The SDMLib solution to the Model Execution Case for TTC2015

Stefan Lindel, Albert Zündorf
Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73, 34121 Kassel, Germany

slin|zuendorf@cs.uni-kassel.de

This paper describes the SDMLib solution to the Model Execution case for the TTC2015 [**?**]. We solved all case variants and did all performance tests. For this case we generated the Java implementation of the activity diagram classes with SDMLib in order to have an efficient model representation. Then we modeled the operations using SDMLib model transformations. These model transformations were embedded into methods of the activity diagram classes leveraging the overriding of methods for the distinction of different behavior for different kinds of activity nodes. Our solution deviates from the case description in the handling of tokens: instead of consuming and recreating tokens we use just one token and allow it to be at several places at a time and we just move the token forward through the activity diagram. This results in more elegant modeling and faster execution.

1 Introduction

We assume that the reader is familiar with the description of the TTC2015 model execution case [**?**]. This paper describes the SDMLib [**?**] solution to the TTC2015 model execution case. The task is to execute activity diagrams via model transformations. One shall show, how model transformations fit for this purpose. The case descriptions comes with an example implementation that uses a token game for execution that is borrowed from Petri Nets. Basically, the example implementation suggests that an activity node may be executed if there is a token offered at each incoming control flow arc and that the activity node consumes all these tokens, executes any inner action and creates new token offerings on each outgoing arc. Fork and join nodes get a special treatment using a sub-token that counts how many of the parallel activities have been executed already and to deduce when the join is complete.

We think the proposed token handling is pretty complicated and inefficient. To come up with a simpler solution, we removed all token related classes from the example solution and replaced them with a new Token class that has a to-many association `currentElements` to class `NamedElement`, cf. Figure **??**. We use only a single `Token` object that may have multiple `currentElements` at a time. On execution, one of the `currentElements` is chosen and the corresponding link is moved forward to the next `NamedElement`. In addition, the `token` is attached to the current `Activity` via a to-one association named `token`. To count how many parallel actions have reached a join node, we use attribute `noOfVisitors` provided by class `ActivityNode`. Actually, only objects of class `JoinNode` need this attribute, but by providing it generally, the modeling of the interpreter becomes simpler.

Figure **??** shows an object diagram depicting the activity diagram of test 2 of the model execution case during execution. The `InitialNode i14` and the `ForkNode f3` have already been added to the `Trace t15`. `Activity a1` has a `Token t2` currently pointing to `ForkNode f3`. On execution, the `ForkNode` will remove itself from the set of `currentElements` of the `Token` and will add its outgoing `ControlFlow` objects `c12` and `c4` to the `currentElements` instead. In the next turn, one of the control flows (e.g. `c12`)

Submitted to:
TTC 2014

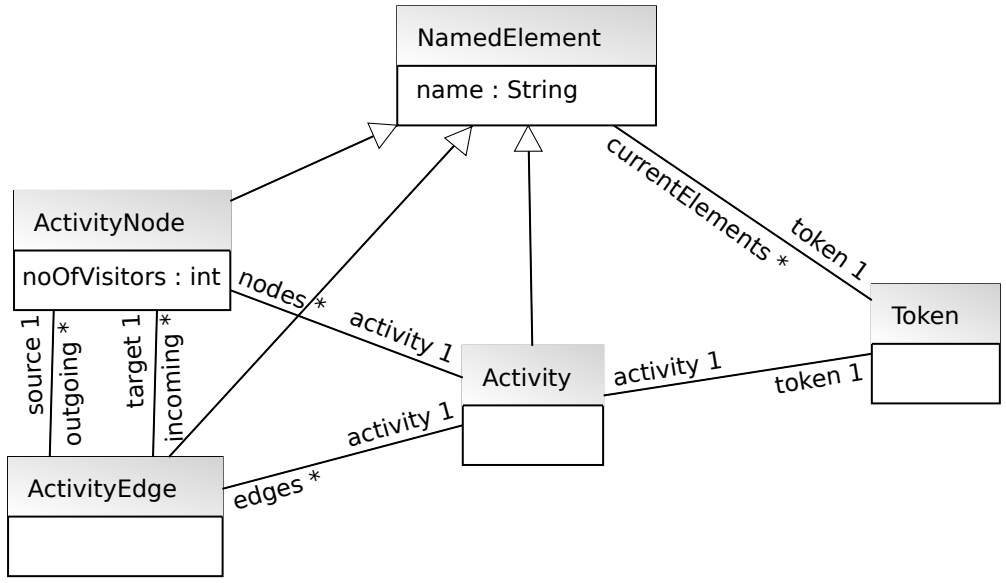

Figure 1: Simplified Token Handling

will remove itself from the `currentElements` and add its target object (e.g. `o11` instead. In addition, the `noOfVisitors` attribute of the target object is incremented. Later on, when the `JoinNode` `j7` is executed, `j7` checks its `noOfVisitors`. If this is lower than the number of incoming `ControlFlows`, not all parallel executions have reached the `JoinNode` yet and thus, the `JoinNode` deletes the `currentElements` link but does not forward it. Only when `noOfVisitors` indicates that all parallel branches have reached the `JoinNode`, the `currentElements` link is forwarded to the `outgoing` `ControlFlow`.

2 The model execution transformations

As a start, Listing **??** shows the Java source code that builds and runs the SDMLib model transformation initializing the variables of an activity. Figure **??** shows this transformation graphically[1].

```
1 class Activity {
2     public void initVariables(){
3         ActivityPO activityPO = new ActivityPO(this);
4         VariablePO localVariablePO = activityPO.hasLocals();
5         ValuePO valuePO = localVariablePO.hasInitialValue();
6         localVariablePO.createCurrentValue(valuePO);
7         localVariablePO.doAllMatches();
8     }
```

Listing 1: Initialize variables transformation in Java

In SDMLib a model transformation is called a *Pattern* and it consists of *Pattern Objects* and *Pattern Links* that are matched against actual model objects. For the initialization of activity variables we use a Pattern with three Pattern Objects: `activityPO`, `localVariablePO`, and `valuePO`. The constructor call

[1] SDMLib is able to render a model transformation as HTML or SVG.

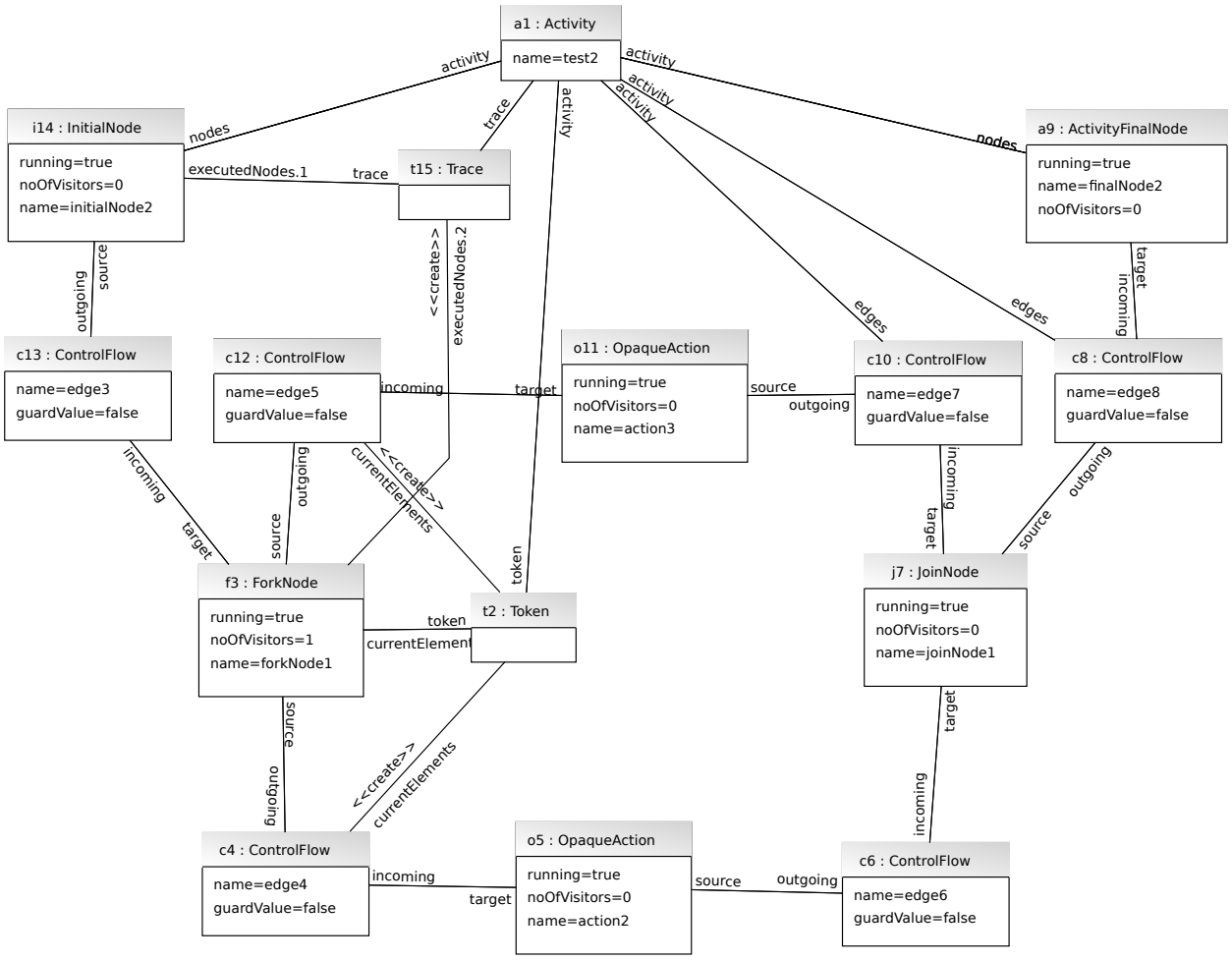

Figure 2: Moving the Token through the Activity Diagram

`new ActivityPO(this)` creates the Pattern and adds the `activityPO` Pattern Object to it and binds `activityPO` to the current model object `this`. This means, the Pattern Object `activityPO` is directly matched against the model object `this`. It will also serve as start for the pattern matching process.

Next, the command `activityPO.hasLocals()` creates the Pattern Object `localVariablePO` and a Pattern Link of type `locals` that connects `activityPO` and `localVariablePO`. Then, the pattern matching is initiated and SDMLib tries to find model objects of type `Variable` that are connected to the current `Activity` object via a `locals` link. If there are multiple candidates, the candidates are stored for as possible matches. One of the candidates is chosen as the current match. If there is no match for a given Pattern Object, backtracking is initiated and SDMLib tries to chose other candidates for previously visited Pattern Objects and then revisits the current Pattern Object. If backtracking fails, too, the whole matching fails. In the current example case let us assume that there are two variables `v1` and `v2`. Thus Pattern Object `localVariablePO` will be matched e.g. against `v1` and `v2` will be stored as alternative candidate.

SDMLib generates the Method `hasLocals()` within class `ActivityPO` from the association `locals` between the classes `Activity` and `Variable`. For each association role such a `has` method is generated

Figure 3: Initialize variables transformation

in the corresponding PO class. These has methods create a Pattern Link according to the role name and a Pattern Object according to the role's target class.

Line 5 of Listing **??** extends the search Pattern by an valuePO Pattern Object connected to localVariablePO via an initialValue link. Next, line 6 uses method createCurrentValue to extend our model transformation by an action that creates a currentValue link between the model objects matched by localVariablePO and valuePO. This create action is executed only if the Pattern has a successful match.

Finally, line 7 calls method doAllMatches. Method doAllMatches triggers the backtracking of the Pattern search, i.e. we go back to choices where still alternatives are available. In our example, this is the matching of localVariablePO to var1. Thus, localVariablePO is now re-matched against v2 and the remaining pattern matching, i.e. the search for a value and the creation of a currentValue link is executed again. Method doAllMatches triggers backtracking until the Pattern search and execution fails. Overall, now all local variables of the current activity are initialized.

Model transformation initVariables is the first operation called within method run() of class Activity, cf. Listing **??**. Similarly, method input() uses an doAllMatches transformation to assign input values to variables. Lines 5 and 6 each look-up the set of all ActivitNode model objects within the current activity. To implement to-many associations SDMLib generates special set classes for all model classes as in this case class ActivityNodeSet. These set classes inherit from a general container class and in addition for each method of the model class SDMLib generates a similar method in the corresponding set class. For example the method withRunning(boolean) of class ActivityNode() results in a similar method in class ActivityNodeSet. In the set class, the generated method iterates through all contained elements and forwards the method call to each of them. Thus, line 5 of Listing **??** is finally calling method withRunning(boolean) on each ActivityNode in the current Activity. This sets the state of all activity nodes to running. Similarly, line 6 sets the noOfVisitors attribute of all activity nodes to 0;

```
1  class Activity {
2      public void run(){
3        this.initVariables();
4        this.input(input);
5        this.getNodes().withRunning(true);
6        this.getNodes().withNoOfVisitors(0);
7
8        ActivityPO activityPO = new ActivityPO(this);
9        ActivityNodePO activityNodePO = activityPO.hasNodes();
10       InitialNodePO initialNodePO = activityNodePO.instanceOf(new InitialNodePO());
11
12       activityPO.createTrace();
13       tokenPO = activityPO.createToken();
14       tokenPO.createCurrentElements(initialNodePO);
15
16       // run the token
17       Token token = tokenPO.getCurrentMatch();
18
19       while ( ! token.getCurrentElements().isEmpty())
```

```
20          {
21              NamedElement first = token.getCurrentElements().first();
22              first.run();
23          }
24
25          this.getNodes().withRunning(false);
26      }
```

Listing 2: Method Activity.run() in Java

Figure 4: Starting Activity.run() transformation

Lines 8 to 14 of Listing **??** build and run the central model transformation employed in method `Activity.run()`. This model transformation is shown graphically in Figure **??**. Again, the Pattern starts with an `activityPO` Pattern Object bound to the current `Activity` model object, cf. line 8. This is extended by a `nodes` link to an `activityNodePO`, cf. line 9. This time we especially look for an activty node of type `InitialNode`. In the current version of SDMLib we have to use a special `instanceOf()` method to model this type check in our Pattern. This results in another Pattern Object of the desired type in line 10. In the graphical visualization this is rendered by an `instanceOf` link to another Pattern Object of the desired type. However, these two Pattern Object will match against the same model object. As this is somewhat intricate, we plan to enhance SDMLib to generate specific `hasNodesOfTypeInitalNode` methods that include the type check, internally.

Once we have identified the initial node, we create a `Trace` object (line 12) and a `Token` object (line 13). Finally, the method call `createCurrentElements(initialNodePO)` creates a `currentElements` link between the model objects matched by `tokenPO` and `initialNodePO` (line 14).

Generally, the described model transformation searches through all nodes of the given activity in order to find the node of type `InitialNode`. This has a runtime complexity of O(n) in the number of activity nodes. However, in the example cases, the initial node is always the first node in the list of activity nodes. Thus, the pattern search always succeeds on the first activity node it visits and thus the actual runtime is O(1).

Once the `Trace` and the `Token` object are created, the actual execution of the activity diagram is driven by lines 17 through 23 of Listing **??**. First, we look up the model object `token` that correspond to the Pattern Object `tokenPO` (line 17). The loop of line 19 uses the `currentElements` link of our `token` object as a queue, it looks-up the first element and calls `run()` on it. The `run` method will remove the corresponding `currentElements` link and add new (successor) elements to the `currentElements` instead. Note, `currentElements` may point to `ActivityNode` objects as well as to `ActivityEdge` objects. Thus, loop variable `first` uses the common super type `NamedElement`.

Method `run()` of class `NamedElement` is overridden within its subclasses to achieve specific behavior for the various activity diagram elements. Listing **??** and Figure **??** show the general behavior of activity nodes.

```
1  class ActivityNode {
2      public void run(){
3          ActivityNodePO activityNodePO = new ActivityNodePO(this);
4
5          // add to trace
```

```
6         TracePO tracePO = activityNodePO.hasActivity().hasTrace();
7         tracePO.createExecutedNodes(activityNodePO);
8
9         // consume token
10        TokenPO tokenPO = activityNodePO.hasToken();
11        tokenPO.destroyCurrentElements(activityNodePO);
12
13        // forward token to all outgoing edges
14        ActivityEdgePO activityEdgePO = forkNodePO.hasOutgoing();
15
16        tokenPO.createCurrentElements(activityEdgePO);
17
18        activityEdgePO.doAllMatches();
19    }
```

Listing 3: Method ActivityNode.run() in Java

Figure 5: General ActivityNode.run() transformation

Generally, the model transformation executing an `ActivityNode` starts with an activityNodePO Pattern Object bound to the model object `this`, cf. line 3 of Listing **??**. Then, line 6 uses a chain of `has` operations to look-up the owning `Activity` and the attached `tracePO`. Line 7 adds the current `ActivityNode` to the `Trace`. Then, we look up the `tokenPO` that is attched to the current `ActivityNode` (line 10) and remove the corresponding `currentElements` link (line 11). Now we forward the token. Thus, line 14 looks for `outgoing activityEdgePO` matches and line 16 adds such `ActivityEdge` objects to the current `Token`. As there may be multiple `outgoing ActivityEdge` objects, line 18 asks the current Pattern to apply on all matches. Thus all `outgoing ActivityEdges` are added to the `currentElements`.

Note, the activity diagrams used as test cases provided by case description have no usual activity nodes that have more than one outgoing control flow. Only, fork nodes and decision nodes have multiple outgoing edges. For fork nodes, the general behavior works fine. For decision nodes, we override the `run()` method and extend the general execution pattern by a check for the guard of the outgoing `ActivityEdge`. Only if the guard is true, the corresponding activity edge is added to the `currentElements`. For decision nodes, it is guaranteed, that only one outgoing control flow has a guard that evaluates to true. Thus, we do not need an `allMatches` for decision nodes. For `JoinNodes` we just extend the general `ActivityNode.run()` pattern with a check whether the `noOfVisitors` equals the number of incomming `ControlFlows`. Only then the `Token` is forwarded.

Listing **??** and Figure **??** show the execution of `ControlFlow` objects. Line 4 starts with a `controlFlowPO` Pattern Object bound to the current `ControlFlow` model object. Line 5 adds the current `tokenPO`. In any case, we destroy the `currentElements` link to the `Token` as the `ControlFlow` is now executed. Now we want to ensure that the guard of the `ControlFlow` allows the execution. Actually, this is not necessary as the decision node does not add a `ControlFlow` to the `currentElements` unless its guard is true. However, for completeness, `ControlFlow.run()` checks this condition, too. Unfortunately, there are two different cases to consider: first the `ControlFlow` may have no guard at all. Then it shall be consider to be true. And second, if the `ControlFlow` has a guard, than the value of that guard has to be true. To cover both cases at once, we ensure that the `ControlFlow` has no guard with value false. This may fail if there is no guard or if the guard is true. If it fails, we move the token forward. In our model transformation we use a negative

application condition *NAC*, cf. line 11 through 18. The sub pattern within the NAC tries to find a match. If that succeeds, the NAC fails and the overall pattern is not executed, any more. Line 13 and 14 look-up a `Guard` at the `controlFlowPO` and test that this `Guard` is an instance of a `BooleanVariable` and that this `BooleanVariable` has a `currentValue`. Line 16 then ensures that the `currentValue` is instance of a `BooleanValue` and that the `BooleanValue` has the `value` `false`.

```
1  public class ControlFlow extends ActivityEdge{
2     @Override
3     public void run(){
4        ControlFlowPO controlFlowPO = new ControlFlowPO(this);
5        TokenPO tokenPO = controlFlowPO.hasToken();
6
7        // in any case remove from currentElements
8        tokenPO.destroyCurrentElements(controlFlowPO);
9
10       // add successor if guard allows
11       controlFlowPO.startNAC();
12
13       ValuePO valuePO = controlFlowPO.hasGuard()
14                              .instanceOf(new BooleanVariablePO()).hasCurrentValue();
15
16       valuePO.instanceOf(new BooleanValuePO()).hasValue(false);
17
18       controlFlowPO.endNAC();
19
20       // OK, move token
21       ActivityNodePO targetPO = controlFlowPO.hasTarget();
22
23       tokenPO.createCurrentElements(targetPO);
24
25       // count visits
26       targetPO.exec((node) -> node.incrementNoOfVisitors(1));
27    }
```

Listing 4: Method ControlFlow.run() in Java

Figure 6: General ActivityNode.run() transformation

If there is no guard preventing it, line 21 of Listing **??** identifies the `target` of our `ControlFlow` and line 23 adds this target to the `currentElements`. Finally, line 26 uses a lambda expression to add an operation to our model transformation that on execution increments the `noOfVisitors` of the `target`.

`OpaqueAction` nodes may have a number of `expressions` attached to them. `Expression` objects provide their own `run()` methods executing them. Thus, for `OpaqueAction` nodes we override the `ActivityNode` `run()` method to call the `Expression.run()` method on each expression. The expressions use various subclasses and various enumeration types to distinguish between different operations. Thus, each subclass provides its specific `run()` method and these specific `run()` methods use traditional switch statements to

deal with the corresponding enumeration types, cf. Listing **??**. Alternatively, we might have provided Model Patterns for each case, however evaluating expression trees is not really the application domain for model patterns.

```
1  public class IntegerCalculationExpression extends IntegerExpression
2  {
3      @Override
4      public void run()
5      {
6          IntegerValue val1 = (IntegerValue) this.getOperand1().getCurrentValue();
7          IntegerValue val2 = (IntegerValue) this.getOperand2().getCurrentValue();
8          int op1 = val1.getValue();
9          int op2 = val2.getValue();
10
11         int result = 0;
12
13         switch (this.getOperator())
14         {
15         case ADD:
16             result = op1 + op2;
17             break;
18
19         case SUBRACT:
20             result = op1 + op2;
21             break;
22
23         default:
24             throw new UnsupportedOperationException(""+this.getOperator());
25         }
26
27         this.getAssignee().setCurrentValue(new IntegerValue().withValue(result));
28     }
```

Listing 5: Method IntegerCalculationExpression.run() in Java

3 Results

Once we decided to come up with our own concept for moving tokens, it was pretty straight forward to develop the corresponding model transformations. The simplified token concept also resulted in model transformations that do very little search through to-many associations. The model transformations mainly look-up the current situation and and check all kinds of conditions on it. Thus, we think the execution is reasonably fast. The following table shows our performance measurements executed on a laptop with a 64 Bit Intel Dual Core i7 CPU M620 2.67GHz with 8 GB memory.

performance test	variant 1	variant 2	variant 3.1	variant 3.2
execution time (milli seconds)	9.99 ms	9.25 ms	9.38 ms	14.05 ms

For the performance measurement we did the usual tricks like warming up the Java virtual machine hot compiler by executing each activity 1000 times before measurement. We than ran each test 5 times and computed the average runtime. Overall, we think the performance test cases are a little bit to small to measure the model transformation execution time without side effects and overheads from other things running in the virtual machine.

4 Summar

Overall, the model execution case fits very well to SDMLib. It was quite straight forward to model the different execution steps and the different steps have a complexity that justifies the usage of model transformation in comparison to hand written Java code. The class model provided with the case uses a lot of inheritance and enumeration types. Actually, SDMLib can still be improved in dealing with inheritance. This is current work. Enumerations are used e.g. for the operators in expression trees. We evaluate such expression trees with usual Java code. Model transformation seem not to give leverage here.