VMTS Solution of Case Study: Leader Election

László Siroki, Tamás Vajk, István Madari, and Gergely Mezei

Budapest University of Technology and Economics, Budapest 1111, Hungary,
vmts@aut.bme.hu,
http://vmts.aut.bme.hu/

Abstract. Model-driven software development makes models and model transformations first-class citizens in software development. As programming is mostly replaced by modeling, verification of model transformation plays an important role in the process. We introduce our approach for validating a model transformation described by graph transformation rules and a control flow language. We present a method that translates the problem into the Prolog logical programming language, and Prolog creates a proof of correctness using its deduction engine.

1 Introduction

Visual Modeling and Transformation System [1] (VMTS) is a general purpose metamodeling environment. VMTS facilitates the creation of metamodels and instance models in a user friendly, visual way. In VMTS, the primary way to transform models is graph rewriting. Our approach is based on two modeling languages: the Visual Control Flow Language (VCFL) and the Visual Transformation Definition Language (VTDL). The activity diagram-like VCFL models control the execution order of the rewriting rules, while the rewriting rules are expressed with VTDL models. These VTDL models define the searched (left hand side, LHS) pattern and its replacement (right hand side, RHS). Currently, the LHS and the RHS patterns are visually merged to a single model. Our transformation engine applies an instance-based matcher with support for constraints. Both the control flow models and the rewriting rules are converted into executable code by model traversing techniques.

2 Problem Interpretation

This paper is based on a case study for the GraBaTs 2009 contest [2]. In the case description, a simple leader election protocol is given that selects the process having the smallest identifier. In the contest, we have to model the protocol using graph transformation rules and validate the protocol by showing that there will never be two processes declaring themselves as leaders. We dealt only with static rings, we did not focus on allowing processes to leave or join the ring. This simple task can be interpreted in many ways. On one hand, we can suppose that the correctness of the protocol is not proved and try to validate it. On the other hand

we can suppose that the protocol itself is correct, but it needs to be proved that our transformation conforms to the protocol. Since the correctness of the protocol is relatively easy to prove by mathematical induction, we have tried to focus on our transformation mapping and show that the output of our transformation is correct. We showed the correctness by checking constraints on the output model.

There is a design decision that affects the complexity and performance of testing and verification considerably. The protocol can be modeled in a fully realistic manner, where the processes emit and process the messages in arbitrary order, independently from each other. This leads to a very high number of possible runs, although the result is the same or similar. In this case, testing all possible variations can take a long time. The other extreme is to focus on the result and to design the transformation in such way that it conforms to the protocol, but not all possible scenarios of the protocol can happen during the execution. We have chosen the later case, but the only restriction used is that each *Process* sends a message, when the election starts. Our approach is a framework in which formal proofs can be created for the correctness of the transformation. This is realized by translating the models, the transformation rules and the control flow into a logical programming (LP) language. We have chosen Prolog, but the idea can be applied to other LP languages as well. We focused only on terminating transformations. Translating the problem into equivalent Prolog representation allows us to answer questions like "From a given input model, what are all the possible outcomes of the transformation?", "Is some given criterion true in all cases?", "What if the input is determined only by some constraints?". These questions can be expressed by simple Prolog queries. Furthermore, if the Prolog predicates of the transformation rules work in both directions, it is possible to determine the input models from the output.

3 Domain and Sample Input Model

The metamodel that represents the problem space contains a *Process*, a *Channel* and a *Message* model item. The *Process* defines a *ProcessId* attribute, an *IsProcessed* attribute that shows if the process has sent its message and an *IsLeader* boolean attribute that should be set to *true* by only one process in the input model by the transformation. *Channel* elements represent the links between processes. Finally, the *Message* model item can be contained by processes or by channels, it holds the *ProcessId* value of its generator *Process*. To use the input model in the Prolog proving process, a translation is needed from the model to Prolog code. As Prolog is a type-free language, the metamodel is only used to define clause names. Each model node is represented by a compound term that contains the name of the meta element; a GUID that identifies the node; and the list of attribute values. An edge between two nodes is expressed with a compound term that contains the edge type; the GUID of the edge; and the GUIDs of the left and right end of the edge. Fig. 1 illustrates the Prolog representation of the above input model. The sample shows that a model is defined by a name, and two lists: list of contained nodes and edges.

```
'LeaderElectionInstance'(
 model( [
  %A Process with ProcessId = 1, IsProcessed = '', IsLeader = false
  'Process'(GUID1,1,'',false),
  'Channel'(GUID2),...
 ],[
 ...
  'EdgeProcessChannel'(GUID3,GUIDLeft,GUIDRight),
 ])).
```

Fig. 1. Sample input model: Prolog representation

4 Control Flow

The Visual Control Flow Language (VCFL) has the following basic elements: Start node, end node, rule container and flow edge. The control flow graph for the case study is depicted on Fig. 2. The icon in the upper right corner of the boxes determines how the rule should be applied: (i) exhaustive mode - the rule will be applied repeatedly until it cannot be applied anymore (ii) normal mode - the rule will be applied at once only. Rule containers reference transformation

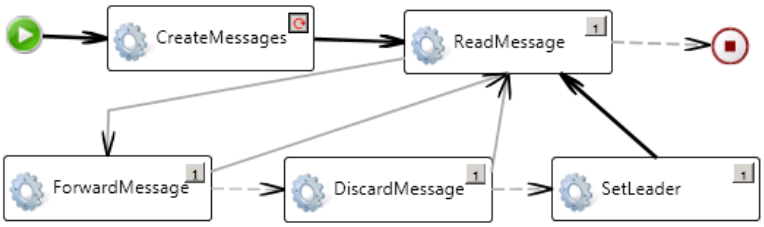

Fig. 2. Control flow graph

rules and have a property that determines whether the rule should be applied only once or exhaustively. In Prolog, from each rule container, one clause of the *'RuleContainer'/3* predicate is created. The arguments of the predicate are the name of the rule container, which is unique in the control flow; the name of the transformation rule to be applied and the number of times the rule should be applied. The flow edges connect the rule containers and determine the order in which the rules are used. The edge has a property which determines when the edge should be followed: if the previous match was successful (thin gray arrow), unsuccessful (thin dashed arrow) or in both cases (thick black arrow). From each flow edge one or two clauses of the *'FlowEdge'/3* Prolog predicate are created. The three arguments are the name of the preceding rule container, the condition, which can be success or failure, and the name of the following rule container. If the edge is marked 'on success only' or 'on failure only', one corresponding clause is generated, otherwise two clauses. Fig. 3 shows an example.

```
...
'FlowEdge'('ForwardMessage',failure,'DiscardMessage').
'FlowEdge'('ForwardMessage',success,'ReadMessage').
...
'RuleContainer'('ForwardMessage','Rule_ForwardMessage',1).
'RuleContainer'('CreateMessages','Rule_CreateMessage',(1,*)).
...
```

Fig. 3. Control Flow Graph: Prolog representation

5 Transformation Rules

We created five rules that implement the protocol, they are depicted on Fig. 4. The white boxes and black edges are elements which are not to be changed, just matched. Gray elements are to be modified, blue elements are to be newly created and red elements are to be deleted on the application of the rule.

The only difference against the task specification is that the message processing consists of two phases: the process reads a message from the incoming channel and processes the message in the next phase. The *Rule_CreateMessage* rule simulates the event when a process sends a message containing its ID. The rule matches a *Process*, its outgoing edge and the *Channel* on the other end, then creates a new *Message*, sets its *ProcessId* to be the same as the *ProcessId* of the matched *Process* and creates a containment edge between the *Channel* and the *Message*. The *Rule_ReadMessage* rule simulates the event when a process reads a message from its incoming *Channel*. This rule matches a *Process*, its incoming

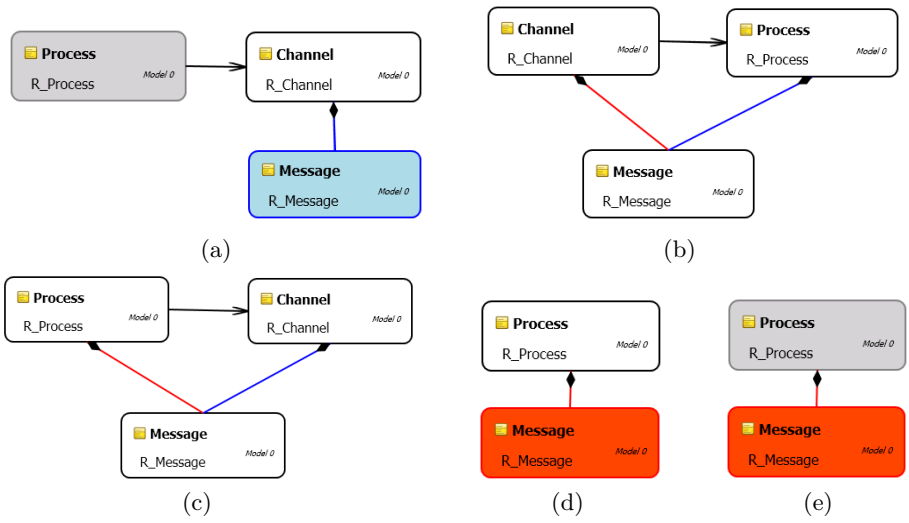

Fig. 4. Transformation rules. (a) Rule_CreateMessage (b) Rule_ReadMessage (c) Rule_ForwardMessage (d) Rule_DiscardMessage (e) Rule_SetLeader

edge and the *Channel* on the other end and a *Message* attached to the *Channel*; then it deletes the containment edge between the *Channel* and the *Message* and creates a new containment edge between the *Process* and the *Message*. The rules *Rule_ForwardMessage*, *Rule_DiscardMessage* and *Rule_SetLeader* are responsible for processing the previously read message. Their application condition investigates the relation between the *ProcessId* of the *Process* and the *Message*. According to the protocol the *Rule_ForwardMessage* forwards the message if its *ProcessId* is less then the *ProcessId* of the *Process*, *Rule_DiscardMessage* discards, if it is greater, and *Rule_SetLeader* sets the current *Process* as the leader, if the *ProcessId*s equal. The rules are translated to Prolog as follows: From each rule, a clause of the predicate *apply_rule/3* is created. The first argument is the previous state of the model as it is described in Section 3, the second is the name of the rule to be applied, the last is the resulting state of the model. The *apply_rule/3* predicate states that from the model described by the first parameter, by applying the rule, we can get the third parameter as the result. Prolog predicates can work in multiple directions. If all arguments are given, Prolog can check if they are consistent. If the input model and the rule is given, it can calculate all possible outputs. If only the input model is given, it can calculate all applicable rules and all possible results while applying one of the rules. If the realization of the predicate allows, the input model can be also the output, given the output model and optionally the rule to be applied. The latter requires the rule to be reversible, which narrows the range of involved calculations (i.e. a difficult-to-invert function cannot be used to calculate values on the RHS). Fig. 5 shows the Prolog code generated from the *Rule_CreateMessage*. Lines 3 to 8 are matching the LHS in the input graph, line 6 is an application condition which ensures that the selected Process has not sent its message yet, from line 10, the new graph is constructed.

```
1  apply_rule(PrevState, 'Rule_CreateMessage', NextState):-
2    PrevState=model(PNodes0, PEdges0),
3    member('Channel'(R_CHANNEL_Id, R_CHANNEL_Attributes_IsProcessed), PNodes0),
4    member('EdgeProcessChannel'(E_PC_Id, R_PROCESS_Id, R_CHANNEL_Id), PEdges0),
5    %Application Condition
6    R_PROCESS_Attributes_IsProcessed="",
7    select('Process'(R_PROCESS_Id, R_PROCESS_Attributes_ProcessId,
8     R_PROCESS_Attributes_IsProcessed, R_PROCESS_Attributes_IsLeader), PNodes0, PNodes1),
9    %Imperative Code for modifications
10   R_MESSAGE_Attributes_ProcessId_new=R_PROCESS_Attributes_ProcessId,
11   R_PROCESS_Attributes_IsProcessed_new="Processed",
12   NextState=model(NNodes, NEdges), gensym(guid,E_CM_Id),  gensym(guid,R_MESSAGE_Id),
13   NNodes=['Message'(R_MESSAGE_Id,R_MESSAGE_Attributes_ProcessId_new),
14    'Process'(R_PROCESS_Id, R_PROCESS_Attributes_ProcessId,
15    R_PROCESS_Attributes_IsProcessed_new, R_PROCESS_Attributes_IsLeader)|PNodes1],
16   NEdges=['EdgeChannelMessage'(E_CM_Id, R_CHANNEL_Id, R_MESSAGE_Id)|PEdges0].
```

Fig. 5. Graph Transforamtion Rule: Prolog representation

6 Results

After translating the input model, the rules and the control flow graph, a small Prolog program can be used to validate the result of the transformation. The Prolog code needed for running the transformation can be found in the Appendix [1]. We have created two queries: (i) the first one gives all possible transformation runs. Steps contain the executed steps in order, *Nodes* and *Edges* contain the nodes and edges of the resulting graph, respectively. The member predicate selects the leader *Process* from the nodes; (ii) the second one adds a new condition to the previous, representing a question whether there is a case when the leader has an ID larger than one. The answer for this query is "false.", meaning that the transformation is working correctly in spite of the nondeterminism.

Exhaustively checking all possible outcomes of the transformation with Prolog leads to stack overflow at higher model sizes. The size limit depends on the complexity of the input model, the control flow graph and the capabilities of the Prolog engine in use. With the given transformation and using SWI-Prolog, the solution works up to 4 processes if the processes are in ascending order of ProcessIDs, and works up to 6 if the order is descending. The order determines the number of possible transformations due to nondetermism. The order of the IDs are significant because of the number of steps a message can take. If the order is descending, every message is discarded at next process, except the message with the smallest ID, which is the only one which is forwarded. In the other case, when the order is ascending, all messages are forwarded until they reach the process with the smallest ID, which will discard them. That is why there is a difference between the number of possible transformation runs (Table 1).

	ascending		descending	
No. of processes	time	possibilities	time	possibilities
3	0.1 s	60	0.1 s	20
4	5.5 s	12600	0.2 s	210
5	out of stack	37837800	2 s	3024
6	out of stack		34 s	55440
7	out of stack		too much	1235520

Table 1. Performance

References

1. VMTS Team, "Visual Modeling and Transformation System website." http://vmts.aut.bme.hu/.
2. "GraBaTs Tool Contest." http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/.