Benchmark Description

In the *pressure* benchmark [1] a thread increments a counter while another thread guards the counter from exceeding a maximum value. However, there exists some thread interleavings such that the condition is violated. The *pressure-safe* program limits the simulation time to ensure that the violation does not occur. The threads are effectively non-interfering most of the time as [1] has already observed. However, detecting this non-interference is very difficult, for both static POR as well as dynamic POR, because a read-write dependency exists. A stateful search however can avoid re-exploration of the equivalent states. The variants *pressure-sym*, *pressure-sym2* and *pressure-sym.nb* additionally use symbolic values, more counter variables and an unbounded simulation, respectively.

The *symbolic-counter* benchmarks similar to the pressure benchmark family have an increment and guard thread to ensure a counter variable stays within a predefined range. The counter is initialized with a symbolic variable and is regularly reset during the computation. Another thread performs synchronization and controls simulation time.

The *buffer* program fills an array using multiple threads and then checks the result. All threads write to a different memory locations, thus they are effectively non-interfering. It is based on the buffer-ws-pX benchmark from [5] and has been modified to run without a simulation time limit. Consequently, it has a cyclic state space and cannot be verified with a stateless method.

The *rbuf* benchmarks perform an unbounded simulation that modifies a 32 bit unsigned int value by moving its bits in each step. The principle using two threads and a 8 bit value is shown in Fig. 1. This principle is naturally extended to 32 bit values and four threads (*rbuf-4* and *rbuf2-4*). The value is initialized symbolically and in each step checked that the number of high bits is equal before and after each bit rotation.

The *condition-builder* runs indefinitely but has a finite state space, since every possible execution path runs into a cycle. In this case, the program will repeat its behavior after $k = 8$ or $k = 16$ steps. Essentially, the program manages a boolean condition $b = True$ and a symbolic integer $n = x_1$, which is initially constrained to be in the range $\{0, ..., k-1\}$. The thread C repeats the following behavior indefinitely: it runs for k steps and then verifies the constructed condition. In each step C notifies the threads A and B. The threads A and B will update the condition b in each step i (starting with 0, ending with $k-1$), where $i \neq \lceil (k-1)/2 \rceil$ and $i \neq \lfloor (k-1)/2 \rfloor$, to either (A) $b \wedge (n \neq i)$ or (B) $b \wedge (n \neq (k-1-i))$. So b_{i+1} will have either the new value $b_i \wedge (n \neq i) \wedge (n \neq (k-1-i))$ or $b_i \wedge (n \neq (k-1-i)) \wedge (n \neq i)$, depending whether $\xrightarrow{A,B}$ or $\xrightarrow{B,A}$ is executed. Regardless of the scheduling decision, the same terms will be added to b, but they appear in different orders. All terms can be arbitrarily rearranged since the logic *and* operator is commutative.

The *toy-sym*, *pc-sfifo-sym-1* and *pc-sfifo-sym-2* benchmark respectively are based on the *toy*, *pc-sfifo-1* and *pc-sfifo-2* benchmarks from [2]. The concrete initial values have been replaced with symbolic ones. The *kundu* benchmark and its variants are based on [4], and the *simple-fifo* benchmark family appeared in [3].

Figure 1: The rotating (bit-) buffer benchmark is available in two configurations *rbuf* and *rbuf2*. In every time step a new value is computed based on the current value of the buffer. The left figure shows the principle operation of the *rbuf* benchmark and the figure on the right for the *rbuf2* benchmark.

References

[1] N. Blanc and D. Kroening. Race analysis for SystemC using model checking. *TODAES*, 15(3):21:1–21:32, June 2010.

[2] A. Cimatti, I. Narasamdya, and M. Roveri. Software model checking SystemC. *TCAD*, 32(5):774–787, 2013.

[3] D. Große, H. M. Le, and R. Drechsler. Formal verification of abstract SystemC models. In B. Becker, V. Bertacco, R. Drechsler, and M. Fujita, editors, *Algorithms and Applications for Next Generation SAT Solvers*, number 09461 in Dagstuhl Seminar Proceedings, 2010.

[4] S. Kundu, M. Ganai, and R. Gupta. Partial order reduction for scalable testing of systemc tlm designs. In *DAC*, pages 936–941, 2008.

[5] H. M. Le, D. Große, V. Herdt, and R. Drechsler. Verifying SystemC using an intermediate verification language and symbolic simulation. In *DAC*, pages 116:1–116:6, 2013.