The Arbiter Tree Case Study

1 Application Context

This case study models a mutual exclusion protocol based on a tree of binary arbiter processes (see figure 1 for an instance with four clients). Client processes are situated at the leaves of the tree. In order to gain access to the shared resource, they may send a request to their respective parent, which in turn passes the request on to its parent, etc. When the root process of the tree receives a request, it generates a grant which is then propagated back down. When the client is done with the resource, it sends a release signal.

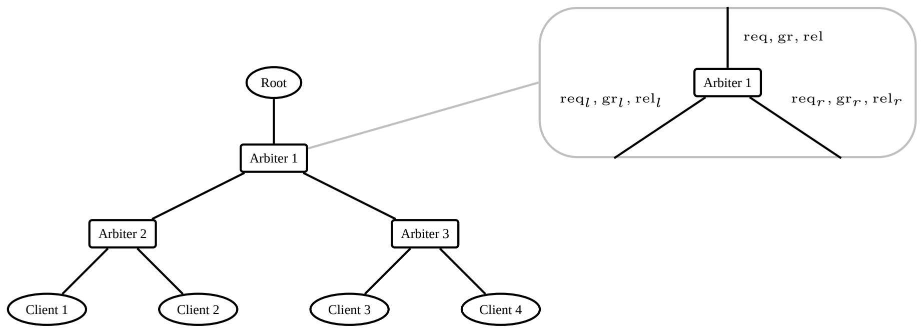
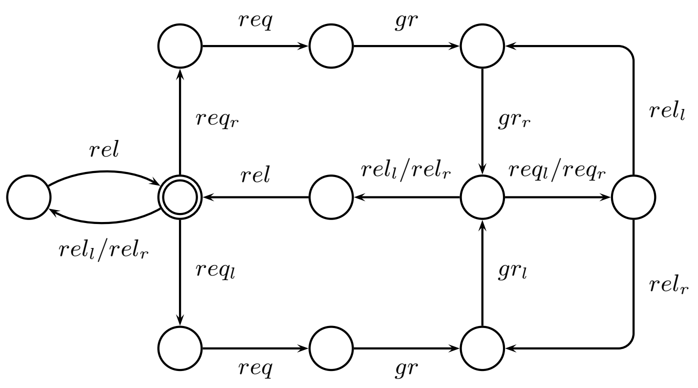

Root
Arbiter 1
Arbiter 2
Arbiter 3
Client 1
Client 2
Client 3
Client 4
req, gr, rel
Arbiter 1
$\mathrm{req}_l, \mathrm{gr}_l, \mathrm{rel}_l$
$\mathrm{req}_r, \mathrm{gr}_r, \mathrm{rel}_r$

Fig. 1. Arbiter tree: Access to a shared resource is controlled by binary arbiters arranged in a tree, with a central root process

In order to avoid blocking client processes, arbiters need to be ready to receive requests from one of their children even when they are already processing one for the other child. In this case it makes sense to send a grant to the second child as soon as a release is received from the first, instead of first forwarding the release upwards and sending a new request. This can be applied asymmetrically (giving a grant to the left child first, and always forwarding releases from the right child to the parent) in order to not monopolize the resource.

2 Model

We model the arbiter tree using finite automata for the root, arbiter, and client processes. The instances are parameterized by the height H of the tree, and contain $2^H - 1$ arbiters and 2^H client processes. The smallest example ($H = 2$) has $1.02 * 10^6$ product states; this increases to $1.88 * 10^{104}$ states in the largest instance($H = 6$).



//

Fig. 2. A faulty arbiter automaton

The case study uses an incorrect implementation which eventually allows several client processes to access the resource simultaneously. This situation results from a faulty client process sending spurious release signals, and a flaw in the arbiters which makes them not check for this possibility and discard such a signal. One such faulty arbiter is shown in fig. 2.

3 Verification Results

Our heuristics are implemented in UPPAAL/DMC which is our extension of UPPAAL for directed model checking. In [1], we compared the performance of UPPAAL/DMC's greedy search and UPPAAL's randomised depth first search (rDF), which is UPPAAL's most efficient standard search method across many examples.

Our results clearly demonstrate the potential of our heuristics. The heuristic searches consistently find the error paths much faster. Due to the reduced search space size and memory requirements, they can solve all problems. At the same time, they find, by orders of magnitude, *much* shorter error paths in *all* cases.

References

1. Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. In Antti Valmari, editor, *Model Checking Software. Proceedings of the 13th International SPIN Workshop (SPIN 2006)*, volume 3925 of *Lecture Notes in Computer Science*, pages 19–34. Springer-Verlag, 2006.