Using UML as Visual Programming Language

Hans-Josef Köhler, Ulrich Nickel, Jörg Niere, and Albert Zündorf

Department of Computer Science, University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany;
e-mail: [hjk|duke|nierej|zuendorf]@uni-paderborn.de

Abstract: This paper proposes to use UML class diagrams and UML behavior diagrams like collaboration diagrams, activity diagrams, message sequence charts, and state-charts as a visual programming language. We describe a code generator that generates a (Java) implementation of an application from its UML specification. Thereby, we define a formal semantics for these UML diagrams and we define how different (kinds of) diagrams are combined to a complete executable specification.

Generally, generating code from UML behavior diagrams is not well understood. Frequently, the semantics of a UML behavior diagram depends on the topic and the aspect that is modeled and on the designer that created it. In addition, UML behavior diagrams usually model only example scenarios and do not describe all possible cases and possible exceptions.

We overcome these problems by restricting the UML notation to a subset of the language that has a precise semantics. In addition, we define which kind of diagram should be used for which purpose and how the different kinds of diagrams are integrated to a consistent overall view.

1 Introduction

UML focuses on early phases of the sofware life-cycle like object-oriented analysis and object-oriented design, cf. [BRJ99]. Thus, UML behavior diagrams usually model typical scenarios describing the desired functionality, only. Our work focuses on the design and implementation phase. We are looking for executable specifications. In [JZ98] we first proposed to combine UML class-diagrams, UML activity-diagrams, and UML collaboration-diagrams to so-called *story-diagrams*. Story-diagrams do not just show scenarios but specify the over-all behavior. Story-diagrams have a precise operational semantics that allow their automatic translation to an object-oriented programming language like Java or C++. [ZSW98] proposes story driven modeling as a systematic approach to use UML- and story-diagrams for the development of an object-oriented application. Based on this work, in [FNTZ98] we first described the Fujaba environment that allows the editing of story-diagrams and provides a code generator translating story-diagrams to Java code that implements the specified structure and behavior of an application.

So far, story-diagrams are especially suited for the specification of complex application specific object-structures and their evolution over time. However, story-diagrams lack means for the specification of reactive objects and concurrent executions. Concurrent execution and reactive behavior is a typical application domain for state-charts. State-charts, however, abstract from the concrete states of coordinated objects, by definition. State-charts do not deal with application specific object-structures, appropriately. Thus, this paper proposes to combine state-charts and story diagrams to form an executable specification language that allows to specify reactive behavior as well as complex application specific object-structures.

Chapter 2 introduces the simulation of a simple production process as a running example for this paper. Chapter 3 discusses the translation of class-diagrams to Java code as a basis for the translation of behavior diagrams. Chapter 4 desribes a table-driven approach to the implementation of state-charts that enables a straight-forward translation of state-charts to Java code. This enables us to combine state-charts and story-diagrams in chapter 5 yielding a complete yet executable specification language and a code generator for its translation to Java. Chapter 6 summarize our work and out-lines current and future work.

2 Running Example

This paper uses the simulation of a simple production process as running example. This production process models a factory with various manufacturing places and with shuttles transporting goods from one manufacturing place to another. The example stems from the ISILEIT project funded by the German National Science Foundation (DFG). The goal of the project is the development of a formal and analysable specification language for manufacturing processes. In addition, a code generator shall provide automatic code generation for driving the constituent parts of a manufacturing process, namly shuttles, robots, assembly lines, etc. Note, In a modular factory all these constituents have local control. We plan to model the manufacturing process up-front and to simulate its functionality in order to validate if everything works correctly and than to generate the software that runs the constituents of the manufacturing process. Overall, more flexible processes adjusting to market demands more quickly will be achieved.

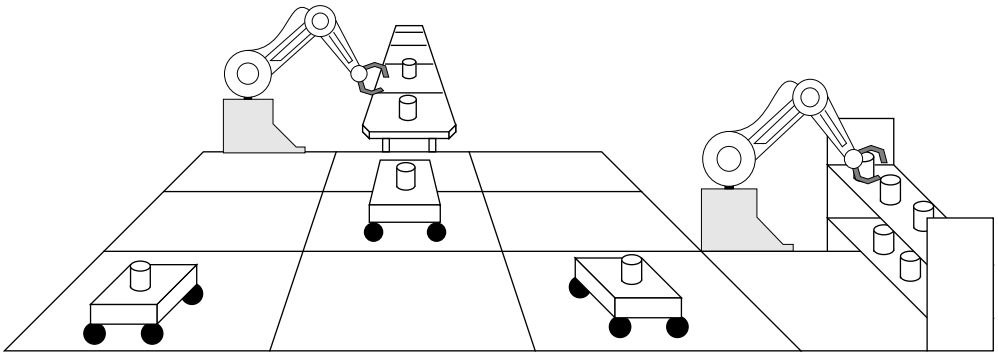

Figure 1 Simple factory example

Figure 1 shows a scenario of a sample factory. The factory is modeled as a flat building without levels and pillars in it. The floor is layered with rectangle shaped fields allowing to address certain positions in the building and serving as a matrix. The factory contains certain kinds of production places. A production place is e.g. an assembly line, where goods arrive and are loaded on shuttles, or a storage, where goods can be stored. In Figure 1 there are three shuttles moving across the floor transporting goods from the assembly line at the top of the figure to the storage at its bottom right side, autonomously.

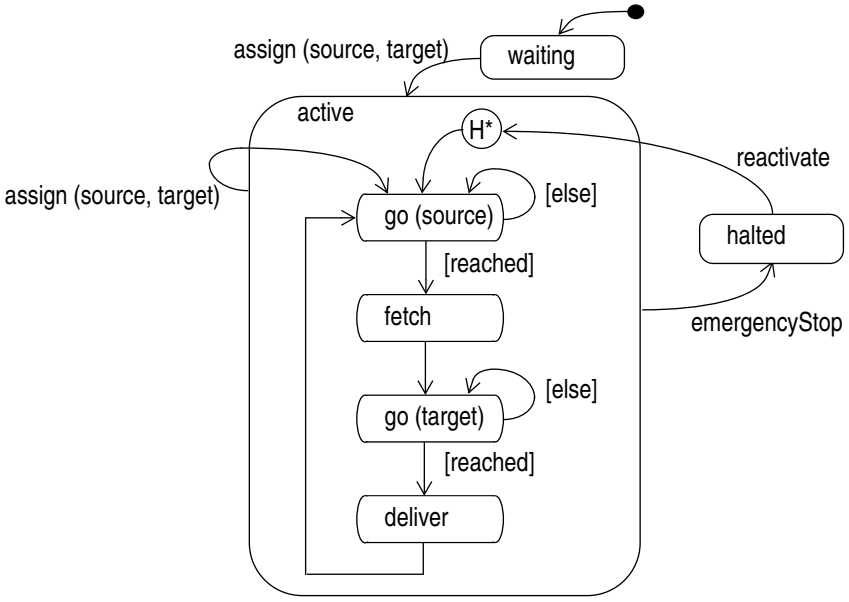

Figure 2 State-chart of a shuttle

The basic behavior of a shuttle is specified in the state-chart shown in Figure 2. Initially, a shuttle is just waiting. Then the shuttle is instructed to transport goods from a source place to a target place by sending it an assign event. The shuttle reacts on such an event by switching from the waiting state to the active state and by storing its source and target place, internally. The active state is a complex state with state go(source)

as its initial sub-state. In state go(source) the shuttle tries to go to the source place of its current order. It finds his way autonomously using its own routing algorithm and avoiding colissions with other shuttles. Once the shuttle reached its source place it fetches a good and switches to the go(target) state. When the shuttle reaches its target place it delivers the transported good, turns back to the go(source) state, fetches another good, goes to its target, and so on. The shuttle will worn-out this procedure until it "dies".

But our shuttles are reactive systems. Thus, a shuttle may be reassigned with a different instruction at any time by just sending it another assign event. In case of an emergency stop, a shuttle is halted by an emergencyStop event. In state halted the shuttle may be reactivated by sending it a reactivate event. When a shuttle is reactivated, it switches into the history state of the active state. The history state stores the last sub-state of the shuttle before it switches into the halted state and recalls that state when the shuttle is reactivated.

Actually, our example specification is much more complex. We have downstripped the example considerably to facilitate its understanding.

3 From Class Diagrams to Code

In this section we introduce the FUJABA code generator which generates Java code for classes, attributes, method declarations, and associations. Figure 3 shows a screen-shot of the FUJABA environment [FNTZ98] with the UML class-diagram for the production process simulator example. The class-diagram is a straight forward design of the factory example. So there exist classes like Shuttle, Storage, AssemblyLine, or Field. The classes Storage and AssemblyLine inherit from class Place.

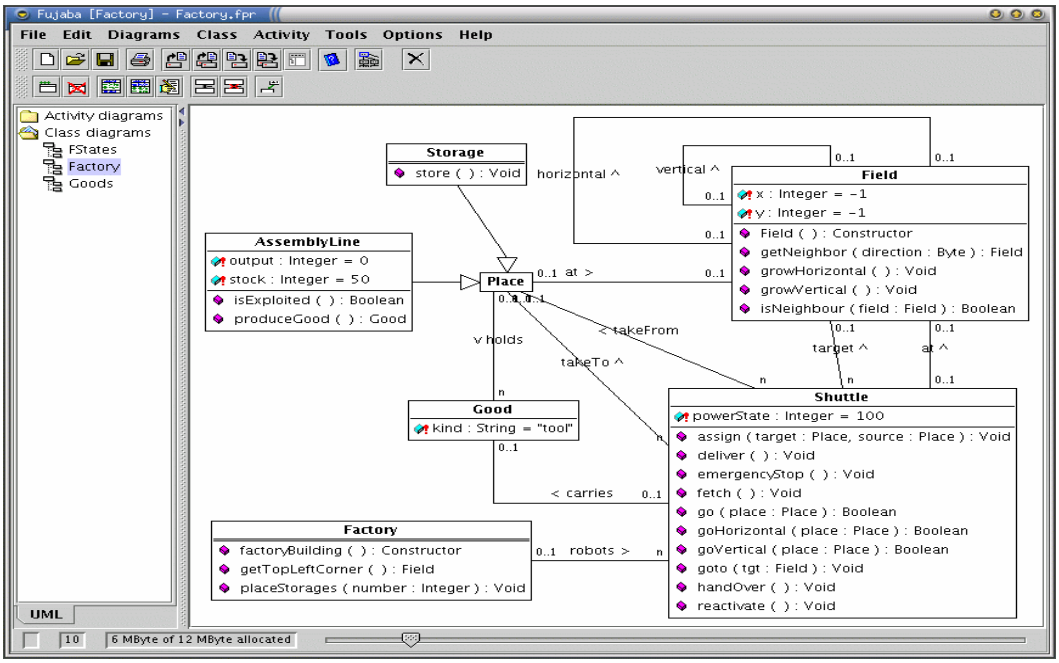

Figure 3 The Fujaba environment showing the example class-diagram

The translation of class diagrams to an object-oriented programming language is straight-forward and provided by most current OO-CASE tools. Figure 4 shows a cut-out of the Java code that the Fujaba environment generated for class Shuttle and class Field of our example. The Fujaba generator translates UML classes to Java classes, cf. line 3. Methods are translated into Java method declarations, cf. the go- and assign-methods line 15 and 16. The method bodies are normally empty (but a designer may specify a body using UML behavior diagrams see section 4 and 5). According to the Java Beans style guides, we translate attributes to private Java attributes accessible via appropriate get- and set-methods, cf. line 5 to 10.

UML associations are usually bi-directional. Thus, we implement associations by pairs of references in the respective classes. For multi-valued associations like the (reverse direction of the) target association we use standard container classes provided by the java generic library [JGL99]. In order to guaranty the consistency of the pairs of references that implement an association, the respective access methods for reference attributes call each other. For example, the addToShuttles method in line 24 first checks if the shuttle still exists

```
 1: import java.util.*
 2: import com.objectspace.jgl.*;
 3: public class Shuttle{
 4:     private int powerState = 100;
 5:     private int getPowerState ( ) {
 6:         return powerState;
 7:     } // getPowerState
 8:     private int setPowerState (int newState) {
 9:         return  powerState = newstate;
10:     } // setPowerState
11:     private Field target;
12:     public getTarget () { ... }
13:     public setTarget () { ... }
14:     ...
15:     public void go () { ... }
16:     public void assign (Place source,
17:                                 Place target) { ... }
18: ... } // Shuttle
```

```
19: import java.util.*;
20: import com.objectspace.jgl.*;
21: public class Field {
22:     private OrderedSet shuttles
23:                         = new OrderedSet ();
24:     public void addToShuttles (Shuttle shuttle) {
25:         if (!this.hasInShuttles (shuttle)){
26:             this.shuttles.add (shuttle);
27:             elem.setTarget (this);
28:     }  } // addToShuttles
29:     public void removeFromShuttles (...) { ... }
30:     public boolean hasInShuttles (Shuttle shuttle) {
31:         return (this.shuttles.get(shuttle) != null));
32:     } // hasInShuttles
33:     public Enumeration elementsOfShuttles ( )
34:     {  return this.levels.elements (); }
35:     ...
36: } // Field
```

Figure 4 Java code for class Shuttle and Field

in the container by calling the hasInShuttles method (cf. line 25). If the shuttle is unknown than it is added to the container (line 26). Next, the method setTarget is called on the added shuttle, passing the field object itself as parameter. This establishes the reverse reference. This implementation of associations is close to the strategy of [Rhap].

The proposed generation of code for UML class diagrams provides the basis for the translation of behavior diagrams.

4 Reactive Behavior via State-charts

State-charts can be used for many different purposes. They can model the behavior of a whole application or just of a single method. State-charts may model all possible sequences of method invocations on a certain object or they may model state dependent reaction of certain objects on the receiption on certain events. Each of these uses leads to a totally different semantics of the given state-chart. In addition, the different usages employ different state-chart language features. In order to generate code from a state-chart one must clearly identify which part of an application and which behavioral aspect of this part is modeled by the provided state-chart.

In this paper we assume that state-charts are used to model the reaction of objects on events send to them. Thus, state-charts are attached to (reactive) classes. We turn events into so-called *event methods* of the corresponding classes in order to provide an uniform and convinient way of invoking some service on an object. The event methods build the public interface of the modeled class. Entry, exit, and do actions and actions attached to transitions are implemented as so-called *action methods* of the modeled class. These methods get private visibility since they are intended to be called (directly or indirectly) from the event methods, only. Thus, a state-chart defines which action methods will be called as reaction on a(n event) method invocation and depending on the object's current state.[1]

Our approach is inspired by the state-design pattern, cf. [GHJV95], and by [AT98] and by the Rhapsody case tool [HG96, Rhap]. However these approaches generate specific new classes for each state employed in the state-chart. These specific classes possibly contain a huge amount of redundant and duplicated code. [Doug98, chapter 6.2.3] uses a generic array based state-table. However, [Doug98] has some problems dealing with complex nested states. (We will discuss this point at the end of this chapter.) Our approach adapts the idea of [Doug98] but uses an object-oriented implemention of the state-table. Figure 5 shows the design of our state-table classes. Any reactive object like class Shuttle of our running example becomes a

1. Action methods may call event methods on other reactive objects, thereby sending them events.

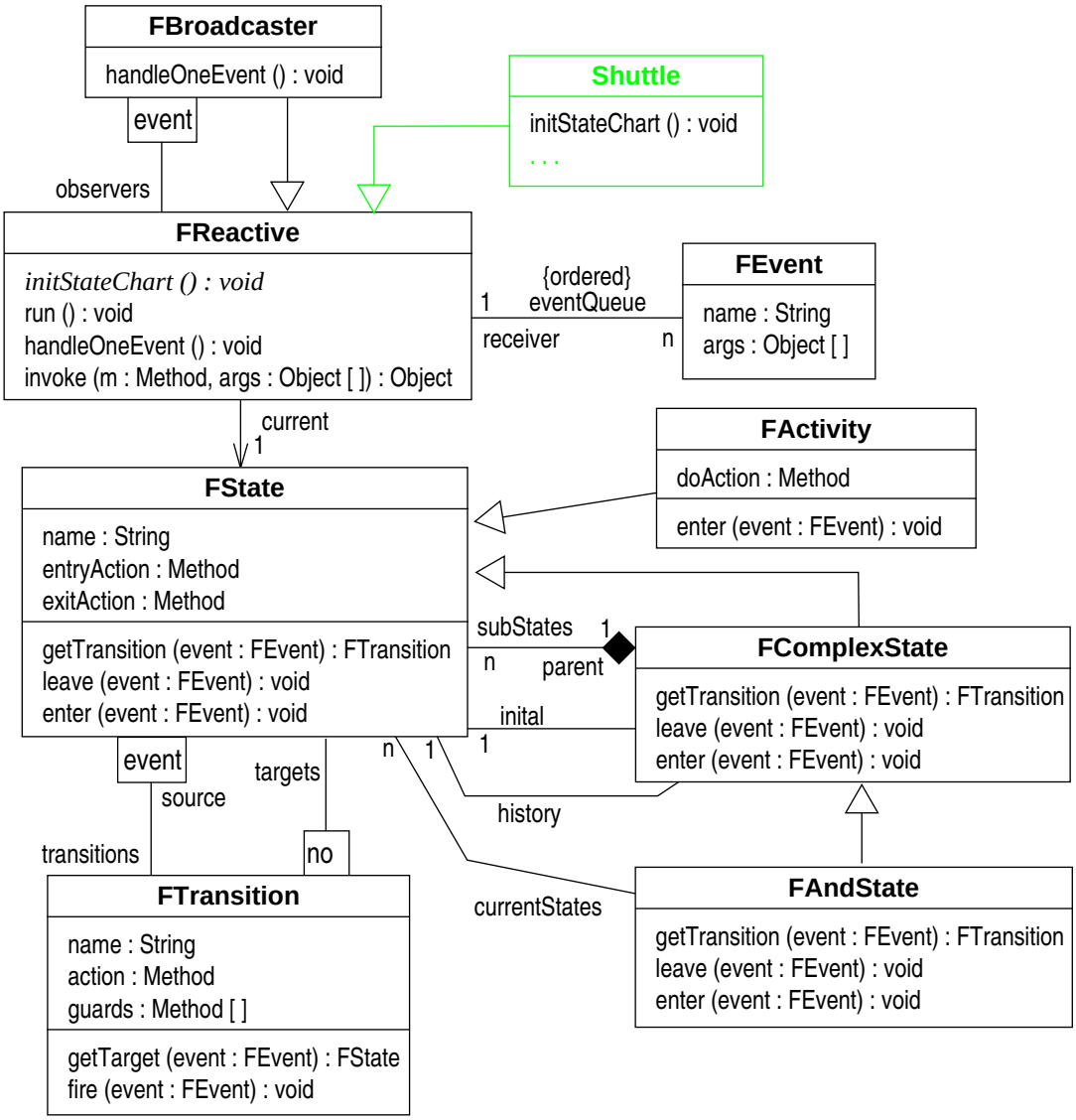

Figure 5 Generic State-Chart Classes

subclass of class FReactive.[2] Class FReactive provides a pointer to the current state of the reactive object and an abstract initStateChart method and a handleOneEvent method and an invoke method. The initStateChart method is used to create the object's state-table. Figure 6 shows a cut-out of the runtime object structure created by method initStateChart (as redefined in class Shuttle). The runtime object-structure of Figure 6 represents (a cut-out of) the statechart shown in Figure 2. There is a master state that contains the first level nested states waiting, active, and halted via subStates links. In turn the complex state active contains the states go_source, fetch, etc. The transition objects show their firing event name. Each state has a (hash-)table of out-going transitions qualified by event names, e.g. from state waiting one reaches transition t1 via key "assign". Each transition has an array of target states. Normally, this array contains only a single entry, e.g. transition t1 leads to state active. However, a state may have multiple outgoing transitions labeled with the same event but distinct by mutal exclusive guard conditions, cf. state go_source of Figure 2. Such transitions are represented by a single transition object with multiple targets, see transition t2. Note, transitions connected to complex states are represented by a single FTransition object that is connected to the corresponding FComplexState objects, e.g. transition t1 and t5.

As already discussed, sending an event to an reactive object is done by just calling the appropriate event method. Each reactive object has its own event queue inherited from class FReactive, cf. Figure 5. Event methods just have to encapsulate the signaled event into an FEvent object and push this event object into the event queue of the reactive object. For example, Figure 7 shows the implementation of method assign of class Shuttle. It just creates an FEvent object passing the event name as first parameter to the constructor call (line 3) and an object array containing the parameters source and target as second parameter (line 4).

2. If this creates a multiple inheritance situation, we provide an interface version of class FReactive, too.

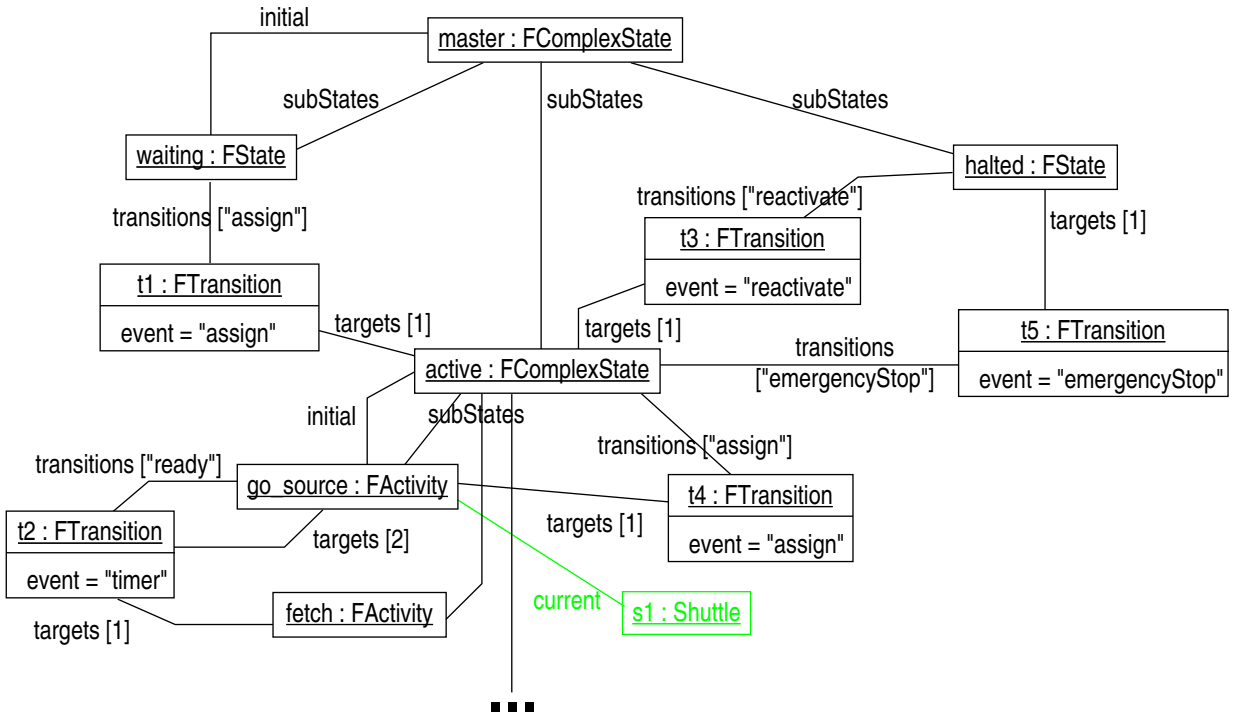

Figure 6 State-table for the state-chart of Figure 2 (cut-out)

```
1: public class Shuttle extends FReactive { ...
2:     public void assign (Place source, Place target) {
3:         FEvent event = new FEvent ("assign",
4:                                    new Object [ ] {source, target})
5:         this.addToEventQueue (event);
6:     } // assign
7: ... } // Shuttle
```

Figure 7 Method assign of class Shuttle

Finally, the event method calls method addToEventQueue to append the new event object to the reactive object's event queue.

Note, in our example application (area) all events are explicity targeted to their receivers. (One actually calls a method on the receiving object.) However, usually state-charts assume a broad-cast mechanism distributing events to all available reactive objects. To support this style of event handling, our framework provides the class FBroadCaster, cf. Figure 6. In case of broadcast events, the application creates an implicit broadcaster object. All reactive objects (interested in this kinds events) register themselfes at the broadcaster for the events they are interested in. Broadcast events are created as usual and then pushed into the event queue of the broadcaster. The broadcaster handles the received events by forwarding them to the event queues of the reactive objects that registered their interest in this kind of events. Note, an application may employ its own or additional broadcasters, e.g. in order to establish broadcasting of certain events for certain groups of reactive objects. Such group events are just send to the group broadcaster object.

Our framework provides two different ways to consume events from the event queue of a reactive object. First, the object may run in its own thread. Therefore, method run of class FReactive contains an event loop listening to the event queue. If an event is available, method run invokes method handleOneEvent to consume it. Method run waits for the return of handleOneEvent and then it seeks for the next event. Second, all reactive objects may be controlled by a simulation environment. In this mode, the simulation environment loops through the set of reactive objects and calls method handleOneEvent on them. The latter mode allows a stepwise synchronous execution of the event handling for debugging purposes. Note, the two different execution modes might consume events in different orders. However, in a well designed system all possible event orders should yield the same overall result. Of course this is not easy to ensure. Our simulation mode just allows to execute events one by one, in order to test their effects, first. Once single events work fine, one can start looking for the concurrence problems.

Figure 8 shows method handleOneEvent. Method handleOneEvent consumes events based on the state-table approach. It first retrieves the event to be consumed from the event queue, cf. line 3. In case on an empty event queue line 4 aborts the execution. In line 5, handleOneEvent retrieves the current state object.

```
 1: public class FReactive implements Runnable { ...
 2:    synchronized public void handleOneEvent () {
 3:       FEvent event =  this.popFromEventQueue ();
 4:       if (event == null) { return; }          // no event ==> abort
 5:       FState current = getCurrent ();
 6:       FTransition transition = current.getTransition (event);
 7:       if (transition == null) { return; }     // no transition ==> abort
 8:       FState newState = transiton.getTarget (event);
 9:       if (newState == null) { return; }      // no target ==> abort
10:       FState parentState = current;
11:       do {
12:          current = parentState;
13:          current.leave (event);
14:          parentState = current.getParent ();
15:       } while (current != transition.getSource ());
16:       transition.fire (event);
17:       newState.enter (event);
18: } // FReactive
```

Figure 8 Method handleOneEvent of class FReactive

(In the example of Figure 6 shuttle s1 would retrieve go(source) as its current state.) Next, the current state object is queried for an outgoing transition corresponding to the current event. Method getTransition of class FState just looks up the corresponding transitions hash-table. In case of a ready event[3] state go(source) would thus retrieve transition t2. If the transitions table contains no entry for the current event there might still be an appropriate transition leaving a surrounding complex state. Therefore, the getTransition query visits its ancestor states recursively (using method getParent ()). Thus, in case of an assign or emergencyStop event state go(source) would retrieve transition t4 or t5, respectively. If no appropriate event is found, handleOneEvent terminates, cf. line 7. [4]

Line 8 asks the found transition for its target state. Normally, this causes just a table lookup. However, a guarded transition has to check the different guards to determine a fulfilled one. Recall, the guard evaluation operations are attached to the application specific classes, here Shuttle, since they might access private attributes and operations of the application classes. Thus, class FTransition has an array of Method objects. During construction of the state-table via method initStateChart, we obtain pointers to the guard evaluation methods, using package java.lang.reflect, Java's application programmer's interface to its runtime type information, cf. [Flan97]. Such a method pointer may be executed via method invoke provided by class FReactive. (Method invoke again relies on generic mechanisms provided by java.lang.reflect.) Method getTarget gets the current event as its parameter in order to provide access to the event's argument array. In addition, the event knows its receiver since the eventQueue association is bi-directional, cf. Figure 5. This enables getTarget to invoke the methods stored in its guards array one after the other on the event receiver until the first guard returns true or it runs out of guards. If a guard succeeds, the corresponding target state is retrieved looking up the targets table. Otherwise, getTarget returns null. In the later case, method handleOneEvent terminates in line 9.

If a valid transition is found, we have to execute the exit operation of the state(s) we leave and the action attached to the transition and the entry operation of the state(s) we enter. Line 11 to 15 traverse the getParent operation to climb up possible nested states until the actual source of the choosen transition is reached. Line 13 calls method leave at each nesting level. Method leave invokes the exit operation stored in the corresponding attribute, if one is provided. Next, line 16 calls method fire on the transiton object. Method fire inspects the action attribute of the transition and invokes the stored method, if provided. Finally, line 17 calls method enter on the target state. Normally, method enter just invokes the entry method stored in the corresponding attribute, if provided. However, the target state may be a complex nested state, again. In that case, the enter method determines the inital state of the contained substates via the inital association and calls its enter method, recursively. Eventually, a simple state is reached and method enter stores this state as the new current state in the actor object. Note, state-charts allow simple states to have a do activity. Such states are represented using FActivity objects. Class FActivity overrides the enter method of class FState. In addition to the normal job, the overriden enter method invokes the doAction stored in the FActivity object and on termination of the doAction it raises a ready event in order to fire potential triggerless transitions of the corresponding reactive object.

3. Note, state go(source) of Figure 2 has two outgoing triggerless transitions. Such triggerless transitions are fired by an implicit ready event raised by their source state upon termination of its do activity.
4. Note, in case of an inheritance hierarchy for events, the transitions hash-table of an FState object may contain a given transition once for the event name shown in the corresponding state-chart and in addition one time for each subkind of that event. (So far, our example application area employs very flat event hierarchies, only. However, for more complex event hierarchies, we might have to redesign our transition retrieval approach.)

Note, the described behavior of method handleOneEvent implements a so-called run-to-completion semantics for our state-chart framework.

So far, we have discussed how our generic state-table implementation deals with events, nested states, guarded transitions, and entry and exit actions. Some more effort is necessary to deal with the remaining state-chart features. For example, in case of a history state method leave of class FState has to store the former state using the history association which must be visited by the enter method later on, cf. Figure 5. In case of an and-state the actor object just stores the and-state as its current state. The and-state in turn stores the current states of all concurrent substates. Method getTransition, leave, and enter of class FAndState deal with these multiple current states, appropriately. Deferred events require an additional event queue. In order to allow multiple reactive objects of the same kind that live in a shared memory space (e.g. a large number of shuttles in our simulation environment) to share a common state-table, we employ the Flyweight pattern described in [GHJV95]. For details of these mechanisms please refer to [Köhl99].

The proposed state-table based translation of state-charts allows to deal with complex state-charts and complex state-chart features in an uniform and generic way. Following the state-table approach it is fairly simple to generate the bodies of event methods and the state-table setup routine.[5] The state-table approach avoids the generation of special purpose state-classes with large amounts of code. Extending the generation of code from class diagrams, the translation of state-charts covers the implementation of complex control-flow specifications for reactive objects.

Of course, the state-table approach causes some runtime overhead for table look-ups and the interpretative style of event execution. In simple cases the current state of an object can be determined from the value of its attributes or by inspecting the object's neighbors. One might also use an extra (enumeration) attribute, e.g. int myState, that stores the current state, explicitly. In this case the event method is often implemented via a *switch-case* statement of if-then-else-if chains (cf. [BRJ99] page 338). Such a switch-case implementation is basically restricted to the implementation of a finite state automata. More complex state-chart concepts like nested states, and-states, history-states, etc. (cf. Figure 2) cause serious problems for the switch-case approach. For example, the code handling a transition that leaves a complex nested state is duplicated for each elementary state. Each time one has to implement the execution of the exit operations of each nesting level and then one has to call the transition action and the appropriate entry actions of the reached (nested) state(s). If the state-chart contains an and-state, the object either needs to be able to be in more than one state at a time or one has to flatten the and-state by building all possible combinations of the elementary states of all and-state alternatives (and by introducing appropriate transitions). This might lead to a large number of states and in turn to huge switch statements.

[Doug98, chapter 6.2.3] uses an array based state-table that is indexed by the current state (number) and the current event (number). For each state and event the state-table stores which transition fires, i.e which new state is reached and which actions are to be performed. However, the array based state-table solution of [Doug98] still has problems dealing with complex nested states. The state-table stores elementary states only. Thus, transitions leaving a nested state at a higher level (like the emergencyStop and assign transitions in Figure 2) have to be splitted into explicit transitions for each sub-state. More seriously, complex and-states still require the building of all possible combinations of their sub-states.

Note, compared to [HG96, Rhap] we employ more than one event queue. This allows a more concurrent handling of events using multiple threads. However, this probably changes the 'order' in which events are consumed and thus has semantic relevance. In addition the usual problems attached to concurrent executions, like race conditions and deadlocks, are raised. Of course, a sound specification should avoid such problems. Static checking for (recognizable yet frequently occuring) specification errors raising these problems is the subject of our current work.

5 Combining Activity and Collaboration Diagrams

The previous chapter described the implementation of state-charts using table-driven event methods that trigger the execution of certain action methods, where the latter do the actual work. For a complete high-level visual programming language, appropriate means for the specification of these action methods are still

5. The code generator for state-charts is just under construction, cf. [Köhl99].

missing, in order to avoid that one has to deal with the nasty details of current textual programming languages. State-charts provide sophisticated means for the specification of (concurrent) control flows for reactive objects. However, by purpose state-charts abstract from the complex application specific data structures that build up the concrete states of a system. State-charts do not explicitly deal with values of attributes or with links to other objects nor with the evolution and changes of this object-structures caused by the execution of operations or action methods.

The specification of application specific object-structures is a well known application area for graph grammars, cf. [Roz97]. Basically, a graph grammar rule allows the specification of changes to complex-object-structures by a pair of before and after snapshots. The before snapshot specifies which part of the object-structure should be changed and the after snapshot specifies how it should look like afterwards, without caring how this changes are achieved. While graph grammars are appropriate for the specification of object-structure modifications, they lack appropriate means for the specification of control flows. Even the well known graph rewriting system Progress [SWZ95] provides only textual control structures.

To overcome these problems, we propose to combine state-charts and graph-rewriting rules. We use state-charts (and activity diagrams) to specify complex control flows and graph rewriting rules to specify the entry, exit, do, and transition actions that deal with complex object-structures. In order to facilitate the use of graph rewriting rules for object-oriented designers and programmers, we propose to adopt UML collaboration diagrams as a notation for object-structure rewriting rules. In UML, collaboration diagrams do not have a precise execution semantics, but model only possible scenarios. Using graph grammar theory we are able to define an execution semantics for collaboration diagrams, easily, thus enabling their translation to an object-oriented programming language.

Originally, collaboration diagrams are intended to model scenarios of complex message flows between a group of collaborating objects.[6] In addition, collaboration diagrams allow to depict the effects of operations in terms of changed attribute values and created and destroyed objects and links. Thus, the intial situation modeled by a collaboration diagram corresponds to the left-hand side of a graph grammar rule. Accordingly, the situation resulting from the execution of the collaboration diagram corresponds to the right-hand side of that graph grammar rule. This view allows the execution and translation of collaboration diagrams using techniques known from the graph grammar field, cf. [SWZ95, Zü96, FNTZ98].

Figure 9 shows an example for the combination of state-charts and object-structure rewrite rules. The left half of Figure 9 shows a refinement of state fetch of Figure 2. startFetch, the initial state of the shown diagram, has an activity shape. The activity shape is a short-hand notation for states that contain a do action

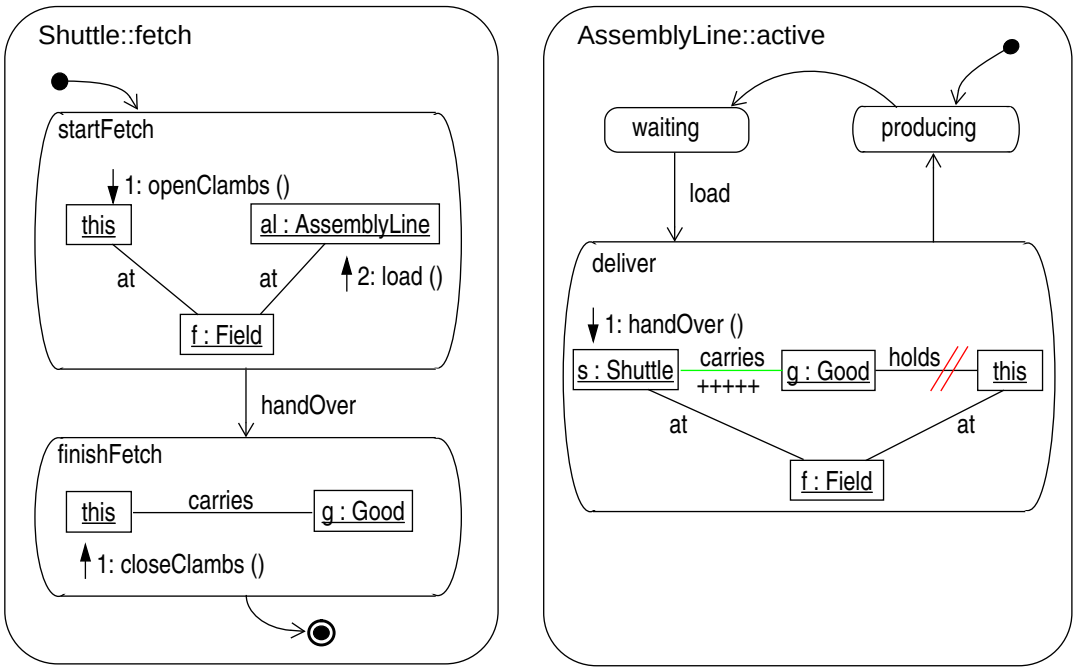

Figure 9 Refinement of state fetch of Figure 2

6. This use of collaboration diagrams is equivalent to UML sequence diagrams, cf. [BRJ99].

and that may have triggerless outgoing transitions fired on termination of the do action. In our example, activity startFetch is specified via an object-structure rewriting rule that shows three objects, this, f, and al. The this object is attached to the Field object f via an at link and the same holds for the AssemblyLine object al. We interpret this, al and f as variables and the shown links as logical constraints on the allowed values of these variables. Based on this interpretation, such a diagram is executed by binding the specified variables to concrete object instances such that all specified constraints are fulfilled.

```
 1: public class Shuttle extends FReactive { ...
 2:     boolean doStartFetch () {
 3:         Field f = this.getAt ();
 4:         if (f == null) { return false;}    // abort, signal failure
 5:         AssemblyLine al = f.getAssemblyLine ();
 6:         if (al == null) { return false;}  // abort, signal failure
 7:         this.openClambs ();  // step 1
 8:         al.load ();                // step 2
 9:     } // doStartFetch
10: } // Shuttle
```

Figure 10 Method doStartFetch of class Shuttle

In our example variable f is bound to the field object that is connected to the current shuttle object (bound to variable this) via an at link. In turn, al is bound to the AssemblyLine object that is attached to that field object via an at link. Once all variables are bound to appropriate object instances the specified change effects and method invocations are executed. Activity startFetch show no object-structue changes but two method invocations. Thus, in step 1 startFetch calls method openClambs on the this object, i.e. the current shuttle. In step 2 startFetch calls method load at the AssemblyLine al. After that the do action terminates and state startFetch is ready to accept the next event. Let us assume, that the AssemblyLine object bound to variable al is in the state waiting, cf. the left-hand side of Figure 9. Thus the load method invocation on al generates a load event that causes al to switch to activity deliver. Activity deliver is again specified by an object-structure rewriting rule. In activity deliver variable this represents the current assembly line. deliver determines its field f, the shuttle s attached to field f and a good g it holds. Activity deliver shows two object-structure changes. First, the holds link connecting variable this and g is cancelled by two small lines. This is executed by deleting the corresponding link. Second, the + symbols attached to the carries link connecting variables s and g indicates that such a link is created. Finally, activity deliver calls method handOver at shuttle s and terminates. On termination of activity deliver, the outgoing triggerless transition fires and the corresponding assembly line object switches to activity producing. Once our shuttle object receives the handOver event the corresponding transition fires and activity finishFetch is triggered. finishFetch just checks whether the shuttle actually carries a good and then it closes its clambs and terminates. Thereby, the terminal state is reached and the whole fetch activity terminates.

To implement the story diagram shown in Figure 9, its control flow parts are translated as shown in chapter 4. In addition our code generator translates the object-structure rewrite rules to normal Java code. For example the code for the do action of activity startFetch is shown in Figure 10. Object-structure rewrite rules provide much more language features for the manipulation of application specific data structures. They deal with attribute values, arbitrary application conditions, excluded links and nodes, optional and set-valued nodes, etc. Altogether object-structure rewrite rules provide sophisticated means to specify changes to complex application specific object-structures. Combined with state-charts one yields a powerful visual programming language.

6 Conclusions

This paper discussed the use of UML diagrams as a visual programming language. For class diagrams the translation to an object-oriented program is straight-forward although the uniform handling of associations is not yet common practice. The translation of state-chart is based on a table driven approach inspired by [Doug98]. The use of collaboration diagrams and their translation to Java is taken from our previous work [FNTZ98, JZ98, ZSW98]. The main contribution of this paper is the combination of state-charts and collaboration diagrams. The resulting specification language combines the power of state-charts, providing sophisticated means for modeling concurrent reactive objects, with the power of graph grammars, providing appropriate means for the specification of application specific object-structures on a very-high level of abstraction.

In addition, this paper describes a possible semantics for the various UML diagrams (via their translation to Java). This may facilitate the reading of certain UML diagrams (by thinking in terms of the corresponding

implementation concepts) and clarify ambigous modelings. In addition, one may learn which UML diagram should be used for which purpose and how different UML diagrams may be combined to cover mixed cases.

Currently, the Fujaba environment supports editing of and code generation from class diagrams and story diagrams that already combine collaboration diagrams, sequence charts, and activity diagrams. The extension towards state-charts is under construction, a first alpha version is scheduled for July 1999.

Our current work deals with the coordination of multiple concurrent reactive objects that change their number, their interrelationships and their communication channels at runtime. Since the changes of the object structure are specified by the graph grammar part of our language which provides a rich theory, we hope to be able to reason about simple cases of questions like race conditions, lifeness and safety.

The current prototype of the Fujaba environment is available as free software and comprises about 140 000 lines of pure Java code. The release of Fujaba is available via:

http://www.uni-paderborn.de/cs/fujaba.html

Acknowledgements

The following people contributed substantially to this work with fruitful discussions, careful proof readings and a lot of suggestions: Prof. W. Schäfer, Dr. A. Wagner, Dr. R. Heckel, J. Wadsack. Thank your very much

References

[AT98] J. Ali, J. Tanaka: *Implementation of the Dynamic Behavior of Object Oriented System*; IDPT Vol. 4, 19998, Proc. of third biennial world conference on integrated desgin and process technology, 281-288, ISSN No. 1090-9389, Society for Design and Process Science (1998)

[BRJ99] G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language User Guide*; Addison Wesley, ISBN 0-201-57168-4 (1999)

[Doug98] B. P. Douglass: *Real Time UML*; Addison Wesley, ISBN *0-201-32579-9 (1998)*

[Flan97] D. Flanagan: *Java in a Nutshell*; 2nd edition, O' Reilly, ISBN 1-56592-262-x (1997*)*
see also: http://java.sun.com/products/jdk/1.1/docs/api/Package-java.lang.reflect.html

[FNTZ98] T. Fischer, J. Niere, L. Torunski, A. Zündorf: *Story Diagrams: A new Graph Grammar Language based on the Unified Modelling Language and Java*; in Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, November 1998, LNCS, Springer Verlag, to appear (1999)

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Desgin Patterns*; Addison Wesley, ISBN 0-201-63361-2 (1995)

[HG96] D. Harel, E. Gery: *Executable Object Modeling with Statecharts*; Proc. 18th Int. Conf. on Software Engineering (ICSE '18), Berlin, pp 246-257, IEEE, SIGSOFT, ISBN 0-8186-7246-3 (1996)

[JGL99] Technical reference of the generic collection library for Java http://www.objectspace.com/jgl/

[JZ98] J.-H. Jahnke and A. Zündorf: *Specification and Implementation of a Distributed Planning and Information System for Courses based on Story Driven Modelling*; in proceedings of the Ninth International Workshop on Software Specification and Design April 16-18, Ise-Shima, Japan, IEEE Computer Society, pp. 77-86, ISBN 0-8186-8439-9

[Köhl99] H. J. Köhler: *Using UML as Visual Programming Language*; Master Thesis, Dep. Computer Science, University of Paderborn, in preparation (1999)

[Rhap] The Rhapsody case tool reference manual; Version 1.2.1, ILogix, http://www.ilogix.com/

[Roz97] G. Rozenberg (ed): Handbook of Graph Grammars and Computing by Graph Transformation, World Scientific, 1997.

[SWZ95] A. Schürr, A. J. Winter, A. Zündorf. *Graph grammar engineering with PROGRES*. In W. Schäfer, Editor, Software Engineering - ESEC '95. Springer Verlag, 1995.

[ZSW98] A. Zündorf, A. Schürr, and A. J. Winter: *Story Driven Modeling*; submitted to acm Transactions on Software Engineering and Methodology, August 21st 1998.

[Zü96] A. Zündorf: A Development Environment for PROgrammed Graph REwriting Systems; (in German), Dissertation, RWTH Aachen, Germany, 1996.