Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 94 (2004) 29–38
www.elsevier.com/locate/entcs

A GXL Schema for Story Diagrams

Chunyan Meng[1] and Kenny Wong[2]

Department of Computing Science
University of Alberta
Edmonton, Canada

Abstract

This paper briefly outlines the process of generating story diagrams, and describes a GXL schema to represent them. These diagrams result from a static, heuristic reverse engineering analysis and combine the data and behavioral information of UML activity and collaboration diagrams. This paper also summarizes potential issues in representing and visualizing story diagrams.

Keywords: GXL, reverse engineering, schema, story diagram

1 Introduction

UML is a popular object-oriented modeling notation used in software development [9]. The story diagram is a combination of the UML activity and object collaboration diagrams. This diagram was originally developed as a graph grammar language for forward engineering [6,7]. When used for reverse engineering, it can be used to better understand the control behavior and object-level dependencies of the software being analyzed [5]. For example, the story diagram can present the notions of links being created and deleted between objects within the control flow of the program [2].

In our project of generating story diagrams from Java source code, we used GXL (Graph eXchange Language [3]) to represent the story diagram graph. GXL supports a general graph model based on typed, attributed, ordered,

[1] Email: chunyan@cs.ualberta.ca
[2] Email: kenw@cs.ualberta.ca

doi:10.1016/j.entcs.2004.01.006

directed, hierarchical graphs [4]. The actual variety of software graph (e.g., abstract syntax tree) may be modeled in UML which itself is a graph and thus representable in GXL. Such schemas are especially important for the interchange of information about software [13].

Section 2 uses an example to illustrate how GXL is used to represent the method-level story diagram and how the information can be extracted from the source code to form the story diagram. Section 3 summarizes some of the issues we encountered in designing a GXL schema. Section 4 concludes the paper.

2 Representation

This section describes how we represent story diagrams with GXL. The analysis and visualization process involves several steps:

(i) parse Java source code,
(ii) produce XML markup on Java code,
(iii) generate story diagram in GXL,
(iv) present GXL story diagram, and
(v) export story diagram as SVG.

The current implementation of this process (See Figure 1) is done in two parts: fact extraction (first three steps) and visualization (last two steps). Step 1 creates an internal Java abstract syntax tree (AST). Step 2 resolves identifiers and produces a exportable XML document that represents the Java AST. Step 3 traverses the XML document, and computes the story diagram as a set of objects which are saved in GXL according to the GXL DTD 1.0 and our schema for story diagrams. Step 4 also uses this set of "GXL" objects but draws the diagram using the Monarch graph library [10]. Step 5 exploits the Monarch library to produce an SVG file, an XML document containing drawing directives.

2.1 Example method-level story diagrams

In the following, we give an example method-level story diagram of a simple single-block method. The Java code of the method is shown in Figure 2.

The GXL file of the story diagram graph is shown in Figure 3. For brevity, some nodes and edges are omitted.

In the method-level story diagram, the control flow is represented as a flowchart-like activity diagram, while the behavior of each block in the method is shown as a collaboration diagram. In the example, various statements lead

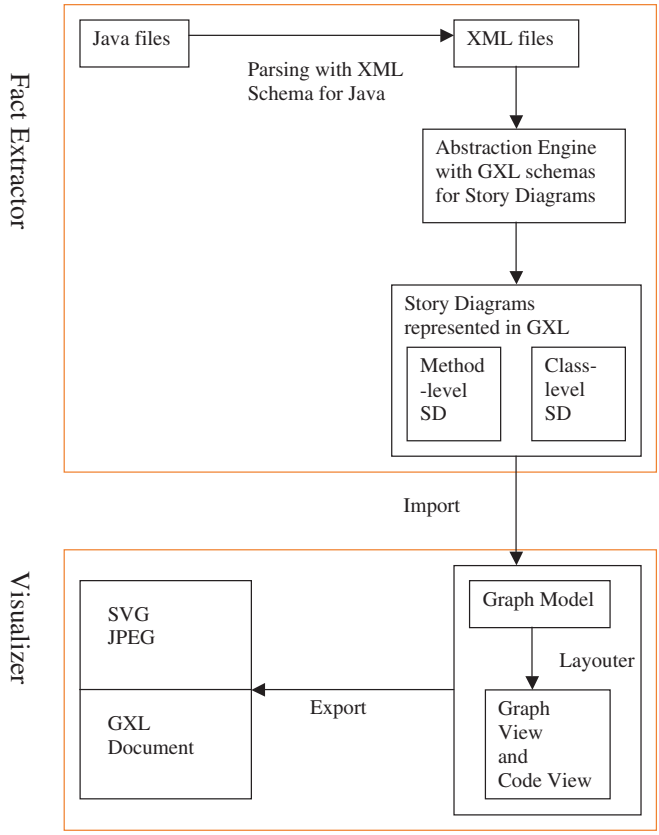

Fig. 1. Architecture of the visualization tool

```
public void createCourseFor(String instructor){

   Professor p = getProfessor(instructor);
   Course c1 = new Course();
   p.setCourse(c1);

   // for some reason, change the course:
   Course c2 = new Course();
   p.setCourse(c2);
   p.deleteCourse(c1);
}
```

Fig. 2. Java sample method

to dependencies being formed between objects. A red color is used to show if an object dependency gets broken in the block. An example story diagram is presented in figure 4. On the story diagram, we use different colors to represent different object types and edge types as illustrated in 5. For example, a yellow edge means a method call invoked upon an object.

```
<graph id='g368'  type='method'>
  <attr name="name" value="createCourseFor">
  </attr>
  <attr name="formal-argument"
    value="instructor"/>
  <node id="g258" type="start">
    <attr name="statement" kind=""
     value="start"/>
  </node>
  <node id="g239" type="activity">
    <attr name="code" kind=""
     value="..."/>
    <attr name="instructor" kind=""
     value="activity"/>
    <graph id='body-g240' type='default'>
    ...
    </graph>
  </node>
  ...
  <edge id="829" type="new-declared"
   from="g239"  to="g259">
    <attr name="label" kind="" value=" "/>
  </edge>
</graph>
```

Fig. 3. Portion of the GXL story representation

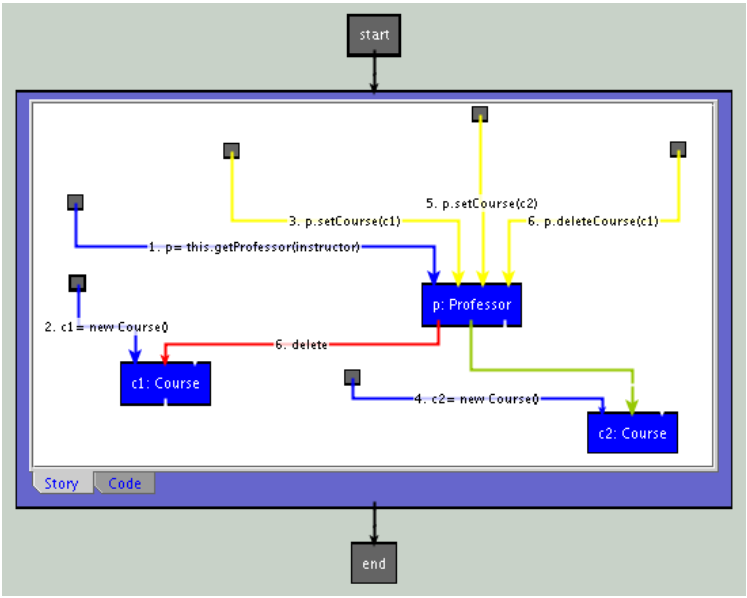

Fig. 4. Presented method-level story diagram

The process of generating story diagrams in GXL involves three major steps:

(i) Group operations to form an activity: The object collaborations existing in a basic block are treated as an activity, which is represented by a node

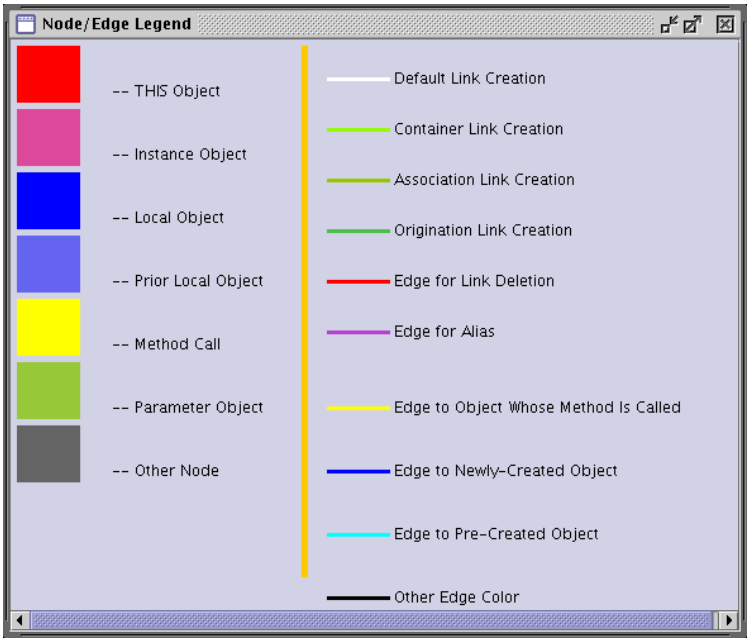

Fig. 5. Color scheme used in the story diagram

with a nested graph for the method invocations upon objects and dependencies formed or destroyed between the objects. Between the activity nodes are control flow relationships representing conditional or sequential execution. To manage the visual complexity if there are too many object collaborations in a block, the block may be further decomposed to form sub-activities to be presented in separate sequentially linked nodes.

(ii) Extract object collaborations: In the activity nodes (depicted as boxes), the specific actions are illustrated by object collaboration diagrams. The method calls or statements are marked by numbers according to the order in which they are invoked. For each method, start and end nodes depict the entry and exit points of the control flow.

(iii) Extract link creation and deletion: Knowing the data dependencies or links formed or destroyed between objects can help programmers understand the detailed object interactions. Heuristics are used to extract these links statically. A green edge depicts a link that has been formed, and a red edge depicts one that is now broken. The following sub-section describes the current heuristics.

2.2 Heuristics for link creation and deletion

Link creation and deletion are distinct features that separate story diagrams from normal UML collaboration diagrams. Since the relationships between the objects are not always direct, and might occur through library routines, some heuristic rules are applied. The main approach considers the keywords of the method name at the site of a method call upon an object. If the call contains arguments, the referred objects are checked against the following rules to see if a link potentially exists between it and the receiving object of the method call.

(i) Container relationships: one object is added into a container object. This kind of dependency can be distinguished from keywords like `add`, `put`, `insert`, etc., and the data type of popular container names. The link edge type is "contain".

(ii) Association relationships: one object has some relation with the other one. For instance, a Course object is set to a Teacher object (who teaches the course). An Office object can be set to a particular level of a Building object. This kind of dependency can be distinguished from keywords like `setAt`, `set`, and `set*` where * may represent the data type of the argument of the method call. The link edge type is "setRelation".

(iii) Origination relationships: one object is returned through another object. This dependency can be discovered from keywords like `get*`, `generate*`, and `produce*` where * may represent a defined object type returned by the method call. The link edge type is "getFrom".

Link deletion rules are similar (with keywords such as `remove`, `delete`, etc.), with the added case of `set*(null)`. For example, if `anObj` is discovered to be linked to some other object, then `anObj.setComponent(null)` will break the link between the two objects.

2.3 GXL schema for the method-level story diagram

To have the story diagram represented in a consistent way, we defined a GXL schema. See Figure 6. The GXL schema is represented in a UML class diagram. This schema describes the kinds of nodes and edges that comprise the elements of a story diagram. Since story diagrams are a result of static analysis, some heuristics are used to infer the instances of links being created or deleted between objects.

Based on our schema, the top view of the story diagram is like a control flow diagram. However, the story diagram graph is a nested graph and not flat, where lexical blocks may contain further lexical blocks, with basic blocks

at the lowest level.

Currently, we are working towards a consistent schema for the representation and visualization of story diagrams at higher levels of abstraction. Representation aspects include appropriate text labels with source code fragments. Visual aspects include the shapes and colors of nodes and edges. At present, this information is stored as attributes of the corresponding nodes or edges.

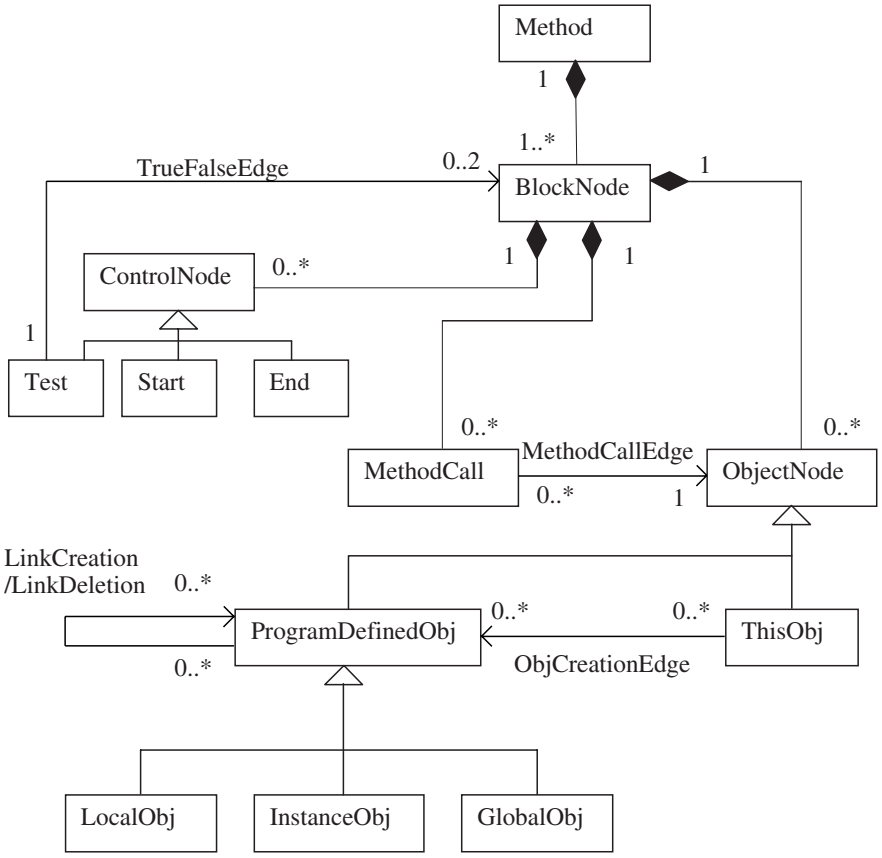

Fig. 6. Method-level story diagram schema

3 Issues and Future Work

This section describes a few of the implementation issues we encountered when using GXL schema to present story diagrams.

3.1 Linking diagrams to code

In the story diagrams, we need to express fragments of the original source code as labels in the diagrams. Thus, there is a need to link the story diagram graph to the textual source code from which it is derived.

We have two XML-based components: one is for marking up the Java source code and the other is GXL for expressing story diagrams. The relationship between these source-oriented and graph-oriented representations is interesting, since the user may want to switch views between the source code and the story diagram. For example, systems like SHriMP can support a similar capability [8]. For our purpose, we need some consistent color scheme to relate elements visually in the source code view to the story diagram view. Also, to color the code fragments in the story diagram properly, we need to carry along the XML markups of Java as values attached to certain GXL nodes. Figure 7 is an example code view of the story diagram in Figure 4.

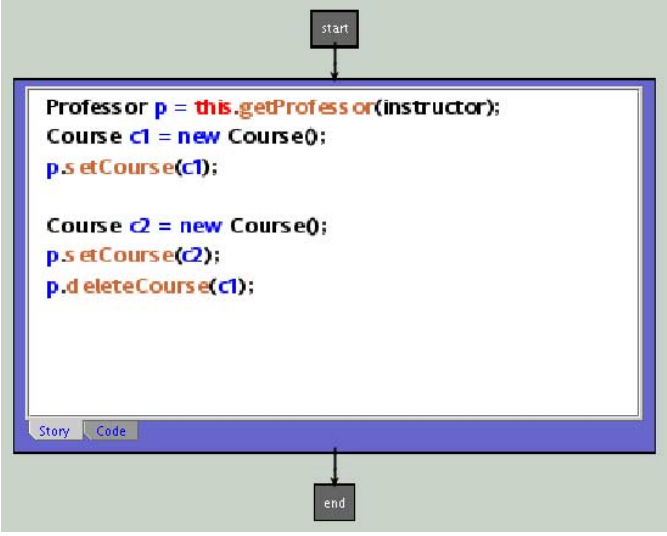

Fig. 7. The code view of the method-level story diagram

The XML markups can be carried with the GXL objects for even the higher-level story diagrams, allowing ways to bring the user from the high-level views down to the source code view. A cross-reference mechanism like SHriMP [11] that considers schemas defined for different levels of abstractions can be applied to make the visualization tool more useful.

3.2 Higher-level story diagrams

The class-level story diagram is extracted from details in the XML Java AST document, not from the abstracted method-level story diagram. We are exploring ways to extend our schema in a "modular" way to incorporate analogous notions of the story diagram at higher and higher levels of abstraction or granularity. By defining a modular and consistent schema, we intend to be able to selectively exchange story diagram information at different abstraction levels. The multi-layered approach proposed by Cox and Clarke [1] could be considered in integrating the conceptual schemas for the story diagrams at different abstraction levels. The challenge is how to implement the interface between different abstraction levels and to ease the exchange of story diagram information.

3.3 Layout of compound graphs

In generating control flow diagrams in GXL format, we encountered the need for compound subgraphs where edges need to span the boundaries of two subgraphs at different levels (i.e., are not contained wholly within a nested subgraph). According to [12], we should put these edges in the least-common-ancestor graph node. These kinds of dependencies create graph-layout challenges.

3.4 Evaluation

Also, more work is needed to validate our heuristics, assess the scalability of the visualization, and evaluate the usability of the implementation.

4 Conclusions

GXL can be used to represent method-level story diagrams using the schema outlined in this paper. XML markup over Java code and the GXL representation can be used at the same time to present the code view and graph view of the story diagram. Extending the schema for higher-level story diagrams is ongoing work. Also, difficulties arise in the graph layout of complicated compound graphs such as story diagrams.

References

[1] Cox, A. and C. Clarke, *Multi-layered data-modeling*, in: *Proceedings of the 1st International Workshop on Meta-Models and Schemas for Reverse Engineering—ateM 2003 (Victoria, BC, Canada)*, 2003.

[2] Fischer, T., J. Niere, L. Torunski and A. Zündorf, *Story diagrams: A new graph grammar language based in the unified modeling language*, in: *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation—TAGT '98 (Paderborn, Germany)*, 1999, pp. 112–121.

[3] GUPRO, *GXL (1.0) Document Type Definition (Dagstuhl Edition)* (2002).
URL http://www.gupro.de/GXL/dtd/print.html

[4] Holt, R., A. Winter and A. Schürr, *GXL: Toward a standard exchange format*, in: *Proceedings of the 7th Working Conference on Reverse Engineering—WCRE 2000 (Brisbane, Australia)*, 2000, pp. 162–171.

[5] Jahnke, J., H. Müller, N. Mansurov and K. Wong, *Fused data-centric visualizations for software evolution environments*, in: *Proceedings of the 10th International Workshop on Program Comprehension—IWPC 2002 (Paris, France)* (2002), pp. 187–196.

[6] Jahnke, J. and A. Zündorf, *Specification and implementation of a distributed planning and information system for courses based on story driven modelling*, in: *International Workshop on Software Specification and Design—IWSSD-9 (Ise-Shima (Isobe), Japan)*, 1998, pp. 78–86.

[7] Jahnke, J. and A. Zündorf, *Applying graph transformations to database re-engineering*, in: *Handbook of Graph Grammars and Computing by Graph Transformation* (1999), pp. 267–286.

[8] Michaud, J., M.-A. Storey and H. Müller, *Integrating information sources for visualizing Java programs*, in: *Proceedings of the International Conference on Software Maintenance—ICSM 2001 (Florence, Italy)*, 2001, pp. 250–260.
URL http://citeseer.nj.nec.com/michaud01integrating.html

[9] Rumbaugh, J., I. Jacobson and G. Booch., "The Unified Modeling Language Reference Manual," Addison-Wesley, 1999.

[10] Singleton Labs, *Graph visualization and diagrams: MonarchGraph* (2003).
URL http://www.singleton-labs.com/mgraph.html

[11] Storey, M.-A., K. Wong, F. Fracchia and H. A. Müller, *On integrating visualization techniques for effective software exploration*, in: *Proceedings of IEEE Symposium on Information Visualization—InfoVis '97 (Phoenix, Arizona)*, 1997, pp. 38–45.

[12] Winter, A., *Exchanging graphs with GXL*, in: *Proceedings of the 9th International Symposium on Graph Drawing—GD 2001 (Vienna, Austria)*, 2001, pp. 485–500.

[13] Winter, A., B. Kullbach and V. Riediger, *An overview of the GXL graph exchange language*, in: *Software Visualization—Dagstuhl Seminar (Dagstuhl, Germany)*, 2001, pp. 324—336.