Compiling shared variable programs into CSP

A.W. Roscoe
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
Bill.Roscoe@comlab.ox.ac.uk

Abstract— **We present a compiler from a simple shared variable language into CSP. This allows the application of CSP-based tools such as FDR when analysing programs written in the other language. The translation into CSP makes it easy to be flexible about the semantics of execution, most particularly the amount of atomicity that is enforced. We examine ways available to translate specifications we may wish to prove into the refinement checks supported by FDR, in particular looking at ways of proving liveness specifications under fairness assumptions. Various examples are given, including mutual exclusion algorithms.**

Keywords— **CSP; FDR; shared variable; compilation; mutual exclusion; model checking; fairness**

Contents

I. Introduction

The world of concurrency now has some powerful tools for the automated checking and verification of programs. Naturally, each tool has its own input language and tends to be optimised for the way that language expresses problems. Obviously this tendency towards narrowness makes the tools hard to compare and restricts their application. FDR [2], [10], the tool the author is most associated with, could be said to illustrate this problem particularly clearly. Its input language is Hoare's CSP [4] (extended by many of the capabilities of functional programming: the machine readable language is termed CSP$_M$ [2], [11], [14]) and its primary mode of operation is to check CSP refinement. FDR has proved very successful in the hands of people with a mastery of CSP, since that notation often has the ability to express problems remarkably succinctly and elegantly. Unfortunately, however, that is still not a widely-possessed skill. In order to make its power available to a wider audience, there have been a number of efforts in which input in other notations, for example UML-RT protocol and capsule state diagrams [17], StateMate StateCharts [3] and security protocols [7], has been processed for input into CSP.

By and large these alternative inputs have been from worlds where, as in CSP, processes largely communicate by passing messages along channels or at least it is natural to model and judge programs in ways that match the idea of CSP events. On the other hand, some of the best-known work on model checking (for example much of that in SPIN [5]) is done in the world in which programs communicate via shared variables instead.

The purpose of this paper is to report on a compiler which automatically translates programs in a simple shared variable language into CSP and allows them to be checked on FDR. It should not be surprising that this is possible in principle, since it has long been known (see [4], for example) that one can model a variable as a process parallel to the one that uses it. The user process then reads from, or writes to, the variable by CSP communication. There is no reason why there cannot be several processes interacting in this way with the same variable, thereby sharing it. It might be thought (and this was certainly in the author's mind on first contemplating the idea) that generating lots of extra parallel processes in this way would be enormously inefficient. Doubtless that is true in most practical models of parallel implementation, but it turns out to be far from true in the world of FDR, at least provided that the number of variables is not huge. An interesting comparison is provided by the “lazy spy” processes used in modelling cryptographic protocols [12], [13], [11], where it turns out to be far more efficient to use networks of many (typically hundreds) of simple parallel processes than a very complex sequential one.

The reason for this is that FDR is optimised to handle networks built out of component processes with a moderate number of states that can be pre-computed before beginning to enumerate the state-space of the complete system. In the case of imperative programs, it is therefore actually helpful to separate the state needed to model the variables into separate processes from that required to model the control flow of the overall program. This means that we have a fortunate coincidence between the long-known method of modelling imperative programs, and the pragmatics of modelling things efficiently for FDR.

The compiler itself is written in CSP$_M$, and defines functions which, given a shared-variable program, produce a parallel CSP program that implements it. The text of the expanded CSP program is not produced explicitly: rather it exists in a virtual way within the CSP compiler resident in FDR. The compiler is called `share2.csp` and to use it you `include` it in highly stylised `.csp` files which are loaded into FDR but themselves contain little or nothing that looks like CSP, only a representation of the shared variable program and (usually) specification information.

At present `share2.csp` is just a proof of concept, with a very simple structure of data types. There is certainly no attempt at present to rival either the breadth or efficiency of the established model checkers for this type of system such as SPIN. The work in this paper should be viewed either as an interesting exercise paving the way for a further developments, or perhaps as a case study in coding using CSP$_M$.

This paper is organised as follows: the next section defines the input language and describes how programs in it are translated into CSP; we then address the issue of specification and how the output from the compiler can be customised to help in this process; finally we discuss a few simple case studies that illustrate how the compiler has been used.

The compiler and a range of case studies are available via `http://web.comlab.ox.ac.uk/oucl/research/areas/concurrency/tools.html`. Quite a lot of what follows explains the basics – and more interesting details – of what the compiler does. A reader interested in other details should look through the text of the compiler.

Acknowledgements

I am grateful to Michael Goldsmith, David Walker Ranko Lazić and Steve Brookes for discussions on this work, and to Formal Systems for implementing the special check discussed at the end of this paper.

This work was funded by grants from EPSRC and US ONR.

II. The language and compiler

A program comprises a list of parallel processes and an initialisation of variables. Each sequential component is taken from the following syntax:

```
Cmd ::=
  Skip | Cmd;Cmd | Iter(Cmd) |
  While(BExpr,Cmd) |
  if BExpr then Cmd else Cmd |
  iv := IExpr | bv:= BExpr|
  Sig(Signals) | ISig(ISignals,IExpr) |
  Atomic(Cmd)
```

The constructs here that are not immediately obvious are `Iter`, which runs a piece of code over and over for ever, `Sig` and `ISig`, which have the effect of outputting signals to the outside world, and `Atomic`, which ensures that whatever command is inside it is executed atomically. All of these will be discussed further later. `BExpr` and `IExpr` respectively represent boolean and integer expressions, and `bv/iv` boolean and integer variables. Integer expressions are formed from constants, variables and the obvious operations of addition, subtraction, multiplication, division and remainder (mod). Boolean expressions are built from constants, variables, `Not`, `And` and `Or`, and comparisons between integer expressions.

Separate initialisation of variables is needed because of the way in which processes share variables: we want to ensure that the initialisation takes effect before *any* process reads a given variable. Two of the parameters given to the compiler are default initial values for boolean and integer variables respectively; a variable therefore has to be explicitly initialised if we want to give it a different value. The initialisation component of a program therefore provides lists of the boolean and integer variables that are non-standard, together with their values.

Since CSP$_M$ has no string-handling capability, it cannot be given long pieces of text representing a program and be expected to parse them. Therefore the CSP$_M$ compiler only gets going at the point where the program has been converted into a member of a data type whose construction reflects the structure of our language. Therefore we will assume that our starting point is a CSP$_M$ script in which the program is defined as a member of such a type. It should be straightforward to create a parser which inputs programs in a more conventional format and outputs scripts of this form.

The following are the data types of commands (i.e., sequential program threads), and integer and boolean expressions on which the compiler operates. Obviously, `Cmd` is just a slight adaptation of the syntax at the start of this section, except that both binary and list-based sequencing commands are present to give the user some flexibility.

```
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

datatype IExpr =
IVar.ivnames | Const.{MinI..MaxI} |
Plus.IExpr.IExpr |
Minus.IExpr.IExpr | Times.IExpr.IExpr |
Div.IExpr.IExpr | Mod.IExpr.IExpr

datatype BExpr =
BVar.bvnames | True | False |
Not.BExpr | And.BExpr.BExpr |
Or.BExpr.BExpr | Eq.IExpr.IExpr |
Gt.IExpr.IExpr | Ge.IExpr.IExpr
```

The values of the integer type are limited to being between two user-defined limits. In practice one usually cannot make these limits very far apart – the default is {0..20} – because of the state explosion problem. These same limits are applied by the compiler (both at compile and run-time) to all integer values that arise during computations. This is an issue we will return to later.

The types of boolean and integer variables, namely `bvnames` and `ivnames` contain two sorts of object: basic variables `BV.i` and `IV.i`, and components of the arrays `BA.i` and `IA.i`. These components `BA.i.j` and `IA.i.j` are members of the sets `bvnames` and `ivnames`. At present, the main purpose of arrays is to provide variables for each of a list of processes indexed by some parameter. The value of each array index used is determinable at compile-time when the system is laid out. It would not be a difficult step to extend this to arrays where the indices were computed at run-time.

The first stage the compiler goes through is analysing all the thread processes to see what variables they use, so that a separate process can be created to hold the value of each. Here, each accessed component of an array is treated as a separate variable and gets a separate process. Note that if arrays were to be extended to allow indices to be calculated at run-time, then a value-holding process would have to be created for each array component independently of whether it was actually going to be used.

Variables can be used by programs in one of three ways: they can be written to, they can be read from as part of the calculation of an expression that is more complex than just the variable, or they can form a value-expression in their own right. We distinguish the second and third of these because of the way we handle expression evaluation: instead of this being done by the thread process itself, any expression that is more complex than either a constant expression or an unaltered variable is delegated to a special parallel process to evaluate. This has the advantage of separation of concerns, greatly simplifying the CSP code (and hence, quite probably, the compile time) of the thread processes. Therefore the analysis phase identifies all the non-trivial expressions in the thread programs so that a calculator process can be created for each. No attempt is made at present to identify re-used expressions. While this would certainly be harmless in the case of expressions within a single thread process, there would be the potential for subtle semantic effects if a calculator process for an expression shared by two threads were shared.

A calculator process simply collects all the variable values it requires after being prompted by a thread process to perform its task, and then returns the appropriate computed value.

Having performed this analysis, the compiler then sets up a network of CSP processes to run the program: one for each thread process, one for each variable (naturally, potentially shared between the threads) and one for each of the expressions it has identified. There are two main options on how the network is structured: for simplicity here we describe the original version, and will later discuss the amendment (which provides for a different model of expression evaluation). Except for the mechanisms that may be needed to implement atomicity, which we will discuss later, the thread processes do not communicate directly with each other, but rather with the calculator and variable processes. The CSP structure of the resulting process in the absence of any `Atomic` constructs is:

```
((Threads [|{|ieval,beval,ireq,breq|}|] Exprs)
[|{|iveval,ivwrite,bveval,bvwrite |}|] Vars)
```

where each of `Threads`, `Exprs` and `Vars` is the interleaving (`|||`) of all the processes of the given type identified during the analysis phase. The channels synchronised on above are

- `ireq` and `breq` are used to trigger expression calculators (respectively integer and boolean) to do their tasks. These channels are parameterised by indices automatically generated at the analysis phase.
- `ieval` and `beval` respectively return the values calculated by the expression calculations: they are parameterised by the same index and the value.
- `iveval`, `ivwrite`, `bveval`, `bvwrite` allow reads and writes from variables. Note that both thread and expression processes are allowed (by the above construct) to communicate with the variables using these, though the expression calculations only actually require reads, of course.

With the exception of the mechanisms to support atomicity, the coding of the various constituent processes is not very complex. For example, an integer variable process (name `x` and value `v`) is just

```
IVAR(x,v) = iveval.x!v -> IVAR(x,v)
         [] ivwrite.x?w -> IVAR(x,w)
```

An expression calculator works by first inputting the variable values it requires (noting that a boolean expression – the sort shown – can contain both integer and boolean variables), substituting the values for the variable names, outputting the evaluated expression when all this is done, and then beginning again. In the following definition the parameter `n` is the automatically assigned index for the expression; `e` is the expression itself.

```
BExprProc(n,e) =
(breq.n -> instantiateb(e,n,varsb(e)));
                                BExprProc(n,e)
```

```
instantiateb(e,n,(<>,<>)) =
let v=evaluateb(e) within
  if bok(v) then beval.n.bval(v) -> SKIP
            else outofrange -> STOP

instantiateb(e,n,(<x>^xs,ys)) =
bveval.x?v ->
instantiateb(subsb(x,v,e),n,(xs,ys))

instantiateb(e,n,(<>,<y>^ys)) =
iveval.y?v ->
instantiateb(subsbi(y,v,e),n,(<>,ys))
```

Any expression that generates an out-of-range integer value halts its evaluation with the error event `outofrange`.

The analysis phase also identifies whether we need to compile our processes with support for atomic execution, and produces a slightly modified syntax for the thread processes. The main modifications are to analyse each value that is required by the program to see if it is a constant, an unaltered variable, or needs to be calculated, and produce syntactic variants accordingly (in the case of a calculation, the expression itself is replaced by the name of the process to which evaluation is delegated). So, for example, we get two clauses of the function `MainProc` which maps structures in the modified syntax to their CSP implementations. One is for for while loops in which the guard is a variable:

```
MainProc(WhileV_.(b,p)) =
(bveval.b?x ->
if x then
  (MainProc(p);MainProc(WhileV_.(b,p)))
else SKIP)
```

and in which it is an expression index `b` that has to be calculated:

```
MainProc(WhileE_.(b,p)) =
         (breq.b -> beval.b?x ->
           if x then
                 (MainProc(p);
                  MainProc(WhileE_.(b,p)))
                else SKIP)
```

The infinite iteration construct `Iter.P` is simpler because no evaluation of the boolean is required:

```
MainProc(Iter_.p) = MainProc(p);
                    MainProc(Iter_.p)
```

and any while loop that happens to have a constant boolean guard can be compiled to either this or to SKIP!

We have included signal events in our syntax to aid in the specification and understanding of the programs we write. They are intended purely as communications from the thread processes to the external observers of the system: no mechanism is provided for one thread to listen to another one's signals.

Two sorts of signals are provided in the syntax: one for the execution of an unparameterised action, which is trivially implemented in CSP:

```
MainProc(Sig_.x) = x -> SKIP
```

The other does the same, but allows for the output of an integer variable (which has the same options for how to work it out: constant, variable or calculated expression). Since a signal naturally parameterised by a boolean can be replaced by a pair of signals, in this prototype these were not supported explicitly.

With the exception of Atomic.P, a reader familiar with CSP should now find it easy to follow the rest of the clauses of the function MainProc, which implements the thread processes recursively over syntax.

Example II.1 Mutual exclusion The initial examples experimented with on the compiler were mutual exclusion algorithms: ones designed to ensure that, of several processes that might potentially want to execute some critical section, no more than one does at a time. A collection of these, all presented in an input language similar to ours and hand-translated into CCS, can be found in a paper by Walker [16].

One of these, Hyman's algorithm [6], does not preserve mutual exclusion. It involves two processes indexed 1 and 2 which use boolean variables b[1] and b[2] and an integer variable t ranging over $\{1,2\}$. If we add the CSP signals css.i and cse.i respectively to denote the beginning and end of process i's critical section, then it can be written (i being "this" process and j=3-i being the other one):

```
while true do
{b[i] := true;
 while t!=i do
 {while b[j] do skip;
  t := i};
 css.i; cse.i;
 b[i] := false}
```

In the language of the compiler this is

```
b = BVar.BA.1
t = IVar.I.1

P(i) = let
         j = 3-i
       within
       Iter.(
        SQ.<bassign(b.i,True),
            While.(Not.(Eq.t.Const.i),
                  Sq.(While.(b.j,Skip),
                      iassign(t,Const.i))),
            Sig.(css.i),
            Sig.(cse.i),
            bassign(b.i,False)>)
```

Note that the declarative nature of the underlying CSP$_M$ language (of which this last excerpt is an example) allows us to give one of the boolean arrays and one of the integer variables the names from the earlier program fragment. The functions bassign and iassign are shorthand for operations that perform boolean and integer assignment on objects like b.1 and t. It may look strange to see Const.i, but of course by the time i is instantiated when the network is constructed it really is a constant (1 or 2).

If the compiler is run on the processes <P(1),P(2)> with the b.i initialised to false and t to 1, we get a network of seven processes: two threads, three variables and two expression calculators (for the guards Not.(Eq.t.Const.i), $i \in \{1,2\}$). We can specify that the mutual exclusion property is achieved by performing a trace check against the CSP specification

```
SPEC = css?i -> cse!i -> SPEC
```

after hiding all of the non-signal events the processes perform. This check fails and generates the following trace (after the hidden events are restored)

```
<bvwrite.BA.1.2.true,
breq.BE.1, iveval.I.1.1, beval.BE.1.true,
bveval.BA.1.1.false,
bvwrite.BA.1.1.true,
breq.BE.2, iveval.I.1.1, beval.BE.2.false,
ivwrite.I.1.2,
breq.BE.1, iveval.I.1.2, beval.BE.1.false,
css.1, css.2>
```

This shows that if P(2) starts off, setting b[2] to true and finding that its boolean guard t!=2 (confusingly numbered boolean expression 1 by the compiler) is true, it enters its inner loop whose guard (b[1]) is false, so the loop terminates. Before `P(2)` does anything else, `P(1)` sets `b[1]` to `true` and is allowed to terminate its outer `while` loop immediately because `t` is 1 initially. `P(1)` is already in the position to begin its critical section, as is `P(2)` after it has set `t` to 2 and discovered that its outer `while` loop now terminates. Both processes can now enter their critical sections, which violates the specification.

There is, of course, nothing original in this flaw in the algorithm: it is presented simply as an illustration of how the debugging output from FDR allows us to understand what has happened in the original code.

In future developments, it ought to be possible (as has now been done with Casper [7]) not only to input programs into the compiler in a more natural syntax, but also to have a post-processor for debugger output that translates traces like the above into things that more obviously relate to actions of the original programs.

A. *Atomicity*

Where several programs are operating on the same set of variables, there is obviously a considerable potential for them to interfere with each others' actions. This is often desirable, since it provides the mechanism by which processes can talk to each other. On the other hand, it is easy to be surprised by the sort of consequences this interference can have. Consider, for example the consequences of running the following parallel programs:

- `x := y` in parallel with `y := x`,
- `z := x+y` in parallel with `x := x+1;y:=y-1`.

Depending on how the operations of evaluating the expressions on the right and the writing-back of the results take place, the first can have three results: `x` and `y` both taking the initial value of `x`; both taking the initial value of `y`, or having them interchanged. Executing the second pair can have the paradoxical result that `z` ends up with a value less than the sum of the initial values of `x` and `y`, even though 1 certainly gets added to `x` before it gets taken from `y`. This happens when the first program reads the value of `x` before the second has done anything, and the value of `y` after it has finished.

In order to avoid such surprises, it is sometimes necessary to specify that certain parts of a thread process's behaviour are *atomic*, namely that they take place as though they were single actions. Any atomic part of a program will never have effects from other threads appearing part way through, and neither will any other thread see any of the intermediate states it might create. Thus if we declare the program fragment

```
  x := x+1; x := x+1
```

to be atomic, then it is equivalent in its behaviour to `x := x+2`, whereas, run in parallel with

```
  y := x; x := 3
```

a non-atomic version could behave very differently.

In practical terms, atomicity comes at a cost, since it requires interlocks between the thread executions, which might well slow a program down, and may well lead to more complex implementation strategies.

The compiler has two different strategies for supporting atomic execution: there is a boolean parameter which will cause all expression evaluations to be atomic (i.e., no write to a variable that is read by an expression calculator happens during the evaluation), and there is the `Atomic.P` construct in the command syntax. The first of these is easy to implement as part of the CSP constructed by the compiler: the network of threads, expressions and variables is augmented by a new process

```
AtomicExps = ivwrite?_ -> AtomicExps
       [] bvwrite?_ -> AtomicExps
       [] breq?x -> beval.x?_ -> AtomicExps
       [] ireq?x -> ieval.x?_ -> AtomicExps
       [] start_atom?_ -> AtomicExps
       [] end_atom?_ -> AtomicExps
```

which synchronises with the network on the events it uses, thereby preventing writes to variables between an expression evaluation starting and finishing. The last two lines will be explained shortly.

Whenever there is an `Atomic.P` construct anywhere in the program, each thread process has a regulator process added to it (running in parallel). This enables us to prevent any more than one thread being in an `Atomic` at once[1], and prevents the thread process it governs doing anything of significance while another process is in an atomic phase. Note that this is a rather stronger statement than that atomic phases are critical sections in the language of mutual exclusion. In the following definition of the regulator process, `rest` is an action that is subsequently renamed to all

[1]Allowing this would not only be somewhat counter-intuitive, but it can also introduce deadlock.

reads and writes that this thread might do, plus all the ireq and breq commands to start expression calculations. The parameter d is the maximum depth of nested Atomic constructs that appear in the program[2].

The following definition is of the regulator for thread 0. The rest are generated by renaming in the compiler.

```
AtReg(0,np,d) =
   rest -> AtReg(0,np,d)
[] start_atom?i:{1..np-1} -> AtReg'(i,np,d,1)
[] start_atom.0 -> AtReg(0,np,d)
[] end_atom.0 -> AtReg(0,np,d)

AtReg'(i,np,d,j) =
   j < d & start_atom.i -> AtReg'(i,np,d,j+1)
[] j > 1 & end_atom.i -> AtReg'(i,np,d,j-1)
[] j == 1 & end_atom.i -> AtReg(0,np,d)
```

To mesh with this, the corresponding clause in the definition of the thread process is as follows:

```
MainProc(Atomic_.p) =
start_at -> MainProc(p);end_at -> SKIP
```

In thread i, these two actions are respectively renamed to start_atom.i and end_atom.i.

When these processes are present, the network is re-structured so that each thread is combined with all the expression calculators it uses before the above regulator process is added. That is because the regulator process needs to prevent these reading variables when another process is in an atomic phase.

It will always be true that making sections of a program atomic will create a refinement, in the sense that (aside, obviously, from the special actions which implement atomicity), there is no sequence of actions possible after making a section atomic that is impossible without.

Care has been needed to prevent the two different forms of atomicity interfering with each other to cause deadlock. That is why the process AtomicExps stops processes entering an Atomic section while an expression calculation is under way, and why it is important that while one process is in an atomic section, no other must be allowed to request an expression calculation. Without these things, deadlock can easily occur.

From a practical point of view, one would hope to keep the use of Atomic limited, and have it used on only small sections of programs (assignments or small groups of assignments, for example).

B. Alternative model of expression evaluation

The model of evaluation above is necessary when we do not want expression evaluation to be atomic, since we then have to see in feedback how the multiple reads[3] of a particular one interleave with writes. However, in the case where it is atomic, the sequence of request, multiple reads and a reporting-back of the result is arguably at too low a level. Certainly the traces reported by FDR are substantially increased in length, and the number of states found in the main check increased, by this detail.

To allow the user to avoid these problems, an alternative compiler is provided in which each expression calculator holds the value of each relevant variable so that it is constantly in a position to report the value of the expression. A calculator process thus listens to all writes to these variables. On the other hand it no longer has to instigate reads of the variables, and since it is always up-to-date the initial request messages are no longer needed. The act of a thread process evaluating an expression thus becomes a single action rather than two plus the number of variables.

Obviously this means the plumbing of the overall network is now different, since the calculator processes now have to synchronise on appropriate write actions and other connections are no longer needed:

```
Exprs =
(|| (n,e):IEs @ [IEAlpha(n,e)]
                IExprProc(n,e,vs(e)))
[AllIEAlphas||AllBEAlphas]
(|| (n,e):BEs @ [BEAlpha(n,e)]
                BExprProc(n,e,bvs(e)))
```

The new structure of how the expression calculators are set up is above: each has its own alphabet and

[2]There is no point in nesting them from a programming point of view, since the outer one will completely subsume all those inside it. Since, however, there is no obvious reason why nesting should be forbidden, we have to count how many atomic sections are entered and left.

[3]In fact, the definition given in the compiler is arguably not elaborate enough to deal with this problem in full generality, since the compiler chooses a specific order in which the various reads happen, and will therefore only detect problems that appear when this order is followed. If an implementation were to perform the reads in some other order, then it might well uncover a problem that checking the output of the compiler did not. The solution to this, namely exploring all possible orders of reading the variables for each expression, has not been followed because it is very dangerous in terms of state explosion.

the final parameters are mappings from the relevant variables to their initial values.

The advantages of the new model are as anticipated: namely fewer states and shorter traces. Aside from losing the flexibility over atomic expressions, the main disadvantage is a usually more complex compilation phase to the FDR checks.

The current intention is to retain both the original process model and this new one as alternatives.

III. Specification

The conventional way of specifying a system in the context of FDR is to design a specification process which it must refine, in the sense of traces, stable failures, or failures-divergences.

The fact that our language allows for the communication of signal events means that we can use the same general model. This was done successfully in the case of the mutual exclusion example given above, but there are factors which mean that we need to be careful before thinking of it as the solution.

- While for simple specifications such as mutual exclusion or the avoidance of some error event, the construction of a simple trace specification to check should present no problem to most users, it goes against the spirit of what we are doing to expect users to construct anything more than extremely simple CSP specifications.
- Given the nature of the language we are presenting systems in, it is quite likely that users will wish to formulate specifications in terms of the variables of the programs.
- The different model of computation means that ideas such as refusals and divergences need to be rethought, rather than have the user simply apply the CSP understanding of these things.
- It is much more likely that we will want to make *fairness* assumptions about process execution in this new language than in CSP.

Let us consider the statement made above about failures and divergences. The roles of failures in CSP is to detect points in a process's history where, possibly because of internal synchronisation needs within the process, it is unable to accept some set of events that is offered to it. Now processes in our new language do not really interact with their environment: all they do is go through the steps of a computation and perhaps emit signal events that we would expect the environment to observe rather than influence. Nor is there any way in which a process can get stalled (in the sense that it simply sits and does nothing) waiting for another one. Each thread process, unless it is held up by another one doing something atomic, can always perform its next action until is has terminated. All of that means that the concept of a refusal set is of very questionable use in specifying processes in our language.

In usual CSP analyses, divergence is considered deadly. In contemplating parallel systems built in our new language, with all the "internal" (i.e., non-signal) events hidden, one could almost say that it would be unusual to find one that could not diverge! So why this contrast? It is actually the flip side of the discussion in the preceding paragraph: because our language provides no synchronisation mechanism between parallel threads, it is common for processes to enter a waiting loop rather like the one we see in the mutual exclusion program above:

```
while b[j] do skip
```

simply marks time until the boolean `b[j]` is unset by another thread. Obviously, if the process with this loop got eternally scheduled and the other one never was, this would certainly lead to divergence. At least, FDR would certainly find a divergence, though we might well want to assume that in reality our system is implemented *fairly*, meaning that every thread process that will always eventually get to perform an action, and in many cases that sort of assumption would eliminate this sort of divergence. (To a large extent it could be said that a divergence caused by waiting loops in a fair implementation is the way that systems behave that would deadlock if written in CSP: imagine the five dining philosophers, each in a loop waiting for a fork to be put down.)

Thus divergence checking can, in appropriate circumstances, tell us useful things about processes written in our language, but we have to be careful to understand just what a given check is telling us. We will return to this subject later when we come to discuss fairness.

A. Safety checks

In many formalisms supporting specification, it is natural to discriminate between *safety* and *liveness* conditions. The former say (in some sense appropriate to the model) that the target program never takes an action which is either wrong in itself (in models where we judge processes by their actions) or which leads to a state that is demonstrably incorrect (where they are judged by state). Thus a program that never does anything at all will satisfy every safety condition in most formalisms. Liveness conditions are ones that specify that progress is made. For example, in the world of mutual exclusion, the statement that there are never two processes simultaneously in a critical section is a safety condition. On the other hand, saying that whenever a process wants to enter a critical section then it eventually does so is a liveness condition.

If using FDR, the only way to judge a safety condition is via a trace check, but this might be as simple as the claim that the event `error` never occurs. If there is any claim you wish to make about the values of the variables of a program which can be translated into a boolean expression `b` in the language syntax, then adding

```
Atomic.Cond(b,Skip,Sig.error)
```

as an extra parallel thread will throw up the `error` event just if the values of the variables of the program can ever reach a state in which `b` is false. (Since FDR checks all execution paths, the fact that in any single one `b` is checked only once does not invalidate this: if `b` ever were false then there is an execution of the augmented program in which it is evaluated at precisely that moment.) We might term this a *monitor* process since it does not change the state, but merely watches.

For example, we could replace the current signals in the mutual exclusion example by each process having a boolean variable, which it sets when it enters a critical section and unsets when it leaves. We would the simply perform the above check with the assertion that there is never more than one such boolean true.

If the condition is one which is only supposed to be true in certain control states of the thread processes, then something has to be done so that the assertion knows when it is meant to hold. If it only depends on the state of a single process, then the simplest thing to do may well just be to insert an assertion into that point of the relevant thread. Otherwise it may be necessary to add variables to the thread processes simply for the purpose of determining when an assertion is supposed to hold. Actually, the new variables introduced for mutual exclusion above can be regarded as falling into this category: obviously they both signal particular control states of the programs they belong to, and we could say that the assertion `False` has to be true whenever both programs are in a relevant state. This may seem a little contorted, but there are more natural applications of this idea. For example, if the processes represent bank accounts and the overall program contains a mechanism for transferring money between these, then we may want to specify that the total amount of money in the accounts always equals the initial quantity, provided that no transaction is currently active. Thus we might have, for each account, a `quiescent` variable, and ensure that they sum up correctly when these are all true.

A thread process that is monitoring the state of a system and implementing a safety check can evolve, meaning that the condition it demands of the system changes with time. Two possibilities here are:

- The monitor might read one or more variables at one point in the program and use these values in making a claim at a later time. For example, we could specify that the variable `x` is non-decreasing with the monitor

```
  x1 := x;
  Atomic.(if x1 > x then Sig.error else Skip)
```

where `x1` is a variable used only for the monitor.

- We could say that condition `b1` must be true for all times up to a time when `b2` becomes forever true as follows:

```
   Atomic.(if b1 then Abort else Skip);
   Atomic.(if b2 then Skip else Sig.error)
```

where `Abort` is a command like `while true do Skip` which never terminates or does anything interesting. The point here is that if `b1` were at one moment false and at that same moment or a later one `b2` failed, then the above would signal error. The check succeeding therefore means that `b2` must be true at the first moment (if any) when `b1` fails, and for ever after.

If one wanted to make an assertion about the final state of a program (that terminates, which not all sensible programs do in this world), you need a monitor process that knows when all the other threads have finished. There are two real options here: instead of running the monitor in parallel with the rest of the the threads it could be run in sequence (though still in parallel with the variable processes). That causes the immediate problem that the compiler does not presently support such structures, so it would have to be modified. Also, it would then be problematic linking the final assertion to things that may have been observed earlier about the system. The other option, and the one we recommend for now at least, is to use a new variable to determine when all the ordinary threads are finished: if `running` is a new variable initialised to the number of ordinary threads, and each thread `P` is replaced by the process `P;Atomic.(running := running - 1)` then the parallel monitor

```
Atomic.(if (not b) and (running = 0)
        then Sig.error else Skip)
```

will discover if the system can ever terminate without satisfying `b`.

In conclusion, to specify a safety condition there are two basic approaches available: the first is to insert signal events and use a traditional CSP trace specification process with a trace refinement check. The other is to insert a monitor process which keeps an eye on the developments (specifically, the values of the shared variables) within the program, bearing in mind that we may have to add special variables into the threads to help it. All one then does is to get the monitor process to raise a flag via an error event if the specification is violated, and test for this via a simple traces check (along with other errors like integer overflow that may be possible).

B. Counting computation steps

In specification languages like Linear Temporal Logic (LTL) used as a vehicle for defining properties of this general category of program, there is often a concept of computation steps. In other words, a specification will often define relationships between one state, and what becomes of it after the next action. In the same sense that proponents of true concurrency worry about whether linearised transition system semantics for process algebras are realistic, the author does not find the concept of "next state" an especially attractive one in a context where there are a number of unsynchronised parallel processes all running together. Nevertheless it can be useful to get a handle on the number of computation steps that individual thread processes and the entire system have made, and this handle or a variation upon it ought to be able to be used for the type of specification referred to above. It is certainly vital for getting anywhere with the idea of liveness specification, as we shall find below.

In most presentations of LTL, the judgement of what a computation step is is made via some operational semantics. While our mapping into CSP certainly does provide an operational semantics for programs in our language, it could reasonably be argued that the steps this implies (particularly in the version in which expression evaluation gets divided into several actions) are often too fine-grained. If we are dividing a behaviour up into steps, it seems logical to adopt the following pair of principles:

- Every distinct state (i.e., mapping of variables to values), should appear: no step must not be so large that such a state should be missed.
- If the program goes on running for ever, even without changing the state, this should show up as infinitely many steps.

The first of these two principles is safeguarded if we associate a computation step with the completion of each assignment, for it is only assignments that change the state. To achieve the second we need to associate a step with enough other sorts of commands that no program can go on for ever without executing an infinity of them. To do this it is sufficient to associate a step with each signal and each execution of `Skip`.

In order to be able to make use of these execution steps in specifications, we need to ensure they are visible, and in addition (especially when considering fairness) to know which thread process has performed a given action. There is therefore a modified version of the compiler which inserts the extra action `step.n` on each `Skip` performed by thread process `n`, and adds a tag to the write and signal actions so we can see from the outside who has performed each action.

This allows a number of interesting types of specifications to be checked which relate to what must – or what must not – have occurred within the program after a given pattern of computation steps. One of these is a quantitative approach to fairness that is discussed in the next section.

C. Liveness

We might divide liveness specifications up into two sorts: ones where, if satisfied, we know this at some finite point in the process's execution, and ones which can seemingly only be judged in terms of infinite behaviours. An example of the first sort would be "if process A wishes to enter a critical section, then eventually it does so". An example of the second would be "if process A sends infinitely many messages to B, then infinitely many get through". Sometimes ones of the second sort are equivalent to ones of the first: if you think about it, the second statement above is equivalent to the statement "if, starting from any state, A sends an unending sequence of messages to

B, then eventually at least one of them will be delivered". For if A puts infinitely many in, then by some finite point one will have been delivered, and starting from that point A will still put an infinite number in; so a second one will emerge, and so on.

It only really makes sense to consider ones that are of the first sort, since otherwise there is not really much hope that we will be able to resolve them finitely. Indeed, this is a very common assumption to make: many researchers take as their basic hypothesis that a safety property is one that is a closed set in a topological sense, and that a liveness one is an open set. In common parlance this really means that if some behaviour of a process fails a safety property then this will show up in an arbitrary finite time, and that if a given behaviour of it satisfies a liveness property then that too will show up in a finite time. Unfortunately there is an asymmetry here that makes liveness conditions harder. The problem is that for a whole process to fail a safety condition it is sufficient that a single behaviour does, so the failure of the whole process shows up finitely. On the other hand, for it to succeed in a liveness condition, the lengths that we have to look at its probably infinite range of behaviours to judge this may well not be bounded.

In a CSP context, the idea of liveness really breaks up into two parts: the finitary concept of a *failure* – if the current trace is s and the failure (s, X) does not belong to the process, then we can guarantee that something will be accepted if the process is offered X – and the infinitary possibilities of divergences and infinite traces. Failures are not that relevant to the models our compiler builds, so we we concentrate on the latter. By and large we can expect to have to show either that there are no infinite traces for a given program (i.e., it is guaranteed to terminate), or that all infinite traces show some measure of progress which must appear finitely.

In either case this can usually be reduced conveniently to the absence of divergence: if hiding either all events, or all events other than some which (either naturally, or because they have been inserted for this purpose) represent progress creates no divergence, then the respective properties above are satisfied.

These ideas apply to the world of the compiler, though it is worth noting that in order to prove that some condition `b` on the state is eventually satisfied in any infinite behaviour it is not sensible – without extra machinery that we will shortly introduce – to try to prove that the original program with the addition of the extra thread

```
While.(Not.b,Skip);Sig.Success
```

creates a system which, hiding everything prior to `Success`, is divergence-free. There are two related reasons for this:

- This monitor process might effectively exclude the rest of the program, preventing it making the progress we are analysing for, or
- the monitor might never get to perform the vital test once `b` is established by the rest of the program.

In both these cases, the strategy became the victim of unfair executions: one or more of a set of parallel threads not getting infinitely many turns to do something in an infinite time.

Very often, however, we have to rely on fairness not so much for detecting progress as for achieving it in the first place. This might be because of the phenomenon of processes obviously stuck in a waiting loop as discussed earlier, or for more subtle reasons. The point perhaps is, that if someone writes a program in a parallel language in which no non-terminated process (at least superficially) is ever unable to proceed, then it is not unreasonable for him or her to assume that no process is ever going to be delayed indefinitely by the implementation. Put another way, it seems reasonable to discard all infinite behaviours in which one of the thread processes only has a finite number of actions even though it never terminates.

In the context of the style of liveness specification discussed above, it means we should only consider a given divergence to be an error if it is fair. There are two approaches we can take to this, one of which involves adapting FDR itself.

Within the standard capabilities of FDR it is possible to attack this sort of fairness in a way which conveys extra quantitative information but which adds significantly to the computational complexity of a check. For simplicity assume that we are trying to prove that, from some defined point in a program's execution, an event signalling progress (e.g., the `Success` event in the 'eventually `b` monitor) always occurs in a finite time on the assumption of fairness. If you believe that some finitary pattern of turns which is implied by fairness guarantees this event, then the check of your belief is both finite and proves what is required. The best way to express these patterns is probably in terms of the *step* events discussed in the last section.

In the case of two thread processes, we only need consider a single sort of pattern, namely that the sequence of `step.i` actions has an alternating subsequence of a given length. (This means that even though the individual processes may get many consecutive turns, they alternate at least the stated number of times.)

Testing if a given number of alternations in this manner imply the progress event is straightforward to carry out on FDR: all one has to do is put a process in parallel that detects when the alternations have occurred and says so by means of an event. If it is impossible for this event to occur without the progress event first, then fairness implies our liveness specification. If it fails this is for one of two reasons, either there is a fair execution that never exhibits the progress event, or we simply have not tried enough alternations.

With more processes executing fairly the analogous test is to insist on a subsequence of steps that occur in some fixed rotating order. With more than two there is a choice of rotating order. While any order will, for a sufficiently large number of turns, prove any progress result of the form outlined above, the choice of order can have a large influence on the number of turns required to force progress and hence the complexity of the check.

It is similarly possible to do finitary checks for fair divergences: for example, is there a fair divergence in which there are never more than N actions of other processes between consecutive actions of any individual one? For a finite-state process *either* one of these checks will fail for sufficiently large N *or* one of the earlier sort will succeed. The problem is that as we progress through either of these families of checks the state space will get larger and it may well not be possible to resolve the issue for a system whose stand-alone state space is comparatively moderate.

In reality it may well be useful to quantify the way in which fairness implies progress (as set out in the preceding paragraphs) if this is tractable, but we really need a more efficient way of resolving whether or not fairness does imply progress.

D. Adapting FDR

Any liveness check of the sort discussed in the previous section can be reduced – by appropriate manipulations of the CSP – into a check for a fair divergence: an infinite sequence of computation steps which include an infinite number from each thread process. All that is required is to hide those actions which follow the point from which the progress event must eventually occur, but not that event itself. The liveness check then succeeds if there can be no fair divergence preceding the progress event.

There is no way of formulating this as a single check within the standard functions of FDR. One can test for *unfair* divergences, because all one has to do is look for divergence when the actions of a proper subset of the threads is hidden, but that tells one little of value.

The solution is to tell FDR the events corresponding to the different threads, and then only report divergences which include at least one from each of these sets in the cycle. To perform this check, FDR has been adapted so that it calculates the strongly connected components (SCCs) of the graphs of process states under ordinary τ actions alone and these together with the actions of the various thread processes that would comprise a fair divergence. (An SCC is a maximal set of nodes that are mutually reachable in the directed graph: these can be computed by a variety of efficient algorithms related to depth-first search.)

A fair divergence is possible if and only if either there is any non-trivial SCC in the first graph (one in which either there there are at least two nodes or a single node with a τ to itself) or an SCC F in the second graph in which, for each of the sets A of actions, there is an edge labelled by a member of A between nodes of F.

Obviously the first of of these can be decided by the standard divergence check. The second can be checked using the new form of FDR assertion (specially implemented for this purpose: I am grateful to Formal Systems for implementing this and the US ONR for funding it):

```
assert P :[fair divergence free]: S
```

where `S` is a set of actions such that a fair divergence must have one of each. The `S` actions are not hidden within the definition of `P`, so what this really determines is whether there is a fair divergence if `S` were to be hidden. (All the actions of a given thread are renamed to a single member of `S` before this check is performed.)

The flavour of model-checking in the presence of fairness that this new function performs is closely related to the problem of checking specifications over Büchi automata described in [15].

Example III.1 Mutual exclusion revisited
Dekker's algorithm (taken here from [16]) (like Hy-

man's, designed to be executed in two threads labelled 1 and 2) does pass the mutual exclusion safety check that the earlier one failed.

Following the convention once again that `i` is this process and `j = 3-i` is the other one, it can be written:

```
While true do
{csi.i -> Skip;
 b[i] := true;
 While b[j] do
   {if t==j then
              {b[i] := false;
               While t==j do Skip;
               b[i] := true}
   }
 css.i -> cse.i -> Skip;
 t := j;
 b[i] := false}
```

The additional signal event `csi.i` denotes the *intention* of thread `i` to perform a critical section. An obvious liveness requirement is that when either process wishes to perform a critical section it is allowed to. This can be tested by "hiding" all actions that follow `csi.i` except for the event `css.i` (this can be accomplished by one-to-many renaming and using a regulator process in a manner similar to that used on page 401 of [11], a process that can be automated by the compiler). By "hiding" here we mean mapping the actions of the two threads following a `csi.i` to separate visible actions that can be supplied as the final argument to the fair divergence check. When this check is performed the process passes, meaning that the liveness specification is satisfied.

If, on the other hand, we make the algorithm more "deferential" by adding an initial wait for the other process's boolean to become false.

```
While true do
{csi.i -> Skip;
 While b[j] do Skip;
 b[i] := true;
 While b[j] do
   {if t==j then
              {b[i] := false;
               While t==j do Skip;
               b[i] := true}
   }
 css.i -> cse.i -> Skip;
 t := j;
 b[i] := false}
```

While this does not destroy the safety property it does invalidate liveness: a fair divergence is found in which corresponds to one process doing an infinite number of critical sections while the other performs the new "after you" loop.

It should be noted that the concept of fairness described here is different from that in Walker's paper [16], since it is based on computation steps rather than being specific to the world of mutual exclusion algorithms.

Again, no real originality is claimed for the results on mutual exclusion here: the purpose of the example is just to illustrate the operation of the compiler and the new FDR check.

Most of the other examples from [16], together with various examples not about mutual exclusion, can be found on `state2.csp`'s web site referred to in the introduction.

Note that it would be almost ludicrous to assume any sort of atomicity in analysing a mutual exclusion algorithm. Naturally, none of the author's analyses of these algorithms do impose any atomicity.

IV. Conclusions and further work

The work in this paper has demonstrated that it is a practical proposition to use CSP$_M$ and FDR to model check programs written for shared variable languages. All the checks referred to in this paper execute practically instantaneously on the current implementation of FDR, so in no sense should they be regarded as the limit to what can be handled.

The flexibility of modelling and the links with the well understood semantics of CSP mean that this form of modelling provides, potentially, a useful tool for examining details of the execution of shared variable programs.

The ability of the work we describe to examine the consequences of atomicity in detail mean that this or a similar tool might well prove useful in work studying this and related ideas in the world of shared variable languages, for example [1].

There is a need for input in a form other than as a `.csp` file, but it should not be hard to create an input format containing the program in a natural form and enough other information to create the checks (as was done for security protocols in Casper [7], for example). There are no obvious limitations on the language used to specify the sequential threads other than the need (exemplified by the very finite type of integers we have used) to keep state spaces manageable.

There is no reason in principle why we could not allow "mixed-mode" programs, in which conventional CSP (or some other form of message passing) and the use of shared variables are combined together.

Giving an operational semantics for one language in terms of another (as we have done here) provides a natural bridge by which results of proved for the second language can be applied to the first. One possibility here is the application of results about data independence in CSP [8] to shared variable programs without the assumptions about atomic assignments that are usually made (see [9], for example).

Related to this last point, treating chosen variable processes as holding symbols for values rather than values themselves might well allow us, in suitable (largely data independent) cases, to prove results about systems with larger or general types rather than a specific small example.

References

[1] S.D. Brookes, *Transfer Principles for Reasoning about Concurrent Programs*, to appear in the proceedings of MFPS 2001, ENTCS.

[2] Formal Systems (Europe) Ltd., *Failures-Divergences Refinement*, Documentation and user manual `www.formal.demon.co.uk/fdr2manual/index.html`

[3] K. Fuhrmann and J. Hiemer, *Formal Verification of STATEMATE-Statecharts*, `citeseer.nj.nec.com/255163.html`, 2001.

[4] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[5] G. Holzmann etc., *The SPIN model checker*, Home page: `netlib.bell-labs.com/netlib/spin/whatispin.html`

[6] H. Hyman, *Comments on a problem in concurrent programming*, CACM **9** 1, 1966.

[7] G. Lowe, *Casper, a compiler for the analysis of security protocols*, Journal of Computer Security, **6**, pp 53-84.

[8] R.S. Lazić, *A semantic study of data independence with applications to model checking*, Oxford University D.Phil thesis, 1999.

[9] R.S. Lazić and D. Nowak, *A unifying approach to data-independence*, In Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000), Lecture Notes in Computer Science. Springer-Verlag, August 2000.

[10] A.W. Roscoe, *Model-checking CSP*, in "A Classical Mind, Essays in Honour of C.A.R. Hoare", Prentice-Hall 1994.

[11] A. W. Roscoe, *The Theory and Practice of Concurrency*, Prentice Hall, 1998.

[12] A.W. Roscoe and M.H. Goldsmith, *The Perfect "Spy" for Model-Checking Cryptoprotocols*, Proceedings of DIMACS workshop on cryptographic protocols, 1997, `dimacs.rutgers.edu/Workshops/Security/program2/goldsmith.html`

[13] P.Y.A. Ryan, S.A. Schneider, G. Lowe, M.H. Goldsmith and A.W. Roscoe, *Modelling and Analysis of Security Protocols*, Addison Wesley 2001.

[14] J.B. Scattergood, *The Semantics and Implementation of Machine-Readable CSP,* D.Phil., Oxford University Computing Laboratory, 1998.

[15] M. Vardi and P. Wolper, *An Automata-Theoretic Approach to Automatic Program Verification,* Proceedings of LICS 1986, pp322-331. (Kluwer).

[16] D.J. Walker *Automated Analysis of Mutual Exclusion Algorithms using CCS*, Formal Aspects of Computing **1**, pp273-292, 1989.

[17] P. Whittaker, G.M. Reed and M.H. Goldsmith, *Formal Methods Adding Value Behind the Scenes*, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '00), CSREA Press, 2000.,