On Semantic Issues in Story Diagrams

Matthias Tichy, Matthias Meyer, and Holger Giese
Software Engineering Group
University of Paderborn, Germany
[mtt|mm|hg]@uni-paderborn.de

ABSTRACT

Story Diagrams provided by Fujaba are a powerful visual formalism for the specification of structural transformations. Their visual appearance is based on UML activity and UML collaboration diagrams. The semantics draws on graph transformation systems and therefore has a solid foundation. However, as reported in this paper, there are several subtle problems with the semantics and the code generation of Story Diagrams. These problems have been discovered during our efforts to make Story Diagrams applicable for embedded real-time systems and to verify Story Diagrams. We will outline a selection of identified semantic problems in this position paper and will discuss several possible solutions we see which seem more appropriate than the present solution.

1. INTRODUCTION

One cornerstone of the Fujaba Tool Suite and its standard notation to describe behavior are Story Diagrams [2]. They provide a powerful yet intuitive visual formalism for the specification of structural transformations. A central element of Story Diagrams which describe the single transformation steps are Story Patterns which extend UML collaboration diagrams in a natural manner to describe pattern for partial instance situations which should be queried and the side effects which should take place when a match for the pattern has been found. These Story Patterns are then composed to complex transformation by means of UML activity diagrams which describe the control flow between the basic actions.

The semantics of Story Patterns and Story Diagrams [4] is based on graph transformation systems. Graph transformation systems provide a direct mapping for most of the concepts in Story Patterns and Story Diagrams. However, more complex UML concepts such as multiplicities or qualified associations have to be covered differently.

In this position paper, we will report about several subtle problems with the semantics and code generation for Story Diagrams and Story Patterns which have been discovered.[1] We at first looked into ways to make their benefits available in more demanding application areas such as embedded real-time systems where the in the paper outlined present "solutions" are often not acceptable. In addition, we started looking into the problem of verifying Story Patterns [1] and Story Diagrams which lead to the discovery of further problems of semantic issues.

[1]We considered the current version of Fujaba 4 and its built-in Java code generation.

We will outline a selection of detected semantic problems in this position paper and will discuss several possible solutions we see. We will therefore initially review what we consider a appropriate semantics and the resulting requirements the code generation should fulfill in Section 2. To further structure our discussion of semantic issues, we at first look into an number of problems which arise already for Story Patterns in Section 3. Then, we consider in Section 4 problems which relate to the control flow between the Story Patterns when combining them in an overall Story Diagram. Finally, we close the paper with a discussion of our finding and an outlook on planned future work.

2. PREREQUISITES

What is a good semantics or more specifically in our case what are semantic weaknesses which have to be avoided when defining a language and supporting it with a tool? Let us first clarify that we do *not* want to discuss the definition of the semantics or its elegance or appropriateness but rather only the resulting semantics of the language relevant for those who use a language.

Our guideline for a "good" formal semantics can be characterized by the following phrase:

> A formal semantics should assign an unambiguous meaning to each syntactically allowed phrase of the language.

Using this simple statement we can derive the following required properties for the formal semantics:

- The semantics should not be *incomplete*. This means that the semantics must assign a meaning to all syntactically correct instances of a language.
- Each instance of the language should have a unique meaning and thus the semantics should not be *ambiguous*.

While these criteria are important, in the considered cases the semantic issues are not only related to the formal semantics. We also have a *compiler semantics* which is determined by the behavior resulting from the code generation. In addition, we have something like an *intuitive semantics* characterized by the intuitive understanding of an average developer. Therefore, consistency between these semantics is of crucial importance.

From a purely formal point of view we could ignore the intuitive semantics and only require that the compiler semantics respects the formal semantics (otherwise we have a

semantically incorrect code generation). However, in practice it is important to ensure that both the formal semantics as well as the compiler semantics are not at odds with the intuitive semantics. Otherwise, the resulting language is highly misleading and prone to error.

Using our former requirements as starting point, we can then derive the following weaknesses which should be avoided if possible:

- We have an *unexpected instance* if the intuitive understanding cannot map any reasonable meaning to it while it might have a nicely defined formal semantics or compiler semantics.[2]
- We have an *unexpected error* if there are instances of the language which have a proper intuitive meaning. However, the formal or compiler semantics are erroneous for this instance of the language.
- There should be no instances of the language where the formal semantics is in conflict with the intuitive understanding of the average developer. Nevertheless, this formal semantics can be objectively considered as reasonable. We name such ambiguities due to the conflict between intuition and defined semantics *counterintuitive*.

In the following, we present the semantic issues in Story Patterns and Story Diagrams as well as classify them by the above defined weaknesses.

3. ISSUES IN STORY PATTERNS

3.1 Undefined Order of Creations

One of the key features of Story Patterns is the ability to model dynamic changes of object structures. Story Patterns are frequently used to specify the deletion or creation of objects and links between them.

3.1.1 Problem Description

The upper part of Figure 1 shows a class diagram modeling two classes A and B and a one-to-one association ab between them. The Story Pattern below first creates two objects b1 and b2 of type B. Next, links of type ab are created between both new B objects and the bound object a (of type A).

Since ab is a one-to-one association, object a can only be connected to one B instance at a time. Thus, the Story Pattern is intuitively invalid and classifies as an *unexpected instance*. Another problem is, that the order in which the links are created is not defined. Consequently, if only one B object is connected with a after the execution of the pattern, it is not clear which (b1 or b2).

The order in which objects are created is not defined either. In general, the order of object creations does not matter. However, in situations where the creation of objects fails, e.g. because of insufficient memory (cf. Section 4.2), it is unclear which of the creations have been successful. The same is true for failed link creations.

[2]In this case it would be better to have certain constraints which must hold for a valid instance in addition to the syntactical correctness by means of explicit *well-formedness rules*. However, due to computational limits, for very expressive languages not all well-formedness rules required to exclude unexpected instances can be effectively checked.

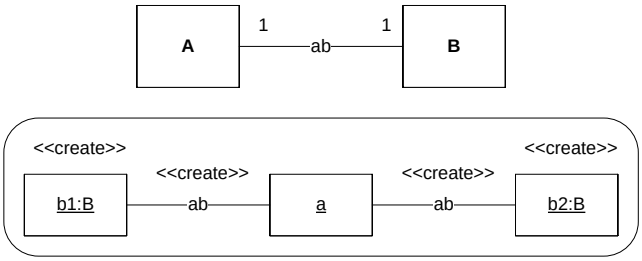

Figure 1: Undefined Order of Creations

3.1.2 Current Semantics

According to the semantics definition in [4], in case of a to-one association, an existing link is replaced upon creation of a new one. In addition, all object creations are executed first and all links are created afterwards. The current code generation adheres to these definitions.

The order in which objects or links are created is undefined and thus non-deterministic.

3.1.3 Similar Problems

Similar problems arise in case of ordered to-n associations, where linked objects are stored in a list. Assume the ab association in Figure 1 is an ordered one-to-n association. Then, both B objects are connected to a after the execution of the Story Pattern. However, it is undefined again, whether b1 will be before b2 in the list of linked objects or the other way round.

Furthermore, the language definition in [4] allows for inserting single objects into an ordered association at specific positions. In Figure 2, the first object linked to a via the ordered ab association is bound to b1. Then, a new object b2 is created and linked to a as direct successor of b1 (indicated by the next arrow). This is in accordance with the formal semantics. The current code generation, however, does not support this feature.

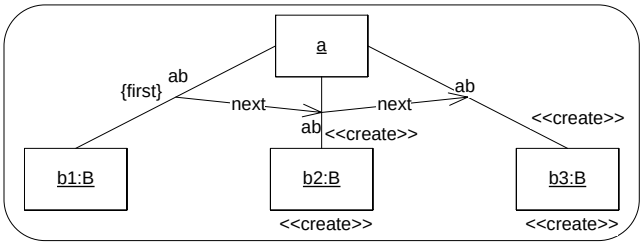

Figure 2: Inserting a sequence into an ordered association

In the above figure, another object b3 is created and linked to a as successor of b2. This insertion of a sequence of objects in a defined relative order is not supported at all.

3.1.4 Possible Solutions

Conflicting link creations in case of to-one associations can be detected at specification time and should simply be forbidden.

The designer should be able to explicitly define the order of creations by assigning an index to each creation. Further-

more, it should be possible to mix object and link creations by defining a global order of creations. Thereby, the complete creation of more important structures could be favored over other creations.

For the insertion of objects into ordered associations at specific positions, the current code generation has to be fixed. In order to enable the insertion of whole sequences of objects, the semantics definition as well as the code generation would have to be extended.

3.2 Matching links between sets

Story Patterns allow the binding of an arbitrary number of objects of the same type to a single multi-object or set variable [4]. Set objects greatly simplify the manipulation of multiple objects. Creating a link to a set object, for example, results in the creation of a link to each contained object. The change of an attribute value specified at a set object is done to all its members.

3.2.1 Problem Description

The class diagram in the upper part of Figure 3 defines three classes A, B, and C. Instances of B may be connected to an arbitrary number of instances of A and C via associations ab and bc, respectively. Furthermore, A and C participate in the n-to-n association ac.

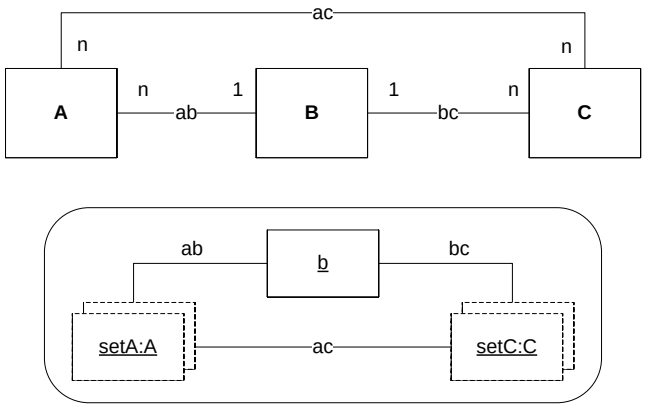

Figure 3: A link between two sets

The Story Pattern shown below the class diagram binds all objects of type A which are connected via ab to the bound object b (of type B) to the set object setA. All objects of type C which are linked to b via bc are bound to the set object setC.

In addition, a link of type ac is required between the two set objects. The problem here is, that the meaning of this link between the two sets is not clear. Since there is an intuitive understanding which is neither supported by the formal semantics nor by the compiler semantics (cf. below) we classify this problem as an *unexpected error*.

3.2.2 Intuitive Semantics

In fact, there are at least two interpretations of a link between two set objects which are equally intuitive:

1. the link requires each object in one set to be connected with all objects in the other set, which we call a *total* connection between the sets.
2. the link requires that each member of one set is connected with at least one member of the other set, which we call an *any* connection between the sets.

If the underlying association is bidirectional, the above definitions hold in both directions.

3.2.3 Current Semantics

The semantics definition given in [4] explicitly forbids links between sets. However, Fujaba allows their specification but the current code generation is not able to handle them and aborts throwing an error message.

3.2.4 Possible Solutions

The two intuitive interpretations described in 3.2.2 could be formally defined and supported by the code generation. The desired interpretation of a particular link could be specified by assigning a stereotype - ≪total≫ or ≪any≫ - to it.

3.2.5 Similar Problems

In case of the Story Pattern presented in Figure 3, the two sets are bound independently starting from b and the link between them has to be checked afterwards. However, the same interpretations for links between sets are applicable to the binding of one set starting from another set as well. This would require additional code generation strategies.

When more than one link is specified between the same two sets, the connections between all their members have to be in accordance with all specified links. In order to determine the sets, they have to be filled initially. Then, for each link, all objects which are not connected appropriately are removed from the sets. This process is repeated until both sets are stable which has to occur eventually since no objects are added anymore.

Other situations, such as more than two connected set objects or when single objects can be bound only via a (chain of) set objects, require further investigation.

3.3 Qualified Associations

Qualified associations resemble data structures which use keys to store and retrieve values, e.g. hash tables. Fujaba supports two types of qualified associations. The first type implicitly uses an attribute of the associated class as key. In addition, association roles can be used as keys. The second one allows free keys which must be explicitly specified for the links in the story patterns.

3.3.1 Problem Description

Figure 4 shows an abstract example of three classes A, B, and C. The classes A and C are associated with B via two qualified associations of the first type using the role attributes as keys. If we consider the Story Pattern in the lower part of Figure 4, we have the following problem concerning the link creation: If the link between objects a and b is created first, the key b.c is null but this key is used to add object b to the qualified association. Only after the link between b and c is created, the role attributes in b have the correct value. But then the null-key is already used for the first link. The problem source is the usage of specific access methods for the create link of the qualified association. These access methods implicitly call the read access method on the qualifier which return the wrong or null key.

We classify this problem as *unexpected error* according to the classification in Section 2.

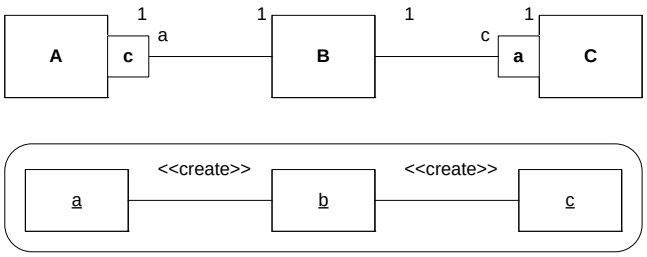

Figure 4: Double Qualified Associations

3.3.2 Intuitive Semantics

As both qualifiers are bound in the Story Pattern, a correct creation of the links is indeed possible. The implicit call of the read access method on the qualifier should be replaced by simply using the known bound qualifier.

3.3.3 Current Semantics

In the current implementation, the wrong code implicitly using the read access methods on the qualifier are used. There is no semantics given in [4]

3.3.4 Possible Solution

The intuitive semantics should be implemented.

3.3.5 Similar Problems

If we only consider a single qualified association with a qualifier based on another association, there is a similar problem. Here, the order of the different link creation actions is important. If all normal links are created before the qualified link, everything is fine. If not, the wrong key (null) would be used. A related problem occurs when links are deleted. For the special case of the `removeYou`-method this behavior has been implemented recently.

4. ISSUES IN STORY DIAGRAMS

4.1 For-Each Activities

For-each-Activities are used to perform activities on all possible bindings. Basically, they describe a loop where *for each* binding the loop body is executed.

4.1.1 Problem Description

The problem is here what happens when the loop body changes the set of possible bindings by creating or deleting elements.

4.1.2 Current Semantics

In [4], a *fresh matches* semantics is employed for for-each loops. The *fresh matches* semantics means that if during execution of the for-each loop new bindings are created, those bindings are additionally matched in the loop condition. Thus, the loop may not terminate which is not feasible.

We classify this semantics as *counterintuitive* as for-each suggests an independent processing (without side effects in between the different body incarnations and no side-effects between the body and the initial set of bindings) according to the classification in Section 2.

Unfortunately, the *fresh matches* semantics is not always correctly implemented in the current version of Fujaba. The behavior is dependent on the employed association implementation. If for example a linked list is used in the implementation of a to-many association, new objects are always placed at the end of the linked list. The linked list is used for iterating over all elements in the to-many association in the binding process. Thus, newly created bindings are matched in the for-each loop. Actually, linked lists are used in combination with hash sets for ordinary to-many associations to guarantee that new elements are found in the bindings.

Sorted and ordered associations are problematic as new links can be arbitrarily inserted into both association's data structures. Thus, a newly created link can potentially be inserted before the current iterator element in the data structure. Consequently, this potentially new binding will not be found in the for-each loop.

In a hard real-time system, the compliance with certain deadlines is of utmost importance. If Story Diagrams are employed in hard real-time environments, a worst case execution time must be known in order to guarantee that the deadlines are satisfied. The fresh matches semantics is thus not suitable in the context of hard real-time systems as new bindings are also processed by the for-each loop and thus the execution time may be arbitrarily prolonged (cf. [3]).

4.1.3 Intuitive Semantics

It is debatable whether the employed *fresh matches* semantics is intuitive. Zündorf decided to employ the fresh matches semantics but said that iterated story patterns should not create new matches.

> "To avoid these kinds of vague semantics one should not write iterated story patterns that create new matches or that destroy more than one match at a time. If this rule is regarded, the semantics becomes clear and simple. Due to our experience, it is easy to respect this rule in practice and iterated story pattern have proven to be very handy in various situations."

Consequently, we propose that new matches should not be regarded in for-each loops.

4.1.4 Possible Solution

Zündorf even mentioned a different semantics, the *pre-select* semantics [4]. Here, all possible bindings are stored before the application of the for-each loop. Newly created matches are then not considered. Unfortunately, this behavior requires additional memory in order to store all matches bindings. We propose to employ the *pre-select* semantics [4] to avoid the inherent problems of the *fresh matches* semantics.

Another solution could be to provide a (probably pessimistic) heuristic which can decide whether the story patterns in the for-each loop can potentially create a new binding. An easy approach is to forbid the creation of the objects and links which can create a new binding. Though, this approach may be overly pessimistic.

4.2 Unsuccessful Creations

Story Patterns allow the specification of creation of links and objects. The creation of a link typically also involves the creation of objects inside the employed data structure and its accompanying algorithms. If Story Patterns are employed in environments with tight resources, this object creation may fail. This problem is often neglected in standard

applications, too. Even there, the memory has a fixed upper bound and, consequently, an object creation may also fail.

4.2.1 Problem Description

If a Story Pattern contains a set of object and link creation operations, any one of these object creations may fail leaving the graph in an undefined state. Unfortunately, the developer has no chance to react to or even know about a failed object creation.

We classify this problem as *incomplete* and *counterintuitive* according to the classification in Section 2. The semantics is counterintuitive as the execution of a Story Pattern is regarded as atomic. The semantics is incomplete as failures are not appropriately considered.

4.2.2 Current Semantics

In the current formal semantics, all object creations and deletions are executed whenever the left hand side matches. Failures are not considered. The code generation resembles this formal semantics. Tough, the code generation implies a certain ordering of the creation and deletion actions. Consequently, if a failure occurs during execution of those actions, all actions prior to the failed one are executed successfully, while the remaining ones are not executed due to the thrown OutOfMemoryError in case of the current Java code generation. As the generated code does not catch the OutOfMemoryError the Story Diagram is left, too.

4.2.3 Possible Solution

We propose creation checks as additional syntactic elements (see Figure 5 from [3]). A creation check is executed after binding the left hand side of a Story Pattern. If the creation check fails, the Story Pattern is left via a creation check failure transition. The developer can then react to a creation check failure e.g. by removing other objects and/or links. Furthermore, we recommend refinements for the creation check failure transitions which allow to distinguish for which object or link the creation check failed.

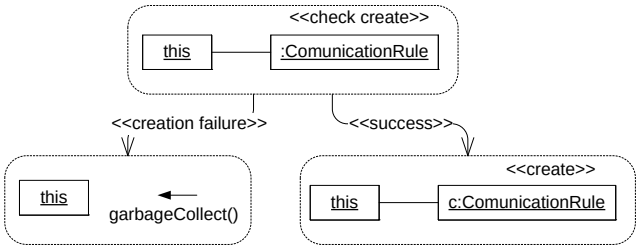

Figure 5: Creation check syntax proposal.

An alternative way of handling the creation problem could be to adopt transaction behavior (Atomicity). Then, the atomicity property guarantees that either all or none of the creation operations are executed. This naturally also holds for deletions of objects or other transformations from the left to the right hand side of the Story Pattern. If the Story Pattern is left via the success transition, a binding has been found *and all* transformations have been successfully applied.

5. CONCLUSIONS

We discussed several semantic issues of Story Diagrams in this paper. While some are easily addressed like the presented problem of qualified associations, others need a more thorough discussion in order to come up with a suitable solution (see for-each loops). For these problems, we gave a starting point for subsequent discussions. Maybe, different solutions are required for the different domains in which Story Diagrams are used. Nevertheless, we believe that all of those problems are solvable. Then, appropriate solutions increase the power of Story Diagrams even further.

6. REFERENCES

[1] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *Proc. of the* 28^{th} *International Conference on Software Engineering (ICSE), Shanghai, China*. ACM Press, 2006.

[2] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In G. Engels and G. Rozenberg, editors, *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764, pages 296–309. Springer Verlag, November 1998.

[3] M. Tichy, H. Giese, and A. Seibel. Story Diagrams in Real-Time Systems. In *Proc. of the 4th International Fujaba Days 2006, Paderborn, Germany*, September 2006. submitted.

[4] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001.