Communicating Process Architectures 2003
Jan F. Broenink and Gerald H Hilderink (Eds.)
IOS Press, 2003

occam for reliable embedded systems: lightweight runtimes and model checking

Denis NICOLE
Department of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, UK.
dan@ecs.soton.ac.uk

Sam ELLIS
ARM Ltd.,110 Fulbourn Road
Cambridge, CB1 9NJ, UK
sam.ellis@arm.com

Simon HANCOCK
Airbus, Filton, Bristol, BS99 7AR, UK
simon.hancock@airbus.com

Abstract. We describe some more recent developments of the SPoC system. We describe a new module in the occam compiler which performs substantial simplifications of the run-time demands made by the compiled code. This has been used successfully both to target a simple PIC microcontroller and to generate input for the SMV model checker.

1. Introduction

The initial development of the SPoC occam compilation system was reported[1] at a previous conference of this series. This is a robust programming environment, suitable for commercial code development. In its original form, it accepted the occam 2.5 programming language and converted a complete parallel occam program into a single program in the C programming language. A simple scheduler implemented within the C code allows the execution of the parallel occam program by the cooperative sequencing of the various program fragments. No operating system support for multithreading or concurrency is required. The resultant code is highly portable and has run on big- and little-endian processors, UNIX and Microsoft operating systems and dedicated signal processors. Macros for the `gdb` debugger permit the debugging of running programs at the occam source level. Additional experimental versions of SPoC have supported distributed message-passing architectures, allcache shared memory systems, dynamic remote process creation and channel migration.

Despite its wide application, there were two key application areas for which SPoC was not entirely suitable: microcontrollers and model checking. In both cases, this was because the domain required that the run-time support demands be even "lighter" than for SPoC. In the case of the microcontrollers, the burning need was to minimise the amount of run-time state and thus of RAM required for an occam program; a 16F84[2] has only 36 bytes of data storage and 1024 words of instruction storage. For model checking, not only did the

amount of run-time state have to be minimised to reduce the amount of checking to a reasonable level, but the restricted execution model of mainstream model checkers had to be targeted. Newer version of the PIC, such as the 16F628, relax the memory constraints considerably; the increase in data memory to 224 bytes is particularly welcome.

We believe we have achieved a reasonable degree of success in tackling both these domains; perhaps one of our key achievements has been to eliminate the learning curve for both targets. The occam programmer is able to perform interesting model checking entirely at the occam level, with no need to learn either a FSM description language or the handling of temporal logic expressions.

2. The occam simplifier

In addition to their limited memory, the PIC microcontrollers have only a restricted stack depth for subroutine return addresses. A module for SPoC was developed which performed a variety of standard compile-time optimizations:

1. Elimination of `SKIP` processes.
2. Constant propagation and elimination of unreachable `IF` and `CASE` processes.
3. Replicator unrolling. In occam, `PAR` replicators are always constant, as are many `ALT`, `IF` and `SEQ` counts. Elimination of these explicit replicators greatly simplifies the run-time and allows many new opportunities for constant propagation. This unrolling is essential to reduce the amount of run-time state exposed to the PIC.
4. Function and procedure inlining. Occam has a very clean copy-in copy-out semantics for procedure parameters and thus inlining, even in the presence of free variables, is relatively straightforward. Again, this gives additional opportunities for constant propagation and dead code elimination.
5. Array inlining. Traditional array indexing imposes unacceptable demands on the PIC architecture. Not only is array indexing awkward to code, the manipulation of interesting arrays is incompatible with the PIC's very small data store. The compiler thus eliminates all array references, replacing them as necessary with generated scalar variables.

As is typical with an optimizing compiler, the various optimizations interact strongly and multiple passes through the optimizer are required to realize all the benefits of the simplifications. It will be seen that many of the optimizations have the effect of reducing run-time dynamic state (array indexes, loop counters, subroutine stacks, at the expense of increasing the code size. This is a fortuitous combination as for both microcontrollers and model checkers, the code length (or model size) is not normally a limiting factor; the dynamic state, on the other hand, is strongly restricted. The reader should also note that occam would support even more aggressive optimizations; it is, for example possible to remove[3] almost all the parallelism and internal channels from an occam code.

3. The PIC run-time

There are a variety of obvious simplifications for the PIC. Firstly, floating-point arithmetic makes little sense on such a simple architecture and all support for floating point is dropped. Similarly, `INT`s are configured as sixteen bit. `INT32` and `INT64` are, however, supported although little used. Another occam feature which is expensive on the PIC is overflow detection for integer arithmetic. This is supported by default, but a compiler option allows the support to be removed, improving code performance. Multiplication is not supported in the PIC hardware, so a special code block, which takes its operands from

fixed memory locations, is used to perform multiplies in software. This is against the inlining philosophy of the compiler, but the considerable size of the multiply code precludes its inlining. Division is currently not supported.

As noted earlier, replicated `PAR`s are unrolled by the optimizer. We still need, however, to maintain state to manage the join at the end of the PAR block; this takes up one byte in memory. The process scheduler uses one bit per deschedule point to indicate which code blocks are runnable; when the running process becomes blocked, the scheduler searches these bits for another runnable code block.

Channels are implemented as one bit per out (`!`) process; an inputting (`?`) process scans the out bits to find a runnable transfer. `ALT`s offer no special problems here although this does effectively result in a polling implementation `ALT`.

Proper support for timers and ports in the compiler makes it easy to write a demonstration code which will, say, flash several LEDs at different rates or implement full duplex serial communications.

4. Model checking

Model checking has become a preferred technique among researchers for assuring the correctness of concurrent hardware or software systems. In short, a complex system is built from a set of interacting finite state machines. An exhaustive simulation is then performed of all possible histories for the complex system. In each state of each history, the property under test is evaluated. The model checker can then report whether the property can ever be violated and, usefully, can directly emit a shortest history to a violation. It is essential in standard model checkers that the number of possible states (instantaneous configurations) of the system be finite; furthermore the complexity of the model checking run grows rapidly with the size of this state.

There are several commercial-grade model checkers available to the programming community. Well-known ones include SMV[4,5], SPIN[6] and FDR[7]. These are remarkably efficient programs but are not yet taken up widely in industry. Perhaps the main obstruction to their adoption is the substantial *semantic-gap* between programming languages and model-checker input. The issues here are by no means as dramatic as with, say, proof systems, but are nevertheless substantial for the working programmer. The program typically has to be rendered from C, Java or occam source into a wholly alien state-machine representation. Furthermore, the property under test has to be described in a form of temporal logic: a predicate calculus with a few extra quantifiers. These two foreign languages are enough to deter all but the most persistent programmers. An alternative formulation of model checking tests whether one program is a refinement of another (i.e. has a lower degree of nondeterminism); this is hardly easier to explain to the typical programmer.

Fortunately, there are ways to de-skill both the key steps in setting up a model checker. First, we can directly check the program in its high level language. There are clearly potential problems here; the variables in a typical program imply a great deal of state. Thus it is normally necessary to simplify the program under test by only tracking the variables which are essential to the property being tested; this is a form of *program slicing*. This simplification does not damage the reliability of the model check; wherever an untracked variable is used in, say, an `IF` test, its value is treated as undetermined and both branches of the test are explored. Thus the model checker is always pessimistic; if insufficient variables are tracked, then the check gives a false failure, never a false success.

We eliminate the need for temporal logic by replacing the check of our program in

isolation with a check in an environment. For occam, we just join the external channels up to another occam program representing the environment. Then we typically need only two tests:

1. Safety properties are tested by checking if an error state can ever be reached.
2. Liveness properties are tested by checking if deadlock (`STOP`) can ever be reached.

We do not explicitly treat fairness. Ideas of *weak fairness* add a great deal of complexity to models of concurrency. We instead take the aggressive view that, for real-time codes, failure of any component to make progress within a fixed and finite number of *ticks* represents failure and causes a transition to the error state.

5. A model checking example: LTSA

The LTSA[8, 9] model checker is especially designed as a simple introduction to the technology. It is easy to use and can be run as an applet over the World Wide Web. A very simple example of its use gives some indication of what can easily be achieved, and where problems arise. Let us try to evaluate the correctness of the following code which purports to offer mutual exclusion between two threads. We define a turn variable:

```
int turn = 1;
```

and in each thread (me=0, 1 respectively), we guard the critical section as follows:

```
while (turn != me);
/* loop waiting for my turn */
/* CRITICAL SECTION */
turn = 1- me;
```

We model this system in LTSA's FSP language as the parallel composition of three machines, representing the turn variable and the two threads which share it:

```
TURN1 = (set0->TURN1|unset1->TURN0|isset1->TURN1),
TURN0 = (set0->TURN1|unset1->TURN0|isunset0->TURN0).
P0 = (isunset0-> claim0 -> release0-> set0->P0).
P1 = (isunset1-> claim1 -> release1-> unset1->P1).
||SYS=(TURN1||P0||P1).
```

This does not look much like the original C. Note also that we have replaced a busy polling implementation with a non-busy one. We have included claim and release transitions so that we can join up an environment which ensures that claims and releases can take place only in the permitted order. This new system, which checks the safety property of this mutex becomes:

```
TURN1 = (set0->TURN1|unset1->TURN0|isset1->TURN1),
TURN0 = (set0->TURN1|unset1->TURN0|isunset0->TURN0).
P0 = (isunset0-> claim0 -> release0-> set0->P0).
P1 = (isset1-> claim1 -> release1-> unset1->P1).
property MUTEX=(claim0->release0->MUTEX|claim1->release1->MUTEX).
||SYS=(TURN1||P0||P1||MUTEX).
```

Here the `property` keyword is a shorthand which adds extra transitions into an error state from all states on all events in the vocabulary of `mutex` which are not explicitly given transitions to other states. Thus the explicit transitions of `mutex` become the only safe transitions which the machine should accept. This system is accepted by LTSA as being without error or deadlock, so the safety property is true; the example *is* a mutex.

We can go on to check a liveness property. We require that the mutex be able to offer the (free) critical section to either thread. We check this by composing the mutex with code

which makes an *internal choice* about which thread will request the mutex. Possible deadlock would imply that the mutex fails to accept this imposed choice. In occam, such internal choice is easy to express; we just use an `ALT` with two `SKIP` guards. It's a bit trickier in LTSA; we need to represent in the LTSA language an ALT with two inputs on the same channel, then hide this spurious channel. In FSP, we get:

```
TURN1 = (set0->TURN1|unset1->TURN0|isset1->TURN1),
TURN0 = (set0->TURN1|unset1->TURN0|isunset0->TURN0).
P0 = (isunset0-> claim0 -> release0-> set0->P0).
P1 = (isset1-> claim1 -> release1-> unset1->P1).
EXT = (t->claim0->release0->EXT|t->claim1->release1->EXT)\{t}.
||SYS=(TURN1||P0||P1||EXT).
```

As the reader might expect, LTSA reports a deadlock when presented with this input; our crude mutex actually forces strict alternation and to do better we need Peterson's[10] algorithm.

So what has been learnt? First, crude tests (no temporal logic) are entirely adequate to test both safety and liveness. Second, even simple tutorial model checkers have an alien input language. Finally, occam is at least as expressive as FSP both for the system under test and for its environments.

LTSA is a very attractive system for learning about model checking. Checking software is, however, an intensive task and it is not appropriate to use a visual learning tool for this purpose. Production-quality model checkers are readily available [5,6,7]; SMV makes a good choice as it has been adopted by Cadence, thus improving our chances in the future of model checking across the hardware-software boundary.

6. Model checking occam

We have chosen to implement the model check as a new back-end for SPoC. This makes heavy use of the optimiser developed for the PIC back-end to simplify the generated SMV code and to reduce the amount of program state. As explained above, it is necessary to annotate the occam program with compiler directives to indicate which variables are to be tracked by SMV. This is done by introducing directives of the form

```
#PRAGMA TRACK x 0::10
```

into the occam source. This directive not only requests the tracking of variable `x`, it also asserts that `x` has range $0 \leq x \leq 10$. A violation of this range assertion will be flagged as an error by SMV. The generated SMV code is remarkably similar to that for the PIC in overall structure. For example, the *runnable* bits associated with code fragments on the PIC are simply combined to give a program counter per concurrent thread which increments only at deschedule points. The implementation of ALT depends on the version of SMV that is in use. There is a simple version for Cadence Berkeley SMV which uses guarded set expressions. Unfortunately, this version is not open source, and a rather more complex implementation with potentially many cases has to be used with Carnegie-Mellon's open source SMV.

As well as generating the SMV code for the model check, it is important that, in the event of failure of the check, the SMV history of the failure can be rendered in language the occam programmer can understand. This back annotation of SMV state into occam state is handled by a Perl script which maps the SMV program counters back into occam source lines. This trace history can then be followed to the failure. Information to drive this script is generated by SPoC. By default the model checker tests the Temporal Logic (CTL)

expression

```
AG AF termination
```

This expresses in temporal logic the assertion that the occam program will terminate. Deadlock and most forms of livelock will have the effect of making this test fail.

References

[1] M. Debbage, M. Hill, D. A. Nicole and S. M. Wykes, Southampton's Portable Occam Compiler (SPOC). In: R. Miles and A. Chalmers(ed.) *Progress in Transputer and OCCAM, Research (WoTUG-17)*. ISBN: 9051991630. IOS Press, Amsterdam, 1994.

[2] Information on PIC Microcontrollers may currently be found on the World Wide Web at: http://www.microchip.com/

[3] A. W. Roscoe and C. A. R. Hoare, The Laws of Occam Programming, Programming Research Group Technical Monograph PRG-53, ISBN: 902928341, 1986.

[4] K. L. McMillan, Symbolic Model Checking, ISBN: 792393805, Kluwer Academic Publishers, Norwell, Ma., 1993.

[5] Information on SMV may currently be found on the World Wide Web at: http://www-2.cs.cmu.edu/~modelcheck/smv.html or http://www-cad.eecs.berkeley.edu/~kenmcmil/

[6] Information on SPIN may currently be found on the World Wide Web at: http://spinroot.com/spin/whatispin.html.

[7] Information on FDR may currently be found on the World Wide Web at: http://www.fsel.com/software.html.

[8] J. Magee and J. Kramer, Concurrency: State Models & Java Programs, ISBN: 471987107, Wiley, Hoboken, NJ., 1999.

[9] Information on LTSA may currently be found on the World Wide web at: http:// http://www-dse.doc.ic.ac.uk/concurrency/ltsa-v2/index.html

[10] G. L. Peterson, Myths about the mutual exclusion problem. *Information Processing Letters 12*, 3, 115—116 (1981)

[11] The source code and working examples of the software described in this paper may be found on the World Wide Web at: http://www.hpcc.ecs.soton.ac.uk/software/spoc/