SVA, a tool for analysing shared-variable programs

David Hopkins and A.W. Roscoe[1]

Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD, UK

Abstract

In [6], Roscoe described a prototype compiler that allowed straightforward shared variable programs to be analysed using FDR, by writing a compiler in its CSP$_M$ language. This allowed, for example, a high degree of control over atomicity but lacked a proper input language and an interpreter for and counter-examples found. In this paper, we first propose a concrete syntax for the input language, and then describe a GUI which takes this as input, drives a modified compiler and FDR, and then provides a clear explanation of counter-examples in suitable format for users of the language.

Keywords: CSP, shared variable, FDR, verification

1 Introduction

The purpose of this paper is to describe extensions to Roscoe's *share2.csp* compiler, described in [6]. These extensions turn it from a difficult to use kernel into a tool that can be used straightforwardly in practise, teaching etc, and consider how it might be used to handle classes of infinite-state systems such as Lamport's Bakery Algorithm. We call this tool SVA, for Shared Variable Analyser.

share2.csp is not a compiler in the conventional sense of the word. Rather, it is a CSP$_M$ [5] program that takes a simple shared variable program and simulates its execution by creating a network of processes which run in parallel as a communicating-process model of the execution of the object program. Any compiler for CSP, such as that used by FDR, will therefore generate a model, for analysis or implementation, of the original program. Its input programs are represented in a CSP data type rather than ASCII. SVA allows a natural input language written in ASCII and, more importantly, interprets debugging output clearly and in a way consistent with that language.

[1] Email: bill.roscoe@comlab.ox.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

In this paper, which is a summary of a third-year undergraduate project undertaken by Hopkins while supervised by Roscoe, we give an introduction to the tool's background, structure and applications.

2 Background

We need to understand certain features of the *share2.csp* compiler. We give a summary here; a full description can be found in [6].

As already mentioned *share2.csp* – itself a CSP$_M$ script – takes its input in the form of a sequence of CSP$_M$ data structures, each one representing a single process. The main `Cmd` data-type of sequential programs is:

```
datatype Cmd = Skip | Sq.(Cmd,Cmd) | SQ.Seq(Cmd) | Iter.Cmd
             | While.(BExpr,Cmd) | Cond.(BExpr,Cmd,Cmd)
             | Iassign.(ivnames,IExpr) | Bassign.(bvnames,BExpr)
             | Sig.Signals | ISig.(ISignals,IExpr)
             | Atomic.Cmd
```

There are constructs for sequential composition, infinite iteration, while-loop, integer and boolean assignment, emitting signal events without or with integer data, and marking a command for atomic execution.

A complete program is a list of sequential programs, each representing a separate thread, together with declarations of the variables (which include arrays) and constants that are used in them. The default mode of operation is for the programs to proceed with their steps interleaved arbitrarily. The `Atomic.P` construct declared that the code `P` is to be executed without any of the other programs in the system doing anything until `P` is finished. Again, by default, the evaluation of an expression is carried out in a number of steps of acquiring values from variables and evaluating the expression. However the compiler allows a flag `atomic_exprs` to be set, in which case these evaluations are all treated as single steps. For example the expression term representing `x-x` will always evaluate to 0 under this flag, but not necessarily if `x`'s value can change between its two fetches. There are similar data types `IExpr` and `Bexpr` representing integer and Boolean expressions, with the expected operations represented by constructors.

The user of *share2.csp* is expected to declare a number of constants such as the numbers of Boolean and integer variables, and the number and sizes of arrays. SVA is able to compute several of these, but others still need to be declared at present. For example, while the type of integers is used, programs are only permitted to use the part of it between the constants `MinI` and `MaxI`. If a variable goes out of range at run time, the special signal `outofrange` occurs. Under normal circumstances the user will want to check (using FDR) that this does not occur during the execution of the program. The user must declare signals as CSP channels (null type or a subtype of integer as appropriate) and declare the sets of them to *share2.csp*.

The compiler creates one process for each variable, non-trivial expression and thread in the shared variable language. If atomicity is used, the it creates machinery to regulate how the threads behave relative to each others' atomic sections.

3 ASCII language

The ASCII syntax we devised used conventional notation for variable declaration, assignment, while loops and sequential composition, plus Boolean and integer operations within expressions. We added straightforward constructs for iteration, atomic evaluation, and signals. A complete description of the syntax can be found in the manual [2]. An easy-to-understand example program that calculates `gcd(18,15)` is

```
isig output;
int a,b,c;
P(m,n) = {a := m; b := n;
          while b > 0 do skip;
          isig(output, a)}

Q() = {iter if b > 0 then
              {c := a % b; a := b;b := c;}}

Prog = <P(18,15),Q()>
```

The parser/translator translates the first component of this to

```
P(m,n) = Sq.(Iassign.(I.1,Const.m),Sq.(Iassign.(I.2,Const.n),Sq.
     (While.(Gt.IVar.I.2.Const.0,Skip),ISig.(output,IVar.I.1))))
```

CSP (for example for creating specifications) can be included in a script by prefixing it with `%%`. This device can be used to include specifications written in CSP directly in the script. For example, using signals that represent the start and end of a critical section by a labelled process, mutual exclusion is expressed

```
%% Mutex = css?x -> cse!x -> Mutex
```

Two forms of specification have been included directly in the language. These are assertions respectively that a set `S` of signals never occurs, and that a Boolean expression `b` is always true. They are respectively written `assert nosignal S in Prog` and `assert always b in Prog`. We hope to extend these to a significant subset of LTL.

4 Trace interpreter and GUI

If a program fails its specification, we want to be able to understand the counterexample that our model checker provides. Unfortunately the counterexamples provided by FDR for *share2.csp* are extremely detailed, reflecting the innermost details of the translation of our little language to CSP, and replacing variable names by index numbers. As with Casper [4], we need a way of interpreting this output in a suitable language for the domain in which we are working.

SVA provides this, integrated into a simple GUI that manages the process of parsing, running and counter-example interpretation. With a choice of granularity, each action in the trace is given to the process that performed it and explained in

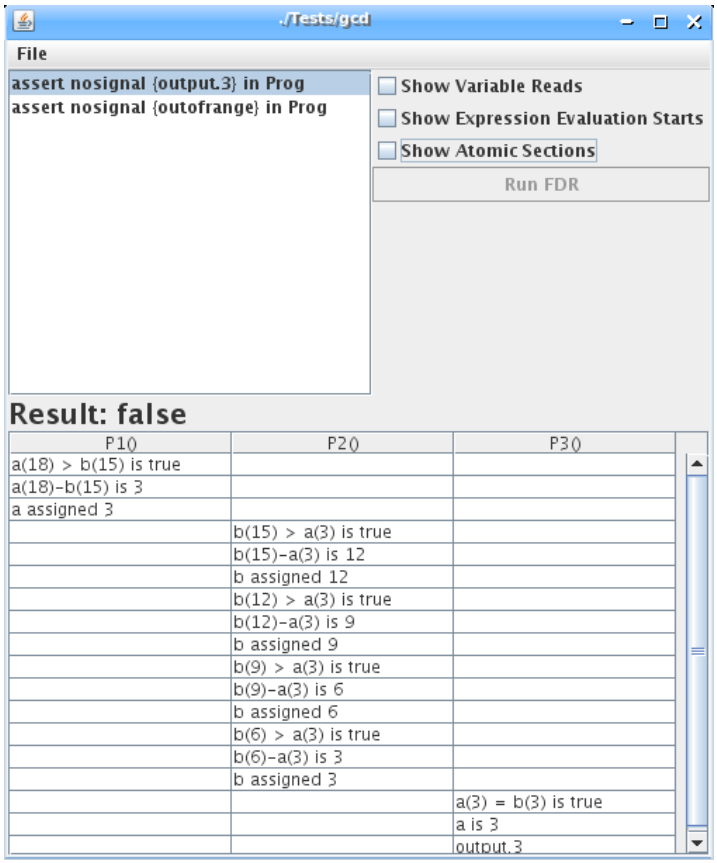

Fig. 1. SVA GUI screenshot

simple language. An example for our elementary gcd program is shown in Figure 1.

5 Advanced features

The biggest case study we have examined using SVA is Lamport's bakery algorithm [3] (a mutual exclusion algorithm). This cannot be modelled in its full generality because whatever bound on integers is chosen for SVA, it will be exceeded. The solution we adopted, and which we expect to present in full in a later paper, is a variation on the idea of data abstraction. Instead of running an integer-based program on SVA, we run one that records the relative order of the threads' ticket values (or, more properly, the n^2 values representing process i's view of process j's ticket) rather than their actual values. It is not standard data abstraction because the *successor* operation needs to be modelled nondeterministically: the position of the successor of a particular point in the order might either be the next already-recorded value, or might be strictly less than that value. Files implementing and explaining this can be found at [2].

It is a challenge implementing this abstract type within SVA. The approach we took was to allow the thread processes themselves to use integers, but to have a

separate monitor process that preserved the invariant that the ticket views that are non-special (see below) take a contiguous initial subset of the available values.

The standard version of SVA was not up to this since it interleaves the actions of the threads and the monitor arbitrarily, meaning that the invariant may well not get preserved promptly. We actually need to give the monitor priority, at least every time one of the ticket views gets changed. We implemented two alternative mechanisms to achieve this: the possibility of giving some threads higher priority than others, and a mechanism that runs an atomic monitor program every time one of a specified set of variables gets changed. We found the latter to be much more efficient. Details of these constructs can be found in the manual [6].

By this means we were able to provide yet another verification of the bakery algorithm for three nodes, unusual in the sense that it was a finite state model checking run. In total, 9 values of ticket type were required including the two special constants used by the algorithm to denote a node not in contention for a critical section, and one that is just beginning to seek one.

6 Conclusions

We believe that SVA should be a good tool for teaching low level shared variable programming, since it is easy to use and provides detailed feedback and explanations of the behaviours it finds. Files analysing a range of applications, including other mutual exclusion algorithms can be found at [6]. Its use of FDR is no obstacle since that tool is now freely downloadable for non-commercial use.

It illustrates the way in which tools can be created for other notations by translation into CSP and using FDR in the background. Casper [4] is the best known such tool. CSP's great expressive power often allows translations in ways that are economical in state space, and, provided variables take relatively limited ranges of values, simply providing a process to hold the value of each seems to work well. Another example of this is is the tool for Statemate Statecharts [1] described in [7].

References

[1] D.Harel, H Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman and A.S. Tauring, *Statemate, A working environment for the development of complex embeded systems*, IEEE Trans on Software Engineering **16**, 4, 1990.

[2] David Hopkins and A.W. Roscoe, *The SVA website*,

[3] Leslie Lamport, *A new solution of Dijkstra's concurrent programming problem,* CACM **17**, No 8 (1974)

[4] G. Lowe, *Casper: A compiler for the analysis of security protocols*, JCS **6**, 1, 1998.

[5] A.W. Roscoe, *The theory and practice of concurrency*, Prentice-Hall, 1997.

[6] A.W. Roscoe, *Compiling Shared Variable Programs into CSP*, Proceedings of PROGRESS workshop, Eindhoven 2001 `web.comlab.ox.ac.uk/oucl/research/areas/concurrency/tools/sva/`

[7] A.W. Roscoe and Zhenhzong Wu, *Verifying Statemate Statecharts Using CSP and FDR*, Proceedings of ICFEM 2006