Model Transformations for Program Understanding: A Reengineering Challenge

Dipl.-Inform. Tassilo Horn
horn@uni-koblenz.de

University Koblenz-Landau, Campus Koblenz
Institute for Software Technology
Universitätsstraße 1, 56070 Koblenz, Germany

March 23, 2011

Abstract

In Software Reengineering, one of the central artifacts is the source code of the legacy system in question. In fact, in most cases it is the only definitive artifact, because over the time, the code has diverged from the original architecture and design documents. The first task of any reengineering project is to gather an understanding on what the system does, and how it does it. Therefore, a common approach is to use parsers to translate the source code of the legacy system into a model conforming to the abstract syntax of the programming language the system is implemented in, which can then be subject to querying. Despite querying, transformations can be used to regenerate more abstract models of the system's architecture.

This transformation case deals with the creation of a state machine model out of a Java syntax graph model. It is derived from a task that originates from a real reengineering project.

1 Objective of the Case

The objective of the reengineering case presented here is to transform an abstract syntax graph into a simple state machine.

There are two major challenges involved in this transformation task. The first challenge is an issue of *performance* and *scalability*. The input model of the transformation is an abstract syntax graph, which is naturally large for any reasonable program. In this concrete case, the smallest input model contains more than 6,500 elements, although it is an abstract syntax graph of only a very small program. Larger models with up to a million elements might be used during the evaluation to stress-test the used transformation language implementations.

The second and more important challenge is that the transformation task involves *complex, non-local matching* of elements. For example, the core task, which is described in Section 3, demands that the transformation should create one state for each Java class that derives directly or indirectly from an abstract Java class named ``State.'' There are no restrictions on the depth of the inheritance hierarchy, so the ``State'' class and its subclasses may be located arbitrarily far in the input model. However, the structure of the path from subclass to superclass is clearly specified by the input metamodel and must be utilized by transformations, e.g., by recursive pattern matching or similarily expressive matching constructs, an example being subpattern matching as provided by GrGen.NET [JBK10].

2 Context of the Case

The SOAMIG[1] project deals with the migration of legacy systems to Service-Oriented Architectures by means of model-driven techniques. One of the two legacy systems on which the approach is being evaluated is a monolithic Java system, which is operated with a Swing graphical user interface.

This user interface consists of around 30 different masks, which often relate to conceptually self-contained functionalities that might be implemented as services in the reengineered target system. Furthermore, the order in which masks are activated and which successor masks can be activated from a given mask gives good hints about the orchestration of the target system services.

The masks are implemented as plain Java classes using the Swing toolkit. Many masks are very complex with many user interface elements and even more input validation code, which complicates tracking down the relationships between the individual masks. However, the graphical user interface is based on a state machine concept, and uses some strict coding conventions in the implementation. As such, any masks can be seen as states, and when another mask is activated, it can be seen as a transition. The trigger of this transition is usually a click on some button, and possibly additional actions are performed just before the transition, like validating and storing the user input.

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. This syntax graph consists of more than 2.5 million nodes and edges. The transformation heavily exploits the coding conventions taken as a basis for the implementation of the graphical user interface. The resulting state machine model contains all information about the possible sequences in which masks can be activated, what triggers are responsible for a transition, and what additional actions are performed when transitioning. However, it consists of less than 100 nodes and

[1] `http://www.soamig.de`

edges, and it can be visualized and printed. Therefore, it is of great value for the understanding of the legacy system.

The transformation case proposed here is directly derived form this reengineering project's task. Instead of using the syntax graph of the proprietary system, a toy example implementing the well-known TCP protocol state machine using very similar coding conventions is used.

The next section describes how to get the up-to-date versions of this description and the relevant metamodels and models. Section 4 then provides a detailed task description with one core task and two extensions. The evaluation criteria that are used to judge the solutions are discussed in Section 5.

3 The Task Description Project

The project that includes this description, the input Ecore metamodel, three input EMF syntax graph models, and the Java source code that was parsed to generate two of the input models, as well as the output Ecore metamodel can be cloned from the case submitter's Mercurial[2] repository.

To do so, use the following command on the command line, or an equivalent instruction in the Mercurial GUI of your choice[3]. Everyone has read access using the username *anonymous* and the password *secret*.

```
% hg clone \
    https://anonymous:secret@hg.uni-koblenz.de/horn/tcp-state-extract
```

This will create a new folder `tcp-state-extract` in the current working directory containing the project contents. The subdirectory `case` contains the task description, the subdirectory `src` contains the Java source code that is parsed to generated the syntax graph model `jamopp/1_small-model.xmi` conforming to the Ecore metamodel `jamopp/java.ecore`. The subdirectory `src2` contains a slightly more complex variant of the code in `src`, and from that the second input model `jamopp/2_medium-model.xmi` is generated. The target metamodel is `statemachine/StateMachine.ecore`.

The reference solution implemented using the GReTL transformation language [HE11] is also included as `solution/ExtractStateMachines.gretl`.

To update the task description project, use:

```
% hg pull -u
```

Alternatively, an archive file containing the complete repository contents can be downloaded from `http://www.uni-koblenz.de/~horn/tcp-state-extract.tar.gz`. This archive also contains the mercurial metadata, so if you decide to install mercurial later on, you can update to the most recent version with `hg pull -u` as above.

[2]`http://mercurial.selenic.com/`

[3]e.g., *TortoiseHG*, available at `http://tortoisehg.bitbucket.org/`

The case should be discussed at the TTC 2011 discussion groups[4], and any errors or obscurities in either the description itself or in the models will be fixed in time.

4 Detailed Task Description

In this section, the transformation task is explained in details.

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 task is divided into one mandatory *core task* (Section 4.1) and two optional *extension tasks* with slightly increased complexity (Section 4.2 and Section 4.3).

The conventions used to implement the state machine in Java that should be exploited by transformations are explained in terms of concrete Java syntax, i.e., by using source code examples. However, the most relevant metamodel elements are named, too. In the following, source metamodel elements are written underlined (e.g., `MethodCall`), and target metamodel elements are typeset with a typewriter font (e.g., `Transition`).

Source Metamodel and Models. The primary source model of the transformation is the Java abstract syntax graph `jamopp/1_small-model.xmi` conforming to the Java metamodel provided as Ecore [SBPM09] file `jamopp/java.ecore`. The model is generated by parsing the source code in the `src` directory of the task description project using the *JaMoPP* (*Java Model Parser and Printer*, [HJSW09]) tool developed at the Technical University of Dresden.

The source code in the src directory whose abstract syntax graph is the input model of the transformation task is a toy-implementation of the TCP-protocol state machine[5]. The coding conventions that transformations have to utilize are discussed in the descriptions of the core and extension tasks below.

The input model contains any information present in the source code. It consists of about 6,500 elements.

The second input model `jamopp/2_medium-model.xmi` is generated by parsing the source code contained in the `src2` directory. It is slightly larger (6,800 elements), and it is more complex, e.g., the specialization hierarchy is deeper, and there is a deeper nesting of statements in method bodies.

A third input model `jamopp/3_big-model.xmi` is also provided. It was generated by parsing the source code of a Java project containing about 900 classes and 220.000 lines of code. Into this project, the classes implementing the TCP state machine were integrated. This model has about the size and complexity of the model in the original reengineering scenario.

[4]Follow the *Groups* link at `http://planet-mde.org/ttc2011`.

[5]`http://en.wikipedia.org/wiki/Transmission_Control_Protocol`

All provided input models implement the TCP state machine according to the conventions specified in the task description below, so the target state machine model is always the same (without respect to the order of elements).

Although the input models are provided as EMF models, the transformation tools are not restricted to that technological space [KBA02]. To make sure that all solutions use equivalent models, both in terms of the metamodel as well as the model sizes, it may be imported into the native format of the transformation framework, but it is not allowed to create new input models using custom parsers.

Target Metamodel. As target metamodel, the very basic state machine metamodel shown in Figure 1 is used.

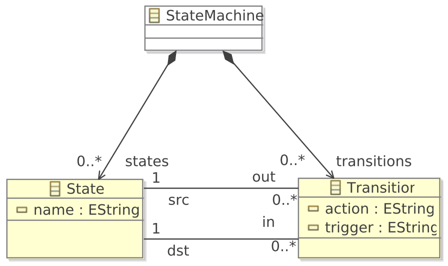

Figure 1: The target Ecore metamodel

A `StateMachine` consists of an arbitrary number of `States` and `Transitions`.

Any `Transition` starts at exactly one `src-State`, and it ends at exactly one `dst-State`. Every `State` may have an arbitrary number of outgoing `Transitions` (`out`), and an arbitrary number of incoming `Transitions` (`in`).

Every `State` has a `name`, and any `Transition` may be caused by a `trigger`, and as a result of its activation, an `action` might be performed.

If possible, transformations should produce a valid EMF XMI [OMG07] instance of the statemachine/StateMachine.ecore metamodel. However, if the used transformation language is not built upon EMF and doesn't have an EMF export facility, then it is acceptable to provide the output model in some understandable output format. For example, a visualization of the state machine could be appropriate (cf. Figure 4), or a plain-text representation.

4.1 The Core Task

The core task should create a state machine model that contains all entities, i.e., all `States` and all `Transitions` with the appropriate references set. Additionally, the `name` attribute defined for the `State` class must be set. The initialization of the `trigger` and `action` attributes are left for the extension tasks.

In the following, the coding conventions used to implement the TCP state machine in plain-java are discussed in terms of concrete Java syntax and by using the SynSent class contained in the `src` directory, which is shown in Listing 1.

```java
public class SynSent extends ListeningState {
        private static State instance;

        public static State Instance() {
                if (instance == null) {
                        instance = new SynSent();
                }
                return instance;
        }

        public void close() {
                Closed.Instance().activate();
        }

        @Override
        protected void run() {
                switch (getReceivedFlag()) {
                case SYN:
                        send(Flag.SYN_ACK);
                        SynReceived.Instance().activate();
                        return;
                case SYN_ACK:
                        send(Flag.ACK);
                        Established.Instance().activate();
                        return;
                default:
                        break;
                }
        }
}
```

Listing 1: The SynSent class

The coding conventions relevant for the core task are as follows:

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].

   In the example from Listing 1, SynSent extends the abstract ListeningState state class, and that in turn extends the abstract State class.

2. A Transition is encoded by a method call (`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.

   Transformation may assume that the activation of the next state always has the form NewState.Instance(). activate(). The case that the singleton instance is stored in a variable or returned by some other method except for Instance() can be neglected.

In the example, there are three transitions.

In the close() method, there is one transition from the current state (SynSent) into the Closed state (line 12).

In the run() method, there are another two transitions. In line 20, there is a transition into the SynReceived state, and in line 24, there is a transition into the Established state.

The target model state names should be set according to the Java classes they were created for.

The outcome of the core task is a state machine with 11 states and 21 transitions between the states. A visualization of that state machine produced by the reference solution is shown in Figure 2.

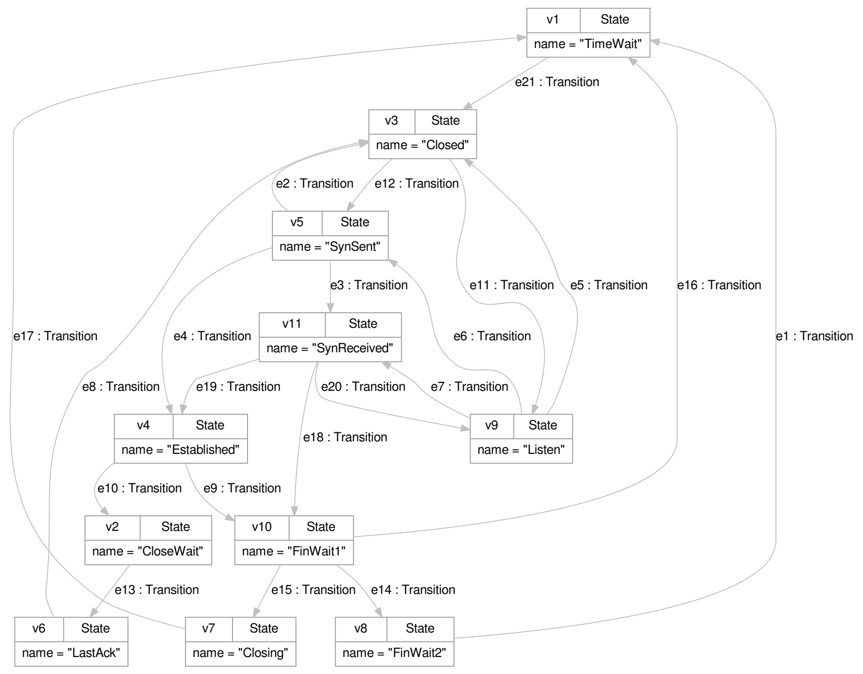

Figure 2: The state machine produced according to the core task

4.2 Extension 1: Triggers

This extension task deals with the trigger attribute of transitions. There are three different coding conventions that a transformation has to exploit to set the correct trigger value. These three conventions and one fallback rule are specified as follows.

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.

   For example, the activation of Closed in line 12 of Listing 1 occurs in the close() method, so the trigger is close.

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 constant (`members.EnumConstant`) used as condition of the corresponding case is the trigger.

   For example, when activating SynReceived in line 20 of Listing 1, the trigger is SYN. When activating Established in line 24, the trigger is SYN_ACK.

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.

   For example, when activating the Closed state in line 9 of Listing 2, the trigger is TimeoutException.

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 −−.

```
1  public class TimeWait extends State {
2          // some code elided ...
3
4          @Override
5          protected void run() {
6                  try {
7                          timeWait();
8                  } catch (TimeoutException e) {
9                          Closed.Instance().activate();
10                 }
11         }
12 }
```

Listing 2: Parts of the TimeWait class

Transformations can assume that the four different cases can be matched without ambiguity, e.g., there is no catch block activating some state inside a surrounding switch, or vice versa.

The target state machine that is produced by the reference transformation implementing the core and the first extension task is shown in Figure 3.

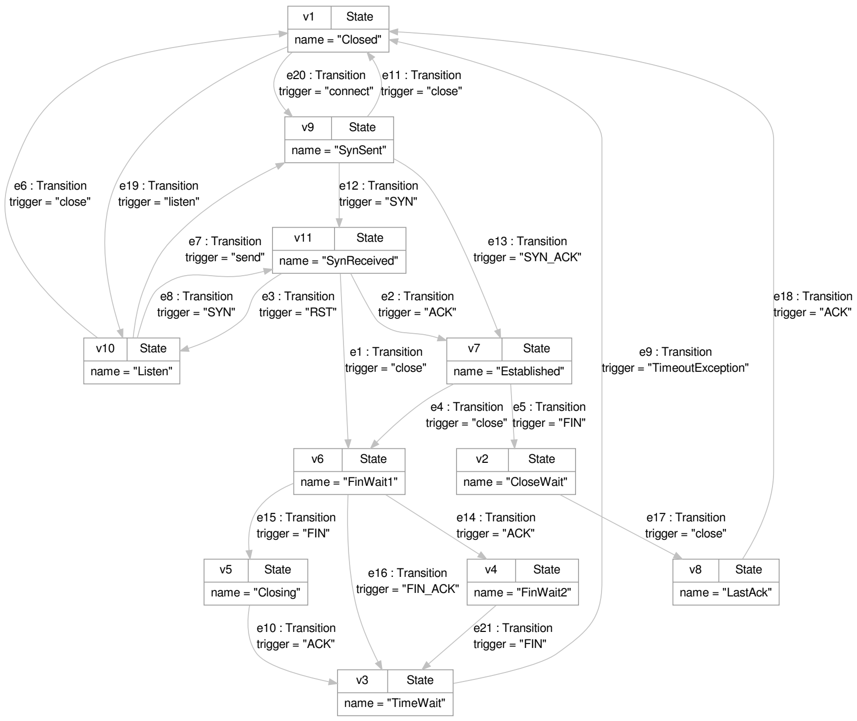
v1 State
name = "Closed"
e20 : Transition trigger = "connect"
e11 : Transition trigger = "close"
v9 State
name = "SynSent"
e6 : Transition trigger = "close"
e19 : Transition trigger = "listen"
e12 : Transition trigger = "SYN"
e7 : Transition trigger = "send"
v11 State
name = "SynReceived"
e13 : Transition trigger = "SYN_ACK"
e8 : Transition trigger = "SYN"
e3 : Transition trigger = "RST"
e2 : Transition trigger = "ACK"
e18 : Transition trigger = "ACK"
v10 State
name = "Listen"
e1 : Transition trigger = "close"
v7 State
name = "Established"
e9 : Transition trigger = "TimeoutException"
e4 : Transition trigger = "close"
e5 : Transition trigger = "FIN"
v6 State
name = "FinWait1"
v2 State
name = "CloseWait"
e15 : Transition trigger = "FIN"
e14 : Transition trigger = "ACK"
e17 : Transition trigger = "close"
v5 State
name = "Closing"
e16 : Transition trigger = "FIN_ACK"
v4 State
name = "FinWait2"
v8 State
name = "LastAck"
e10 : Transition trigger = "ACK"
e21 : Transition trigger = "FIN"
v3 State
name = "TimeWait"

Figure 3: The state machine with the trigger attribute set according to the first extension task

4.3 Extension 2: Actions

The task of the second extension is to set the action attributes of transitions. The action of a transition is specified as follows.

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.

   For example, the action attribute of the transition corresponding to the state activation call in line 20 of Listing 1 has to be set to SYN_ACK. The action attribute of the transition corresponding to the activation call in line 24 has to be set to ACK.

2. If there is no call to send() in the activation call's block, the action of the corresponding transition shall be set to −−.

For example, the activation call in line 12 of Listing 1, and the activation call in line 9 of Listing 2 perform no action, and thus the attribute has to be set to $--$.

A visualization of the complete target state machine produced by the reference transformation is shown in Figure 4.

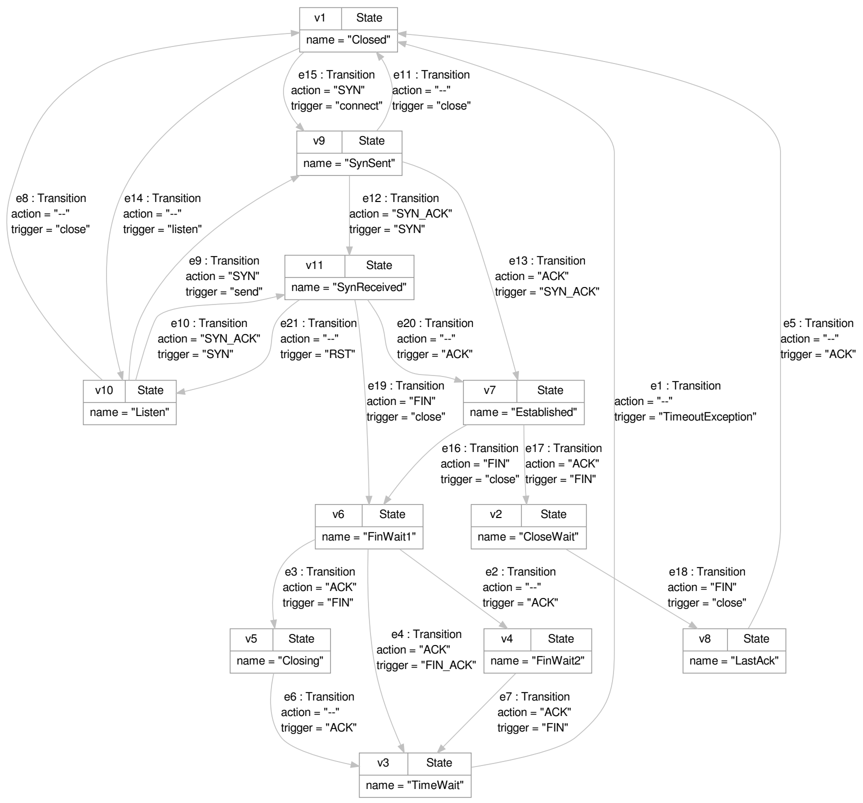

Figure 4: The final state machine after performing the core and both extension tasks

5 Evaluation Criteria

The case project contains an OpenDocument spreadsheet `evaluation/evaluation_sheet.ods`, which is used to judge the solutions.

As motivated in Section 2, the goal of this transformation case is supporting program understanding. By facilitating a set of coding conventions, model transformations can be used to accomplish the task of extracting the implicitly encoded state machine, in contrast to modeling it manually by thoroughly inspecting all relevant classes of the system in question.

In order to have a feasible solution, the time needed for writing and executing the transformation must be comparable to the time that would be needed for a code inspection and manually modelling the state machine. However, if we assume that the set of coding conventions derived from the initial brief inspection is correct, we can also assume that the transformation produces a correct output without human mistakes.

Since the speed of writing a solution cannot be judged directly, we assume that **understandability** and **conciseness** of the solution are the objectively ascertainable properties that directly relate to the time needed for implementing that solution. Ideally, each coding convention described in Section 4 results in a transformation rule, and the statement of the convention is clearly visible.

The **correctness** of the solution is another important point. If the model created by the transformation is the foundation of weighty decisions in a reengineering project, it would be fatal if the transformation created a model that doesn't reflect the reality.

In that respect, the **completeness** of a solution is closely entangled to correctness, because an incomplete model may also lead to false decisions.

The last property that will be judged is the **performance**. It is much less important than the other criteria, but of course such a transformation should be applicable on common hardware in acceptable time.

References

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[HE11] Tassilo Horn and Jürgen Ebert. The GReTL Transformation Language. Technical report, University Koblenz-Landau, Institute for Software Technology, 2011. to appear, draft at `http://www.uni-koblenz.de/~horn/gretl.pdf`.

[HJSW09] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. JaMoPP: The Java Model Parser and Printer. Technical Report TUD-FI09-10, Technische Universität Dresden, Fakultät Informatik, 2009. `ftp://ftp.inf.tu-dresden.de/pub/berichte/tud09-10.pdf`.

[JBK10] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. GrGen.NET. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(3):263--271, July 2010.

[KBA02] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, 2002.

[OMG07] OMG. MOF 2.0 / XMI Mapping Specification, v2.1.1, 2007.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition) (Eclipse)*. Addison-Wesley Longman, Amsterdam, 2nd revised edition (rev). edition, January 2009.