SLAB Benchmarks

Ingo Brückner[1], Klaus Dräger[2], Bernd Finkbeiner[2], and Heike Wehrheim[3]

[1] Carl von Ossietzky Universität, 26129 Oldenburg, Germany
ingo.brueckner@informatik.uni-oldenburg.de
[2] Universität des Saarlandes, Fachrichtung Informatik, 66123 Saarbrücken, Germany
{draeger|finkbeiner}@cs.uni-sb.de
[3] Universität Paderborn, Institut für Informatik, 33098 Paderborn, Germany
wehrheim@uni-paderborn.de

Abstract. Abstraction and slicing are both techniques for reducing the size of the state space to be inspected during verification. In the paper [1], we present a new model checking procedure for infinite-state concurrent systems that interleaves automatic abstraction refinement, which splits states according to new predicates obtained by Craig interpolation, with slicing, which removes irrelevant states and transitions from the abstraction. The effects of abstraction and slicing complement each other. As the refinement progresses, the increasing accuracy of the abstract model allows for a more precise slice; the resulting smaller representation gives room for additional predicates in the abstraction. The procedure terminates when an error path in the abstraction can be concretized, which proves that the system is erroneous, or when the slice becomes empty, which proves that the system is correct.

1 Experiments

We have implemented the new model checking procedure as a small prototype tool named SLAB (for *Sl*icing *ab*stractions). SLAB is implemented in Java (JRE 1.5) and relies on Andrey Rybalchenko's *CLP-Prover* [2] for satisfiability checking and interpolant generation. In Table 1, we give running times of SLAB for a collection of standard benchmarks. Our experiments were carried out on an Intel Pentium M 1.80 GHz system with 1 GByte of RAM. For comparison, we also give the running times of the *Abstraction Refinement Model Checker* ARMC [3] and the *Berkeley Lazy Abstraction Software Verification Tool* BLAST [4] where applicable.

2 Benchmarks

Our benchmarks include a finite-state system (Deque), an infinite-state discrete system (Bakery), and a real-time system (Fisher).

2.1 Deque

The *Deque* benchmark is an abstract version of a cyclic buffer for a double-ended queue. We model the cells of the buffer by n flags, where *true* indicates a currently allocated cell. Initially, all but the first flag are *false*. Adding or deleting an element at either end is represented by toggling a flag under the condition that the values of the two neighboring flags are different: $(true, true, false) \leftrightarrow (true, false, false)$ and $(false, true, true) \leftrightarrow (false, false, true)$. The error condition is satisfied if there are no unallocated cells left in the buffer. Source code of the corresponding transition constraint system that is fed into SLAB for an instance of the Deque benchmark with five buffers is depicted in Figure 1.

Table 1. Experimental results for SLAB vs. ARMC and BLAST: number of iterations of the refinement loop and running times in seconds on the benchmarks Deque (with $5, \ldots, 9$ cells), Bakery (with $2, \ldots, 5$ processes), and Fisher (with $2, 3, 4$ processes). (BLAST is not applicable to the real-time system Fisher.)

	SLAB		ARMC	BLAST
specification	iterations	time (s)	time (s)	time (s)
Deque 5	6	1.34	3.80	2.23
Deque 6	6	1.92	27.65	5.64
Deque 7	8	2.70	255.63	13.64
Deque 8	8	3.15	1277.85	36.63
Deque 9	10	4.80	timeout	90.17
Bakery 2	29	6.30	2.56	9.26
Bakery 3	47	33.53	24.97	1943.17
Bakery 4	71	128.53	988.69	timeout
Bakery 5	96	376.56	timeout	timeout
Fisher 2	42	9.26	3.37	N/A
Fisher 3	335	126.05	339.21	N/A
Fisher 4	2832	2605.85	timeout	N/A

2.2 Fisher

Fisher's algorithm, as described in [6], is a real-time mutual exclusion protocol. Access to a resource shared between n processes is controlled through a single integer variable *lock* and real-time constraints involving two fixed bounds $C1 < C2$. Each process uses an individual (resettable) clock c to keep track of the passing of time between transitions. Each process first checks if the lock is free, then, after waiting for no longer than bound C1, sets *lock* to its (unique) id. It then waits for at least C2, and if the value of the lock is unchanged, accesses the critical resource. When leaving, it frees up the lock. As in the previous benchmark, an error occurs if two processes access the critical resource at the same time. Source code of the corresponding transition constraint system that is fed into SLAB for an instance of the Fisher benchmark with two processes is depicted in Figure 1.

2.3 Bakery

The *Bakery* protocol [5] is a mutual exclusion algorithm that uses *tickets* to prevent simultaneous access to a critical resource. Whenever a process wants to access the shared resource, it acquires a new ticket with a value v that is higher than that of all existing tickets. Before the process accesses the critical resource, it waits until every process that is currently requesting a ticket has obtained one, and every process that currently holds a ticket with a lower value than v has finished using the resource. An error occurs if two processes access the critical resource at the same time. Source code of the corresponding transition constraint system that is fed into SLAB for an instance of the Bakery benchmark with three processes is depicted in Figure 3.

3 Conclusions

On our benchmarks, SLAB outperforms both ARMC and BLAST, and scales much better to larger systems. It appears that the abstract state space constructed by SLAB grows much more slowly in the number of predicates than the (fully exponential) state space considered by standard predicate abstraction: Figure 4 depicts the relation between the number of predicates and the number of abstract states in intermediate abstractions from the verification of the Bakery protocol with three processes.

References

1. Brückner, I., Dräger, K., Finkbeiner, B., Wehrheim, H.: Slicing abstractions. In: Proceedings of the International Symposium on Fundamentals of Software Engineering (FSEN). (2007) Accepted for publication. This paper was awarded the FSEN *Best Paper Award*.
2. Rybalchenko, A.: CLP-prover. http://mtc.epfl.ch/~rybalche/clp-prover/ (2006)
3. Podelski, A., Rybalchenko, A.: ARMC: the logical choice for software model checking with abstraction refinement. In: Proc. PADL'07. (2006)
4. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. In: Proc. SPIN '03. Volume 1427 of LNCS., Springer-Verlag (2003) 235–239
5. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Communications of the ACM **17**(8) (1974) 435–455
6. Manna, Z., Pnueli, A.: Clocked transition systems. Technical Report STAN-CS-TR-96-1566, Computer Science Department, Stanford University (1996)

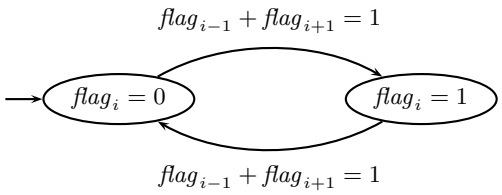

Fig. 1. An automaton modeling one of the buffer cells in the Deque benchmark. Note that indices are modulo N, and for $i = 0$ the state on the right is initial, i.e. $flag_0$ is initially set.

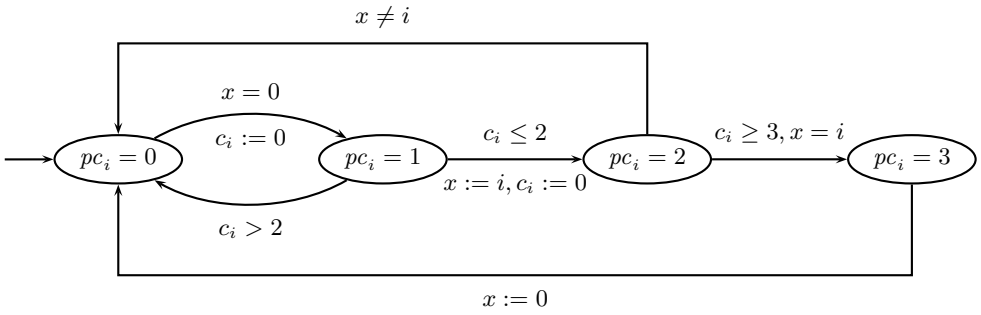

Fig. 2. One of the automata in the Fischer benchmark. The passing of time is modelled by a separate tick transition that can increase all clocks by some arbitrary positive amount.

```
r(p(pc(init),...),p(pc(state1),...), [],
  [_n0P=0,_e0P=0,_pc0P=1,_n1P=0,_e1P=0,_pc1P=1,_n2P=0,_e2P=0,_pc2P=1],1).
r(p(pc(state1),...),p(pc(state1),...),[_pc0=1],
  [_n0P=_n0,_e0P=1,_pc0P=2,_n1P=_n1,_e1P=_e1,_pc1P=_pc1,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],2).
r(p(pc(state1),...),p(pc(state1),...), [_pc0=2,_n1>=_n2],
  [_n0P=_n1+1,_e0P=_e0,_pc0P=3,_n1P=_n1,_e1P=_e1,_pc1P=_pc1,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],3).
r(p(pc(state1),...),p(pc(state1),...), [_pc0=2,_n2>=_n1],
  [_n0P=_n2+1,_e0P=_e0,_pc0P=3,_n1P=_n1,_e1P=_e1,_pc1P=_pc1,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],4).
r(p(pc(state1),...),p(pc(state1),...), [_pc0=3],
  [_n0P=_n0,_e0P=0,_pc0P=4,_n1P=_n1,_e1P=_e1,_pc1P=_pc1,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],5).
r(p(pc(state1),...),p(pc(state1),...), [_pc0=4,_e1=0],
  [_n0P=_n0,_e0P=_e0,_pc0P=5,_n1P=_n1,_e1P=_e1,_pc1P=_pc1,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],6).
r(p(pc(state1),...),p(pc(state1),...), [_pc0=5,_n1>=_n0],
    [_n0P=_n0,_e0P=_e0,_pc0P=6,_n1P=_n1,_e1P=_e1,_pc1P=_pc1,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],7).
r(p(pc(state1),...),p(pc(state1),...), [_pc0=6,_e2=0],
    [_n0P=_n0,_e0P=_e0,_pc0P=7,_n1P=_n1,_e1P=_e1,_pc1P=_pc1,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],8).
r(p(pc(state1),...),p(pc(state1),...), [_pc0=7,_n2>=_n0],
    [_n0P=_n0,_e0P=_e0,_pc0P=8,_n1P=_n1,_e1P=_e1,_pc1P=_pc1,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],9).
r(p(pc(state1),...),p(pc(state1),...), [_pc0=8],
    [_n0P=0,_e0P=_e0,_pc0P=1,_n1P=_n1,_e1P=_e1,_pc1P=_pc1,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],10).
r(p(pc(state1),...),p(pc(state1),...), [_pc1=1],
    [_n1P=_n1,_e1P=1,_pc1P=2,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],11).
r(p(pc(state1),...),p(pc(state1),...), [_pc1=2,_n0>=_n2],
    [_n1P=_n0+1,_e1P=_e1,_pc1P=3,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],12).
r(p(pc(state1),...),p(pc(state1),...), [_pc1=2,_n2>=_n0],
    [_n1P=_n2+1,_e1P=_e1,_pc1P=3,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],13).
r(p(pc(state1),...),p(pc(state1),...), [_pc1=3],
    [_n1P=_n1,_e1P=0,_pc1P=4,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],14).
r(p(pc(state1),...),p(pc(state1),...), [_pc1=4,_e0=0],
    [_n1P=_n1,_e1P=_e1,_pc1P=5,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],15).
r(p(pc(state1),...),p(pc(state1),...), [_pc1=5,_n0>=_n1],
    [_n1P=_n1,_e1P=_e1,_pc1P=6,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],16).
r(p(pc(state1),...),p(pc(state1),...), [_pc1=6,_e2=0],
    [_n1P=_n1,_e1P=_e1,_pc1P=7,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],17).
r(p(pc(state1),...),p(pc(state1),...), [_pc1=7,_n2>=_n1],
    [_n1P=_n1,_e1P=_e1,_pc1P=8,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],18).
r(p(pc(state1),...),p(pc(state1),...), [_pc1=8],
    [_n1P=0,_e1P=_e1,_pc1P=1,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n2P=_n2,_e2P=_e2,_pc2P=_pc2],19).
r(p(pc(state1),...),p(pc(state1),...), [_pc2=1],
    [_n2P=_n2,_e2P=1,_pc2P=2,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n1P=_n1,_e1P=_e1,_pc1P=_pc1],20).
r(p(pc(state1),...),p(pc(state1),...), [_pc2=2,_n0>=_n1],
    [_n2P=_n0+1,_e2P=_e2,_pc2P=3,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n1P=_n1,_e1P=_e1,_pc1P=_pc1],21).
r(p(pc(state1),...),p(pc(state1),...), [_pc2=2,_n1>=_n0],
    [_n2P=_n1+1,_e2P=_e2,_pc2P=3,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n1P=_n1,_e1P=_e1,_pc1P=_pc1],22).
r(p(pc(state1),...),p(pc(state1),...), [_pc2=3],
    [_n2P=_n2,_e2P=0,_pc2P=4,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n1P=_n1,_e1P=_e1,_pc1P=_pc1],23).
r(p(pc(state1),...),p(pc(state1),...), [_pc2=4,_e0=0],
    [_n2P=_n2,_e2P=_e2,_pc2P=5,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n1P=_n1,_e1P=_e1,_pc1P=_pc1],24).
r(p(pc(state1),...),p(pc(state1),...), [_pc2=5,_n0>=_n2],
    [_n2P=_n2,_e2P=_e2,_pc2P=6,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n1P=_n1,_e1P=_e1,_pc1P=_pc1],25).
r(p(pc(state1),...),p(pc(state1),...), [_pc2=6,_e1=0],
    [_n2P=_n2,_e2P=_e2,_pc2P=7,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n1P=_n1,_e1P=_e1,_pc1P=_pc1],26).
r(p(pc(state1),...),p(pc(state1),...), [_pc2=7,_n1>=_n2],
    [_n2P=_n2,_e2P=_e2,_pc2P=8,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n1P=_n1,_e1P=_e1,_pc1P=_pc1],27).
r(p(pc(state1),...),p(pc(state1),...), [_pc2=8],
    [_n2P=0,_e2P=_e2,_pc2P=1,_n0P=_n0,_e0P=_e0,_pc0P=_pc0,_n1P=_n1,_e1P=_e1,_pc1P=_pc1],28).
r(p(pc(state1),...),p(pc(error),...), [_pc0=8,_pc1=8],
    [],29).
```


Fig. 3. Source code of the transition constraint system for an instance of the Bakery benchmark with three processes.

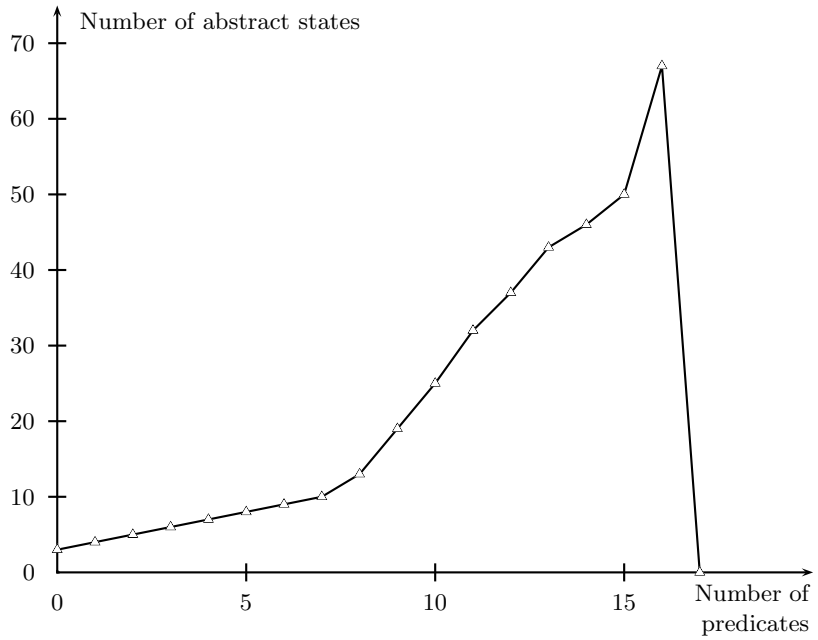

Fig. 4. Relation of the number of abstract states and the number of predicates in intermediate abstractions during the verification of the Bakery protocol with three processes.