A GrGen.NET solution of the Model Migration Case for the Transformation Tool Contest 2010

Sebastian Buchwald Edgar Jakumeit

June 3, 2010

1 Introduction

The challenge of the Model Migration Case [1] is to migrate an activity diagram from UML 1.4 to UML 2.2. We employ the general purpose graph rewrite system GrGen.NET (www.grgen.net) to tackle this task. After a short description of the GrGen.NET system, we give an introduction into our solution of the core assignment, followed by a discussion of the various extensions proposed by the case authors. Finally, we conclude.

2 What is GrGen.NET?

GrGen.NET is an application domain neutral graph rewrite system developed at the IPD Goos of Universität Karlsruhe (TH), Germany [2]. The feature highlights of GrGen.NET regarding practical relevance are:

Fully Featured Meta Model: GrGen.NET uses attributed and typed multigraphs with multiple inheritance on node/edge types. Attributes may be typed with one of several basic types, user defined enums, or generic set and map types.

Expressive Rules, Fast Execution: The expressive and easy to learn rule specification language allows straightforward formulation of even complex problems, with an optimized implementation yielding high execution speed at modest memory consumption.

Programmed Rule Application: GrGen.NET supports a high-level rule application control language, Graph Rewrite Sequences (GRS), offering logical, sequential and iterative control plus variables and storages for the communication of processing locations between rules.

Graphical Debugging: GrShell, GrGen.NET's command line shell, offers interactive execution of rules, visualising together with yComp the current graph and the rewrite process. This way you can see what the graph looks like at a given step of a complex transformation and develop the next step accordingly. Or you can debug your rules.

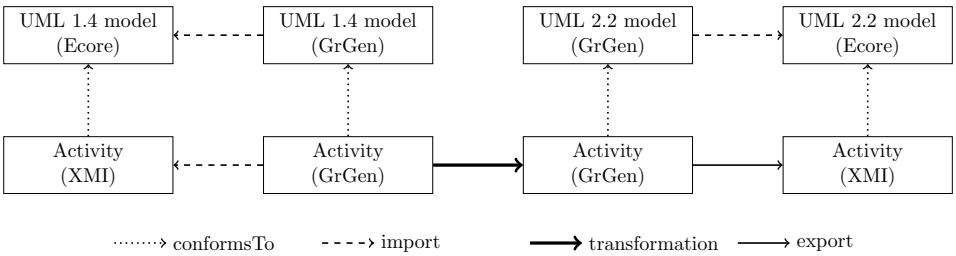

Figure 1: Processing steps of the model migration. The transformation and the XMI export are written in GrGen.NET languages. Import is handled by a supplied import filter, which generates .gm files as an intermediate step.

3 The Core Assignment

The task of the core assignment is to migrate an activity diagram conforming to an UML 1.4 metamodel to a semantically equivalent activity diagram conforming to an UML 2.2 metamodel. The aim of the task is to evaluate the solutions regarding correctness, conciseness and clarity, or better to evaluate the participating tools in how far they allow for such solutions. Before the transformation can take place, the activity diagram needs to be imported from an Ecore file describing the source model and an XMI file specifying the graph. Afterwards the resulting activity diagram has to be exported into an XMI file conforming to a given Ecore file describing the target model.

3.1 Importing the Graph

As GrGen.NET is a general purpose graph rewrite system and not a model transformation tool, we do not support importing Ecore metamodels directly (we directly support GXL and GRS files). Instead we offer an import filter generating an equivalent GrGen.NET-specific graph model (.gm file) from a given Ecore file by mapping classes to GrGen node classes, their attributes to corresponding GrGen attributes, and their references to GrGen edge classes. Inheritance is transferred one-to-one, and enumerations are mapped to GrGen enums. Class names are prefixed by the names of the packages they are contained in to prevent name clashes. Afterwards the instance graph XMI adhering to the Ecore model thus adhering to the just generated equivalent GrGen graph model is imported by the filter into the system to serve as the host graph for the following transformations. The entire process is shown in Figure 1 above.

3.2 Transformation

The transformation is done in several passes, one pass for each node type and edge type, respectively. Each pass consists of the iterated application

of one single rule which matches a node or edge of an UML 1.4 type to process and rewrites it to its corresponding UML 2.2 target type. Instead of handling the edges together with the nodes we process them separately. This relieves us from taking care of the multiplicities of the incoming or outgoing edges, which allows for a simple solution built from very simple rules. This approach is possible because of the availability of a retype operator which allows to change the type of a node (edge) keeping all incident edges (nodes).

We start the detailed presentation of the solution with the following example rule:

```
rule transform_ActionState {
    state:minuml1_ActionState;

    modify {
        opaque:uml_OpaqueAction<state>;

        eval { opaque._name = state._name; }
    }
}
```

Rules in GrGen consist of a pattern part specifying the graph pattern to match and a nested rewrite part specifying the changes to be made. The pattern part is built up of node and edge declarations or references with an intuitive[1] syntax: Nodes are declared by `n:t`, where `n` is an optional node identifier, and `t` its type. An edge `e` with source `x` and target `y` is declared by `x -e:t-> y`, whereas `-->` introduces an anonymous edge of type `Edge`. Nodes and edges are referenced outside their declaration by `n` and `-e->`, respectively. The rewrite part is specified by a `modify` block nested within the rule. Usually, here you would add new graph elements or delete old ones, but in this case we only want to retype them (also known as relabeling in the graph rewriting community). Retyping is specified with the syntax `y:t<x>`: this defines `y` to be a retyped version of the original node `x`, retyped to the new type `t`; for edges the syntax is `-y:t<x>->`. These and a lot more language elements are described in more detail in the extensive GrGen.NET user manual [2].

In the example a node `state` of type `ActionState` (mind the package name mangling) is specified in the pattern part to get matched. In the rewrite part it is specified to get retyped to a node `opaque` of type `OpaqueAction`. Furthermore the `name` attribute of the original node is assigned to the `name` attribute of the new, retyped node in the attribute evaluation `eval`.

[1] it was used in the discussion forum for this case as textual notation to describe the patterns

Most of the rules are as easy as this one. Only for a few types, the rewriting additionally depends on further local information or the context where the graph element appears in. An example for further local information to be taken into account is the rule for the transformation of the `Pseudostate` nodes. An `alternative` construct is used here (namespace prefixes were removed due to space constraints):

```
rule transform_PseudoState {
    state:Pseudostate;

    alternative {
      Initial {
        if { state._kind == PseudostateKind::_initial; }
        modify {
            initial:InitialNode<state>;
            eval { initial._name = state._name; }
        }
      }
      Join {
        if { state._kind == PseudostateKind::_join; }
        modify {
            join:JoinNode<state>;
            eval { join._name = state._name; }
        }
      }
      Fork { /* similar to the cases above */ }
      Junction { /* similar to the cases above */ }
    }

    modify {
    }
}
```

There are four possible target types for the source type, so four different alternative cases are specified, each relabeling to one of the types in their nested rewrite part. The correct type depends on the kind value of the source node; this condition is checked by an attribute condition given within the `if`-clause (fitting to the rewrite).

An example for context dependency is the rewriting of the `Transition` nodes. If they are linked to a node of `ObjectFlowState` (rewritten to `Pin`), they get retyped to nodes of type `ObjectFlow`, otherwise to nodes of type `ControlFlow`. This is expressed again with an `alternative` statement as you can see below, with a case for the transformation to control flow, preventing by negative patterns that it matches on the object flow situation, and two almost identical cases for the incoming and outgoing object flow.

```
rule transform_Transition {
    transition:Transition;

    alternative {
      controlFlow {
        negative {
           transition <-:StateVertex_incoming- :uml_Pin;
        }
        negative {
           transition <-:StateVertex_outgoing- :uml_Pin;
        }
        modify {
            cf:uml_ControlFlow<transition>;
            eval { cf._name = transition._name; }
        }
      }
      incomingObjectFlow {
        transition <-:StateVertex_incoming- :uml_Pin;
        modify {
            of:uml_ObjectFlow<transition>;
            eval { of._name = transition._name; }
        }
      }
      outgoingObjectFlow { /* similar to case above */ }
    }

    modify {
    }
}
```

As a final example for the transformation core let us have a look at one of the rules for the retyping of the edges – they follow the GrGen design target of handling nodes and edges as uniform as possible: they are nearly identical to the rules for the retyping of the nodes:

```
rule transform_StateMachine_transitions {
    -e:minuml1_StateMachine_transitions->;

    modify {
        -:uml_Activity_edge<e>->;
    }
}
```

The four shown rules are applied from within the graph rewrite script for the core solution executed by the GrShell:

```
import original_minimal_metamodel.ecore
       evolved_metamodel.ecore
       original_model.xmi core.grg
# Transform nodes
xgrs ... | transform_ActionState* | transform_PseudoState*
         | transform_Transition* | ...
# Transform edges
debug xgrs ... | transform_StateMachine_transitions* | ...
```

The `xgrs` keyword starts an extended graph rewrite sequence, which is the rule application control language of GrGen (prepending `debug` before `xgrs` allows you to debug the sequence execution in GrShell). The single rules are applied iteratively by the star operator until no match is found. They are linked by eager ors which get executed from left to right, yielding the disjunction of the truth values emanating from iteration execution (a rule which can get applied because a match is found in the graph succeeds(`true`), whereas a rule for which no match is found fails(`false`); the star operator always succeeds).

Overall our solution complies to the following scheme:

- For each node or edge type there is one rule relabeling an element of this type, often containing nothing more than this relabeling, sometimes using alternatives to decide between possible target types depending on the context.
- Each of this rules gets applied exhaustively, one rule after the other; first handling all node types, then handling all edge types (thus a few context dependent rules match against nodes/edges of types from the source and target model).

The modulare nature of this approach facilitates extensions regarding the support of additional UML elements and the realization of alternative semantics (see section 4). (Please note that it would be possible to shrink the number of rules down to one to be applied iteratively using an alternative with a lot of cases; or by using strings instead of types to encode the model types, transforming them by string replacement using map types and map lookup. But we prefer to give straight forward real-world solutions.)

3.3 Exporting the Graph

Our XMI exporter consists of several graph transformation rules that traverse the graph hierarchically while emitting the corresponding XMI tags. The following rule exports an activity:

```
rule emit_activity {
  activity:uml_Activity;
  activity -:DumpEdge-> d:DumpNode;

  modify {
    emit("  <packagedElement xmi:type=\"uml:Activity\"");
    emit(" xmi:id=\"" + d.id + "\"");
    emit(" name=\"" + activity._name + "\">\n");
    exec(emit_activity_nodes(activity));
    exec(emit_activity_edges(activity));
    exec(emit_activity_groups(activity));
    emit("  </packagedElement>\n");
  }
}
```

Since we need a unique ID for each node, we connect each node to a `DumpNode` node that provides such an ID. The `emit` statements emit the given strings; here they fill up the template of the XMI tag with node attributes. The rules executed by the `exec` statement are responsible for emitting all nodes, edges, and groups of the current activity, respectively.

4 The Extensions

In addition to the core task three extensions were proposed:

- Alternative Object Flow State Migration Semantics
- Concrete Syntax
- XMI

We will discuss them and our solutions to them in the following sections.

4.1 Alternative Object Flow State Migration Semantics

The goal of this extension is to transform a node of type `ObjectFlowState` linked to nodes of type `Transition` to a node of type `ObjectFlow` only, instead of transforming them to a node of type `Pin` linked to nodes of type `ObjectFlow`. The purpose of this task is to evaluate how well the transformation tools can cope with transformations which do not map source pattern elements injectively to target pattern elements. Or to put it in another way: require real graph rewriting instead of only graph relabeling. As GrGen.NET is a graph *rewrite* system in the first place, this does not cause any problems:

```
rule transform_ObjectFlowState2 {
  state:ObjectFlowState;
  s1:StateVertex <-:Transition_source- t1:Transition;
  t1 -:Transition_target-> state;
  state <-:Transition_source- t2:Transition;
  -:Transition_target-> s2:StateVertex;
  state <-:Partition_contents- p:Partition;

  modify {
    delete(state, t2);

    flow:ObjectFlow<t1>;
    flow -:ActivityEdge_target-> s2;
    flow <-:ActivityNode_incoming- s2;
    flow -:ActivityEdge_inPartition-> p;
    flow <-:ActivityPartition_edge- p;

    eval { flow._name = state._name; }
  }
}
```

4.2 Concrete Syntax

The goal of this extension is to transform the concrete syntax, i.e. the user drawn diagram layout, from the given diagram to the concrete syntax of the tool under consideration. As GrGen.NET was originally developed for handling compiler intermediate language graphs which do not possess a user drawn layout we do not offer a concrete visual syntax showing user editing. So we cannot transform any concrete syntax — but as the ultimate goal of a concrete syntax is a nice layout, we want to present a solution of a different kind: we offer a highly customizable graph viewer with automatic layout – which comes near to the concrete syntax of the activity diagram given as you may see in Figure 2.

In a lot of situations an automatic layout is the better choice as it delivers a very nice presentation without user interaction (besides a few lines of configuration code). You can configure the graph viewer named yComp which gets executed from GrShell to use one of several available layout algorithms – with hierarchic, compilergraph and organic being the most useful ones. You can configure for every available node or edge type in which colour with what node shape or edge style it should be shown, with what attribute values or fixed text as element labels or tags it is to be displayed, or if it should be shown at all. Further on you can configure graph nesting by registering edges at certain nodes to define a containment hierarchy, causing the nodes to become displayed as subgraphs containing

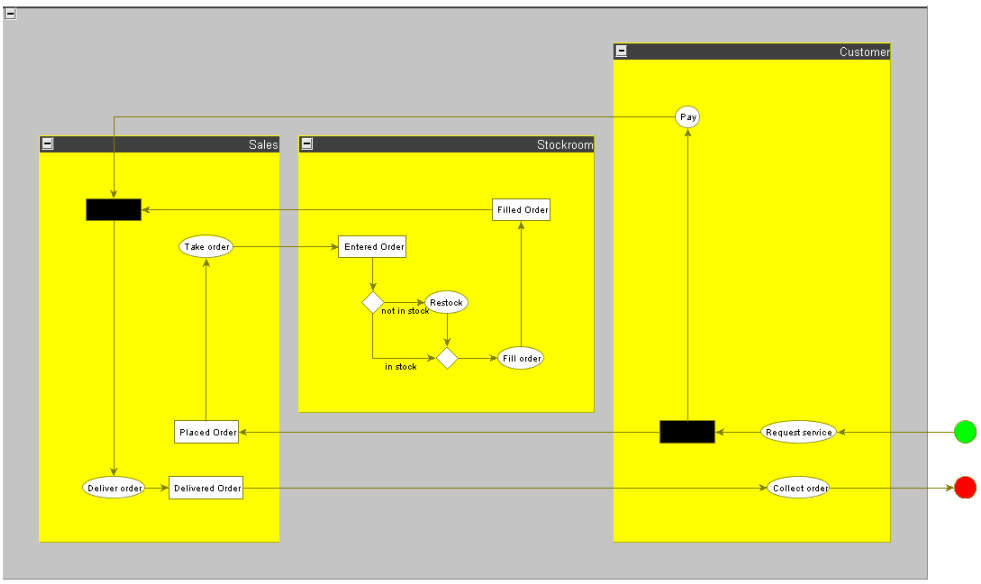

Figure 2: The transformed UML 2.2 activity diagram as displayed by yComp

the elements to which they are linked by the given edges. Additionally it offers automatic cutting of hierarchy crossing edges, marking the begin and end by fat dots, allowing to jump to either one by clicking on the other. You can easily define a layout matching your graph class by a few dozen lines of configuration information and afterwards you get a fully automatic, high-quality layout of your instance graphs.

In the following listing, we show an excerpt from our configuration file for customizing the graph layout of the UML 2.2 diagram (the first two lines ensure that each `ActivityPartition` node contains all nodes that are connected by an outgoing `uml_ActivityPartition_node` edge):

```
dump add node uml_ActivityPartition group by
          hidden outgoing uml_ActivityPartition_node
dump set node uml_ActivityPartition labels off
dump add node uml_ActivityPartition shortinfotag _name

dump add edge uml_ActivityNode_inPartition exclude

dump set node uml_DecisionNode shape rhomb
dump set node uml_DecisionNode labels off
dump set node uml_DecisionNode color white
```

4.3 XMI

The goal of this extension is to import an activity diagram given in XMI 1.x instead of XMI 2.x. The task is to write an import filter for an outdated format used in the model transformation community. While we did write an import filter for XMI 2.x (a slightly extended version of the filter originally introduced for the GraBaTs 2009 Reverse Engineering case [3]), we will not write a filter for XMI 1.x. The XMI 2.x filter allows to use the transformation capabilities of GrGen.NET with data in the Ecore/XMI format common to the model transformation community briding the graph rewriting and the model transformation communities; but supplying another Ecore/XMI filter just for the sake of this contest is beyond our scope. And we think it is out of scope even for the TTC as such: we doubt writing an import filter is a worthwhile challenge for a transformation tool contest comparing the transformation capabilities of the competing tools in order to foster the progress in software engineering. If it really were, we would like to propose to the authors from the model transformation community to follow our example bridging both worlds by writing an import filter for GXL, the standard in the graph rewriting community.

5 Conclusion

In this paper we presented a GrGen.NET solution to the Model Migration challenge of the Transformation Tool Contest 2010. The activity diagram conforming to the UML 1.4 metamodel was imported by an import filter under remapping to the graph concepts supported by GrGen. It was transformed to a semantically equivalent activity diagram conforming to an UML 2.2 metamodel using graph *relabeling*: this ability of retyping nodes (edges) while keeping their incident edges (nodes) allowed us to give a very concise and simple solution to the core task of the Model Migration challenge, exhaustively relabeling nodes then edges with very simple rules until the entire graph was transformed. Retyping of elements from the source model to different target types depending on further, context information was possible by using alternatives in our patterns. The first extension requiring real graph *rewriting* was solved easily with one additional declarative graph rewrite rule, in an intuitive syntax similar to the one specified by the authors. The second extension was not tackled directly due to the lack of a concrete syntax; but we presented an alternative solution (performing even better in a lot of cases) regarding the ultimate goal of a concrete syntax with our graph viewer yComp, delivering an excellent automatic layout of arbitrary data from a few lines of configuration information. The third extension was not tackled at all as we regard it off-topic at least for us.

References

[1] Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.: Model Migration Case for TTC 2010. http://is.ieis.tue.nl/staff/pvgorp/events/TTC2010/cases/ttc2010_submission_2_v2010-04-22.pdf (2010)

[2] Blomer, J., Geiß, R., Jakumeit, E.: The GrGen.NET User Manual. http://www.grgen.net (2010)

[3] Buchwald, S., Jakumeit, E., Kroll, M.: A GrGen.NET solution of the Program Comprehension case for the GraBaTs 2009 Contest (2009) http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/submissions/grabats2009_submission_13-final.pdf.