Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 148 (2006) 173–186
www.elsevier.com/locate/entcs

Tool Modeling with Fujaba

Leif Geiger Albert Zündorf

University of Kassel, Software Engineering Research Group,
Wilhelmshöher Allee 73, 34121 Kassel, Germany
{leif.geiger|albert.zuendorf}@uni-kassel.de

Abstract

This paper is a small tutorial on tool building with Fujaba. With the help of a small case study, we exemplify how the different requirements of an environment for a visual language may be addressed using Fujaba graph transformations. This covers abstract and concrete syntax, static and operational semantics, and model transformations. This case study shows, how the more sophisticated language elements of Fujaba may be exploited in modeling complex aspects of the desired CASE tool. In addition, we address some not graph grammar related aspects in building such an environment, e.g. the graphical user interface and multi-user support.

Keywords: syntax and semantics of visual models, model transformation, meta CASE tool, Fujaba

1 Introduction

This paper ist an extended version of [4]. We show how the Fujaba approach may be used to build a modeling environment for simple statecharts. Although we try to keep our example as simple as possible, we will try to address all modeling aspects for a typical CASE tool, i.e. abstract syntax, model transformations, operational semantics, consistency checking, and concrete syntax.

The first section gives a short overview on the graph grammar approach supported by Fujaba. This is followed by a series of sections addressing the different aspects required for CASE tool building.

doi:10.1016/j.entcs.2005.12.017

2 Fujaba's graph transformation approach

Fujaba is a successor of the PROGRES language and environment, cf. [7]. Fujaba employs a graph transformation language with pretty similar language features and underlying semantics as the PROGRES system. This means, Fujaba is based on set theory. This may also be compared with the single pushout approach.

Fujaba employs typed graphs with labeled edges and nodes. Nodes have an identity and may have typed attributes. Edges are modeled by a set of triples, each consisting of a start node, an edge label, and a target node. This has the implication, that two edges with the same source and target node and the same label are not possible. Fujaba employs an explicit graph schema depicted as a UML class diagram. Conformance to the graph schema is enforced, statically, i.e. at compile time.

Usually, Fujaba graph transformations are restricted to injective matchings. However, a special *maybe* clause allows injective matchings where the gluing condition is enforced, statically. Fujaba allows to delete nodes in unknown context, i.e. we have no dangling edge condition. On the other hand, Fujaba graph transformations are not easily reversed. Fujaba has attribute conditions and simple negative application conditions. As a simple amalgamation concept, Fujaba employs optional and set-valued nodes that may match to multiple nodes in the host graph. Finally, Fujaba provides simple path expressions as means of abstraction for sequences of edges.

Fujaba graph transformations employ UML collaboration diagram notation, cf. Figure 3. This notation shows the left-hand and the right-hand side of a rule in a single diagram. Elements of the core graph are shown in black and without stereotype markers. Elements belonging to the left-hand side, only, i.e. elements that shall be deleted are marked by red color and a ≪*destroy*≫ marker. Elements that belong to the right-hand side, only, i.e. elements that are to be created are marked by ≪*create*≫ markers. Attribute conditions may be shown in the attribute compartments of the corresponding objects or in additional boolean expressions embraced by curly braces. Attribute modifications may be shown as assignments in the attribute compartments. In addition, UML collaboration messages may be used to call methods on objects or to embed arbitrary Java statements.

Fujaba employs programmed graph transformations. This is achieved by embedding the graph transformation rules in the activity boxes of a UML activity diagram, cf. Figure 4. This combination of activity diagram and graph transformation rules is called a *story diagram*. Story diagrams may employ branches and loops. Story diagrams are attached to method declarations in

the class diagram. So, story diagrams may have parameters and return values. In addition, an implicit *this* object is used referring to the object where the method is invoked on. Methods and thus story diagrams may be invoked using UML collaboration messages. We support recursion, object oriented inheritance, overriding of methods and thereby enable polymorphism.

Parameter names and object names have the whole story diagram as scope. During the application of a graph transformation, names of the employed objects are bound to nodes of the host graph. This binding may be reused in subsequent graph transformations by using the names without their type. Thus an object inscription like $n : T$ requires a new matching to a host graph node while an object inscription like n reuses the binding of a previous match or parameter values.

Graph transformations may inherently have multiple possible matches within a given host graph. Usually Fujaba chooses one of these matches, pseudo randomly (just the first that is found, depending on ordering in internal containers). Using a for-each activity indicated by a double activity shape, the search for matches is continued after an successful rule application until no more new matches are found. A rule application may also fail, at all. To deal with this, story diagrams provide *success* and *failure* transitions.

3 Abstract syntax

In the following sections, the Fujaba approach is used to model a simple visual language. We will model a CASE tool for a small subsets of UML statecharts containing states, transition and or-states. This CASE tool will enable editing of such statecharts, will add simulation support, will offer a model transformation to finite state machines and will support some basic consistency checking.

First, we will start modeling the abstract syntax of our statechart environment. The abstract syntax is frequently referenced as meta model. Fujaba employs explicit graph schemas that are defined using UML class diagrams. Thus, a simple meta model or abstract syntax definition for a statechart environment may be defined as shown in Figure 1.

Note, from such a meta model / class diagram, Fujaba generates Java classes for the different kinds of objects, their attributes and their relationships. Relationships are realized using pairs of forward and backward pointers. For to-many relationships, Fujaba employs different kinds of pre-defined container classes.

From the developers point of view, Fujaba's implementation of relationships turns Java object structures into graphs with bi-directional edges. Provided with a meta model / class diagram / graph schema, our dynamic object

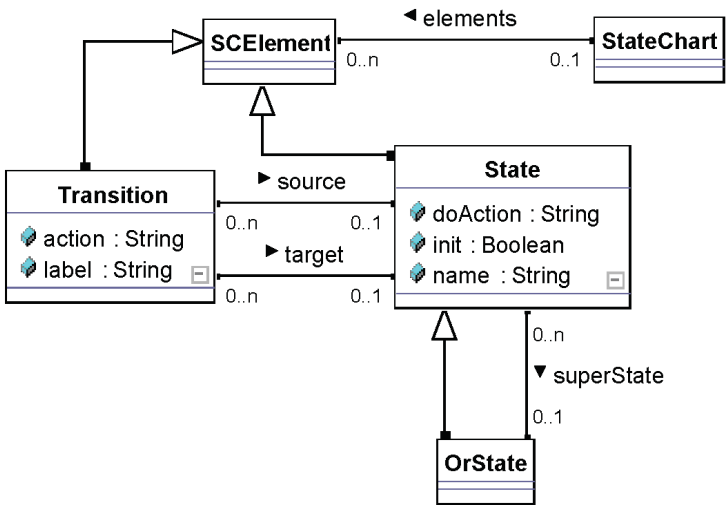

Fig. 1. Class diagram in Fujaba

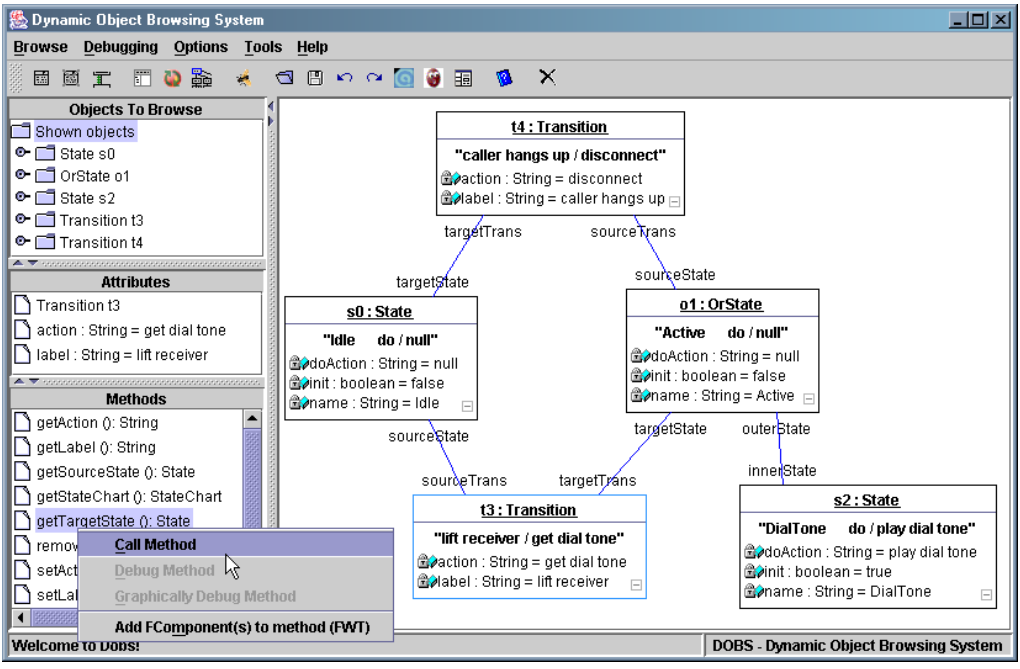

Fig. 2. Abstract syntax in DOBS

browser DOBS may already be used as a simple editor for models / object digrams / graphs, cf. Figure 2. However, DOBS shows the abstract syntax of our model, only. To enable editing in statechart notation, concrete syntax i.e. a graphical user interface still needs to be provided. This is discussed in

chapter 7.

Note, for technical reasons, the Fujaba approach has no explicit notion of a host graph. This means, there is no pre-defined mechanism enumerating all elements of the current graph, if required. Instead, the model itself has to provide some object that may be used to reach all elements of the corresponding graph. Therefore our meta model provides an explicit *StateChart* class and each statechart object / node collects all elements, i.e. states and transitions, of the corresponding statecharts.

4 Model transformations

Based on our meta model, we now discuss model transformations in the sense of model driven architecture. As an example for a simple model transformation we specify the flattening of complex statecharts to plain state machines. We discuss this flattening first, since this allows us to simplify the specification of operation semantics and of consistency checks, later on.

Flattening of statecharts with or-states deals with the replacement of transitions targeting or-states and with the replacement of transitions leaving or-states and with the removal of or-states that have no more transition attached.

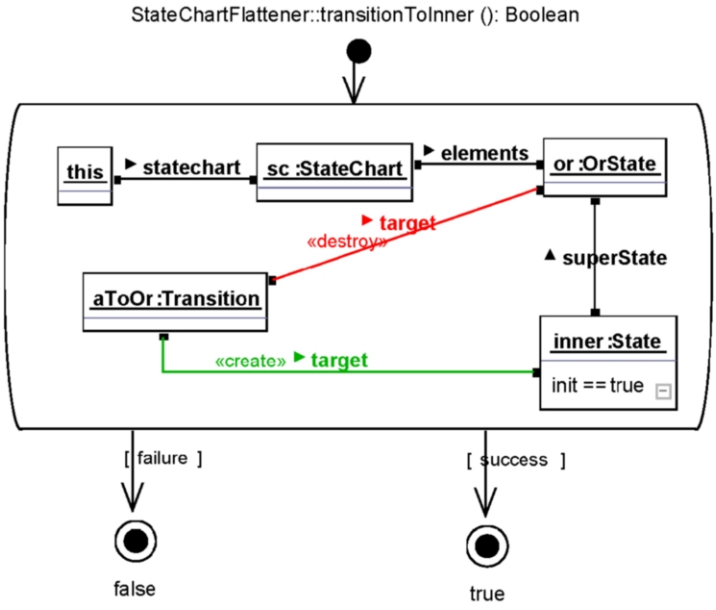

Fig. 3. Replacing transitions targeting or-states

Figure 3 specifies the replacement of transitions targeting or-states. Such transitions are simply re-targeted to the initial state of the statechart embed-

ded within the or-state. Note, Figure 3 employs a new (functional) class *StateChartFlattener* that has a *statechart* reference to *StateChart* objects. The graph transformation in Figure 3 matches a statechart object *sc* containing an or-state *or* that is targeted by a transition *aToOr*. In addition, the graph transformation identifies a sub-state object *inner*, where the *init* attribute has value *true*, i.e. the initial sub-state. As indicated by the ≪*destroy*≫ and ≪*create*≫ markers, the graph transformation of Figure 3 removes the *target* link connecting transition *aToOR* and or-state *or* and adds a new target link leading to sub-state *inner*. If this rule is applied as often as possible, all transitions leading to or-states are redirected to the corresponding initial states.

Note, in Figure 1 class OrState inherits from class State. This means, any time we need a node of type State, a node of type OrState does the job as well (substitutability). For our graph rewrite rule this means, node inner may either match a plain state or an or-state. Thus, our graph transformation works for nested or-states as well.

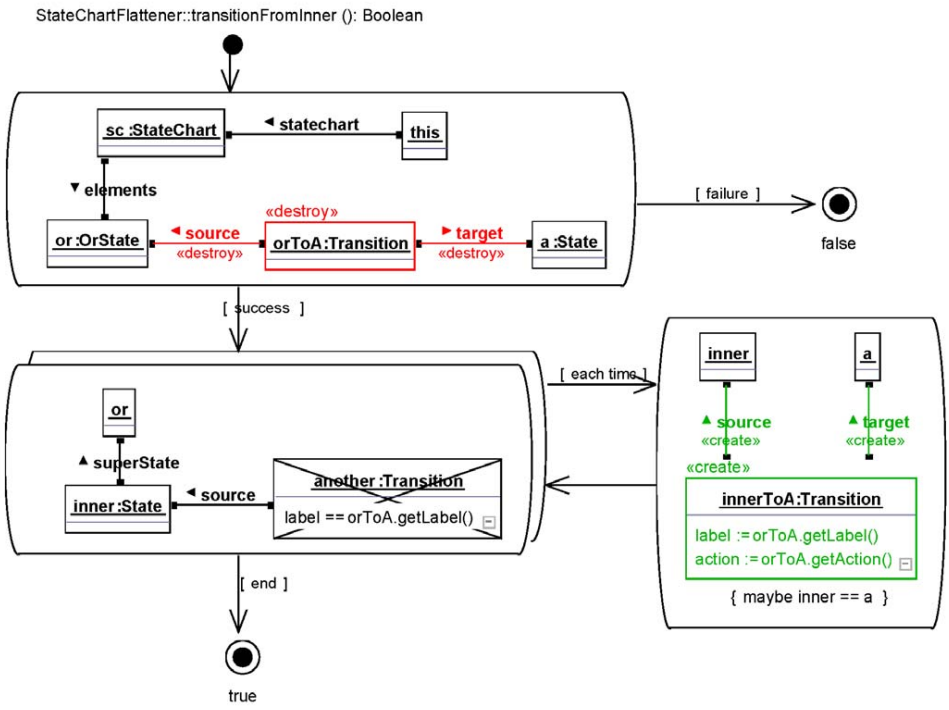

Fig. 4. Replacing transitions leaving or-states

The (programmed) graph transformation of Figure 4 replaces transitions leaving or-states. This is done in three steps. The first graph rewrite rule

identifies a transition *orToA* with a source link to an or-state *or* and destroys it. If this rule has been applied, successfully, the second graph rewrite rule identifies inner states of *or* that do not already have a leaving transition with the same label. Note, Fujaba uses crossed out elements to specify negative application conditions. In this rewrite rule, the object *or* is shown without its type. In Fujaba, omitting the type indicates so-called bound objects. Bound objects are objects that have already been matched to the host graph in a previous step. Thus, a bound object does not compute a new match but it reuses its old match. The second graph rewrite rule also has two stacked shapes. Such a rule is called a *for-each activity*. For-each activities are iteratively applied as long as new matches are found. Due to the *each time* transition in Figure 4, each time when the second graph rewrite rule identifies an inner state without an appropriate leaving transition, the third graph rewrite rule is executed. The negative node *another* prevents the creation of a new transition if the *inner* state has already such a transition. This implements the priority rules of UML statecharts. The third graph rewrite rule creates a new transition leaving the corresponding *inner* state, targeting the same state *a* as the old transition. In addition, the transition label and the transition action are transferred.

Note, in general Fujaba employs isomorphic rule matching, only. However, the *maybe inner==a* clause of the third graph rewrite rule allows nodes *inner* and *a* to be matched on the same host graph object. This handles self transitions.

The graph rewrite rule of Figure 5 employs two negative nodes ensuring that the considered or-state has no out-going and no incoming transition, any more. For simplicity reasons, a third negative application condition ensures that the considered or-state is not embedded in another or-state. This means, we handle nested or-states outside in. If all conditions hold, the or-state is destroyed and all its sub-states are added to the statechart *sc*. In addition, the init flag of the or-state is transferred to its initial sub-state. Thus, if the or-state was a usual state, its initial sub-state becomes a usual state, too. If the or-state was the initial state of the whole statechart, its initial sub-state becomes the new initial state of the statechart.

In Fujaba the graph grammar like application of a set of rules as long as possible needs to be programmed, explicitly. This may be done as shown in the (pseudo) graph transformation of Figure 6. Figure 6 employs a boolean constraint calling our three model transformations. If one of the above transformation is applied (and returns true), we follow the *success* transition, and the boolean constraint is evaluated, again. If no transformation succeeds, the transformation terminates. Thus, the application of transformation *flatten-*

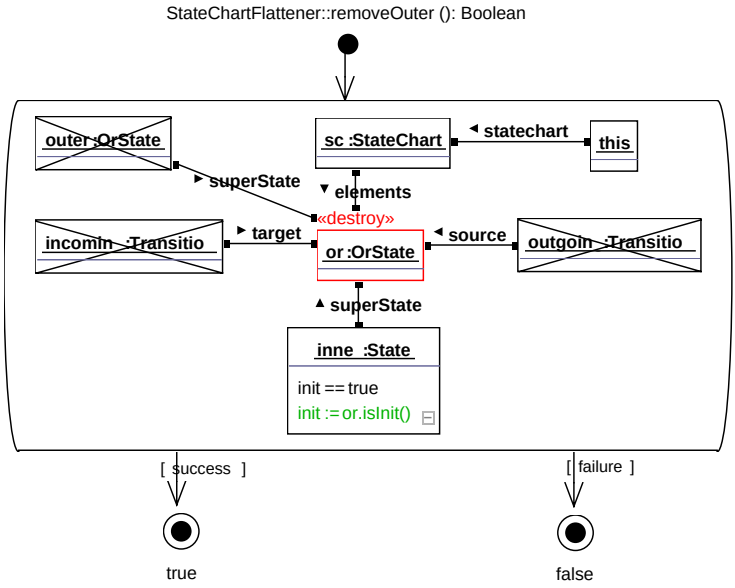

Fig. 5. Removing obsolete or-states

StateChart removes all (even nested) or-states and results in a simple state machine.

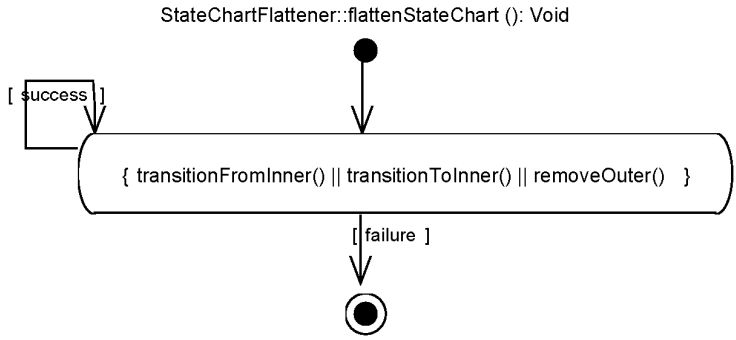

Fig. 6. Employing the transformation rules as long as possible

Note, the boolean *or* operators connecting our three basic model transformations use left precedence and short circuit evaluation. This means, transitionFromInner has higher priority than transitionToInner which again has higher priority than removeOuter. Thus, the proposed way of applying a set of graph transformation implies precedences on the transformation rules. Relying on these precedences, we e.g. might have omitted the negative nodes of Figure 5.

5 Operational semantics

This chapter provides the operational semantics for our statecharts. Of course we could interpret statecharts with (nested) or-states directly. However, this would need some more complicated rules. Thus, to facilitate the example this chapter assumes that the state chart is first flattened and all or-states are properly replaced. Then, the statechart may be executed using the graph transformation of Figure 7.

For handling events, we employ an object of type *FSMSimulator*. This simulator object has a *current* edge marking the currently active state. If method *handleEvent* is called, it tries to identify an outgoing transition *a* with the label provided in parameter *event*. The *maybe current==next* clause allows to handle self transitions. If such a transition exists, the *current* edge is redirected to the target state of the transition. In addition, the transition action and the do-action of the target state are executed. For simplicity reasons, here this is simulated using *System.out.println*. Alternatively, the actions might e.g. employ Java syntax and we could use a Java interpreter like the bean shell [1] to actually execute the actions.

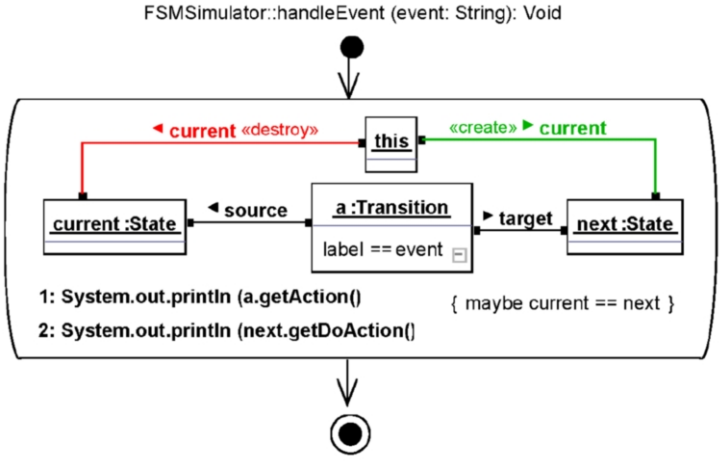

Fig. 7. Firing transitions

6 Consistency checking

Consistency checking is an important functionality of a modeling environment. Usually, there is quite a number of trivial yet useful checks, e.g. transitions need to have source and target, states must have unique names within their scopes, etc. Here, we focus on a little more challenging consistency check: each state should be reachable from the initial state. Again, for simplicity reasons,

we first flatten the or-states in order to deal with simple state machines, only. Our reachability test is done in three steps, cf. Figure 8.

In the first step we just mark all states using an *unreachable* link. Thus, the multi-object *old* collects all state elements of statechart *sc*. In the second step we employ a so-called path expression *(sourceTrans.targetState)**. A simple path expression is just a dotted list of edge labels. It is evaluated by traversing the corresponding links. In our example, edges with label *sourceTrans* lead from states to outgoing transitions. (In Fujaba, every edge has two labels, one for its forward direction and one for the reverse direction.) Accordingly, *targetState* edges lead from transitions to their target states. Thus, the path *sourceTrans.targetState* leads from a given state to all its successor states. In our example, we use the * operator to compute the transitive closure of this basic path expression. In the second graph rewrite rule of Figure 8 the path expression is applied to the initial state of our state machine. Due to the transitive closure, the path expression computes the set of all successor states reached by traversing transitions zero, one, or multiple times. This set is collected in the multi-object *reachables*. Note, the *maybe* clause allows the initial state to be contained in the set of reachable states. Note, in addition to dotted lists of edge labels and to the transitive closure operator, Fujaba path expressions provide an *or* operator computing the union of two path expressions. More complex operators are not yet implemented.

Once the set of reachable states is computed, the third graph rewrite rule of Figure 8 removes the unreachable markers from these nodes. Note, in the third activity the multi object is again a bound object. Thus, the set of objects found in the step before, is reused in this step.

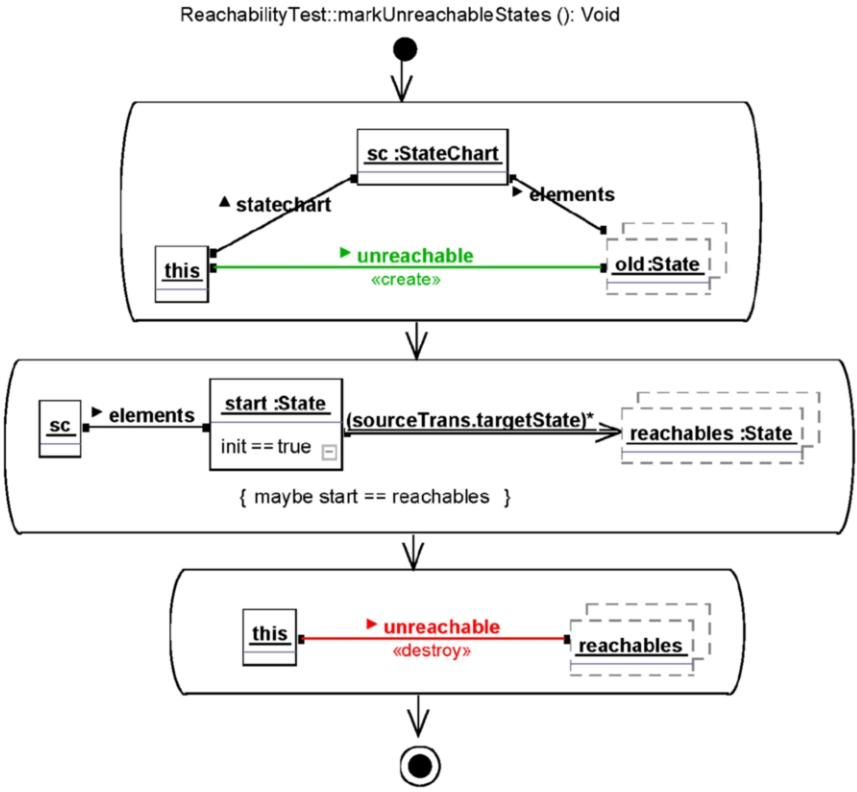

Fig. 8. Reachability test for state machines

7 Concrete syntax

So far we have dealt with the example statechart environment on the level of the abstract syntax only. Once we have modeled our meta model, the model transformations, the operational semantics and the consistency checks, we *just* need a graphical user interface for our new tool.

In principle, since the Fujaba code generator turns our graph schema and our graph transformations in usual Java code, one could use any modern GUI toolkit, e.g. Swing, and any modern GUI builder and just build the GUI in the conventional way. However, for diagrams editors this is very painful. Alternatively, we could use special frameworks for graphical editors, e.g. DiaGen, cf. [3].

However, in this example we used a simple GUI toolkit named Fujaba Window Toolkit (FWT) that we have developed for teaching purposes. FWT provides a set of adapter classes for swing elements. These adapter classes enable the interactive construction of simple GUIs within our Dobs environment,

cf. Figure 9. Figure 9 shows an *FVerticalContainer* object *f5* which contains links to two *FHorizontalContainer* objects *f4* and *f7*. The upper horizontal container contains the checkbox *f3* and the *FTextField f1*. The lower horizontal container contains a *do/* label and another text field *f2*. The upper part of object *f5* shows the swing representation generated by this FWT structure.

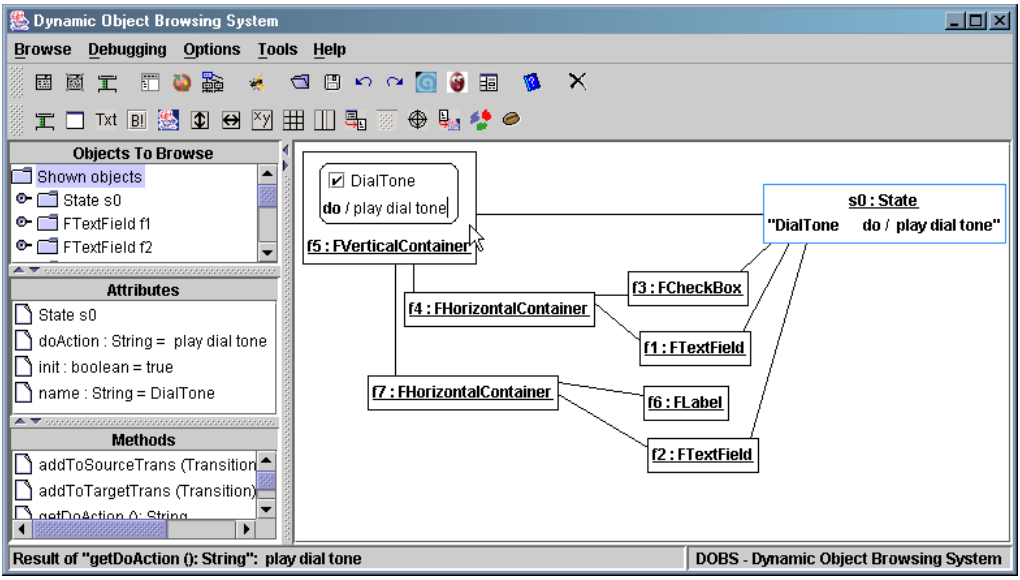

Fig. 9. GUI construction in Dobs

Each *FTextField* object has a *subject* link identifying the corresponding logical object. In addition *FTextField* objects have a special attribute *attrName* containing the name of an attribute of the subject object. With this information, the *FTextField* object is able to depict the value of a certain attribute of its subject object. Similarly, the user may click on the swing representation of an *FTextField* and edit the depicted value. Then, the *FTextField* updates the value of the corresponding attribute of the subject object, automatically. Thus, in our example, the user may edit the name *or* the *doAction* of state *s0* directly via the swing representation contained in *f5*.

If the subject of an FWT container is changed, this change is forwarded to all interested sub-objects. Thus, if we connect *f5* to another state, the swing representation would be connected to the *name* and *doAction* of that new state.

For the representation of a whole statechart, we employ an *FXYContainer* that allows to move its sub-objects. Then we attach the FWT structure for a single state as a prototype to this *FXYContainer*. In addition, we tell the *FXYContainer* to observe the *elements* association of its subject statechart.

Thus, every time we add a new state to the *elements* association of that statechart, the *FXYContainer* adds a copy of the prototype FWT structure for this kind of logical object to its content. In addition, *subject* links between this new FWT structure and the logical object are established. A similar mechanism may be used for transitions. Finally, the *FXYContainer* representing the statechart is embedded in an *FWTFrame* object which creates a new *JFrame* window containing the swing representations of the statechart elements. In this way, we have created the GUI shown in Figure 10.

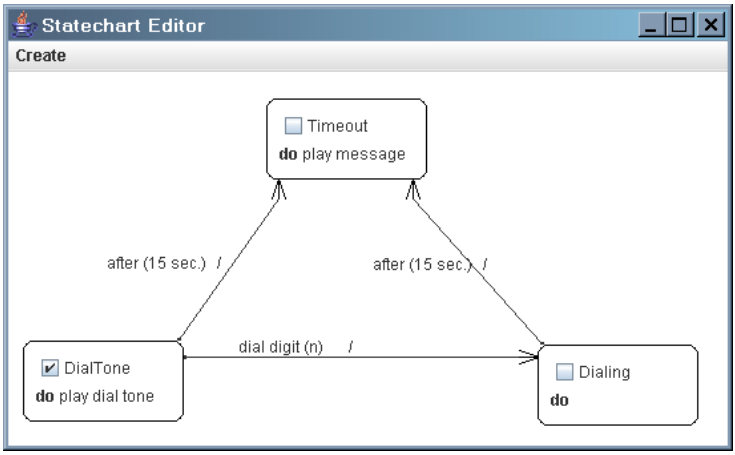

Fig. 10. Simple FWT GUI for a statechart tool

8 Persistency, undo/redo and multi user support

To complete the functionality of our statechart environment, we need to be able to store and retrieve statechart projects. This might easily be achieved using Java serialization mechanisms. However, Fujaba provides a special framework called Common Object Replication Architecture (CoObRA). With a single command, the Fujaba code generation is changed to employ the CoObRA mechanism. Thereby, all changes to the object graph are automatically protocolled. This protocol is then used for storage and retrieval, for undo/redo, for versioning and for merging contributions from multiple authors. The latter mechanism provides multi user support based on optimistic locking concepts. See [8] for more details.

9 Summary

This paper outlined tool modeling with Fujaba with the help of a little case study about a simple statechart environment. We addressed the typical re-

quirements of such an environment, i.e. abstract syntax, model transformations, operational semantics, consistency checking, concrete syntax, and persistency concepts.

These example transformations utilize many of the more sophisticated features of Fujaba e.g. programmed graph rewriting, method invocations, foreach activities, multi objects, maybe clauses, negative application conditions, path expressions, etc. Similar language elements are provided by Progres graph transformations [7], only. Due to our experiences, such sophisticated modeling constructs are mandatory for the specification of complex functionality as required for CASE tools.

In addition to the application logic, a practical tool requires a lot of additional functionality, e.g. a graphical user interface, persistency, undo/redo and XMI based model exchange mechanisms, code generation, etc. In order to facilitate the realization of such functionality and to allow the seamless integration with other Java libraries, Fujaba generates usual Java code from the graph schema and from all graph transformations. This generated code is easily blended with conventionally programmed system parts.

References

[1] http://www.beanshell.org/

[2] I. Diethelm, L. Geiger, A. Zündorf: Systematic Story Driven Modeling, Workshop on Scenarios and State Machines: models, algorithms, and tools; workshop at ICSE 2004, Edinburgh, 2004.

[3] http://www2-data.informatik.unibw-muenchen.de/DiaGen

[4] L. Geiger, A. Zndorf: Statechart Modeling with Fujaba; 2nd International Workshop on Graph-Based Tools (GraBaTs); ICGT 2004, Rom, Italy, 2004.

[5] T. Fischer, J. Niere, L. Torunski: Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Stroy-Driven-Modeling (in german), Diploma Thesis, University of Paderborn, 1998.

[6] Fujaba Homepage, University of Paderborn, http://www.fujaba.de/.

[7] http://www-i3.informatik.rwth-aachen.de/research/projects/progres/

[8] C. Schneider, A. Zündorf, J. Niere: CoObRA - a small step for development tools to collaborative environments; Workshop on Directions in Software Engineering Environments; workshop at ICSE 2004, Scotland, UK 2004

[9] A. Zündorf: Rigorous Object Oriented Software Development, Habilitation Thesis, University of Paderborn, 2001.