Program Understanding case study solution using the Viatra2 framework

Ábel Hegedüüs, Zoltán Ujhelyi, and Gábor Bergmann
Budapest University of Technology and Economics, Hungary
{hegedusa,ujhelyiz,bergmann,ahorvath}@mit.bme.hu

Abstract. The current paper presents a solution of the Model Transformations for Program Understanding: A Reengineering Challenge case study of the Transformation Tool Contest 2011, using the VIATRA2 model transformation tool.

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

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 [1]) 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 [2] 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.

Transformation description. Specification of model transformations in VIATRA2 combines the visual, declarative rule and pattern based paradigm of graph transformation (GT) and the very general, high-level formal paradigm of abstract state machines (ASM) [4] into a single framework for capturing transformations within and between modeling languages [5].
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 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 introduces the Case Study problem which is solved in this paper. Sec. 3 gives an architectural overview of the transformation, while Sec. 4 highlights the interesting parts of our implementation and finally Sec. 5 concludes the paper.

2 Case study

The following case study description is copied verbatim from [7].

The maintenance and reengineering of complex legacy software systems requires good understanding of the behavior of the system. However, these systems are often developed using monolithic techniques and without adequate developer documentation. In order to help developers in understanding a legacy system, automated techniques should be provided to derive essential information from the source code of the system and present it in a more easily understandable form.

As a challenge, [7] proposes the implementation of such a technique within a model transformation framework. The SOAMIG project deals with the migration of legacy systems to Service-Oriented Architectures by means of model-driven techniques. In the project, a GReTL transformation has been developed, which creates a simple state machine model consisting of states and transitions with triggers and actions out of the syntax graph of the legacy system. The resulting state machine model contains all information about the possible sequences.

The overall goal of this task is to create a very simple state machine model for a Java syntax graph model encoding a state machine with a set of coding conventions a transformation has to exploit. The primary source model of the transformation is the Java abstract syntax graph conforming to the Java metamodel provided as Ecore file. The model is generated by parsing the source code of a project using the JaMoPP (Java Model Parser and Printer, [8]) tool developed at the Technical University of Dresden. The proposed transformation demonstrates two useful features of transformation frameworks: (1) performance and scalability and (2) support for complex, non-local matching.

3 Solution Architecture

We implemented our solution for the case study 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.

1 http://www.soamig.de
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.

4 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) [9]. 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 173.

8 complex patterns are handled by the incremental, the other complex patterns by the local search pattern matcher. Finally, the actual manipulation is executed by 5
declarative rules (e.g. create trigger for a given transition, see line 228). There are 2
additional rules for starting and stopping time measurement for different parts of the
transformation (see lines [441] and [449]).

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 statemach-
ine 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 [99] called in line [47] from a forall construct).
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 [65]).
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 found with an other complex pattern that finds the
called class for *Class.Instance.activate()* method calls (see line [173]).
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 228) is called from
line [165]. The triggers are created based on the class method, where the *activate()* call
is found (see pattern in line [292]), the switch case constant (line [298]) or the catch
block exception (line [331]) 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 [371]). The action is created based on the existence of a *send()* method call in the
same statement container (found using the pattern in line [406]) 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 [207]). The interesting part of this rule is
the usage of ASM functions to retrieve the correct target state (line [216]). 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 imple-
mentation. 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 con-
suming 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\(^2\)), 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.

5 Conclusion

In the current paper we have presented our VIATRA2 based implementation for the Program Understanding case study \(^7\).

Solution specific conclusions. 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.

Case study specific conclusions. During the implementation of our solution, we concluded the following:

– EMF is used in almost all case studies as way to provide the same metamodels and input models for each participant, while some case studies also build on further EMF technologies. These properties may be significant obstacles for not EMF (or even Java) based tools, such as GrGen, VMITS or GROOVE.

– While we can understand and agree with the usage of EMF as a way to distribute input models, we feel that these models should be consistent and complete, i.e. the built-in EMF tools should be able to open them without errors. In this case, the input models included references to files of the Java SDK and included model elements with no particular contribution to the problem (e.g. files from java.lang.*).

References


A Solution demo and implementation

Our solution for the Hello World! case can be downloaded from [http://mit.bme.hu/~ujhelyiz/viatra/ttc11-helloworld.zip](http://mit.bme.hu/~ujhelyiz/viatra/ttc11-helloworld.zip)


The SHARE image for demonstration purposes is available at [http://is.tm.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdid=Ubuntu-11_TTC11_VIATRA. vdi](http://is.tm.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdid=Ubuntu-11_TTC11_VIATRA. vdi). The image contains our solution for both the Hello World! and Program Understanding cases.

B Appendix - Program Understanding transformation

```java
// 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.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;
}

// find 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));
      update sm(StateClass) = State; // store Class -> State correspondence
      // 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));
        }
      call createTransitions (StateClass); // create transitions from state
    }
  // for each Transition, find 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"));
}

// find classes which are subtypes of State
@localsearch
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);
}

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

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

// find supertype of class
@localsearch
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
@localsearch
pattern TargetOfNamespaceClassifierReference(NSCRef,Target) = {
  NamespaceClassifierReference(NSCRef);
  NamespaceClassifierReference.classifierReferences(R,NSCRef,ClassRef);
  ClassifierReference(ClassRef);
  ClassifierReference.target(TargetRel,ClassRef,Target);
}

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

// create transitions leading out from StateClass
rule createTransitions(in StateClass) = seq{
  call startTimer("createTransitions");
  // find all transition in class
  // 2. A Transition is encoded by a method call (references.MethodCall),
  // which invokes the next state’s Instance() method (members.Method) re-
  //turning 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,ActivateCRef with
    find ClassCalledWithActivate(ActivateCallClass,ActivateCRef,StateClass)
do let Transition = undef, TransRel = undef,
  SrcRel = undef, OutRel = undef in seq{
    println(buf()," --> Found activate() call 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 to find 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(ActivateCRef, Transition);
  // add action
  call addAction(ActivateCRef, Transition);
class endTimer("createTransitions");

// find the class which is called using an activate() method
pattern ClassCalledWithActivate(ActivateClassRef, ActivatedClassRef, StateClass) = {
    find ClassSubTypeOfState(StateClass); // check that the class is a state
    // reference to Class
    find ReferenceTarget(ActivatedClassRef, StateClass, ActivateClassRef);
    Reference.next(ACRNextRef, ActivatedClassRef, InstanceCall);
    // reference to Instance method
    find MethodCall(InstanceCall, ActivateClassInstance);
    Reference.next(ERNextRef, InstanceCall, ActivateCall);
    find NameOfElement(Name, ActivateClassInstance); // name of Instance
    check(value(Name) == "Instance");
    // reference to activate() method
    find MethodCall(ActivateCall, ActivateMethod);
    find NameOfElement(ActName, ActivateMethod);
    check(value(ActName) == "activate");
}

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

// find method called by Caller
@localsearch
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 pattern
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));
    // find the method where the activate() method call 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 activation call 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
        if(TriggeringElement != undef)
        try choose Name with
        find NameOfElement(Name, TriggeringElement) do seq{
            setValue(Trigger, value(Name)); // use name of chosen element
        }
        else setValue(Trigger, "--");
    }
    call endTimer("addTrigger");
}

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

// find 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);
}

// find 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);
}

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

// find 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);
}

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

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

// check whether a lower catch exists between CatchBlock and the reference
@localsearch
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));
// find the statement container containing the method call
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 activation 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));
          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");
  }

// find parent statement container
@localsearch
pattern ParentStatementContainer(StatementContainer, Expression) = {
  StatementListContainer(StatementContainer);
  ExpressionStatement(Statement);
  StatementListContainer.statements(StmtsR,StatementContainer,Statement);
  ExpressionStatement.expression(ExprRel,Statement,Expression);
}
/* Find 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);
  find MethodCall(SendMethodCall,SendMethod); // method call
  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);
}
/* Find corresponding arguments for a method call */
@localsearch
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 1.1. Transformation code