Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language and Java

Thorsten Fischer, Jörg Niere, Lars Torunski, Albert Zündorf

AG-Softwaretechnik, Fachbereich 17, Universität Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany;
e-mail: [tfischer|nierej|torunski|zuendorf]@uni-paderborn.de

Abstract. Graph grammars and graph rewrite systems improved a lot towards practical usability during the last years. Nevertheless, there are still major problems to overcome in order to attract a broad number of software designers and developers to the usage of graph grammars and graph rewrite systems. Two of the main problems are, (1) that current graph grammar notations are too proprietary and (2) that there exists no seamless integration of graph rewrite systems with common (OO) design and implementation languages like UML and C++ or Java.
Story Diagrams are a new graph rewrite language that tries to overcome these deficiencies. Story Diagrams adopt main features from Progres, e.g. explicit graph schemes, programmed graph rewriting with parameterized rules, negative, optional and set-valued rule elements. Story diagrams extend common graph models by offering direct support for ordered, sorted, and qualified associations and aggregations as known from the object-oriented data model. Story Diagrams adopt UML class diagrams for the specification of graph schemes, UML activity diagrams for the (graphical) representation of control structures, and UML collaboration diagrams as notation for graph rewrite rules. Story Diagrams are translated to Java classes and methods allowing a seamless integration of object-oriented and graph rewrite specified system parts.

1 Introduction

At the last graph grammar conference in Williamsburg four years ago, Blostein stated a number of requirements for the industrial use of graph grammars and graph rewrite systems as a design and implementation means, cf. [BFG96]. They should be less difficult to learn. They should be expressive. It must be possible to use them for fractions of a software system (in order to get started). Even applied to larger fractions, they should work seamlessly with standard system parts. Their execution should be fast and environments are needed.

During the last four years theory, implementation and application of graph grammars and graph rewrite systems improved a lot. In theory, the expressive power of most approaches was increased by attribute conditions, negative application conditions, general constraints, and control structures [Roz97]. Graph grammar and graph rewriting environments emerged and improved, meeting many of the requirements mentioned in [BFG96]. For example the AGG system was extended by a sophisticated graph pattern matching algorithm for the automatic execution of AGG rules, cf. [Rud97]. The Progres environment offers now means for the rapid prototyping of applications from their graph grammar specification [SWZ95a].

Despite these improvements, graph grammars and graph rewrite systems did not yet succeed to attract a broad number of software designers and developers. Two of the main problems are (1) that current graph grammar notations are too proprietary and (2) that there exists no seamless integration of graph rewrite systems and common (OO) design like UML (cf. [UML97]) and implementation languages like C++ or Java.

Story Diagrams are a new graph rewrite language that tries to overcome these deficiencies. Story Diagrams adopt main features from Progres [SWZ95a], e.g. directed, attributed, node and edge labeled graphs, explicit graph schemes, programmed graph rewriting with parameterized rules, negative, optional and set-valued rule elements. However, Story Diagrams extend the Progres graph model by direct support for ordered, sorted, and qualified associations and aggregations. Thus, the data model of Story Diagrams corresponds to the object-oriented data model.

Accordingly, Story Diagrams exploit UML class diagrams for the specification of graph schemes. Story Diagrams adopt UML activity diagrams for the (graphical) representation of control structures. The activities of a Story Diagram contain either program code (like in UML) or graph rewrite rules. The graph rewrite rules use an UML collaboration diagram like notation. Due to our experiences in several industrial projects, this notation looks quite familiar to software engineers.

Chapter 2 introduces the key features of our new graph grammar language. Chapter 3 gives a short introduction of the formal semantics of Story Diagrams. Chapter 4 outlines the translation of Story Diagrams to Java code. Chapter 5 summarizes our results and highlights some future work.

2 Story Diagrams, the language

Story Diagrams rely on an explicit graph scheme that defines static properties of the specified data structures and allows consistency checks of the dynamic specification. We use standard UML class diagrams for this purpose, since they are familiar to our target customers, i.e. software engineers. Figure 1 shows a screen shot of an UML class diagram for a lift simulator used as a running example within this paper. The screenshot is taken from the *Fujaba*[1] *environment* (cf. [FNT98]).

Note, the qualified association between houses and their levels. In standard graph models, multiple links of a certain type attached to a given node are indistinguishable and have no specific order, cf. [Roz97]. The object-oriented data model extends this concept by ordered and sorted and qualified associations. In case of an ordered associations, the

1. Fujaba is an acronym for "From UML to Java And Back Again"

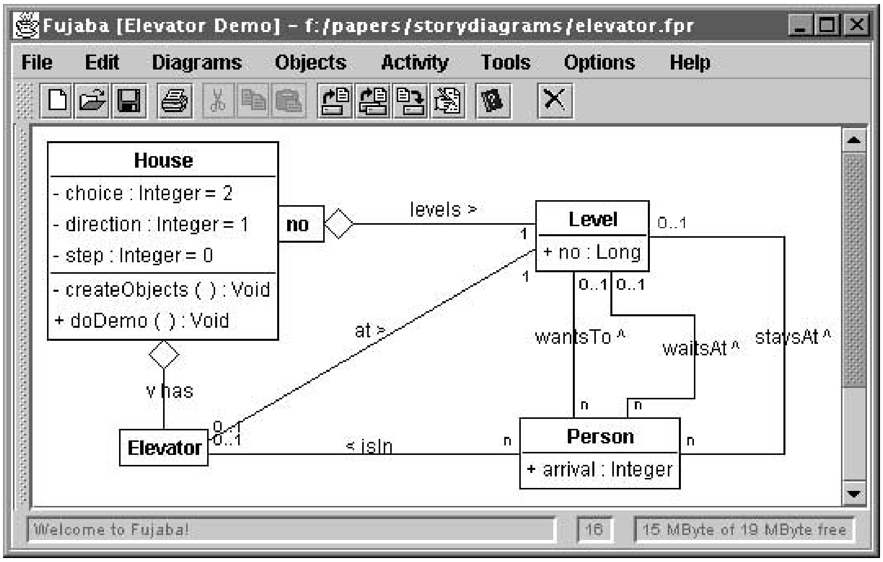


Figure 1 : Screen shot of the elevator class diagram specified with Fujaba

corresponding links attached to a given node build a list with a fixed order (defined at insertion time). Multiple links at a given node that belong to a sorted association are ordered according to a less-than operation on the reached neighbours. Qualified associations allow to distinguish between multiple links via a key value. We consider these extensions as very valuable and thus we incorporated them into Story Diagrams.

As one might have noticed, class House contains two methods createObjects and doDemo. We use Story Diagrams to specify and implement such methods. Figure 2 shows the Story Diagram for method doDemo of class House. Story Diagrams may have formal parameters for passing attribute values and object references. Story Diagrams adapt UML activity diagrams to represent control flow graphically. Thus, the basic structure of a Story Diagram consists of a number of so-called activities shown by big rectangles with rounded left- and right-hand sides. For convenient referring, the activities are numbered at their upper right corner. Activities are connected by transitions, that specify the execution sequence. Execution starts at the unique start activity represented by a filled circle. Execution proceeds following the outgoing transition(s). Multiple outgoing transitions are guarded by mutual exclusive[1] boolean expressions shown in square brackets, e.g. [this.step<100]. Diamond shaped activities express branching. When the stop activity, represented by an “bulls eye”, is reached, method execution terminates.

Despite Story Diagrams, there exists only one other notable graph rewrite language providing explicit control structures, i.e. the Progres environment. The Progres control structures preserve an atomic execution semantic for complex operations, i.e. either all

1. In general, it is not possible to check the mutual exclusiveness of different guards statically. One could check it at runtime. However, so far we generate code that just applies the different guards one after the other until the first evaluates to true.

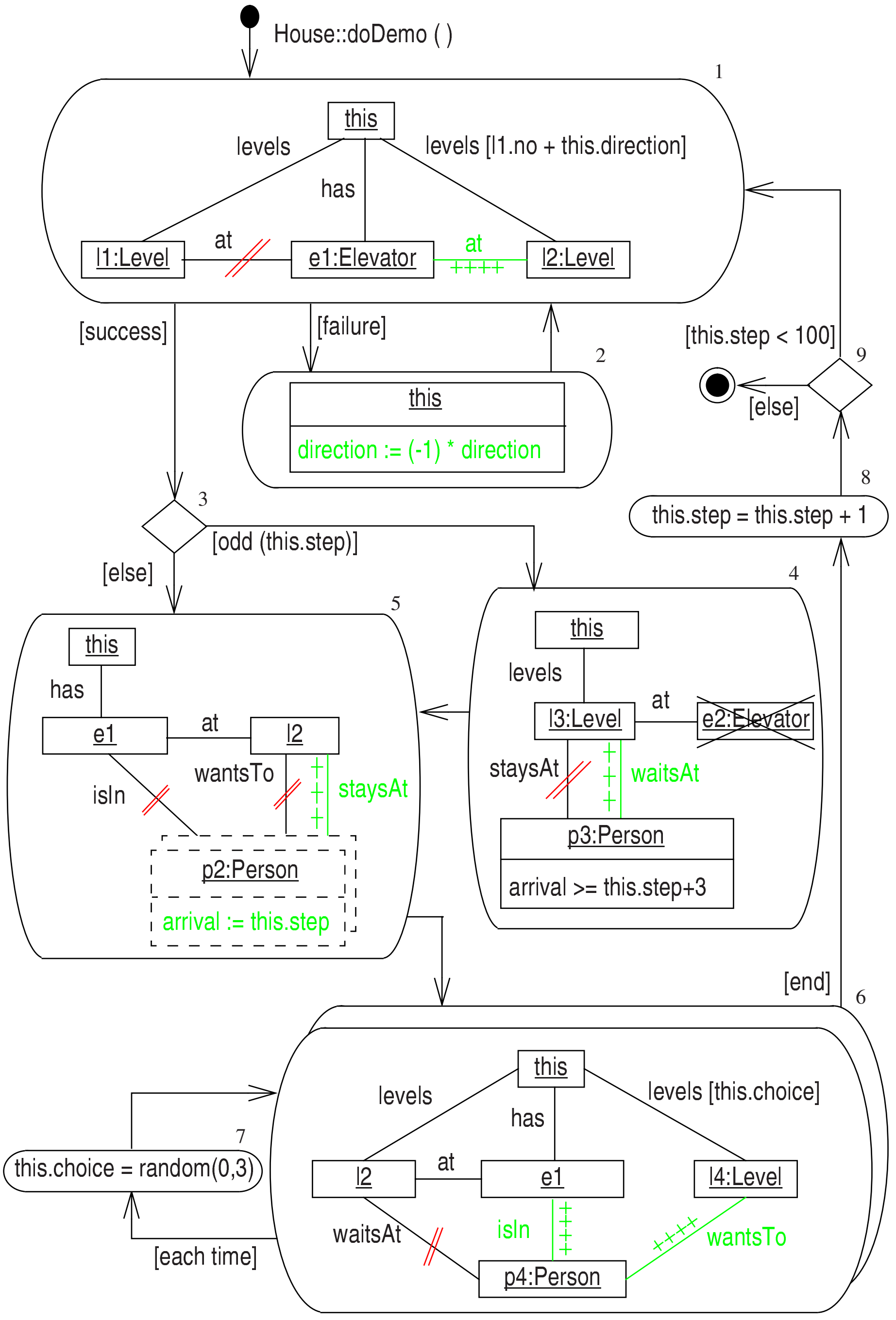


Figure 2 : The Story Diagram for the method doDemo of class House.

steps are executed successfully or none is executed at all. In addition, Progres control structures deal with the nondeterministic nature of graph rewrite rules. In general, a graph rewrite rule might have different matches within a given host graph. Usually, one of these matches is chosen randomly. Within a sequence of rewrite rules the random choice for the first rewrite rule might affect the executability of subsequent graph rewrite steps. The Progres semantics require that a sequence of graph rewrite steps must be successfully executed iff there exists choices for the application of each rule that allow the subsequent rules to be executed, too. To achieve this, Progres employs a complex backtracking mechanism for the execution of control structures.

Although these backtracking mechanisms allow to deal with the nondeterministic nature of graph rewrite rules, properly, they actually prevent the seamless integration of code generated by Progres with standard object-oriented program code. We consider this as a serious drawback with respect to the requirements stated in [BFG96]. Thus, Story Diagram control structures do NOT employ backtracking mechanisms. In case of multiple possible matches one is chosen randomly and this choice is not revisited. The user has to take care of this problem. However, this enables a direct and seamless translation of Story Diagrams into standard object-oriented code. In addition, there exists a reasonable number of big and medium sized Progres specifications of CASE tools, reengineering environments [JSZ96], and application programs originating from various authors summing up to more than 1000 pages. These real world specifications do not make use of the backtracking mechanisms of Progres. At least, there exist large application areas not requiring backtracking.

Story Diagrams support two kinds of activities, statement activities and Story Patterns. A statement activity consists of a chunk of Java code[1], e.g. used for I/O-operations, mathematical computations and method invocations, which are hard to express in graph rewrite rules. A Story Pattern is a graph rewrite rule showing left- and right-hand side within one picture. According to our experiences with 1000 pages of Progres specifications, graph rewrite rules often look-up quite complex patterns but usually they perform only small modifications. In these cases, the graph rewrite rule consists of quite large left- and right-hand sides which are very similar to each other and one has to compare the two sides thoroughly in order to determine the specified effects (modifications). Thus, the combination of left- and right-hand side into a single picture results in a more concise and readable notation.

Actually, we use a kind of UML object or collaboration diagram as notation for graph rewrite rules. This notation is quite familar to our target customers, i.e. software engineers. The depicted object structure defines the left-hand side of our rule. Creation of objects and links is shown by attaching a series of + symbols. Deletion is shown by canceling objects and links with two parallel lines.[2] Optionally, created objects and links

1. So far, this code is not checked by the Fujaba environment but just copied into the generated method implementation. Adequate compile time checks are current work.
2. In UML collaboration diagrams, one can already indicate creation and deletion using {new} and {destroyed} constraints, respectively. However, due to our experiences in teaching Story Diagrams to students, these constraints are easily overlooked. Thus, we employ the notational adaption possibilities of UML that allow to depict such constraints using specific icons.

are shown in green color, deleted objects and links are shown in red color. Thus, the left-hand side of the depicted rule consists of the normal objects and edges together with the (red) canceled objects and links. The right-hand side consists of the normal and created objects and links. This notation was inspired by the Sesam notation, cf. [LD96].

In object-oriented terms, a Story Pattern represents a complex boolean condition on a number of bound and unbound variables (like a term in first order logic). Unbound variables are shown as boxes containing name and type, e.g. $\boxed{\underline{\text{e1 :Elevator}}}$. Bound variables are shown as boxes containing only their name, e.g. $\boxed{\underline{\text{e1}}}$. Subsequent Story Patterns may use variables bound in previous Story Patterns (or statements) of the same Story Diagram. In addition, formal parameters and the self reference “this” may serve as bound variables. A link in a Story Pattern represents the boolean condition that the objects matched by the corresponding variables are connected by such a link.

Generally, a Story Pattern is executed by binding all its unbound variables to objects such that the represented condition evaluates to true. If this is possible, the specified modifications are performed and the Story Pattern succeeds, otherwise it fails. One may use the special keywords success and failure as guards for transitions leaving Story Patterns to branch on the success or failure execution of a Story Pattern. Generally, Story Patterns are restricted to isomorphic matches. One may state exceptions explicitly using {maybe v1 = v2} constraints. The identification condition is ensured statically.

For example, Story Pattern 1 of Figure 2 consists of one bound variable (this) and three unbound variables e1, l1, and l2. Elevator e1 needs to be connected to the current house (denoted by this) via a has link and to level l1 via an at link. Levels l1 and l2 are reached from this via levels links. The levels link leading to l2 is qualified by the number of level l1 plus the value of the direction attribute of this (this.direction may contain 1 or -1). If a match is found, Story Pattern 1 cancels the at link between e1 an l1 and adds a new at link between e1 and l2. This models a move of the elevator to the next level in the current direction. Story Pattern 1 may fail if no level l2 exists above or below the current level. In that case, execution follows the [failure] transition and Story Pattern 2 toggles the current direction and revisits Story Pattern 1. Note, attribute assignments are shown in the so-called attribute compartment of a variable. We use the := operator to distinguish attribute assignments from (equality) attribute conditions (shown in the same compartment). The latter employ the == operator.

After a successful move, branch activity 3 is reached. On every odd step, we execute Story Pattern 4. Story Pattern 4 looks for an arbitrary Person p3 staying at an arbitrary level l3 for at least 3 steps. The latter is required by the attribute condition arrival >= this.step+3 shown in the so-called attribute compartment of variable p3. In addition, the *negative* variable e2 *(crossed out by a big X)* requires that no elevator exists at the chosen level l3. If Story Pattern 4 finds a match, it activates person p3 via replacing the staysAt link by a waitsAt link.

Story Pattern 5 contains a set-valued variable p2 shown by two stacked dashed rectangles. Set-valued variables represent not just one node but the set of all nodes that match the depicted constraints. Thus, p2 models the set of all person(object)s in elevator e1 (cf. the isIn link) that want to go to level l2 (cf. the wantsTo link). For all persons in p2, Story Pattern 3 cancels the attached isIn and wantsTo links and creates staysAt links, modeling that these persons leave the elevator and stay at their target level. Note, that variable e1 and l2 have been bound during the execution of Story Pattern 1 and still refer to the same objects in Story Pattern 5. In addition to set-valued variables Story Pattern provide optional variables (shown as dashed rectangle). Optional variables are bound to nodes if possible and ignored otherwise.

Story Pattern 6 represents a so called for-all Pattern indicated by two stacked activity shapes. A for-all Pattern is executed for all possible bindings of its variables. Story Pattern 6 contains two unbound variables, p4 and l4. Thus, each person p4 waiting at the current level l2 stops waiting (cf. the canceled waitsAt link) and enters the elevator (cf. the added isIn link) and presses his personal target button. The latter is simulated by adding a wantsTo link to level l4 which is determined by the value of attribute this.choice.

A for-all pattern may have subsequent activities executed for each of its matches. Therefore, we provide the special transition guards [each time] and [end]. Each time Story Pattern 6 is applied, execution proceeds with activity 7. Activity 7 contains a simple pseudo code statement assigning a new random value to this.choice. Thus, for each Person waiting at the current level, Story Pattern 6 selects a new randomly chosen target level.

The remaining activities count the number of elevator steps and limit the number of simulation steps. Our example contains no ordered or sorted links. For the means provided to deal with ordered or sorted links see [FNT98].

3 Formal foundations

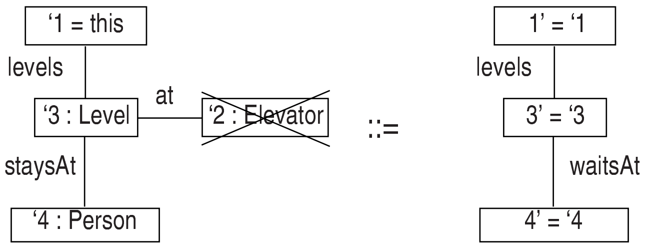
The semantics of Story Diagrams are based on a translation to Progres. Despite sorted, ordered, and qualified associations, the object oriented data model of Story Diagrams corresponds to the directed, attributed, node and edge labeled graphs of Progres. Thus, a Story Pattern corresponds directly to a Progres production where the left-hand side is derived from the normal and canceled Story Pattern parts and the right-hand side corresponds to normal and created parts. Within a Story Diagram, variables bound to objects in one Story Pattern may be used as bound variables in subsequent Story Patterns in the same diagram. This passing of Story Pattern variable values is simulated by Progres production parameters. Attribute handling is translated to Progres conditions and transfer clauses. Optional, set-valued and negative parts have direct correspondencies. For example, the Progres production simulating Story Pattern 4 is shown in Figure 3.

Story Diagrams are restricted to so-called well-formed control flow. Basically, the control flow built by the transitions of a Story Diagram must represent nested sequences, branches, and loops. This enables a translation into Progres sequence-, choose-, and loop-statements. However, much care has to be taken to avoid the special backtracking mechanisms of Progres, that are NOT part of the Story Diagram semantics. Basically, triggering backtracking is avoided by embedding the productions simulating single Story Patterns into a

choose when *Step_X* then sdm__Success := true else sdm__Success := false end

construct. If the execution of Step_X fails, the second choose branch is executed (which

production doDemoStep4 (this : House, out l3 : Level, out p3 : Person)
=



condition '4.arrival >= '1.step+3;
return l3 = 3'; p3 = 4';
end

Figure 3 : Simulating Story Pattern 4 with Progres

will succeed). After all, the special boolean variable sdm__Success signals success or failure of Step_X and may be used for subsequent branching.

Using such a translation, the well defined and elaborated semantics of Progres provides us with sound answers to a lot of detailed questions e.g. concerning the combination of multiple set-valued and optional and negative nodes within one rule or the handling of the identification condition in case of non-isomorphic matches. However, the translation of sorted, ordered, and qualified associations to Progres needs additional efforts since such elements are not provided by the Progres graph model. This translation is not yet formally specified. On the other hand, a semantic definition through a translation into Progres answers semantics questions to graph grammar people, only. It is of little help for our target customers, i.e. software engineers. Thus, a direct formalization staying close to object-oriented concepts is future work.

4 Translating Story Diagrams to Java

Story Diagrams drop the backtracking mechanisms of Progres. This enables the *Fujaba environment* (cf. [FNT98]) to generate standard object-oriented (Java) code from Story Diagrams. This Java code uses normal main-memory objects and does not rely on any huge library (like e.g. Progres generated code relies on GRAS, the Progres object database). Only some small helper functions are used to make the code more readable. Thus this code works seamlessly with other parts of a system (e.g. the graphical user interface). Due to the usage of Java, the code is platform independent.

Figure 4 to 7 show a cut-out of the Java code generated for class House of our example. First, the Fujaba generator translates UML class diagrams to Java. UML classes correspond directly to Java classes, cf. line 3. According to Java style guides, we translate UML class attributes to private Java attributes accessible via appropriate get- and set-methods. For example, attribute direction has methods getDirection and setDirection, cf. line 4 to 10.

According to UML semantics, associations are bi-directional. Thus, we implement associations by pairs of references in the respective classes. For example, association has

```
 1: import java.util.*;
 2: import com.objectspace.jgl.*;
 3: public class House {
 4: private int direction = 1;
 5: private int getDirection ( ) {
 6:    return this.direction;
 7: } // getDirection
 8: private int setDirection (int newDirection){
 9:    return this.direction = newDirection;
10: } // setDirection
11: private OrderedMap levels
12:                   = new OrderedMap ();
13: public void addToLevels (Level elem) {
14:    if (!this.hasInLevels (elem)){
15:       Integer k = new Integer (
16:                              elem.getNo ()));
17:       this.levels.add (k, elem);
18:       elem.setRevLevels (this);
19: }  } // addToLevels
20: public void removeFromLevels (...) { ... }
21: public boolean hasInLevels (Level elem) {
22:    if (elem == null) { return false;
23:    } else {
24:       Integer k = elem.getNo();
25:       return (this.levels.get(k) != null));
26: }  } // hasInLevels
27: public Enumeration elementsOfLevels ( )
28: {  return this.levels.elements (); }
29: public Level getFromLevels (int key) { ... }
30: private Elevator has;
31: public Elevator getHas () { ... }
32: public Elevator setHas () { ... }
33: ...
34: public void createObjects () { ... }
35: public void removeYou () { ... }
```

Figure 4 : Java class House, part a

of Figure 1 is implemented by an attribute has of type Elevator in class House (cf. line 30) and a reverse reference attribute revHas[1] of type House in class Elevator. Again, we provide appropriate access methods for these attributes (cf. line 31 and 32). For multi-valued associations we use standard container classes provided by the java generic library [JGL98]. We use class OrderedSet (balanced sorted trees) for normal (and sorted) associations and class DList (doubly linked lists) for ordered associations. We could implement qualified associations via class HashMap (hash tables). However, we use class OrderedMap (balanced binary tree of key-value pairs) since during pattern matching we frequently traverse the whole collection which is less efficient for hash tables. Thus, association levels of Figure 1 is implemented by attribute levels of type OrderedMap in class House (cf. line 11 and 12) and attribute revLevels of type House in class Level.

For container class attributes we provide appropriate access methods, that allow to add and remove elements (e.g. addToLevels line 13 and removeFromLevels line 20), to check the containment of an element (e.g. hasInLevels line 21), to visit all contained elements (e.g. elementsOfLevels line 27), and in case of a qualified association to retrieve an element via a key (e.g. getFromLevels line 29).

In order to guaranty the consistency of the pairs of references that implement the associations, the respective access methods for reference attributes call each others. For example, in line 18 method addToLevels calls method setRevLevels on the added level object elem, passing itself as parameter. This establishes the reverse reference. In the same way, removeFromLevels resets the revLevels reference at the corresponding Level object. The same holds for set methods of single valued reference attributes. Thus, to establish a pair of references one just calls the appropriate set or addTo method on

1. If no explicit role name is given, we use the prefix ´rev´ for the reverse reference.

```
36: public void doDemo () { ...
37:    while (!sdm__Success){
38:       // story pattern (1)
39:       sdm__Success = false;
40:       try{
41:          e1 = this.getHas(); // bind e1
42:          SDM.ensure (e1 != null);
43:          l1 = e1.getAt(); // bind l1
44:          SDM.ensure (l1 != null);
45:          SDM.ensure (l1.getRevLevels() ==
46:                                           this);
47:          l2 = this.getFromLevels (l1.getNo()
48:                + this.getDirection()); // bind l2
49:          SDM.ensure (l2 != null);
50:          // check isomorphic binding
51:          SDM.ensure (l1 != l2);
52:          e1.setAt (null); // delete link
53:          e1.setAt (l2); // create link
54:          sdm__Success = true;
55:       } catch (SDM.Exception e) { }
```

Figure 5 : Java class House, part b

one of the participating objects and passes the other as parameter. For example, the call e1.setAt (l2) in line 53 of Figure 5 creates an at link between elevator e1 and level l2. For the removal of links, again an appropriate method call on one of the objects suffices, cf. line 52.

Now, the Fujaba generator is able to translate Story Diagrams to standard Java code using the implementation of classes, attributes, and associations described above. Story Diagrams become methods of the depicted classes. The translation of Story Diagrams consists of two main tasks, translating the control flow of Story Diagrams and translating Story Patterns. Story Diagrams are restricted to so-called well-formed control flow. Basically, the control flow built by the transitions of a Story Diagram must represent nested sequences, branches, and loops. This enables a direct translation into Java block-, if-, and while statements. For example, the control flow of Figure 2 corresponds to a while statement (cf. line 37) containing the code of Story Pattern 1 (line 38 to 55) followed by an if statement (line 56, Figure 7) with a success and a failure branch (line 92). Within the success branch activity 3 corresponds to an if statement (line 57) with Story Pattern 4 in its true branch (line 58 to 80). This if statement is followed by Story Pattern 5 (line 82). For-all Story Pattern 6 corresponds to a while statement (line 84) containing activity 7 (line 87). This while is followed by activity 8 (line 89) and an if statement (line 90) for activity 9. The if statement for activity 9 contains a return statement in its else branch (line 91). The failure branch (line 92) of the if statement (line 56) following Story Pattern 1 contains just Story Pattern 2 (line 93).

Our translation of Story Patterns uses the same graph pattern matching optimisation strategies as Progres, cf. [Zün96, SZ94, FNT98]. However, the code of Progres prototypes relies on special control flow interpreters that guide the execution of (non-deterministic) search steps within a big switch statement. Dropping these backtracking mechanisms enabled us to generate more seamless Java code using normal control structures.

Generally, the code generated for a Story Pattern consist of nested loops for each unbound variable in the Story Pattern, cf. Figure 6. At each level, we choose an already bound variable v0 and a link l1 connecting it to a still unbound variable v1. Then, we generate a loop that iterates through the set of nodes reached via l1 and bind these candidates to v1, one after the other. Within the loop, we generate nested if-statements that check all conditions related to v1. Once all conditions related to v1 are considered, we proceed with the next unbound variable. Altogether, this generates nested search loops that iterate through all possible combinations of variable bindings. Once all variables are bound and all condition checks are passed, i.e. at the inner most nesting level, a match is found and code executing the graph rewrite step is reached. In addition, the successful execution is signaled by assigning true to the special boolean variable sdm__Success. At each search loop level this variable is then used to terminate the search process.

```
a: for all (v1 reached from a bound variable via a depicted link) do {
b:    if (condition 1 on v1 holds) {
c:       ...
d:          if (condition k on v1 holds) {
e:             ....
f:                for all (vn reached from a bound variable via a depicted link) do {
g:                   ...
h:                      if (condition vm holds von vn) {
i:                         /* found a match */
j:                         /* execute modifications */
k:                         /* terminate search */
l:                      } // if
m:                   ...
n:                } // for all vn
o:             ...
p:          } // if
q:       ...
r:    } // if
s: } // for all v1
```

Figure 6 : Nested loops and conditions for Story Pattern matching

Actually, we generate slightly different code for the checking of conditions. For large rewrite rules the nested if-statements soon reach an excessive nesting depth. To avoid this, we use the exception mechanism of Java. We replace nested if-statements by a single try-catch block and use a small helper function called SDM.ensure to check conditions. In case of a failure, SDM.ensure just throughs an exceptions and thus leaves the whole try-catch block.

In case of Story Pattern 4 variable this is the only bound variable. The first search loop iterates through all levels l3 that are reached from this via a levels link, cf. lines 60 and 61 to 63 in Figure 7. Additionaly, the try-catch block from lines 64 to 79 surrounds all conditions related to variable l3. For example, line 65 checks whether traversing an at link against its direction[1] reaches an elevator (which is excluded in Story Pattern 4). The second search loop (line 68) iterates through persons p3 that stay at level l3. The body of the second search loop is surrounded by its own try-catch block (cf. line 64 to 79 and 71 to 77). Thus, a search exception raised e.g. in line 72 is caught in line 77 and another

1. to indicate the reverse direction of an association, we use the name prefix ‘rev’

```
56:     if (sdm__Success){
57:       if (Math.odd (this.getStep()){
58:         sdm__Success = false; // activity (4)
59:         Enumeration l3Enum =
60:                     this.elementsOflevels ();
61:         while ( ! sdm_Success
62:               && l3Enum.hasMoreElements ()) {
63:           l3 = l3Enum.nextElement ();
64:           try {
65:             SDM.ensure (l3.getRevAt () == null));
66:             Enumeration p3Enum =
67:                       l3.elementsOfRevStaysAt ();
68:             while ( ! sdm__Success &&
69:               && p3Enum.hasMoreElements ()) {
70:               p3 = p3Enum.nextElement ();
71:               try {
72:                 SDM.ensure (p3.arrival >
73:                               this.getStep () + 3);
74:                 p3.setStaysAt (null);
75:                 p3.setWaitsAt (l3);
76:                 sdm__Success = true;
77:               } catch (SDM.exception e) { }
78:             } // while p3Enum
79:           } catch (SDM.exception e) { }
80:         } // while l3Enum
81:       } // if
82:       ... // story pattern (5)
83:       sdm__Success = true;
84:       while (sdm__Success){
85:         ... // story pattern (6)
86:         if (sdm__Success) {
87:           this.choice=Math.random(0,3); //step 7
88:       } }
89:       ... // statement 8
90:       if (this.step < 100){ // step 9
91:       }  else {  return; }
92:     }  else {
93:       ... // story pattern (2)
94: }  }  }  } // doDemo
95: } // House
```

Figure 7 : Java class House, part c

iteration of the second search loop starts at line 68. In that case, line 69 checks whether (still) another person stays at the current level. If iterator p3Enum contains more elements, line 70 binds variable p3 to the next person and the constraints are checked again. If iterator p3Enum runs out of elements, the inner search loop terminates and the outer search loop tries to bind l3 to the next level. If we run out of levels, we terminate and the Story Pattern has failed (sdm__Success is still false). Once a level without elevator is found and the constraint of line 72 to 73 is fulfilled, the modifications of the Story Pattern are executed (cf. line 74 to 75) and sdm__Success becomes true (cf. line 76) terminating the while statements and signaling success.

The code of Story Pattern 4 use while loops and elementsOf methods in order to traverse to-many associations. In case of to-one association, this is simplified to a association look-up using a get method and a simple check for a not-null value. For example, the code for Story Pattern 1 of Figure 2 first binds variable e1 by calling e1=this.getHas() (cf. line 41, Figure 5). Next, line 42 checks whether this was successful using the condition e1!=null. Once e1 is successfully bound, we call l1=e1.getAt() to determine the current level l1 (line 43). Line 47 to 48 bind variable l2 via the qualified levels association. Line 45 and 46 show a link-constraint example ensuring that l1 is a level of the current house.

If all variables are bound and each constraint is checked, we execute the modifications of the Story Pattern. In our example we cancel the old at link (cf. line 52) and create a new at link connecting e1 and l2 (cf. line 53). Finally, we assign true to variable sdm__Success signaling the successful Story Pattern execution (cf. line 54). This is used in line 56 to determine the next activity.

Note, our pattern matching strategy requires that each (connected component of a) Story Pattern contains at least one bound variable that allows to reach all its unbound variables by traversing links. Other approaches like Progres and AGG have no such limitation. However, those approaches require an explicit graph or at least explicit object extensions in order to be able to search for matches. Managing such an explicit extension in our approach would disable the Java garbage collector to discard objects, automatically. Thus, it would re-introduce the problem of memory leaks to Java and aggravate a seamless integration with other system parts. Thus we decided to abandon such an explicit graph. Due to our experiences with up to 1000 pages of Progres specification, requiring patterns to contain a bound variable allowing to compute all other objects is not a severe restriction. One always finds some kind of root objects that he or she can use to access the desired objects.

5 Conclusions

Story Diagrams aim to push graph grammars to a broader industrial usage. Therefore, Story Diagrams adapt standard object-oriented modeling languages i.e. UML class diagrams, activity diagrams, and collaboration diagrams. Generally, Story Diagrams enhance object-oriented software development methods by appropriate means for modeling the evolution and dynamic behavior of complex object structures. In [JZ98] we propose Story Driven Modeling as a new method for the software development based on Story Diagrams and show its application in a case study. Story Driven Modeling is an extension of [SWZ95a].

The semantics of Story Diagrams is based on its predecessor Progres. However, we enhanced the data model of graphs towards the object-oriented data model. In addition, we dropped the backtracking mechanisms of Progres since extensive experiences have shown that it is seldom used. Dropping the backtracking mechanisms enabled us to translate Story Diagrams and Story Patterns into standard object-oriented Java code. The generated code does not require an extensive library and may be integrated seamlessly with other system parts and is platform independent due to the usage of pure Java. Currently, the generator adapts the graph pattern matching algorithm of Progres. Incorporation of the back jumping techniques developed by [Rud98] is current work.

Currently, the Fujaba environment allows convenient editing of Story Diagrams and UML class diagrams, cf. [FNT98]. Fujaba provides a reasonable number of consistency checks integrating class diagrams and Story Diagrams. The Fujaba generator supports the translation of class diagrams and Story Diagrams and a reasonable subset of Story Patterns to Java code. In addition it comprises a simple object structure browser used to visualize and debug the execution of Story Diagrams. The Fujaba environment is programmed in pure Java and thus available on all standard platforms. The Fujaba environ-

ment underlies the GNU-Licence and is available via the internet under
http://www.uni-paderborn.de/cs/fujaba/index.html

References

[BFG96] D. Blostein, H. Fahmy, A. Grbavec: Issues in the Practical Use of Graph Rewriting. In *Proc. 5th. Int. Workshop on Graph-Grammars and their Application to Computer Science*; LNCS 1073, pp. 38-55, Springer

[FNT98] T. Fischer, J. Niere, L. Torunski: Design and Implementation of an integrated development environment for UML, Java, and Story Driven Modeling. Master Thesis, University of Paderborn (In German)

[JGL98] Technical reference of the generic collection library for Java http://www.objectspace.com/jgl/

[JSZ96] J.-H. Jahnke, W. Schäfer, and A. Zündorf: A Design Environment for Migrating Relational to Object Oriented Database Systems. In *Proceedings of the International Conference on Software Maintenance* (ICSM ´96), IEEE Computer Society 1996

[JZ98] J.-H. Jahnke and A. Zündorf: Specification and Implementation of a Distributed Planning and Information System for Courses based on Story Driven Modeling. In *Proceedings of the Ninth International Workshop on Software Specification and Design* April 16-18, Ise-Shima, Japan, IEEE CS, pp. 77-86

[LS88] C. Lewerentz and A. Schürr. GRAS, a management system for graph-like documents. In *Proc. of the 3rd Int. Conf. on Data and Knowledge Bases*. Morgan Kaufmann, 1988.

[Rud97] M. Rudolf: Design and Implementation of an Interpreter for attributed graph rewriting rules. Master Thesis, Technical University of Berlin (In German)

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

[LD96] J. Ludewig, M. Deininger: Teaching Software Project Management by Simulation: The SESAM Project; in Irish Quality Association (eds.): *5th European Conference on Software Quality*, Dublin, pp. 417-426

[SWZ95a] A. Schürr, A. Winter, A. Zündorf: Graph Grammar Engineering with PROGRES; in: Schäfer W. (ed.): Software Engineering - ESEC '95; LNCS 989, pp. 219-234; Springer (1995)

[SWZ95b] A. Schürr, A. Winter, A. Zündorf: Visual Programming with Graph Rewriting Systems; In: Proc. VL'95 11th Int. IEEE Symp. on Visual Languages, Darmstadt, Sept. 1995, IEEE Computer Society Press (1995)

[UML97] UML Notation Guide version 1.1. Rational Software, http://www.rational.com/uml/

[Zün96] Zündorf A.: Graph Pattern Matching in PROGRES; In:J. Cuny,H. Ehrig,G. Engels,G. Rozenberg (eds): In *Proc. 5th. Int. Workshop on Graph-Grammars and their Application to Computer Science*; LNCS 1073, pp. 454-468 Springer