Kalpa Publications in Computing

Volume 20, 2024, Pages 1–16

Proceedings of 2024 Concurrent Processes Architectures and Embedded Systems Hybrid Virtual Conference

The challenges and triumphs of CSP–based formal verification

A.W. Roscoe and Pedro Antonino

University College Oxford Blockchain Research Centre
Oxford, UK
The Blockhouse Technology Limited
Oxford, UK
awroscoe@gmail.com, pedro@tbtl.com

Abstract

I have spent my long career developing the CSP process algebra, with much attention going to make it usable in real-world problems. In this paper I reflect on this work and try to sum up where we are in 2024. I concentrate on some relatively recent applications including the ideas behind the Coco System and how CSP can be used to support decentralised reasoning in the presence of Byzantine behaviour. I think CSP models are the ideal starting point when you want to get to grips with a challenging issue in practical concurrency.

1 Introduction

Concurrent systems seem much harder to understand than sequential ones. However, current advances in processing power and the need for decentralisation in how society is run both depend completely on concurrent execution. Many practical problems are most naturally modelled by parallel, interacting collections of agents. So, we need the tools to help the humans who have to design and use concurrent systems to understand these systems, upon which everyone's life now depends.

Throughout my life, I[1] have been brought problems about concurrent systems that people find hard to solve, and sometimes even understand: the interactions of parts of chips, distributed databases, multiply redundant fault tolerance, the integration of legacy systems, security, groups of vehicles, parallel algorithms, decentralised systems and blockchain and more. I have always sought to build ways of understanding these problems.

Usually, my first thought was to model a problem in CSP [24, 36, 39], often not to try to solve it immediately, but rather to make it clear what a solution would be. CSP was my language of choice, and frequently I made use of FDR, both because I am an expert on these but also because it is often possible to capture either the whole or part of a problem well that way. Nothing annoys me more than a problem I cannot solve. There is a well-known saying that to a hammer, everything is a nail, and I have often observed that computer scientists are always much keener on using their own piece of theory to model things than anything else. That can certainly be levelled reasonably at the combination of me and CSP.

[1] This paper is written from the perspective of Bill Roscoe

I explored in [40] 10 years ago, how to make these theories more widely accessible. I think one of the best ways of doing this is to give interesting examples. I have already published large collections of examples in coding for FDR with my two books on CSP. Here, I will cover some examples of how CSP theories have been able to successfully solve hugely varying problems in areas that I doubted were tractable.

The main thing I want to demonstrate is just how flexible and powerful CSP is, and to encourage many more people to use it to model concurrent systems and solve seemingly impossible problems. While everything in [40] remains true, I want to convince you that you, too, can use it to solve really interesting problems, just like me and the people who work with me. In fact, much of this paper is devoted to great ideas by other people.

In this paper, I concentrate on things not covered well in my books and therefore relatively recent. Aside from the first example, they are about scaling. Thus, we do not cover compression, data independence, data independent induction or systematic abstraction. Other topics not covered here include partial order reduction [21] and symmetry reduction [22]. All of these are other techniques to achieve scaling.

Though FDR does impose its own restrictions through finiteness, there are an incredibly large number of ways in which one can approach problems in CSP, including levels of abstraction. Because FDR is based on refinement, it naturally supports hierarchical and compositional development, strategies that we will use in this paper.

Aside from what is discussed in my two existing books on CSP [35, 39], I recommend the following papers as background reading on modelling a wide variety of systems [14, 48, 49, 18, 17, 37, 47, 38, 27, 2]. A survey of CSP's practical uses can be found in [16] and the main papers describing CSP model checking are [35, 19, 20].

In this paper, I will not cover Timed CSP [33], though I have thought for some time that FDR's implementation of checking using Ouaknine's work on digitisation [31, 14] over this needs more attention. In this paper I will use the machine-readable CSP-M dialect of CSP that is used by FDR.

Since this paper is aimed mainly at CSP cognoscenti, I omit the usual summary of CSP, but descriptions of the language can be found in many of the books and papers we cite.

This paper surveys some recent work from other papers and commerce as well as introducing two new topics: how to model two-player games to decide who has the winning strategy, and the connection between token invariants and linear algebra that emerges from some of the authors' previous joint work.

2 Two player games

Anyone who has followed my work using FDR will know how keen I am on modelling combinatorial systems. Sometimes these have been combinatorics arising from practical concurrent systems such as the intruder in security models [45], but I also like to use puzzles: peg solitaire, card solitaire [41], sudoku and variants thereof, and puzzles I have bought in shops, frequently made of wood. Being able to solve these illustrates very well how the combination of CSP and FDR can be much better at games than tasks that many people have actually spent much time on.

However, if you think about these examples, they have one thing in common: they are one-player games. The game is entirely set up at the start and what you have to do is find a sequence of moves that solves it. You have no opposition other than the puzzle designer. It is worth noting that except in cases of puzzles where FDR's compressions are effective, CSP does not offer any particular efficiency advantage over other efficient codings of a state space to search through. Rather it confers the advantage of having a ready-built search on top of a concise and elegant way of describing the puzzles.

Compressions are often effective on puzzles with some underlying algebraic structure, or which are close to being one-dimensional. A nice example of the latter is a challenge I was given by Donald Knuth

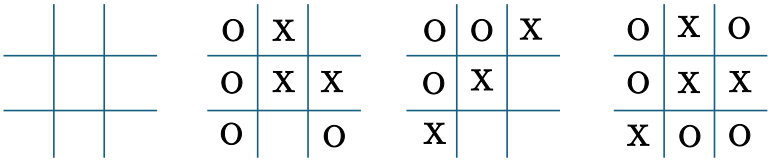

Figure 1: Noughts and crosses: the empty board, a configuration where **O** wins, a configuration where **X** wins, and the configuration of a draw, respectively.

to enumerate the knight's tours in a k by n chessboard, where k is a small fixed natural number and n varies. It turns out that as n increases the compressed state space (hiding moves involving only squares some way from the end where n is increasing) does not change if you add 2 to n for sufficiently large n. This makes it possible to calculate a recurrence in n. My solutions to this are available with the archive for this paper which covers the approaches discussed in this section and the next.[2]

Occasionally, people asked me if CSP and FDR can discover who has a winning strategy in a two-player game such as chess, go or noughts and crosses (tic-tac-toe in US English). So instead of finding a path through a large graph of states, you need to think about every response that the opponent might make to yours. The game tree thus has two separate flavours of branching, that need to be treated differently.

The approach of simply searching for a solution, namely, a single state of the puzzle, that works for the one-player puzzles is obviously not going to work here. I think that people asking me the question about two-player games were interested mainly in whether it can be done elegantly using the process algebra constructs of CSP, since it is obvious that CSP-M is Turing complete. If one coded the state space of a two-player game with moves labelled by A and B as transitions, you could build a process that would run in parallel with it, building up a labelling of the game tree: each reachable trace would become labelled with the best-guaranteed outcome for the agent whose move it is. This testing process could, to make it more practical, be split in parallel like the intruder in cryptographic protocol analysis and implement the α/β pruning algorithm. But I think there is no good reason to do it like that as opposed to non-CSP approaches.

I did find a reasonably elegant approach in process algebra terms that I will illustrate here with noughts (**O**) and crosses (**X**). In this game we have a 3×3 grid, so 9 places in all. **O** goes first, and then they play in turn. If either player gets a row of 3 (vertical, horizontal or diagonal) they win. The approach computes minimax over the game tree, identifying the states (board configurations) that are a win for **O**, a win for **X** or a draw. The rest are all active states or unreachable.

Our approach is to identify process values for `O-win < Draw < X-win`, and a pair of CSP-definable operators that map any group of these respectively to the `min` and `max` of the group in this arbitrary order on the three values. Inserting these operators for respectively **O** and **X** choices, with already finished values at the leaves does the trick. There are many potential choices, but here are three. In the first the result shows up in the number of `a`s in a trace.

```
X-win = a -> a -> Stop
Draw = a -> Stop
O-win = Stop
P min Q = P [|{a}|] Q
P max Q = P |~| Q
```

In the second the result shows up in the initially refused events: they all have the same traces.

[2]The archive is available at https://github.com/pedrotbtl/copa24

```
X-win = a -> STOP [] b -> STOP
Draw = b -> STOP [> a -> STOP
O-Win = X-Win [> STOP
P min Q = P |~| Q
P max Q = P [] Q
```

In the third the result shows up in the initially possible events.

```
X-win = a -> STOP [] b -> STOP
Draw = a -> STOP
O-win = STOP
P min Q = P [|{a,b}|] Q
P max Q = (P ||| Q) [|{a,b}|] X-win
```

The first and last of these work over the traces model, and the middle one requires stable failures. In all three cases the binary `min` and `max` operators extend naturally to indexing over all the branches from a node of the game tree (i.e. the multiple moves available to **O** or **X**). The final parallel composition in the third definition of `max` has the effect of permitting traces of length only up to one, and is not necessary if the same restriction is placed at the root level of the tree.

The core of my CSP script using the first version of this is below. The `sbdia` compression eliminates the τ actions that `|~|` introduces.

```
Coords = {(i,j) | i <- {0..2}, j <- {0..2}}

HLines = {{(i,j) | i <- {0..2}} | j <- {0..2}}
VLines = {{(i,j) | j <- {0..2}} | i <- {0..2}}
Diags = {{(0,0),(1,1),(2,2)},{(0,2),(1,1),(2,0)}}
Lines = Union({Diags,HLines,VLines})

transparent explicate,sbisim,diamond

sbdia(P) = sbisim(diamond(P))

Win(S) = ({l | l <- Lines, l <=S}!={})

Draw(xs,os) = not Win(xs) and not Win(os) and card(xs)==5

P(xs,os) = if Win(xs) then a -> a -> STOP
           else if Win(os) then STOP
           else if Draw(xs,os) then a -> STOP
           else if card(xs) == card(os) then
     sbdia(|~| x:diff(Coords,union(xs,os)) @ P(union(xs,{x}),os))
           else ([|{a}|] o:diff(Coords,union(xs,os)) @
                                  P(xs,union({o},os)))

assert STOP [T= P({},{})      -- fails if not a win for Os
assert a -> STOP [T= P({},{}) -- fails if a win for Xs
```

The checks confirm that the overall game value is `Draw`, meaning that neither player can force a win.

The CSP archive contains a second example based on the well-known game Connect 4, with parameters for the dimensions of the grid and the length of line required to win. The version of FDR4 on my

laptop will not approach the standard game, however, as the structure of the coding drives the FDR CSP compiler in unusual directions.

I do not think that this approach is nice enough to argue that it should be the central approach to analysing two-player games. Maybe someone else can improve on it.

In this section we have shown what is probably just a fascinating algebraic curiosity. In the rest of the paper, I will illustrate three very successful approaches to the classic problem of scalability.

3 Decentralised systems with stochastic malevolence

I have spent much of the last seven years thinking about building and securing blockchains and other decentralised systems with an assumed proportion of malevolent actors whose identities are secret to the good agents. Generally speaking, the latter are assumed to be Byzantine, and could therefore be modelled as the CSP value `Chaos` – one might initially expect. If this were true these systems would resemble analyses of fault tolerance that have been a familiar part of the CSP world since its earliest days.

In fact, these systems always rely on cryptographic constructs to protect themselves against insider malevolent attacks as well as external ones. This means that, as in the now-familiar analyses of intruders in cryptographic protocols [48], malevolent agents have to run this crypto and take advantage of what they can get the good ones to say, and what they can learn and build using the assumed properties of the cryptography. That means that we need to replace `Chaos` by processes based on the Intruder from protocol models. There is nothing alarming about this, as this is well understood.

Happy that I understood how to deal with malevolence, I set about building a simple CSP model of a blockchain. I had expected to use a lot of symmetry reduction because the actual locations where blocks would be stored and perhaps the names (i.e. hashes) of blocks would be unimportant except for their distinctness. I was thinking of a blockchain as a form of glorified linked list, and certainly symmetry reduction gives huge advantages with CSP representations of these.

Unfortunately, it quickly became apparent that I would never be able to include a remotely realistic number of block values, agents, keys, etc, because of state explosion, at least if I wanted to be able to compile an individual agent or run a meaningful check on FDR. This was despite factoring processes into parallel as much as was reasonable and devising a new model of hashing that was a nice abstraction of the real implementation one would expect, and storing complex block values by the resulting efficient symbolic hashes. Whereas, in the standard symbolic cryptographic model used in [35] and similar papers, the hash of `X` is simply modelled as `hash.X`, in my new model we give each agent the right to create (in runs we consider) some small number `H` of hashes, and the hashes are then `hash.k` as `k` varies between 1 and `H`. This enormously reduces the alphabet size and parameter spaces of processes storing hashed. Hashing is carried out by a process that maps input values `X` to one of these indexes. Once the hashing limit is passed, the check is at an end in the style of bounded model checking. As far as possible, we store and communicate these compact hashes rather than the pre-images.

This method of storing blocks already factored out most of the symmetries I was expecting to tackle with the features of FDR4 described in [22], so I did not need to use this.

Symbolically speaking, what we need to do is ensure that every time a value is hashed it gives the same answer and as with the `hash.X` model, that collision-freedom and pre-image resistance are true.

The blockchain model I created with this version of hashing is available with the archive for this paper, but I quickly lost confidence in the applicability of CSP model checking to blockchain at this level of abstraction because to do anything interesting seemed to require an intractable model. Not only intractable for checking, but also for the FDR compiler.

For a long time after this experiment, I did not use CSP in modelling blockchains, but then I realised there was a level of abstraction at which it is very helpful and a particular aspect where it seemed

essential. In this, logical processes are *consensus machines*, which are defined groups of agents given a program to form a consensus on. These are designed so that in the context, which mainly means the assumed worst case of malevolence, we design a degree of agreement between the agents concerned so that if it is reached, there can be no ambiguity about the next action. The actions of the consensus machine coincide with the creation of certificates proving such agreements exist and that an agent is aware of it. All of this is happening in a decentralised world so just because a certificate has been formed does not mean that everyone has, or, sometimes, ever will see this. But I find this a very nice way of understanding and abstracting consensus. This group of agents becomes a state machine which we treat as sequential, though with the nondeterminism arising from its true form carefully modelled.

If the whole system is managed as a single consensus machine, for example implementing classic Byzantine agreement which requires a 2/3 majority on the assumption that less than 1/3 of agents are bad, there is no need for CSP. However, if multiple consensus machines operate in parallel it is very useful.

They might operate in parallel because they control different aspects of the world the decentralised system lives in, or they may perform complementary functions. Conceivably they could be two separate blockchains. In [42, 43], we introduced them as part of a novel consensus protocol that we expect to outperform its competitors and in which decisions about questions are made by one of a number of consensus machines. This section is a summary of the second of those papers. It is easiest to explain with two, which we call primary P and secondary S. The two together have to make a decision, for example on what the next block of a blockchain will be. P is assumed to be safe and efficient if it does decide, but decisions may not get communicated or not happen at all. In CSP terms it may deadlock, but the deadlock is not reliably detectable.

Stochastic analysis of the population of agents, again described in [43], should lead to a comparatively small population to make up P provided that high agreement thresholds are chosen, making it possible for a few good agents to block a bad decision. P does not need enough agents to force a good decision. Under stochastic reasoning, the size of P can generally be independent of the overall number of agents.

S exists to take over when P might have deadlocked: it is more reliable in that a decision is guaranteed, but it may well be a much more laborious process and there may be many more agents in S as opposed to P. There now have to be enough good agents to force agreement at a threshold that blocks bad decisions.

We want to give P a chance, and if that chance is not taken ideally disqualify P and set S going. Unfortunately, that is not possible in a decentralised environment. For example, a clear decision may be made by P in such a way that the certificate for it is known only to a bad agent. That bad agent can sit on the certificate and release it just as S is getting going. Mutual exclusion is inadequate as a solution to this: if P has the mutex token and deadlocks, the whole system is deadlocked. We want to combine P and S into a usually efficient safe and live decision-making mechanism. The fundamental interlock we need is that P and S do not make *different* decisions.

I therefore developed the idea of the *handover* protocol. Since there is no way, thanks to decentralisation that we can preclude both P and S making a decision, we instead ensure that if they both make one, the decision is the same. To do this we need to characterise how consensus machines can interact.

Given the context, the most natural familiar model is to treat P and S as sequential state machines, but we have to do so in ways consistent with the actual decentralised implementation of them and their communication. The most appropriate communication model seems to be to read and write shared "signal" variables, though this needs to be handled with great care as both the consensus machines and the variables (presumably) are formed of decentralised groups containing malevolent agents.

Nevertheless we showed in [43] that even under very weak assumptions about decentralised communication, it is possible to implement this communication with stochastically chosen sets of agents

that are again of constant size.

The approach we take to this is to form sequential CSP models of how these two sorts of entities behave in general. We can then verify sample parallel CSP models of instances of them against this sequential model or analyse the correspondence by other means. The stage is then set for building a combination of these sequential models to verify with FDR.

This is explored in the two papers cited above that Pedro Antonino, Jonathan Lawrence and I have written on this. Both *read*s and *write*s of the signals are split into *begin* and *end* events. These represent different transitions/certificates in a consensus machine. The *begin* phase shows that the consensus machine wants to take the action; the *end* one when there is evidence that it has reached sufficient components of the variable in the case of a write, or the read value is settled and immutable.

The handover protocol, presented in this style is described here. `PM` is the primary mechanism. The predecision on a value `V1` starts a write; it then reads if the secondary mechanism `SM` has started, and if not completes its decision.

```
PM = predecide.V1 -> endwrite1 -> startreadS -> readS?state ->
      if state == Started then STOP
      else decide.V1 -> endwrite2 -> STOP

PMren = PM[[predecide.v <- startwriteP.Predec.v,
            decide.v <- startwriteP.Decided.v | v <- Decisions]]
```

The secondary mechanism checks to see if `PM` has decided or pre-decided, and if so copies that value.

```
SM = timeout -> startreadP ->
     (readP?_:{NullP,Predec.v | v <- Decisions} ->
     startwriteS.Started -> endwriteS -> startreadP ->
     ((readP.NullP -> decideS.V2 -> STOP) []
      ([] v:Decisions @ (readP.Predec.v -> decideS.v -> STOP []
           readP.Decided.v -> decideS.v -> STOP))))
```

The primary machine is modelled as always being able to deadlock, and the two interact via their signals.

```
CMS = (PMren[|Events|]CHAOS(Events)) ||| SM

Signals = PS(NullP)||| SS(NullS)

System = CMS [|{|startwriteP,startwriteS, endwriteP,
      endwriteS, startreadP, startreadS, readP,readS|}|] Signals
```

The model of a variable allows it to hold two values as live — namely the potential result of a read — namely the most recent write to complete, and the most recent to start from the single writer. The value that results from a read is any that was live between the start and end of that read, subject to sequential consistency: an earlier read must return a value written no later than a later read. The following describes this behaviour for the single writer, single reader case, which is what is needed for the handover protocol.

```
PS(x) = startreadP -> PSr(x)
        [] startwriteP?z -> PSw(x,z)
```

```
PSr(x) = readfixP.x -> PSf(x,x)
         [] startwriteP?z -> PSrw(x,z)

PSw(x,z) = endwriteP -> PS(z) [] startreadP -> PSrw(x,z)

PSrw(x,z) = endwriteP -> PSr(z) []
            (readfixP.x -> PSwf(x,z,x)
              |~| readfixP.z -> PSwf(x,z,z))

PSf(x,z) = readP.z -> PS(x)
           [] startwriteP?y -> PSwf(x,y,z)

PSwf(x,y,z) = readP.z -> PSw(x,y) []
              endwriteP -> PSf(y,z)
```

Overall, this both gives us the core of a radically new form of consensus and a foundation for building complex reliable systems containing multiple consensus machines operating side by side, operating in the presence of malevolence. An example of this can be found in [43].

I would have been far from confident in the correctness of this consensus approach without FDR, particularly the correctness of the handover protocol, and would have had difficulty explaining it to others. By using this approach of modular development with hierarchical verification we have been able to verify decentralised systems that could not have been practically handled by modelling all the component agents as a network on FDR. This is a classic approach to scalability.

4 Local model checking

In [28, 29], Jeremy Martin introduced the concept of the State Dependency Digraph, whose size is the sum of the state spaces of the components of a parallel network. This is far smaller than the (order of the) product of these numbers that FDR would often explore. He introduced this method of analysing automatically for a deadlock that generalised a variety of analytic rules that I and others had created [44]. Martin's approach is incomplete: it may fail to prove deadlock freedom even when it is true. It was crucially based on proving the absence of cycles of ungranted requests.

When, many years later, Pedro Antonino began his doctorate with me, our understanding of what is tractable and what is intractable had moved on substantially. From the moment we created it we knew that FDR was built around generally tractable instances of a problem — deciding refinement in CSP models — that is intractable in the worst cases. Its complexity of being PSPACE-hard in the state space of the specification process [26] is even worse than the classical NP-hard. This last point mattered little to us when we first created FDR in 1991, but it may have been significant in the following.

The development of SAT checkers and SMT solvers transformed our perception of what it means to be intractable. Suddenly many instances of NP-complete problems, and in general verification problems translated to the logical languages of Satisfaction and Satisfaction Modulo Theories became practical in a way that we had not previously suspected. However the model-checking versions of this fell short when it came to proving complete results, so bounded model checking [15] became popular, in which systems were only allowed to go wrong for a fixed number of steps.

There were several attempts to create a SAT-checker back end for FDR, notably [32], and though there were some examples on which this worked well, they were not frequent enough.

This work and the idea behind the SDD come together in Pedro's doctoral project. In essence he takes the SDD generalised, so we have a relationship of mutual reachability between states of adjacent

processes. In the base level version of his theory [8, 10], the specification is any predicate on the tuples of states where each component is assigned one its own states. So we can, for example, specify deadlock freedom, namely there is some component that is not blocked. That was the main example that Pedro considered. We can then ask a SAT checker whether there is a deadlocked state where every adjacent pair of states has mutually reachable states. Thanks to the power of SAT checkers this gives a much more precise, though still over-approximating, view of deadlock than the SDD. Other global integrity properties, such as no variable being read uninitialised (where variables are component processes) will work well when they can be expressed as unreachability of states of the system.

In [13, 9, 11, 12], we investigated a variety of ways in which further restrictions can be added into the SAT checks, making them more precise. These include a generalised version of token ring style properties as invariants: it is even possible to discover token invariants using SAT and SMT. In these cases we run such a solver once to discover an invariant, and then run it again to test the specification with the new invariant in place. We even discovered that the most general token invariant on a network can be calculated as the kernel of a linear transformation derived from the component states and their synchronisations. Here we introduce this interesting research topic we discovered a few years ago and invite others to take it up.

In [9, 12], we identified two distinct models of token invariant. In the *conservative* model, the number of tokens (like tokens in a typical token ring) remains constant, while in the second, the *existential* model, it can vary in such a way that they cannot disappear completely. We used SAT checkers to discover both sorts, but did not publish the result that the most general conservative token invariant can be calculated by linear algebra.

We illustrate our approach with the concurrent system `SYS`: a alphabetised parallel composition of `A`, `B`, and `C` (as defined below) with alphabets {`a`, `b`}, {`b`, `c`}, and {`a`, `c`}, respectively.

```
A = a -> b -> A

B = b -> c -> B

C = a -> c -> C
```

We work over a vector space that could either be over the reals or some sufficiently large prime field. This space has dimension N, the sum of the number of (individual) states for components in the network under consideration, and the canonical basis has a one in the component corresponding to a given such component state and 0 elsewhere. Figure 2 details the LTSs for the 3 components of `SYS`. For instance, for this system, this vector space has dimension $N = 6$.

A token assignment t will map each component state to the number of tokens it holds; it can be seen as a vector of size N. If every component state only has 0 or 1 token, the coefficients in this vector will be 0 or 1. A *conservative token invariant* is such a vector where additionally each transition of the overall system preserves the number of tokens in the system. For instance, if a transition involving two components of the system takes them from component states (x_1, y_1) to (x_2, y_2), the sum of tokens in a conservative invariant t must ensure that $t(x_1) + t(y_1) = t(x_2) + t(y_2)$; where $t(x)$ represents the tokens held at x according to t.

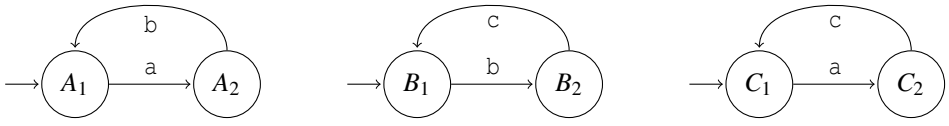

Figure 2: LTSs of components `A`, `B`, and `C`, respectively.

We can concisely define this invariant using a linear system. Considering that a system action (i.e. transition) *act* induces a row vector $w(act)$ of size N with 1s on the components that are at the source and -1s at the targets (and 0 for the remaining component states), the invariant requires that for all actions *act*, the inner product $w(act) \cdot t$ is 0. Given an *action matrix* $\boldsymbol{A}$ where each row corresponds to row vector $w(act)$ for a possible action of the overall system, for t to be a conservative invariant, we now have the simple requirement that $\boldsymbol{A}t = 0$, or in other words t is in the *kernel* of the action matrix.

We use our example system `SYS` to illustrate this matrix's construction. First, one can use pairwise analysis to uncover the transitions of the system. If a system is triple disjoint (like `SYS`), namely, its synchronisations involve at most 2 components, one can use (unary and) pairwise analysis to efficiently uncover the actions of the system. By analysing `SYS`, we identify the following system transitions. We simplify the presentation of our transitions by considering only the components that take part in it. For instance, the first transition indicates that `A` and `B` transition from A_2 and B_1 to states A_1 and B_2, respectively, while component `C` is unaffected by this transition and remains in the same state.

- $(A_2, B_1) \rightarrow (A_1, B_2)$
- $(A_1, C_1) \rightarrow (A_2, C_2)$
- $(B_2, C_2) \rightarrow (B_1, C_1)$

These transitions give rise to the following $\boldsymbol{A}t = 0$ system:

$$\begin{bmatrix} -1 & 1 & 1 & -1 & 0 & 0 \\ 1 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} t_{A1} \\ t_{A2} \\ t_{B1} \\ t_{B2} \\ t_{C1} \\ t_{C2} \end{bmatrix} = 0 \tag{1}$$

The most general invariant I for this network is the kernel of $\boldsymbol{A}$, itself a vector space, which can, of course, be calculated by traditional polynomial algorithms such as Gaussian elimination. For our example, the following t_1, t_2, t_3 and t_4 vectors form a basis for the solutions to the homogeneous system above (i.e. kernel of the A above). These vectors are all valid conservative token invariants for `SYS`. Note that the first three are degenerate and simply say that the three respective component processes are always in exactly one state. The fourth is a non-degenerate invariant.

$$t_1 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, t_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, t_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, t_4 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{2}$$

Let us illustrate now how these token structures can be used for reachability analysis and focus specifically on deadlock checking for this example. Note that a system invariant for a particular conservative token structure is that *(ri) the number of tokens in the initial (system) state is equal to the number of tokens in any reachable (system) state*. Hence, we can use a conservative token structure to calculate the number of tokens in the initial state and use this number to check whether a state is reachable or not. This technique over-approximates the reachable state space of `SYS`: if a system state does not have the same number of tokens as the initial state, then that state is not reachable — given (ri). Thus, the token structures on the basis above can be used to carry out this type of reachability analysis for `SYS`. In the

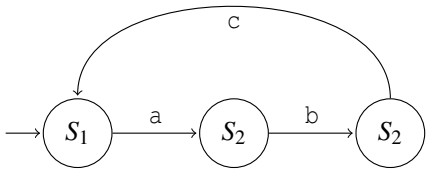

Figure 3: LTS of SYS.

following, we present the reachable states of SYS; Figure 3 depicts the LTS of SYS. One can calculate the number of tokens, for a system state and given a particular token structure, by adding the tokens held at component states in that system state. For example, the number of tokens at S_1 for t_4 is given by $t_4(S_1) = t_4(A_1) + t_4(B_1) + t_4(C_1) = 2$.

- $S_1 = (A_1, B_1, C_1)$; where $t_1(S_1) = t_2(S_1) = t_3(S_1) = 1$ and $t_4(S_1) = 2$
- $S_2 = (A_2, B_1, C_2)$; where $t_1(S_2) = t_2(S_2) = t_3(S_2) = 1$ and $t_4(S_2) = 2$
- $S_3 = (A_1, B_2, C_2)$; where $t_1(S_3) = t_2(S_3) = t_3(S_3) = 1$ and $t_4(S_3) = 2$

The first three token structures t_1, t_2, and t_3 are degenerate, namely, they trivially over-approximate reachability by showing that any system state is potentially reachable. The t_4 structure, however, provides an approximation that is tight enough to show deadlock freedom for SYS. Both deadlocking states of SYS (presented below) have a token count for t_4 that is different from the reachable token count of 2, demonstrating their unreachability and ensuring that this system is deadlock free.

- $D_1 = (A_1, B_1, C_2)$; where $t_4(D_1) = 3$
- $D_2 = (A_2, B_2, C_1)$; where $t_4(D_1) = 0$

We invite anyone interested to research properties of $\boldsymbol{A}$ and I. These may well vary with the type of system considered, for example, how many components can participate in each action. The choice of field to perform the calculations over can also affect the outcome.

As can be seen from the experiments documented in the papers cited here and Pedro's thesis [7], this idea of using the sum of component state spaces as the basis for SAT and SMT solving produces a very exciting advance in scalability in classes of network analysis. I think it should be researched further. I like to call it *local model checking*.

In some sense it is intermediate between static analysis and accurate state exploration. It is interesting to ask whether any other extracts of behaviour that are locally computable can be used in a similar way to the relations exploited here. We are looking for interesting over-approximations to reachability that can be coded in a way suitable for SAT checkers.

5 Compositional modelling in the Coco System

The holy grail for verification researchers should be large-scale successful use of their work by software engineers. Since the world at large did not take sufficient notice of my own team's work on the transputer [23, 34], and particularly the T800 FPU [30], it took the Intel FDiv bug [5] to really drive formal verification in the microprocessor industry, but that has now been successfully embedded for many years.

Though there have been many successful uses of formal methods in software by experts, and take-up in industry of static analysis and assertions, and to some degree formally based test case generation, I am not aware of many large-scale adoptions of formal methods that create significant software that is correct by design.

If, in 1984, you had forecast to me and the other formal methods experts who were active 40 years ago, this fact about 2024 would have been surprising and disappointing. I put this down to a number of factors:

1. The state explosion problem has seemed to limit scalability and has often been a real obstacle.

2. Most verification researchers seem more fascinated with exotic theoretical issues than reducing their research to engineering practice. This is both because they are temperamentally inclined this way and because the academic career structure at universities like mine points them in that direction. Bureaucratic obstacles are often put in their way when they veer towards collaboration and spin-outs, so theory is the easiest path.

3. Lack of enthusiasm from software engineers who already have well-developed approaches. They do not want to learn new and very different techniques when they are happy with what they have. It is typical of people to be scared of new things. Often they could reasonably complain that the researchers devising the methods were not prepared to meet them halfway.

4. Problems for engineers in creating formal specifications: a new way of thinking.

Perhaps the closest to overcoming these simultaneously has taken shape over the past 20 years in the practical research and engineering that has now resulted in the Coco System, the language Coco and the company Cocotec `cocotec.io`. I emphasise that my own direct contribution to the research and engineering work described here has been minor compared to the other three people I refer to below.

Guy Broadfoot, was a very experienced software engineer who, for his MSc project on Oxford's part-time programme, spotted connections between CSP and existing software engineering techniques such as the Box Structure Development Method [25]. Fortunately for him, his daughter Philippa had recently done a doctorate with me on security that involved some very detailed CSP programming. They were thus able to realise his vision in a product ASD (Automated Software Design) that used FDR as a back end.

This proved properties of parallel combinations of state machines specified in a custom notation in a fashion in a largely client-server structure derived from previous software engineering methods. But it was able to prove many classes of errors absent by model checking each component state machine in the context of the specifications of its neighbours and its connections to them. This was a form of compositional development that worked wonderfully with respect to scaling: there were no real limits within the application domains of a number of large manufacturers [1] provided the programs designed complied with some strict rules. These related to the style of system: control rather than numerical, and limited designs to acyclic structures based on client-server relationships.

The ASD model drove both the development of forms of abstraction in CSP, for example carefully distinguishing between "good" and "bad" divergence [46], and influenced later versions of FDR2.

ASD still suffered from some of the issues enumerated above, not least its input notation, its limitations on architectures and specification, and because FDR2 was getting dated.

Development of this technology stalled thanks to commercial and personnel issues as Guy and Philippa ceased to be involved with it. In the meantime, Tom Gibson Robinson was the main person re-creating FDR as FDR3 [19, 20] which made it far more capable than FDR2. This taught us a great deal about refinement checking in the era of highly parallel computers and how well most of it parallelises.

After that we (Philippa, Tom, me and Guy) put together some research proposals incorporating new thinking for an entirely new approach to the same problem. The original concept [6] was that Coco would chiefly be an intermediate language for translating industry standards such as dialects of UML to, and that the Coco program would be verified using the latest FDR and translated to computable source code for the model-based system under development. But over the course of the project Coco was so liked by partners that they tended to use it directly. Tom therefore developed it into the excellent programming language it now is. Also it became clear that the full expressive generality of CSP was not needed in the verification phase, and that coping with that generality put obstacles in the way of dealing with other features of Coco that were desirable, chiefly relating to handling of variables. So in the end, the Coco Platform (Cocotec's principal product) does not use classic CSP and FDR, but a customised process algebra (a CSP-like language in the sense of [39]) based on CSP theory, and an optimised model checker for it.

For details of the Coco language and the Coco Platform, see the website `cocotec.io`.

It has been adopted by a number of customers, most notably on a large scale by ASML in creating the software that runs their photolithography machines [3, 4].

Below I assess the Coco Platform against the four obstacles listed above.

1. The compositional development approach solves the scalability issue more completely than ASD, all the more so with the improvements to model checking discussed above.

2. I sometimes describe myself as a maverick amongst verification researchers for concentrating so much on industrial applications. Working at Oxford University gave us the opportunity to gain funding from EPSRC and Innovate UK to develop Coco. It was surprisingly difficult to extricate the results satisfactorily into the company Cocotec, but fortunately that is now behind us.

3. The Coco language is the most visible step forward in this exercise. It seems to be readily adopted by, and be popular with engineers.

4. It handles a much wider range of specifications, which encompasses lack-of-starvation properties.

Being aimed at industrial users, further developments are now aimed at what customers want to achieve.

In order to handle more general specifications that might stretch the Coco Platform's compositional model of verification, I can imagine adding capabilities based on either CSP compressions or some of the ideas in the previous section. The problem of making the art of general specification easy for traditional engineers remains somewhat open.

It must be emphasised that a commercially targeted tool like the Coco Platform is fundamentally different from one whose motivation is primarily academic. FDR has always been closer to the latter than the former.

6 Conclusions

I hope to have provided yet more evidence that CSP and its models are an excellent language for describing and reasoning about a huge variety of concurrent systems and solving significant problems that in most cases seemed at first to be insoluble. I am particularly cheered by the demonstration of how it can be used in decentralised systems, and particularly by Cocotec's impact.

One of the great challenges of CSP verification that remains is proving the correctness of classes of arbitrary-size parallel systems indexed over finite but unbounded types such as general integers or lists. Some such results can be proved using data independence and data independent induction. However

there is still much to be done there. I like to quote two examples from my own work as challenges to the automated verification community: one is to prove the correctness, for all N, the corrected Bully algorithm description from Chapter 14 of [39]. The second, and probably harder, is to prove the distributed database consistency protocol from Chapter 15 of [36]. (This could either be with an order on updates derived from the authors in some fixed way, or more arbitrary.) A tool that can automatically prove either or both of these will impress me enormously.

Acknowledgements

We are grateful to all who have worked with us on CSP and FDR, but the work reported in this paper had significant contributions from, and in some parts was principally due to, Guy and Philippa Broadfoot, Tom Gibson Robinson and Jonathan Lawrence. It would not have been written without Lindsay Quarrie and Jeremy Martin, the latter of whom will find that he inspired some of the work reported here. The work reported here was funded by EPSRC, Innovate UK, and a Brazilian Government Scholarship.

References

[1] Automated software design and verification. https://impact.ref.ac.uk/casestudies/CaseStudy.aspx?Id=4907. Accessed: 02/07/2024.

[2] Automated verification and validation for defence, aerospace and automotive embedded software. https://www.cs.ox.ac.uk/innovation/research-impact/case-D-RisQ.html. Accessed: 02/07/2024.

[3] Bits & chips: 12 October 2023. https://events.bits-chips.nl/bitschips-event-2023-old/. Accessed: 02/07/2024.

[4] By computers, for computers: Improving scanner metrology software with generated code. https://www.linkedin.com/pulse/computers-improving-scanner-metrology-software-code-lewis. Accessed: 02/07/2024.

[5] Pentium FDIV bug. https://en.m.wikipedia.org/wiki/Pentium_FDIV_bug. Accessed: 02/07/2024.

[6] Reducing cost of software: A scalable model-based verification framework. https://gtr.ukri.org/projects?ref=EP%2FN022777%2F1. Accessed: 02/07/2024.

[7] Pedro Antonino. *Verifying concurrent systems by approximations*. DPhil thesis, University of Oxford, 2018. Available at: https://ora.ox.ac.uk/objects/uuid:f75c782c-a168-49b3-bfed-e2715f027157.

[8] Pedro Antonino, Thomas Gibson-Robinson, and A. W. Roscoe. Efficient deadlock-freedom checking using local analysis and SAT solving. In *IFM*, number 9681 in LNCS, pages 345–360. Springer, 2016.

[9] Pedro Antonino, Thomas Gibson-Robinson, and A. W. Roscoe. The automatic detection of token structures and invariants using SAT checking. In *TACAS*, number 10206 in LNCS, pages 249–265. Springer, 2017.

[10] Pedro Antonino, Thomas Gibson-Robinson, and A. W. Roscoe. Efficient verification of concurrent systems using local-analysis-based approximations and SAT solving. *Formal Asp. Comput.*, 31(3):375–409, 2019.

[11] Pedro Antonino, Thomas Gibson-Robinson, and A. W. Roscoe. Efficient verification of concurrent systems using synchronisation analysis and SAT/SMT solving. *ACM Trans. Softw. Eng. Methodol.*, 28(3):18:1–18:43, 2019.

[12] Pedro Antonino, Thomas Gibson-Robinson, and A. W. Roscoe. Approximate verification of concurrent systems using token structures and invariants. *Int. J. Softw. Tools Technol. Transf.*, 24(4):613–633, 2022.

[13] Pedro Antonino, Thomas Gibson-Robinson, and A.W. Roscoe. Tighter reachability criteria for deadlock freedom analysis. In *FM*, number 9995 in LNCS. Springer, 2016.

[14] Philip Armstrong, Gavin Lowe, Joël Ouaknine, and A. W. Roscoe. Model checking timed CSP. *In Proceedings of HOWARD (Festschrift for Howard Barringer)*, 2012.

[15] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[16] Stephen D Brookes and A. W. Roscoe. CSP: A practical process algebra. In *Theories of Programming: The Life and Works of Tony Hoare*, pages 187–222. 2021.

[17] Sadie Creese and A. W. Roscoe. Formal verification of arbitrary network topologies. 1999.

[18] SJ Creese and A. W. Roscoe. Verifying an infinite family of inductions simultaneously using data independence and FDR. In *International Conference on Protocol Specification, Testing and Verification*, pages 437–452. Springer, 1999.

[19] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3—a modern refinement checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*, pages 187–201. Springer, 2014.

[20] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3: a parallel refinement checker for CSP. *International Journal on Software Tools for Technology Transfer*, 18:149–167, 2016.

[21] Thomas Gibson-Robinson, Henri Hansen, A. W. Roscoe, and Xu Wang. Practical partial order reduction for CSP. In *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings 7*, pages 188–203. Springer, 2015.

[22] Thomas Gibson-Robinson and Gavin Lowe. Symmetry reduction in CSP model checking. *International Journal on Software Tools for Technology Transfer*, 21:567–605, 2019.

[23] MH Goldsmith and A. W. Roscoe. Transformation of occam programs. In *1988 International Specialist Seminar on the Design and Application of Parallel Digital Processors*, pages 180–188. IET, 1988.

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

[25] Philippa J Hopcroft and Guy H Broadfoot. Combining the box structure development method and CSP for software development. *Electronic Notes in Theoretical Computer Science*, 128(6):127–144, 2005.

[26] Paris C Kanellakis and Scott A Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 228–240, 1983.

[27] Gavin Lowe. On CSP refinement tests that run multiple copies of a process. *Electronic Notes in Theoretical Computer Science*, 250(1):153–170, 2009.

[28] Jeremy Malcolm Randolph Martin. *The design and construction of deadlock-free concurrent systems*. PhD thesis, University of Buckingham, 1996.

[29] Jeremy MR Martin and SA Jassim. An efficient technique for deadlock analysis of large scale process networks. In *International Symposium of Formal Methods Europe*, pages 418–441. Springer, 1997.

[30] David May, Geoff Barrett, and David Shepherd. Designing chips that work. *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences*, 339(1652):3–19, 1992.

[31] Joël Ouaknine. Digitisation and full abstraction for dense-time model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 37–51. Springer, 2002.

[32] Hristina Palikareva, Joël Ouaknine, and A. W. Roscoe. SAT-solving in CSP trace refinement. *Science of Computer Programming*, 77(10-11):1178–1197, 2012.

[33] George M Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *International Colloquium on Automata, Languages, and Programming*, pages 314–323. Springer, 1986.

[34] A. W. Roscoe. Occam in the specification and verification of microprocessors. *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences*, 339(1652):137–151, 1992.

[35] A. W. Roscoe. Model-checking CSP. 1994.

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

[37] A. W. Roscoe. TTP: A case study in combining induction and data independence. Technical report, 1999.

[38] A. W. Roscoe. On the expressive power of CSP refinement. *Formal Aspects of Computing*, 17:93–112, 2005.

[39] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[40] A. W. Roscoe. Reflections on the need to de-skill CSP. *Proceedings of CPA 2014*, 2014.

[41] A. W. Roscoe. Card games as pointer structures: case studies in mobile CSP modelling. *arXiv preprint arXiv:1611.08418*, 2016.

[42] A. W. Roscoe, Pedro Antonino, and Jonathan Lawrence. *The Consensus Machine: Formalising Consensus in the Presence of Malign Agents*, pages 136–162. Springer Nature Switzerland, Cham, 2023.

[43] A. W. Roscoe, Pedro Antonino, and Jonathan Lawrence. *Abstracting and verifying decentralised systems in CSP*. Springer Nature Switzerland, Cham, To appear.

[44] A. W. Roscoe and Naiem Dathi. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289–327, 1987.

[45] A. W. Roscoe and M Goldsmith. The perfect spy for model- checking crypto- protocols. 1997.

[46] A. W. Roscoe and Philippa J Hopcroft. Slow abstraction via priority. In *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, pages 326–345. Springer, 2013.

[47] A. W. Roscoe and Zhenzhong Wu. Verifying statemate statecharts using CSP and FDR. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, volume 4260 of *Lecture Notes in Computer Science*, pages 324–341. Springer, 2006.

[48] PYA Ryan, SA Schneider, MH Goldsmith, G Lowe, and A. W. Roscoe. The modelling and analysis of security protocols: the CSP approach.

[49] Steve Schneider. *Concurrent and real-time systems: the CSP approach*. John Wiley & Sons, 1999.