Solving the TTC 2011 Reengineering Case with VIATRA2*

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 current paper presents a solution of the *Program Understanding: A Reengineering 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 to 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 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 graph pattern with an image of the RHS 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.

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)

*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. 136–148, doi:10.4204/EPTCS.74.13

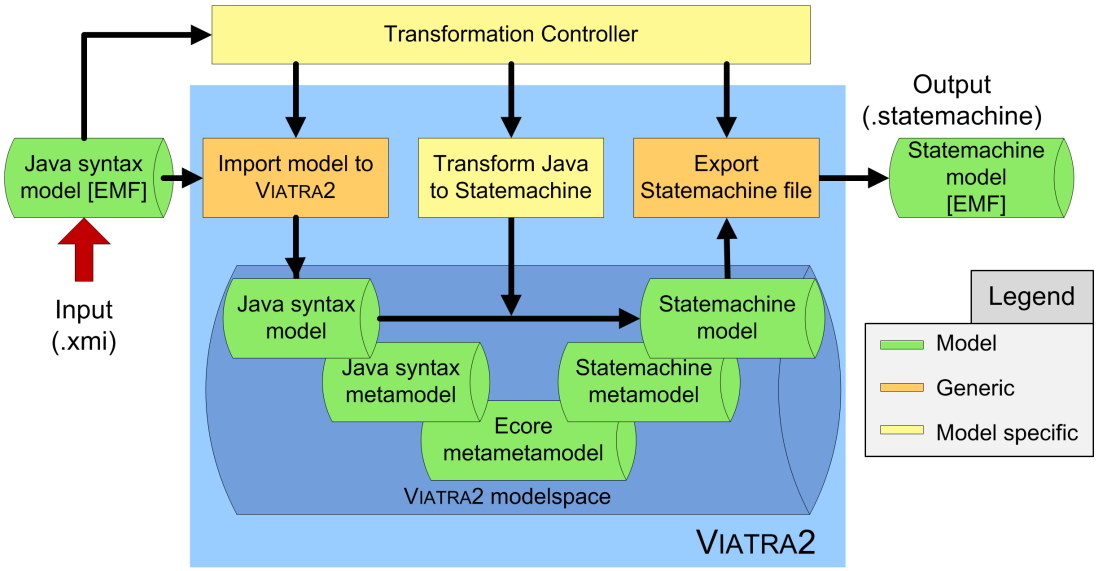

Figure 1: Solution Architecture

incremental pattern matching (INC) are available. This feature provides the transformation designer additional opportunities to fine tune the transformation either for faster execution (INC) or lower memory consumption (LS) [6].

The rest of the paper is structured as follows. Sec. 2 gives an architectural overview of the transformation, while Sec. 3 highlights the interesting parts of our implementation and finally Sec. 4 concludes the paper.

2 Solution Architecture

We implemented our solution for the Program Understanding case study [5] using the VIATRA2 model transformation framework. Fig. 1 shows the complete architecture with both preexisting (depicted with darker rectangles) and newly created components (lighter rectangles). The optional *Transformation Controller* is an extension to the Eclipse framework that provides an easy-to-use graphical interface for executing the underlying transformation (i.e. it appears as a command in the pop-up menu of XMI files); it is, however, possible to execute the same steps manually on the user interface of VIATRA2. From the user perspective, the controller is invoked on an *input XMI* file and the result is an *output Statemachine* file.

Note that the transformation is performed on models *inside the VPM modelspace* of VIATRA2 rather than on in-memory EMF models. Although VIATRA2 does not manipulate EMF models directly, it includes a generic support for handling EMF metamodels and instance models.

In order to understand the transformation we briefly outline the metamodeling approach of our solution. The *Ecore metametamodel* is the base of this support, which was defined in accordance with the actual EMF metamodel of Ecore.

Both the *Java syntax graph and Statemachine metamodels* are defined as instances of this metametamodel, and are imported into VIATRA2 with the generic Ecore metamodel importer. Then the input file is used to *import the Java syntax graph into* VIATRA2 and create the *Java syntax model* which is the

instance of the Java syntax metamodel.

By executing our implemented transformation, we can *transform the Java syntax model to a Statemachine model* which is an instance of the Statemachine metamodel. This *Statemachine model* is then *exported* to create the output Statemachine file.

3 Transforming Java syntax to statemachines (J2SM)

The *J2SM* transformation generates the Statemachine model from the Java syntax graph in the VIATRA2 framework and is implemented in the VIATRA2 Textual Command Language (VTCL) [1]. J2SM can be separated into four parts, (1) the construction of the Statemachine states and their outgoing transitions, (2) the processing of triggers and (3) actions for outgoing transitions, and finally (4) connecting the transitions to the target states.

The complete transformation is around 450 lines of VTCL code including whitespaces and comments (see Appendix B). It includes 21 complex patterns, e.g. the Java class called through an *Instance.activate()* method call can be looked up with the pattern in line 172. Finally, the actual manipulation is executed by 5 declarative rules (e.g. create trigger for a given transition, see line 227). There are 2 additional rules for starting and stopping time measurement for different parts of the transformation (see lines 440 and 448).

The transformation starts with a short initialization phase, where the output buffer for the transformation log is cleared, the time measurement starts and a new statemachine model is created.

Construction of states and transitions. The elements representing the states and transitions of the statemachine are created in the following way:

1. First, states are created for each Java class that is not an abstract subclass of the *State* class (see top-level pattern at line 97, called in line 45 from a forall construct). A recursive pattern finds these classes by traversing supertype edges.
2. Once the state is created, we store the correspondence between the class and the state in an ASM function (essentially a hashmap), the transition handling rule is called (line 63).
3. Since at this point the target states of a transition is probably not available, we only create the *src* and *out* relations.
4. The transitions in a class are identified by another complex pattern that matches the *Class.Instance.activate()* method calls and finds the called class (see line 172). The `below` keyword is used in a subpattern to express transitive containment of the target class reference within the definition of the source class.
5. Once the transition is created, we also store the called class for the transition in the same ASM function to be able to create the *dst* and *in* relations later.

Processing triggers. Next, the rule handling triggers (see line 227) is called from line 163. The triggers are created based on the class method, where the *activate()* call is found (see pattern in line 293), the switch case constant (line 300) or the catch block exception (line 332) that is the closest in the statement hierarchy to the method call. Note that when a catch block is inside another catch block (and similarly for switch cases), the reference solution may choose the outer one for the trigger, while our solution chooses the correct one.

Processing actions. In the following phase, the action part of the transition is created (line 368). The action is created based on the existence of a *send()* method call in the same statement container (found using the pattern in line 403) as the *activate()* call. The name of the action is the same as the enumeration value from the *send()* method call parameter (line 416).

Connecting transitions to targets. Finally, the target of all transitions are handled in the same step using a forall construct (see line 206). The interesting part of this rule is the usage of ASM functions to retrieve the correct target state (line 215). Remember, that the called class is stored for transitions and states are stored for created classes. Therefore, since we iterate through all transitions, the target state can be selected by retrieving the called class for the current transition and the state for that class.

Performance. We used the provided models to test the performance of our implementation. We observed that our framework was unable to handle the biggest model, if we tried to import the complete model, due to VIATRA2's VPM representation consuming more memory than EMF. For the other input models, the total runtime of the plug-in loading, import, transformation and export is around 10 seconds, while the transformation itself is around 2 seconds.

However, if we allow a preprocessing phase, which removes unnecessary parts of the model (with the help of EMF IncQuery[1]), the big model could be transformed. However, this reduced model is almost equal to the medium model, thus it does not demonstrate the scalability of the approach.

Evaluation. The transformation handles the core task and both extensions, therefore it is *complete*. The generated state machines are equal to the provided reference solutions, the source and target of transitions are set, while triggers and actions are also created, which means the transformation is *correct*. The transformation code itself is well-structured and is annotated with comments to increase *understandability*. However, it may be challenging for those unfamiliar with the language. Since the language and the framework are not tailored to EMF, the *conciseness* of the transformation is lower and the *performance* of the framework is limited (as discussed above). As a main development direction, we are working on new tools for more powerful EMF support.

4 Conclusion

In the current paper we have presented our VIATRA2 based implementation for the Program Understanding case study [5].

The high points of our transformation are (i) the reusable patterns, (ii) the easily readable transformation language, (iii) the use of ASM functions for easily retrieving corresponding elements, and (iv) that triggers are created for the correct switch case and catch block (as opposed to reference solution).

On the other hand, import-export of models is required and we cannot handle the largest sample input model due to memory constraints.

[1] `http://viatra.inf.mit.bme.hu/incquery/`

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 Solving the TTC 2011 Program Understanding Case with* VIATRA2. Available at `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu-11_TTC11_VIATRA.vdi`.

[5] Tassilo Horn (2011): *Program Understanding: A Reengineering 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.

[6] 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.

[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

Our implementation for the case study together with the current version of VIATRA2 can be installed from the following Eclipse update site: `http://mit.bme.hu/~ujhelyiz/viatra/ttc11/`. Additionally, the solution is also available an archive file: `http://mit.bme.hu/~ujhelyiz/viatra/ttc11.zip`. Similarly, our solution for the Hello World! case is downloadable from `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 - Program Understanding transformation

```
// metamodel imports
import nemf.packages.classifiers;
import nemf.packages.commons;
import nemf.packages.types;
import nemf.packages.modifiers;
import nemf.packages.references;
import nemf.packages.members;
import nemf.packages.statements;
import nemf.packages.parameters;
import nemf.packages.expressions;
import nemf.packages.statemachine;
import nemf.ecore;
import nemf.ecore.datatypes;

@incremental
machine reengineeringJava{

  asmfunction buf/0;     // output buffer
  asmfunction time/1;    // runtime measurement data
  asmfunction models/1; // storing models
  asmfunction sm/1;      // store for statemachine related elements

  // entry point of transformation
  rule main() = seq{

    // initialize output buffer
    let Buf = clearBuffer("core://reEngineer") in seq{
      update buf() = getBuffer("core://reEngineer");
    }

    call startTimer("main");
    println(buf(), "ReEngineering Transformation started.");

    // create new statemachine
    let StateMachine = undef in seq{
      new(StateMachine(StateMachine) in nemf.resources);
      rename(StateMachine,"A_StateMachine");
      update models("sm") = StateMachine;
    }

    // finds all State subtypes
    /* 1. A State is a non-abstract Java class (classifiers.Class) that
     extends  the abstract class named ``State'' directly or indirectly.
     All concrete state classes are implemented as singletons [GHJV95]. */
    forall StateClass with find NotAbstractStateClass(StateClass) do
     let State = undef, StatesRel = undef, NameRel = undef in seq{
```

```
      println(buf(), " --> Found State class " + name(StateClass));
      // create states in StateMachine
      new(State(State) in models("sm"));
      new(StateMachine.states(StatesRel,models("sm"),State));
      // store Class -> State correspondence
      update sm(StateClass) = State;
      // add name to State
      try choose Name with find NameOfElement(Name,StateClass) do
       let StateName = undef in seq{
        new(EString(StateName) in State);
        setValue(StateName,value(Name));
        rename(State,value(Name));
        new(State.name(NameRel,State,StateName));
      }
      // create transitions from state
      call createTransitions(StateClass);
    }

    // for each Transition, finds target (use sm map)
    call createTransitionTargets();

    call endTimer("main");
    println(buf(), "ReEngineering Transformation ended " + time("main"));
    println(buf(), " RULE: createTransitions ran (in total) for "
     + time("createTransitions"));
    println(buf(), " RULE: createTransitionTargets ran (in total) for "
     + time("createTransitionTargets"));
    println(buf(), " RULE: addTrigger ran (in total) for "
     + time("addTrigger"));
    println(buf(), " RULE: addAction ran (in total) for "
     + time("addAction"));
  }

  // finds classes which are subtypes of State
  pattern ClassSubTypeOfState(Class) = {
    Class(Class);
    find SuperTypeOfClass(SuperType,Class);

    find NameOfElement(Name,SuperType);
    check(value(Name) == "State");

  } or { // transitive matching
    Class(Class);
    find SuperTypeOfClass(SuperType,Class);

    find ClassSubTypeOfState(SuperType);
  }

  // restrict subtypes of State to non-abstract ones
  pattern NotAbstractStateClass(Class) = {
    find ClassSubTypeOfState(Class);
    neg find AbstractClass(Class);
  }

  // finds name attribute for element
  pattern NameOfElement(Name,Element) = {
    NamedElement(Element);
    NamedElement.name(NameRel,Element,Name);
    EString(Name);
  }

  // finds supertype of class
  pattern SuperTypeOfClass(SuperType,Class) = {
    Class(Class);
    Class.extends(Extends,Class,NSClassRef);
```

```
  find TargetOfNamespaceClassifierReference(NSClassRef, SuperType);
  Class(SuperType);
}

// navigate on the classifierReference and target relations to Target
pattern TargetOfNamespaceClassifierReference(NSClassRef, Target) = {
  NamespaceClassifierReference(NSClassRef);
  NamespaceClassifierReference.classifierReferences(ClassRefRel,
    NSClassRef,ClassRef);
  ClassifierReference(ClassRef);
  ClassifierReference.target(TargetRel,ClassRef,Target);
}

// matches abstract classes
pattern AbstractClass(Class) = {
  Class(Class);
  AnnotableAndModifiable.annotationsAndModifiers(ModifierRel,
    Class,Abstract);
  Abstract(Abstract);
}

// create transitions leading out from StateClass
rule createTransitions(in StateClass) = seq{
  call startTimer("createTransitions");
  // finds all transition in class
  /* 2. A Transition is encoded by a methodcall (references.MethodCall),
   which invokes the next state's Instance () method (members.Method)
   returning the singleton instance of that state on which the activate ()
   method is called in turn. This activation may be contained in any of the
   classes' methods with an arbitrary deep nesting. */
  forall ActivateCallClass,ActivateClassRef with
   find ClassCalledWithActivate(ActivateCallClass,
    ActivateClassRef,StateClass) do let Transition = undef,
     TransRel = undef, SrcRel = undef, OutRel = undef in seq{

    println(buf(), "   --> Found activate() methodcall to "
     + name(ActivateCallClass));
    // create Transitions
    new(Transition(Transition) in models("sm"));
    new(StateMachine.transitions(TransRel,models("sm"),Transition));

    rename(Transition, name(StateClass) + "-"
     + name(ActivateCallClass));
    // add source, use correspondence for finding state
    new(Transition.src(SrcRel,Transition,sm(StateClass)));
    new(State.out(OutRel,sm(StateClass),Transition));

    // store reference to the class on the other end of transition
    update sm(Transition) = ActivateCallClass;
    // add trigger
    call addTrigger(ActivateClassRef, Transition);
    // add action
    call addAction(ActivateClassRef, Transition);
  }
  call endTimer("createTransitions");
}

// finds the class which is called using an activate() method
pattern ClassCalledWithActivate(ActivateCallClass,
 ActivatedClassRef,StateClass) = {
  find ClassSubTypeOfState(StateClass); // check that the class is a state

  // reference to Class
  find ReferenceTarget(ActivatedClassRef,
   StateClass,ActivateCallClass);
```

```
  Reference.next(ACRNextRef,ActivatedClassRef,InstanceCall);
  // reference to Instance method
  find MethodCall(InstanceCall,ActivateCallClassInstance);
  Reference.next(ERNextRef,InstanceCall,ActivateCall);
  find NameOfElement(Name,ActivateCallClassInstance); // name of Instance
  check(value(Name) == "Instance");
  // reference to activate() method
  find MethodCall(ActivateCall,ActivateMethod);
  find NameOfElement(ActName,ActivateMethod);
  check(value(ActName) == "activate");
}

// finds reference to target
pattern ReferenceTarget(TargetRef,SourceElement,ReferencedTarget) = {
  Commentable(SourceElement);
  ReferenceableElement(ReferencedTarget);
  IdentifierReference(TargetRef) below SourceElement;
  ElementReference.target(TargetRefRel,TargetRef,ReferencedTarget);
}

// finds method called by Caller
pattern MethodCall(Caller,CalledMethod) = {
  MethodCall(Caller);
  ElementReference.target(TargetRef,Caller,CalledMethod);
  ClassMethod(CalledMethod);
}

// create references between transitions and target states
rule createTransitionTargets() = seq{
  call startTimer("createTransitionTargets");
  println(buf(), " RULE: Creating transition targets");

  forall Transition with find Transition(Transition) do
   let DstRel = undef, InRel = undef in seq{
     println(buf(), " --> Creating target for " + name(Transition));
    // sm(Transition) returns the target class TargetClass
    // sm(TargetClass) returns the corresponding state
    new(Transition.dst(DstRel,Transition,sm(sm(Transition))));
    new(State.in(InRel,sm(sm(Transition)),Transition));
  }
  call endTimer("createTransitionTargets");
}

// simple type wrapper for Transition
pattern Transition(Transition) = {
  Transition(Transition);
}

// add triggers to transition
rule addTrigger(in ActivateClassRef,in Transition) = seq{
  call startTimer("addTrigger");
  println(buf(), "  RULE: Creating trigger for " + name(Transition));
  // finds the method where the activate() methodcall happens
  try choose CallingClassMethod with
   find ParentClassMethod(CallingClassMethod, ActivateClassRef) do
    let Trigger = undef, TriggerRel = undef,
    TriggeringElement = undef in seq{
    println(buf(), "  --> Found class method "
     + name(CallingClassMethod));
    try choose MethodName with
     find NameOfElement(MethodName,CallingClassMethod) do seq{
  /* 1. If activation of the next state occurs in any method except run(),
   then that method's name (members.Method.name) shall be
   used as the trigger. */
      if(value(MethodName) != "run") seq{
```

```
          update TriggeringElement = CallingClassMethod;
        }
    /* 2. If the activation of the next state occurs inside a non-default
     case block (statements.NormalSwitchCase) of a switch statement
     (statements.Switch) in the run() method, then the enumeration con-
     stant (members.EnumConstant) used as condition of the corresponding
     case is the trigger. */
        else seq{
            try choose SwitchCaseConstant with
             find ParentSwitchCaseConstant(SwitchCaseConstant,
              CallingClassMethod, ActivateClassRef) do
              seq{
                println(buf(), "  --> Found case " + name(SwitchCaseConstant));
                update TriggeringElement = SwitchCaseConstant;
            }
    /* 3. If the activation of the new state occurs inside a catch block
     (statements.CatchBlock) inside the run() method,
       then the trigger is the  name of the caught exception's class.*/
           else try choose CatchBlockClass with
            find ParentCatchBlockClass(CatchBlockClass,
             CallingClassMethod, ActivateClassRef) do
             seq{
               println(buf(), "  --> Found catch " + name(CatchBlockClass));
               update TriggeringElement = CatchBlockClass;
           }
    /* 4. If none of the three cases above can be matched for the activation
     of the next state, i.e., the activationcall is inside the run() method
     but without a surrounding switch or catch, the corresponding transition
     is triggered unconditionally. In that case, the trigger attribute shall
     be set to --. */
            else seq{
              println(buf(), "  --> Unconditional trigger");
            }
        }
        new(EString(Trigger) in Transition); // creating trigger
        new(Transition.trigger(TriggerRel,Transition,Trigger));
        if(TriggeringElement != undef)
          try choose Name with
           find NameOfElement(Name,TriggeringElement) do seq{
                // use name of chosen element
                setValue(Trigger,value(Name));
          }
        else setValue(Trigger,"--");
      }
     }
     call endTimer("addTrigger");

}

// finds the class method for a given reference
pattern ParentClassMethod(CallingClassMethod, IdentifierRef) = {
  ClassMethod(CallingClassMethod);
  IdentifierReference(IdentifierRef) below CallingClassMethod;
}

// finds the immediate parent switchcase constant for a reference
pattern ParentSwitchCaseConstant(SwitchCaseConstant,
 ClassMethod, IdentifierRef) = {
  NormalSwitchCase(NormalSwitchCase);
  // parent switchcase
  find ParentSwitchCase(NormalSwitchCase,
   ClassMethod, IdentifierRef);
  // condition of switch
  Conditional.condition(ConditionRel,NormalSwitchCase,Condition);
  IdentifierReference(Condition);
```

```
  EnumConstant(SwitchCaseConstant);
  // referenced constant
  find ReferenceTarget(Condition,NormalSwitchCase,SwitchCaseConstant);
}

// finds immediate parent switchcase, check for lowest parent
pattern ParentSwitchCase(NormalSwitchCase, ClassMethod, IdentifierRef) = {
  ClassMethod(ClassMethod);
  Switch(Switch) below ClassMethod;
  NormalSwitchCase(NormalSwitchCase);
  Switch.cases(CaseRel,Switch,NormalSwitchCase);
  IdentifierReference(IdentifierRef) below NormalSwitchCase;
  // if there is a lower switch, that must be used
  neg find LowerSwitch(Switch, IdentifierRef);
}

// checks whether a lower switch exists between Switch and the reference
pattern LowerSwitch(Switch, IdentifierRef) = {
  Switch(Switch);
  Switch(LowerSwitch) below Switch;
  IdentifierReference(IdentifierRef) below LowerSwitch;
}

// finds the class of the exception used in the parent catch block
pattern ParentCatchBlockClass(CatchBlockClass, ClassMethod, IdentifierRef) = {
  CatchBlock(CatchBlock);
  // parent catch block
  find ParentCatchBlock(CatchBlock, ClassMethod, IdentifierRef);

  CatchBlock.parameter(ParRel,CatchBlock,Parameter);
  // targeted parameter
  find ReferenceTargetOfParameter(Parameter,CatchBlockClass);
}

// finds target for parameter through type reference
pattern ReferenceTargetOfParameter(Parameter,Target) = {
  OrdinaryParameter(Parameter);
  TypedElement.typeReference(TypeRef,Parameter,NSClassRef);
  find TargetOfNamespaceClassifierReference(NSClassRef, Target);
}

// finds immediate parent catch block for reference
pattern ParentCatchBlock(CatchBlock, ClassMethod, IdentifierRef) = {
  ClassMethod(ClassMethod);
  TryBlock(TryBlock) below ClassMethod; // the try block where the catch is
  CatchBlock(CatchBlock);
  TryBlock.catcheBlocks(BlockRef,TryBlock,CatchBlock);
  IdentifierReference(IdentifierRef) below CatchBlock;
  // if there is a lower catch, that must be used
  neg find LowerCatchBlock(CatchBlock, IdentifierRef);
}

// checks whether a lower catch exists between CatchBlock and the reference
pattern LowerCatchBlock(CatchBlock, IdentifierRef) = {
  CatchBlock(CatchBlock);
  CatchBlock(LowerCatchBlock) below CatchBlock;
  IdentifierReference(IdentifierRef) below LowerCatchBlock;
}

// add action to transition
rule addAction(in ActivateClassRef,in Transition) = seq{

  call startTimer("addAction");
  println(buf(), "  RULE: Creating action for " + name(Transition));
  // finds the statement container containing the methodcall
```

```
    try choose StatementContainer with
     find ParentStatementContainer(StatementContainer, ActivateClassRef) do
     let Action = undef, ActionRel = undef in seq{
      println(buf(), "  --> Found container " + name(StatementContainer));
      new(EString(Action) in Transition);
      new(Transition.action(ActionRel,Transition,Action));
    /* 1. If the block (statements.StatementListContainer) containing the ac-
380  tivation Call of the next state additionally contains a method Call to the
     send() method, then that call's enumeration constant parameter's name is
       the action. */
      try choose SendMethodParameter with
       find SendMethodParameterInContainer(SendMethodParameter,
        StatementContainer) do
         try choose Name with
          find NameOfElement(Name,SendMethodParameter) do seq{
                println(buf(), "  --> Found send () parameter "
                 + name(SendMethodParameter));
390             setValue(Action,value(Name));
       }
    /* 2. If there is no Call to send() in the activation call's block,
     the action of the corresponding transition shall be set to --. */
       else seq{
         println(buf(), "      --> No send() in block.");
         setValue(Action,"--");
       }
    }
    call endTimer("addAction");
400 }

  // finds parent statement container
  pattern ParentStatementContainer(StatementContainer, Expression) = {
    StatementListContainer(StatementContainer);
    ExpressionStatement(Statement);
    StatementListContainer.statements(StatementsRel,
     StatementContainer,Statement);
    ExpressionStatement.expression(ExprRel,Statement,Expression);
    Expression(Expression);
410
  }

  /*  finds the EnumConstant used as the Parameter of a send()
   method in a statement container */
  pattern SendMethodParameterInContainer(SendMethodParameter,
   StatementContainer) = {
    StatementListContainer(StatementContainer);
    // parent container
    find ParentStatementContainer(StatementContainer, SendMethodCall);
420
    find MethodCall(SendMethodCall,SendMethod); // methodcall
    find NameOfElement(SendName,SendMethod);
    check(value(SendName) == "send"); // ensure that it is a send()

    find ArgumentOfMethodCall(Argument,SendMethodCall); // argument of send()
    Reference.next(NextRef,Argument,EnumRef);
    // target of the argument
    find ReferenceTarget(EnumRef,Argument,SendMethodParameter);

430 }

  /* finds corresponding arguments for a methodcall */
  pattern ArgumentOfMethodCall(Argument,MethodCall) = {
    MethodCall(MethodCall);
    Argumentable.arguments(ArgRel,MethodCall,Argument);
    Expression(Argument);
  }
```

```
  /* starts the timer corresponding to the RuleName */
  rule startTimer(in RuleName) = seq{
    if(time(RuleName) == undef)
      update time(RuleName) = - systime();
    else
      update time(RuleName) = time(RuleName) - systime();
  }

  /* stops the timer corresponding to the RuleName */
  rule endTimer(in RuleName) = seq{
    if(time(RuleName) == undef)
      update time(RuleName) = 0;
    else
      update time(RuleName) = time(RuleName) + systime();
  }

}
```

Listing 1: Transformation code