Program Understanding: A Reengineering Case for the Transformation Tool Contest

Dipl.-Inform. Tassilo Horn

horn@uni-koblenz.de
Institute for Software Technology
University Koblenz-Landau, Campus Koblenz

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 of the system's architecture. Therefore, a common approach is to use parsers to translate the source code 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 generate more abstract views on the system's architecture.

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

1 Objective and Context

The objective of the reengineering case presented here is to transform an abstract Java syntax graph into a simple state machine. There are two major challenges involved. The first challenge is an issue of *performance* and *scalability*, because the input models are naturally large.

The second and more important challenge is that the transformation task involves *complex, non-local matching* of elements. For example, the core task, 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.

The SOAMIG[1] project dealt with the migration of legacy systems to Service-Oriented Architectures by means of model-driven techniques. One legacy system on which the approach has been evaluated is a monolithic Java system, which is operated with a 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. 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 user interface is based on a state machine concept and uses 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

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

transition is usually a click on some button, and possibly additional actions are performed just before the transition, e.g., validating user input.

A GReTL [4] 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 consisting of more than 2.5 million nodes and edges. The transformation 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 edges, 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 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 the tasks including the relevant metamodels and models. The evaluation criteria used to judge the solutions are discussed in Section 3.

2 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* and two optional *extension tasks* with slightly increased complexity. 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 written with a typewriter font (e.g., `Transition`).

Source Metamodel and Models. The primary source model of the transformation is a Java abstract syntax graph conforming to the JaMoPP Ecore metamodel [2, 3]. The input model contains any information present in the source code. It consists of about 6,500 elements. A second input model is slightly larger and more complex, e.g., the specialization hierarchy is deeper and so is the nesting of statements in method bodies. Lastly, a third industry-size input model is provided, which was generated by parsing the source code of a Java project containing about 900 classes and 220,000 lines of code resulting in a model constisting of nearly one million elements.

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 (not accounting for the order of elements).

Target Metamodel. As target metamodel, the very basic state machine metamodel shown in Figure 1 is used. 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` has a `name`, and any `Transition` may be caused by a `trigger`, and as a result of its activation an `action` might be performed.

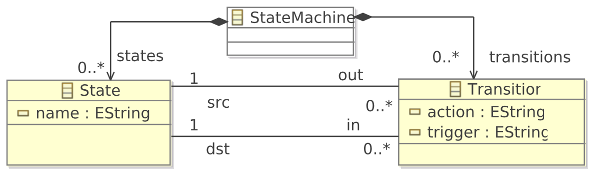

Figure 1: The target Ecore metamodel

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.

Below, 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.

```
 1 public class SynSent extends ListeningState {
 2   private static State instance = new SynSent();
 3         public static State Instance() { return instance; }
 4         public void close() { Closed.Instance().activate(); }
 5         protected void run() {
 6                 switch (getReceivedFlag()) {
 7                 case SYN: send(Flag.SYN_ACK);
 8                           SynReceived.Instance().activate();
 9                           return;
10                 case SYN_ACK: send(Flag.ACK);
11                               Established.Instance().activate();
12                               return;
13 } } }
```

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 [1]. In 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. It may be assumed that a transition always has the form NewState.Instance().activate(). 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 7). In the run() method, there are another two transitions. In line 11, there is a transition into the SynReceived state, and in line 14, 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.

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 the transition 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 7 of Listing 1 occurs in the `close()` method, so the trigger is `close`.
2. If the transition 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 11 of Listing 1, the trigger is SYN.
3. If the transition occurs inside a catch block (`statements.CatchBlock`) inside the `run()` method, then the trigger is the name of the caught exception's class.
4. If none of the three cases above apply, 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 −−.

Transformations may 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.

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 transition to the next state 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 in line 11 of Listing 1 has to be set to SYN_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 transition in line 7 of Listing 1 performs no action, and thus the `action` has to be set to −−.

A visualization of the target state machine produced by the reference solution is shown in Figure 2.

3 Evaluation Criteria

As motivated in Section 1, 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, it can be assumed that the transformation produces a correct output without human mistakes.

Since the speed of writing a solution cannot be judged directly, **understandability** and **conciseness** are used as objectively ascertainable measures relating to the implementation effort, weighted with 30%. Ideally, each coding convention described in Section 2 results in a transformation rule in which the statement of the convention is clearly visible.

The **correctness** of the solution is also important. If the model created by the transformation is the foundation of weighty decisions, it would be fatal if it didn't reflect the reality. The **completeness**

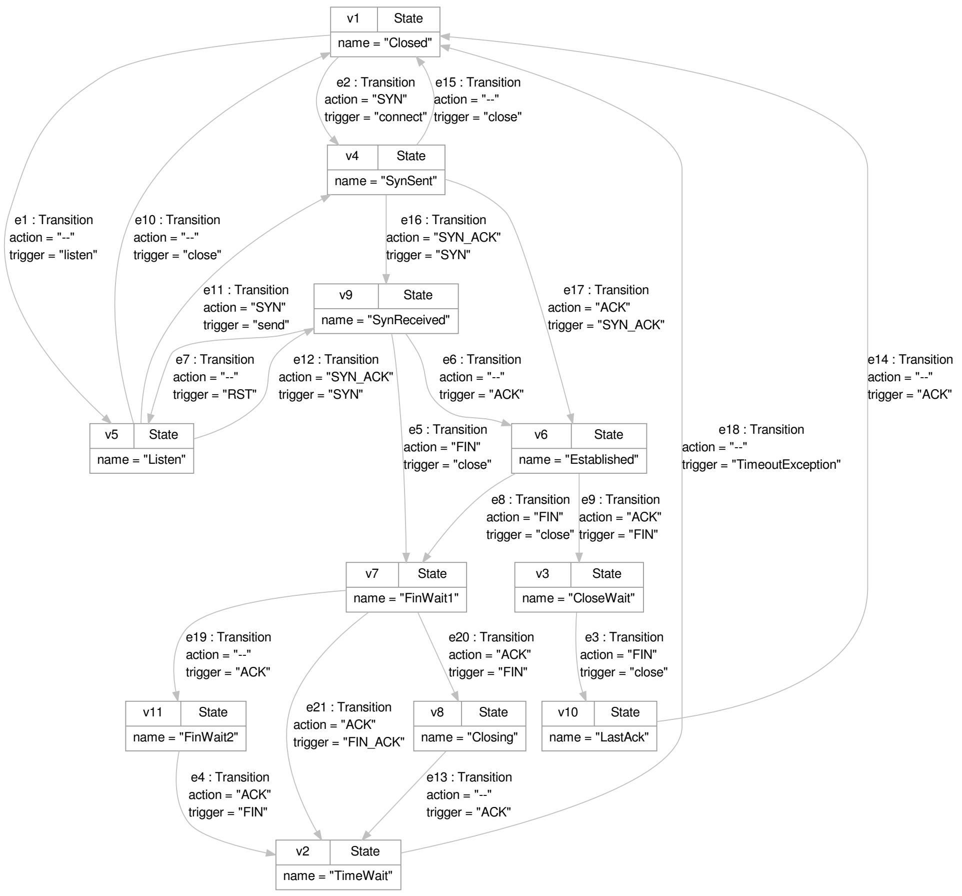

Figure 2: The final state machine after performing the core and both extension tasks

of a solution is closely entangled to correctness, because an incomplete model may also lead to false decisions. Both correctness and completeness are weighted with 17.5%.

The last property that will be judged is the **performance**, weighted with 5%. 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

[1] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (1995): *Design Patterns*. Addison-Wesley.

[2] Florian Heidenreich, Jendrik Johannes, Mirko Seifert & Christian Wende (2010): *Closing the Gap between Modelling and Java*. In M. van den Brand, D. Gasevic & J. Gray, editors: *Software Language Engineering*, *Lecture Notes in Computer Science* 5969, Springer, pp. 374–383, doi:10.1007/978-3-642-12107-4_25.

[3] Florian Heidenreich et al. (2009): *JaMoPP: The Java Model Parser and Printer*. Technical Report TUD-FI09-10, Technische Universität Dresden, Fakultät Informatik.

[4] Tassilo Horn & Jürgen Ebert (2011): *The GReTL Transformation Language*. In Jordi Cabot & Eelco Visser, editors: *Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, *Lecture Notes in Computer Science* 6707, Springer, pp. 183–197, doi:10.1007/978-3-642-21732-6_13.