CSP: a practical process algebra

Stephen D. Brookes and A.W. Roscoe
CMU and Oxford

February 12, 2021

Abstract

We recall our work with Tony Hoare in developing the process algebra form of CSP. The semantics we developed with him, based on sets of observable linear behaviours, led to a very distinctive style of practical application using refinement checking, as embodied in the FDR model checker. We outline the history of FDR, showing how its power has been enhanced over time, and we showcase some major industrial applications to demonstrate its versatility. We show that CSP is a process algebra with firm semantic foundations and a wide range of practical applications. In doing so we pay tribute to Tony's profound and continuing influence.

1 Introduction

In this paper we look at how CSP was developed: the language, its semantics, its implementation, and mechanized verification tools. We show how the refinement checker FDR evolved, and we survey some examples of the practical applications these developments have inspired. These include some of the most exciting industrial and academic applications of formal methods. Overall we want to demonstrate that the CSP framework combines theory with practice in a natural manner, and is widely applicable in the real world. This kind of combination, driven by an interplay between intuition, theoretical investigation, and tool development, is typical of Tony's attitude toward science and research.

The first part of the story of CSP and FDR amounts to a re-telling of the history recounted in Bill's contribution to Tony's 60th birthday Festschrift, held at Oxford in 1994 [56], offered here with the benefit of hindsight. Steve has added to this account with his own perspective, again tempered with experience gained by the passing of time. We include some thoughts prompted by memories of later symposia and events in honour of Tony. Our account may appear a little revisionistic in places, as we look back over events whose long-term influence, significance, or place in the overall scheme may not have been so clear at the time.

CSP in more or less its present form was developed in the late 1970s, as Tony worked to develop an appropriate abstract framework for modelling patterns of communication and concurrency. In fact Tony's own ideas evolved over this period, as can be seen from his publication record. By “current form” we mean the process algebra version of CSP described in the journal article [11] and later in Tony’s and Bill’s books [28, 57, 59]. The name came into existence earlier, used for an imperative programming language of “communicating sequential processes”, as outlined in Tony’s CACM paper [26]. By the late 70’s Tony had been thinking about the nature of concurrency for over ten years: how to structure it, how to reason about it, and what makes it difficult.

This paper recounts some key stages in the evolution of CSP, from its origins to its utility, the emergence of later variant forms of CSP, and how it came to be used in conjunction with formal verification tools, principally FDR. Tony’s direct day-to-day influence at the start of this story was immense, as we both became deeply involved in an intensely interactive research collaboration with him, aiming to put CSP on solid semantic foundations. The prominence of Tony’s rôle gradually reduced over the years as more people got involved, so that by the time when FDR itself was born the tool designers were operating more or less independently of him. Nevertheless his influence continued to be felt, both through the decisions made in the creation of CSP and the philosophy he imparted to it and to us. We have attempted to capture these influences in this paper, and we have tried to capture the flow of ideas and motivations without providing much mathematical detail. Another paper on FDR’s industrial impact, with a slightly different emphasis and slightly more technical detail of some of the stories related here, is [21]. In many ways that paper should be thought of as a counterpart to this one. For a deeper look at the mathematical underpinnings one can consult the original journal article known (at least to its authors) as HBR [11], and Bill’s books.

Tony has summed up his general philosophical attitude in a number of wise adages, many of which are frequently quoted. Since concurrency and distributed systems are undoubtedly complex, and abstraction is so evident in his design of CSP, its theories, and its use, industrial and otherwise, we invite our audience to recall this one:

> In the development of the understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction. Abstraction arises from the recognition of similarities between certain objects, situations, or processes in the real world and the decision to concentrate on these similarities and to ignore, for the time being, their differences.

Tony’s persistent search for realistic abstractions, grounded in intuition and understanding, has been a major driving force in the development of CSP.

2 Prehistory 1978-91

The two of us began doctoral studies at the Oxford University Programming Research Group (PRG) in 1978. We both began working with Joe Stoy. We had studied Maths together for the three years before that, as undergraduates at University College Oxford. In our final undergraduate year we had learned about the Scott-Strachey approach to denotational semantics, and its domain-theoretic foundations, in a series of lecture courses taught by Stoy. We were among the first cohort of students to obtain copies of Stoy's then new textbook [66], published in the United States but not yet generally available in the United Kingdom. Around the same time we studied domain theory in more depth with Dana Scott, and Steve served as a teaching assistant for Scott's lectures on the subject[1]. While Strachey had passed away in 1975, Scott remained a prominent figure in the small world of Oxford Computer Science, though his position and office were nominally associated with the Philosophy department. Tony's own academic grounding had been in Classics (known in Oxford as Greats, a degree which includes philosophy and logic), before his career path led him into computer science, so between us we covered quite a range. Tony had already developed an abiding interest in logic, philosophy and computers arising from his studies and his experience as a scientific translator [33]. At the time there was a feeling of excitement in the air, concerning denotational semantics and its potential applicability to an ever widening range of programming language paradigms. Concurrency offered a particularly interesting challenge.

Tony's journal article titled *Communicating Sequential Processes* appeared in 1978, published in the Communications of the ACM. Of course this was the origin of the acronym CSP. In this usage the name refers to an imperative language of sequential processes, each with a private internal state, operating concurrently and communicating by synchronized message passing. This language design offers an alternative and striking contrast to shared-memory parallelism, in which processes interact by reading and writing to a shared global state. Hoare's language was an elegant generalization of Dijkstra's non-deterministic guarded commands, extended with synchronous communication and concurrency. The syntactic constraint that processes in CSP are sequential rules out nested parallel composition, but allowed Tony to provide a conceptually simpler informal account of program behaviour. Despite this limitation Hoare's language was powerful enough to express a number of compelling example programs, and the paper argued forcefully for the benefits of a clean, simple language design. Although the CACM paper did not dwell on semantic issues in any depth, the combination of imperative features such as assignment with message-passing and concurrency raised some obvious questions. What kind of semantic domains would be required to properly account for these features in combination, using the techniques of denotational semantics?

Before continuing it might be helpful to offer a brief glossary for some technical terms needed in our account. In the denotational framework one must define semantic domains whose elements represent abstract meanings for programs, and then define a syntax-directed semantic function that maps programs into meanings. The main challenge is to find *suitable* sets of meanings to serve as semantic domains: abstract, mathematically tractable, and detailed enough

[1]After Dana took Steve with him to CMU in 1981, Bill taught domain theory based on Dana's *information systems* for many years at Oxford.

to properly account for program behaviour; furthermore the semantics must be compositional. The word "suitable" refers to the need for meanings to be "compositional": it must be possible to define the meaning of a program from the meanings of its syntactic subphrases. This characterization of a basic tenet of denotational semantics is attributed to Christopher Strachey (quoted in Stoy's book), but similar principles date back to Frege and even earlier, albeit in settings far removed from computer science. In contrast, the *operational* style of semantic description involves the definition of an abstract machine whose executions describe program behaviour. One typically defines a set of configurations for the abstract machine, and a transition relation describing the steps taken by the machine as it executes the program. There are, arguably, some advantages to the operational method: there is less need to invent semantic domains, and it may be quite straightforward to specify step-by-step behaviour. But these two approaches, denotational and operational, should really be seen as complementary instead of competitive. We began our foundational investigations with an open mind, using operational intuitions to guide our denotational quest.

David Park had given a denotational semantics for a simple shared-memory parallel language, in which processes denote sets of traces built from steps that describe the state changes made by the atomic actions performed by the process, allowing for interference by an "environment" of other processes running in parallel. Matthew Hennessy and Gordon Plotkin gave a shared-memory semantics using a semantic domain of "resumptions". Plotkin also gave an operational semantics for a version of imperative CSP that allowed nested parallel composition and included a more flexible mechanism for naming processes, showing that Hoare's syntax restrictions, imposed for pragmatic reasons, need not actually cause semantic complications. But none of these semantic accounts readily fit the bill for the CSP process algebra. Park's model is not obviously adaptable for a communication-based language; the technical details supporting resumption semantics are rather intricate, involving a recursively defined semantic domain built with the Plotkin powerdomain construction. Plotkin's operational model sheds no light on the kind of denotations that might be suitable for the process algebra.

Tony was certainly aware of the need for solid semantic foundations. He was surely aware of the work of Park and Plotkin. We both studied this work carefully, as well as Milner's work on CCS (A Calculus of Communicating Systems). All of these people were well known to Tony. Park was a frequent visitor to Oxford, at one time sharing an office with us in the Programming Research Group building on Banbury Road. Robin Milner has spoken eloquently of his deep personal friendship with Tony, and their long series of wide ranging discussions on the nature of concurrency. Foundational questions, and concern about the potential complications that might arise, must have influenced Tony's decision to disallow nested parallelism in the original CSP language, and this resulted in a language for which he felt comfortable giving an intuitive account. He was able to do this without too much concern for technical foundations, but in a manner convincing enough that the CACM paper is nowadays seen as a classic of clear exposition. When he did turn his attention to foundations, he chose to focus instead on a more abstract process algebra, devoid of assignable variables and mutable state. For historical reasons, to maintain links with the original language, the process algebra is still known as CSP; in retrospect, since the process algebra does permit nested parallelism, it might have been more accurate to have chosen the name CPP, for *Communicating Parallel Processes*.

By late 1978 Tony had already proposed a simple model of communicating processes, based on prefix-closed sets of finite traces. A trace is a sequence of events representing the input and output actions in which a process may engage. There is a very simple intuition behind this trace semantics, and it amounts to the first step towards our ultimate semantic framework based on observable linear behaviours. In essence, a sequence of actions is the simplest possible form of observable behaviour one can imagine. But it quickly became clear that trace semantics was too simple, as it was impossible to distinguish between a process that could either communicate or stop, and a process that must communicate.

The two of us recall a series of meetings held in Tony's office, in which Tony outlined the traces model and its defects, and the challenge emerged: to develop a more expressive model, capable of accounting for non-determinism and phenomena such as deadlock, based on Tony's strong intuitions about concurrency. As we recall, we two started to play with sets of traces equipped with "acceptance sets", ordered by superset, but after a short while we convinced ourselves and Tony that it would be more natural to work instead with the complementary notion of "refusal sets". We used the term "failure" to refer to a trace paired with a refusal set, so this was the origin of that nomenclature.

Even before this, Bill's first efforts under Tony's supervision had involved creating a (traditional domain-theoretic) semantics for the nascent CSP process algebra. In January 1979 Steve implemented this semantics using SIS (Semantic Implementation System), essentially a giant lambda-calculus reduction engine, built by Strachey's former student Peter Mosses, that allowed for test-bedding of denotational definitions. It soon became clear that this path of exploration would require computational power far beyond the resources available at the PRG. Even in the somewhat better equipped Nuclear Physics Laboratory we found that analysing the behaviour of a simple recursive process for recognizing palindromes over a two-letter alphabet was impractical. We abandoned that line of development and looked instead for a more tractable semantics and a more efficient way to perform analysis.

We feared that to handle the nondeterminism inherent in concurrent execution we might need to use powerdomains, and perhaps even a recursively defined domain akin to resumptions. [2] Eventually it became clear that we could in fact design a denotational model for CSP without resorting to such complexities: we could manage quite well with sets of suitably generalized traces (failures, divergences, and so on). Such sets, formulated with appropriate closure properties, form domains ordered by set inclusion. Equally well, they form domains

[2] There are three widely known powerdomain constructions: the Hoare, Smyth, and Plotkin powerdomains. This nomenclature is yet more evidence of Tony's wide influence. The use of his name in this context refers not to his work on CSP but to a connection with Hoare Logic and partial correctness.

when ordered by reverse containment, or "refinement". Loosely speaking, and grossly over-simplifying the technical challenges, powersets turned out to be sufficient. This was because Hoare, in paring down to the essence of concurrency, had stripped CSP of all of the other complexities such as procedure calls, rich data types and first class functions, namely the features of regular programming languages that typically require more semantic sophistication.

Subsequently Steve and Bill worked to flesh out a mathematical framework to validate the overall aim: processes should denote sets of observable behaviours, where the notion of observability should be closely tied to intuition and capable of handling deadlock and non-determinism. The realization that it is appropriate to model processes in this manner was driven by two important intuitions, advocated very strongly by Tony:

1. Processes and specifications are essentially the same thing: sets of observable behaviours. In the first case this represents the behaviours that a process might be seen to perform. In the second it is the set of behaviours that a specification permits. This leads to a strong focus on behavioural specifications: assertions about processes that all their individual behaviours are correct, rather specifications ones which need to go beyond this, for instance asserting that a process treats all members of a set A of events symmetrically. So Tony, in his book, concentrates on the "satisfies" relationship: every trace of P satisfies some predicate R on traces. Specifications of this form are termed *behavioural*.

2. Refinement lies at the core of reasoning about CSP. Refinement $P \sqsubseteq Q$ is interpreted as reverse containment on sets of behaviours: $P \supseteq Q$. This is natural under these conditions, making refinement very intuitive and easy to exploit: a process P satisfies behavioural specification S if and only if $S \sqsubseteq P$. For every behavioural specification R there is a least refined process P_R that satisfies R, and we immediately get that a process Q satisfies R if and only it refines P_R.

It is typical of Tony that, having taken inspiration from elegant but potentially Baroque theories he would seek to find his own style, one that was both simple and intuitively right, without requiring deep background knowledge of domain theory by way of justification.

Traces alone gave insufficient information to account for deadlock and non-determinism, so we looked for ways to enhance traces with extra information. Tony's instinct was to look for a solution that was just expressive enough, rather than to go for one that could potentially capture too much detail: search for the most abstract solution that is just concrete enough to do the job. Hence the "traces plus" philosophy that has been embodied in the CSP approach to semantic modelling ever since. In essence what we did was to use richer collections of behaviours as the need arose.

A major step along the way, indeed the step that paved the way for FDR, concerns the failures model of CSP. This emerged as the fruit of an intensive collaborative effort between Tony and the two of us, culminating in the JACM paper [11]. This paper introduced the concept of modelling concurrent processes as sets of failures. A failure is a finite trace plus a set of events refused at the end of it; this coupling of a trace with a refusal set is the minimal way to equip traces with additional information that allows a compositional treatment of deadlock and nondeterminististm. However, that paper left open some issues that we only resolved later with the introduction of divergences and divergence strictness. To begin to understand these notions, first think about the traces model. First note that trace sets, ordered by set inclusion, form an especially simple kind of Scott domain – a complete lattice, with both a least element ($STOP$, the process with the empty set of traces) and a greatest element (RUN, the process that has all possible traces). RUN is also the least element under the refinement ordering. It should come as no surprise that any process algebra term $F(p)$ with a free process variable p represents a monotone function of the trace set that is put in place of p: the more traces p has, the more $F(p)$ has. Also note that monotonicity means the same thing, even if we reverse the ordering.

In denotational semantics one computes the meaning of a recursive term defined by $p = F(p)$ as a fixed point of the function F: usually the least fixed point in some ordering on the underlying semantic domain. In the traces model, as outlined above, there are two obvious choices. The one suggested by the obvious implementation of recursion – a recursive term is simply unwound – is the least fixed point with respect to the subset ordering; this fixed point contains all and only the traces that are in $F^n(STOP)$ for some finite number n, so obtainable by finitely many unfoldings of the recursion. The intuition behind this is simple: a finite trace of the recursive process always "appears" in a finite time, so belongs to $F^n(STOP)$ for some n. A recursive process is deemed to have just those traces that can be proven to exist by unfolding its definition.

However, taking a specification-oriented view, one might argue instead that a recursive term should be assumed to have any behaviour it cannot be proved not to have. This suggests interpreting a recursive process as a least fixed point with respect to the refinement ordering, or equivalently as a greatest fixed point with respect to set inclusion. In some ways this attitude might have been closer to Hoare's philosophy at the time. In [27] he is a little vague about this, stating that there is an analogy with BNF grammars, but this idea seems to align more with the subset-least approach than the refinement-least approach. In any case it was of no importance for any of the examples in his paper, for in common with virtually all sensible elementary recursive definitions in CSP, they all have the *unique fixed point property*, meaning that the greatest and least fixed points are equal.

With the failures model, presented in [29, 11], we had no such luxury of choice: the structure of the model meant that there was no subset-least element. This led us to use instead the refinement-least fixed point. Thus the denotation of the recursion that unwinds for ever, namely $p = p$, and some other unguarded examples, is $CHAOS$, the process with every trace and every possible refusal set. This, and the decision to identify any process that could perform an infinite sequence of hidden actions with $CHAOS$, seemed to say that a process engaged in such a divergence can refuse anything, which is a natural idea, and also hints strongly at divergence strictness. As Tony said, a process that might diverge is a catastrophe. Incidentally, for guarded recursive definitions the lack of choice here is irrelevant, because again there is a unique fixed point and thus it must coincide with the greatest one [28, 54].

Unfortunately the failures model was still not quite what we needed; we noticed some subtle defects, which we spotted in algebraic terms: an intuitively reasonable law of process equivalence (involving the hiding operator and divergence) was not validated by the semantics. Divergence was interpreted as the existence of a potentially infinite sequence of invisible actions, and the failures model did not properly account for this kind of undesirable behaviour. This made it difficult to reconcile the picture of a process as a set of failures with operational intuition. These problems were resolved by clearly distinguishing between stable process states (ones with no invisible action available) and unstable process states (where a possible invisible step exists), and by recording divergence explicitly. More than anything else the key to this resolution was to understand the relationship between the behaviours recorded in CSP models and an operational semantics for CSP that we had worked out recently, expressed in the form of a Labelled Transition System [10, 54, 14]. This LTS style of operational description was also used for CCS [43, 42].

This series of development steps, gradually incorporating more detail to overcome problems, culminated in the failures-divergences model [12], in which divergence is modelled as a catastrophe: any process with the capability of diverging is equated with the refinement-least process. This form of divergence-strictness became a key ingredient in the standard theory of CSP.

A feature common to all of these trace-based models of CSP, from (just) traces to failures and failures+divergences, is the use of *prefix-closed* sets of traces. But why prefix-closure? Again the answer goes back to Tony's desire to connect with intuition. If one can observe a sequence of actions, it makes sense to say that one must also have been able to observe each finite prefix of that sequence, stage by stage as the sequence evolves. So these semantic models all deal with traces that represent *partial* or *incomplete* program executions.

Looking back, it is now evident that we made a number of design choices, based on intuitions about program behaviour and the nature of observable behaviour. Of course these choices helped to focus our technical investigations and guided us towards our choice of semantics. Other design choices could have been made, if we had been motivated to take a different view. For example in a series of later papers, Brookes shows how to develop a semantic framework based on *complete* traces, including infinite traces, noting that it then becomes possible to reason about liveness properties (something good eventually happens) as well as safety properties (something bad never happens). Absence of deadlock, and freedom from divergence, can both be seen as safety properties, so one might say in retrospect that we chose to focus on safety properties alone. Brookes also described a denotational semantic model using partial orders (pomsets) rather than traces (linear orders), embracing ideas from so-called “true concurrency”.

As already mentioned, Milner's CCS is another prominent process algebra, developed at Edinburgh during a period of years that overlaps with the gestation of CSP at Oxford. Milner's CCS book, published by Springer, appeared in 1980 before HBR (1981 in draft form as a tech report and in JACM 1984) and Tony's CSP book (1985). but after Hoare's CACM paper (1978). It is fair to say that the two process algebras emerged independently and grew up separately. According to Milner, in CCS "There is nothing canonical about the choice of the basic combinators, even though they were chosen with great attention to economy. What characterises our calculus is not the exact choice of combinators, but rather the choice of interpretation and of mathematical framework." The two process algebras are based on very different interpretations and mathematical foundations. CCS processes were interpreted operationally with labelled transition systems, using "bisimulation" as the notion of process equivalence; bisimulation analyses the branching behaviour of processes. In CSP processes denote sets of linear behaviours, and process equivalence is based on failures and divergences. As a rough and ready comparison, CSP is trace-based, with a linear-time philosophy, while CCS is tree-based with branching-time. The identification of CCS and CSP as two halves of this dichotomy should be taken with a grain of salt, as it over-simplifies the story, but it does serve to emphasise their differences. Nevertheless these two process algebras have many shared aims, and each has had major impact in the practical world of modelling, specifying and verifying concurrent systems. For a detailed account of the relationship between models of CSP and CCS the reader is encouraged to look at a paper by Brookes[**?**]. Tools for model-checking based on CCS (and its successors) and bisimilarity have been successfully deployed in a variety of industrial settings, a situation that resembles that of CSP and FDR.

As an aside, Tony's attitude towards operational semantics has always been broad and not constrained by the convenient "tyranny" of labelled transition systems. For example, an entirely different approach based on algebraic reductions to something resembling normal form, as in [61, 30], was used in Tony's work on unifying theories [31].

By the time we published the failures+divergences model, CSP and its semantics were already finding applications, notably through occam and the Transputer, of which the story is told elsewhere in this volume. CSP, meanwhile, was still largely regarded as a blackboard language, used for developing high level and generally small illustrative models as experiments, for studying the construction of classes of systems such as routing networks. CSP was of course ideally suited for this latter rôle, because of its built-in capacity for compositional reasoning about properties such as freedom from deadlock: see [13, 60, 39, 1], for example. As a slight rephrasing of a well known saying, to justify our evolutionary approach to semantic development, we offer:

Industrial necessity is the mother of invention (Principle A)

Over and over again this principle has driven important innovation in CSP and related tool development and use. Frequently that innovation has been theoretical: we were driven to build theory by the need to realise some industrial need. Reluctance to having to say “that can’t be done” to a user can be a remarkable motivator. This was first evident in the development of the first industrially relevant verification tool, the occam transformation system, which manipulated a language that is essentially the process algebra version of CSP with the addition of imperative state. This was because of the practical utility of occam, particularly in the design of the transputer itself where it was treated as a low level language, meaning that the occam transformation could directly manipulate hardware designs. It is so much more exciting to develop theories and tools with an immediate practical objective to test success.

Another saying that may be adapted to our story is

A little theoretical knowledge can be a dangerous thing. (Principle B)

By this we mean that if some theory tells you that if some problem is too hard to solve, particularly in the worst case, that does not necessarily imply that it is not worth trying to create a practical solution. The prime example of this in our tale is as follows: as the need for automated formal methods tools became apparent during the 1980’s, and model checkers for other notations came onto the scene, a paper [34] was published which seemed to indicate that there was no point in trying to do this for CSP: Kannellakis and Smolka established that equivalence checking between processes over any CSP-style behavioural model is, in the worst case PSPACE-hard, measured in terms of the state-space size of processes. This size is the number of states of the corresponding LTS, which can be exponential in the size of the process description in CSP. PSPACE-hard is a higher (worse) complexity class than NP-complete, which at least in the late 1980’s was regarded as an indication of impracticality by many.[3] Equivalence checking is of the same complexity as refinement checking. Since it was apparent that we would likely need a refinement checker to best automate CSP verification, this negative result put a damper on our ambition to create one, and certainly delayed its emergence by several years.

3 FDR1 1991-96: Communication, fault tolerance and the beginning of time

Fortunately, however, Principle A above rescued Bill and his group from failing to understand Principle B. While the best known work in its collaboration with inmos centred on analysis through occam, the modelling of transputers also involved CSP in various ways, particularly in the group’s work relating to the H2 or T9000 transputer with its pipelined processor and dedicated multiplexed link hardware [55]. Geoff Barrett’s analysis of the latter led to him asking us to build a CSP refinement checker to support this.

[3]The perception of how intractable NP problems are has changed significantly in the intervening years, thanks to the emergence of SAT-checkers, although PSPACE-hard problems are still mostly viewed as impractical. Thus in the late 1980’s we thought that “Automating the analysis of CSP models looks intractable because it is NP-hard.” Nowadays we might have concluded the same but with NP replaced by PSPACE.

Forced to think about the problem, we realised that the obvious algorithm for checking equivalence or refinement between two automata P and Q in any CSP model (say failures-divergences) is indeed exponential in the worst case in the state spaces of each of them, as P,Q are normalised or "determinised" [4], meaning that for each trace of the input process there is only one state of its normal form. In the worst case the normal form has exponentially many states (essentially 2^N) compared to the number N of states of the input process. This is illustrated with FDR examples in [57].

1. In most practical cases the normal forms are much smaller than this bound, frequently being smaller than the original process.

2. In most cases where P represents a specification process, it is significantly smaller than the implementation Q: often with low single figures of states.

3. To check the refinement $P \sqsubseteq Q$ there is no need to normalise Q: in the worst case the product state space of Q and the normal form of P is explored, using an algorithm quite different from those previously used to verify automata against LTL formulae and similar.

Roscoe formulated FDR's refinement checking algorithm with David Jackson, a former student of Mike Reed, at a meeting in Auch, France, in summer 1991 that was memorable for many more reasons than that. By that time Mike, Bill, Michael Goldsmith and others had already created the company Formal Systems (Europe) Ltd (FSEL) as the vehicle for the continuing collaboration with inmos, and so it was FSEL that created FDR (standing for Failures Divergence Refinement) in the summer and autumn of 1991. They were excited that it was able to handle Geoff's examples, and indeed handle CSP systems with tens of thousands of states on machines such as the Sun workstations, which were then ubiquitous in our circles.

FDR has always supported the refinement checking of pairs of finite state CSP processes in at least the following three models: the traces model and failures/divergences model as already discussed, plus the *stable failures model*. The existence of FDR forced Bill to discover the last (though he was also influenced by a conversation in MIT with Albert Meyer about his work with Lalita Jategaonkar). This model makes a clear distinction between stable failures, which it records, namely a trace coupled with a set of events a τ-free (i.e. stable) state cannot perform, and the non-acceptance from a set of events because a process is diverging. The stable failures model models a process as its finite traces and stable failures. Its most obvious use is as a more efficient way of analysing processes which are known to be divergence-free so that the expense of calculating divergence information is wasted. But it has many others.

This was not the only time when industrial necessity caused us to discover a model that our theoretical expectations had concealed from us: the same thing

[4]This means determinised in the automata theoretic sense, which does not turn them into deterministic CSP processes: annotations are used to make individual states nondeterministic where necessary.

happened later with the revivals model [58]. Yet again industrial necessity removed blinkers that had been placed by our theoretical expectations.

As constructed, FDR could naturally prove the absence of two of the three fundamental pathologies of concurrency: deadlock and divergence (which can manifest itself as a network engaging in an infinite series of hidden internal communications). Their respective absences reduce to simple refinement checks respectively over stable failures and failures-divergences.

Completing the set of pathologies, FDR has also always been able to calculate whether a process is deterministic in the extensional sense arising from the failures-divergences model, rather whether its operational semantics is structurally, or intensionally deterministic. In this it reaches firmly back to Tony's original instincts about understanding systems. Determinism cannot be expressed directly as a refinement, but separate cunning reductions to refinement were discovered by Bill and Ranko Lazić: both involve two copies of the object process P in the check so that you can compare P against itself.

The paper [56] reported on the first version of FDR, which with the benefit of hindsight we will rename FDR1, with that paper also positing ideas that took shape in later versions. The most interesting example in that paper illustrated an "off piste" use of FDR, namely not really examining a concurrent system as such but a combinatorial system – the game of peg solitaire – reformulated as one. This shows beautifully how tools of its ilk can solve problems which are really difficult for humans, and do so reliably. This type of demonstration that model checkers, at least in some cases transcend what humans can achieve, is always valuable.

At the time of [56] FDR could handle a version of Solitaire with about 0.5

While with occam we had a practical language that could describe serious implementations, we built FDR for what was at that point a blackboard language with a rather flexible notation, to some extent in terms of what process operators it allowed, and certainly in the way that process "state", namely communications introduced by input, by choice constructs and by parameterisation, and used in constructing data-based communications, were handled. The language of FDR1 was an attempt to capture the style of CSP in Tony's book, and so had a declarative semantics, both binary and indexed versions of most operators, and a "script" style of presenting definitions including single and mutual recursions. Thus a program was a script of channel declarations, definitions of non-process objects and (typically recursive) processes side by side. It was, however, clearly just an approximation to what was needed,

Heavily influenced by the style of functional programming represented by Haskell, in the mid 1990s this developed into a language (mainly developed by Bryan Scattergood [65]) which could reasonably be described as a Haskell-like functional language with

1. Essentially no support for character and string handling.

2. Finite sets of sub-process objects as first class objects, because these had always played a vital role in CSP (for various indexing and selection pur-

poses). (The type of sets has no restriction to being finite, but the range of things one can do with infinite sets is very limited.)

3. An extra construct . (Infix dot) for building compound objects (often communications), following the event-building conventions that had arisen in CSP and other process algebras. It typically attaches components to channel names as in c.2.true, and is overloaded to create complete and partial objects in user-defined types.

4. A distinguished type representing CSP processes.

5. Notation for describing assertions: properties such as refinement of processes that FDR would check.

With only small modifications, this CSPM language remains what FDR uses today.

FDR quickly became popular among those using CSP and therefore rapidly "implemented" many small to medium sized examples that had already been studied in CSP-based papers such as communications protocols, potentially deadlocking systems and a wide variety of puzzles. The transputer/occam community, of which we were a part, became frequent users of FDR for experimentation both in academic and industrial circles as is demonstrated by the many papers that emerged from it in the years roughly 1992-96. Notable users in that period included IBM UK Labs at Hursley and a commercial and academic group led by Jan Peleska at Bremen. IBM used it in investigating various concurrent systems, often in customer-specific work. The Bremen group used it for test generation in avionics, and they and others have continued to use it for test generation [50, 51].

However the most interesting project for Roscoe in the early days of FDR was a collaboration with Charles Stark Draper Labs[5] of Cambridge Massachusetts [9]. They were building a quadruply replicated processor (for fault tolerance, so FTP) from transputers and required verification of its properties. Tony had long before introduced the idea of modelling faulty components in CSP and analysing the resulting, frequently nondeterministic, systems that are built with them. From the standpoint of CSP the key insight here was that in order for multiply redundant systems with voting to work properly, the individual behaviours in the absence of faults must be deterministic. If not, a four lane system might find itself voting between four different but valid answers, or two pairs of equal ones, even without faults; and that in the case where one had a fault it would be impossible to distinguish the incorrect answer from three different but correct alternatives.

The FTP implementation made significant use of measuring the passage of time, not something incorporated with the CSP of [28] or implemented in the first versions of FDR. On the other hand, a real time version of CSP had been developed by Mike Reed and Roscoe [53], but the dense time model it used meant that it was then far removed from FDR analysis. It was for the

[5]Most of Bill's collaborations with Draper were led for them by Neil Brock.

FTP work, combined with a challenge from Connie Heitmeyer reported in [25] (namely the level crossing example) that Roscoe developed the *tock*-CSP dialect that includes a special event representing the regular but discrete passage of time. In this, all timed processes were obliged to synchronise on *tock*, and the timing of component processes is captured by the places there they can or must communicate *tock*. There are expositions of *tock*-CSP in [57] and [59], the second of which compares the two timed versions.

When developed, *tock*-CSP was seen as a work-around to meet industrial necessity in place of Timed CSP, and indeed it was used in a wide range of academic and industrial case studies in that decade. It often then seemed more popular in practical use than the untimed interpretation of CSP.

The FDR1 era also saw the beginnings of the application of FDR to security both at Oxford and elsewhere, requiring the following:

- The descriptions of CSP systems over very complex data and data types, and the encapsulation of arbitrary attacker behaviour.

- Capturing complex requirements, some of which can be represented behaviourally, and some not.

- The latter requires novel forms of abstraction.

These emphasised the shortcomings of the relatively "cheap and cheerful" machine-readable dialect of CSP supported by FDR1, and pushed forward CSPM as discussed above.

Applications also highlighted two shortcomings in the core implementation of FDR: firstly, in common with other model checkers at the time, it performed very poorly when forced to use virtual memory. That was a big issue because computers had far less memory than we see today: in the early 1990's say 8 or 16Mb was typical in a high-end workstation, while hard (magnetic) discs would be 200Mb or more. The second was the way it recognised that a parallel state (represented as a packed representation of the enumerated component state) can perform an action. Specifically, FDR1 recognised the combinations of component states that were positively engaged in each top level action. This had the tendency of exploding in some cases, particularly with many-way synchronisation and/or when many states of a component process can perform that component's contribution.

4 FDR2 1994-2007 at FSEL: Protocols, abstraction and industrial applications

The success of the FDR tool and the known drawbacks convinced us, via Formal Systems, to create FDR2: a complete rewrite which solved the above problems. The memory management problem was solved for breadth-first search by the realisation that we could much more efficiently consider all the successor states for a given ply together in sorted order rather than individually, meaning that memory access is far from the random pattern seen when storing states in hash tables, as was then the state of the art hash tables. Initially this was done by storing states in sorted lists, later these evolved to B-Trees. (Checks involving divergence were not helped by this, because these use depth first search.) The second problem was addressed by Bill's creation of supercombinators[6] as a novel implementation technique for CSP and similar languages.

At the same time there was growing realisation of the problems caused by the state explosion problem whereby the number of states of a concurrent system (and thus the time and space required to tabulate them) typically grows exponentially with the size of the system. Many approaches to overcoming this were already being developed for a variety of model checking tools, including compression, data independence, partial order reduction and BDD representations. FDR2 initially followed the approach of compressions based on CSP models, and a specialised partial order approach called *chase* which, with the help of semantic insight from the user could collapse the state spaces of deterministic and similar systems involving many τs.

chase was created for intruder modelling in computer security, as part of what was probably the most influential use of FDR. Before Gavin Lowe [37] showed that it was possible to model and analyse cryptographic protocols in CSP and FDR, analysis tools for these were specialised and narrow. Gavin's demonstration that it was possible with general purpose tools triggered an explosion of research in that area and led to a huge expansion of the understanding and provability of these protocols. There were many papers and a book [64] on this approach.

As part of this general project Gavin created Casper [38], a tool which translates a protocol model in typical notation into a CSP script that checks chosen properties of the protocol. So effective was this that mediocre students could produce really impressive-looking work. Casper was the first major exercise in automatic translation into CSP for verification using FDR, something that has been repeated for widely different notations since. Indeed most major practical uses of FDR since have been by this route because, whether we like it or not, the industrially effective use of verified software engineering are almost always based on languages that engineers and domain experts already use to describe and develop systems.

The *chase* function was introduced to allow crypto protocol intruders – the processes that accumulate and use in any feasible way the information the can glean from spying on legitimate parties – to be modelled in CSP for analysis on FDR. The natural CSP model or the intruder is a sequential process applying closure under an inference system of everything that has been learned to date. However, that simple representation creates a component process that is too large for FDR to model in its usual mode. *chase* was part of the solution: dividing the intruder into the parallel composition of small components and driving all available inferences in arbitrary order. That it was there a valid

[6]The name supercombinator was inspired by a similar idea used in functional programming by John Hughes [32].

operation comes back to Tony's concentration on the extensional property of determinism: the pre-*chase* intruder is deterministic, so the fact that *chase* resolves what look like nondeterministic choices in it is guaranteed not to change its semantics.

By the mid 1990s Bill's group were working a lot with a group of researchers at DRA (later DERA and QinetiQ) on potential uses of FDR in their work, including security and reliability analysis, led by Colin O'Halloran. Shortly after developing the *chase* model of the intruder, Bill was presented by DRA with a problem which was summarised as representing the reliability and fault tolerance of the integration of two (then unspecified) legacy systems. Since much of that analysis was similar to analysing the reactions of an inference system under a wide variety of changing stimuli, he recognised similarities with the intruder model and so sketched a solution involving *chase*. Within a few weeks two things became clear: first that this suggestion had been a resounding success and secondly that the pieces of legacy software were those controlling British nuclear submarines and US cruise missiles (TLAM) [46].

The point of this whole exercise was creating the mandated Release to Service case of a piece of hardware, an important part of the regulatory framework around UK defence systems and similar safety-critical arenas. This initiated a large programme of using CSP and FDR within DRA/DERA/QinetiQ of release-to-service cases, as reported in [71], including large parts of the avionics of the Typhoon jet fighter: Figure 1.

Crucial to most of these applications is the ability of CSP to form abstractions of systems. Abstraction, as explained in the quotation in the Introduction, means getting rid of the detail you do not need. Tony provided CSP with abstraction mechanisms through many-to-one renaming and hiding, which he separated from parallel composition. Roscoe supplemented these with mechanisms such as lazy abstraction, which is usually the correct form beyond the traces model, mixed abstraction and slow abstraction, the latter being directly industrially motivated [62].

Translating more languages into CSP

Much of the work on these release to service cases was carried out using tools for translating other programming notations into CSP. This involved similar integrations using variations on the *chase*-based technology, and also verifying systems themselves (as opposed to integrations) using tools for translating notations such as Statecharts into CSPM.

Even though the logical model of Statecharts, with hierarchies of state machines and actions, and prioritised actions, and indeed shared variables, is so different from CSP, it proved possible to translate them into CSP thanks to the powerful renaming capabilities and multi-way synchronisation Tony had built into the language. So for example at times when a high priority process can do nothing, it might perform a *no-op* action which is renamed to synchronise with any of the lower priority actions thus enabled [63].

Tony suggested in his early writings on CSP that variables, and in particular shared variables, could be modelled by running them in parallel with each other and the threads that use them. The system becomes (in CSPM) `Threads [|{|read,write|}|] Vars`, where `Vars` is the interleaving of a simple process `Var(n,i(n))` where `n` is the name of the variable and `i(n)` is its initial value:

```
Var(n,x) = read?m!n!x -> Var(n,x)
           [] write?m!n?y -> Var(n,y)
```

In the case where shared variables are the only means of the threads interacting, `Threads\verb` can itself be a simple interleaving.

On the surface this coding seems terribly inefficient because the state spaces of Vars and Threads both multiply up unconstrained. However re-arranging the system so that each `Var(n,i(n))` is grouped with a thread that makes extensive use of it, and then compressing these groups proves enormously effective as demonstrated in creating the SVA tool [59] (Chapters 18 and 19): this takes a pure shared variable program and models it (with many options) in CSP. So effective is this coding that it models all variables, not just shared ones, in this way.

One of the keys to the practical industrial use of a tool like FDR is finding classes of industrial scale systems where the state explosion problem can be tackled reliably. We need non-specialist engineers to be able to apply the tool to the systems they develop without needing to be creative in finding new ways of cutting huge state spaces each time.

This is true of compressing variable-thread combinations as set out above. It is true of systematic uses of *chase* and similar over structures where it is predictably effective. It is also typically true of Statecharts, because though their semantics seem complex, they impose a deliberately (though intuitively surprising) sequential behaviour on systems thanks largely to the elaborate priority rules that we have already referred to.

As reported in [21, 70], two former Oxford students, the father-and-daughter team of Guy Broadfoot and Philippa Hopcroft, identified the possibility of combining CSP and FDR and existing software engineering approaches [8] to create a compositional approach to the development of state-machines based embedded control systems. They concentrated on building something that engineers can use for building reliable systems: they used a fixed, tree-like architectural style in which components interact with neighbours, expecting responses from neighbours and seeking a number of standard properties of the resulting systems such as responsiveness. At the same time each component has a design specification and interface specifications. The core of the method is to establish that each component's design meets its own interface specification and the standard specifications. The overall tree structure and these results then guarantee the overall coherence of the system.

This approach, which was christened Analytical Software Design (ASD), was based on a tabular notation that was intended to be accessible by engineers. This notation was based on the Sequence-Based Specification method [52]. The

Figure 1: Well established impact: Military release-to-service (Typhoon fighter) and Verified embedded software (ASML photolithography)

resulting tools could capture designs, generate and interpret the FDR2 checks of coherence, and generate code implementing the resulting systems. It has had considerable success in that is was adopted by a number of large companies (e.g. Philips Healthcare and the photolithography company ASML: see Figure 1) who developed enormous (hundreds of thousands of lines of code) verified systems and achieved the improved quality and reduced costs as reported in [70]. In terms of scale and scalability of verification, it was by some way the most successful we knew of, representing a form of compositional verification. This was possible because of the refinement, modelling and abstraction capabilities of CSP and because ASD's inventors were able to find such a useful class of systems that was simultaneously tractable to verify and challenging to develop correctly without verification.

5 FDR2 into academia: exploring implicit checking, and Timed CSP (2007-12)

As we write this, FDR is approaching 30 years old. For its first roughly 10 years, it was actively developed by FSEL: while sales of the software never came remotely close to paying for this work, research-style contracts from organisations like DERA who wanted further functionality were sufficient. For the next few years, while the tool itself was having great technical successes, the funding environment changed and it was no longer possible to fund core development in the "commercial" environment of FSEL. That, and possibilities for exploring further model checking techniques in the context of FDR, made it logical to move the tool to what was in many ways its spiritual home, the Computer Science Department of Oxford University. So from 2007-11 FDR2 development was supported by an EPSRC grant there [3], covering

- The implementation of Timed CSP. This was reported in [4], though an important gap has only been filled very recently, as we will see later. The implementation was possible thanks to the work of Joel Ouaknine who showed [45] that Timed CSP could be reinterpreted over a discrete time domain in such a way that satisfaction by the same term of continuous and discrete properties are closely related. He further showed how the discrete interpretation could be modelled by FDR with some small additions: modified versions of two operators to give them the correct treatment of *tock*.
- Experimentation with several techniques which, in other contexts, had proved successful in reducing the state explosion problem, including SAT

checking [48] based implementations of refinement, CEGAR [47] and Counter Abstraction [40].

- The implementation of more models of untimed CSP, in addition to the three discussed above.

The Timed CSP implementation was very successful in terms of coping with complex implementations, but lacked support for compositional reasoning, something only rectified very recently [41].

The SAT checking and CEGAR experiments failed to produce the spectacular results anticipated. In the case of SAT this was despite making the usual compromise of restricting to bounded depth model checking. Probably this was because the steps of creating the usual two-layer FDR representation of a system plus the normal form specification plus the translation into propositional calculus created too many variables. The SAT experiment was carried out twice with similar results.

It became clear that the implementation of additional models is quite complex given the niche user communities they are likely to attract. With the benefit of hindsight the most significant addition to FDR2 in this dimension was the priority operator proposed in [59].

In its final years, including the above, FDR2 had an intractably complex code base which made natural developments such as multi-core and cluster parallelisation difficult to realise and support, though a prototype was reported in [24]. FDR2 had no integrated type-checker for CSPM, as a consequence of considerable complexities introduced by its treatment of . (infix dot), which was a major drawback. (There was a separate but slow and cumbersome type-checker for its last few years.)

6 FDR3 and FDR4: back to basics and into the cloud (2012-19)

As we have seen, FDR2 was around for many years (about 18) and went through many versions. At the end it was increasingly in need of a re-write. Fortunately salvation stepped forward in the person of Tom Gibson-Robinson who essentially single-handedly took this on, both during his spare time in his unrelated doctoral project, and later under sponsorship from DARPA found for us by our old collaborators Draper. The result was FDR3 which thus, like FDR1, arose out of an industrial collaboration, in this case the use of FDR as part of the “red team” (attacking, fault finding) activities in the HACMS (high assurance autonomous vehicle) project.

The key references on the development of FDR3, including more detail and objective comparisons, are [19]. It did not carry forward some of the less used features of FDR such as the additional models.

Tom quickly solved the type-checking problem by excluding cases of “.” that were little or never used. This is, as one might have hoped all along, fully integrated into the tool.

Tom simply re-wrote the entire tool from scratch, producing cleaner implementations of the same core CSP manipulations and greatly improving on the B-tree structures used to store bulk states. FDR3 using a single core gives both speed-up and better memory performance of a factor typically 2-3 over FDR2 [19].

All versions of FDR have used a two-level implementation strategy for CSP: the tool identifies the low-level components (generally the sequential components) of a system, compiles these into explicit state machines, and devises recipes for combining these into an overall machine using (since FDR2) supercombinators and strategies such as compression. The compilation stage was frequently a major hurdle for industrial application of FDR2, but Tom redesigned this completely and greatly reduced this problem.

Just as the user interface was redesigned completely between FDR1 and FDR2, so it was again for FDR3. The main impact of this in use was on debugging (of processes that fail to meet their specifications), where FDR3 gives much more information in tabular form and is integrated with a CSP simulator.

By the time that FDR3 was created, the hitherto relentless speeding up and power of a single processing core had substantially levelled off and was being replaced by multi-core, a phenomenon which itself provided a bridge to cluster architectures. Not everything that FDR does parallelises readily, but fortunately

1. FDR frequently spawns off separate tasks such as the compressions of component processes that can be farmed out to separate cores.

2. The core BFS search used for refinement checks parallelises extremely well by the simple expedient of choosing a randomising hash function whose range is the number of cores, making core n responsible for storing and checking the states with hash value n, and trading states between the respective cores (e.g. as B-trees) each ply.

The chief hurdle to parallelisation is algorithmic on any checking mode that requires depth-first search (DFS) or similar activities such as analysing fairness via analysing strongly-connected components of large directed graphs. Such activities (e.g. the DFS used for FDR's divergence check, or algorithms such as Tarjan's or Dijkstra's [68, 17] for finding SCCs, do not seem to parallelise well, though efforts such as [37] give partial solutions.

The first release of FDR3 implemented this on multi-core systems, frequently achieving at least linear speed up with the number of cores as described in [20]. FDR3, with all this capability, became a much more capable and professional tool where, at least in an academic context, the limits of what it could cope with were problematic much less frequently than before.

Thus there was less burning need to make FDR available on supercomputer style clusters than the need for extra capacity in previous years. But it was clearly possible and potentially valuable where a practical check had a huge number of states. Tom therefore created a version of FDR that runs on MPI cluster architectures, as was reported in [20]. The result was a version that, for all the clusters we could then easily get our hands on in 2015, namely up to over 1000 cores, we again got close to linear speed up – sometimes a little better, sometimes a little worse.

Having moved through the thousands to the millions and then the billions of states processed by FDR, we deliberately created one with nearly 1.2 trillion explicit states, namely a large version of Lamport's bakery algorithm generated by SVA, using its inbuilt compressions, without which there would have been orders of magnitude more (we estimated 10^{20}). This ran on the Amazon EC2 cloud (64 times 16 cores) in about 5.7 hours and 6Tb of store, namely about the same absolute time and very similar memory per state that FDR1 did on restricted versions of peg solitaire with about 600,000 states 23 years earlier. We will discuss this further later.

Recall that FDR has long supported the partial order method *chase* whose validity is mainly restricted to systems known to have the extensional determinism property. Many other types of partial order methods – rules that allow regions of process state spaces to be ignored when searching for potential counter-examples – have been proposed by the model checking community based on other, generally more operational restrictions, for example [49]. The stubborn set partial order reduction [72] implementation was added to FDR and reported in [22].

Modern approaches to implementing concurrent, generally multi-core systems frequently depend on shared memory governed by locks or lock-free structures. Such systems frequently involve structures naturally modelled using pointer-based cells such as queues, trees and linked lists. These represent a major problem for the approach SVA and similar front ends to FDR take to representing state because of the allocation of N cells to N jobs can happen in $N!$ ways, with there being no difference in how the result behaves. The type of cell names is symmetric: any permutation of these names results in identical externally observable behaviour. Tom Gibson-Robinson and Gavin Lowe [23] developed methods of detecting such symmetry in CSPM and factoring it out of refinement checks. In effect that means reducing each discovered state to a symmetry normal form. Where applicable this can reduce the state space by a factor f up to $N!$, where N is the size of the symmetric type (typically the number of cells in the heap model).

Pragmatically this allows us to develop and analyse shared memory programs with much more interesting data types than SVA, but with otherwise similar reach.

Both partial order reduction and symmetry reduction are accessed in FDR by invoking special modes of check. Like all the other modes of implicit model checking that FDR supports they sometimes work wonderfully and sometimes get in the way. FDR has always been able to handle explicit checking efficiently over a powerful and expressive language, and more than anything else this is its core strength.

In 2016, taking account of the many new capabilities since FDR3 was released, the FDR4 release was announced.

FDR3 and FDR4 are much superior tools to FDR2, but in terms of industrial

Figure 2: Verifying autonomous vehicles: catching snails to prevent disease; UAV for underwater engineering

applications the period 2012-19 was a period of active evolution rather than revolution. The paper [21] that we have already identified was written in 2016 and comprehensively describes the range of applications the Oxford group were in contact with.

The beginning of this period saw the establishment of D-RisQ[7] a spin-out of QinetiQ including most of the team we worked with there, and the end of it saw the final spin-out of Cocotec[8] from Oxford. D-RisQ's product ModelWorks is looking to broaden the previous work of QinetiQ in translating systems of state machines described as Statecharts into CSPM. ModelWorks's applications include the automotive industry [69, 7] and an autonomous underwater vehicle [18] for carrying out underwater engineering tasks: Figure 2.

Cocotec manages FDR4's academic licensing and commercial sales: see `https://cocotec.io/fdr/`. We will describe more of this company's work in the next section.

D-RisQ collaborated with Oxford, Draper and many others in the HACMS project as reported in [21, 19, 44] and continues to work in autonomous vehicles, in particular on safety "bubbles" or limiters for the freedom of AI-based control. It is clear that use of FDR in connection with verifying autonomous systems is becoming increasingly popular. Similar work is progressing at the University of York with the tools RoboChart involving FDR analysis, itself with applications such as a bio-security project described in the following abstract from a letter from Augusto Sampaio, another former student of Tony and leader of the thriving research in CSP, FDR and other formal methods in North-East Brazil

> I have been working on a project with Ana Cavalcanti and Jim Woodcock, and their group at York, on the semantics and verification of design and simulation of robotic systems. This also entails the translation of a UML-like graphical notation into CSP and verification using FDR. Together with the Laboratory of Immunopathology (LIKA-UFPE) we are currently applying this in the domain of bio surveillance. In one of the applications (in the context of public health) A drone carries a camera to locate schistosomiasis transmission vectors and has a kind of mechanical arm (or claw) to capture the snails, in rural areas that make access to public agents difficult. This is ongoing work.

See Figure **??**. The Brazilian group (Augusto is based at UFPE, Recife) also has projects in more traditional areas such as (ordinary) aerospace.

[7]`https://drisq.com`
[8]`https://cocotec.io`

7 The future: for CSP and beyond (2019-)

The reason we have highlighted the year 2019 is that it marked the full spin-out from Oxford of Cocotec. It is led by Tom Gibson-Robinson and Philippa Hopcroft. Cocotec was spun out for the Coco Platform: the integration of a new language *Coco* designed for event-driven software and a model checker Cosmos designed to check Coco programs via translation to a further, intermediate language. The Coco Platform generates executable code from the Coco description.

Cosmos uses a process model which, though based on state machines, is significantly different from that used by FDR for CSP. The intermediate language used by the Coco Platform is without some of the features that make CSP hard to model, and adds support for others such as imperative state, which are important for the creation of real event-driven software. These changes mean that it is no longer necessary to base the implementation on supercombinators, though CSP models, abstraction and refinement are still core. Coco itself is a modern imperative programming language with support for event-driven programs based on state machines.

By the choice of design language and handling tree-like topologies using compositional verification, the Coco Platform is able to achieve near infinite scalability in the number of parallel state machines operating.

Coco is designed to be used directly by engineers or be a target for translation from dialects of UML and other languages for model-based architectures. Large-scale trials have been completed successfully with industrial partners. Each component is checked in the context of the interfaces that its neighbours provide, ensuring that it in turn provides the interface it is required to offer, and that pathologies such as deadlock and non-responsiveness do not occur.

By cutting down from the enormous generality of CSP to an architectural template that corresponds to how many engineers design event-driven systems, and finding a specification structure that can be resolved without state explosion, the creators of the Coco Platform have hit a real sweet spot. In the enormous multi-dimensional space of system development they have identified an industrially significant domain where the theories and ideas developed for CSP can be customised and turned into a potentially important tool for industrial use.

That is not to say that ideas for using and developing FDR itself have dried up. There has been a major programme of developing SAT and SMT techniques based on over-approximating state spaces [1, 2], rooted in collecting and adding to local information about interactions between components, which unlike the earlier attempts at using SAT have been very successful.

We see a potential use for both FDR and the Coco platform in the currently fashionable world of blockchain [67], which is itself a fascinating model of concurrency. Blockchain actors have complex event-driven behaviours, much of which seems susceptible to the development environment provided by the Coco platform. More interestingly to the CSP community is the possibility of using CSP and FDR to model the protocols that govern consensus mechanisms and similar and their effects on the developments of blockchains. It is clear that these models will mix

- the characteristics of complex linked structures composed of cells created from a notional "heap" and so will use symmetry reduction;
- symbolic representations of operations such as hashing and encryption, as in the usual CSP cryptoprotocol models [38, 37]
- careful modelling of the "bad guy" or attacker, a particular strength of CSP.

Any need to add further models directly implemented in FDR has been eliminated by the strange discovery of model shifting [41], a technique using priority that implements arbitrary CSP in terms of the simplest ones. So for example this makes it possible to implement refinement checking over the Timed Failures model and thus do compositional reasoning over Timed CSP with FDR. The overheads of model shifting are generally small.

8 Reflections on FDR

We now understand that by throwing huge computer resources at FDR we can get it to resolve huge problems measured in terms of state space, at least over a large part of what it does. What is new is the guilt factor: doing this consumes energy so it no longer seems appropriate to do this just to prove it can be done, for example on supercomputers far larger than the clusters referred to earlier. However, in honour of this paper, Bill worked with some collaborators in China to run a variety of large checks on a cluster there, using 64 32-core, 2.5GHz Xeon processors with 16TB of RAM, so essentially twice the size of EC2 as reported in [20]. We will report these results in a technical report, but one example is we were able to verify a 2T state version of the bakery algorithm on this machine in 7 hours and 13TB of memory.

Perhaps the most noticeable phenomenon in supercomputing in the 6 years since 2014 has been the Charge of the AI Brigade (i.e. GPUs).

We might note that the advance in capability in the 2014 cluster check was (assuming the traditional 18 month doubling) well ahead of the Moore's law curve implied by the 1992 comparator. Note that between [] in 1994 and [] in 2014, the explicit state capability of FDR increased by about one million: a factor of 1000 came from parallelisation, the rest from clock speed and better algorithms. It seems unlikely that much larger clusters of the sorts of cores that FDR expects to run on will be remotely common in future: if there are genuine requirements for really large checks in future then it may be necessary to adapt FDR and its algorithms to work on GPU architectures, as some researchers have attempted with other model checking paradigms [5, 6] without tremendous success. We are not optimistic that this will be particularly successful for FDR either. Unfortunately it seems unlikely that model checking will drive the development of computing platforms in the way that machine learning is doing.

The improvements to FDR and the advance in computing power have made an enormous difference to the complexity of systems that can be modelled and analysed by the tool. But the state explosion problem has not gone away: in the common case where the number of states grows exponentially, one will inevitably reach a point where it is necessary to throw so many extra resources at the problem to get one more increment that even if one can get them it does not seem worth it. In most such cases, to verify a realistic sized system (as opposed to just an interestingly large one) one will need an effective way of countering state explosion.

Another issue to this is the challenge of modelling successfully in CSP for regular system designers, especially under the strictures of handling the state explosion issue discussed above. Despite our hopes and efforts in design and education over the years, this is a huge blockage. Tools such as SVA, ModelWorks and Casper (all of which integrate FDR), and the Coco Platform (integrating Cosmos) solve this problem in their own domains by offering pre-digests of the sorts of systems these tools can represent, creating just the right models and taking advantage of the character of the systems they analyse to choose the right modelling approach. They represent the best — perhaps only realistic — approach to mass adoption.

Of course we believe that CSP theory, FDR and perhaps other tools which like Cosmos are based on that theory have a big place in the future of verification. Experience shows that this is most likely to work on systems with some of the following characteristics:

- Multi-threaded with complex interactions
- Involved but boundable combinatorics
- Interacting state machines

It is not likely to work well in verifying computations over types such as the integers or reals, where exact computations must be checked over types too large to enumerate. The current state of the art suggests that SAT and SMT solvers are best for such systems. We would like to see practical and sound ways of combining the two.

The reader might have noticed the theme of autonomous vehicles running through a number of recent applications, including two of our pictures. Since these frequently have strong safety-critical questions to answer, the use of formal methods such as CSP/FDR that can show unsafe behaviour impossible is most appropriate, but seems in tension with the popular vision of such vehicles as being controlled by AI/machine learning. It makes perfect sense for the parameters within which AI is permitted to act being constrained by a formally proved framework, in exactly the same way a human driver might be.

In this paper we have concentrated on the FDR style of model checking and its applications. Model checking is an enormous subject, to the extent that it has recently acquired a substantial handbook [15]. Bill contributed to the chapter on model checking in process algebra [16], where the approach we have detailed here is contrasted against ones based on modal logic and bisimulation-like equivalences, but some of the other chapters are also relevant to the issues we have raised here.

Looking at that book, we can reflect on the decisions we made about FDR. We initially concentrated on refinement over the well-known models of CSP and this, as it turned out, could be done very efficiently, usually linear in the state space of the process being checked, leading to what seemed to us (i.e. the team behind FDR) impressive applications. Because FDR was so closely based on CSP semantics, it never seemed natural to incorporate specification and semantic approaches which did not fit easily with CSP: these included fairness and bisimulation-based proof techniques. The latter has never been missed in practical circumstances, because the range of properties achievable using refinement has proved adequate for all practical challenges where infinitary patterns of behaviour were *not* involved. In practical examples based on infinite traces, the lack of support for reasoning about fairness has been an issue. For mixing fairness with the idea of failures was never satisfactory, as failures contain enough information to make one think was relevant to specifiying what is fair or not, but not actually enough. To illustrate this, the concept of *slow abstraction* introduced, motivated by an industrial example, in [62] shows how FDR can reason about fairness-related concept of "unstable failures", namely ones that arise over divergences where τ actions happen sufficiently fairly to show temporary acceptance/refusal sets, is anything but compositional: it only makes sense at the top level. Certainly this is somewhere where CSP theory made us very late. FDR has, despite experimental introduction, never fully embraced the rich trace properties such as fairness that can emerge from Büchi, Streett and similar automata [35], as perhaps it should have done.

In this paper we have already alluded to how other model checking techniques such as SAT/SMT checking and CEGAR of CSP-style properties relate to FDR, and how continuous-time reasoning relates to it. The integration with other approaches, such as SAT/SMT analysis of detailed calculation alluded to above, and perhaps probabilistic model checking [36], seem a worthwhile future project. Historically, compatibility with CSP theory has been important to FDR development rather than our own implementation of semantic concepts that do not fit so easily with CSP but have been successful elsewhere.

Acknowledgements

The preparation of this paper has been assisted by many of our collaborators and the major users of FDR. These include Philippa Hopcroft and Tom Gibson-Robinson, Zhiqiang Yang and Liu Han who experimented with the 2048 core cluster, Augusto Sampaio and Colin O'Halloran.

References

[1] Pedro Antonino, Thomas Gibson-Robinson, and A.W. Roscoe. Efficient deadlock-freedom checking using local analysis and sat solving. In *International Conference on Integrated Formal Methods*, pages 345–360. Springer, 2016.

[2] Pedro Antonino, Thomas Gibson-Robinson, and A.W. Roscoe. The automatic detection of token structures and invariants using sat checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 249–265. Springer, 2017.

[3] Philip Armstrong, Michael Goldsmith, Gavin Lowe, Joël Ouaknine, Hristina Palikareva, A.W. Roscoe, and James Worrell. Recent developments in FDR. In *International Conference on Computer Aided Verification*, pages 699–704. Springer, 2012.

[4] Philip Armstrong, Gavin Lowe, Joël Ouaknine, and A.W. Roscoe. Model checking timed CSP. In Andrei Voronkov and Margarita Korovina, editors, *HOWARD-60. A Festschrift on the Occasion of Howard Barringer's 60th Birthday*, pages 13–33. EasyChair, 2014.

[5] Jiří Barnat, Petr Bauch, Luboš Brim, and Milan Češka. Designing fast ltl model checking algorithms for many-core gpus. *Journal of Parallel and Distributed Computing*, 72(9):1083–1097, 2012.

[6] Jiří Barnat, Luboš Brim, and Milan Češka. Divine-cuda-a tool for gpu accelerated ltl model checking. *arXiv preprint arXiv:0912.2555*, 2009.

[7] John Botham, Gunwant Dhadyalla, Antony Powell, Peter Miller, Olivier Haas, David McGeoch, Arun Chakrapani Rao, Colin O'Halloran, Jaroslaw Kiec, Asif Farooq, et al. Picassos–practical applications of automated formal methods to safety related automotive systems. Technical report, SAE Technical Paper, 2017.

[8] Guy H Broadfoot and PJ Hopcroft. Introducing formal methods into industry using cleanroom and CSP. *Dedicated Systems Magazine*, 2005.

[9] Neil A Brock and David M Jackson. Formal verification of a fault tolerant computer. In *[1992] Proceedings IEEE/AIAA 11th Digital Avionics Systems Conference*, pages 132–137. IEEE, 1992.

[10] Stephen D. Brookes. *A model for communicating sequential processes.* PhD thesis, Oxford, 1983.

[11] Stephen D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3):560–599, 1984.

[12] Stephen D. Brookes and A.W. Roscoe. An improved failures model for communicating processes. In *International Conference on Concurrency*, pages 281–305. Springer, 1984.

[13] Stephen D. Brookes and A.W. Roscoe. Deadlock analysis in networks of communicating processes. *Distributed Computing*, 4(4):209–230, 1991.

[14] Stephen D. Brookes, A.W. Roscoe, and D.J. Walker. An operational semantics for CSP. Technical report, Oxford University Computing Laboratory, 1986.

[15] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of model checking*, volume 10. Springer, 2018.

[16] Rance Cleaveland, A.W. Roscoe, and Scott A Smolka. Process algebra and model checking. In *Handbook of Model Checking*, pages 1149–1195. Springer, 2018.

[17] Edsger Wybe Dijkstra. *A discipline of programming*. Prentice-Hall Englewood Cliffs, 1976.

[18] Simon David Foster, Yakoub Nemouchi, Colin O'Halloran, Nick Tudor, and Karen Stephenson. Formal model-based assurance cases in isabelle/sacm: An autonomous underwater vehicle case study. In *Formal Methods in Software Engineering (FormaliSE 2020): Proceedings of the 8th International Conference*. ACM, 2020.

[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*, 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(2):149–167, 2016.

[21] Thomas Gibson-Robinson, Guy Broadfoot, Gustavo Carvalho, Philippa Hopcroft, Gavin Lowe, Sidney Nogueira, Colin O'Halloran, and Augusto Sampaio. FDR: from theory to industrial application. In *Concurrency, Security, and Puzzles*, pages 65–87. Springer, 2017.

[22] Thomas Gibson-Robinson, Henri Hansen, A.W. Roscoe, and Xu Wang. Practical partial order reduction for CSP. In *NASA Formal Methods*, pages 188–203. Springer, 2015.

[23] Thomas Gibson-Robinson and Gavin Lowe. Symmetry reduction in CSP model checking. *International Journal on Software Tools for Technology Transfer*, 21(5):567–605, 2019.

[24] Michael Goldsmith and Jeremy Martin. The parallelisation of FDR. In *Proceedings of the Workshop on Parallel and Distributed Model Checking*, 2002.

[25] Constance L Heitmeyer, Bruce G Labaw, and Ralph D Jeffords. A benchmark for comparing different approaches for specifying and verifying real-time systems. Technical report, Naval Research Lab Washington DC, 1993.

[26] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[27] C.A.R. Hoare. A model for communicating sequential process. 1980.

[28] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[29] C.A.R. Hoare, Stephen D. Brookes, and A.W. Roscoe. *A theory of communicating sequential processes*. Oxford University Computing Laboratory, Programming Research Group, 1981.

[30] C.A.R. Hoare, Ian J. Hayes, He Jifeng, Carol C Morgan, A.W. Roscoe, Jeff W. Sanders, I Holm Sorensen, J. Michael Spivey, and Bernard A Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987.

[31] C.A.R. Hoare and He Jifeng. *Unifying theories of programming*. Prentice Hall Englewood Cliffs, 1998.

[32] John Hughes. *Graph reduction with super-combinators*. Oxford University. Computing Laboratory. Programming Research Group, 1982.

[33] C.B. Jones and A.W. Roscoe. Insight, inspiration and collaboration. In *Reflections on the Work of C.A.R. Hoare*, pages 1–32. Springer, 2010.

[34] 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. ACM, 1983.

[35] Nils Klarlund. Progress measures for complementation omega-automata with applications to temporal logic. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 358–367. IEEE Computer Society, 1991.

[36] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002.

[37] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer, 1996.

[38] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of computer security*, 6(1-2):53–84, 1998.

[39] JMR Martin, S Jassim, et al. A tool for proving deadlock freedom. In *Proc. of the 20th World Occam and Transputer User Group Technical Meeting*, 1997.

[40] Tomasz Mazur and Gavin Lowe. CSP-based counter abstraction for systems with node identifiers. *Science of Computer Programming*, 81:3–52, 2014.

[41] David Mestel and A.W. Roscoe. Translating between models of concurrency. *Acta Informatica*, pages 1–36, 2020.

[42] R Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[43] Robin Milner. *A calculus of communicating systems*. Springer-Verlag, 1980.

[44] Colin O'Halloran, Thomas Gibson-Robinson, and Neil Brock. Verifying cyber attack properties. *Science of Computer Programming*, 148:3–25, 2017.

[45] 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.

[46] Colin O'Halloran. Assessing safety critical COTS systems. In *Towards System Safety*, pages 65–74. Springer, 1999.

[47] Hristina Palikareva. *Techniques and tools for the verification of concurrent systems*. PhD thesis, Oxford, 2012.

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

[49] Doron Peled. Ten years of partial order reduction. In *International Conference on Computer Aided Verification*, pages 17–28. Springer, 1998.

[50] Jan Peleska. Test automation for safety-critical systems: Industrial application and future developments. In *International Symposium of Formal Methods Europe*, pages 39–59. Springer, 1996.

[51] Jan Peleska. Applied formal methods–from CSP to executable hybrid specifications. In *Communicating Sequential Processes. The First 25 Years*, pages 293–320. Springer, 2005.

[52] Stacy J Prowell and Jesse H Poore. Sequence-based software specification of deterministic systems. *Software: Practice and Experience*, 28(3):329–345, 1998.

[53] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theor. Comput. Sci.*, 58(1-3):249–261, June 1988.

[54] A.W. Roscoe. *A mathematical theory of communicating processes.* PhD thesis, University of Oxford, 1982.

[55] 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.

[56] A.W. Roscoe. Model-checking CSP. In *A classical mind*, pages 353–378. Prentice Hall International (UK) Ltd., 1994.

[57] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[58] A.W. Roscoe. Revivals, stuckness and the hierarchy of CSP models. *The Journal of Logic and Algebraic Programming*, 78(3):163–190, 2009.

[59] A.W. Roscoe. *Understanding Concurrent Systems.* Texts in Computer Science. Springer, 2010.

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

[61] A.W. Roscoe and C.A.R. Hoare. The laws of occam programming. *Theoretical Computer Science*, 60(2):177–229, 1988.

[62] A.W. Roscoe and Philippa J. Hopcroft. Slow abstraction via priority. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods*, pages 326–345. Springer-Verlag, Berlin, Heidelberg, 2013.

[63] A.W. Roscoe and Zhenzhong Wu. Verifying Statemate Statecharts using CSP and FDR. In *International Conference on Formal Engineering Methods*, pages 324–341. Springer, 2006.

[64] Peter Ryan, Steve A Schneider, Michael Goldsmith, Gavin Lowe, and A.W. Roscoe. *The modelling and analysis of security protocols: the CSP approach.* Addison-Wesley Professional, 2001.

[65] JB Scattergood. *Tools for CSP and Timed CSP.* PhD thesis, DPhil thesis, Oxford University, 1997.

[66] Joseph E Stoy. *Denotational semantics: the Scott-Strachey approach to programming language theory.* MIT press, 1977.

[67] Melanie Swan. *Blockchain: Blueprint for a new economy.* O'Reilly Media, Inc., 2015.

[68] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[69] NJ Tudor and J Botham. Proving properties of automotive systems of systems under ISO 26262 using automated formal methods. 2014.

[70] Oxford University. Automated software design and verification. `https://impact.ref.ac.uk/casestudies/CaseStudy.aspx?Id=4907`.

[71] Oxford University. Automated verification and validation for defence, aerospace and automated embedded software. `https://impact.ref.ac.uk/casestudies/CaseStudy.aspx?Id=19859`.

[72] Antti Valmari. A stubborn attack on state explosion. In *International Conference on Computer Aided Verification*, pages 156–165. Springer, 1990.