Probing the Depths of CSP-M: A new FDR-compliant Validation Tool

Michael Leuschel and Marc Fontaine

Institut für Informatik, Universität Düsseldorf**
Universitätsstr. 1, D-40225 Düsseldorf
{leuschel,fontaine}@cs.uni-duesseldorf.de

Abstract. We present a new animation and model checking tool for CSP. The tool covers the CSP-M language, as supported by existing tools such as FDR and PROBE. Compared to those tools, it provides visual feedback in the source code, has an LTL model checker and can be used for combined CSP ∥ B specifications. During the development of the tool some intricate issues were uncovered with the CSP-M language. We discuss those issues, and provide suggestions for improvement. We also explain how we have ensured conformance with FDR, by using FDR itself to validate our tool's output. We also provide empirical evidence on the performance of our tool compared to FDR, showing that it can be used on industrial-strength specifications.
Keywords: CSP, Tool Support, Model Checking, Animation, B-Method, Integrated Formal Methods, Specification Language Design, Logic Programming.

1 Introduction

CSP and B can be effectively used together in a complementary way to formally specify systems [29, 3, 5]. B can be used to specify abstract state and can be used to specify operations of a system in terms of their enabling conditions and effect on the abstract state. CSP can be used to give an overall specification of the coordination of operations.

The overall goal of this research project was to obtain an animation and model checking tool which could be used to validate such combined B and CSP specifications. A major requirement was that this tool should deal with full B specifications in AMN (abstract machine notation) as well as with full CSP specifications in CSP-M (machine readable CSP), so that existing industrial tools such as Atelier B [28] or FDR [7], could be applied to validate the individual B or CSP specifications in isolation.

** A substantial part of this research has been sponsored by AWE, plc within the project "ProCSB" as well as by the EU funded FP7 project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

After previous attempts with other tools,[1] in this project it was decided to extend the ProB tool [16], in order to deal with CSP-M specifications. Indeed, ProB already deals with full AMN and provides both animation and model checking facilities. ProB can also be applied with reasonable effort to new specification languages, provided a Prolog interpreter for that language is available. In earlier work, ProB was already extended to validate combined B and CSP specifications [5]. However, this work was based on the earlier interpreter [15], which only treats a very small subset of CSP-M (e.g., there are no replicated operators, no channel declarations, no let declarations or lambda constructs, etc.), and used its own incompatible syntax (which was designed at the time to enable easier parsing by Prolog). In other words, the previous tool was an academic prototype;[2] what is required is a new tool which can handle existing real-life CSP-M specifications.

A major hurdle of this research project was the development of a new parser and interpreter for full CSP-M for use within ProB. The issue of synchronising the B interpreter with the CSP interpreter is mainly achieved in the same manner — via Prolog unification — as in the earlier work [5]. Hence, in the rest of this paper, we will only briefly discuss the issue of synchronising the B interpreter with the CSP interpreter and concentrate on presenting the new CSP-M parser, type checker, and interpreter. This is also warranted, since we have actually also gained a new powerful tool to animate and model check CSP-M specifications. This tool overcomes some of the drawbacks of existing tools FDR and PROBE, and adds several new features:

- precise syntax error highlighting in the source code
- precise semantic error highlighting in the source code
- visual feedback on complicated events in the source code
- the ability to deal with infinite state sub-processes and to some extent also main processes
- graphical visualisation of the state space of CSP-M specifications
- the ability to apply LTL model checking to CSP-M [20]
- a new type checker for CSP-M [6]
- a web-interface using Google-Web-Toolkit (`www.myprob.de`)

In the remainder of this paper you will find various critiques of CSP-M, and also of FDR and PROBE. To clarify our position, we believe that FDR and PROBE are two very useful tools, and that CSP-M is a very attractive specification language. There are also aspects of CSP that are dealt with much better by FDR and PROBE, than by our tool (e.g., deeply nested processes or compression [25]). Exactly because we believe that CSP-M and the existing tools are relevant, are

[1] E.g., combining the commercial BToolkit animator from BCore with the commercial PROBE animator [8] from Formal Systems Ltd. in the EPSRC funded project GR/R96859/01 "Verification and Development of Interacting Software Components".

[2] Indeed, our new interpreter alone is roughly six times the size of the interpreter in [5].

we interested in improving and clarifying the language. We hope that we have provided insights and a new tool of value to the formal engineering methods community beyond CSP $\parallel$ B.

Outline We start off be outlining the difficulties of treating CSP-M in Section 2, also pinpointing some semantic issues of the language, such as lack of substitutivity of equals. We describe our Prolog interpreter in more detail in Section 3 and the new parser in Section 4. In Section 5 we discuss how we have ensured compliance of our tool with the existing CSP-M tools. In Section 6 we present empirical results of using our new tool. We conclude with discussions and more related work in Section 7.

2 Challenges of Validating CSP-M

While core CSP [10] is a relatively succinct language, full CSP-M as supported by FDR is a very extensive and expressive specification language. Some of the aspects that make CSP-M a challenging target for a formal validation tool are:

- CSP-M is a dynamically typed language.
- CSP-M comprises a higher-order functional programming language.
- CSP-M contains an extensive range of datatypes: booleans, integers, tuples, associative tuples, sequences, sets, and combinations thereof. The use of an associative datatype is rather unique, and enables some elegant formalisations. On the other hand it considerably complicates the life of the tool developer, as we show below.
- CSP-M has grown and matured over a considerable period of time, and many features were added, e.g.:
  - complex pattern matching, even allowing combinations of patterns to be expressed using the `@@` operator. One unique aspect of CSP-M is the ability to use the concatenation operator `^` inside function patterns. E.g., one can define a function that computes the last element of a list by: `last(s^<x>) = x` (no mainstream functional or logic programming language allows this).
  - complex channel matching with associative tuples and trailing question marks which can match tuples of arbitrary length.
  - closure operations on partially constructed datavalues; partially constructed datavalues can be passed to functions.
  - nested function definitions, which can appear in many places (e.g., inside an individual channel output expression)

The complexity of CSP-M is also reflected in its syntax, which makes parsing surprisingly hard (see Section 4). As we will show below, some parts of CSP-M also have an overly complicated semantics, which is furthermore formalised neither in Roscoe's book [24] nor in the FDR manual. We will discuss some of the subtle, problematic points in the remainder of this section. How our implementation copes with higher-order functions and nested functions is explained in Section 3.

Dynamic Typing. CSP-M, as accepted by FDR and PROBE [8], is a dynamically typed language. Indeed, while channels must be statically typed by the user with channel declarations, CSP-M provides no way of declaring types for functions and variables. Moreover, the existing tools FDR and PROBE accept specifications (without warning or runtime error message) in which there is no way to statically type variables or functions, as the following example shows:

```
datatype COL = red | green
channel ci:{0..4}
channel cc:COL
BUF(<>) = STOP
BUF(<n,val>^s) = ci!n -> if n<2 then ci!val->BUF(s) else cc!val->BUF(s)
MAIN = BUF(<1,1,3,red,1,4>)
```

Here the argument to BUF is a sequence consisting of mixed integers and colours, the type of every second item depending on the value of the preceding integer. Hence, at least as far as the existing tools FDR and PROBE are concerned, CSP-M is a dynamically typed language. This means, e.g., that an interpreter cannot rely that operation arguments (e.g., for arithmetic operations) have the correct type and must perform type checks at runtime.

There now is a prototype static type checker available for download from Formal Systems. It would reject the above specification; but it will also sometimes reject legal specs (see further below). For critical applications, we believe that it is good practice to write only specifications that are accepted by such a static type checker, even if the tools FDR and PROBE can deal with dynamically typed specifications. Still, in order to be compliant with these existing tools, we also support dynamic typing, but we have also developed an (optional) static type checker [6].

The Trouble with Tuples. Associative tuples can be constructed using the dot (.) operator. This, however, is not the only use of the dot operator and actual meaning depends on the context of use and on prior datatype declarations. Take, e.g., the following CSP-M specification:

```
channel a,a': {0..9}.{0..9}   {- Meaning 1. -}
datatype R = r.{0..9}         {- Meaning 2. -}
channel b,b':R
MAIN = Rep1(a,a',2.3);        {- Meaning 3. -}
       Rep2(b,b',r.3)         {- Meaning 4. -}
Rep1(c,c',x) = c!x -> SKIP [| {|c|} |] c?y?z -> c'!y!z -> SKIP
Rep2(c,c',x) = c!x -> SKIP [| {|c|} |] c?y?z -> c'!y -> SKIP
```

This example (which is type correct according to Formal Systems type checker) shows that the dot has actually four different meanings:

1. Here the dot generates a cartesian product of two sets, and not a tuple of two sets. This is made clear that if one defines `T = {0..9}.{0..9}` and then tries to declare `channel a,a':T` one gets the FDR error message "Value

{0,1,2,3,4,5,6,7,8,9}.{0,1,2,3,4,5,6,7,8,9} is not a set." Thus, inside a channel declaration the dot acts as a Cartesian product, and substitutivity of equals does not hold: one *cannot* replace `{0..9}.{0..9}` by `T` even though they are declared to be equal. One could hence argue that CSP-M is not referentially transparent.
2. Here we construct a set of records. This is similar to point 1, but this time the first argument is not a set but a constructor. This use of the dot occurs inside a datatype declaration.
3. Here the dot operator constructs an associative tuple, which can be deconstructed into its two constituents using the `c?y?z` channel prefix in `Rep1`.
4. In contrast to point 3, here we do *not* construct a tuple, but a record. The difference becomes apparent in the channel prefix `c?y?z` in `Rep2`, where this time `y` gets bound to the entire record and `z` is bound to the empty tuple. Hence the output `c'!y` is *not* a type error and is accepted by FDR. Note that calling `Rep2(a,a',2.3)` would lead to a type error.[3]

Meanings 3. and 4. occur outside of channel and datatype declarations, and they can be distinguished by checking whether the first component of a dot is a record constructor declared in a datatype declaration. Also note that there is no way to statically distinguish between case 3 and 4, as a slight adaptation of the above example shows:

```
Rep3(c,c',x1,x2) = c!x1.x2 -> SKIP [| {|c|} |] c?y?z -> c'!y!z -> SKIP
MAIN2 = Rep3(a,a',2,3); Rep3(b,b',r,3)
```

This time the occurrence of the dot operator inside `Rep3` will, depending on its arguments, either construct a tuple (for the call `Rep3(a,a',2,3)`) or a record (for the call `Rep3(b,b',r,3)`).

We have faithfully implemented these various uses of dot, checking conformity with FDR and PROBE (see Section 5). However, we believe that this aspect of CSP-M should be simplified in future, at least using another operator for record construction. Indeed, even to several CSP-experts (some authors of books about CSP), the difference in behaviour between point 3 and 4 came as a surprise. Only after experimenting with FDR and PROBE and discussions with Michael Goldsmith (one of the main developers and researchers behind FDR) did we realise that there was a subtle, but important difference between the third and fourth use of the dot operator. We think that such semantic pitfalls should be avoided in languages aimed at formal modelling of critical systems.

As a side-note, the empty tuple to which `z` is bound inside `Rep2` does have no syntactic denotation in CSP-M. We also believe that this should be remedied in future revisions, as it prevents certain source-to-source transformations to be applied to CSP-M specifications (e.g., constant folding cannot be applied if the constant happens to be the empty tuple).

[3] More precisely, FDR says "Mismatch calculating augment processing communication !y near line 9 of ThreeDots.csp on channel a' for event a'.8.9".

Associativity of Tuples. The associativity of tuples poses some major headaches, both for static analysis of CSP-M specifications, as well as for animation and model checking tools. For example, given a prefix `c3!x?val!y -> ...` and a channel declaration `channel c3:{0..4}.COL.{0..4}` one would assume that it is possible to statically infer the type of `val` to be of type `COL`. As the following example shows, this is not the case:

```
datatype COL = red | green
channel a:{0..4}
channel c3:{0..4}.COL.{0..4}
MAIN = a?x?e -> if x<2 then P(x.red,e) else P(x,x)
P(x,y) = c3!x?val!y -> STOP  -- here val can be either Int or COL
```

Let us examine the `MAIN` process after `a?x?e`: the variable `x` will contain a number between 0 and 4 and `e` will always contain the empty tuple. If the number is smaller than 2, `val` inside the `P` process will be of type integer and not of type colour. If `x` is greater or equal to 2, then `val` will be of type `COL`.

This means that channel prefixing is a surprisingly complex operation in CSP-M. In addition to mystifying users, the above also makes the life of the tool developers much harder. For example, the result is that the Formal Systems type checker sometimes fails to type-check a program, even if it can in principle be statically typed. More concretely, if `MAIN` is replaced by `MAIN = a?x?e -> P(x.red,e)` then `val` is always of type integer, but the type checker fails to type-check the specification.

We also found out that in some circumstances FDR and PROBE did not fully implement all consequences of the associativity of the tuples. Indeed, changing the definition of `P` above into the following one, leads to an "unsupported comparison processing communication error" in both FDR and PROBE:

```
channel b:COL
P(x,y) = c3!x?val!y -> if x<2 then a!val -> STOP else b!val -> STOP
```

Our tool does support all of the above specifications. Still, as in the case of the different meanings of the dot operator above, we believe that this aspect of CSP-M should be cleaned up in future versions. The additional expressiveness incurred by associativity and flexible channel matching is in our opinion not sufficient to counterbalance the lack of predictability and clarity for the user.

3 The CSP-M Prolog Interpreter

In earlier work [5], synchronisation of B and CSP was achieved by

- having interpreters for B and CSP both in Prolog and
- using Prolog unification to synchronise the two interpreters.

This provided an animator and model checker for CSP $\|$ B specifications.
This aproach has proven to work very well and has been kept in this project. Figure 1 shows the general architecture of our development. The "Sync" box

required only relatively minor changes to the existing code base from [5]. The CSP-Parser and Interpreter, however, have been completely re-developed. As already mentioned, the previous parser was incompatible with FDR and the previous interpreter only supported the core CSP language, leaving many of the difficult aspects of CSP on the side, notably the ones detailed in Section 2. This required the development of several intricate compilation techniques and explains the sixfold increase in code size of our new interpreter.

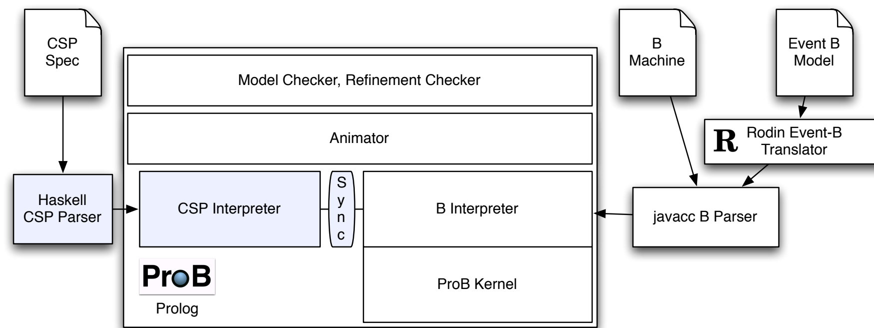

Fig. 1. Architecture

We describe the new parser in Section 4. The new CSP-M interpreter was inspired by the earlier interpreter in [5], which in turn was inspired by [15]. Just as in [5], co-routining was used to enable us to write a compositional interpreter, and translating the operational semantics rules of Roscoe [24]. The Prolog code implements a ternary relation *cspm_trans*, where $cspm_trans(e, a, e')$ means that the CSP-M expression e can evolve into the expression e' by performing the event a.

Why co-routining? In classical Prolog literals within a goal are selected strictly from left to right. Co-routines (sometimes also called delay or wait declarations) [22] enable a programmer to influence Prolog's selection rule via `when`, `block` or `freeze` declarations. For example, take the following two clauses, defining list membership:

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

Now take the query goal `?- member(X,L), L=[1], X=2.` In classical Prolog, the leftmost literal `member(X,L)` will be selected first. As `member(X,L)` has infinitely many solutions, the above query will not terminate and the Prolog interpreter will never realise that there is no solution for the query. By adding,

e.g., a block declaration `:- block member(?,-).` , we ensure that all calls to `member` will delay until the second argument is not a variable anymore. For the above goal, Prolog will thus suspend the call `member(X,L)`, and execute `L=[1]` first. This will awaken `member(X,L)`, which now finds only a single solution `X=1`, after which the call `X=2` simply fails.

For a pure logic program, these declarations do not modify the logical meaning, but they can be used to improve the runtime behaviour of the program. In particular, one can ensure that an interpreter for CSP will terminate, or be much more efficient, in circumstances where Prolog's classical left-to-right selection rule would lead to non-termination or to unacceptable performance.

Take the following example (part of the regression test for our new tool):

```
MyInt = {0..99}
channel ch: MyInt.MyInt
channel out,out2: MyInt
T(x) = ch!x?y:{z|z<-MyInt, z<x} -> out!y -> T(y) []
       ch!x?y:{z|z<-MyInt, z>=x} -> out2!y -> T(y)
S(x) = ch?y!x -> S(y)
MAIN =  T(1) [| {| ch |} |] S(2)
```

Let us examine how the interpreter handles the synchronisation operator `[|{|ch|}|]` by looking at one of the inference rules from [24]:

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{P \mid\mid {}_X Q \xrightarrow{a} P' \mid\mid {}_X Q'} \quad (a \in X)$$

Here $P = T(1)$, $Q = S(2)$ and X is the closure of ch, i.e., $X = \{ch.0.0, ch.0.1, \ldots, ch.99.99\}$. In order to determine whether this rule is applicable, the interpreter will first compute the outgoing transitions of $T(1)$. One problematic issue here is that the interpreter does not yet know the value of y and hence cannot yet determine whether y is a member of the set comprehensions inside T's definition. One solution would be to simply enumerate all possible values for y, as indicated by the type of the channel.[4] This is actually the approach employed by FDR and PROBE. The drawback, however, is that a lot of effort can be wasted if the type of the channel value is large (e.g., here T(1) has 100 possible events), and there is no way to treat channels with unbounded types.

We have overcome this limitation by employing co-routining: basically the value of y will be left uninstantiated (a free Prolog variable) and the membership test delays until the value is known. As it is possible that some channel fields are never given an explicit value, the interpreter is wrapped into an outer layer which will enumerate any remaining uninstantiated channel fields at the very end.

Several other constructs will delay until their arguments are sufficiently instantiated. This has efficiency advantages, but also allows one to translate the

[4] In general, there can be a mixture of inputs and outputs, and information can flow both ways.

logical inference rules from [24] in a relatively straightforward way, without having to worry in which order to evaluate the subexpressions nor having to perform additional analyses. The statespace of the above specification, as computed by our tool, is shown in the left of Fig. 2.

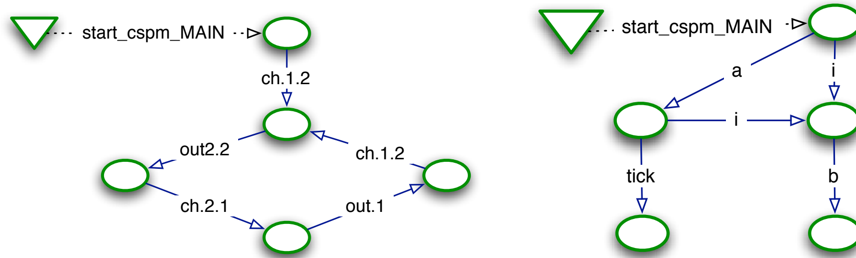

Fig. 2. Two statespaces as displayed by ProB

Here is a more realistic example with unbounded channel types. The overall state space of MAIN is finite, and our model checker can (quickly) check the entire state space. The only trace of the system is $\langle out.1, out.1, out.2, out.3, out.5, gen.5\rangle$. Neither PROBE nor FDR can be used with this specification, because the channel types are not bounded.

```
channel out,gen: Int
FibGen(N,M) = if M<10000 then out!M -> FibGen(M,N+M) else STOP
Take(n) = if n>1 then out?_ -> Take(n-1)
                 else out?x -> gen!x -> STOP
FibSeq = FibGen(0,1)
MAIN = FibGen(0,1) [| {| out |} |] Take(5)
```

Our tool can also deal with truly infinite state processes, in the sense that they can be animated and partially validated. For example, our tool can find the deadlock in `MAIN = P [| {a,b} |] Q` where `P = (a -> P) [[ a<-b, b<-a]]` and `Q = a->Q [] b->a->b->STOP` after less than 0.01 s, whereas FDR goes into an infinite loop.

Very often a natural specification style is to have sub-processes which are infinite-state and only the global composition makes the system finite. For example, the following is a quite natural pattern, having an infinite state server process, but which is constrained by the environment so that the entire system (MAIN) is finite state and can be model checked. This specification can be exhaustively model checked using our tool, but FDR cannot deal with it because the tool tries to expand and normalise the infinite state `Server` process *before* conjoining it with the `User` process.

```
ID = {0..3}
channel new, ping, ack: ID
channel shutdown
Server =  new?id -> (Server ||| Serve(id)) [] shutdown -> STOP
Serve(id) = ping?id -> ack!id -> Serve(id)
User = new?i -> UserActive(i)
UserActive(i) = ping!i -> ack!i -> UserActive(i)
MAIN = Server [| {| new,ping,ack,shutdown |} |] User
```

Recursion and precompilation Our tool stores the CSP-M functions in a precompiled Prolog database of definitions. It uses the so-called non-ground representation [9], meaning that variables in CSP-M functions are represented by variables in the Prolog code. This enables us to use unification and Prolog argument indexing for efficient function lookup (see, e.g., the McCarthy benchmark in Section 6).

While the non-ground representation leads to an efficient interpreter, care has to be taken with local variables to ensure that different uses and unifications of the same local variable do not interfere with each other. A pre-compilation phase of our tool ensures that all local variables are pushed down into their own Prolog clauses, so that fresh copies are obtained before every use. Take for example the following CSP-M specification:

```
Repeat(X) = (X ; Repeat(X))
RepHalf = Repeat(out?x -> out!x/2 -> SKIP)
```

Here, the local variable `x` is "re-used" on every recursive call to `Repeat`. By generating an extra function (named here `RepHalf->__23`) for the prefix construct introducing the local variable, we ensure that every time a new copy of the local CSP-M variable is required, a fresh Prolog variable will be generated (as the function `RepHalf->__23` will be represented by an individual Prolog clause and every time the clause is used, fresh copies of the local variables are produced):

```
Repeat(X) = (X ; Repeat(X))
RepHalf = Repeat(RepHalf->__23)
RepHalf->__23 = out?x -> out!x/2 -> SKIP
```

Let expressions and lambda lifting Nested let expressions are implemented using a static compilation technique called lambda lifting (or closure conversion) [13] (see also Chapter 13 of [14]). The idea is that every free variable of a nested function body is converted into an additional argument of the function, and the local function is replaced by a renamed global function. Take for example the following CSP-M specification:

```
Repeat(X) = (X ; Repeat(X))
RepCube(x) = Repeat(out!x -> (let f(m)=m*x  within out!f(m*x) -> SKIP))
```

We generate a new global function definition for the inner function `f`, adding the free variable `x` as extra argument of the function. The function is also given a special name (here `RepCube*f__1`), to avoid name clashes with user-defined functions and other lifted functions:

```
Repeat(X) = (X ; Repeat(X))
RepCube(x) = Repeat(out!x -> out! RepCube*f__1(x*x,x) -> SKIP)
RepCube*f__1(m,x) = m*x
```

Currying and Lambda Expressions In addition to the standard syntax for functions, CSP-M also allows functions to be defined in curried form. In the following `f` is defined as a curried function and the `MAIN0` process should output 1.2 on the `out2` channel:

```
channel out2: {0..16}.{0..16}
f(x)(y) = out2!x!y -> SKIP
MAIN0 = Run(f(1))
Run(fc) = fc(2)
```

Curried function definitions are detected by our parser and our pre-compilation phase translates curried functions into lambda expressions. For example, internally the above function `f` is represented as if it was defined by the following:

```
f(x) = \y @ out2!x!y -> SKIP
```

This of course begs the question how CSP-M lambda abstractions have been implemented. First, the arguments of a lambda abstraction (i.e., the `y` in the example above) are frozen during a pre-compilation phase to ground Prolog terms (using the `numbervars` built-in, which instantiates all the free variables in a Prolog term to ground terms of the form `'$VAR'(`n`)`, where n is a number). When a lambda abstraction is applied, then its parameters are lifted again to real Prolog variables, which are then unified with the actual parameter values, after which the body of the lambda abstraction is evaluated. This scheme allows lambda abstractions to be applied multiple times with different parameter values.

Tracking Source Code Positions. Our parser was designed from the start to keep track of source code positions. Indeed, one aspect of FDR that can be frustrating, is that some of the error messages contain only approximate or no source code positions (e.g., for the error message "Function applied to a value outside its domain").

Our tool's source code positions are not only used to pinpoint syntax errors, but also to show to the user the exact location of certain "runtime" errors (e.g., when a process outputs a value that is outside of the channel type), as well as give feedback on events in the animator. Indeed, when animating the user can find out which locations in the source code contributed to a particular event (see Figure 3). This also works for τ events. This is achieved by the parser adding source code information inside the abstract syntax tree and by the interpreter keeping track of all source code positions that are involved in a certain event.

```
Operate = (balance?a?v -> Operate
        [] transferReq?s?a1?a2?ok ->
             (transferExec!s!a1!a2 -> Operate
              [] abort -> Operate)
        [] logout -> mainB)
-- OZ Part Bank
-- We represent the current balance bal as a set of
-- pairs (account-id, value). This requires some
-- auxiliary functions defined below:
ValSet = \b,a @ { v | v <- Val, m((a,v),b) }
pick({x}) = x
PickVal = \b,a @ pick(ValSet(b,a))
withdrawOK = \b,a1,a2,s @
          not(a1==a2) and
          (PickVal(b,a1) - s >= 0)
upd = \b,a,v @
   let
          bminus = diff(b,{(a,vold) | vold <-Val })
   within
          union(bminus, {(a,v)})
-- The set of customers is defined as a concrete
-- subset of UserID. It appears as a global parameter
-- of the process OZB.
cust = {u1, u2}
OZB(bal,transferOK) =
      (m(bal,Set({(a,v)|a<-AccID,v<-Val})) and
       m(transferOK,Bool)) &
          (
          ([] (u,ok): {(u,m(u,cust)) | u <- UserID } @
               login.u.ok -> OZB(bal,transferOK))
          []
           ([] (a,v) :
            {(a,PickVal(bal,a)) |
               a <- AccID, card(ValSet(bal,a))==1 } @
            balance.a.v -> OZB(bal,transferOK))
         []
          ([] (s,a1,a2,ok):
          {(s,a1,a2,withdrawOK(bal,a1,a2,s)) |
             s<-Sum, a1<-AccID, a2 <-AccID,
            card(ValSet(bal,a1)) == 1 } @
           transferReq.s.a1.a2.ok -> OZB(bal,ok))
         [] transferExec?s?a1?a2 ->
```

Fig. 3. Source code highlighting the bank model from [2] (partial view)

4 The new Parser in Haskell

As part of this work, we also implemented a new CSP-M parser. A major change from the existing FDR parser [26] is the incorporation of source code information inside the abstract syntax tree. Another design goal was that the code for the new parser should be understandable, and that it should be easy to extend and reuse the code in future projects. The reference implementation of the CSP-M parser [26] uses the bison parser generator, but the grammar, that actually serves as input for bison, is itself generated with a Perl script. This however, quite effectively obfuscates the parser and makes extending it very difficult.

Furthermore, apart from serving as input to our new CSP-M validation tool, we are also currently adding CSP-M support to the Eclipse IDE, as well as producing new Haskell based tools for type checking CSP-M. To allow this interoperability, our parser can deliver parse-results in different output formats, e.g., as a set Prolog facts or as a Java object representing the abstract syntax tree. Our parser was developed with the Glasgow Haskell Compiler (GHC) and is currently available as dynamic-link-library for four different architectures. No knowledge of Haskell is required to use the parser.

Internally the parser is based on the Parsec combinator parser library [11]. Our practical experience, implementing a combinator parser for CSP-M in Haskell, was overall positive. In total the parsing library contains about 4400 lines of Haskell. The specification of the CSP-M syntax is clearly separated from the generation of the results (for Prolog/Java/Haskell). The performance of our CSP-M parser is definitely sufficient for our applications. For the specification of the alternating bit protocol (`abp.csp`), which is about 12KB long, the hole process from source file to Java AST takes about 50ms on a 1.3GHz Pentium laptop. Our parser is also capable of parsing big auto generated files. For example, we used the parser to process a 3.5MB file which is the CSP transliteration of a large labelled transitions system (as described in Section 5). Just parsing of this file takes about 22 seconds. A complete run of the command-line-interface version of the parser which includes, parsing, renaming/bound variable analysis, translation to a 19.7MB Prolog file, translation to a 9.4MB Haskell file and dumping a lot of information to the terminal, takes about 1 minute and 40 seconds.

In retrospective, the syntax (and semantics) of CSP-M contains many awkward and possibly doubtful features. For example, in CSP-M '`<`' and '`>`' are used for arithmetic comparison *as well as* the beginning and end of sequences and a parser has to deal with special cases like '`<true, 2>1, false>`', i.e. a comparison inside a list of Boolean expressions. These features are also the reason why it is very difficult to rigourously describe the syntax of CSP-M in terms of, i.e. a context free grammar. Luckily, with the Parsec combinator parser library, it is not important that the language one wants parse is in $LL(k)$ or $LALR(k)$. Infinite lookahead is no problem and the language does not even have to be context-free. This flexibility turned out to be invaluable for our implementation.

5 Ensuring Compliance via Refinement Checking

To ensure that our tool is compliant with FDR we have added over 85 CSP-M regression tests. These are a mixture of existing examples (mostly from the books [24] and [27]) and artificially constructed, contrived specifications. For every regression test, a test sequence has been stored and is checked.

Furthermore, unless the CSP example cannot be treated by FDR (e.g., because of the use of an unbounded channel type), we perform a complete two way failures-divergence refinement test using FDR. For this, we have implemented a feature in ProB to dump the computed statespace into a CSP-M file. For example, the statespace on the left of Figure 2 is converted into the following:[5]

```
include "ComplicatedSync.csp"
assert MAIN [FD= Nroot
assert Nroot [FD= MAIN
Nroot =   (  {- start_cspm_MAIN-> -} N8 )
N8 =  ( ch.1.2->N9 )
N9 =  ( out2.2->N10 )
N10 =  ( ch.2.1->N11 )
N11 =  ( out.1->N12 )
N12 =  ( ch.1.2->N9 )
```

By running FDR in batch mode on this file, we perform a two way failures-divergence refinement check between the original CSP-M specification (MAIN) and the statespace computed by ProB (Nroot).

More precisely, if a state i in the state space has n outgoing transitions labelled with $e_j \neq \tau$ leading to s_j and m outgoing transitions labelled with τ leading to t_k, the process definition for i will look like this, where $\triangleright$ is the "time-out" operator:[6]

$$N_i = (e_1 \rightarrow N_{s_1} \,\Box\, \ldots \,\Box\, e_n \rightarrow N_{s_n}) \,\triangleright\, (N_{t_1} \sqcap \ldots \sqcap N_{t_m})$$

Furthermore, if any of the e_j are the tick event then instead of generating $e_j \rightarrow N_{s_j}$ we generate $SKIP$ (note that N_{s_j} must be a deadlocking state) and if any of the e_j are the interrupt event `i` then we generate $STOP \triangle N_{s_j}$.

Note that this testing was very useful in uncovering bugs as well as subtle semantic misunderstandings on our behalf, and ensures a high-level of conformance of our tool. It also caught the fact that FDR does not fully comply with Roscoe's operational semantics (as far as tick and Omega are concerned). Indeed, for the following example the two-way refinement check uncovered that FDR behaved differently from our tool (and from PROBE):

```
MAIN = a->SKIP /\ b->STOP
```

[5] `start_cspm_MAIN` is an artificial event of the ProB animator. Our translator automatically comments those events out.

[6] Thanks to Michael Goldsmith for suggesting the use of $\triangleright$ to convert arbitrary ProB statespaces into CSP-M.

Our tool computes the statespace as displayed in the right of Fig. 2. The encoding of the statespace in CSP-M is as follows:

```
include "TickOmegaFDRBug.csp"
Nroot =   (  {- start_cspm_MAIN-> -} N3 )
N3 =  ( a->N4 [] (STOP /\ N5) )
N4 =  ( SKIP  [] (STOP /\ N5) )
N5 =  ( b->N7 )
N7 = STOP
```

The Nroot process does not allow the trace $\langle a, tick, b\rangle$ which FDR computes for MAIN. The Roscoe semantics contains the following rule for the interrupt $\triangle$:

$$\frac{P \stackrel{tick}{\rightarrow} P'}{P \triangle Q \stackrel{tick}{\rightarrow} \Omega}$$

Hence, the sequence $\langle a, tick, b\rangle$ should not be allowed. After feedback from Michael Goldsmith, it turns out that FDR implements an earlier semantics (it uses $P \triangle Q \stackrel{tick}{\rightarrow} P' \triangle Q$), and actually behaves differently from PROBE in that respect (unless the `'-no-omega'` flag is given to PROBE on startup).

6 Empirical Evaluation

As our tool has not yet been tuned for speed, a full-scale empirical evaluation would be premature. Still, to give an idea about the strengths of our tool, we believe the following comparison with FDR to be useful. However, it should not be seen as an extensive benchmarking comparison between the two tools.

The experiments were run on MacMini with 1.83 GHz Core Duo processor running Ubuntu 7.04 with 512 MB inside of the Parallels Desktop virtualization[7] environment. All experiments involved deadlock checking of the MAIN process, and both ProB 1.2.7 and FDR 2.82 were used with default settings. Note that FDR only displays entire seconds and 0 s thus means < 0.5 s. Also, all of the non-zero timings of FDR were obtained using a stopwatch (as often the time indicated by FDR was substantially below the real time needed, as part of the precompilation time was not taken into account). The entry "**" means that FDR stopped with the message: "readEventMap failed, failed to compile ISM".

The alternating bit protocol is a sample file written by Roscoe, and accompanying Chapter 5 of [24]. Another example we tested is Crossing, a 374 line specification of a level-crossing gate, from the book webpage of [27][8]. Bank v4 is a controller accompanying a B machine from [30]. Bank Secure OZ is the model of a bank with a security automaton, translated from CSP-OZ to CSP, from [2]. This model by Bill Roscoe describes a level crossing gate using discrete-time

[7] This was to be fair wrt FDR, as FDR 2.82 was at the time only available as a PPC version for Mac.

[8] `http://www.cs.rhul.ac.uk/books/concurrency/`

Table 1. Empirical Results for Deadlock Checking

Specification	States	Transitions	ProB	FDR
Alternating bit protocol	1852	4415	2.17 s	0 s
Bank v4	892	1231	0.53 s	0 s
Bank Secure OZ	322	3556	18.84 s	5 s
Crossing	5517	12737	53.93 s	0 s
FibGen	8	7	0.00 s	132 s
GenericBuffer1	24	28	0.01 s	12 s
GenPrime	1240	2258	0.75 s	142 s
McCarthy0	202	201	0.40 s	4 s
McCarthy1	10002	10001	4.04 s	17 s
McCarthy2	30002	30001	14.96 s	**
Peterson v1	58	115	0.40 s	0 s
Peterson v2	215	429	1.23 s	0 s
Scheduler1_CSP	4174	18031	8.66 s	2 s
Scheduler0_6_Bexp	2189	14581	2.19 s	6 s
Server	14	17	0.01 s	**

modelling in untimed CSP. Peterson v1 and v2 are two versions from [27] of the Peterson mutual exclusion algorithm. FibGen is the example from Section 3, but with bounded channel types. I.e., this is an example of a large process composed with a small process (Take). GenPrime is an example of two processes generating numbers, and a deadlock is found if they generate the same number; the state and transition numbers are at the ones at the time when ProB found the deadlock. McCarthy0 is an example with pure recursive computation, computing the McCarthy function for x = 0..200 and then deadlocking. McCarthy1 and McCarthy2 are the same as McCarthy, but for x = 0..10000 and x = 0..30000. Scheduler1_CSP is the first refinement of a process scheduling specification from [17] in natural CSP style for 5 processes. Scheduler0_6_Bexp is the CSP representation of the statespace of the abstract B model of this process scheduling specification, for 6 processes. This experiment checks to what extent the tools can deal with large, simple processes (2191 CSP processes with 14601 prefix operations and 12393 external choices). Server is the example from Section 3. McCarthy1, FibGen and GenPrime can be found in Appendix A.

One can see that if a file is well tuned for FDR, then FDR can apply its compression techniques [25] very effectively, and FDR is considerably faster than ProB (e.g., Crossing). On the other hand, for specifications which are not specifically designed for FDR, the speed difference can be as dramatic, but in the favour of our tool. E.g., our tool deals better when subprocesses are large, but the whole system is not (FibGen). Our tool also seems to be faster at dealing with large automatically generated CSP files (Scheduler0_6_Bexp) and possibly with recursion (McCarthy0). FDR seems to have a problem with larger state spaces (McCarthy1 and McCarthy2; computing the McCarthy function for x>100 involves no recursion).

We can conclude that there are extreme differences between the two tools, and that they thus complement each other quite nicely.

7 More Related Work, Discussion and Conclusion

Other interesting tools for CSP-M are Casper [21] and CSP-Prover [12]. Apart from our own earlier work [15, 5], our combined CSP and B tool is most strongly related to the csp2B tool [3]. The csp2B tool allows specifications to be written in a combination of CSP and B by compiling the CSP to a pure B representation which can be analysed by a standard B tool. The CSP supported by csp2B covers a small subset of CSP. On the Z and CSP side there is the model checker for Circus [31]. As far as using Prolog for process algebras is concerned, the model checking system XMC implemented in XSB Prolog contains an interpreter for value-passing CCS [23]. This version of CCS is much smaller in scope than CSP-M.

We have covered almost all of the CSP-M language in our interpreter, but a few constructs have not yet been implemented at the time of writing. Basically, only two primitives (extensions and productions) are not yet supported, some restrictions on channel input patterns apply[9] and mixing of closure operations with set operations (especially diff) is not yet fully supported. On the other hand, we do support some valid CSP syntax which generate errors with FDR.[10] Also, we have not yet tuned our tool for speed. But as the empirical results have shown, the tool is already applicable to realistic specifications. In future, we also plan to apply some of ProB' symmetry reduction techniques [18, 19] to CSP-M.

In conclusion, we have presented a new FDR-compliant tool for validating CSP-M specifications, with many unique features making it complementary to existing tools. It is much broader in scope than the implementation in [15, 5]. While developing the tool, we have uncovered some problematic issues with CSP-M, such as multiple meanings of the dot operator and that substitutivity of equals does not hold. We foresee the following potential practical applications of our new tool:

- Validation of combined B and CSP specifications. To our knowledge our toolset is unique with respect to this capability. In future it should even be possible to validate combined Z and CSP specifications as well as combined Event-B and CSP specifications within the the Rodin [4] platform [1]. Note that one can also use ProB's refinement checker to check pure B specifications against pure CSP specifications for property validation; see [5].
- Teaching process algebras and CSP. Source code highlighting of events is especially useful for beginners. ProB's interface and graphical visualisation have also proven to be very useful in a teaching environment.

[9] They can only contain variables, tuples, integers and constants; e.g., the doubtful `ch?(y+1,x)` is not accepted by our tool, but is accepted by FDR.

[10] E.g., FDR sometimes generates an error "unsupported comparison processing communication error" when using an if statement on parts of associative tuples received on a channel.

– Existing CSP applications, as a complement to FDR, so as to find certain errors more quickly and highlighting errors directly in the source code, and for the first time providing an LTL model checker [20] for CSP.
– Supporting new kinds of specifications, employing infinite state processes or sub-processes.
– Safety critical applications where the specification needs to be validated by independently developed tools.

Acknowledgements

We are grateful for feedback on our new tool from Neil Evans, Steven Schneider, Helen Treharne, Edd Turner and for feedback on our paper from Daniel Plagge and anonymous referees. We are also grateful to Phil Armstrong, Michael Goldsmith, and Bill Roscoe, for insights about the semantics of CSP-M.

References

1. J.-R. Abrial, M. Butler, and S. Hallerstede. An open extensible tool environment for Event-B. In *ICFEM06*, LNCS 4260, pages 588–605. Springer, 2006.
2. D. A. Basin, E.-R. Olderog, and P. E. Sevinç. Specifying and analyzing security automata using csp-oz. In F. Bao and S. Miller, editors, *ASIACCS*, pages 70–81. ACM, 2007.
3. M. Butler. csp2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12:182–198, 2000.
4. M. Butler, C. B. Jones, A. Romanovsky, and E. Troubitsyna, editors. *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*, LNCS 4157. Springer, 2006.
5. M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pages 221–236, Newcastle upon Tyne, 2005. Springer-Verlag.
6. M. Fontaine and M. Leuschel. Typechecking csp specifications using haskell (extended abstract). In *Proceedings Avocs 2007*, pages 171–176, Oxford, UK, 2007.
7. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement — FDR2 User Manual (version 2.8.2)*.
8. Formal Systems (Europe) Ltd. *Process Behaviour Explorer (ProBE User Manual, version 1.30)*. Available at `http://www.fsel.com/probe_manual.html`.
9. P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 421–497. Oxford Science Publications, Oxford University Press, 1998.
10. C. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
11. G. Hutton and E. Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.
12. Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, LNCS 3440, pages 108–123. Springer, 2005.

13. T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings Conference on Functional Programming Languages and Computer Architecture*, LNCS 201. Springer-Verlag, 1985.
14. S. P. Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
15. M. Leuschel. Design and implementation of the high-level specification language CSP(LP) in Prolog. In I. V. Ramakrishnan, editor, *Proceedings of PADL'01*, LNCS 1990, pages 14–28. Springer-Verlag, March 2001.
16. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
17. M. Leuschel and M. Butler. Automatic refinement checking for B. In K.-K. Lau and R. Banach, editors, *Proceedings ICFEM'05*, LNCS 3785, pages 345–359. Springer-Verlag, 2005.
18. M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry reduction for B by permutation flooding. In *Proceedings B2007*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.
19. M. Leuschel and T. Massart. Efficient approximate verification of B via symmetry markers. In *Proceedings International Symmetry Conference*, pages 71–85, Edinburgh, UK, January 2007.
20. M. Leuschel and D. Plagge. Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In Y. A. Ameur, F. Boniol, and V. Wiels, editors, *Proceedings Isola 2007*, volume RNTI-SM-1 of *Revue des Nouvelles Technologies de l'Information*, pages 73–84. Cépaduès-Éditions, 2007.
21. G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998.
22. L. Naish. An introduction to MU-Prolog. Technical Report 82/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, March 1982 (Revised July 1983).
23. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In O. Grumberg, editor, *Proceedings CAV'97*, LNCS 1254, pages 143–154. Springer-Verlag, 1997.
24. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1999.
25. A. W. Roscoe, P. H. B. Gardiner, M. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking csp or how to check 10^{20} dining philosophers for deadlock. In *Proceedings TACAS '95*, pages 133–152, 1995.
26. J. B. Scattergood. *Tools for CSP and Timed-CSP*. PhD thesis, Oxford University, 1997.
27. S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 1999.
28. F. Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com`.
29. H. Treharne and S. Schneider. How to drive a B machine. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *ZB'2000*, LNCS 1878, pages 188–208. Springer, 2000.
30. H. Treharne, S. Schneider, and M. Bramble. Composing specifications using communication. In D. Bert, J. P. Bowen, S. King, and M. A. Waldén, editors, *ZB*, LNCS 2651, pages 58–78. Springer, 2003.
31. J. Woodcock, A. Cavalcanti, and L. Freitas. Operational semantics for model checking circus. In *Proceedings FM'05*, LNCS 3582, pages 237–252. Springer, 2005.

A Some Benchmark Files

McCarthy1

```
channel out:{0..999999}
McCarthy(n) = if n>100 then
                  n-10
               else
                  McCarthy(McCarthy(n+11))
Test(n,m) = if n<m then out!McCarthy(n) -> Test(n+1,m) else STOP
MAIN = Test(0,10000)
```

FibGen

```
channel out,gen: {0..9999}
FibGen(N,M) = if M<10000 then out!M -> FibGen(M,N+M) else STOP
Take(n) = if n>1 then out?_ -> Take(n-1)
                 else out?x -> gen!x -> STOP
MAIN = FibGen(0,1) [| {| out |} |] Take(5)
```

GenPrime

```
 channel out,comm:{0..99999}
 MAIN =  Gen(99999,7) [| {| comm |} |] Gen(99998,29)
 Gen(x,d) = out!x -> if x<d then Gen(x,d) else Gen(x-d,d)
           []
           comm.x -> STOP
```