The TTC 2015 Model Execution Case

Tanja Mayerhofer and Manuel Wimmer

Business Informatics Group, Vienna University of Technology, Austria
{mayerhofer,wimmer}@big.tuwien.ac.at

Abstract. This paper describes a case study for the Transformation Tool Contest (TTC) 2015 concerning the execution of models. The case foresees the specification of the operational semantics of a subset of the UML activity diagram language with transformation languages. In particular, the computation of the end result of the execution of the activity diagrams is targeted as well as the provisioning of a precise trace for the complete execution. The evaluation concerns the correctness of the operational semantics specifications, its understandability and conciseness, as well as its performance.

1 Introduction

Executable models are being used for decades in computer science since the introduction of Petri nets and state machines to name just a few prominent examples. In the past years, they also became with Executable Domain Specific Modeling Languages (xDSMLs) and executable UML (xUML) an important research line in Model-Driven Engineering (MDE) [1, 2]. As a prerequisite for having executable models, the operational semantics of the modeling languages have to be explicated. In general, there are two approaches used for defining the operational semantics of models [5]. First, one may incorporate the runtime concepts into the metamodel of the modeling languages to represent execution states and define transformation rules for evolving the execution states of a model. Second, one may delegate the execution of models by mapping the modeling languages to some existing formalisms which already provide execution support. The second approach has been already covered in the past by previous TTC cases. However, the first approach has not been subject to investigation for a TTC case yet.

We believe that having a dedicated TTC case for the direct specification of the operational semantics within a language's metamodel is of major interest for the transformation community due to two reasons. First, there is already a large body of work discussing how to implement the operational semantics for modeling languages using different kinds of languages including also several model transformation languages (cf. for instance [3,4,6]). Second, with efforts such as fUML [8] and xDSMLs [1], language engineers have to reside on mature techniques to define the operational semantics for their modeling languages in a concise, reusable, and scalable way.

To shed some light on the current state-of-the-art of defining operational semantics with model transformations, we propose in this case description the task of specifying the operational semantics of a subset of the UML activity diagram language covering several control flow concepts and a simple expression language. The provided metamodel for this subset of the UML activity diagram language already contains the necessary runtime concepts. So the task for TTC attendees is to define a transformation,

which specifies the operational semantics of the UML activity diagram language by updating the runtime state of executed UML activity diagrams. Thus, the input model for the transformation is a UML activity diagram in its initial runtime state and the output model of the transformation is the final runtime state of the UML activity diagram including a trace of the execution. Due to these characteristics, the transformation is considered to be in-place and endogenous [7]. Please note that the transformation is only concerned with the abstract syntax of the models, which may trigger modifications in the concrete syntax, e.g., for model animation. However, the latter is not in the focus of this case.

2 The Transformation

This section describes the artifacts needed for solving this case, namely the UML activity diagram metamodel, a description of its operational semantics, as well as an example transformation trace.

2.1 Metamodel

Figure 1 shows an excerpt of the metamodel of the activity diagram variant considered in this case. It shows the basic concepts for modeling activities. Activities (metaclass Activity) consist of variables (metaclass Variable), activity nodes (metaclass ActivityNode), and activity edges (metaclass ActivityEdge).

Variables. For variables we distinguish between Integer variables and Boolean variables (metaclasses IntegerVariable and BooleanVariable). Variables may define an initial value (reference initialValue), where we again distinguish between Integer values and Boolean values (metaclasses Value, IntegerValue, and BooleanValue). Variables can serve as local variables or input variables of an activity (references locals and inputs of Activity).

Activity Nodes. There are two types of activity nodes available, namely control nodes (metaclass ControlNode) and actions (metaclass Action).

Control nodes can be used to define the start of an activity (metaclass InitialNode), the end of an activity (metaclass ActivityFinalNode), alternative branches of an activity (metaclasses DecisionNode and MergeNode), and concurrent branches of an activity (metaclasses ForkNode and JoinNode).

Actions constitute the fundamental unit of executable behavior and their execution represents some processing in the modeled system. In this case we consider so-called opaque actions (metaclass OpaqueAction), which can define an ordered sequence of expressions (metaclass Expression). Which kinds of expressions are supported, can be seen in the excerpt of the metamodel depicted in Figure 2.

We distinguish between Integer expressions and Boolean expressions (metaclasses IntegerExpression and BooleanExpression). An Integer expression processes two Integer variables (references operand1 and operand2). Integer calculation expressions (metaclass IntegerCalculationExpression) either perform a summation or subtraction

of these variables (enumeration IntegerCalculationOperator) and assign the resulting value to another Integer variable (reference assignee). Integer comparison expressions (metaclass IntegerComparisonExpression) compare the variables according to the defined comparison operator (enumeration IntegerComparisonOperator) and assign the resulting value to a Boolean variable (reference assignee). For Boolean expressions we distinguish between unary expressions and binary expressions (metaclasses BooleanUnaryExpression and BooleanBinaryExpression) that assign the resulting value to a Boolean variable (reference assignee). Unary Boolean expressions apply the logical operator *NOT* (enumeration BooleanUnaryOperator) on a Boolean variable (reference operand). Binary expressions apply the logical operators *AND* and *OR* on two Boolean variables (references operand1 and operand2). Please note that computed values cannot be assigned to input variables of activities.

Activity Edges. Activity edges are used to connect activity nodes with each other. Control flow edges (metaclass ControlFlow) define the flow of control among activity nodes. They may define a Boolean variable whose value serves as guard condition for the control flow (reference guard). Guard conditions are only allowed for outgoing control flow edges of decision nodes.

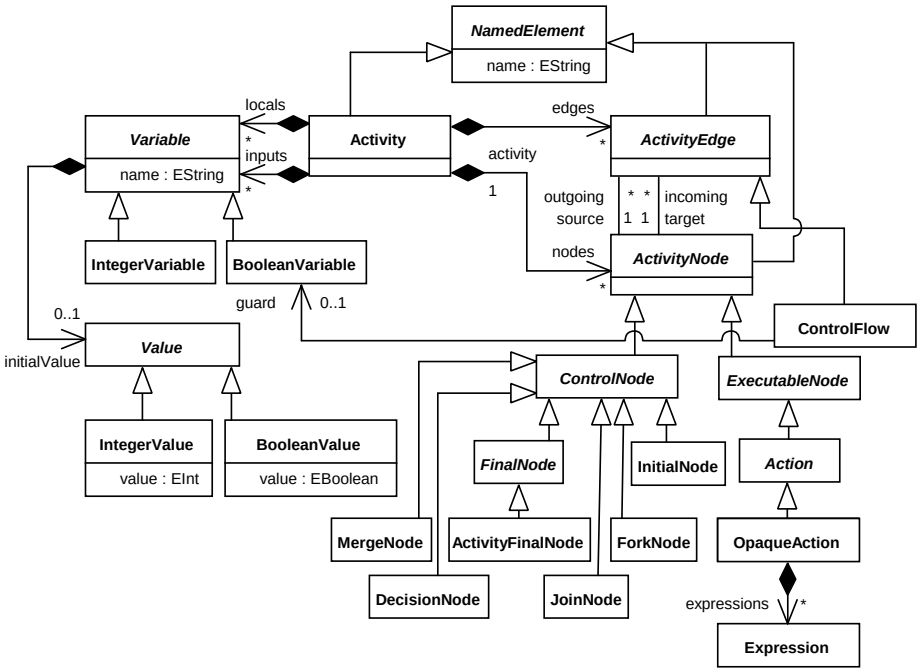


Fig. 1. Metamodel for UML activity diagrams (activities, variables, edges, nodes)

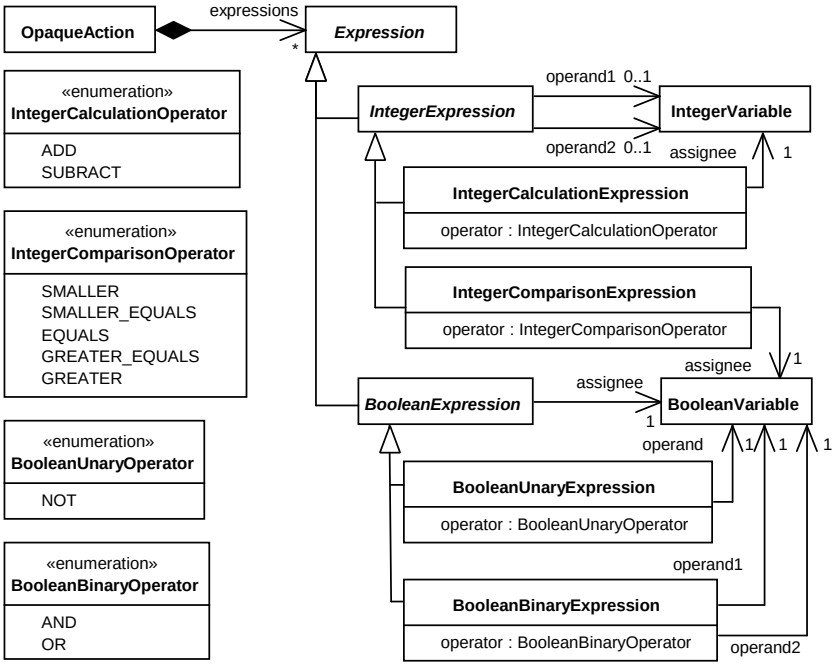


Fig. 2. Metamodel for UML activity diagrams (expressions)

2.2 Operational Semantics

An operational semantics defines the semantics of a modeling language by specifying the steps of computation required for executing a model conforming to the modeling language. This means, that an operational semantics defines an interpreter for the modeling language, which can be regarded as state transition system defining how an executing model progresses from state to state. Therefore, an operational semantics consists of two parts: *(i)* the definition of the *runtime concepts* needed for capturing the state of an executing model and *(ii)* the definition of the *steps of computation* involved in performing transitions of the executing model from one state to another state. Please note that runtime concepts may include also the definition of additional input values required for the execution of a model.

While the runtime concepts needed for defining the state of an executing model can be defined by applying metamodeling techniques, the steps of computation progressing the executing model to a new state has to be defined with transformation languages.

In the following, we discuss the runtime concepts and steps of computation defining the operational semantics of the UML activity diagram language. Please note that the defined operational semantics are based on the semantics of fUML [8].

Runtime Concepts. Figure 3 shows the metamodel defining the runtime concepts for the UML activity diagram language considered in this case. The runtime concepts are

depicted in orange color, while the metaclasses are depicted in white color. We distinguish between four types of runtime concepts: runtime concepts for capturing *(i)* the token flow among activity nodes, *(ii)* the current values of variables, *(iii)* the trace of an activity diagram, and *(iv)* input values that may be provided to an activity diagram.

The semantics of activity diagrams is based on a definition of token flow semantics similar to the token flow semantics of Petri nets (cf. [9] for a formalization of the semantics of UML activity diagrams using Petri nets as semantic domain). Informally speaking, an activity node is executed, when all required control tokens are available through incoming control flow edges. After the execution of an activity node is completed, control tokens are offered to the successor nodes via outgoing control flow edges. The runtime concept Token and its subclasses ControlToken and ForkedToken define how tokens are represented during execution. Thereby, forked tokens originate from the execution of fork nodes, splitting a control flow (reference baseToken) into multiple concurrent flows. Tokens are always owned by the activity node (reference heldTokens of ActivityNode) offering the tokens via activity edges to successor nodes (reference offers of ActivityEdge). The representation of token offers is defined by the runtime concept Offer.

Variables defined for an activity are initialized with their initial value (cf. Figure 1, reference initialValue of Variable), meaning that the value is copied and set as current value of the variable (reference currentValue). The current value of variables is updated during the execution of an activity by the execution of opaque actions defining assignments to these variables.

For capturing tracing information, the runtime concept Trace is defined keeping an ordered list of executed activity nodes (reference executedNodes).

For input variables of an activity (cf. Figure 1, reference inputs of Activity), input values to be processed by the execution of the activity may be provided. For representing these input values, the runtime concepts Input and InputValue are defined.

Please note that the defined runtime concepts extends the metamodel of the UML activity diagram language, i.e., they extend the metaclasses defined in the metamodel with additional attributes and references, and add additional metaclasses to the metamodel. This can, for instance, be done with the *package merge* operation known from UML and MOF. In our reference implementation, we introduced them directly into the metamodel of the UML activity diagram language.

Steps of Computation. The following steps of computation are defined for executing activity diagrams.

1. *Initialization of Variables.* The executed activity receives input values (cf. Figure 3, metaclass InputValue) for its defined input variables and initializes the current values of the input variables accordingly (cf. Figure 3, reference currentValue of Variable). Furthermore, also the current values of the local variables are initialized according to their initial values (cf. Figure 1, reference initialValue of Variable).
2. *Setting Activity Nodes as Running.* All nodes contained by the executed activity are set as being running (cf. Figure 1, attribute running of ActivityNode).
3. *Execution of Initial Node.* The initial node of the activity is executed. The execution of the initial node consists of producing a single control token, adding this control

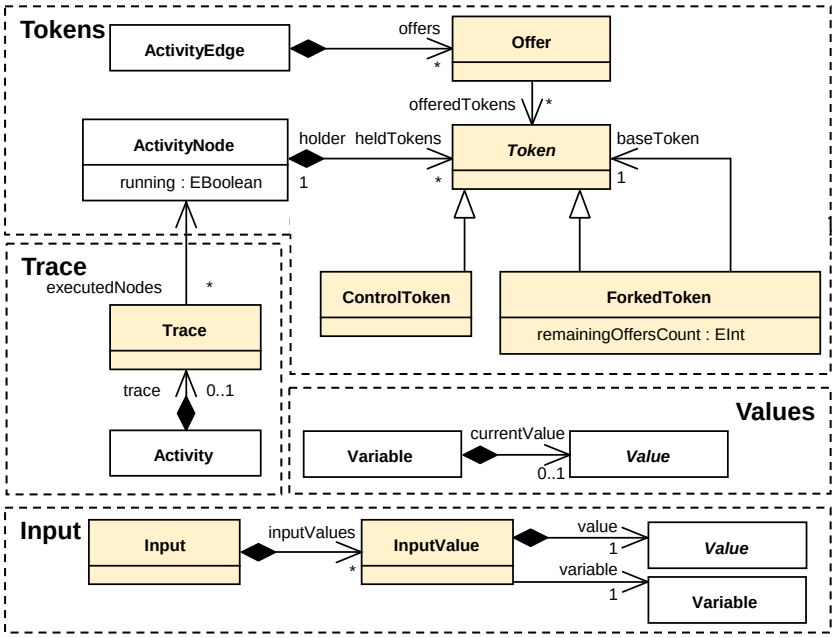


Fig. 3. Runtime concepts of UML activity diagrams

token to the initial node (cf. Figure 3, reference heldTokens of ActivityNode), and offering it via the outgoing control flow edges of the initial node to successor nodes (cf. Figure 3, reference offers of ActivityEdge). Please note that an activity has to contain exactly one initial node.

4. *Determination of Enabled Nodes.* The currently enabled nodes of the activity are determined. An activity node is enabled if it is set as being running and if all incoming control flow edges provide token offers.
5. *Selection and Execution of One Enabled Node.* One of the enabled nodes is selected and executed. Executing an enabled node consists of three steps:

   i *Consumption of Offered Tokens.* All tokens offered to the node via incoming control flow edges (in the following also referred to as *incoming tokens*) are consumed. This leads to the removal of all token offers of all incoming control flow edges (cf. Figure 3, reference offers of ActivityEdge). Furthermore, in the case of a consumed control token, the control token is removed from the offering (i.e., preceding) node (cf. Figure 3, reference heldTokens of ActivityNode). In the case of a consumed forked token, the forked token's remaining offers count is decremented by one (cf. Figure 3, attribute remainingOffersCount) and only if the remaining offers count is then equal to zero (i.e., every successor node of a fork node has processed the forked token), the forked token is removed from the offering node. Furthermore, if the forked token's base token (cf. Figure 3, reference baseToken) has not be removed from its offering node

(cf. Figure 3, reference holder of Token), the base token is removed from its offering node (independent of the remaining offers count).
Please note, that the operational semantics does not support implicit forking. For instance, consider an initial node having two succeeding actions (i.e., the initial node has two outgoing control flow edges each leading to a distinct action). In this case, the single control token produced and held by the initial node is offered by two offers—one offer for each outgoing control flow edge—to the two succeeding actions. However, only one of the actions can be executed, because the execution of one action will remove the token from the initial node making the offer to the other action obsolete, i.e., the token cannot be consumed by this second action anymore and it can thus not be executed.

ii *Execution of the Node's Behavior.* After the consumption of offered tokens, the behavior of the node is executed. Depending on the type of the node, control tokens may be produced, which are added to the node (cf. Figure 3, reference heldTokens of ActivityNode) and offered to successor nodes via the node's outgoing control flow edges (cf. Figure 3, reference offers of ActivityEdge).

iii *Tracing.* Furthermore, the executed node is added to the trace kept by the executed activity (cf. Figure 3, reference trace of Activity and reference executedNodes of Trace).

The behavior of the different types of activity nodes executed in Step *(ii)* is defined as follows:

(a) *Opaque Actions.* The expressions defined by an opaque action are executed in sequential order. The semantics of expressions consists of applying the defined operator on the current values of the defined operand variables and assigning the resulting value to the defined assignee variable as current value.
After all expressions have been executed, one control token is created for each outgoing control flow edge and offered to successor nodes via the respective edge.

(b) *Fork Nodes.* A fork node produces for each incoming token a forked token, whose base token is set to the corresponding incoming token and whose remaining offers count is set to the number of outgoing edges (cf. Figure 3, reference baseToken and attribute remainingOffersCount of ForkedToken). The created forked tokens are offered via all outgoing control flow edges of the fork node.

(c) *Decision Nodes.* A decision node evaluates the guard conditions of its outgoing control flow edges, i.e., it determines whether the Boolean variables defined as guard conditions have the Boolean value *true* set as current value. The incoming tokens are offered via the edge whose guard condition is fulfilled. Please note that only one guard condition is allowed to be fulfilled.

(d) *Join Nodes and Merge Nodes.* Join nodes and merge nodes offer the incoming tokens on all outgoing control flow edges.

(e) *Activity Final Nodes.* An activity final node terminates the execution of the containing activity causing all nodes contained by the activity to be set as not running (cf. Figure 3, attribute running of ActivityNode).

6. *Repetition of Steps 4 and 5 until Termination.* Step 4 and Step 5 are repeated until no activity nodes are enabled anymore constituting the termination of the activity.

Please note that in case an activity final node has been executed, no node is enabled anymore, because all nodes are set as not running.

In the provided reference implementation, the steps of computation are implemented using Java and EMF. Therefore, we introduced operations into the Java classes generated by EMF for the presented metamodel of the UML activity diagram language. In the following, we provide an overview of the most important operations.

Activity

```
/*
 * Receives input values for the activity's input variables
 * and starts the execution of the activity.
 */
void main(List<InputValue> inputValues);
/*
 * Initializes the activity's local and input variables.
 */
void initialize(List<InputValue> inputValues);
/*
 * 1. Sets all nodes of an activity as running.
 * 2. Fires the initial node.
 * 3. Determines the currently enables nodes.
 * 4. Selects one of enabled nodes and fires it.
 * 5. Repeats steps 3 and 4 until no node is enabled anymore.
 */
void run();
```

Activity Node

```
/*
 * Returns true if the node is enabled, false otherwise.
 */
boolean isReady();
/*
 * Consumes all tokens provided via incoming edges.
 */
List<Token> takeOfferedTokens();
/*
 * Adds tokens to node (heldTokens).
 */
void addTokens(List<Token> tokens);
/*
 * Executes the behavior of the node.
 */
void fire(List<Token> tokens);
/*
 * Offers tokens via all outgoing edges.
 */
void sendOffers(List<Token> tokens);
/*
```

```
 * Removes token from node (heldTokens).
 */
void removeToken(Token token);
/*
 * Returns true, if all incoming edges offer tokens,
 * false otherwise.
 */
boolean hasOffer();
```

Activity Edge

```
/*
 * Returns all offered tokens and destroys all offers.
 */
List<Token> takeOfferedTokens();
/*
 * Creates offer for provided tokens.
 */
void sendOffer(List<Token> tokens);
```

Action

```
/*
 * Executes an action.
 */
void doAction();
```

Expression

```
/*
 * Executes expression.
 */
void execute();
```

2.3 Example Transformation Execution Trace

Figure 4 shows an example of an activity diagram in UML notation. Please note that we provide with our reference implementation a textual concrete syntax for the UML activity diagram language, which is used for defining the UML activity diagrams used in our test suite (cf. Appendix A).

The activity shown in Figure 4 defines two variables: the input variable *internal* and the local variable *noninternal* both of type Boolean. The local variable *noninternal* is initialized with the value *false*. Furthermore, the activity consists of one initial node, one decision node, one fork node, one join node, one merge node, one activity final node and eight opaque actions, which are connected by 15 control flow edges. Noteworthy about the activity is, that the opaque action *register* defines one expression notinternal = ! internal and that the outgoing edges of the decision node define the two variables as guard conditions.

Figure 5 and Figure 6 visualize and explain the execution of this example activity diagram in the case that the value *true* is provided as input for the input variable *internal*. These figures illustrate the execution of the activity nodes contained by the UML

activity diagram one by one. Control tokens and offers of control tokens are shown in orange color. Forked tokens and offers of forked tokens are shown in blue color. Updates of important features are also highlighted in color. The execution is shown until the execution of the action *manager interview*. The complete trace of the example is as follows: initial node *initial* - opaque action *register* - decision node *decision* - opaque action *get welcome package* - fork node *fork* - opaque action *assign to project* - opaque action *add to website* - join node *join* - opaque action *manager interview* - opaque action *manager report* - merge node *merge* - opaque action *authorize payment* - activity final node *final*. Please note, that the opaque actions *assign to project* and *add to website* could also be executed in reverse order.

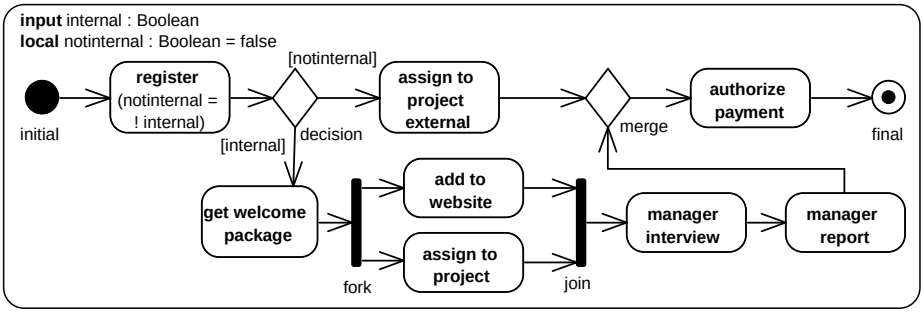


Fig. 4. Example activity diagram (UML notation)

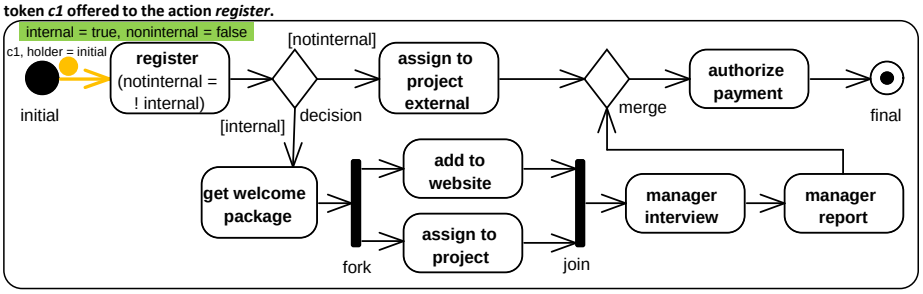
2.4 Variations

As the presented UML activity diagram language is quite extensive, solution developers may choose to implement it only partially. We foresee the following three case variations.

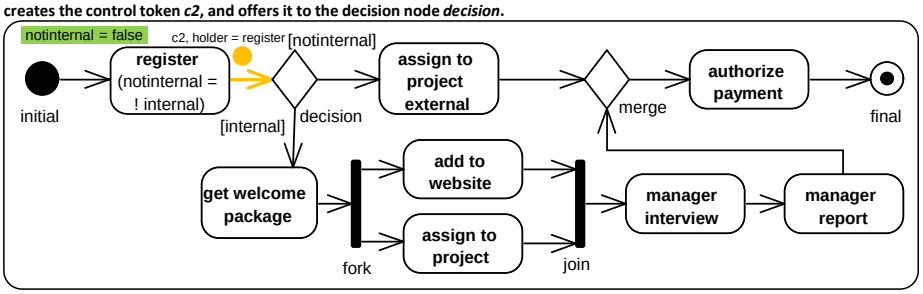
Variant 1: Simple Control Flow. The first variant considers only the following concepts of the UML activity diagram language: Activity, initial node, activity final node, opaque action (without expressions), control flow edge. This means that only the operational semantics of these concepts has to be implemented by solution developers choosing this case variant. The following runtime concepts have to be implemented for this variant: Offer, token, control token, trace. We consider this subset of concepts to be the minimal one that should be implemented by all solution developers.

Variant 2: Complex Control Flow. The second variant considers compared to the first variant the following additional concepts: Fork node, join node, decision node, merge node, local Boolean variables, and Boolean values. Only the runtime concept forked token as well as current values of Boolean variables have to be implemented additionally compared to the first variant.

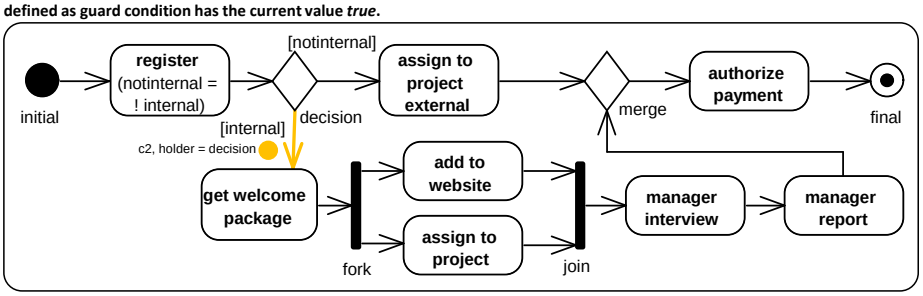
1. The variables are initialized, the nodes are set as running, the initial node is executed leading to the creation of the control token *c1* offered to the action *register*.



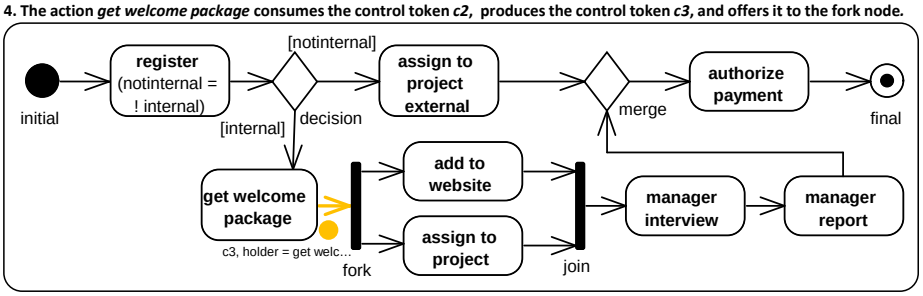
2. The action *register* consumes the token *c1*, executes the defined expression leading to an update of the variable *non-internal*, creates the control token *c2*, and offers it to the decision node *decision*.



3. The decision node *decision* offers the control token *c2* to the opaque action *get welcome package*, because the variable *internal* defined as guard condition has the current value *true*.



4. The action *get welcome package* consumes the control token *c2*, produces the control token *c3*, and offers it to the fork node.



5. The fork node *fork* produces the forked token *c4* for the incoming control token *c3* (i.e., the forked token's base token). The remaining offers count is set to 2, because the fork node has two outgoing control flow edges. The forked token c4 is offered to the successor actions via two distinct offers.

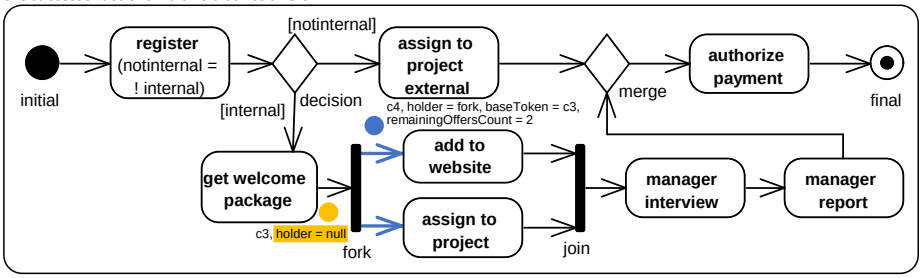
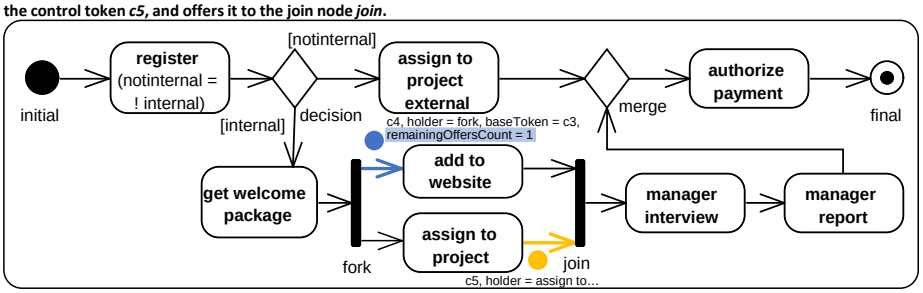
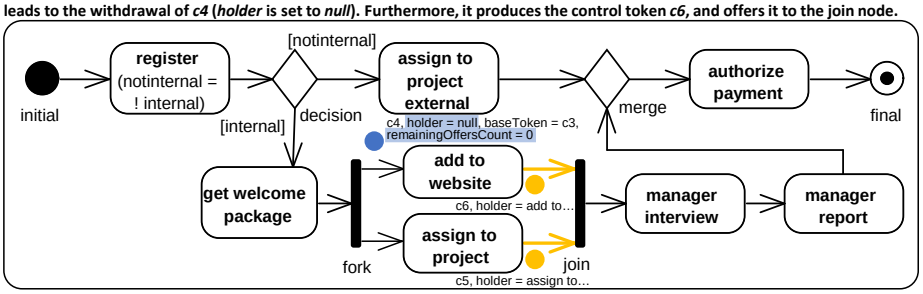


Fig. 5. Visualization of the execution of the example activity diagram (part 1)

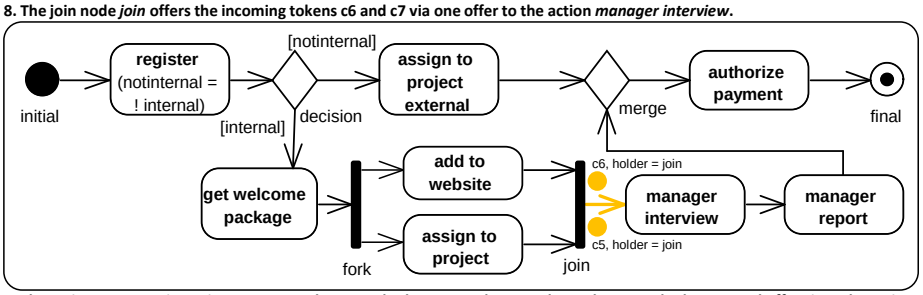
6. The action *assign to project* consumes its token offer for *c4* leading to an update of *c4*'s remaining offers count to 1, produces the control token *c5*, and offers it to the join node *join*.



7. The action *add to website* consumes its token offer for *c4* leading to an update of *c4*'s remaining offers count to 0, which in turn leads to the withdrawal of *c4* (*holder* is set to *null*). Furthermore, it produces the control token *c6*, and offers it to the join node.



8. The join node *join* offers the incoming tokens c6 and c7 via one offer to the action *manager interview*.



9. The action *manager interview* consumes the control tokens *c5 and c6*, produces the control token *c7*, and offers it to the action *manager report*.

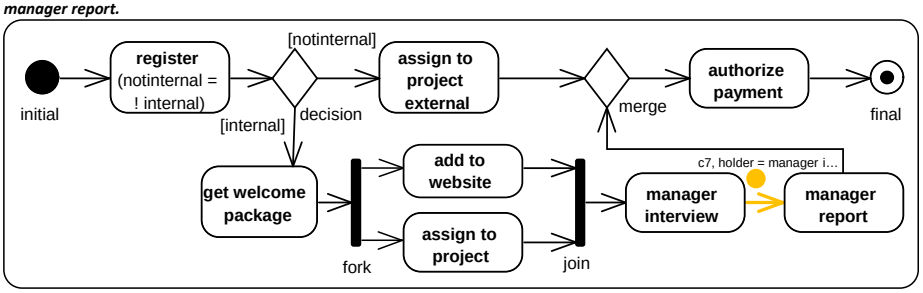


Fig. 6. Visualization of the execution of the example activity diagram (part 2)

Variant 3: Expressions The third variant considers the complete UML activity diagram language. Thus, the following additional concepts have to be implemented compared to the second variant: Input variables, Integer variables, Integer values, and all expression types. Also the runtime concepts input and input value have to be implemented in addition.

2.5 Reference Implementation

The reference implementation for this case may be found at the following open source code repository: `https://code.google.com/a/eclipselabs.org/p/moliz/source/browse?repo=ttc2015`. It consists of the following Eclipse plug-in projects.

org.modelexecution.operationalsemantics.ad: Contains the metamodel of UML activity diagram language including definitions of the runtime concepts required as part of its operational semantics, and the Java code generated by EMF for the metamodel including the steps of computation of the operational semantics.

org.modelexecution.operationalsemantics.ad.test: Provides a test suite for verifying the correctness of the implemented operational semantics. The test suite is explained in Section 3.

org.modelexecution.operationalsemantics.ad.grammar,
org.modelexecution.operationalsemantics.ad.grammar.ui,
org.modelexecution.operationalsemantics.ad.input.grammar,
org.modelexecution.operationalsemantics.ad.input.grammar.ui: Xtext projects implementing a textual concrete syntax for activity diagrams and input values.

For running the reference implementation, the *Eclipse Modeling Tools (version Luna)*[1] are required. Furthermore, the *Xtext plug-in* for Eclipse[2] has to be installed additionally.

3 Test Suite

As part of the reference implementation, we provide a test suite meant for evaluating the correctness and performance of the implemented operational semantics. Each test case contained by the test suite executes a single activity and asserts the resulting trace. If local variables are manipulated by the activity, also their final values are asserted. The provided test cases are described in the following.

test1: Tests the operational semantics of initial nodes, opaque actions, activity final nodes, and control flow edges.

test2: Tests the operational semantics of fork nodes and join nodes.

[1] Eclipse Modeling Tools may be downloaded from `http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/lunasr2`.

[2] Xtext can be installed in Eclipse using the menu *Help - Install Modeling Components*.

test3: Tests the operational semantics of decision nodes, merge nodes, and local Boolean variables.

test4: Tests the operational semantics of expressions.

test5: Tests the operational semantics of input variables.

test6: Defines the example activity diagram explained in Section 2.3.

testperformance_variant1: Performance test for variant 1 of this case. The UML activity diagram of this test cases comprises 1,000 sequential opaque actions.

testperformance_variant2: Performance test for variant 2 of this case. The UML activity diagram of this test cases comprises 100 concurrent branches each one comprising 10 opaque actions.

testperformance_variant3_1: Performance test for variant 3 of this case. Like the test case *testperformance_variant2*, the UML activity diagram of this test cases comprises 100 concurrent branches each one comprising 10 opaque actions. Furthermore, for each concurrent branch one variable exists that is incremented by the opaque actions lying on this branch.

testperformance_variant3_2: Performance test for variant 3 of this case. The UML activity diagram of this test case comprises 18 activity nodes including a loop introduced through decision and merge nodes. Due to the loop, 1,001 activity node executions occur.

Solution developers, who chose to implement variant 1 of the case only have to consider the test cases *test1*, and *testperformance_variant1*. For variant 2, the test cases *test2*, *test3*, and *testperformance_variant2* have to be considered additionally. Solutions for variant 3 have to consider all test cases.

A run configuration for executing all test cases is provided by the test project (*org.modelexecution.operationalsemantics.ad.test/TestSuite.launch*). In case a solution builds upon the Eclipse Modeling Tools and uses the provided metamodel, it is only required to override or modify the operation *executeActivity(String modelPath, String inputPath)* of the class *TestSuite* located in the project *org.modelexecution.operationalsemantics.ad.test*. This operation is responsible for loading the UML activity diagram to be executed as well as the activity diagram's input values, executes the UML activity diagram, and provides the trace captured during the execution as output. However, if a solution does not build upon the Eclipse Modeling Tools or does not use the provided metamodel, the test suite has to be re-implemented accordingly by the solution developers to demonstrate the solution's correctness and assess its performance.

The test project also contains textual representations of the traces obtained by the reference implementation for the UML activity diagrams of all test cases (folder *trace*). This textual representation comprises the execution order of the activity nodes of the respective UML activity diagram as well as the final values of local variables. Please note, that in the case of concurrent branches in the activity diagram, the execution order captured by the provided trace represent only *one* valid execution order.

4 Evaluation Criteria

The task of this case is to use a model transformation tool to implement the described operational semantics for a subset of the UML activity diagram and to execute models conforming to this language. Submissions are evaluated according to the following criteria.

4.1 Correctness

It is mandatory that solutions demonstrate that they have the intended behavior for all test cases covered by the test suite described in Section 3.

4.2 Understandability and Conciseness

Peer reviews will be used to assess qualitatively the conciseness and understandability of all solutions. We envision online reviews involving multiple rounds in order to reach consensus among all participants. Understandability and conciseness are used as measures to reason also about the implementation effort for the operational semantics specifications.

4.3 Performance

Performance is measured by logging how long the developed transformations need to execute the models, i.e., to produce the final runtime state. Therefore, the test cases *testperformance_variant1*, *testperformance_variant2*, *testperformance_variant3_1*, and *testperformance_variant3_2* described in Section 3 are used, depending on the case variant chosen by the solution developers. The execution time should be measured in milliseconds, e.g., in Java using java.lang.System.nanoTime(). Please note that reading the input model and writing the output model is not considered to be part of this performance evaluation.

4.4 Overall Assessment

The final score for submitted solutions will be calculated by summing up a maximum of 100 points in total, while for the correctness, understandability and conciseness, and performance, 25, 50, 25 points are reserved, respectively. Therewith, we want to strongly emphasize the importance of having concise and understandable operational semantics specifications in order to best support language engineers in developing, maintaining, and extending such kind of specifications as they are considered to be one of the most critical artifacts in the language engineering process.

References

1. Bryant, B.R., Gray, J., Mernik, M., Clarke, P.J., France, R.B., Karsai, G.: Challenges and Directions in Formalizing the Semantics of Modeling Languages. Computer Science and Information Systems 8(2), 225–253 (2011)

2. Combemale, B., Crégut, X., Garoche, P.L., Thirioux, X.: Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification. Journal of Software 4(9), 943–958 (2009)
3. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: Proceedings of the 3rd International Conference on The Unified Modeling Language (UML). LNCS, vol. 1939, pp. 323–337. Springer (2000)
4. Jézéquel, J., Barais, O., Fleurey, F.: Model Driven Language Engineering with Kermeta. In: Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE). LNCS, vol. 6491, pp. 201–221. Springer (2009)
5. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Model Transformation Intents and Their Properties. Software & Systems Modeling pp. 1–35 (2014)
6. Mayerhofer, T., Langer, P., Wimmer, M., Kappel, G.: xMOF: Executable DSMLs Based on fUML. In: Proceedings of the 6th International Conference on Software Language Engineering (SLE), LNCS, vol. 8225, pp. 56–75. Springer (2013)
7. Mens, T., Van Gorp, P.: A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science (ENTCS) 152, 125–142 (2006)
8. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.1 (August 2013), http://www.omg.org/spec/FUML/1.1
9. Syriani, E., Ergin, H.: Operational Semantics of UML Activity Diagram: An Application in Project Management. In: Proceedings of the 2nd Workshop on Model-Driven Requirements Engineering Workshop (MoDRE). pp. 1–8. IEEE (2012)

A Textual Concrete Syntax

For defining UML activity diagrams more conveniently, we provide a textual concrete syntax implemented with Xtext. Listing 1 shows the exemplary UML activity diagram discussed in Section 2.3 and shown in Figure 4 in this textual concrete syntax.

```
activity Test7 (bool internal) {
  bool notinternal = false

  nodes {
    initial initialNode7 out(edge42),
    action register comp {notinternal = !internal} in(edge42) out(edge43),
    decision decisionInternal in(edge43) out(edge44, edge45),
    action assignToProjectExternal in(edge44) out(edge56),
    action getWelcomePackage in(edge45) out(edge46),
    fork forkGetWelcomePackage in(edge46) out(edge47, edge48),
    action assignToProject in(edge47) out(edge49),
    action addToWebsite in(edge48) out(edge50),
    join joinManagerInterview in(edge49, edge50) out(edge51),
    action managerInterview in(edge51) out(edge52),
    action managerReport in(edge52) out(edge53),
    merge mergeAuthorizePayment in(edge53, edge56) out(edge54),
    action authorizePayment in(edge54) out(edge55),(*final finalNode7 in(edge55)
  }

  edges {
    flow edge42 from initialNode7 to register,
    flow edge43 from register to decisionInternal,
    flow edge44 from decisionInternal to assignToProjectExternal [notinternal
        ],
    flow edge45 from decisionInternal to getWelcomePackage [internal],
    flow edge46 from getWelcomePackage to forGetWelcomePackage,
```

```
    flow edge47 from forkGetWelcomePackage to assignToProject,
    flow edge48 from forkGetWelcomePackage to addToWebsite,
    flow edge49 from assignToProject to joinManagerInterview,
    flow edge50 from addToWebsite to joinManagerInterview,
    flow edge51 from joinManagerInterview to managerInterview,
    flow edge52 from managerInterview to managerReport,
    flow edge53 from managerReport to mergeAuthorizePayment,
    flow edge54 from mergeAuthorizePayment to authorizePayment,
    flow edge55 from authorizePayment to finalNode7,
    flow edge56 from assignToProjectExternal to mergeAuthorizePayment
  }
}
```

Listing 1. Example activity diagram (textual concrete syntax)