActivityFinalNode::acceptEnabled* and *ActivityFinalNode::accept* in the appendix [17].

FlowFinalNode – accept
Within a given snapshot, a flow final node can accept tokens, if they are offered to any incoming control flow. When the flow final node accepts tokens, the tokens are removed from an activity instance (an *execution*) referred by the snapshot. If there are no enabled events for any node within the activity instance, the snapshot is removed and the activity instance is destroyed. It is defined in the *FlowFinalNode::acceptEnabled* and *FlowFinalNode::accept* in the appendix [17].

3.2 Reachability Graph

A graph of possible execution traces of an activity act_0 in the context of a set of activities $\mathcal{A}$ is defined as:

$$G(\langle act_0, \mathcal{A} \rangle) = \langle V, A \rangle \tag{4}$$

where:

V – is a set of graph vertices; each vertex is labeled by a configuration which is reachable from the initial configuration c_0 of the activity act_0 (c_0 is an initial configuration when a snapshot of the configuration represents the act_0 instance initialized, i.e. locations are distributed within nodes and edges with respect to rules defined in [13, p. 376]. It is reflected in the *Snapshot::init* [17]).
A – is a set of graph arcs; each arc is labeled by a transition; and has form $\langle u, t, v \rangle \in A$, where $u, v \in V$ are vertices labeled by c_u, c_v : *Configuration*, and t : *Transition*.

The graph is directed. The root of the graph represents an initial configuration c_0, while leafs correspond to final configurations. A sequence of transitions starting from the c_0 results in one of the possible execution traces of the set of activities. The set of all possible execution traces corresponds to semantics of the activities.

We will use the function $label : V \rightarrow Configuration$, which for a given vertex of the graph of possible execution traces returns a configuration labeling the vertex. The graph are constructed iteratively starting from: $V = \emptyset$, $A = \emptyset$.

1. The initial vertex is set up:
   a. The initial configuration c_0 for the activity act_0 is created:
      $c_0 = Configuration :: new(Set\{Snapshot :: init(act_0, Sequence\{\})\})$
   b. Let v_0 be a vertex labeled by the configuration c_0: $V \leftarrow V \cup \{v_0\}$
2. The set of leaf-vertices is defined:
   $V_{leaves} = \{v \in V | label(v) \in c_0 \rightarrow closure(succesors) \rightarrow select(succesors \rightarrow isEmpty())\}$
3. For each leaf-vertex $v \in V_{leaves}$:
   a. For each snapshot of leaf-vertex $snap \in label(v).snapshots$:
      (1) The set of events which enable transitions for snapshot *snap* is defined:
      $EnabledEvents = snap.execution.type.node \rightarrow collect(n|n.enabledEvent(snap))$
      (2) For each enabled event $e \in EnabledEvents$:
         i. The set of new configurations C' are created: $C' = e.node.transition(e)$
         ii. For each new configuration $c' \in C'$:
            (a) The transition t is created: $t = Transition :: new(e.node, e, c, c')$
            (b) Let v' be a vertex labeled by configuration c': $V \leftarrow V \cup \{v'\}$
            (c) Let a be an arc of the form $\langle v, t, v' \rangle$: $A \leftarrow A \cup \{a\}$
   b. Check if any new vertex has been created: $\bigvee_{v \in V_{leaves}} \bigvee_{v' \in V} \bigvee_{t \in Transition} \langle v, t, v' \rangle \in A$, if yes go to step 2.

4 Conclusions

We gave a precise interpretation of a set of Activities in form of set of execution traces. We developed the transformation of a set of activities into the formally precise abstract behavior model, i.e. the graph of possible execution traces. Nodes of the graph correspond to a configuration, arcs correspond to transitions between two configurations, triggered when an event occurs. The provided semantics cover flows of control and object tokens taking into consideration variety caused by control nodes.

The transformation is formulated as the set of OCL rules, which makes it easily portable to transformation frameworks and model-based frameworks. It is the starting point for model analysis, execution (model debugging) tools, or support in checking correctness of model transformations.

In the future works, we will extend the semantics to other parts of the UML Activity notation. We will add support for new type of actions, e.g., call behavior action, send signal action, accept event action, and so far. We will support values transportation in object tokens. Moreover, we want to define semantics for parallel processing.

We want to develop the tool for generating a reachability graph of a set of activities. Next, our goal is to prepare more advanced tool supporting Use-Case specification process, model transformations, etc.

References

1. Daw, Z., Cleaveland, R., Vetter, M.: Formal verification of software-based medical devices considering medical guidelines. Int. J. Comput. Assist. Radiol. Surg. **9**(1), 145–153 (2014)
2. Daw, Z., Cleaveland, R.: An extensible operational semantics for UML activity diagrams. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 360–380. Springer, Cham (2015)
3. Daw, Z., Cleaveland, R.: Comparing model checkers for timed UML activity diagrams. In: Science of Computer Programming, pp. 277–299 (2015)
4. Eshuis, R., Wieringa, R.: Tool support for verifying UML activity diagrams. IEEE Trans. Softw. Eng. **30**, 437–447 (2004)
5. Grönniger, H., Reiß, D., Rumpe, B.: Towards a semantics of activity diagrams with semantic variation points. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 331–345. Springer, Heidelberg (2010)
6. Keller, R.M.: Formal verification of parallel programs. Commun. ACM **19**(7), 371–384 (1976)
7. OMG Object Constraint Language 2.4. http://www.omg.org/spec/OCL/2.4/. Accessed 03 Feb 2014
8. OMG Unified Modeling Language 2.5. http://www.omg.org/spec/UML/2.5/. Accessed 1 Mar 2015
9. Reggio, G., Leotta, M., Ricca, F.: Who knows/uses what of the UML: a personal opinion survey, model-driven engineering languages and systems. In: 17th International Conference, MODELS 2014, Valencia, Spain, pp. 149–165. Springer (2014)
10. Reggio, G., Leotta, M., Ricca, F., Clerissi D.: What are the used activity diagram constructs? – A survey. In: 2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), IEEE (2014)
11. Roubtsova, E.: Advances in behavior modeling. In: Advances in Computers, pp. 49–109. Academic Press, Orlando (2015)
12. Störrle, H.: Semantics of control-flow in UML 2.0 activities. In: Bottoni, P., Hundhausen, C., Levialdi, S., Tortora, G. (eds.) Proceedings of the IEEE Symposium on Visual Languages and Human- Centric Computing (VL/HCC), pp. 235–242 (2004)
13. Störrle, H.: Semantics of UML 2.0 acitivities. In: International Symposium on Visual Languages/Human Computer Centered Systems, pp. 235–242 (2004)
14. Störrle, H.: Towards a petri-net semantics of data flow in UML 2.0 activities. Technical report TR 0504, University of Munich (2004)
15. Störrle, H.: Semantics and verification of data flow in UML 2.0 activities. Electron. Notes Theor. Comput. Sci. **127**, 35–52 (2005)
16. Störrle, H., Hausmann, J.H.: Towards a formal semantics of UML 2.0 activities. In: Liggesmeyer, P., Pohl, K., Goedicke, M. (eds.) Software Engineering, Fachtagungdes GI-Fachbereichs Softwaretechnik. LNI, vol. 64, pp. 117–128. GI (2005)
17. Walkowiak, A., Gall, D.: An approach to semantics for UML activities – appendix. https://www.dropbox.com/s/11jrz5zo6lk5vxr/OCL_Appendix.pdf?dl=0. Accessed 31 May 2017