Communicating Process Architectures 2014
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2014
© 2014 The authors and Open Channel Publishing Ltd. All rights reserved.

Reflections on the need to de-skill CSP

A.W. ROSCOE

Department of Computer Science, University of Oxford, UK
Bill.Roscoe@cs.ox.ac.uk

Abstract. CSP is a hugely expressive language, and in FDR it has a remarkably effective model checker, and yet it is not as widely used as these things might suggest. In this paper we argue that this is at least in part because it requires too much expertise to use CSP effectively in a way that maps systems efficiently to a form suitable for FDR.

Keywords. CSP, FDR, model checking

Introduction

The combination of CSP and FDR is able to model and reason about a wide range of systems, giving accurate and efficient models, where "efficient" means that the models contain no extra states that are by-products of the modelling, only ones that are natural in the system being modelled. However such modelling has the reputation for being difficult, and in truth the number of people with the necessary expertise to do such modelling is not nearly as large as this author would like.

In this paper we conjecture several reasons for this, and suggest ways of getting round some of them. We look at four classes of issues, namely ones arising from the need to create models that FDR can handle, ones arising from the sheer expressive power of CSP, the effective use of the techniques CSP and FDR support for coping with the state explosion problem, and the construction of specifications. The following sections cover these in turn.

This paper discusses CSP and FDR as they are in mid 2014, namely with CSP as described in the author's second book on the subject [23] and soon after the initial release of FDR3 [5]. Some of the developments anticipated herein have been discussed with others and owe much to them, in particular Tom Gibson-Robinson, Sasha Boulgakov, Michael Goldsmith and Colin O'Halloran.

In this paper we assume a reasonable familiarity with CSP as described in the introductory sections of the author's books [19,23], and expect that most readers will have at least seen FDR2 or FDR3 used.

1. Keeping on the right side of FDR

FDR [18,1,5] is an explicit refinement checker for CSP that uses model-checking techniques. Here, by *explicit* we mean that, at least at the top level, in its usual mode it enumerates the states of the system it is analysing one at a time. It compares two processes to see if one, normally the implementation being examined, refines a second one, normally representing the specification that we want the implementation to satisfy. Refinement is judged relative to the observability of chosen types of behaviour in the two processes, normally one of *traces*, the combination of traces and *failures* or the combination of failures and *divergences*. (The respective refinement relations based on these models are written $\sqsubseteq_M$ where M is one of T, F and FD.)

It supports a wide range of techniques for trying to cut the number of states it has to visit, and runs highly optimised algorithms for executing its central model of what it expects a system described in CSP to look like. Each of these adds a dimension of complexity to the use of CSP and FDR. It is the second of these that we concentrate on in this section.

In a nutshell, that model is of some *high-level* combination $C[P_1,\ldots,P_n]$ of *low-level* components P_i, where the components have to be small enough to compile into explicit state machines before the overall combination is considered. $C[]$ takes these and switches between a few (usually one) way of combining subsets of states of the P_i using combinations of parallel, hiding and renaming operators: FDR's *formats*. In this paper we will concentrate on the one-format case for simplicity.

This means that FDR does not at present cope with anything like the full generality of CSP. However, since it is such a powerful tool it is understandable that those using CSP since FDR has existed have tended to write CSP that FDR can handle efficiently. This had probably had significant effects on the development of CSP itself, making us concentrate, for example, on static rather than dynamic networks, and in general on the finitary. It has certainly had effects on the sorts of models people build, since FDR historically requires:

1. All processes must be finite-state, whether the process under consideration as implementation or, in the case of refinements, the specification on the left-hand side. This applies not only to complete processes used as specifications or implementations, but also to their components.
2. The ranges of parameters used in each recursive definition over terms such as the infinite buffer definition

$$\begin{aligned} BUFF_{\langle\rangle} &= left?x \rightarrow BUFF_{\langle x\rangle} \\ BUFF_{s\frown\langle a\rangle} &= ((left?x \rightarrow BUFF_{\langle x\rangle\frown s\frown\langle a\rangle}) \sqcap STOP) \\ &\qquad \Box\ right!a \rightarrow BUFF_s \end{aligned}$$

   needs to be kept finite. The above definition, of course, violates this since the parameter varies over all finite sequences from the underlying data type. So we cannot use the above at any point in an FDR check.
3. No recursion is possible through a variety of CSP operators such as parallel, hiding, renaming and the left-hand sides of sequential composition.
4. All but small processes are chiefly formed as the parallel combination of sequential components, which individually need to have size which is significantly smaller than overall processes. (A small amount of manipulation is possible outside the parallel composition using operators such as sequential composition, prefix and interrupt thanks to FDR's implementation of multiple formats within one check.) In many cases such factorisation is natural, because the system is naturally parallel, but in others we find ourselves expressing a process as a parallel combination solely so that FDR can handle it.
5. Alphabets are finite, of a size that FDR can store and perform moderately complex operations on.

All of these things can put obstacles in the way of people modelling systems for FDR. We comment briefly on the effect of each of these below.

1. While it would obviously be good to be able to prove properties of infinite-state systems, and there is a huge amount of research on specialised aspects of this problem, it is unrealistic to expect at this time that a general-purpose tool will prove properties of infinite-state systems unless it is programmed to spot various special cases and handle these appropriately. Knowing what to do to trigger these, in most cases, would itself

probably require arcane programming skills. (There are exceptions to this in the area of infinite data independent types, which we will discuss below.)
However one might reasonably hope to be able to explore part of the state space of an infinite-state system, exploring for counter-examples to specifications. This could be by putting our process in parallel with a regulator process (e.g. limiting the number of actions) that is believed to keep it finite, or by just leaving a non-terminating check running until the user or system gives up on it. Both of these are at present impossible.
More annoying is the impossibility of putting an infinite-state specification on the left-hand side of a refinement check. Many natural specifications, including obvious ones such as $tr \downarrow a \leq tr \downarrow b$ (which means either that the number of a events does not exceed the number of bs, or that the sequence of values communicated along a is a prefix of the ones communicated along b) are infinite state. For though, in many cases, a finite-state implementation will only exercise a finite portion of the state space of such a specification, FDR is at present incapable of handling specifications such as the most general buffer: $Buff_{\langle\rangle}$ as defined above. There are a number of ways of working around this situation, the most common of which is to find a finite-state refinement of the infinite-state specification that the implementation satisfies. Doing this properly can be a significant challenge.

2. In many way the most annoying aspect of this restriction is that it applies to the component processes of parallel compositions. For even if in the actual network, only a small set of parameter values may be visited, we cannot use component which when left by themselves can visit either infinitely many parameter values or too large a finite set. The best known example of this is the intruder process in cryptoprotocol models such as those of [24]. The natural CSP model of this intruder – which has an initial knowledge of cryptographic data, absorbs further such data by overhearing, trapping or being sent messages on *learn*, and can always send any message buildable from this knowledge by feasible cryptographic means on *say*, is $Spy(initial)$ where

$$\begin{aligned} Spy(X) = {} & learn?x \rightarrow Spy(close(X \cup \{x\})) \\ & \Box\; say?x : X \cap Messages \rightarrow Spy(X) \end{aligned}$$

where $close(X)$ is the set of "facts" that the intruder can build from the members of the set X. The range of such facts, for a given protocol, is typically in the range 100 to several thousand. This size of range is well within the capabilities of FDR, but the powerset of it, over which X ranges, is far too large to be enumerated.
Thus coping with FDR's limitations proved to be an important part of the research in developing our models of this type of protocol, and led to the creation of the parallel model of the intruder, with one process per potentially learnable fact, together with the development of the *chase* operator on transition systems that forces an arbitrary τ action (equivalent here to an available inference of a new fact for the intruder) to occur if one is available – an operation which is valid on this type of system but not on all.
Of course enumerating the full state space of the intruder (given the statistics above) would be impossible even at the enhanced speed at which FDR runs the high-level check once compilation is complete. While we do get an advantage from using a faster mode of operation than compilation to run the intruder, the main advantage comes because, in the context of the model where it runs with agents running the target protocol accurately, only a tiny proportion of the possible states of the intruder are visited. The main gain is from avoiding compiling a huge number of states unnecessarily.
More generally, the need to restrict the parameter spaces of all processes, and particularly component processes, to be finite and not over-large is sometimes inconvenient

and can require extra ingenuity, up to including that demonstrated in the example above.

3. The reason why the forms of recursion discussed above (e.g. through parallel and hiding) are banned is because they lead to trees of terms that simply keep expanding in a syntactic sense, even if there might be reductions that are possible via knowledge of the laws of CSP. Use of such recursions was very common in the earlier, pre-FDR, age of CSP, though typically to build interesting models of infinite state systems including dynamic networks. Typical amongst these "old-fashioned" examples were

$$\begin{aligned} Stack &= in?x \to Cell(x);\ Stack \\ &\quad empty \to Stack \\ Cell(x) &= out!x \to SKIP \\ &\quad \Box\ in?y \to Cell(y);\ Cell(x) \end{aligned}$$

$$B = left?x \to (B[right \leftrightarrow left]right!x \to COPY)$$

which respectively implement unbounded stack and buffer processes without using infinitely parameterised recursions.

4. The treatment discussed above of the cryptoprotocol intruder is one example of how models often need to be re-factored to square with FDR's preference for the parallel combination of relatively small component processes. As FDR technology and the computers it runs on have developed, the effective bound on the size of such a component process has grown from a few thousand to over a million, but even in large examples it can handle one frequently hears the complaint that FDR spends a lot more time compiling the component state machines than it does running the eventual check. That is usually caused by large component processes.
One frequently encounters this problem when creating CSP models of processes with variables/parameters. These often occur when modelling programs created in different notations, such as

   - Statecharts: a single sequential component
   - Shared variable programs: a sequential thread process
   - State machine models arising from notations such as B, UML and ASD [8].

   In all of these, processes frequently combine a relatively small control state (e.g. something like program counter in the case of a sequential thread process) with the possibly large combinations of values contained in its variables or parameters, where these parameters include the ranges of values input at different points. So for example a program with 10 control states and six integer variables ranging over $\{0..9\}$ would have up to 10^7 states. The natural CSP models of such components would mean that FDR will compile each such state separately, even though relatively few of them may be visited in the context of the complete check.
CSP implementations of the above, such as those by the author of Statecharts [26] and shared variable programs (SVA, see Chapters 18 and 19 of [23]) are sometimes factored into a separate process for the control state and for each variable. This is much more difficult to do properly than a straightforward sequential model of the component, particularly when (as with the use of *chase* in the intruder model) we need to use techniques such as *chase* and compression to eliminate the additional states created by the actions needed (e.g. variable reads and writes) to make the parallel model work.
Just as with the intruder, there are two potential sources of speed-up from factoring an apparently sequential component into a parallel composition. The first comes even when much of the component's state space is visited in the check proper, from a

(hopefully) significantly reduced amount of compilation, which is replaced by faster functions enumerating and compressing the composition. The second comes where most of the states that would be compiled in the sequential model are not visited. The first of these can make a noticeable difference to performance, but it is the second type where huge advantages are gained.
Such models can also make the use of FDR more scalable. This is certainly true for SVA, where many small examples would work very well with a model where all a process's variables were folded into it as parameters (with writes to shared variables being synchronised between threads). However the use of the factored model makes SVA scale far better to threads with more variables or ones with larger ranges. (For example the 1.2 trillion state check reported in [6], of a version of the bakery algorithm, gives each of six threads access to 6 shared and one local variables with a range of 31 ($\{0\dots30\}$) values each, plus six shared booleans and various local counters and booleans. Compiling such a parameterised sequential thread, with billions of states, would be way out of range.)

5. The restriction to reasonably-sized alphabets, like the similar restriction to parameter spaces, means that in modelling real systems where types are larger or infinite, we have to make abstractions to smaller types. Sometimes such abstractions can be made rigorously, as in data independent reasoning [10]. Sometimes we have to guess that the ranges selected cover all interesting states, or merely hope.

We now propose some possible approaches which might serve to reduce some of this complexity.

1.1. Lazy compilation

Lazy compilation takes us away from the model where sequential components are compiled into completely enumerated state machines, so that compilation is over by the time that normalisation (for the specification) or running the check using FDR's supercombinator model begins.

Rather it will compile only those parts of a process that are demanded by the running of an actual check, with the functions of the compiler being called by need as a check evolves. When we choose to compile a particular component lazily, only the transitions of its initial state are compiled up-front, and subsequent states are compiled only when visited.

This will bring an advantage in cases where only a small portion of a large machine is visited in a check. It will allow checks involving infinite specifications and infinite component processes, which can potentially be infinite either because of infinite parameter space (where "infinite" includes too large finite spaces such as the set parameter in the intruder example) or because of breaching the syntactic constraints on recursion discussed above.

This advantage might appear because, as with the cryptoprotocol models, only a small portion of a lazily compiled component are visited in the context of the natural, complete model. Alternately it might appear because a counter-example is found quickly, or because the natural system is placed in a constraining context which indirectly constrains the components.

It will allow infinite-state specifications provided only finitely many of these states are exercised in a given refinement check. This will also require the use of *lazy normalisation*, a function that has been present in FDR for years (though not in the presently released version of FDR3) and which does for the computation of the normal form used on LHS of refinement checks exactly what lazy compilation does for the computation of component state machines. This use thus corresponds to handling the specification doubly lazily. We might note here that in the common case where the LHS of a refinement check is just a sequential component, there can be no point in *compiling* it lazily unless the *normalisation* is also lazy, for normal eager normalisation would force the compilation of the whole process.

The fact that a finite-state *Impl* refines the infinite-state *Spec* does not guarantee that the above method will be able to prove it finitely. Consider the trace specification given by the process P_0 defined

$$P_n = (n > 0 \,\&\, down \to P_{n-1}) \,\square\, (up \to P_{n+1})$$

and the implementation $Ups = up \to Ups$. Though $P_0 \sqsubseteq_T Ups$ is obvious to a human, running FDR with the two lazy operators applied to P_0 would generate a state pair[1] corresponding to every (P_n, Ups) and therefore fail to terminate.

This can happen when one can, as here, find an infinite set of the normal form states of the specification which are refined by the same state of the implementation.[2] This may well be less common for failures-divergences refinement than for trace refinement, because it is harder there for sets of processes to be consistent: over both the traces and stable failures model, any set of processes is consistent because of the existence of the refinement-top elements of these models, usually identified with *STOP* and the simply divergent **div** respectively. Over failures-divergences there are infinitely many different maximal, and inconsistent, elements, namely the deterministic processes. For example the failures-divergences specification of a buffer, whose normal form has essentially the structure of the process $Buff_{\langle\rangle}$ defined earlier, has all its normal-form states inconsistent. It follows that any attempt to prove that a finite-state process refines the doubly lazy $Buff_{\langle\rangle}$ is guaranteed to terminate on a large enough computer.

Lazy compilation is particularly attractive for replacing complex parallel implementations of nodes in parameterised systems. In making up such parallel implementations we usually, for reasons discussed above, need to apply some sort of compression-like function to the parallel constructions of components. There are two possible approaches to this with the present FDR:

- Use *chase*: this is not always applicable, for example it is not with SVA (at least using the present structures) because (non-τ) reads and writes to shared variables from other threads might be discarded by *chase* in favour of the hidden ones by the local thread. However if *chase* is applicable it has the advantage of working out the states of the component lazily when it is run within the context of the complete system. Thus, just as with lazy compilation itself, we are not forced to evaluate unneeded states.
- Use state space compressions such as *diamond* and bisimulations. This is always applicable, but has the disadvantage that all the states need to be enumerated before the compression is possible. In this case it is possible that the parallel models of components will be too large to compress when the lazy compilation approach on the sequential models would work perfectly well. For example attempting this sort of compression the parallel intruder model would certainly fail in all but the most trivial of cases.

Lazy compilation of natural parameterised components is therefore most likely to be attractive from the point of view of efficiency in cases where *chase* is not applicable, but it is always likely to be better from the points of view of ease of programming and clarity, the latter being potentially very important in contexts where we are trying to build a proof of a (for example) safety-critical system.

It follows that lazy compilation promises to alleviate many of the obstacles identified in this section.

[1] P_0 and its trace normal form have identical structures.

[2] In the absence of this for a particular infinite-state *Spec* and finite-state *Impl*, the FDR check in which *Spec* is treated doubly lazily is guaranteed to terminate.

While we first identified lazy compilation as an interesting potential technology in the mid-1990s when we were developing cryptoprotocol models, on the grounds that it was a potential solution to modelling the intruder, the architectural complexity it would bring coupled with the invention of the parallel-plus-*chase* solution to the problem in hand has meant that it has never, to date been an option in FDR. This should be put right in the near future thanks to the doctoral project of Sasha Boulgakov, but this is not the right place to report on the details of that work or the approach being used by him.

It seems very unlikely that we would want to make lazy compilation the default mode for FDR – except perhaps if we were constructing a particularly straightforward version for inexperienced users. This is because it seems certain that lazy compilation will carry a significant penalty in speed and memory consumption in cases were all of a component process or specification's states are used in a check.

Similarly there are potential problems with its use in some cases: there can be no point in applying compression functions such as *diamond* and bisimulations to a process P that is being lazily compiled unless the unit being compressed itself severely limits the reachable state space of P.

FDR3 has very successful parallel implementations for multi-core and clusters, and there are reasons to believe that effective parallelism will be more difficult in the presence of lazy compilation.

Time will tell, and certainly lazy compilation will remove some unpleasant barriers to the use of FDR. However it remains to be seen what the balance of ease will be between the possibility of avoiding some of the more difficult forms of CSP coding, and the skill needed to ensure that lazy compilation and the lazy normalisation of specifications are used in appropriate ways.

1.2. Automation of data independence analysis

Data independence, first studied in the context of CSP by Lazić [10], provides a way of generalising results about systems analysed defined over small types to results about the same parameterised system defined over general types.

A program is only susceptible to this type of reasoning if it uses the type in a restricted way, basically handling members of the type in question as uninterpreted tokens, though depending on the exact flavour of data independence used it may be allowed to compare them for equality or even index arrays over them.

There are basically three different styles of data independent reasoning that have been applied in CSP: we discuss these briefly in turn.

1.2.1. Threshold theorems

The best-known sort of data independence reasoning is to analyse a specification and implementation to determine a *threshold*: a size beyond which the result of the check $\mathit{Spec}(T) \sqsubseteq \mathit{Impl}(T)$ does not change, for the data independent type T. Lazić's results cover three sorts of threshold:

- The threshold is usually 1 when *Spec* and *Impl* have no equality tests involving members of T, and *Spec* does not constrain which member of T *Impl* communicates at any point. Thus it is possible to verify properties such as deadlock- and divergence-freedom for *Impl* without equality tests by checking the case where $T = \{1\}$.
- The threshold is usually 2 when the same conditions apply except that *Spec* is allowed to constrain what member of T is communicated by *Impl*. For example the check that $\mathit{Buff}_{\langle\rangle} \sqsubseteq_{FD} P$ can be restricted to $T = \{1, 2\}$ when P is data independent in T without equality checks.

- In cases where there are equality checks one can sometimes prove from the structure of these that one of the two cases above still applies, but otherwise we have to compute an ad hoc threshold by counting the number of separate values of type T that co-exist in the check at one time.

The property of data independence in type T is very close to it being *polymorphic* in the sense of conventional type checking. Therefore now that we have an integrated type-checker in FDR3 it should be relatively straightforward to modify it to check to see if a type is data independent. Hopefully it will give feedback when this fails, just as the type checker does.

Calculating thresholds in the first two cases above, and upper-bound thresholds in the third cases, should be a reasonably straightforward matter. This seems like a worthwhile addition to FDR, making the central case of data independent reasoning easy to use: a case where it becomes possible to prove results about certain infinite-state and over-large finite-state systems by means of a relatively compact refinement check.

1.2.2. Data independent induction

This technique, introduced by Creese and the author [3,4] allows us to extend the scope of data independent reasoning into an area forbidden in the basic case above, namely arbitrary-sized networks where all the nodes treat the type of node identities data independently. It expects the verifier to invent a (usually sequential) process as the inductive hypothesis (referred to as a *Superstate* process) that models partially constructed networks, and then uses the techniques of the previous section to verify that this hypothesis (i.e. that the partial network refines the *Superstate*) is true for a class of subnetworks that includes the complete ones for every allowable network. We do not present this in detail here, but the reader can find it set out, with an example, in [23].

Clearly the automation of basic data independent reasoning envisaged in the previous section would be helpful in this, but it does not solve the central problem of formulating the *Superstate*, which is by some way the most challenging aspect of this technique.

The automatic creation of inductive hypotheses is an active, though difficult, topic of research in verification, and a reasonably amount of effort has been expended on it devoted to its application in data independent induction. The author believes that in order for this technique to be accessible to more than a few specialists, some significant success in the automated derivation of inductive hypotheses will be needed.

1.2.3. Ad hoc data independence

The original motivation for working on data independence in CSP was to prove that general, rather than small finite, implementations of cryptoprotocols were correct and thereby build these general proofs. The particular nature of the protocol models meant, as it turned out, that the threshold theorems alluded to earlier only applied in very limited ways to them.

It did prove possible to apply data independent reasoning to protocol models, as shown in [20,2,9]. One of the main tricks used was using explicit dynamic mappings that reduce infinite sets of cryptographic objects to small finite sets – in effect equivalence classes based on the states of knowledge of models of trustworthy nodes within the network.

Unfortunately it is hard to see how, other than by checking for data independence and supporting threshold reasoning where it applies, one can envisage any systematic support for this type of analysis.

1.3. Symbolic execution

The term "symbolic model checking" has two quite different interpretations. The more common is as a term for methods which encode state spaces in logical form (typically propositional calculus) and then use logical tools such as SAT checkers to analyse them.

The other meaning, and the one we have in mind here, is replacing the expansion of the data types communicated along channels and used in data parameters of component processes, by symbols representing arbitrary such values. Thus instead of the familiar process

$$COPY = left?x \rightarrow right!x \rightarrow COPY$$

having $N + 1$ states, where the communicated type has size N, it would have two, where one would have a symbol for a value.

This is an attractive alternative to data independence, and would apply in substantially the same examples. It offers one of the best options for calculating more exact thresholds in data independence, though quite possibly a full symbolic check would be more efficient. It would also be very helpful in calculating the inductive hypotheses required for data independent induction, though there it would not be a complete solution.

However symbolic checking of the sort we are discussing carries difficulties with it, not the least of which is keeping track of what becomes known of the relationship between symbols of the same type as a check evolves. The latter is particularly true on the specification (normal form) side of a refinement check, so it is possible that an implementation of this type of check would need to use severely restricted specifications.

1.4. Summing Up

From the discussion in this section the author concludes that, of the options considered here, lazy compilation and support for basic data independence analysis are well worth doing. Both offer the chance to reduce the skill required for building and analysing complex systems, with lazy compilation also offering to broaden the applicability of FDR and in some cases offering improved efficiency.

2. An embarrassment of riches

We have only come to understand in the past few years just how expressive the CSP notation is. In particular the author [22,23] showed that, slightly extended as set out in [23], it is a universal language for a central class of languages for which one can define an operational semantics over labelled transition systems. In other words, any operator over LTSs with a *CSP-like* operational semantics can actually be translated into CSP in such a way that, the operational semantics of the translation is strongly bisimilar to that of the operator in question for languages without CSP termination $\checkmark$, and otherwise very close to it in a sense defined in Chapter 9 of [23], where the termination case is set out and proved in full. A CSP-like operational semantics is one which each operator

- Never copies an argument that has already started: processes that might have performed actions cannot be closed.
- Has no negative premises in clauses: actions only depend on the ability of some set of arguments to perform a specified action each (except in the case of the next bullet point all the actions performed by the arguments must be visible, though not necessarily all the same action).
- If the operator's first-step actions depend on one of its arguments P, then it must allow P to perform a τ action without otherwise affecting the state (with the τ becoming an action of the operator).
- Amongst those arguments who are still present after an action, ones who have not participated remain in the states they were before, and ones that have participated move to the result state of their contributing action.

The definition allows argument processes to be discarded both by actions to which they have contributed (visibly) and by ones (never promoted τs) involving other actions. It allows complex patterns of synchronisation resulting either in visible or τ actions.

The result – particularly the strong bisimulation aspect – explains why CSP is good at producing efficient, "tight" models of diverse systems, in other words models without extra states that are simply an artifact of the model.[3]

The CSP "machine" used in the proof uses some of the almost devilish tricks that have been devised to express things in CSP over the years. These include:

- Multi-way synchronisation. This is of course allowed in the CSP-like world, but it is also used in the machine to implement simpler interactions. Much use is made of regulator processes that synchronise with processes (which may well have internal synchronisations of their own) so that we can select which ones are allowed and which are not.
- One-to-many renaming. If we want a single event in a component process is play a number of alternative roles in the result – possibly synchronising with different combinations of events from other processes, with some being hidden and some not – one can rename it to a different event for each role. In the proof of the CSP-like languages theorem this is taken slightly beyond the extreme of renaming every event to every partial function from the arguments of an operator to their events, where the image of this argument is the event being renamed. Thus we rename an event to every potential role it might have in synchronising with a collection of other events to deliver a specified result event. Which of these combinations actually happen is then left in the hands of synchronisation with a regulator process.
  Other uses of many-to-one renaming include the CSP cryptoprotocol model, where communications between trustworthy nodes are renamed both to themselves and extra copies for the intruder to manipulate, simulating its assumed control of the network.

There are many other uses of the combination of one-to-many renaming and regulator processes. For example one can use it to implement operators that rename different instances of the same event to different targets, or to hide selected copies of one event. Those familiar with it have come to regard it almost as a magic wand.

Other constructs in the expressiveness machine include harnesses to put processes in to allow them to be closed down either by their own actions or by actions of external processes, when the operator being simulated demands one of these things. These two possibilities are implemented using the throw $P\,\Theta_A\,Q$ and interrupt $P \triangle Q$ operators respectively, the difference between the two being that the close-down of P and start-up of Q is instigated by P in the case of throw and Q in the case of interrupt.

The expressiveness of CSP is further amplified by the addition of the priority operator [23,25], which goes beyond the "CSP-like" world because it does have negative premises. An interesting theoretical challenge is to formulate a similar characterisation to CSP-like which takes the additional capabilities of this operator into effect.

We have identified the possibility of supporting CSP-like operators directly in FDR3 rather than having to rely on translation to CSP. That would certainly make the analysis of notations involving such operators significantly easier, though of course the overall ease of use would depend heavily on the usability and familiarity of the CSP-like language represented by the new notation.

The theory underlying the expressiveness result also suggests that one might want to find a general, and very possibly graphical scheme, for setting out the action rules (supercombi-

[3]Examples of extra states come, for example, in notations where a communication has to be factored into separate send and receive actions.

nators) of a given system. If an interface can be made intuitive it might provide an interesting alternative to having to define such rules in a process algebra. We imagine that process algebra would remain one of a number of alternatives for defining the component processes for such a composition.

2.1. Summing up

CSP is an immensely expressive language, but accessing this expressiveness can require very detailed and ingenious programming in CSP. We have concluded that support for the implementation of CSP-like languages would certainly help those familiar with other notations that fit in this category, and that research may be desirable on convenient graphical ways of specifying the sorts of harness for CSP combinations that supercombinators can implement.

Of course another way of approaching this problem would be to add further operators into CSP to make expressing particular sorts of concept easier, but the author has not come up with any particular ideas that would be useful in a sufficiently wide range of examples to make the extra learning and other requirements for adding a new operator worth it.

3. Taming the exponential dragon

Perhaps the most significant practical challenge to all sorts of model checking is the state explosion problem, by which the state-space size of an example tends to grow exponentially with the number of parallel components, the number of variables and for other reasons.

This has led to the development of technologies such as data independence as described earlier, state space compressions and similar ways of reducing the sizes of the machines analysed, partial order methods for avoiding visiting some of the states and transitions of a machine, symmetry reduction, and the development of SAT-based technologies. Another example is CEGAR (Counter-Example Guided Abstraction Refinement), a technique that attempts automatically to find a smaller anti-refinement of a system that nevertheless satisfies the original specification. All of these, and more, have been attempted in conjunction with FDR: most of them are described in at least one of [23,16,17,14,15].

The problem with these techniques is that none of them is a reliable solution to the state explosion problem. Unlike the case with FDR's basic explicit mode where each state of the system is considered in turn, all of them have a risk of not working well and event making things worse.

A problem with some of them is that actually using them requires a good deal of skill and judgment. For example in using compression, the programmer has to decide:

- How to map the network structure of a given network into a tree of CSP operator applications in order to get good compression performance. The classic example of this is with the dining philosophers, which compresses ideally (i.e. the compressed state space does not grow as further processes are added to a partially completed chain) if built up as a chain of processes, adding consecutive philosopher-and-fork pairs (of the same index) so that each added pair always links to the network so far, with the newly synchronised actions being immediately hidden. On the other hand, failure to do this hiding or putting the network together in a significantly different order, such as combining all the philosophers, and all the forks, and then putting these two collections in parallel, will result in exponential blow-up.
- Which compression, or *chase*, to apply, and in the latter case is it valid in this context.
- Is it better to use a modular compression strategy, where the network is decomposed into parts, and each of these compressed (so compression is being applied at one level in the syntax tree); or leaf compression (where just the leaf processes are compressed);

or inductive compression (what is implied for the philosophers above: processes are added one at a time with the result being freshly compressed each time); or something intermediate between these?

These decisions will be based both on the effectiveness of the compression in terms of state-space sizes and the time and space taken for the compression itself. The author has compressed machines with more than 10^7 states, but these operations do take much more time and space than simply running a refinement check over the same process uncompressed.

These decisions are complicated by the fact that the present implementations of compression are sequential, while of course FDR3's checking is done in parallel. It can perform multiple compressions in parallel, but inductive compression is inherently sequential if the underlying compression method is.

The only support that FDR currently provides for compression are the CSP_M scripts `compression.csp` and `compression09.csp` (the latter being an update created in 2009) which implement strategies such as leaf and inductive compression based on decisions made by the programmer about the order of compression and which compression operator to use.

The skill and effort required in applying compression would be reduced if the following were implemented:

- The compression counts (including input and output state sizes) were made available to the programmer.
- Compressions can be made to abandon (returning the original machine rather than a compressed one) if the compression is either taking too long or proves ineffective.
- We could implement automated support for deciding how best to compress an arbitrary network. Such support would certainly look at how much hiding is available how early when a particular syntax tree is chosen, and may well take the sizes of the components into account.
- Parallelisation of the compression functions would mean that decisions could be made more-or-less independently of the machine we expect a check to run on.
- It would be useful to have a check for the validity of *chase*. Such checks could be syntactic or themselves based on refinement checking, noting that for large examples we might need to use a partial check.

Moving beyond compression, we have already discussed automation to support data independence.

With the exception of SLAP [14], a tool for the static analysis of divergence that was integrated with the last versions of FDR2, none of the other methods discussed at the start of this section have moved to the point where they are really usable by the non-expert. They all have either technical or user interface issues that need to be resolved before they become serious general-purpose tools.

Nevertheless, the more of them we have, the more decisions need to be made by the programmer-verifier, against a background of uncertainty. That suggests two things:

- The approaches to conquering state explosion need to be made easy to use individually, both to stop them from looking forbidding and to make them easy to try separately or in combination to experiment over which works best.
- It would be useful to have some automated guidance, perhaps based on analysis of the individual components and their local interactions, about which method it might be good to try. Such guidance might be provided by some heuristics set out by humans, or even involve something like machine learning.

3.1. Summing up

The priority issue here is providing better support for the use of compression along the lines set out above, together with the data independence support already identified. Beyond this, thought needs to be given to supporting the user who is not accomplished expert when further tools for tackling state explosion are made generally available.

4. What blueprint for specifications?

Most model checkers use forms of logic (e.g. modal or temporal) as the way they specify properties of the systems they analyse. FDR uses the combination of a semantic model and a specification process, written in CSP, to judge an implementation.[4]

The fact that FDR checks for refinement gives it the ability to support software engineering ideas such as compositional and incremental development, that tools only supporting logical analysis cannot. However the great majority of uses of FDR relate to checks of the form $\mathit{Spec} \sqsubseteq F(\mathit{Impl})$, where $F(\cdot)$ is a simple context which (if not the identity function) might just hide some of its arguments actions or place a parallel restriction on which events can happen.

Therefore being able to formulate a specification as a CSP process is central to the successful use of FDR unless the user is only interested in a few standard checks. For trace checks the author does not think this is more difficult than any other notation, though of course those who come to use the tool with a lot of experience of using some logic such as LTL might well prefer the ability to use their personal favourite. Certainly providing a range of specification interfaces, essentially by demand from user communities, would be helpful.

It might be noted that notations such as LTL often contain infinitary properties that require different algorithms from standard FDR ones, though in this particular case it has been shown [11,12] how to handle large parts of the languages within FDR's existing functionality. In cases like this it will either be necessary to restrict the specification language so that it can readily be translated into the CSP refinement framework, or decide if the extra work and compromises are worth it. The areas that are likely to prove problematic are ones involving eventuality analysis. In other tools these are frequently handled by translation to Büchi automata and analysis of the infinitary acceptance criteria on these using cycle analysis. These sit outside the usual compositionality framework of CSP, and lead to algorithms that are intrinsically much less parallelisable than those of FDR. So, while there might well be an argument for using them in carefully thought-out cases, we would not want them to be come as pervasive in FDR as they are elsewhere.

In general, for any notation that specifies, in a finitary way, what events are allowed to occur in a process, it ought to be straightforward to add support for it to FDR.

Failures properties, namely ones that can specify the availability of particular actions as well as which can happen, are trickier to get right in CSP. Stating the minimum availability amongst communications in each state of a specification can be tricky. Even when just two actions (a and b) are permitted for a process, we have to choose between

- $a \to P \,\Box\, b \to Q$ (both a and b must be available)
- $a \to P \sqcap b \to Q$ (at least one of a and b must be available)
- $(a \to P \,\Box\, b \to Q) \sqcap b \to Q$ (b must be available), and symmetrically for a
- $\mathit{STOP} \sqcap (a \to P \,\Box\, b \to Q)$ no requirement on what is available.

[4]There are exceptions to this rule arising from FDR's ability to check for deadlock, divergence and nondeterminism, and when the implementation appears in a complex context in the check, or on both sides of the check [21,13]. However the case quoted here is the central one, and to some extent all the other possibilities for FDR are reduced to refinements that need checking.

It seems to the author that there is a strong argument for attempting to create a framework in which a process representing the allowed traces of a process is annotated with the actions (or alternative actions) that an implementation process must have available in order to satisfy this specification. Such a framework might, for example, be graphical.

There is not that much to say about the divergences part of failures-divergences specifications, since usually such specifications simply ban divergence. In some cases, however, specifications are allowed to diverge where the traces reached trigger "don't care" conditions: the implementation can do anything it likes after reaching this trace. To support this we might well want to add a "don't care" annotation onto the availability framework described above, noting that states thus labelled should have no availability notation or outgoing actions.

4.1. Summing up

In general would be a good thing if FDR supported as much as it can of specification notations which people actually wanted to use with it. There seems to be a good argument for finding a way, such as that outlined above, for making the specification of what events a process must have available as clear and straightforward as we can.

Conclusions

This paper has looked at how we might simplify some aspects of using FDR. However we have to face the paradox that the more features are added to make specific uses simpler, the more complex it will get in general.

We have identified a number of technologies, in particular lazy compilation, that should remove some of the need for expertise in CSP/FDR users. Wherever possible the application of these should be made automatic rather than requiring further expertise.

There is a strong argument for creating interfaces to FDR with differing levels of expertise, say the elementary one alluded to earlier which uses lazy compilation[5] by default and makes complex functionality inaccessible, a "normal" one and an "advanced" one.

However, we perhaps need to face the reality that for widespread use in particular communities we may have to deliver customised versions that use the specification and perhaps also implementation languages of the said communities.

References

[1] P. Armstrong, M.H. Goldsmith, G. Lowe, J. Ouaknine, H. Palikareva, A.W. Roscoe, and J. Worrell. Recent developments in FDR. In *Computer Aided Verification* CAV, pages 699–704. Springer LNCS 7358, 2012.

[2] P.J. Broadfoot and A.W. Roscoe, *Embedding agents within the intruder model to detect parallel attacks*, Journal of Computer Security, **12**, 2004.

[3] S.J. Creese and A.W. Roscoe. Verifying an infinite family of inductions simultaneously using data independence and FDR. In *Proceedings of FORTE/PSTV'99*, October 1999.

[4] S.J. Creese and A.W. Roscoe. Data independent induction over stuctured networks. In *Proceedings of PDPTA 2000*, 2000.

[5] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A.W. Roscoe. FDR3: a modern refinement checker for CSP. In *Proceedings of TACAS*. Springer LNCS 8413, 2014.

[6] T. Gibson-Robinson, and A.W. Roscoe. FDR into the Cloud. In *Proceedings of CPA 2014.*

[7] C.A.R. Hoare. *Communicating sequential processes.* Prentice Hall, 1985.

[8] P.J. Hopcroft and G. Broadfoot. Combining the box structure development method and CSP for software development. *ENTCS*, 128(6):127–144, 2005.

[9] E. Kleiner. *A web services security study using Casper and FDR* Oxford University D.Phil thesis, 2008.

[5]The practicality of this will depend on how efficient lazy compilation turns out to be, and whether there is any user overhead when employing it.

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

[11] M. Leuschel, A. Currie and T. Massart. How to make FDR spin LTL model checking of CSP by refinement. *Proceedings of FM 2001*.

[12] G. Lowe. Specification of communicating processes: temporal logic versus refusals-based refinement FAC **20** 2008.

[13] G. Lowe. On CSP refinement tests that run multiple copies of a process ENTCS **250**, 2009.

[14] J. Ouaknine, H. Palikareva, A.W. Roscoe, and J. Worrell. Static livelock analysis in CSP. In *CONCUR 2011–Concurrency Theory*, pages 389–403. Springer, 2011.

[15] N. Moffat, M.H. Goldsmith and A.W. Roscoe. A representative function approach to symmetry exploitation for csp refinement checking. *Formal Methods and Software Engineering*, 2008.

[16] H. Palikareva *Techniques and Tools for the Verification of Concurrent Systems*. Oxford University D.Phil thesis, 2012.

[17] H. Palikareva, J. Ouaknine, and A.W. Roscoe. Sat-solving in CSP trace refinement. *Science of Computer Programming*, 77(10):1178–1197, 2012.

[18] A.W. Roscoe. Model-checking CSP. In *A classical mind, essays in honour of C.A.R. Hoare*. Prentice-Hall 1994.

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

[20] A.W. Roscoe. Proving security protocols with model checkers by data independence techniques. *Proceedings of CSFW 1998*.

[21] A.W. Roscoe. On the expressive power of CSP refinement. FAC **17** (2), 2005.

[22] A.W. Roscoe. *On the expressiveness of CSP*. Manuscript 2009. See `http://www.cs.ox.ac.uk/files/1383/expressive.pdf`

[23] A.W. Roscoe. *Understanding concurrent systems*. Springer, 2010.

[24] A.W. Roscoe and M.H. Goldsmith. The perfect spy for model-checking cryptoprotocols. *Proceedings of DIMACS workshop on the design and formal verification of cryptographic protocols*, 1997.

[25] A.W. Roscoe and P.J. Hopcroft. Slow abstraction through priority. In *Theories of Programming and Formal Methods*. Springer LNCS 8051, 2013.

[26] A.W. Roscoe and Zhenzhong Wu. Verifying statemate statecharts using CSP and FDR. *Formal Methods and Software Engineering*, 2006.