State space of the append rule system

Arend Rensink

November 2010

Abstract

In this note we analyse the state space of the append rule system, and we give a combinatorial formula for the state space size for a set of start states.

1 Introduction

One of the performance tests used in GROOVE (and reported in, e.g., [2, 1]) is the append rule system. This models a concurrently invoked method that appends an element to a list by recursively calling itself on each successive element until it reaches the end of the list, at which point it extends the list and returns. The system consists of the four rules displayed in 1.

It should be noted that this rule system uses neither typing nor flags. The meaning of the rules is as follows:

next recursively invokes append on the next cell of a list. The value to be appended is pointed to by the x-edge. Due to the negative application condition, the rule is only applicable if the next list cell does not contain the value to be appended.

append creates a new list cell and gives it the indended value (through a val-edge). Control returns to the caller by giving it a reaturn value (the node labelled void).

stop stops the traversal of the list, in case the next list element contains the value that is to be appended. Control returns to the caller, in the same way as for the append rule.

return ends a method that has a return value (which must have been inserted by a recursively called sub-method), by passing on the return value to the caller.

Depending on the number of concurrent invocations, the state space can quickly grow large: the size is exponential in the number of append-invocations, where the base is (essentially) the length of the list. The quickly growing size is why we have used thi rule system as a test case for GROOVE, both for correctness and for performance.

However, in order to legitimately use the rule system for testing the correctness of GROOVE, we need an independent means of determining the correct outcome. For that purpose, in this note we derive formulas for the state and transition counts of the state spaces generated for a certain class of state graphs. Typicalstart graphs are shown in Figure 2.

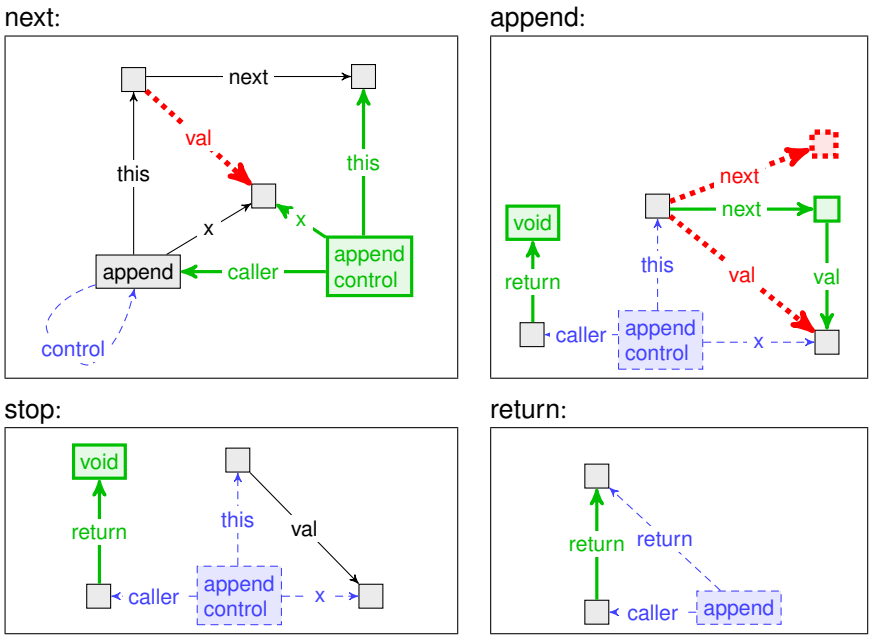


Figure 1: Rules of the append transformation system

2 Analysis of the state space

We investigate three different scenarios for the append system:

No sharing. This refers to a start graph with (initially) two concurrent append invocations, where the elements being appended are not already in the list and are not the same for both methods. An example (with list size 5) is the first graph in Figure 2.

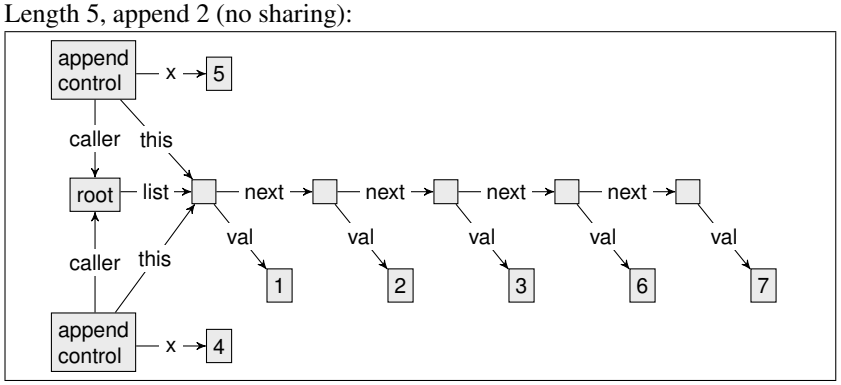
List sharing. This refers to a start graph where there is an append invocation attempting to append an element that is already in the list. This will not succeed: the method wil be aborted by the stop-rule, and have no effect on the list. An example is the second graph in Figure 2.

Call sharing. This refers to a start graph where two append invocations attempt to append the same element (which is not already in the list). In this case the first invocation to reach the end of the list will succeed in appending the element; the second invocation will then abort (with the stop-rule) because the element is already present. An example is the third graph in Figure 2.

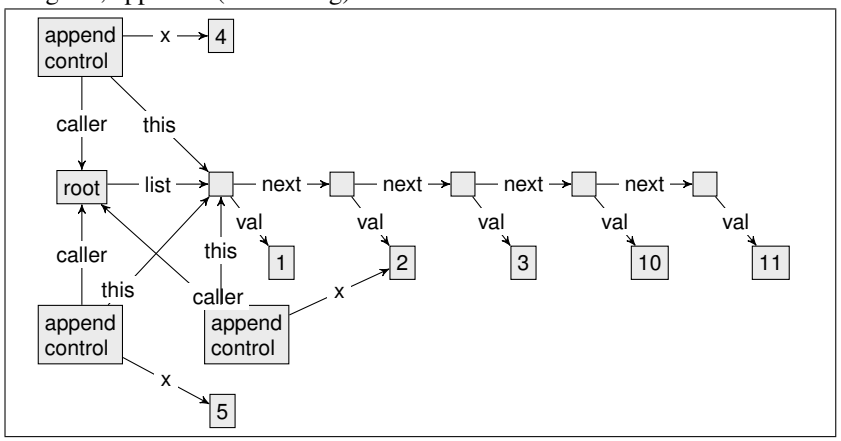
2.1 No sharing

In the scenario without sharing, we invoke 2 concurrent instances of append on a list of varying length; the elements to be appended are neither in the list nor shared among the two invocations. Figure 3 shows the state space for a list of length 5; in fact, the start graph is the top graph in Figure 2. The initial state is on top; there are two final states, namely the two bottom corners. Clearly, the state space is nondeterministic:

Length 5, append 2 (no sharing):



Length 5, append 3 (list sharing):



Length 5, append 3 (call sharing):

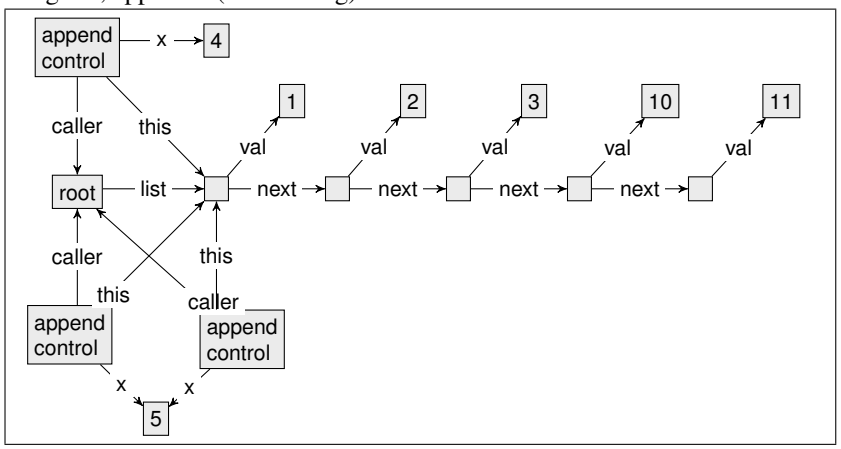


Figure 2: Typical start graphs for the append rule system

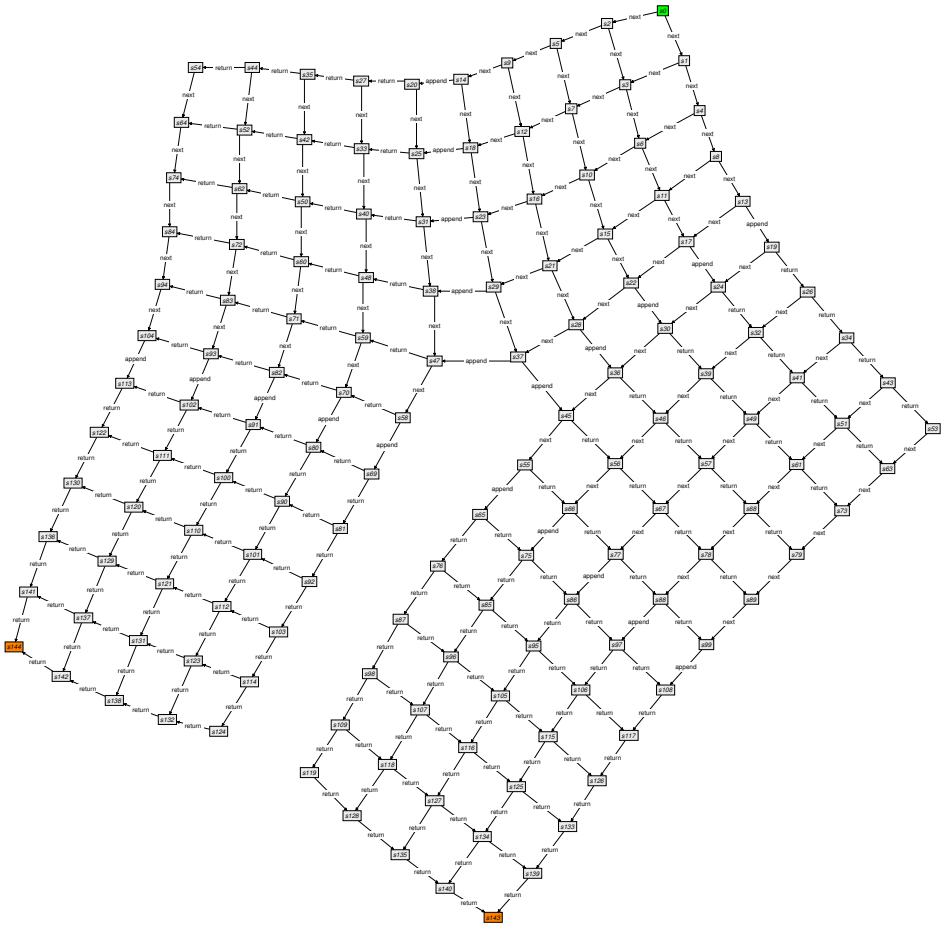


Figure 3: State space of 2 concurrent appends on a list of length 5

there is a choice in which of the append methods "wins" in making it to the end of the list and appending its element first. Apart from this choice, all transformations are (locally) confluent.

$$s_1 : x \mapsto 5x^2 + 4x$$
$$t_1 : x \mapsto 10x^2 + 2x - 4$$

$$s_2 : x \mapsto 2i\, s_1(x)$$
$$t_2 : x \mapsto 2i\, t_1(x) + (2i-1)s_1(x)$$

i is the position of the shared list node

$$s_3 : x \mapsto \frac{11x^3 + 31x^2 + 20x}{2}$$
$$t_3 : x \mapsto 3s_3(x) - \frac{19x^2 + 49x + 22}{2}$$

3 Conclusion

The analysis reported in this note was inspired by the observation that the current version of GROOVE reports different values for the transition count of the append state spaces than the one listed in [2]. Clearly one of the versions has to be erroneous. Our analysis shows that it was the past version, and not the current one, that is wrong: in particular, the transition counts (but not the state counts) in Table 2 of that paper differ from Table **??** (they are too high), whereas the current GROOVE version generates precisely the numbers in the table. We do not have a hypothesis regarding the source of the error.

References

[1] A. Rensink. Isomorphism checking in GROOVE. In A. Zündorf and D. Varró, eds., *Graph-Based Tools (GraBaTs)*, vol. 1 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, September 2007.

[2] A. Rensink, Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In H. Ehrig, G. Engels, F. Parise-Presicce, and G. Rozenberg, eds., *International Conference on Graph Transformations (ICGT)*, vol. 3256 of *Lecture Notes in Computer Science*, pp. 226–241. Springer Verlag, 2004.