Saying Hello World with VIATRA2 - A Solution to the TTC 2011 Instructive Case*

Ábel Hegedüs Zoltán Ujhelyi Gábor Bergmann

Fault Tolerant Systems Research Group
Department of Measurement and Information Systems
Budapest University of Technology and Economics, Hungary

hegedusa@mit.bme.hu ujhelyiz@mit.bme.hu bergmann@mit.bme.hu

The paper presents a solution of the *Hello World! An Instructive Case for the Transformation Tool Contest* using the VIATRA2 model transformation tool.

1 Introduction

Automated model transformations play an important role in modern model-driven system engineering in order to query, derive and manipulate large, industrial models. Since such transformations are frequently integrated into design environments, they need to provide short reaction time to support software engineers.

The objective of the VIATRA2 (VIsual Automated model TRAnsformations [9]) framework is to support the entire life-cycle of model transformations consisting of specification, design, execution, validation and maintenance.

Model representation. VIATRA2 uses the VPM (Visual and Precise Metamodeling) approach [8] for describing modeling languages and models. The main reason for selecting VPM instead of a MOF-based metamodeling approach is that VPM supports arbitrary metalevels in the model space. As a direct consequence, models taken from conceptually different domains (and/or technological spaces) can be easily integrated into the VPM model space. The flexibility of VPM is demonstrated by a large number of already existing model importers accepting the models of different BPM formalisms, UML models of various tools, XSD descriptions, and EMF models.

Graph transformation (GT) [3] based tools have been frequently used for specifying and executing complex model transformations. In GT tools, *graph patterns* capture structural conditions and type constraints in a compact visual way. At execution time, these conditions need to be evaluated by *graph pattern matching*, which aims to retrieve one or all matches of a given pattern to execute a transformation rule. A *graph transformation rule* declaratively specifies a model manipulation operation, that replaces a match of the LHS (left-hand side) graph pattern with an image of the RHS (right-hand side) pattern.

Transformation description. Specification of model transformations in VIATRA2 combines the visual, declarative rule and pattern based paradigm of graph transformation and the very general, high-level formal paradigm of abstract state machines (ASM) [2] into a single framework for capturing transformations within and between modeling languages [7]. A transformation is defined by an ASM machine that may contain ASM rules (executable command sequences), graph patterns, GT rules, as well as ASM functions for temporary storage. An optional main rule can serve as entry point. For model manipulation and pattern matching, the transformation may rely on the metamodels available in the VPM model space; such references are made easier by namespace imports.

*This work was partially supported by ICT FP7 SecureChange (ICT-FET-231101) European Project.

Van Gorp, Mazanek and Rose (Eds.):
Fifth Transformation Tool Contest (TTC 2011)
EPTCS 74, 2011, pp. 302–324, doi:10.4204/EPTCS.74.25

Transformation Execution. Transformations are executed within the framework by using the VIATRA2 interpreter. For pattern matching both (i) *local search based pattern matching* (LS) and (ii) *incremental pattern matching* (INC) are available. This feature provides the transformation designer additional opportunities to finetune the transformation either for faster execution (INC) or lower memory consumption (LS) [5].

2 Transformation tasks

The VIATRA2 framework has been applied to the subtasks of the *Hello World!* case [6] using the VIATRA2 Textual Command Language (VTCL) [1]. Since the case was intended to provide beginners with instructive transformations, we decided to show as many features of the VIATRA2 framework as possible, while also keeping each example as simple as possible. Therefore, each task was solved by multiple variants, which may share pattern definitions, but are otherwise different from each other. The differences are explained at the description of each task.

2.1 Hello World!

This task included very basic model construction, model-to-model and model-to-text transformations. Since VTCL combines two formalisms for specifying graph transformations, we implemented the different parts of the task with both declarative ASM rules (see Listing 1) and graph transformation rules (see Listing 2). The difference is how much of the model manipulation is described declaratively by GT rules (composed of a LHS and RHS pattern, and potentially an additional ASM action), as opposed to elementary manipulation operations issued in ASM rules; see the homepage[1] for advice on choosing between the two alternative approaches in practice. Although the task seems almost trivial, the accompanying transformation definition is not especially short. This may be taken as a disadvantage of VTCL, but we would like to note, that the we feel that the verbose self-descriptive nature of VTCL aids in comprehension.

2.2 Count Matches with certain Properties

This task included parts, where the number of matches are counted using transformations. In the ASM variant (see Listing 4), the counting and matching are clearly separated, since the patterns describe what we look for and the forall construct iterates through the matches to count them.

To demonstrate reusability and modularity in VTCL, each part reuses the same ASM rule for creating the result structure. Additionally, the solution references graph patterns defined externally in a separate VTCL file (see Listing 3), which acts as a library of common graph patterns. Several further solutions also reuse these patterns. In each case, the VTCL machine corresponding to the library must be loaded first.

The verbosity, once again, is partly voluntary: many of the statements in the graph patterns of Listing 3 merely assert the type of nodes, and could be safely omitted thanks to type inference. While such verbosity aids in understanding the graph pattern, it does not neccessarily place any additional burden on the developer, as patterns like these can easily be created by selecting some related elements in the model and then exporting their configuration as a graph pattern.

[1] `http://wiki.eclipse.org/VIATRA2/UseCases/TransformationBestPractices`

The upcoming match counting feature of VIATRA2 is still under development (and currently only partially supported), but it will allow for a more elegant solution (see Listing 5) of this task. We expect that this solution (and other transformations using match counting within a graph pattern) will be fully supported in release 3.3 of VIATRA2.

2.3 Reverse Edges

In this task, the edges of the graph had to be reversed by the transformation. As before, we created both an ASM (see Listing 6) and a GT rule variant (see Listing 7). However, here the ASM variant shows an interesting feature of VIATRA2, as the relations are not modified, only their type is changed from *src* to *trg*, and the other way around. Furthermore, we implemented a third variant (see Listing 8) that reverses the edges by switching the target of the *src* relation with the target of the *trg* relation.

Note that using the appropriate conditional language elements (`if`, `try`), our solutions are tolerant of dangling edges. By ignoring this possibility, the transformation could have been made somewhat simpler.

2.4 Simple Migration

In this task, the input graph is transformed to a graph conforming to another metamodel. As the case description did not specify it, we implemented both a copy (see Listing 9 and Listing 11) and an in-place (retyping) variant (Listing 10 and Listing 12) for both the core and the topology changing transformation. The copy variants simply create the graph using the other metamodel, while the in-place variants use the above mentioned feature of VIATRA2 and change the type of the elements without modifying the rest of the model.

2.5 Delete Node with Specific Name and its Incident Edges

This task included delete transformations for one specific node and it's incident edges. We implemented an ASM (Listing 13 and Listing 15) and a GT variant (Listing 14 and Listing 16) for both the core task and the optional task.

2.6 Insert Transitive Edges

Finally, the last task dealt with inserting transitive edges between nodes. In this case we provide three versions, each with two implementations using ASM rules and GT rules.

- First, closely following the original problem specification, we insert edges between nodes that are 2-hop reachable through an inner node (see Listing 17 and Listing 18), i.e. if there are two nodes that are not connected directly, but through an intermediate node, we establish a direct connection between them.
- In the next version, we iterate this step as long as applicable so that eventually all transitive reachability edges are inserted (see Listing 19 and Listing 20). Note, that although there is only a slight difference in the code of the first two versions, they are independent in implementation.
- Finally, we present a solution where all missing transitive reachability edges are detected and inserted in a single step (see Listing 21 and Listing 22). The transformation relies on a graph pattern that expresses full transitive reachability in the graph, using the recursive pattern definition feature of VIATRA2. One could even argue that actually inserting the missing reachability edges is not necessary in many cases if such pattern matching capability is at our disposal. Note that

due to the recursive nature of the pattern, the current version of the incremental pattern matcher would not work correctly (in case of deleting edges from graphs containing cycles), therefore the default local search-based graph pattern matcher is used. For the other tasks and solution variants the incremental pattern matcher is used.

3 Conclusion

In the current paper we have presented our VIATRA2 based implementation for the Hello World! case study [6].

The high points of our solution are (i) the different variants that are self-descriptive and instructive, (ii) the reusable patterns, (iii) the support for recursive matching and (iv) match counting. Furthermore, the dynamic type modification (metamodel manipulation in general) is a highly usable feature especially for migration problems.

On the other hand, since VIATRA2 does not handle EMF models natively, importing and exporting of models is required. Furthermore, as the transformation language is quite verbose, transformations may appear more complex than they really are (note however, that conciseness was not our primary goal when creating these instructive examples).

Our overall impression is that this simple case study is an excellent basis of comparison of various model transformation tools, filling a long-standing gap.

References

[1] András Balogh & Dániel Varró (2006): *Advanced Model Transformation Language Constructs in the VIATRA2 Framework*. In: *ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006)*, ACM Press, Dijon, France, pp. 1280–1287, doi:10.1145/1141277.1141575.

[2] E. Börger & R. Stärk (2003): *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag.

[3] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors (1999): *Handbook on Graph Grammars and Computing by Graph Transformation*. 2: Applications, Languages and Tools, World Scientific.

[4] Ábel Hegedüs, Zoltán Ujhelyi & Gábor Bergmann (2011): *SHARE demo related to the paper Saying Hello World with* VIATRA2 *- A Solution to the TTC Instructive Case*. Available at `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu-11_TTC11_VIATRA.vdi`.

[5] Ákos Horváth, Gábor Bergmann, István Ráth & Dániel Varró (2010): *Experimental assessment of combining pattern matching strategies with VIATRA2*. *International Journal on Software Tools for Technology Transfer (STTT)* 12, pp. 211–230, doi:10.1007/s10009-010-0149-7.

[6] Steffen Mazanek (2011): *Hello World! An Instructive Case for the Transformation Tool Contest*. In Pieter Van Gorp, Steffen Mazanek & Louis Rose, editors: *TTC 2011: Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011*, EPTCS.

[7] Dániel Varró & András Balogh (2007): *The Model Transformation Language of the VIATRA2 Framework*. *Science of Computer Programming* 68(3), pp. 214–234, doi:10.1016/j.scico.2007.05.004.

[8] Dániel Varró & András Pataricza (2003): *VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML*. *Journal of Software and Systems Modeling* 2(3), pp. 187–210, doi:10.1007/s10270-003-0028-8.

[9] VIATRA2 Framework: An Eclipse GMT Subproject: Available at `http://www.eclipse.org/gmt/`.

A Solution demo and implementation

The deployable implementation and source code is available as an Eclipse online update site (`http://mit.bme.hu/~ujhelyiz/viatra/ttc11/`) and the project including the transformations as an archive (`http://mit.bme.hu/~ujhelyiz/viatra/ttc11-helloworld.zip`)

The SHARE image [4] usable for demonstration purposes contains our solution for both the Hello World! and Program Understanding cases.

B Appendix - Hello World! transformations

B.1 Hello World!

```
import datatypes; // imported parts of the model-space are usable by local name
import nemf.packages;
import nemf.ecore.datatypes;

@incremental // uses incremental pattern-matcher
machine helloWorldASM{
  rule main() = seq{
    println("2.1 Hello World transformation started");

    println("Creating Simple Model with ASM Rule");
    call createSimpleModelInstance(); // invokes the ASM Rule
    let Greeting = undef in seq{ // define local variable
      println("Creating Extended Model with ASM Rule");
      call createExtendedModelInstance(Greeting);
      println("Executing model-to-text with ASM Rule");
      call outputGreeting(Greeting);
    }

    println("2.1 Hello World transformation finished");
  }

  // ASM Rule variant of simple Hello World model instance creation
  rule createSimpleModelInstance() =
   let Greeting = undef, Text = undef, TextRelation = undef in seq{
    // entity creation with explicit parent (in)
    new(helloworld.Greeting(Greeting) in nemf.resources);

    new(EString(Text) in Greeting);
    // setting entity value to some primitive datatype value
    setValue(Text,"Hello world");
    // create relation between elements
    new(helloworld.Greeting.text(TextRelation,Greeting,Text));
  }

  // ASM Rule variant of extended Hello World model instance creation
  rule createExtendedModelInstance(out Greeting) =
   let GreetingMessage = undef, GreetingMessageRelation = undef,
   Text = undef, TextRelation = undef, Person = undef,
    PersonRelation = undef, Name = undef, NameRelation = undef in seq{
    new(helloworldext.Greeting(Greeting) in nemf.resources);

    new(helloworldext.GreetingMessage(GreetingMessage) in Greeting);
    new(helloworldext.Greeting.greetingMessage(GreetingMessageRelation,
     Greeting,GreetingMessage));
    new(EString(Text) in GreetingMessage);
    setValue(Text,"Hello");
    new(helloworldext.GreetingMessage.text(TextRelation,GreetingMessage,Text));
```

```
    new(helloworldext.Person(Person) in Greeting);
    new(helloworldext.Greeting.person(PersonRelation,Greeting,Person));
    new(EString(Name) in Person);
    setValue(Name,"TTC Participants");
    new(helloworldext.Person.name(NameRelation,Person,Name));
  }

  // ASM Rule variant of model-to-text transformation
  rule outputGreeting(in Greeting) = let Output = undef, ResR = undef,
   Result = undef in seq{
    /* parameters of "choose" are set by the patternmatcher
     based on matches to the patterns after "find" */
    try choose GreetingMessageText,PersonName with
     find TextAndNameForGreeting(Greeting,GreetingMessageText,PersonName) do seq{
      new(result.StringResult(Output) in nemf.resources);
      new(EString(Result) in Output);
      new(result.StringResult.result(ResR,Output,Result));
      // value can be set baseed on values from other elements
      setValue(Result,(value(GreetingMessageText) + " " + value(PersonName) + "!"));
    }
  }

  // finds (or creates) Greeting, GreetingMessage.Text and Person.Name
  pattern TextAndNameForGreeting(Greeting,Text,Name) = {

    helloworldext.Greeting(Greeting) in nemf.resources;

    helloworldext.GreetingMessage(GreetingMessage);
    helloworldext.Greeting.greetingMessage(GreetingMessageRelation,
     Greeting,GreetingMessage);
    EString(Text);
    helloworldext.GreetingMessage.text(TextRelation,GreetingMessage,Text);

    helloworldext.Person(Person);
    helloworldext.Greeting.person(PersonRelation,Greeting,Person);
    EString(Name);
    helloworldext.Person.name(NameRelation,Person,Name);
  }
}
```

Listing 1: Hello World transformation, ASM variant

```
import datatypes; // imported parts of the model-space are usable by local name
import nemf.packages;
import nemf.ecore.datatypes;

@incremental // uses incremental patternmatcher
machine helloWorldGT{
  rule main() = seq{
    println("2.1 Hello World transformation started");

    /* "choose" executes once or fails if it cannot, the "try" keyword will let
     the transformation continue even if the "choose" fails */
    try choose with apply createSimpleModelInstanctGT()
     do println("Creating Simple Model with ASM Rule");
    let Greeting = undef in seq{
      try choose with apply createExtendedModelInstanctGT(Greeting)
       do println("Creating Extended Model with ASM Rule");
      println("Executing model-to-text with ASM Rule");
      try choose with apply outputGreetingGT(Greeting) do skip;
    }
    println("2.1 Hello World transformation finished");
  }

  // finds (or creates) Greeting, GreetingMessage.Text and Person.Name
```

```
pattern TextAndNameForGreeting(Greeting,Text,Name) = {

  helloworldext.Greeting(Greeting) in nemf.resources;

  helloworldext.GreetingMessage(GreetingMessage) in Greeting;
  helloworldext.Greeting.greetingMessage(GreetingMessageRelation,
   Greeting,GreetingMessage);
  EString(Text) in GreetingMessage;
  helloworldext.GreetingMessage.text(TextRelation,GreetingMessage,Text);

  helloworldext.Person(Person) in Greeting;
  helloworldext.Greeting.person(PersonRelation,Greeting,Person);
  EString(Name) in Person;
  helloworldext.Person.name(NameRelation,Person,Name);
}

// GT Rule variant of simple Hello World model instance creation
gtrule createSimpleModelInstanctGT() = {
  // the "precondition" is true before the application of the GT Rule
  precondition pattern empty()= {
    // negative application condition (must not match)
    neg pattern existsGreeting(Greeting) = {
      helloworld.Greeting(Greeting);
    }
  }
  // the "postcondition" is true after the application of the GT Rule
  postcondition pattern createdGreeting(Text) = {
    helloworld.Greeting(Greeting) in nemf.resources;
    EString(Text) in Greeting;
    helloworld.Greeting.text(TextRelation,Greeting,Text);
  }
  action { // additional ASM based manipulations after GT Rule application
    setValue(Text,"Hello world");
  }
}

// GT Rule variant of extended Hello World model instance creation
gtrule createExtendedModelInstanctGT(out Greeting) = {
  precondition pattern empty()= {
    neg pattern existsGreeting(Greeting) = {
      helloworldext.Greeting(Greeting);
    }
  }
  postcondition find TextAndNameForGreeting(Greeting, Text, Name)

  action {
    setValue(Text,"Hello");
    setValue(Name,"TTC Participants");
  }
}

// GT Rule variant of model-to-text transformation
gtrule outputGreetingGT(in Greeting) = {
  precondition find TextAndNameForGreeting(Greeting, GreetingMessageText, PersonName)

  postcondition pattern outputString(Result) = {
    result.StringResult(Output) in nemf.resources;
    EString(Result) in Output;
    result.StringResult.result(ResR,Output,Result);
  }
  action{
    setValue(Result, value(GreetingMessageText) + " " + value(PersonName) + "!");
  }
}
```

```
}
```

Listing 2: Hello World transformation, GT variant

B.2 Common patterns for the Graph metamodels

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

machine graphPatterns
{
  // simple type wrapper for Graph
  pattern Graph(Graph) = {
    graph1.Graph(Graph);
  }

  // simple type wrapper for Node
  pattern SimpleNode(Node) = {
    graph1.Node(Node);
  }

  // finds the nodes and name relation of a node
  pattern NodesRelations(Graph,Node,NodesRelation,NameRelation) = {
    graph1.Node(Node);
    graph1.Graph(Graph);
    graph1.Graph.nodes(NodesRelation,Graph,Node);
    EString(Name);
    graph1.Node.name(NameRelation,Node,Name);
  }

  // finds name from the name relation of a node
  pattern nameOfNode(NameRelation,Name) = {
    graph1.Node(Node);
    EString(Name);
    graph1.Node.name(NameRelation,Node,Name);
  }

  // simple type wrapper for Edge
  pattern Edge(Edge) = {
    graph1.Edge(Edge);
  }

  // simple type wrapper for Edge in Graph
  pattern EdgeOfGraph(Graph,Edge) = {
    graph1.Edge(Edge);
    graph1.Graph(Graph);
    graph1.Graph.edges(EdgesRelation,Graph,Edge);
  }

  // finds the edges relation for a node
  pattern EdgesRelation(Graph,Edge,EdgesRelation) = {
    graph1.Edge(Edge);
    graph1.Graph(Graph);
    graph1.Graph.edges(EdgesRelation,Graph,Edge);
  }

  // finds src relation for Edge
  pattern srcAndRelForEdge(Edge,From,SourceRelation) = {
    graph1.Node(From);
    graph1.Edge(Edge);
    graph1.Edge.src(SourceRelation,Edge,From);
  }

  // finds trg relation for Edge
```

```
pattern trgAndRelForEdge(Edge,To,TargetRelation) = {
  graph1.Node(To);
  graph1.Edge(Edge);
  graph1.Edge.trg(TargetRelation,Edge,To);
}

// finds looping edges
pattern loopingEdge(Edge) = {
  find edgeFromToInternal(Edge,Node,Node);
}

// From is connected with an edge To
shareable pattern edgeFromToInternal(Edge,From,To) = {
  graph1.Node(From);
  graph1.Node(To);
  find srcAndRelForEdge(Edge,From,SourceRelation);
  find trgAndRelForEdge(Edge,To,TargetRelation);
}

// From is connected with an edge To
shareable pattern edgeFromTo(From,To) = {
  find edgeFromToInternal(Edge,From,To);
}

// From is connected with an edge To and both in Graph
pattern edgeFromToInGraph(From,To,Graph) = {
  find edgeFromToInternal(Edge,From,To);
  graph1.Graph(Graph);
  graph1.Graph.edges(EdgesRelation,Graph,Edge);
}


// finds isolated nodes
pattern isolatedNode(Node) = {
  graph1.Node(Node);
  neg find srcAndRelForEdge(Edge,Node,SourceRelation); // is not a source
  neg find trgAndRelForEdge(Edge,Node,TargetRelation); // is not a target
}

// three node in a circle
pattern circleOfThreeNode(Node,Inner1,Inner2) = {
  graph1.Node(Node);
  find edgeFromTo(Node,Inner1);
  find edgeFromTo(Inner1,Inner2);
  find edgeFromTo(Inner2,Node);
}

// edge with either source or target missing
pattern danglingEdge(Edge) = {// has source but no target
  find srcAndRelForEdge(Edge,From,SourceRelation);
  neg find trgAndRelForEdge(Edge,To,TargetRelation);
} or { // has target but no source
  find trgAndRelForEdge(Edge,To,TargetRelation);
  neg find srcAndRelForEdge(Edge,From,SourceRelation);
}

// finds the Source of Edge and the corresponding node in the evolved model
pattern OldAndNewSourceOfEdge(Edge,Source,Node2) = {
  find srcAndRelForEdge(Edge,Source,SourceRelation);
  graph2.Node(Node2);
  relation(Traceability,Source,Node2);
}

// finds the Target of Edge and the corresponding node in the evolved model
pattern OldAndNewTargetOfEdge(Edge,Target,Node2) = {
```

```
    find trgAndRelForEdge(Edge,Target,TargetRelation);
    graph2.Node(Node2);
    relation(Traceability,Target,Node2);
  }

  // finds traceability relations between nodes
  pattern TraceabilityRelation(Traceability) = {
    graph1.Node(Node);
    graph2.Node(Node2);
    relation(Traceability,Node,Node2);
  }

  // finds From and To of an edge and the corresponding new nodes
  shareable pattern oldAndNewEdgeFromTo(From,NewFrom,To,NewTo) = {
    find edgeFromTo(From,To);
    graph1.Node(From);
    graph3.Node(NewFrom);
    graph1.Node(To);
    graph3.Node(NewTo);
    relation(Tr1,From,NewFrom);
    relation(Tr2,To,NewTo);
  }

  // finds N1 node
  pattern N1Node(Node) = {
    graph1.Node(Node);
    EString(Name);
    graph1.Node.name(NameRel,Node,Name);
    check(value(Name) == "n1");
  }

  // Edge is connected to Node
  pattern connectedEdge(Node,Edge) = {
    find srcAndRelForEdge(Edge,Node,SourceRelation);
  } or {
    find trgAndRelForEdge(Edge,Node,TargetRelation);
  }

  // From and To (in Graph) are 2-hop transitively connected but not explicitly
  pattern transitiveEdgeMissing2hop(From,To,Graph) = {
    find edgeFromToInGraph(From,Inner,Graph);
    find edgeFromToInGraph(Inner,To,Graph);
    neg find edgeFromToInGraph(From,To,Graph);
  }

  // From and To (in Graph) are transitively connected but not explicitly
  @localsearch
  pattern transitiveEdgeMissing(From,To,Graph) = {
    find transitiveConnected(From,To,Graph);
    neg find edgeFromToInGraph(From,To,Graph);
  }

  // From and To (in Graph) are transitively connected
  @localsearch
  pattern transitiveConnected(From,To,Graph) = {
    find edgeFromToInGraph(From,Inner,Graph);
    find edgeFromToInGraph(Inner,To,Graph);
  } or {
    find edgeFromToInGraph(From,Inner,Graph);
    find transitiveConnected(Inner,To,Graph);
  }
}
```

Listing 3: Common graph patterns

B.3 Count Matches with certain Properties

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
machine countMatchesASM{

  rule main() = seq{
    println("2.2 Count matches transformation started");
    println("Counting number of nodes with ASM Rule");
    call countNodes();
    println("Counting looping edges with ASM Rule");
    call countLoopingEdges();
    println("Counting isolated nodes with ASM Rule");
    call countIsolatedNodes();
    println("Counting circles of three with ASM Rule");
    call countCirclesOfThree();
    println("Counting dangling edges with ASM Rule");
    call countDanglingEdges();
    println("2.2 Count matches transformation finished");

  }

  // ASM Rule variant of simple node counting
  rule countNodes() = let Count = 0 in seq {
    // "forall" is executed on each match of the patterns after "find"
    forall Node with find graphPatterns.SimpleNode(Node) do seq{
      update Count = Count+1; // update overwrites a variable
    }
    // creates EMF model for result
    call createResult(Count, "Number of nodes");
  }

  // ASM Rule variant of looping edge counting
  rule countLoopingEdges() = let Count = 0 in seq {

    forall Edge with find graphPatterns.loopingEdge(Edge) do seq{
      update Count = Count+1;
    }
    call createResult(Count, "Number of looping edges");
  }

  // ASM Rule variant of isolated node counting
  rule countIsolatedNodes() = let Count = 0 in seq {

    forall Node with find graphPatterns.isolatedNode(Node) do seq{
      update Count = Count+1;
    }
    call createResult(Count, "Number of isolated nodes");
  }

  // ASM Rule variant of circle of three counting
  rule countCirclesOfThree() = let Count = 0 in seq {

    forall Node,Inner1,Inner2 with
     find graphPatterns.circleOfThreeNode(Node,Inner1,Inner2) do seq{
      update Count = Count+1;
    }
    call createResult(Count, "Number of nodes in circles of three");
  }

  // ASM Rule variant of dangling edge counting
  rule countDanglingEdges() = let Count = 0 in seq {
```

```
    forall Edge with find graphPatterns.danglingEdge(Edge) do seq{
      update Count = Count+1;
    }
    call createResult(Count, "Number of dangling edges");
  }

  // ASM Rule for result storing
  rule createResult(in ResultValue, in Name) = let Result = undef,
   Value = undef, ResR = undef in seq{
    new(result.IntResult(Result) in nemf.resources);
    new(EInt(Value) in Result);
    new(result.IntResult.result(ResR,Result,Value));
    rename(Value,Name);
    setValue(Value, ResultValue);
  }
}
```

Listing 4: Count matches transformation, ASM variant

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
machine countMatchesMC{

  rule main() = seq{
    println("2.2 Count matches transformation started");
    println("Counting number of nodes with MCRule");
    call countNodesMC();
    println("Counting looping edges with MC Rule");
    call countLoopingEdgesMC();
    println("Counting isolated nodes with MC Rule");
    call countIsolatedNodesMC();
    println("Counting circles of three with MC Rule");
    call countCirclesOfThreeMC();
    println("Counting dangling edges with MC Rule");
    call countDanglingEdgesMC();
    println("2.2 Count matches transformation finished");

  }

  pattern countNodesPattern(N) = {
    find graphPatterns.SimpleNode(Node) # N; // counts the number of nodes
  }

  // MC Rule variant of simple node counting
  rule countNodesMC() = seq {
    try choose Count with find countNodesPattern(Count) do
     let Result = undef, ResR = undef, NodeCount = undef in seq{
      call createResult2(Count, "Number of nodes");
    }
  }

  pattern countLoopingEdgesPattern(N) = {
    find graphPatterns.loopingEdge(Edge) # N;
  }

  // MC Rule variant of looping edge counting
  rule countLoopingEdgesMC() = seq {

    try choose Count with find countLoopingEdgesPattern(Count) do
     let Result = undef, ResR = undef, LoopCount = undef in seq{
      call createResult2(Count, "Number of looping edges");
```

```
    }
  }

  pattern countIsolatedNodesPattern(N) = {
    find graphPatterns.isolatedNode(Node) # N;
  }

  // MC Rule variant of isolated node counting
  rule countIsolatedNodesMC() = seq {

    try choose Count with find countIsolatedNodesPattern(Count) do
     let Result = undef, ResR = undef, IsolatedCount = undef in seq{
      call createResult2(Count, "Number of isolated nodes");
    }
  }

  pattern countCirclesOfThreePattern(N) = {
    find graphPatterns.circleOfThreeNode(Node,Inner1,Inner2) # N;
  }

  // MC Rule variant of circles of three counting
  rule countCirclesOfThreeMC() = seq {

    try choose Count with find countCirclesOfThreePattern(Count) do
     let Result = undef, ResR = undef, CircleCount = undef in seq{
      call createResult2(Count, "Number of nodes in circles of three");
    }
  }

  pattern countDanglingEdgesPattern(N) = {
    find graphPatterns.danglingEdge(Node) # N;
  }

  // MC Rule variant of dangling edge counting
  rule countDanglingEdgesMC() = seq {

    try choose Count with find countDanglingEdgesPattern(Count) do
     let Result = undef, ResR = undef, DanglingCount = undef in seq{
      call createResult2(Count, "Number of dangling edges");
    }
  }

  // ASM Rule for result storing
  rule createResult2(in ResultValue, in Name) = let Result = undef,
   Value = undef, ResR = undef in seq{
    new(result.IntResult(Result) in nemf.resources);
    new(EInt(Value) in Result);
    new(result.IntResult.result(ResR,Result,Value));
    rename(Value,Name);
    setValue(Value, ResultValue);
  }
}
```

Listing 5: Count matches transformation, match count variant

B.4 Reverse Edges

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
machine reverseEdgesASM{

  rule main() = seq{
```

```
    println("2.3 Reverse edges transformation started");
    call reverseEdges();
    println("2.3 Reverse edges transformation finished");
  }

  // ASM Rule variant for reverse edges
  rule reverseEdges() = seq{

    forall Edge with find graphPatterns.Edge(Edge) do let SR = undef, TR = undef in seq{
      // finds src relation if exists
      println(" > Reversing " + name(Edge) + " edge.");
      try choose Source, SourceRelation with
       find graphPatterns.srcAndRelForEdge(Edge,Source,SourceRelation) do seq{
         update SR = SourceRelation;
      }
      // finds trg relation if exists
      try choose Target, TargetRelation with
       find graphPatterns.trgAndRelForEdge(Edge,Target,TargetRelation) do seq{
        update TR = TargetRelation;
      }
      if(SR != undef) seq{
        // replace instanceOf relation
        // instanceOf is a relation type, which can be dynamically deleted
        delete(instanceOf(SR,nemf.packages.graph1.Edge.src));
        new(instanceOf(SR,nemf.packages.graph1.Edge.trg)); // and created
      }
      if(TR != undef) seq{
        // replace instanceOf relation
        delete(instanceOf(TR,nemf.packages.graph1.Edge.trg));
        new(instanceOf(TR,nemf.packages.graph1.Edge.src));
      }
    }
  }
}
```

Listing 6: Reverse edges transformation, ASM variant

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
machine reverseEdgesGT{

  rule main() = seq{
    println("2.3 Reverse edges transformation started");
    forall Edge with apply reverseEdgesGT(Edge) do
      println(" > Reversing " + name(Edge) + " edge.");
    println("2.3 Reverse edges transformation finished");
  }

  // GT Rule variant for reverse edges
  // note: add "shareable" keyword before "pattern" to
  // actually reverse looping edges as well
  gtrule reverseEdgesGT(out Edge) = {
    precondition pattern edgeWithRelations(Edge,From,To,SourceRel,TargetRel) = {
      find graphPatterns.srcAndRelForEdge(Edge,From,SourceRel);
      find graphPatterns.trgAndRelForEdge(Edge,To,TargetRel);
    }
    postcondition pattern reversedEdges(Edge,From,To,SourceRel,TargetRel) = {
      find graphPatterns.srcAndRelForEdge(Edge,To,SourceRel);
      find graphPatterns.trgAndRelForEdge(Edge,From,TargetRel);
    }
  }
```

```
}
```

Listing 7: Reverse edges transformation, GT variant

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
machine reverseEdgesRel{

  rule main() = seq{
    println("2.3 Reverse edges transformation started");
    call reverseEdges();
    println("2.3 Reverse edges transformation finished");
  }

  // ASM Rule variant for reverse edges
  rule reverseEdges() = seq{

    forall Edge with find graphPatterns.Edge(Edge) do
     let S = undef, SR = undef, T = undef, TR = undef in seq{
      // finds src relation if exists
      println(" > Reversing " + name(Edge) + " edge.");
      try choose Source,SourceRelation with
       find graphPatterns.srcAndRelForEdge(Edge,Source,SourceRelation) do seq{
         update S = Source;
         update SR = SourceRelation;
      }
      // finds trg relation if exists
      try choose Target,TargetRelation with
       find graphPatterns.trgAndRelForEdge(Edge,Target,TargetRelation) do seq{
        update T = Target;
        update TR = TargetRelation;
      }
      if(T != undef) seq{
        println("   > Reversing target to source: " + name(T));
        if(SR != undef) setTo(SR,T); // change target of relation
        else seq{
          delete(TR);
          new(graph1.Edge.src(SR,Edge,T));
        }
      }
      if(S != undef) seq{
        println("   > Reversing source to target: " + name(S));
        if(TR != undef) setTo(TR,S);
        else seq{
          delete(SR);
          new(graph1.Edge.trg(TR,Edge,S));
        }
      }
    }
  }
}
```

Listing 8: Reverse edges transformation, relation manipulation variant

B.5 Simple Migration

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
```

```
machine simpleMigration{

  rule main() = seq{

    println("2.4 Simple Migration (with copy) transformation started");
    call migrateGraph();
    println("2.4 Simple Migration (with copy) transformation finished");
  }

  // ASM Rule variant of simple migration transformation
  rule migrateGraph() = seq{

    forall Graph with find graphPatterns.Graph(Graph) do
     let Graph2 = undef, GCSRel = undef in seq{
      new(graph2.Graph(Graph2) in nemf.resources); // create graph

      forall Node,NodesRelation, NameRelation with
       // for each node, create a new
       find graphPatterns.NodesRelations(Graph,Node,NodesRelation, NameRelation) do
       let Node2 = undef, Text = undef, TextRel = undef,Traceability = undef in seq{
        new(graph2.Node(Node2) in Graph2);
        new(graph2.Graph.gcs(GCSRel,Graph2,Node2));
        new(EString(Text) in Node2);
        try choose Name with find graphPatterns.nameOfNode(NameRelation,Name) do seq{
          setValue(Text,value(Name));
        }
        new(graph2.GraphComponent.text(TextRel,Node2,Text));
        // store the traceability between old and new node
        new(relation(Traceability,Node,Node2));
      }

      // for each edge, create a new
      forall Edge,EdgesRelation with
       find graphPatterns.EdgesRelation(Graph,Edge,EdgesRelation) do
       let Edge2 = undef,Text = undef,TextRel = undef, EvolvedRel = undef in seq{
        new(graph2.Edge(Edge2) in Graph2);
        new(graph2.Graph.gcs(GCSRel,Graph2,Edge2));
        new(EString(Text) in Edge2);
        new(graph2.GraphComponent.text(TextRel,Edge,Text));
        // each source relation is created to the corresponding node
        forall Source,Node2 with
         find graphPatterns.OldAndNewSourceOfEdge(Edge,Source,Node2) do seq{
          new(graph2.Edge.src(EvolvedRel,Edge2,Node2));
        }
        forall Target,Node2 with
         find graphPatterns.OldAndNewTargetOfEdge(Edge,Target,Node2) do seq{
          // each tagret relation is created to the corresponding node
          new(graph2.Edge.trg(EvolvedRel,Edge2,Node2));
        }
      }

      // delete traceability models
      forall Traceability with
       find graphPatterns.TraceabilityRelation(Traceability) do seq{
        delete(Traceability);
      }
    }
  }
}
```

Listing 9: Simple Migration transformation, copy variant

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;
```

```
@incremental
machine simpleMigrationInplace{

  rule main() = seq{

    println("2.4 Simple Migration (in-place) transformation started");
    call migrateGraphInplace();
    println("2.4 Simple Migration (in-place) transformation finished");
  }

  // ASM Rule variant of simple migration in-place transformation
  rule migrateGraphInplace() = seq{
    // at this point, each Graph is transformed
    forall Graph with find graphPatterns.Graph(Graph) do seq{

      // each node is transformed using instanceOf changing
      forall Node,NodesRelation, NameRelation with
       find graphPatterns.NodesRelations(Graph,Node,NodesRelation, NameRelation) do seq{
        delete(instanceOf(Node,nemf.packages.graph1.Node));
        new(instanceOf(Node,nemf.packages.graph2.Node));
        delete(instanceOf(NodesRelation,nemf.packages.graph1.Graph.nodes));
        new(instanceOf(NodesRelation,nemf.packages.graph2.Graph.gcs));
        delete(instanceOf(NameRelation,nemf.packages.graph1.Node.name));
        new(instanceOf(NameRelation,nemf.packages.graph2.GraphComponent.text));
      }

      // each edge is transformed using instanceOf changing
      forall Edge,EdgesRelation with
       find graphPatterns.EdgesRelation(Graph,Edge,EdgesRelation) do
       let Text = undef,TextRel = undef in seq{
        delete(instanceOf(Edge,nemf.packages.graph1.Edge));
        new(instanceOf(Edge,nemf.packages.graph2.Edge));
        delete(instanceOf(EdgesRelation,nemf.packages.graph1.Graph.edges));
        new(instanceOf(EdgesRelation,nemf.packages.graph2.Graph.gcs));
        new(EString(Text) in Edge);
        new(graph2.GraphComponent.text(TextRel,Edge,Text));
      }
      // the graph is transformed
      delete(instanceOf(Graph,nemf.packages.graph1.Graph));
      new(instanceOf(Graph,nemf.packages.graph2.Graph));
    }
  }
}
```

Listing 10: Simple Migration transformation, in-place variant

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
machine simpleMigrationTopology{

  rule main() = seq{

    println("2.4 Simple Migration (topoogy change with copy) transformation started");
    call topologyChange();
    println("2.4 Simple Migration (topoogy change with copy) transformation finished");
  }

  // ASM Rule variant of topology-changing migration
  rule topologyChange() = seq{
    forall Graph with find graphPatterns.Graph(Graph) do
     let NewGraph = undef, Rel = undef in seq{
```

```
      new(graph3.Graph(NewGraph) in nemf.resources);

      forall Node,NodesRelation, NameRelation with
       find graphPatterns.NodesRelations(Graph,Node,NodesRelation, NameRelation) do
       let NewNode = undef, Text = undef, Traceability = undef in seq{
        new(graph3.Node(NewNode) in NewGraph);
        new(graph3.Graph.nodes(Rel,NewGraph,NewNode));
        new(EString(Text) in NewNode);
        try choose Name with find graphPatterns.nameOfNode(NameRelation,Name) do seq{
          setValue(Text,value(Name));
        }
        new(graph3.Node.text(Rel,NewNode,Text));
        new(relation(Traceability,Node,NewNode));
      }

      forall From,To,NewFrom,NewTo with
       find graphPatterns.oldAndNewEdgeFromTo(From,NewFrom,To,NewTo) do
       let LinksToRel = undef in seq{
        new(graph3.Node.linksTo(LinksToRel,NewFrom,NewTo));
      }
      forall Traceability with
       find graphPatterns.TraceabilityRelation(Traceability) do seq{
        delete(Traceability);
      }
    }
  }
}
```

Listing 11: Simple Migration Topology changing transformation, copy variant

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
machine simpleMigrationTopologyInplace{

  rule main() = seq{

    println("2.4 Simple Migration (topology change in-place) transformation started");
    call topologyChangeInplace();
    println("2.4 Simple Migration (topology change in-place) transformation finished");
  }

  // ASM Rule variant of topology-changing in-place migration
  rule topologyChangeInplace() = seq{
    forall Graph with find graphPatterns.Graph(Graph) do seq{

      forall Node,NodesRelation, NameRelation with
       find graphPatterns.NodesRelations(Graph,Node,NodesRelation,NameRelation) do
       seq{ //
        delete(instanceOf(Node,nemf.packages.graph1.Node));
        new(instanceOf(Node,nemf.packages.graph3.Node));
        delete(instanceOf(NodesRelation,nemf.packages.graph1.Graph.nodes));
        new(instanceOf(NodesRelation,nemf.packages.graph3.Graph.nodes));
        delete(instanceOf(NameRelation,nemf.packages.graph1.Node.name));
        new(instanceOf(NameRelation, nemf.packages.graph3.Node.text));

        forall To with find graphPatterns.edgeFromTo(Node,To) do
         let LinksToRel = undef in seq{
          new(graph3.Node.linksTo(LinksToRel,Node,To));
        }
      }

      forall Edge,EdgesRelation with
```

```
      find graphPatterns.EdgesRelation(Graph,Edge,EdgesRelation) do seq{
        delete(Edge);
      }

      delete(instanceOf(Graph,nemf.packages.graph1.Graph));
      new(instanceOf(Graph, nemf.packages.graph3.Graph));
    }
  }

}
```

Listing 12: Simple Migration Topology changing transformation, in-place variant

B.6 Delete Node with Specific Name and its Incident Edges

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
machine deleteNodeASM{

  rule main() = seq{
    println("2.5 Delete nodes transformation started");
    call deleteNode();
    println("2.3 Delete nodes transformation finished");
  }

  rule deleteNode() = seq{
    try choose N1 with find graphPatterns.N1Node(N1) do seq{
      delete(N1);
    }
  }
}
```

Listing 13: Delete node transformation, ASM variant

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
machine deleteNodeGT{

  rule main() = seq{
    println("2.5 Delete n1 node transformation (GT) started");
    try choose with apply deleteNodeGT() do skip;
    println("2.3 Delete n1 node transformation finished");
  }

  gtrule deleteNodeGT() = {
    precondition find graphPatterns.N1Node(N1)

    postcondition pattern noN1(N1) = {
      neg find graphPatterns.N1Node(N1);
    }
  }

}
```

Listing 14: Delete node transformation, GT variant

```
import datatypes;
import nemf.packages;
```

```
import nemf.ecore.datatypes;

@incremental
machine deleteNodeIncidentASM{

  rule main() = seq{
    println("2.5 Delete nodes transformation started");
    println("Deleting incident edges as well");
    call deleteNodeAndIncidentEdges();
    println("2.3 Delete nodes transformation finished");
  }

  rule deleteNodeAndIncidentEdges() = seq{
    try choose N1 with find graphPatterns.N1Node(N1) do seq{
      forall Edge with find graphPatterns.connectedEdge(N1,Edge) do seq{
        delete(Edge);
      }
      delete(N1);
    }
  }
}
```

Listing 15: Delete node and incident edges transformation, ASM variant

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
machine deleteNodeIncidentGT{

  rule main() = seq{
    println("2.5 Delete n1 node transformation (GT) started");
    println("Deleting incident edges as well");
    try choose with apply deleteNodeAndIncidentEdgesGT() do skip;
    println("2.3 Delete n1 node transformation finished");
  }

  gtrule deleteNodeGT(in N1) = {
    precondition find graphPatterns.N1Node(N1)

    postcondition pattern noN1(N1) = {
      neg find graphPatterns.N1Node(N1);
    }
  }

  gtrule deleteNodeAndIncidentEdgesGT() = {
    precondition find graphPatterns.N1Node(N1)

    action{
      forall Edge with apply deleteIncidentEdgesOfNode(N1,Edge) do skip;
      try choose with apply deleteNodeGT(N1) do skip;
    }
  }

  gtrule deleteIncidentEdgesOfNode(in Node, out Edge) = {
    precondition find graphPatterns.connectedEdge(Node,Edge)

    postcondition pattern noConnectingEdge(Node,Edge) = {
      graph1.Node(Node);
      neg find graphPatterns.connectedEdge(Node,Edge);
    }
  }
```

```
}
```

Listing 16: Delete node and incident edges transformation, GT variant

B.7 Insert Transitive Edges

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
machine transitiveEdgesASM{

  rule main() = seq{
    println("2.6 Transitive edges (R u R^2) transformation (ASM) started");
    call insertTransitiveEdgesOnce();
    println("2.6 Transitive edges transformation finished");
  }

  // ASM Rule variant for inserting edges
  // between each pair of transitively connected nodes
  rule insertTransitiveEdgesOnce() = seq{
    forall From, To, Graph with
     find graphPatterns.transitiveEdgeMissing2hop(From,To,Graph) do
     let TransitiveEdge = undef, Rel = undef in seq{
      new(graph1.Edge(TransitiveEdge) in Graph);
      new(graph1.Graph.edges(Rel,Graph,TransitiveEdge));
      new(graph1.Edge.src(Rel,TransitiveEdge,From));
      new(graph1.Edge.trg(Rel,TransitiveEdge,To));
    }
  }

}
```

Listing 17: Insert transitive edges transformation, ASM variant

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
machine transitiveEdgesGT{

  rule main() = seq{
    println("2.6 Transitive edges (R u R^2) transformation (GT) started");
    forall From, To with apply insertTransitiveEdgesOnceGT(From, To) do skip;
    println("2.6 Transitive edges transformation finished");
  }

  // GT Rule for inserting transitive edges between From and To
  gtrule insertTransitiveEdgesOnceGT(out From, out To) = {
    precondition find graphPatterns.transitiveEdgeMissing2hop(From,To,Graph)

    postcondition find graphPatterns.edgeFromToInGraph(From,To,Graph)
  }
}
```

Listing 18: Insert transitive edges transformation, GT variant

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
```

```
machine transitiveEdgesIterativeASM{

  rule main() = seq{
    println("2.6 Transitive edges (R u R^2) transformation (ASM) started");
    println("Insert edges iteratively");
    call insertTransitiveEdgesIterative();
    println("2.6 Transitive edges transformation finished");
  }

  // ASM Rule variant for inserting edges between each transitively connected nodes
  rule insertTransitiveEdgesIterative() = seq{
    iterate choose From, To, Graph with
     find graphPatterns.transitiveEdgeMissing2hop(From,To,Graph) do
     let TransitiveEdge = undef, Rel = undef in seq{
      new(graph1.Edge(TransitiveEdge) in Graph);
      new(graph1.Graph.edges(Rel,Graph,TransitiveEdge));
      new(graph1.Edge.src(Rel,TransitiveEdge,From));
      new(graph1.Edge.trg(Rel,TransitiveEdge,To));
    }
  }

}
```

Listing 19: Insert all transitive edges iteratively transformation, ASM variant

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

@incremental
machine transitiveEdgesIterativeGT{

  rule main() = seq{
    println("2.6 Transitive edges (R u R^2) transformation (GT) started");
    println("Insert edges iteratively");
    iterate choose From, To with apply insertTransitiveEdgesOnceGT(From, To) do skip;
    println("2.6 Transitive edges transformation finished");
  }

  // GT Rule for inserting transitive edges between From and To
  gtrule insertTransitiveEdgesOnceGT(out From, out To) = {
    precondition find graphPatterns.transitiveEdgeMissing2hop(From,To,Graph)

    postcondition find graphPatterns.edgeFromToInGraph(From,To,Graph)
  }
}
```

Listing 20: Insert all transitive edges iteratively transformation, GT variant

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

machine transitiveEdgesAllASM{

  rule main() = seq{
    println("2.6 Transitive edges (R u R^2 ... u R^n) transformation (ASM) started");
    call insertTransitiveEdgesAll();
    println("2.6 Transitive edges transformation finished");
  }

  // ASM Rule variant for inserting edges
  // between each pair of transitively connected nodes
  rule insertTransitiveEdgesAll() = seq{
    forall From, To, Graph with
```

```
    find graphPatterns.transitiveEdgeMissing(From,To,Graph) do
    let TransitiveEdge = undef, Rel = undef in seq{
     new(graph1.Edge(TransitiveEdge) in Graph);
     new(graph1.Graph.edges(Rel,Graph,TransitiveEdge));
     new(graph1.Edge.src(Rel,TransitiveEdge,From));
     new(graph1.Edge.trg(Rel,TransitiveEdge,To));
   }
  }
}
```

Listing 21: Insert all transitive edges transformation, ASM variant

```
import datatypes;
import nemf.packages;
import nemf.ecore.datatypes;

machine transitiveEdgesAllGT{

  rule main() = seq{
    println("2.6 Transitive edges (R u R^2 ... u R^n) transformation (GT) started");
    forall From, To with apply insertTransitiveEdgesAllGT(From, To) do skip;
    println("2.6 Transitive edges transformation finished");
  }

  // GT Rule for inserting transitive edges between From and To
  gtrule insertTransitiveEdgesAllGT(out From, out To) = {
    precondition find graphPatterns.transitiveEdgeMissing(From,To,Graph)

    postcondition find graphPatterns.edgeFromToInGraph(From,To,Graph)
  }
}
```

Listing 22: Insert all transitive edges transformation, GT variant