Case Study: Verification of a Leader Election Protocol using AUGUR[*]

Barbara König[1] and Vitaly Kozyura[2]

[1] Abteilung für Informatik und Angewandte Kognitionswissenschaft, Universität Duisburg-Essen, Germany
[2] SAP Research, Darmstadt, Germany

Abstract. We consider a case study of a leader election protocol and verify it using the tool AUGUR, which is based on unfolding techniques for graph transformation systems. We first investigate a finite-state variant of the leader election protocol and show how to verify it using McMillan-style unfoldings, avoiding an exponential explosion of the state space. Then, in a next step, we consider a parametric version based on attributed graph transformation. This variant is verified via approximated unfoldings in combination with counterexample-guided abstraction refinement.

1 Introduction

We solve a case study concerning a leader election protocol that one of the authors proposed for the GraBaTs tool contest in 2009, to be held at the Fifth International Workshop on Graph-Based Tools in Zurich, Switzerland.

A simple leader election protocol (according to Chang und Roberts [5]) works as follows: there is a set of processes arranged in a ring, i.e., every process has a unique predecessor and a unique successor. Furthermore each process has a unique Id and there exists a total order on the Ids (assume that Ids are natural numbers).

The leader will be the process with the smallest Id, however no process knows what is the smallest Id at the start of the protocol. Hence every process generates a message with its own Id and sends it to its successor. A received message with content MId is treated as follows by a process with Id PId:

- if $MId < PId$, then forward the message to the next successor
- if $MId = PId$, then the process declares itself the leader
- If $MId > PId$, then do not pass on the message (or alternatively discard it)

The description of the case study called for the verification of this protocol, specified as a graph transformation system. The property to verify is that there will never be two processes declaring themselves as leaders. It was left open whether to consider a finite or an infinite state version.

[*] Supported by DFG project SANDS.

2 Finite-State Version

We now show how to apply unfolding techniques in the style of Winskel [13] and McMillan [12] to verify the protocol. We consider a finite-state variant by slightly modifying the rules given in the case study description (since those are infinite-state). Specifically we make sure that each process generates at most one message.

Note that we work with hypergraphs containing 0-ary, unary and binary edges. The rules are given in Figure 1 where preservation of items (only nodes so far) is indicated by small numbers. The start graph presented here is only an example graph for $n = 3$ processes. In the following we describe how to verify systems with a ring of fixed, but arbitrary size, which does not extend or shrink during runtime (such a behaviour is treated in Section 3). Furthermore the system is modelled in such a way that there are many messages left after having elected the leader, which however does not compromise the correctness of the protocol.

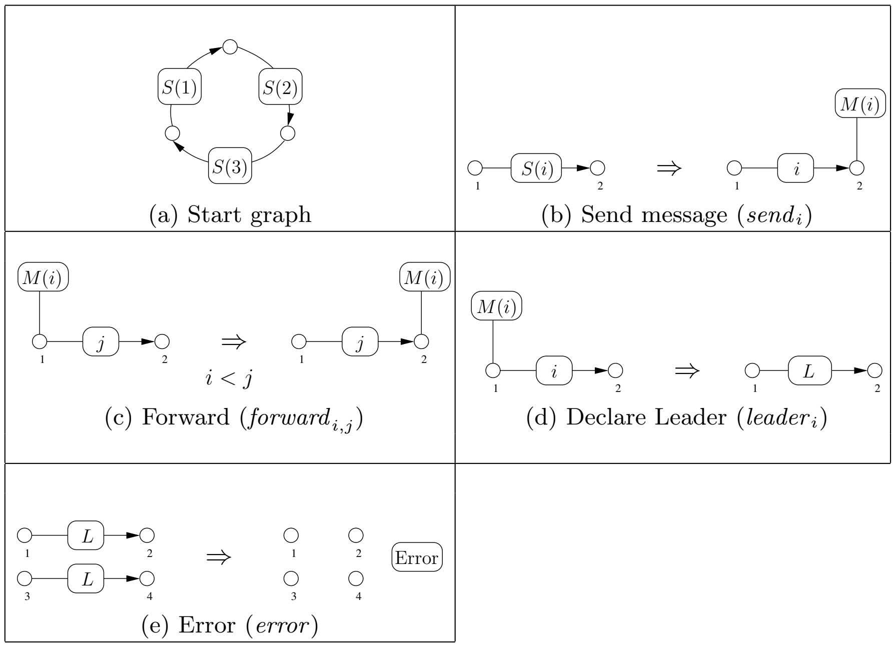

Fig. 1: Leader election (start graph and rewriting rules, finite-state version).

We also add an additional rule that signals an error, i.e., there are two leaders in the ring. Note that in our tool matches of left-hand sides may be non-injective on nodes, but are always injective on edges, hence this last rule correctly indicates an error.

Now, the system as it is specified here suffers from an exponential explosion in the state space. Assume that we have n processes located in ascending order on the ring and each has sent its message. Then, during runtime, the message $M(1)$ might be located at any of the n nodes, message $M(2)$ can be at $n-1$ nodes (it is never at the node between 1 and 2), ..., in general message $M(i)$ can be at one of $n-i+1$ different nodes. This gives us at least $n!$ different states.

We now fix the parameter n and consider the unfolding of the graph transformation system in order to come to terms with the state explosion. The unfolding of a system fully describes its concurrent behaviour in a single branching structure, representing all the possible computation steps and their mutual dependencies, as well as all reachable states; the effectiveness of the approach lies in the use of partially ordered runs, rather than interleavings, to store and handle explanations extracted from the system model.

The unfolding procedure begins with the start graph, looks for left-hand sides and attaches right-hand sides, however without deleting the left-hand sides. Furthermore it introduces (Petri net) transitions recording which left-hand sides can be consumed in order to produce the corresponding right-hand sides. In further steps one looks only for concurrent left-hand sides, i.e., those that can be covered by (Petri net) tokens. This results in a Petri graph, i.e., a Petri net where the places of the net are interpreted as edges of a graph.

Since this system is terminating, it is sufficient to unfold until no more unfolding steps are possible. In finite-state but nonterminating systems it is necessary to look for so-called cut-off transitions in order to keep the unfolding finite (see [1]).

The resulting unfolding contains all reachable graphs, represented by reachable markings in the Petri nets, and instances of all rules which can be applied, represented by transitions. This means specifically that if the error rule is not present in the unfolding, it can never be applied in the original system.

However, taking the rules of Figure 1 above, the exponential blowup *is* present, despite unfoldings being a partial order technique.[3] The reason for this is that due to the deletion and recreation of process j in rule (c) (Forward) there is a large amount of conflict and branching in the unfolding. Hence, we again slightly modify the rules by adding the edge labelled j in rule (c) to the rule interface, i.e., it is preserved by the rule instead of being deleted and recreated. (One way to represent this in Figure 1 would be to add numbers also to hyperedges.) This also means that instead of ordinary Petri nets we have to employ contextual nets, i.e., nets with read arcs [1].

The modification has the effect that the different paths taken by the messages are recorded independently of each other, where the path of message $M(i)$ has length $O(n-i+1)$. Or, to say it in other words, each "forwarding situation" ($M(i)$ is forwarded by j) is only recorded once. So, asymptotically, the size of the unfolding is $O(n^2)$ instead of $O(n!)$.

[3] Note that this blowup might be avoided be using more sophisticated cutoff conditions than the ones we have implemented.

Figure 2 shows the unfolding for $n = 4$. Note that this is only the Petri-net component of the unfolding, we omit the nodes still attached to the hyperedges. The boxes represent transitions where a transition is labelled by the rule name and a unique id. The circles represent places/edges, marked with the edge labels and again a unique id. The dashed lines are the read arcs of the Petri net, recording what is preserved by the transitions. Note also that no transition corresponding to rule (e) (Error) is present and that the system is hence correct.

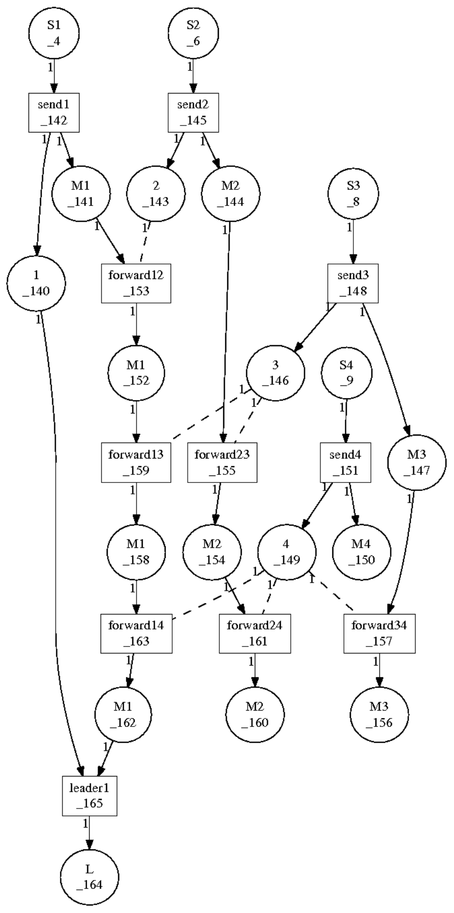

Fig. 2: The unfolding of the leader election protocol for $n = 4$.

The start graph and the rule set for n processes are generated by a C program. The rule set contains $O(n^2)$ rules, i.e., it has approximately the same size than the unfolding itself. Note that here we do not use attributed graphs, hence we can not work with rule schemata.

We then employ a stand-alone extension of AUGUR that produces McMillan-style unfoldings of graph transformation systems. It was developed by Julian Bart in [4] and it is currently being properly integrated into AUGUR 2 with its graphical user interface. The tool can also handle read arcs, but the combination with cut-offs is only correct for read-persistent graph grammars [1, 3].

We let the tool run on several instances of the leader election protocol and obtained the runtimes and sizes of the unfolding recorded in Table 1. While the size of the unfolding is quadratic in n, the runtime does not seem to be quadratic, which is probably due to the overhead caused by the data structures which are needed to produce the unfolding.

n	run time (sec)	places	transitions
4	0.01	19	11
10	0.04	76	56
25	0.94	376	326
50	25.37	1376	1276
75	158.23	3001	2851
100	555.08	5251	5051

Table 1: Runtime results for the finite-state case.

The runtime in this paper have been measured on a machine with 2×Xeon 2.4 GHz and 2GB RAM.

3 Parametric Version

We now treat the parametric case, i.e., we will verify the protocol for all values of n using an approximated unfolding technique [2]. That is, in addition to unfolding steps we add folding steps, which merge parts of the Petri graph in order to obtain a finite over-approximating Petri net.

In order to handle this with a finite rule set we will use an attributed graph transformation system, where specifically the message and process Ids are stored as attributes. Furthermore we introduce several rules for generating a ring of arbitrary size, that is created before the protocol starts. This case study is also considered in [11] and shortly discussed in [10].

The rules of our attributed graph transformation system are given in Fig. 3. We start with a single unary hyperedge representing a counter, which at the beginning has value 1. The counter is increased, subsequently extending the ring (rules “Create First Station” and “Create Station”). Note that stations will be inserted in arbitrary order on the ring (not necessarily in ascending order). Then, at some point we non-deterministically decide that the loop is ready (rule “Loop Ready”) and set the counter to 0.

Now the protocol itself starts, as described earlier in this paper (see rules “Send Message” and “Forward”). Note that messages are only being created if

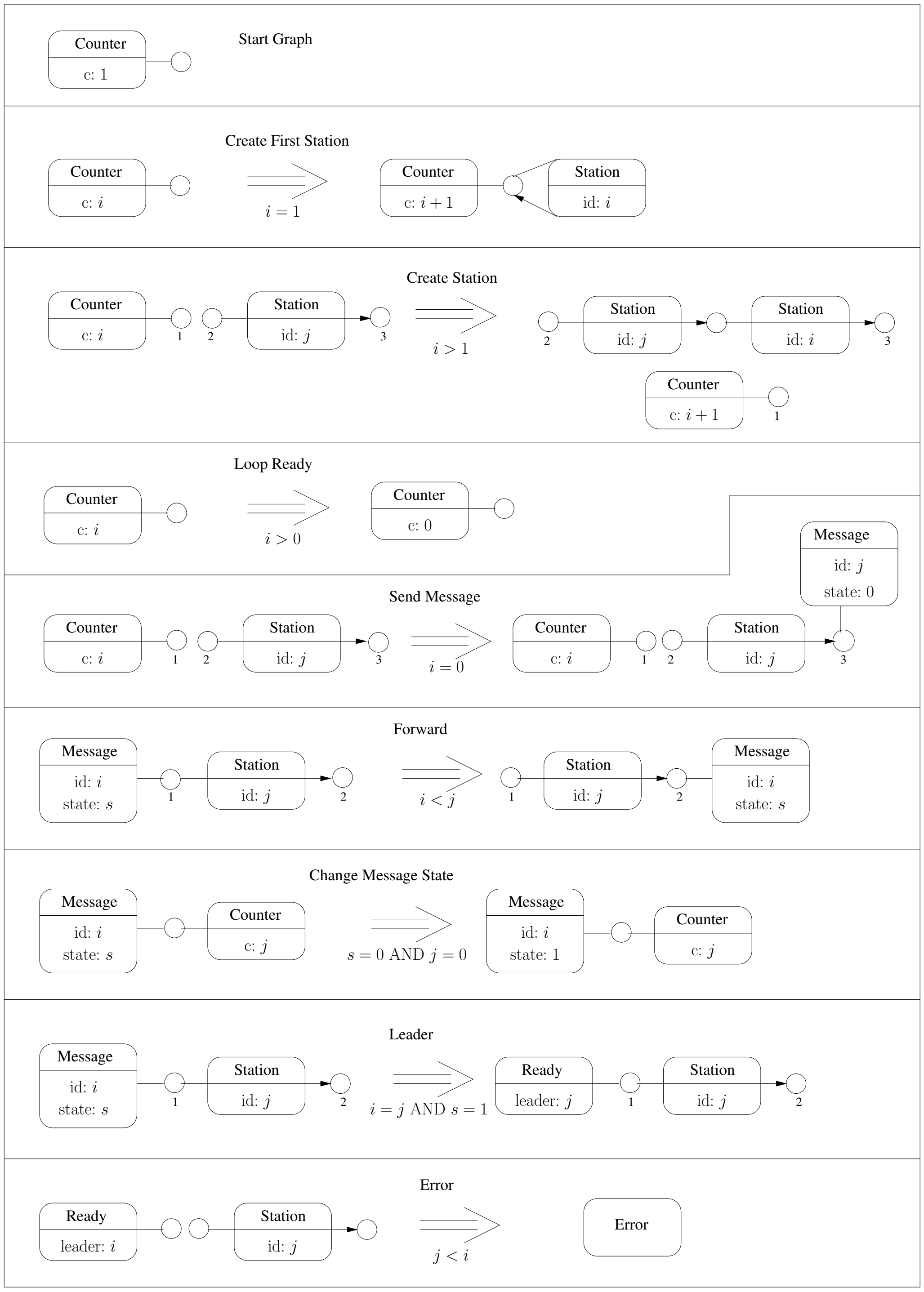

Fig. 3: Leader election (start graph and rewriting rules, parametric version).

the counter is set to 0. Similary to before, a process declares itself the leader (rule "Leader"), by adding an edge labelled "Ready", and there is an error rule that is applied whenever there is a leader, but there is a station with a smaller Id in the ring. (Instead of using this error rule we could also have used an error rule analogous to the one in Section 2).

Now, it remains to explain rule "Change Message State", which is actually a small trick. We use this rule which is quite trivial from the protocol's point of view, but helps us a lot with verification. We mark one chosen place in the ring (in our case we consider the edge "Counter" as being such a marker) and we regard a message as having passed through the whole ring if it comes back to the station which has sent it *and* in addition has visited the marked place in between. A process only declares itself as leader if it receives a message of this kind containig its own Id. Rule "Change Message State" helps us with the analysis of the over-approximations, where stations can be merged. Without it, it is not directly possible to verify the protocol.

Now we want to show that no error edge is ever created, by using approximated unfoldings. Approximated unfoldings are obtained by not only unfolding the system, but folding back at some points in order to obtain a finite over-approximating Petri net with hypergraph structure, also called a Petri graph. Due to the folding steps we merge (potentially unrelated) components and the resulting Petri nets may have more runs than the original system [2]. However, whenever a rule can be applied in the original system, it will also be present in the over-approximation.

During unfolding, we do not treat the attributes in any special way, but as soon as the approximated unfolding is ready, we treat the resulting net as a coloured Petri net [7], where tokens are associated with attributes. Since integers have an infinite domain, we use some predefined abstractions. Here we first take interval abstraction with the interval $[0, 0]$ (i.e., using the abstract values "zero" and "many").

After producing the coarsest unfolding of the attributed graph transformation system and analysing it using coverability graphs we obtain a counterexample of seven steps leading to the edge "Error" and signalling the application of the error rule. Considering this counterexample we see that the approximation is structurally too coarse and can be refined using counterexample-guided abstraction refinement techniques [9, 10]. Afterwards four iterations of abstraction refinement are automatically applied: two with structural refinement and two which refine the attribute abstraction to the interval $[0, 2]$ (using the abstract values "zero", "one", "two" and and "many"). The coverability check of the resulting approximated unfolding tells us, that the edge "Error" is no more coverable. This means that we have successfully verified the protocol.

The verification has been performed by using[4] AUGUR 2 with its graphical user interface. The whole procedure took 48.15 seconds.

Figure 4 shows the hypergraph component of the approximated unfolding (the Petri graph component is too large and is not shown here). The approxi-

[4] The tool is available from `http://www.ti.inf.uni-due.de/research/augur/`.

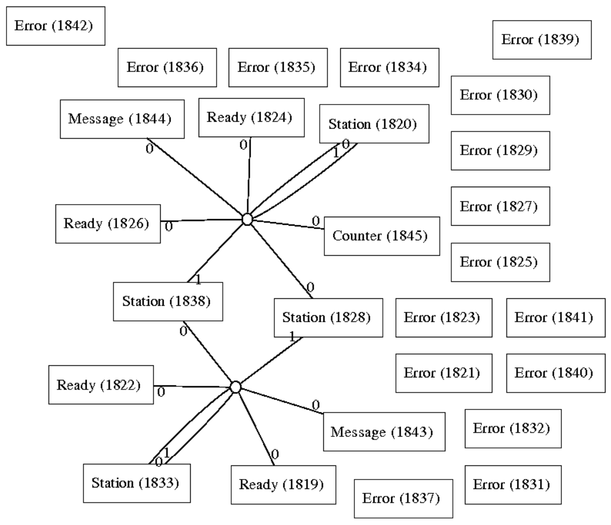

Fig. 4: Hypergraph component of the approximated unfolding.

mated unfolding contains several error edges, but none of them can be covered in the net. Furthermore the ring has been reduced to two nodes, with "station loops" on both of them. Note however that the counter is only attached to one of those nodes, which allows us to record the passage of a message.

The unfolding automatically works in such a way that stations numbered 1820 and 1828 have always Id 1 (i.e., the smallest Id) as attribute value, whereas the stations number 1833 and 1838 have larger Ids.[5] Now the "trick" is that a message sent by one of the latter two stations has to pass the counter and come back before it is received. On this path however it will for sure pass station number 1828 and is not forwarded.

4 Conclusion

We have shown how to verify a leader election protocol using unfolding techniques; more specifically we have shown that we will never obtain two leaders. To show that a leader is eventually chosen is more complex, since this is a liveness property instead of a safety property. In the finite-state case there is a single transition signaling leader election which is not in conflict to any of the other transitions. Hence we can be sure that this event will eventually take place. However, in the parametric case where we use over-approximation it is not directly possible to show successful termination.

[5] Note that in the unfolding edges with the same Id might occur in different branches and are hence all present in the underlying graph structure.

Future work will consist in trying to verify a more complex version of the leader election protocol, introduced in [6] that lowers the number of messages being exchanged.

Note furthermore that we have also addressed the leader election protocol in [8], where we used a backwards analysis procedure whose termination is based on the graph minor theorem. This backwards technique gives us a decision procedure for certain classes of graph transformation systems.

Acknowledgements: We would like to thank the students who worked with us on the project and especially Julian Bart who implemented the McMillan-style unfolding of graph transformation systems.

References

1. Paolo Baldan, Andrea Corradini, and Barbara König. Verifying finite-state graph grammars: an unfolding-based approach. In *Proc. of CONCUR '04*, pages 83–98. Springer-Verlag, 2004. LNCS 3170.
2. Paolo Baldan, Andrea Corradini, and Barbara König. A framework for the verification of infinite-state graph transformation systems. *Information and Computation*, 206:869–907, 2008.
3. Paolo Baldan, Andrea Corradini, Barbara König, and Stefan Schwoon. McMillan's complete prefix for contextual nets. *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, 5100:199–220, 2008.
4. Julian Bart. Effiziente Entfaltungsalgorithmen für Graphersetzungssysteme. Master's thesis, Universität Stuttgart, June 2005. No. 2290.
5. E.J.H. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communication of the ACM*, 22(5):281–283, 1979.
6. A. Itai and M. Rodeh. Symmetry breaking in distributed networks. In *Proceedings of the 22nd IEEE Symposium on Science*, pages 150–158. IEEE Press, 1981.
7. K. Jensen. An introduction to the practical use of coloured Petri nets. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets II: Applications*, pages 237–292. Springer, 1998. LNCS 1492.
8. Salil Joshi and Barbara König. Applying the graph minor theorem to the verification of graph transformation systems. In *Proc. of CAV '08*, pages 214–226. Springer, 2008. LNCS 5123.
9. Barbara König and Vitali Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *Proc. of TACAS '06*, pages 197–211. Springer, 2006. LNCS 3920.
10. Barbara König and Vitali Kozioura. Towards the verification of attributed graph transformation systems. In *Proc. of ICGT '08 (International Conference on Graph Transformation)*, pages 305–320. Springer, 2008. LNCS 5214.
11. Vitaly Kozyura. *Abstraction and Abstraction Refinement in the Verification of Graph Transformation Systems.* PhD thesis, Universität Duisburg-Essen, forthcoming.
12. K.L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.
13. Glynn Winskel. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 325–392. Springer, 1987. LNCS 255.