# A Case Study: Verifying a Mutual Exclusion Protocol with Process Creation using Graph Transformation Systems*

Fernando Luís Dotti[1], Barbara König[2], Osmar Marchi dos Santos[3], and Leila Ribeiro[4]

[1] Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil
[2] Institut für Formale Methoden der Informatik, Universität Stuttgart, Germany
[3] Department of Computer Science, University of York, UK
[4] Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil

**Abstract.** We verify a mutual exclusion protocol with dynamic process creation based on token passing. The protocol is specified using object-based graph grammars. We introduce the protocol and show how the mutual exclusion property and other properties can be verified using the tool AUGUR, a verification tool for graph transformation systems based on an approximated unfolding technique.

## 1 Introduction

Mobile and dynamic systems are becoming more and more common and their influence on our daily life is increasing dramatically. These systems are typically characterized by an infinite state space, dynamic object creation and deletion, mobility and a variable topology. Modelling and analyzing such systems has become a challenging task in the software development process. While there are several promising approaches for modelling dynamic systems, such as Petri nets [Rei80], graph transformation systems [Roz97] or process calculi [SW01], the state of the art in analysis and verification is much less advanced. But analysis methods are crucial in order to ensure safety and reliability of software systems.

In this paper we are working with an analysis technique for graph transformation systems, that (over-)approximates graph transformation systems by Petri nets, allowing us to automatically prove system properties [BCK01]. We use a software tool called AUGUR[5] which is based on this technique.

We conduct a case study, analyzing a mutual exclusion protocol, which has the added feature of dynamic process creation, i.e., the set of processes is not fixed a priori. The example has originally been specified in the framework of object-based graph grammars [DR00], based on message passing. We are able to show that mutual exclusion is indeed ensured and that messages arrive at their correct destinations.

* Supported by the FAPERGS/BMBF project DACHIA and DFG project SANDS.
[5] See `http://www.fmi.uni-stuttgart.de/szs/tools/augur/`

## 2 A Mutual Exclusion Protocol

The following mutual exclusion protocol is specified in the framework of object-based graph grammars [DR00], a specific variant of graph transformation systems. We will not explain the framework in any great detail, but we will give a brief overview: graph transformation rules consist—like all other rewriting rules—of a left-hand and a right-hand side. In object-based graph grammars the left-hand side contains an object and a message sent to this object, the rule describes the reaction of the object upon the reception of the message. This reaction might be the creation of further messages and objects. The left-hand side also contains neighbouring objects in order to be able to correctly describe the embedding of the right-hand side into the graph.

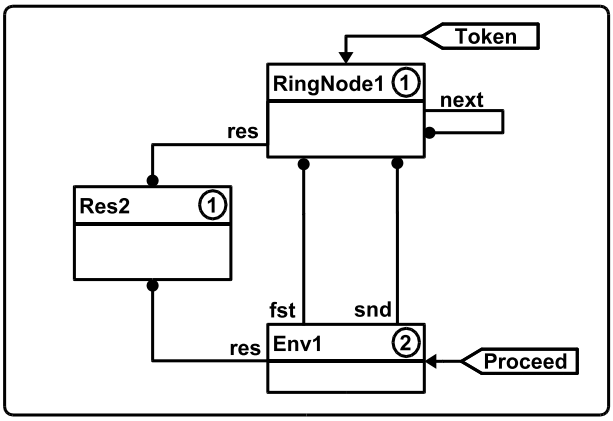


**Fig. 1.** Initial graph.

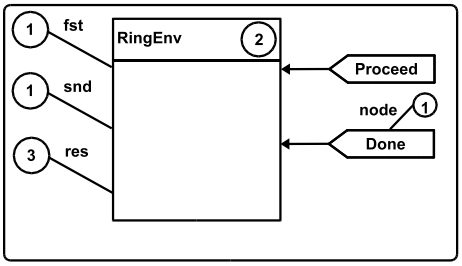


**Fig. 2.** *RingEnv* type graph.

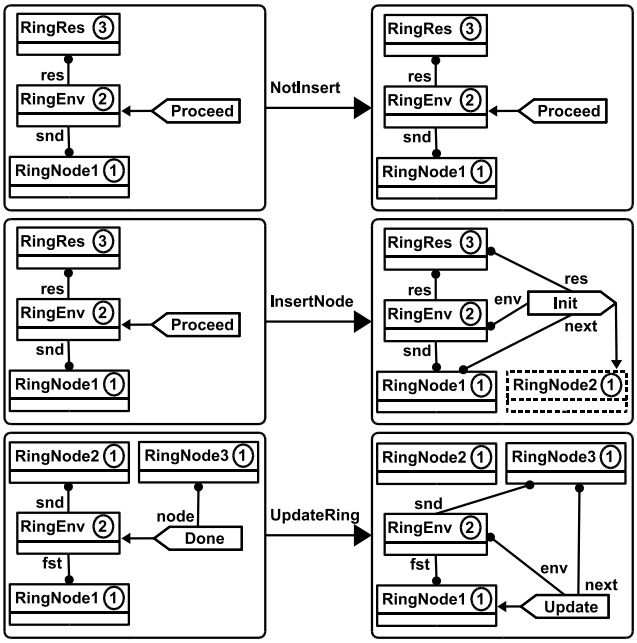


**Fig. 3.** Rules for *RingEnv* entity.

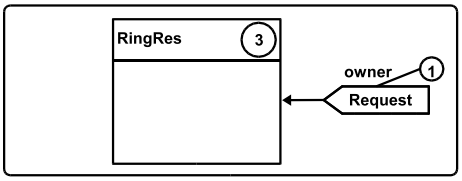


**Fig. 4.** *RingRes* type graph.

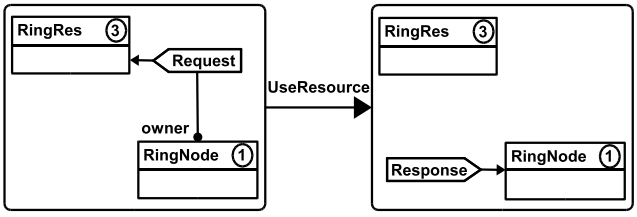


**Fig. 5.** Rules for *RingRes* entity.

Figures 1–7 describe the protocol from the point of view of the three entities *RingEnv* (the environment), *RingNode* and *RingRes* (the resource). Figure 1

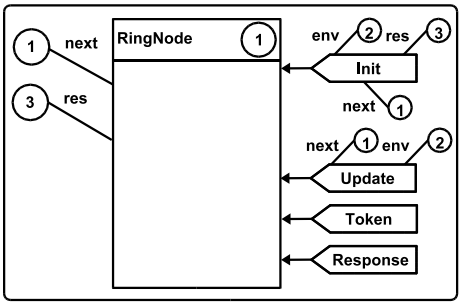


**Fig. 6.** *RingNode* type graph.

depicts the initial graph which consists of a ring consisting of one node only—hence the next-connection originating from this node is pointing back to itself. The environment has two connections to the first and second node of a ring, which in this case coincide, and furthermore there exists a resource known to the node and the environment.

While the system runs new nodes will be created and inserted into the ring by the ring environment. Furthermore a token is passed around and a node which is in possession of the token can access the shared resource.

Hence if the ring environment receives the message *Proceed*, it might create a new *RingNode* and send to it a message *Init* in order to initialize it. Upon reception of *Init*, the *RingNode* creates connections to its neighbours and to the environment, to which it sends message *Done*. The environment receives *Done* and sends *Update* to the first neighbour in order to complete the insertion.

The rest of the protocol is concerned with assigning the resource to the ring nodes. A ring node in possession of message *Token* may pass the token on or send a message *Request* to the resource. The resource acknowledges with *Response*.

The rules are grouped according to the entitities they belong to. Furthermore for every entity there exists a type graph—a subgraph of the type graph of the entire system—specifying arriving messages and possible connections.

A central question in the analysis of this protocol is whether mutual exclusion is guaranteed.

## 3 Analyzing the Protocol

We rely on a technique introduced in [BCK02,BK02,BCK01], which "unfolds" a graph transformation system, beginning with the start graph and producing a branching structure by adding all possible evolutions of the system. The result is a so-called Petri graph, containing both the graph structure of the system and a Petri net, describing object or message creation and deletion. Since the emerging Petri graph is usually infinite, additional over-approximating folding steps are introduced, keeping the unfolding finite.

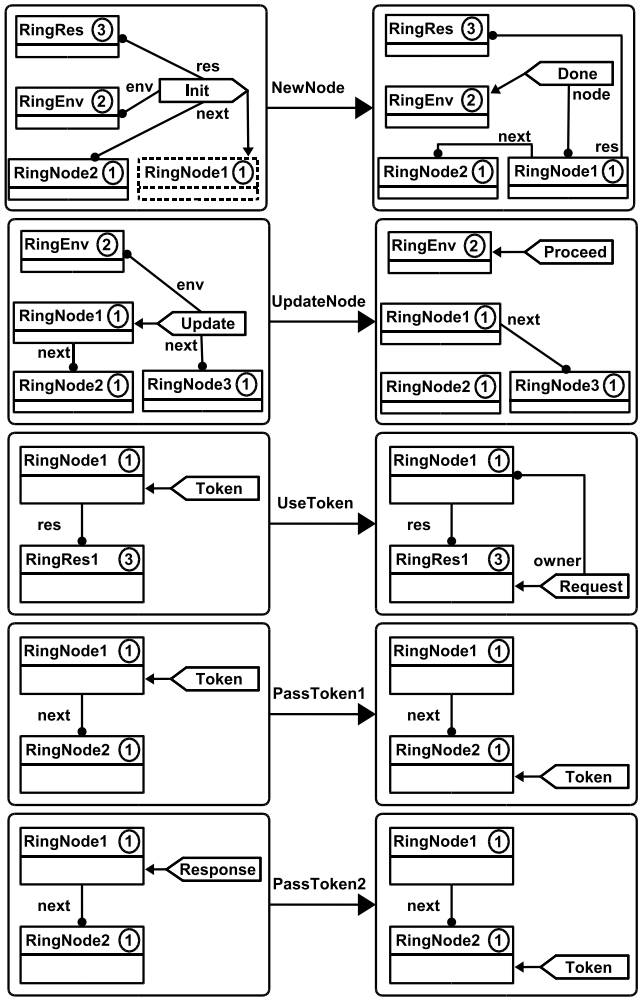


**Fig. 7.** Rules for *RingNode* entity.

### 3.1 Translating the Protocol

In order to be able to use the approximated unfolding technique described above, we first have to translate the mutual exclusion protocol described in Figures 1–7 into hypergraph rewriting. A hypergraph in our setting consists of unlabelled nodes and labelled hyperedges, where a hyperedge can be connected to an arbitrarily long sequence of nodes.

The translation process is mostly mechanical. The nodes of the object-based graph grammar are converted into hyperedges, whereas the connections between nodes become nodes in the hypergraph.

The initial graph of Figure 1, for instance, is converted into the hypergraph depicted in Figure 8. The 0-th tentacle of every hyperedge connects the hyperedge to a node representing its "identity". The rest of the tentacles represent the connections present in the framework of object-based graph grammars.

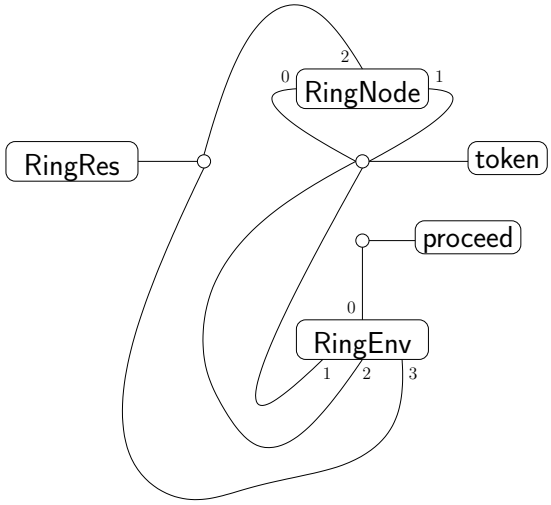


**Fig. 8.** The initial hypergraph.

Graph transformation rules are translated in a similar way. For instance Figure 9 depicts rule *InsertNode*. The dashed lines represent the embedding. Note that since in the framework of hypergraph rewriting, the number of tentacles of a hyperedge is determined by its label, we have to introduce a new hyperedge labelled *RingNodeNew* in order to represent uninitialized ring nodes.

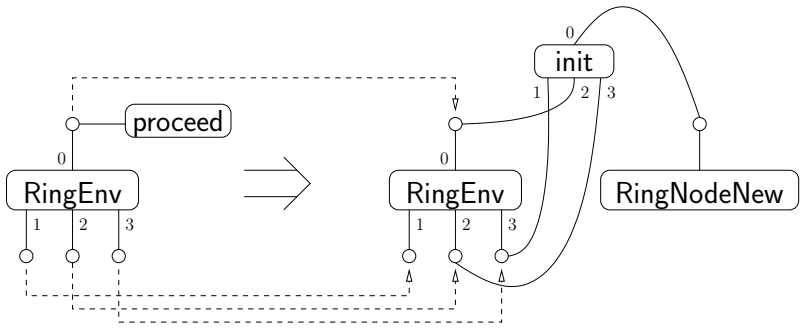


**Fig. 9.** Translation of rule *InsertNode*.

The approximated unfolding algorithm is implemented in a tool called AUGUR, a verification tool for graph transformation systems, which accepts input files in GXL (Graph Exchange Language), an XML standard for the exchange of graph transformation systems.[6]

Below, we show how the initial hypergraph is represented in GXL syntax. The keyword `rel` (= relation) defines a hyperedge, whereas `relend` introduces the tentacles of a hyperedge. Additional syntactic constructs are used to describe graph transformation rules.

```
<Graph edgeids="true" hypergraph="true" edgemode="undirected">
   <node id="rnid" />
   <node id="envid" />
   <node id="resid" />
   <rel>
      <attr name="label">
         <string>RingNode</string>
      </attr>
    <relend target="rnid" startorder = "0" />  <!-- RingNode id -->
    <relend target="rnid" startorder = "1" />  <!-- next RingNode -->
    <relend target="resid" startorder = "2" /> <!-- RingEnv id -->
   </rel>
   <rel>
      <attr name="label">
         <string>RingEnv</string>
      </attr>
    <relend target="envid" startorder = "0" /> <!-- RingEnv id -->
    <relend target="rnid" startorder = "1" />  <!-- first RingNode -->
    <relend target="rnid" startorder = "2" />  <!-- second RingNode -->
    <relend target="resid" startorder = "3" /> <!-- RingRes id -->
   </rel>
   <rel>
      <attr name="label">
         <string>RingRes</string>
      </attr>
    <relend target="resid" startorder = "0" /> <!-- RingRes id -->
   </rel>
   <rel>
      <attr name="label">
         <string>proceed</string>
      </attr>
    <relend target="envid" startorder = "0" />
   </rel>
   <rel>
      <attr name="label">
```

[6] See `http://www.gupro.de/GXL/`

```
        <string>token</string>
      </attr>
    <relend target="rnid" startorder = "0" />
   </rel>
</Graph>
```

### 3.2 Computing the Petri Graph

AUGUR computes the Petri graph, approximating the original system. The output is again a GXL file describing the Petri graph, with additional nodes standing for transitions and additional tentacle types connecting transitions to hyperedges, which is at the same time a place in the Petri net. The following excerpt from a GXL file shows a Petri graph containing three (graph) nodes, a hyperedge (rel) and a node representing a transition.

```
<?xml version="1.0"?>
<gxl>
<graph id="result" hypergraph="true" edgemode="undirected">
<node id="_509">
<attr name="vertex"/>
</node>
<node id="_510">
<attr name="vertex"/>
</node>
<node id="_511">
<attr name="vertex"/>
</node>

[...]

<rel id="_513">
<attr name="label">
<string>response</string>
</attr>
<relend target="_509" role="vertex" startorder="0"/>
<relend target="_537" role="postset">
<attr name="weight">
<int>1</int>
</attr>
</relend>
<relend target="_539" role="preset">
<attr name="weight">
<int>1</int>
</attr>
</relend>
</rel>
```

```
[...]

<node id="_539">
<attr name="transition"/>
<attr name="rule">
<string>PassToken2</string>
</attr>
</node>
</graph>
</gxl>
```

In order to be able to draw the entire Petri graph, we split it into its graph component (depicted in Figure 10) and its Petri net component (see Figure 11). These structures can be computed and visualized directly using AUGUR (see Figure 12 and Figure 13).

Note that the hyperedges of the graph are at the same time the places of the Petri net, marked by the same numbers. The numbers are the unique identities of nodes, hyperedges and transitions, taken directly from the GXL output of the analysis tool. Although a ring might consist of several ring nodes, each with its own identity, these identities have been fused in the approximation and are represented by node 512.

The Petri graph $P$ approximates the original graph transformation systems in the following sense: For every reachable graph $G$ in the system, there exists a reachable marking $m$ in the Petri net with the property that the graph $G'$ obtained by duplicating every hyperedge of $P$ according to its multiplicity in $m$ is a homomorphic image of $G$. For instance the initial marking—which is indicated by bold lines—induces a subgraph into which the initial graph (see Figure 8) can be mapped. In this case the mapping is injective, however, due to approximation this need not necessarily be the case. (For more details see [BCK01].)

This implies that we can draw the following conclusions from a Petri graph:

- If two edges with specific edge labels are *not* connected in the Petri graph, then such edges will also never be connected in a reachable graph.
- Upper bounds for the number of tokens in a place can be seen as upper bounds for the number of edges with a specific label.

In general, properties of a much more complex nature can be verified by combining both the graph structure and the Petri net structure. For instance [BKK03] describes how to translate formulas in a second-order monadic graph logic into formulas on Petri net markings. For our present purposes, the two conclusions described above suffice.

### 3.3 Analyzing Structural Properties

Directly from the graph structure depicted in Figure 10, we can infer information about the possible targets of a message.

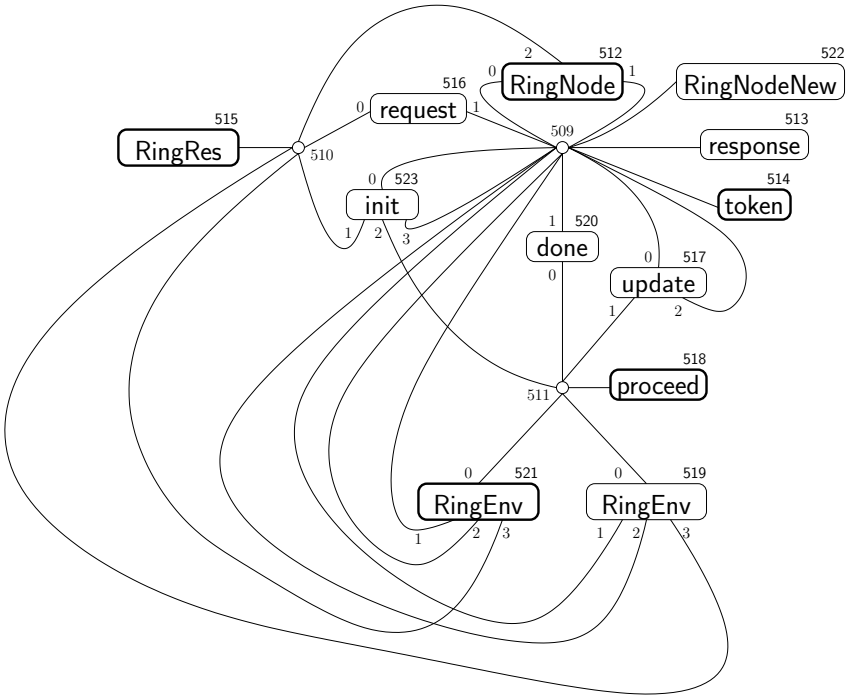


**Fig. 10.** The graph underlying the computed Petri graph.

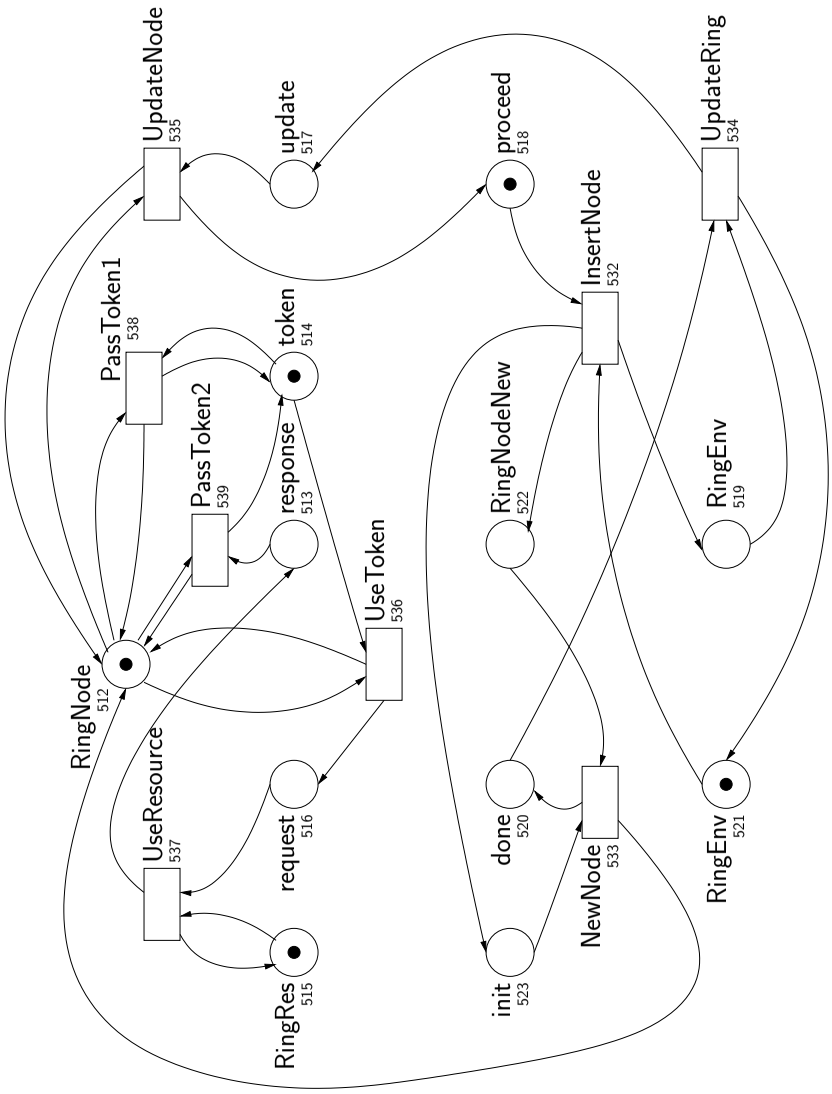


**Fig. 11.** The Petri net underlying the computed Petri graph.

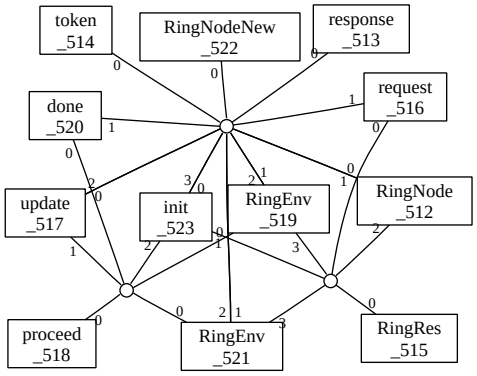


**Fig. 12.** The graph underlying the computed Petri graph (output of the tool AUGUR).

| entity | receives at most the following messages |
|---|---|
| *RingEnv* | *proceed*, *done* |
| *RingRes* | *request* |
| *RingNode* | *response*, *token*, *update*, *init* |

This is consistent with their intended purpose.

### 3.4 Analyzing Multiplicities

More important than the information obtained above is to check that mutual exclusion is guaranteed. Specifically, we want to check that at most one *request* message is sent to the resource at any given moment in time, i.e., we want to establish an upper bound of one token for the place *request*.

We do this by computing the coverability graph of the Petri net in Figure 11, using the Petri net tool LoLA [Sch00]. For this we first convert the Petri net under consideration into the LoLA input syntax, an excerpt can be found below. This conversion into LoLA syntax is automatically performed by AUGUR.

```
PLACE RingNode_512,RingRes_515,request_516,response_513,token_514,
      update_517,init_523,done_520,RingNodeNew_522,proceed_518,
      RingEnv_521,RingEnv_519;

MARKING RingNode_512: 1,RingRes_515: 1,token_514: 1,proceed_518: 1,
        RingEnv_521: 1;

TRANSITION UseResource_537
CONSUME RingRes_515: 1,request_516: 1;
PRODUCE RingRes_515: 1,response_513: 1;

[...]
```

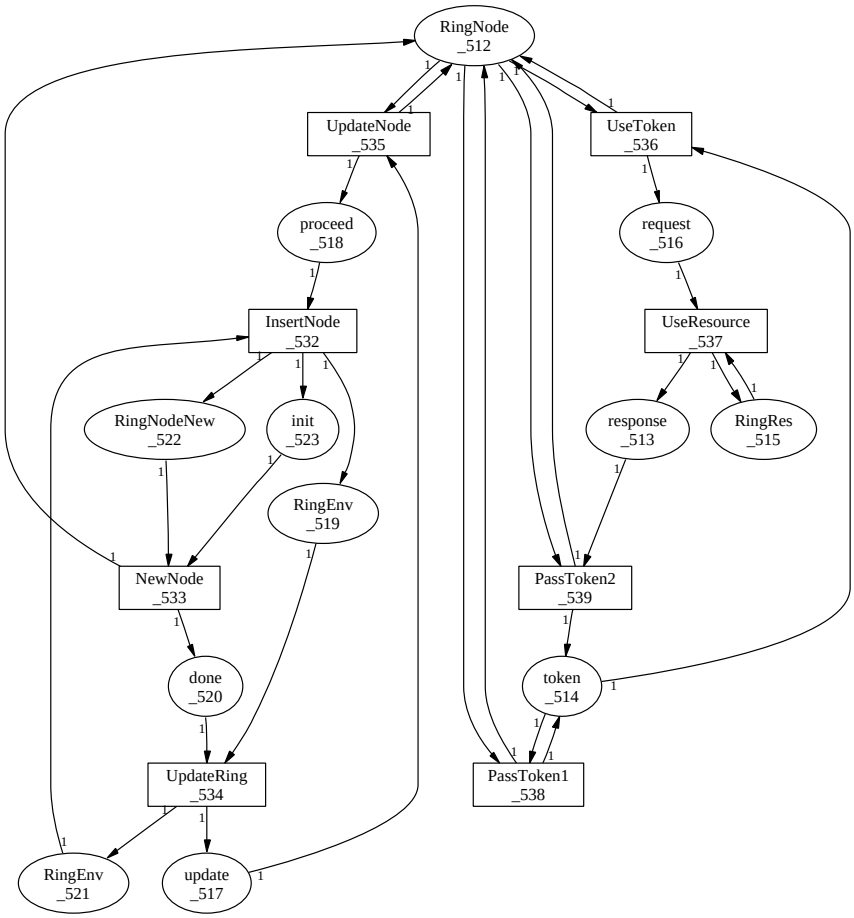


**Fig. 13.** The Petri net underlying the computed Petri graph (output of the tool Augur).

Parts of the coverability graph consisting of states and transitions computed by LoLA are shown below. We do not show the entire graph here, which consists of 24 states and 56 edges. But we remark that in every state, the place `request_516` representing all hyperedges with label *request* always contains at most one token, which establishes the mutual exclusion property.

```
[...]

STATE 1
RingEnv_521 : 1,
RingRes_515 : 1,
proceed_518 : 1,
request_516 : 1,
RingNode_512 : 1

UseResource_537 -> 2
InsertNode_532 -> 23

STATE 0
token_514 : 1,
RingEnv_521 : 1,
RingRes_515 : 1,
proceed_518 : 1,
RingNode_512 : 1

UseToken_536 -> 1
PassToken1_538 -> 0
InsertNode_532 -> 22
```

## 4 Extensions

Another property we would like to analyze is fairness. Is it possible to show that every ring node will—at some point in time—be able to obtain the token and access the resource? Fairness is in general difficult to show with abstract interpretation, because of the over-approximation involved. With the present approximation we can not ensure that the token is passed on to the next ring node as soon as it has been used.

A possible solution would be to first create a ring of ring nodes. As soon as a ring node has obtained the resource, it will change its state. Then it should be possible to reformulate the fairness condition by demanding that every ring node changes its state eventually.

## 5 Conclusion

In the future we plan to conduct more case studies, treating more complex protocols specified by object-based graph grammars as well as algorithms working

on pointer structure. One such case study which has already been concluded is the static analysis of red-black trees, a variant of balanced search trees.

Future research in this area will involve the further development of techniques for abstraction refinement that show a way of dealing with over-approximations that are too coarse. Some preliminary work in this direction has already been presented in [BK02]. Furthermore we plan to investigate the verification of connectivity information—so far we can only show that two components are *not* connected—and the further study of logics and specification languages (see also [BKK03]).

*Acknowledgements:* The second author would like to thank Paolo Baldan and Andrea Corradini for many discussions on approximated unfoldings and Vitali Kozioura, Ingo Walther and Lars Heinemann for implementing the AUGUR tool.

## References

[BCK01] Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR '01*, pages 381–395. Springer-Verlag, 2001. LNCS 2154.

[BCK02] Paolo Baldan, Andrea Corradini, and Barbara König. Static analysis of distributed systems with mobility specified by graph grammars—a case study. In H. Ehrig, B. Krämer, and A. Ertas, editors, *Proc. of IDPT '02 (Sixth International Conference on Integrated Design & Process Technology)*. Society for Design and Process Science, 2002.

[BK02] Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In *Proc. of ICGT '02 (International Conference on Graph Transformation)*, pages 14–29. Springer-Verlag, 2002. LNCS 2505.

[BKK03] Paolo Baldan, Barbara König, and Bernhard König. A logic for analyzing abstractions of graph transformation systems. In *Proc. of SAS '03 (International Static Analysis Symposium)*, pages 255–272. Springer-Verlag, 2003. LNCS 2694.

[DR00] F. L. Dotti and L. Ribeiro. Specification of mobile code systems using graph grammars. In *4th International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 177 of *IFIP Conference Proceedings*, pages 45–64, USA, 2000. Kluwer Academic Publishers.

[Rei80] W. Reisig. A graph grammar representation of nonsequential processes. In H. Noltemeier, editor, *Graphtheoretic Concepts in Computer Science*, pages 318–325. Springer-Verlag, 1980. LNCS 100.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, volume 1. World Scientific, 1997.

[Sch00] Karsten Schmidt. LoLA: A low level analyser. In *Proc. of ATPN (Application and Theory of Petri Nets)*, pages 465–474. Springer, 2000. LNCS 1825.

[SW01] Davide Sangiorgi and David Walker. *The $\pi$-calculus—A Theory of Mobile Processes*. Cambridge University Press, 2001.