The GROOVE Simulator: A Tool for State Space Generation

Arend Rensink

University of Twente
P.O.Box 217, 7500 AE Enschede, The Netherlands
rensink@cs.utwente.nl

1 Introduction

The tool described here is the first part of a tool set called GROOVE (*GRaph-based Object-Oriented VErification*) for software model checking of object-oriented systems. The special feature of GROOVE, which sets it apart from other model checking approaches, is that it is based on graph transformations. It uses graphs to represent state snapshots; transitions arise from the application of graph production rules. This yields so-called *Graph Transition Systems* (GTS's) as computational models.

The simulator does a small part of the job of a model checker: it attempts to generate the full state space of a given graph grammar. This entails recursively computing and applying all enabled graph production rules at each state. Each newly generated state is compared to all known states up to isomorphism; matching states are merged, in the way proposed in [4]. No provisions are currently made for detecting or modelling infinite state spaces. Alternatively, one may choose to simulate productions manually.

This paper describes two examples: Sect. 2 shows the behaviour of a circular buffer and Sect. 3 the concurrent invocation of a list append method. In both cases the behaviour is defined by a graph grammar, but to provide some intuition, Fig. 1 approximately describes the behaviour, using Java code. We conclude in Sect. 4 with a summary of tool design, implementation and planned future extensions.

2 Circular Buffer Operations

We assume the principles of circular buffers to be known. Their representation as graphs is relatively straightforward (see also Fig. 1). The buffer has a set of cells connected by `next`-edges. One of the cells is designated `first` and one `last`. Insertion will occur at `last` (provided this cell is empty) and retrieval at `first` (provided this is filled). A value contained in a cell is modelled by a `val`-labelled edge to an unlabelled node. The cell is empty if and only if there is no outgoing `val`-edge. (In the Java code of Fig. 1 this corresponds to a `null` value of the `val` attribute.)

J.L. Pfaltz, M. Nagl, and B. Böhlen (Eds.): AGTIVE 2003, LNCS 3062, pp. 479–485, 2004.

```
public class Buffer {
    private class Cell {
        Cell next; Object val;
    }
    private Cell first, last;

    // Precondition: last.val == null
    // Executed atomically
    public void put(Object val) {
        last.val = val;
        last = last.next;
    }

    // Precondition: first.val != null
    // Executed atomically
    public Object get() {
        Object result = first.val;
        first.val = null;
        first = first.next;
        return result;
    }
}
```

```
class Node {
    private Node next;
    private int val;

    public void append(int x) {
        if (this.val == x) {
            // Rule "stop"
            return;
        } else if (this.next == null) {
            // Rule "append"
            Node aux = new Node();
            aux.val = x;
            this.next = aux;
            return;
        } else {
            // Rule "next"
            this.next.append(x);
            // Rule "return"
            return;
        }
    }
}
```

Fig. 1. Approximate Java descriptions of the examples

Fig. 2 shows the simulator tool after loading the relevant graph grammar. The GUI of the simulator has two panels: a directory of the available rules with their matches in the current graph, and the current graph itself — in this case the initial graph, modelling a three-cell empty circular buffer. The latter panel can also display the currently selected rule and the resulting GTS (insofar generated), instead of the current graph. The example grammar has two rules: `get` for the retrieval of an element from the buffer and `put` for insertion. As usual, each rule prescribes when it applies to a given graph and what the effect of its application is. There are four types of nodes and edges:

- Thin black solid nodes and edges, which we call *readers*: they are required to be in a graph in order for the rule to apply, and are unaffected by rule application;
- Thin blue double-bordered nodes and dashed edges, which we call *erasers*: they are required in order for the rule to apply, and are deleted by rule application;
- Fat green solid nodes and edges, which we call *creators*: they are *not* required to be in the graph, and are created by rule application.
- Fat red double-bordered nodes and dashed edges, which we call *embargoes*: they are *forbidden* to occur in a graph in order for the rule to apply.

More precisely, a rule application is based on a *matching*, which is a mapping of the readers and erasers of the rule to corresponding elements of the graph that cannot be extended with any of the embargoes. For instance, as Fig. 2 shows,

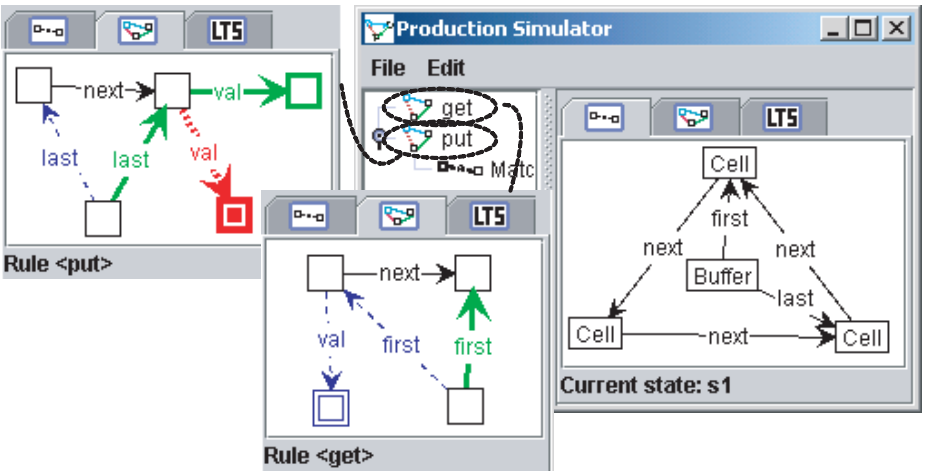

Fig. 2. Rules and initial graph of the circular buffer example

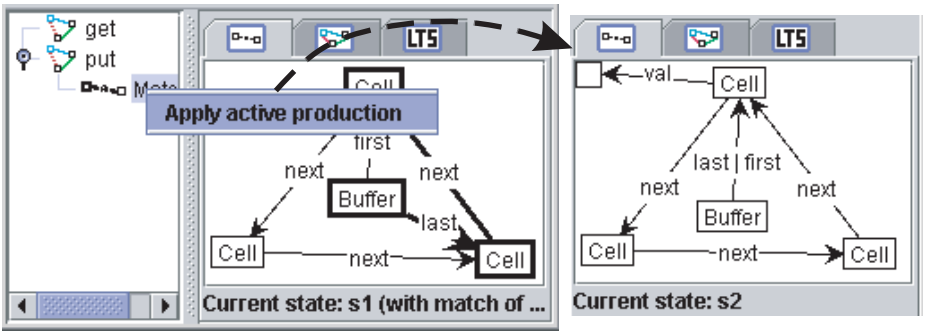

Fig. 3. Application of the `put` rule to the initial graph

`put` has a single match in the initial graph whereas `get` has none. The application deletes the elements matching the erasers and adds elements matching the creators. Fig. 3 shows how one may select and apply a matching in the simulator (the graph is fat where the matching applies). Besides "walking through" the rule applications in this fashion, the simulator tool also can (attempt to) recursively compute *all* possible applications. This gives rise to a GTS, with the set of all graphs generated by the grammar as states, connected by rule applications. Although the GTS is generally infinite, there are many cases in which it is not — in fact, for grammars that model computing systems, an infinite state space is arguably an error indication. For the circular buffer, the state space is quite small, consisting a single state for each possible number of filled cells: see Fig. 4.

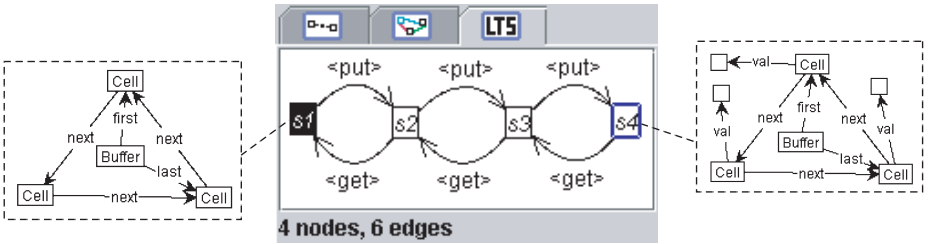

Fig. 4. Graph transition system of the circular buffer example

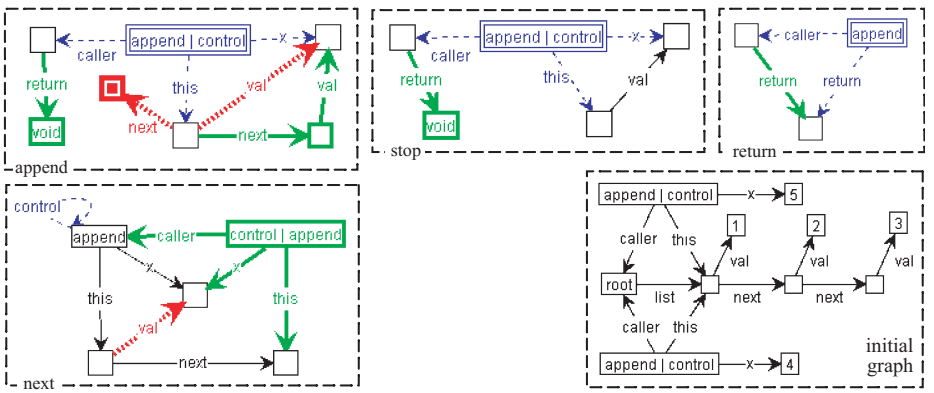

Fig. 5. Rules and initial graph of the concurrent append example

3 Concurrent Appending

The second example is a list append method. In this case we do not assume the method to be atomic; instead we model it as a recursive invocation. The example shows many features typical for the dynamics of object-oriented programs. Running methods are modelled by nodes, with local variables as outgoing edges, including a `this`-labelled edge pointing to the object executing the method. Each (recursive) method invocation results in a fresh node, with a `caller` edge to the invoking method. Upon return, the method node is deleted, while creating a `return` edge from its caller to a return value — in this example always `void`. It follows that the traditional call stack is replaced by a chain of method nodes. The top of the stack is identified by a `control` label or an outgoing `return` edge. The particular method in this example only appends a value that is not already in the list, as suggested by the code in Fig. 1. The corresponding graph grammar is shown in Fig. 5. The initial graph contains two concurrently enabled invocations; we expect the simulator to tell us whether these invocations may interfere. Due to the ensuing race condition, the system has two legal outcomes: either the value 4 is appended before 5, or vice versa. For instance, Fig. 6 shows

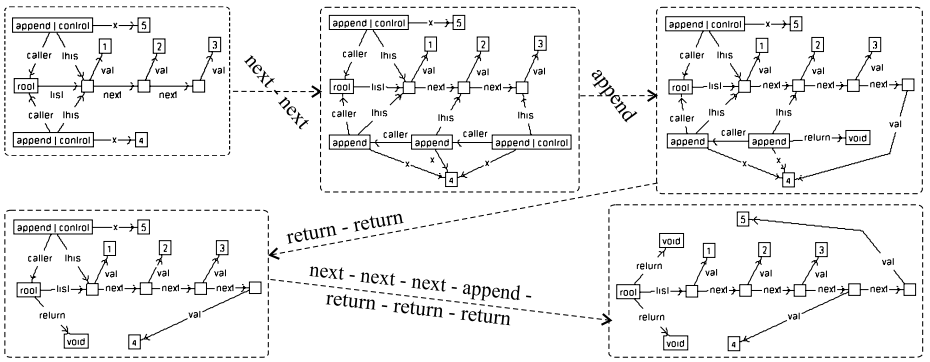

Fig. 6. Trace of graph transformations for the `append` function

(part of) the sequence of states in which the "bottom" `append` proceeds first, so that 4 "wins" from 5.

The full GTS, comprising 57 states and 92 transitions, is shown in Fig. 7. The race condition can be readily recognised, as can the fact that the final states are precisely the two legal ones. We conclude that the `append` method as modelled by this graph grammar is non-interfering.

4 Design and Implementation

We finish by discussing a number of issues in the design and implementation of the GROOVE tool set in general, and the simulator in particular.

Theoretical Background. There is a long history of research in graph transformation; see [5] for an overview. GROOVE follows the algebraic approach; we hope to reap the benefit of the resulting algebraic properties in future extensions (see below). GROOVE uses non-attributed, edge-labelled graphs without parallel edges (the node labels shown in the examples above are actually labels of self-edges) and implements the single-pushout approach with negative application conditions (see [3, 2]); however, the design of the tool is modular with respect to this choice, and support for the double-pushout approach can be added with a single additional class.

Design. The tool is designed for extensibility. The internal and visual representation of graphs are completely separated, and interfaces are heavily used, for instance to abstract from graph and morphism implementations. The most performance-critical parts of the simulator are: finding rule matchings, and checking graph isomorphism. The first problem is, in general, NP-complete; the second is in NP (its precise complexity is unknown). Fortunately, the graphs we

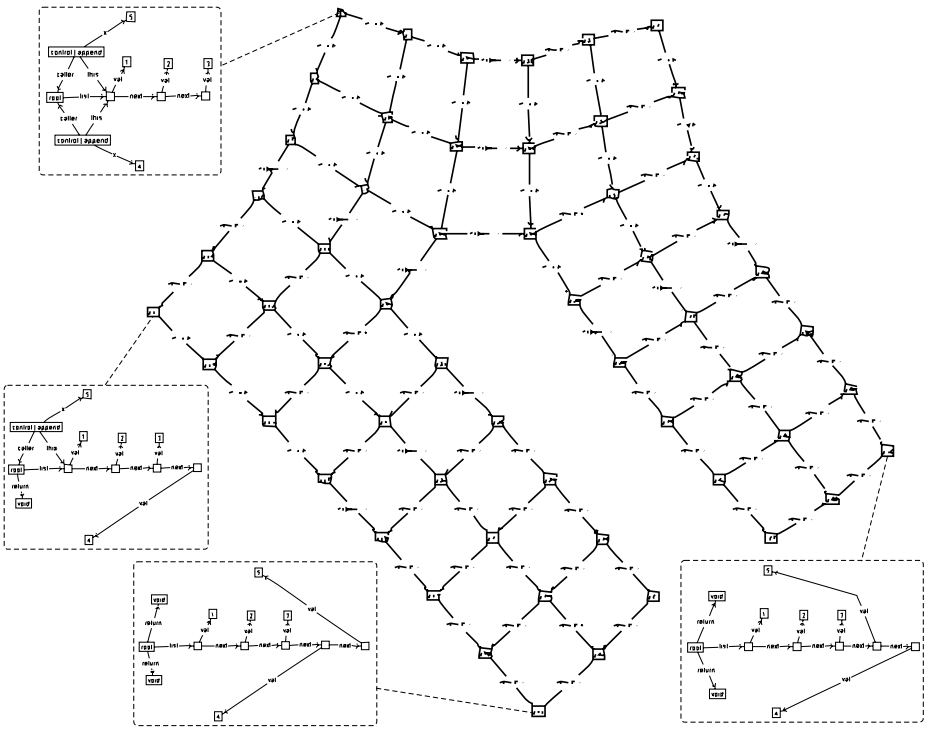

Fig. 7. Graph transition system generated from the `append` function

are dealing here tend to be "almost deterministic" (we will not make this precise) so that the complexity is manageable. The GROOVE tool uses strategies to ensure modularity for the algorithms used.

Implementation. The tool is implemented in Java, and developed under Eclipse. It currently consists of (approximately) 70 classes in 8 packages, comprising 25,000 lines of code. The implementation makes use of several existing open-source components: `jgraph` for graph visualisation, `xerces` and `castor` for handling XML, and `junit` for unit testing. The implementation is a research prototype; we are yet far from being able to tackle realistic problems. Currently the largest example we have simulated is approximately 20,000 states; this takes in the order of half an hour.

Interchange Formats. The modularity of GROOVE also extends to the serialisation and storage of graphs and graph grammars. Currently the tool uses GXL (see [8]), but in an *ad hoc* fashion: production rules are first encoded as graphs by tagging the erasers, creators and embargoes, and then saved individu-

ally; thus, a grammar is stored as a set of files. In the future we plan to migrate to GTXL (see [7]) as primary format.

Extensions. The simulator can be improved in many ways. Some extensions planned for the (near) future are: support for the double-pushout approach and graph types, partial order reduction using production rule independence; and especially, *shape graph*-like abstractions (cf. [6]), for the (approximate) representation of infinite state spaces. Furthermore, we plan to extend the GROOVE tool set beyond the simulator functionality. This includes a front-end to compile code to graph grammars (a prototype of a Java byte code translator has been developed) and a back-end to model check the resulting GTS's. In [1] we have taken a first step toward developing a logic to reason about the dynamic aspects of GTS's. See [4] for a global overview of the programme.

The GROOVE simulator tool and some sample graph grammars (the above examples among them) can be downloaded from:
`http://www.cs.utwente.nl/~groove`.

References

[1] Dino Distefano, Arend Rensink, and Joost-Pieter Katoen. Model checking birth and death. In R.A. Baeza-Yates, U. Montanari, and N. Santoro, editors, *Foundations of Information Technology in the Era of Network and Mobile Computing*, pages 435–447. Kluwer, 2002. 485

[2] Annegret Habel, Reiko Heckel, and Gebriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996. 483

[3] M. Löwe. Algebraic approach to single-pushout graph transformation. *TCS*, 109(1–2):181–224, 1993. 483

[4] Arend Rensink. Towards model checking graph grammars. In Leuschel, Gruner, and Lo Presti, editors, *Proceedings of the* 3rd *Workshop on Automated Verification of Critical Systems*, Technical Report DSSE–TR–2003–2, pages 150–160. University of Southampton, 2003. 479, 485

[5] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume I: Foundations. World Scientific, 1997. 483

[6] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM ToPLaS*, 20(1):1–50, January 1998. 485

[7] Gabriele Taentzer. Towards common exchange formats for graphs and graph transformation systems. In Ehrig, Ermel, and Padberg, editors, *UNIGRA*, volume 44 of *ENTCS*. Elsevier, 2001. 485

[8] Andreas Winter, Bernt Kullbach, and Volker Riediger. An overview of the GXL graph exchange language. In Diehl, editor, *Software Visualization*, volume 2269 of *LNCS*, pages 324–336. Springer, 2002. 484