Modelling and analysing a simple blockchain using CSP and FDR

A.W. Roscoe and Jonathan Lawrence

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

Abstract

We present a CSP (Communicating Sequential Processes) model of a simple blockchain system and its analysis using the refinement checking tool FDR (Failures Divergences Refinement). We build on ideas developed for the successful models of cryptographic protocols that model adversary behaviours symbolically, and introduce new ways of recording and communicating large and implicitly recursive data values, namely blocks that include hashes of other blocks. We propose a model for malign agents that is similar to the intruder model for protocols. We illustrate how blockchains work and how bad decisions can lead to pathologies such as forking, diagnosed by tracking the evolution of the blockchain. We discuss potential ways of using CSP and FDR to model other aspects of blockchains and decentralised systems.

1 Introduction

In recent years, blockchain technology [10, 2, 20] has gained significant attention due to its decentralised, secure, and (eventually) immutable nature. It has applications ranging from cryptocurrencies to distributed ledger technologies. However, modelling and verifying blockchain systems, especially their behaviour under various conditions, poses several challenges. One powerful tool for such formal modelling is Communicating Sequential Processes (CSP) [8, 11, 12], a formal language used to model and verify concurrent systems.

This paper demonstrates the application of CSP in modelling a generic blockchain system. Specifically, we utilise CSP to model the structure and behaviour of a blockchain, focusing on the coding of several crucial aspects: block storage and hashing mechanisms, the behaviour of good versus bad agents, and the use of watchdog-based specifications.

Furthermore, we address the challenge of state space and alphabet size reduction to make the model tractable for model checking tools such as FDR (Failures-Divergence Refinement) [18, 11, 12, 4] Our main innovation is a new and efficient model for managing cryptographic hashing symbolically in such a way that it achieves compression in a similar way as the real function – essential for blockchain.

Through this demonstration, we explore how CSP can provide insights into blockchain functionality and how optimization techniques can make such models computationally feasible.

We also explore techniques for introducing specifications on complex data structures using *watchdogs* [6], namely processes that run alongside a system S without affecting their behaviour but raise an error flag if S does something wrong. The style of watchdog we use solves seemingly intractable state space issues.

While there is considerable merit in modelling distributed and decentralised systems in CSP without worrying about the feasibility of verification on tools like FDR, our aim in this paper is to create models that are tractable for FDR, at least in small instantiations. This allows us to verify that the models really do reflect the behaviour we want, to verify properties and to see how emergent behaviours arise.

This paper develops a theme discussed briefly in [16], which we found needed further development.

2 Background

2.1 Blockchain

The term *Blockchain* means different things to different people, even though they are all talking about the same thing. Superficially, a blockchain has a root or Genesis block which is its beginning, and from there a growing chain of blocks – namely structures with multiple records and transactions collected into each block – where each block contains a strong record of its predecessor via a cryptographic *hash link*. The predecessor of each block, and hence all the predecessors back through a finite number of steps to the Genesis blocks, are completely determined via an inductive argument from this link.

The exciting things about blockchains to a computer scientist are how the next, or successor, block is formed, and the fact that blockchains are typically formed by a decentralised community of independent agents with non-trivial trust issues and where some agents may be downright bad and hellbent on upsetting the coherence of the chain. Different trust relationships and assumptions about this *consensus* process give rise to different sorts of blockchains but the fundamental aims are the same. Given an agreed set of assumptions about a trust model, the consensus protocol is supposed always to generate a unique successor for each block B in the chain. It should never stop forming unless there is general agreement about this. And there must never be two plausible and seemingly permanent successors of any block – there should never be a *fork* in a chain. Two good agents must not believe in inconsistent versions of the chain.

What we study in this paper is how generic models of consensus interact with chain development, by creating what we believe is the first blockchain model cast in process algebra that can be deeply queried by verification technology.

Thus a blockchain is both a simple data structure and a variety of approaches to building and securing this structure in different circumstances and different trust models. The rationales for creating blockchains are widely varying.

1. Public blockchains [10, 2, 20] – the most challenging sorts to build and justify – solve many of the problems of building a decentralised currency, and simultaneously depend on these to motivate participation. As you might expect, a public blockchain is one where it is open to anyone to be a participating agent, which makes quantifying trust a challenge. In this context they are often called *distributed ledgers*.

2. A blockchain is a very strong guarantor of the integrity of the records it contains, which gives these a strong role, at least potentially, in guaranteeing the integrity of public records in many different areas [19, 3].

3. Contracts – so called Smart Contracts – can be stored on and executed by a blockchain [2] , making them the responsibility of a decentralised community rather than a designated individual.

4. Even though cryptocurrency has a reputation for being an unregulated Wild West of finance, blockchains are actually a wonderful platform for defining and agreeing on regulations and ensuring transparency of process in their implementation.

It is because of this context that our blockchain models typically have a mixture of good agents, who follow all the rules, and bad ones who are equipped to follow rules but can actually do anything subject to a reasonable expectation that they could really do it. Thus they are typically constrained by their inability to break encryptions or guess things correctly without evidence, and by our sometimes writing constraints on them that prevent them saying things that good agents would never believe.

Blockchains ensure authentication and integrity, not privacy. However they can be combined with other privacy solutions to achieve the combination of integrity and privacy.

So, as well as being a data structure and class of technical approaches to building and maintaining them, blockchains can be a great business opportunity, a way of building alternative way of financial settlement that avoids hated (by some) regulatory restrictions, a way of implementing those same restrictions, or a core of public record keeping.

In this paper we concentrate on how agents interact to create new blocks, modelling that formally so we can look for interesting properties of choices of consensus and how bad agents are modelled. The choice of application is irrelevant to us.

2.2 CSP

CSP (Communicating Sequential Processes) is a formalism used for reasoning about concurrent and distributed systems. These systems and their components are captured as *processes*. In this section, we provide a very brief summary of CSP with a focus on the language constructs we need to make the paper self-contained; see [8, 11, 12] for detailed introductions to CSP.

A CSP process can perform instantaneous events and it can also synchronise with its environment on them; this environment can comprise other processes or some notional external observer. Using these primitives one can model and analyse different patterns of interaction.

The main language constructs that we use in this paper are[1] :

- The processes `STOP`, and `SKIP` respectively do nothing, and terminate immediately with the signal ✓. `RUN(A)` and `CHAOS(A)` can each perform any sequence of events from `A`, but while `RUN(A)` always offers the environment every member of `A`, `CHAOS(A)` can non-deterministically choose to offer just those members of `A` it selects, including none at all.

- `a -> P` *prefixes* `P` with the single communication `a` which belongs to the set Σ of normal visible communications. Similarly `[] x : A @ x -> P(x)` (replicated external choice) offers a deterministic choice over `A` and then behaves accordingly.

- CSP has several *choice* operators. `P [] Q` and `P |~| Q` respectively offer the environment the first visible events of `P` and `Q`, and make an internal decision via τ actions whether to behave like `P` or `Q`.

- `P \ X` (hiding) behaves like `P` except that all actions in `X` become (internal and invisible) τs.

- `P [[ R ]]` (renaming) behaves like `P` except that whenever `P` performs an action `a`, the *renamed* process must perform some `b` that is related to `a` under the relation `R`. `R` is specified using the CSP_M mapping syntax.

- The CSP_M mapping syntax, `[[from <- to | gen ]]` permits general renaming(s) of one or more events to others. Note the counterintuitive direction of the arrow. The optional generator(s) `gen` take the form `v <- S` (defining variable `v` that ranges over set `S`), which allows renaming of families of events. Mappings may be one-one (the usual case), many-one or one-many, enabling relational renamings.

- `P [| A |] Q` is a *parallel* operator under which `P` and `Q` act independently except that they have to agree (i.e. synchronise or handshake) on all communications in `A`. A number of other parallel operators can be defined in terms of this, including `P ||| Q = P [|{}|] Q` in which no synchronisation happens at all.

[1] We use exclusively the machine-readable ASCII version of CSP syntax (CSP_M), rather than the "blackboard" syntax and symbols often used in books and papers.

- `P ; Q` (sequential composition) behaves like `P` until is successfully terminates at which point `Q` takes over.

CSP has several styles of semantics that can be shown to be appropriately consistent with one another. In this paper, we are concerned with *behavioural* semantics, namely, the meaning, or semantics, of a CSP process is given by the externally visible communications it performs. The best known behavioural models of CSP are based on the following types of observation: *Traces* are sequences of visible communications a process can perform. *Failures* are combinations (s,X) of a finite trace s and a set of actions that the process can refuse in a *stable* state reachable on s. A state is stable if it cannot perform τ. *Divergences* are traces after which the process can perform an infinite uninterrupted sequence of τ actions, in other words diverge. The models are then:

- `T` in which a process is identified with its set of finite traces;
- `F` in which it is modelled by its (stable) failures and finite traces;
- `FD` in which it is modelled by its sets of failures and divergences, both extended by all extensions of divergences: it is *divergence strict*.

2.3 FDR

FDR [18, 11, 12, 4] is a model checker that analyses refinement between finite-state processes defined in CSP. In this paper, we use its latest version: FDR4.[2]

CSP$_M$ is the machine-readable version of CSP, extended with a functional programming language, and is the input language for FDR. It can be used to define complex network of process and data operations succinctly. CSP and FDR have been applied in many domains; a survey of important practical applications can be found in [1]. Perhaps the best-known application is the Security Protocol checker Casper [9] which, given an abstract representation of a cryptographic protocol and some security objectives for it, generates a CSP script which checks to see if the objectives are met. Similarly, compilers have been written from other notations to CSP such as Statecharts [7] and shared-variable programs (see Chapters 18 and 19 of [12]); in a sense, these applications use CSP to formally capture the semantics of these source notations. We hope it may be possible to do the same for blockchain constructions.

FDR verifies refinements of the form `Spec [X= Impl`, where `Spec` is a process representing a specification in one of the standard CSP models `X`, usually traces, stable failures or failures-divergences. `Impl` is a CSP representation of the system being verified. To check whether a process `Impl` satisfies a particular property, `Spec` is constructed to represent the most general process (in the relevant model) exhibiting the required property, known as the *characteristic process* for that property. FDR checks typically scale better in the size of `Impl` than the size of `Spec`, as the latter has to be normalised as part of the refinement checking process.

FDR supports several techniques for tackling the state explosion problem, including hierarchical compression and symmetry reduction [5]. The algorithms underpinning FDR are set out in [11, 12, 4, 15].

3 Blockchain CSP model

Any CSP model of a real system represents an abstraction of reality: something that deliberately captures some details and forgets others. It is a good model if it stands alone and captures useful detail.

[2] Available at https://cocotec.io/fdr/.

Here we aim to capture the interactions leading to block selection without going into details of actual cryptography or transactions.

In this section we develop our model. We first explain our abstraction of blocks and the associated hashing and storage mechanism, then the behaviour of both good and bad agents, including communications between them. Finally we specify some properties that we expect to hold in the overall system and outline how we use FDR to verify these.

Our model is available to download and consists of three CSP files to analyse with FDR.[3] They `include` each other as needed. The first implements our novel hashing model; the second puts together the agents that use this model and forms the complete implementation; the third introduces our main specification that is run as a watchdog: literally it watches the system and barks when things go wrong. All the files have parameters that can be adjusted. The first two files are described in this section, the final one in the next.

3.1 Block storage and hashing

This section explains the mechanics of block storage and hashing. This script simulates a storage mechanism where each agent has its own storage of blocks. The key features of this model include:

- **Hashing Process:** A simple symbolic hash function is used to map blocks to hash values. The hash values are stored and can be used for lookups during consensus verification.
- **Agent-specific Block Storage:** Each agent has a separate store for the blocks they have authored. This storage simulates the behaviour of a blockchain where each agent can add new blocks or verify the integrity of existing ones. While it seems to imply that the implementations envisaged will also have separate stores for each agent's blocks, this is not so. Rather we imagine that the blocks will in reality be well replicated with each cryptographically signed by its author. The separate storage in fact partitions the set of blocks by who has signed them.
- **Block Lookup:** Agents can query their storage to retrieve blocks based on their hash, simulating the blockchain's lookup functionality.
- **Symbolic signature:** Hashes take the form (a,k) for the agent a who has created and signed the block whose hash this is, and k indicating that this is the kth distinct hash value created by a. In our model there are few distinctions, except in representation, between a signed block and its hash, and all agents must respect the cryptographic boundaries that hashing and signature impose *on the symbols* even though the cryptography is not actually implemented.

A real block contains a lot of data. CSP models designed for FDR simply cannot deal with such ranges any more than they handle ranges of identities, keys and messages in cryptographic protocols. We therefore severely restrict the data that gets handled in our models. Unless, for the properties we are analysing, we need to compute with actual transaction data — for example following ledger properties — we tend to cut the carried data down to a bare minimum and appeal to data independence arguments. In looking at the structures of chains and attempted chains that are created we can usually cut the carried data down to a single value, thus eliminating this as a cause of combinatorial explosion. That is the world we investigate in the models introduced in this paper: therefore the CSP script makes provision for variety in block contents but we have not so far needed to consider cases where more than two possible contents existed. This phenomenon will be familiar to anyone who has use data independence.

[3] Available at https://blockchain.univ.ox.ac.uk/research-papers/modelling-and-analysing-a-simple-blockchain-using-csp-and-fdr/.

Our blocks thus each consist of its author agent ID, this contents holder, the integer level of the block and the hash of the predecessor block. We could extend this by other fields if we want to investigate their properties — for example ones supporting consensus mechanisms such as bids to participate, and security mechanisms such PKIs or hooks [17].

```
M=2  -- Block contents
blockcontents = {1..M}   -- abstraction from block contents
N=3  -- agents other than genesis holder
agents = {1..N}
MaxLev = 5  -- number of links back to Genesis
levels = {0..MaxLev}

-- a block takes one of two forms
datatype Blocks = Genesisblock | Block.agents.levels.blockcontents.hashvals
```

The level means the number of hash links required to reach the Genesis Block. It is in fact redundant information as it can be discovered by tracking back, but it provides a nice feature for real chains and a natural source of consistency checks. In the simple model we present here, we make no attempt to model a real consensus algorithm. Rather we model a simple case where the legitimate creator of each block is a universally agreed function of the chain to date. There are potential variations within this: for example, will anyone accept a block proposed by the wrong agent? A block takes one of two forms: it is either the `GenesisBlock`, or it is a regular block:

```
datatype Blocks = GenesisBlock | Block.agents.levels.blockcontents.hashvals

author(GenesisBlock) = 0
author(Block.a._._._) = a

contents(Block._._.c._) = c

level(GenesisBlock) = 0
level(Block._.l._._) = l

link(Block._._._.h) = h
```

We also define some projection functions to extract specific fields from block values.

Quite distinct from usual model: we store values by a small range of symbols per agent, and these are treated symbolically as signed. We discuss how this is enforced when describing how agents are modelled in the next section. In the real world, hashes compress the objects they are applied to and yet, paradoxically, a one-to-one relationship is maintained between them: so it will be in our symbolic world. In the real world this is achieved through complex one-way functions and combinatorics making the mathematically possible computationally infeasible. In symbolic worlds — whether that of the cryptographic protocol models or our new one — we need to build models of agents in such a way that they respect by programming what we assert to be impossible in the real world. Thus we treat our model of a signed hash — simply a pair (a,x) where a is the agent who has created and signed it and x is the small integer index that says this is the xth hash that a has performed — as a unit that cannot be created other than faithfully by a and cannot be pulled apart and modified. However it is instantly recognisable in our model as a hash created and signed by a.

```
HL = 3           -- hashes per non-genesis agent
```

```
HashLimit(a) = if a==0 then 1 else HL  -- number of blocks per identity

-- the set of distinct hashes we allow each agent a to create:
hashvalsl(a) = {(a,i) | i <- {1..HashLimit(a)}}

hashvals = {v | a <- AAgents, v <- hashvalsl(a)}
```

We compress the size of the alphabet by having agents pass the signed hashes of blocks around rather than the signed blocks that would be sent in practice. In our model, hashes are used only to provide hash links rather than their many other cryptographic functions. These hash links are designed to provide uniqueness, and anyone who has published one is happy for anyone know the block it was derived from, so preimage resistance is not an important property in this context – our model depends on the channel `readbyhash` and so could not coexist with dependence on preimage resistance.

In replacing communicated blocks by their hashes we do not want to deny anyone information so at least for this type of hash we provide a hashing oracle: a Hasher that both creates new hashes and looks up old ones. When asked to create a new signed hash it first determines if this agent has hashed-and-signed this value before, to determine if what it should really do is return the same value as last time. The Hasher applies the correct agent's signature and calculates these new hashes, and allows anyone to look up an existing one. Thus, if I am sent a message containing such a symbolic signed hash, I can look it up. This justifies the "send by hash" model we adopt.

In passing and reflecting on this, it is obvious that the reason that blockchains use hash links in blocks is that using the actual block itself would be impractical through storage requirements. The crucial quality of hashing used here is the 1-1 mapping with compression. The signature that our model creates is perhaps better viewed as a signature on the hashed block. Good agents will never sign a block that purports to be created by anyone else. Obviously a bad agent might try this, but it can only sign its own name and no good agent will ever accept as genuine anything it has signed that claims to have been authored by someone else. These things limit the plausible attacks that bad agents might try and also allow us to make our model more efficient. We simply limit the Hasher to signing with the block author's signature. In the spirit of many model-checking approaches, to reduce the state space we limit the number of hashes each agent can create per run via a parameter `HL`. The hashes that agent `A` can create are thus the HL symbolic pairs `{ (A,1), (A,2), ... (A,HL) }`. These are allocated to the first HL distinct values that `A` chooses to hash in order on a first-come-first-serve basis. These are all block values authored by `A`.

Our script contains two models of the `Hasher(A)` process that handles `A`'s hashes. The first is a simple sequential process indexed sequence of up to `HO` distinct block values and (explicitly or implicitly) the result of `A` hashing them.

This representation has many states to compile, and as with many other CSP models for FDR there are advantages in splitting the process into a component for each hash it is permitted to create. In this case it becomes a pipeline of `HL` cells which pass the tasks of lookup and assigning hash values through the cells from 1 to up to `HL`. Cell `j` manages the hash symbol `(A,j)` and binds either `null` or a block `B` to it. It can look up what was hashed to get this, prevent a new symbol being created when `A` tries to hash `B` again, and will allocate its own symbol to a freshly hashed block just when its own symbol is the first not to have been used yet.

If there are K blocks that agent `A` might want to sign and hash, the sequential Hasher has $K!/(K-HL)!$ basic states that need to be compiled. The parallel Hasher has `HL` components, each of which has $K+1$ basic states including its empty one. Because of the way FDR works it is much more efficient to use the latter. The pipelined model of the parallel hashers means that there is more asynchrony than in the sequential one, but if this removed by adding a regulator forcing hash input and output events to

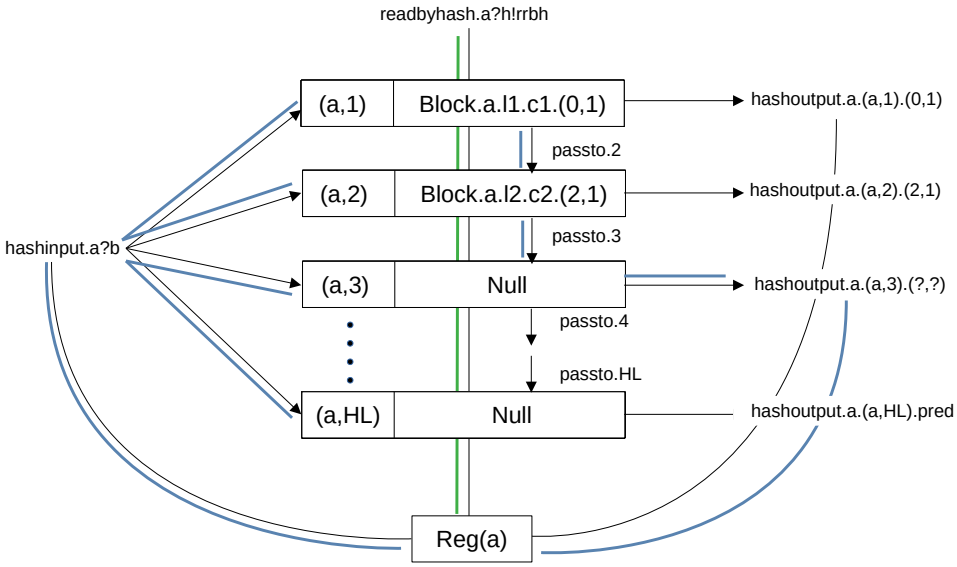

Figure 1: Structure and operation of the distributed hasher for agent `a`.

alternate strictly, it becomes equivalent to the sequential model, and it is this form that we use. It also makes it deterministic and so safe to apply the chase operator which forces through invisible tau actions.

There is a close analogy with the even more drastic improvement gained in the parallel implementation of the cryptographic protocol intruder of [8, 11, 12], right down to the use of the partial order "compression" `chase` that is used to remove calculation-based intermediate states in the parallel implementation justified by (for example) the determinism of the overall process. The fundamental things to realise about our novel symbolic hashing model are firstly that the greater compression it provides — essentially by dereference — is needed for exactly the same reasons that the compression of regular hashing is needed in real-world blockchains. Secondly it is a good model for the standard assumptions about cryptographic hashing just when the programming builds these in, for example ensuring the functions remain injective. Recall that our model builds cryptographic signature into its symbolic representation, simply because we use identifiable symbols for the hashes by respective agents and made all parties (good and bad) treat them as signed objects.

Superficially, blockchains seem to have much in common with traditional linked lists. After all, they are, literally, *linked lists*. There proved to be much less of a connection than we expected. The difference comes from the realisation that conventional linked lists are formed from pointers to the locations where the records are stored, whereas in a blockchain the pointers are to values.

In a blockchain, the locations where blocks are stored are essentially irrelevant, and indeed blocks can be (and usually are) arbitrarily replicated without changing the semantics. Thus, it is legitimate to think of the hasher processes as a representation of a set of values, rather than giving any hint as to how

these values should be stored.

So, while conventional linked lists are typically represented by processes representing locations in store, these processes are not needed to represent blockchains. One thing we expected blockchain models to inherit from linked lists was the need to use symmetry reduction — in linked lists this is because any permutation on the roles of the locations would create a behaviour that was equivalent up to any reasonable definition of correctness.

In our blockchain models we record the state as a set of the blocks that exist, namely have been created and recorded. They will usually have different roles and not be interchangeable. While potentially there may be deep symmetries in some blockchain models, the complex relationships between blocks, hashes and agents made this pointless in ours.

Thus, though blockchain models pose many problems managing the growth of state spaces, it turns out that fact that they are represented by structures of values as opposed to values held in locations does create one important simplification when building CSP models.

3.2 Blockchain agents

As detailed in the previous section, agents communicate their proposals to each other by as signed block hashes against the background of a publicly accessible hasher where these can be looked up. The basic actions of a good agent `A` — which always holds a view of the current chain head and consequently the whole chain from it back to the Genesis block — are as follows:

1. It will communicate its view of the head `h` to another agent `B` via a point-to-point message. This is `currhead.A.B.h`.

2. It can also receive such a message `currhead.B.A.h'`, It then compares `h` and `h'`: if `h'` is seems to be a (not necessarily one-step) legitimate successor of `h`, it adopts `h'` as the head. Otherwise it sticks with `h`.

3. If it is `A`'s turn to create a new block it does so, with level one more than `h`'s, and backlink to `h`. It inserts the new block `B` into the hasher, getting hash `h"` which is then adopted as its head view. Of course it only creates on block per turn.

It will thus be apparent that the main parameter of a good agent, aside from its own name, is its current view of which block is the current head of the chain. In our simple model a good node will only accept new blocks that extend this: in blockchain parlance, a good agent regards its current view of the head as immutable or final. This is why it *traces* each proposed block back to check it is a successor of its current head parameter.

To trace a hash `h'` (as required in 2 above) the good agent looks up the block `B` associated with `h'`, and then traces `B`'s predecessors until it reaches down to the level of `h`. If the result is `h` then `h'` it accepts `h'` as a successor of `h`. The good agents start off with the belief that the head is the Genesis block, whose hash is `(0,1)` — the first and only hash created by a notional and inactive founder agent, and which has no predecessor and level `0`. In our model this is done by a slave process running in parallel with the main process implementing an agent, taking tracing instructions, following these out in the hash database implemented by the hasher, and returning a result.

Our current model uses a simple criterion to test whether a given agent `A` has the right to create the block that follows a given one in the chain. Therefore, if a block seems to have been created by an agent who is really good, then it really has. If no bad agent is allowed to create some block, then no bad one appearing in that place will be believed by any good agent.

A bad agent will, like the intruder in the cryptographic protocol world, be able to do anything it could do with the knowledge that it would have if it were good, and everything it might overhear, but is

not bound by the rules. For example, it can create one of its own signed hashes at any time, but can only hash something it can know.

A. It does not have a view on what the head of the chain is, but can tell anyone that the head is any block hash that has been stored in the hasher, whether by itself or someone else.

B. It listens to others' views of what is head, but ignores this and takes no action since it does not particularly care.

C. It can make up any block `B` it wants to (subject to this being possible for it) and hash it so that `B` or `B`'s hash can be used.

The code that implements the bad agent described above is:

```
Badnode(a) = nodeon.a -> BadnodeOn(a)

BadnodeOn(a) =
      (readbyhash.a?h?Found._ ->
       currhead.a?_:diff(Agents,{a})!h -> BadnodeOn(a))
   [] (currhead.a?_:diff(Agents,{a})?h:myhashes(a) ->                BadnodeOn(a)
   []
       (readbyhash.a?h?Found.b ->
            if level(b)<MaxLev then
             hashinput.a?_:newblocks(a,h,level(b)+1) ->
              hashoutput.a?_?_ -> BadnodeOn(a)
            else BadnodeOn(a))
   []
       currhead?b:diff(Agents,{a})!a?_ -> BadnodeOn(a)     `
```

Modulo the precise details of interactions with the hasher, we expect that a good agent `A` will refine how that same `A` would behave if bad.

We have thus created, in the combination of a set of good and bad agents and the hasher, a decentralised system that is a reasonable abstract model for a simple blockchain.

3.3 Validating the model

We can now test the model by running checks on FDR, but before we do this, we will discuss ways to optimise the model. By this we mean being able to deal with larger and more interesting experiments without losing significant behaviours, especially ones we might not expect. Since we do not really care what bad agents see or think, the core behaviours will be characterised in how the chain looks to the good agents: we need to know how the chain looks and processes in the eyes of the collective good.

Our bad agents can build badly structured blocks — ones that are apparently authored by one agent but signed by another; or ones whose level is not its predecessor's level plus one. They can hash such blocks and use these hashes, but anywhere these are used will not be accepted by a good agent provided it builds in — as we do — natural checks. We can significantly reduce both the compilation and checking complexity of running our model on FDR by eliminating such blocks from consideration. With three good agents and these restrictions our model easily runs for MaxLevel=10 and HL=5, duly creating a chain of length 11. Thanks to the restrictions on the intruder detailed in the previous paragraph, replacing a good agent by a bad one leaves it highly tractable. The biggest check we tried was the last one with MaxLev replaced by 14, which took 20 minutes evenly split between compilation and enumerating just over 21M states.

3.4 Specification

FDR expects its users to design specifications to test their systems against. Usually these specifications are properties we want them to satisfy, but occasionally we hope they fail. For example when we want to solve a puzzle like peg solitaire or Sudoku there is a well tried technique of coding the puzzle in CSP with legal moves as the possible events. By adding an extra event that the puzzle can perform when it is solved, and specifying this cannot happen, FDR can frequently find solutions by saying that this specification has failed, even though this is exactly the result we were hoping for.

Similarly, in our blockchain model, we can test it against specifications that say things like "it never creates a chain of length 10", and hope this is false, as well as checking that some error conditions cannot occur. We can also test for well known pathologies such as deadlock and observe how the state space varies with various parameters.

4 Watching the blockchain

In this section we describe how the third file, which implements the watchdog, works.

Aside from basic structural issues that should not happen under any reasonable circumstances given the checks we have built into our model, the watchdog is designed to look for cases where a pair of good agents have come to disagree in a significant way on the head block of the chain. For a basic property of blockchains should be that good agents agree reasonably promptly on the growth of a chain of blocks.

So to create the specification built into the watchdog we pick a pair of good agents and test how their views develop relative to each other. Let's call them Alice and Bob. At any time they will respectively believe in candidates `HA` and `HB` as the head of the chain. Since there are updated separately there clearly do not always agree. In particular the model allows `HA` or `HB` to be updated by any block for which respectively HA or HB is on the predecessor path. The specification must allow for that.

In general terms we would like the watchdog to specify how much Alice and Bob can meaningfully disagree about chain development. A natural way to capture this is to specify that if `PA(j)` and `PB(j)` are the blocks at level `j` on the paths from `HA` and `HB` to the root, and +j+ is at least k steps behind both `HA` and `HB` (with $k \geq 0$ a parameter) then `PA(j)` and `PB(j)` are equal.

If $k = 0$ this specifies that no fork can develop at all, and if $k > 0$ it means that the shorter arm of any fork can be no longer than k.

In fact the way our particular model treats good agents means they check the validity of a candidate head demands that it directly develops its current head. This means that even a violation at $k = 0$ would mean that Alice and Bob can never agree again.

Apart from checking basic coherence properties of our blockchain model, we want to be able to analyse how patterns of blocks can appear and what the good agents participating see and believe. In order to create specifications of the latter we seem to need to embed the good agent's beliefs and the graph representing the predecessor relationship within the specification. Even with an apparently functional model a very wide range of predecessor relationships amongst the hashes representing blocks might arise — at least any acyclic graph where each of a subset of the hashes hash has a unique predecessor except the genesis hash which is at the and of all such paths. This number grows very rapidly with the number of hashes. Indeed it quickly becomes too large for FDR to be tasked to enumerate them all.

We have solved it by using a trace watchdog [6] in place of a conventional specification, namely a process that runs in parallel with the system S synchronising on relevant actions but never stopping an S action before "barking": watching S and identifying when the latter performs an unwanted trace. Such traces prompt it to signal something has gone wrong by communicating a simple event we check for, namely `bark`.

In our case we implement the watchdog in parallel, just like the pipelined hashers described earlier. We give the watchdog a parallel component — a Cell — for each possible hash: in our model there are `1 + N * HL` of them. This records whether this hash has been allocated yet to a block, and if so what that block's hash pointer — its predecessor is. Unless the predecessor is the genesis hash we expect that to have a predecessor too — or it barks – and so on back to the genesis block. This achieves a parallel record of the evolving tree of hashes that caused the combinatorial problems.

To detect issues with the tree and the good agents' beliefs we have the `Cell(h)`s record relevant things about the good agents, for example, which of these agents thinks `h` is the hash of the current chain head or where this block is on the path from its head back to the root.

In common with other representations of complex data types using parallel compositions, it has the huge advantage that only the states that the system actually visits are explored. In this respect it is just like the well-known parallel intruder and other "bit sets". Written accurately, such watchdogs should neither noticeably increase nor decrease the state space that the system naturally has without it. If system +S+ does not make its watchdog `WD` bark, `S` and `S[|AWD|]WD` should be trace equivalent, where `AWD` is the alphabet that `WD` watches.

The watchdog we implemented is a slightly simplified version of the description above. For a non-genesis hash its state is described `Cell(h,pred,hd1,hd2,p1,p2)` where `h` is this hash, `pred` is `Null` or `h`'s predecessor, and the other parameters are booleans which are true if `h` is the current head in the minds of agents 1 and 2, or more generally if it is on the path back from. these two, respectively. It chooses two representative good agents to track: in examples where agents 1 and 2 are good it singles them out and compares their knowledge of the evolving blockchain – hence the presence of the four booleans above. This parametrisation allows us to introduce events `bark.h1.h2` which is synchronised between the cells of `h1` and `h2` when agent 1 thinks `h1`is the head, agent 2 thinks `h2`is head, and neither is on the path back from the other. The watchdog nodes can work these out, by a process that works like the tracing that good agents perform, because of them knowing the predecessor relationship.

Thus, the watchdog barks when good agents have inconsistent ideas about the head: a simple fork.

If, instead of the booleans `p1` and `p2`, we recorded the numbers of steps back to `h` from the two heads, we could make the watchdog more discriminating about barking.

5 Results

All the experiments we have done were on systems with a small number of agents, typically 2-4, of whom no more than one was bad. Increasing any of the parameters increases the alphabet size, the compilation time and the running time of the checks. We carried them out on a Macbook Pro laptop with a 12 core CPU and 96Gb of memory. Once we had arrived at our final model we ran some largish checks such as one with `N=3` including one bad agent, `MaxLev=12`, `M=1` and `HL=5`. A check to simply enumerate the states required 4 minutes to compile and 8.5 minutes to run through the states, finding 20M states: rather slow per state because of the complexity of the system and the use of `chase`. In this model there were 36,712 visible events.

Increasing `MaxLev` to 15 and `HL` to 6 increases the alphabet size to 61,858. There are now 58M states that took 40 minutes to run. This illustrates the problem of state explosion but does indicate we can examine behaviour to a reasonable depth.

To test the watchdog we ran three sorts of check. The first, where there were no bad agents and the `myturn(a,h)` function always allowed exactly one creator, namely it was deterministic, found no barking. The second had one bad agent and a deterministic `myturn`, and the third had no bad agents but a nondeterministic `myturn` which sometimes allowed more than one agent to proceed. The second and third, as expected, did cause forking and hence barking. Example traces for these are shown below.

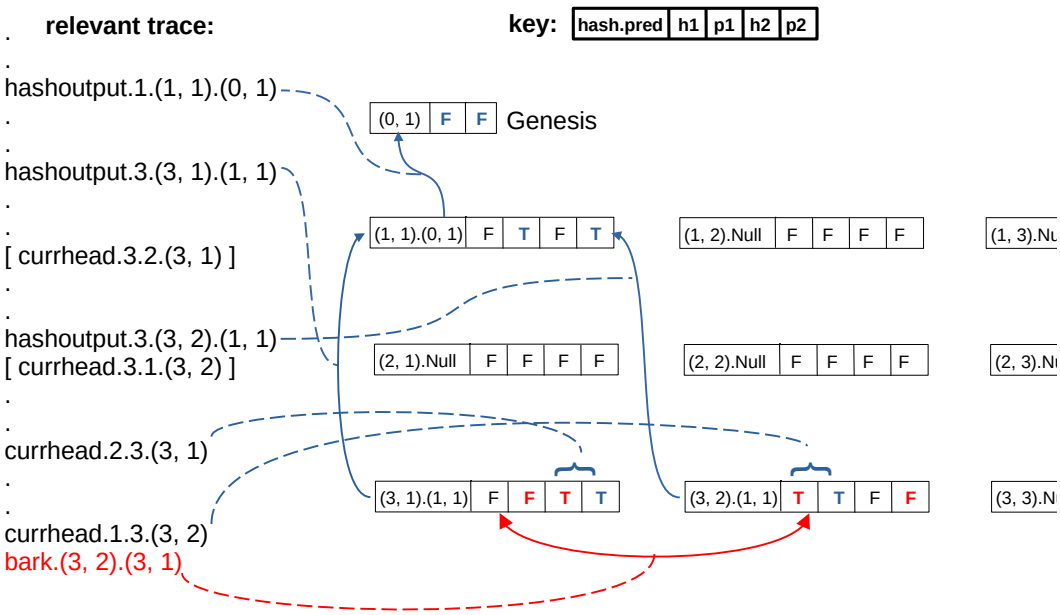

Figure 2: Final watchdog state after an example error trace.

```
nodeon.1
readbyhash.1.(0, 1).Genesis
nodeon.3
hashinput.1.Block.1.1.1.(0, 1)
hashoutput.1.(1, 1).(0, 1)
readbyhash.3.(1, 1).Block.1.1.1.(0, 1)
hashinput.3.Block.3.2.1.(1, 1)
hashoutput.3.(3, 1).(1, 1)
nodeon.2
currhead.3.2.(3, 1)
readbyhash.3.(1, 1).Block.1.1.1.(0, 1)
readbyhash.2.(0, 1).Genesis
readbyhash.2.(3, 1).Block.3.2.1.(1, 1)
readbyhash.2.(1, 1).Block.1.1.1.(0, 1)
hashinput.3.Block.3.2.2.(1, 1)
hashoutput.3.(3, 2).(1, 1)
currhead.3.1.(3, 2)
readbyhash.1.(1, 1).Block.1.1.1.(0, 1)
readbyhash.2.(0, 1).Genesis
currhead.2.3.(3, 1)
readbyhash.1.(3, 2).Block.3.2.2.(1, 1)
readbyhash.1.(1, 1).Block.1.1.1.(0, 1)
currhead.1.3.(3, 2)
bark.(3, 2).(3, 1)
```

```
nodeon.1
readbyhash.1.(0, 1).Genesis
hashinput.1.Block.1.1.1.(0, 1)
hashoutput.1.(1, 1).(0, 1)
nodeon.3
currhead.1.3.(1, 1)
readbyhash.3.(0, 1).Genesis
readbyhash.3.(1, 1).Block.1.1.1.(0, 1)
readbyhash.3.(0, 1).Genesis
readbyhash.3.(1, 1).Block.1.1.1.(0, 1)
nodeon.2
hashinput.3.Block.3.2.2.(1, 1)
hashoutput.3.(3, 1).(1, 1)
currhead.3.2.(3, 1)
readbyhash.2.(0, 1).Genesis
readbyhash.2.(3, 1).Block.3.2.2.(1, 1)
readbyhash.2.(1, 1).Block.1.1.1.(0, 1)
readbyhash.2.(0, 1).Genesis
currhead.2.1.(3, 1)
readbyhash.1.(1, 1).Block.1.1.1.(0, 1)
readbyhash.1.(3, 1).Block.3.2.2.(1, 1)
readbyhash.1.(1, 1).Block.1.1.1.(0, 1)
readbyhash.1.(3, 1).Block.3.2.2.(1, 1)
hashinput.1.Block.1.3.2.(3, 1)
hashoutput.1.(1, 2).(3, 1)
readbyhash.2.(3, 1).Block.3.2.2.(1, 1)
currhead.1.3.(1, 2)
hashinput.2.Block.2.3.1.(3, 1)
hashoutput.2.(2, 1).(3, 1)
currhead.2.1.(2, 1)
bark.(1,2).(2,1)
```

The first of these traces, with most of the events not relevant to the watchdog removed, is driving the watchdog in the figure that illustrates it.

Overall it was clear that, as expected, the state explosion problem has a big impact on the size of system our model can handle, these limits were significantly less restricting than we had feared. On the other hand we discovered that the state space was more affected than expected by the nondeterminism of a bad agent or ambiguous block creation.

6 Related Work

Previous research has explored the application of formal methods, including CSP, to blockchain systems. One notable approach is the use of CSP for modelling and verification of consensus protocols and distributed ledger systems (e.g., Bitcoin, Ethereum). In particular, researchers have studied how to represent cryptographic properties, consensus mechanisms, and the interaction of agents in a blockchain network.

In [13] the authors apply CSP and FDR to model and verify the core of a blockchain consensus protocol.

However, most of these models tend to ignore the practical computational aspects such as state space explosion when applying model checking tools. Our work builds on prior research by addressing this challenge using a symbolic representation of hash functions and an approach for reducing the complexity of the model. This reduction allows us to scale the model for analysis using FDR, making the verification of blockchain protocols more feasible for real-world scenarios. We are thus able to model whole blockchains where previous work has concentrated on small aspects.

7 Conclusions

In this paper, we have demonstrated how CSP can be used to model a simple blockchain system and analyse its behaviour using the FDR model checker. We introduced techniques to reduce the state space and alphabet size, making the verification process computationally feasible. Our approach provides insights into the design and verification of blockchain protocols, particularly in how agents interact with the blockchain and verify its integrity.

The results demonstrate that the CSP model, optimised with state reduction techniques, can be successfully verified using FDR. Despite the complexity of the blockchain system, our approach allows for tractable verification, making it possible to model real-world blockchain systems with high confidence.

Future work could explore more complex and complete blockchain systems, including a consensus mechanism for example, to demonstrate that this would eliminate the forking observed in the model in this paper. Some consensus mechanisms envisage the possibility of temporary forking amongst *non-final* blocks near the head and rely on verification and negotiation to eliminate it. Others apply carefully chosen criteria before any good agent accepts a proposed block – significantly more careful and discriminating than the simple criteria in the model of the present paper. Having seen the scale of model that our techniques can handle, we are confident that they can be extended to handle interesting, though simple, cases of each of these.

The hashing mechanism used in our model seems far removed from a standard cryptographic hash function, yet fulfils the same purpose. It would be beneficial to formalise the correspondence between our hashing model and a cryptographic hash, to show that they are essentially equivalent. It will be interesting to discover if there are other uses of this model.

Analysis in the traces model of CSP can realistically only find ways in which the blockchain can reach unwanted states such as established forks or non-compliant blocks. CSP and FDR also provide

tools to analyse liveness issues: it would be interesting to analyse circumstances in which we can prove that a blockchain will extend indefinitely. One can imagine that sometimes this will be caused by bad agents failing to make required contributions, and sometimes by them introducing confusion through the things they actually say.

We might minimally expect that a community of agents who all behave well should produce a safe, live blockchain: one that always grows and never fails desired safety properties such as freedom from forking. As limits on bad behaviour are increased, these properties will eventually fail, and our CSP models will help to quantify when this happens.

For the time being, models in our style will have to be severely limited as to parameters such as numbers of agents and number of blocks they allow. Just as is the case in other applications of model checking, it should be possible at the very least to understand how the results will apply to larger systems at least informally and in some cases formally thanks to data independence and similar approaches. Our existing blockchain models, like the early CSP models of cryptographic protocols, are small in every respect: numbers of agents, length of chain etc. In the crypto protocol case these restrictions were eventually relaxed because there were formal proofs that a small number of agents was often sufficient, or members of small types could, with care, be recycled in association with data independence arguments [14]. In that case it could thus often be shown that a proof of correctness in a small model could establish the general correctness of a protocol. It will be interesting to see if this might also apply to blockchains.

References

[1] Brookes, S.D., Roscoe, A.W.: CSP: A Practical Process Algebra, p. 187–222. Association for Computing Machinery, New York, NY, USA, 1 edn. (2021), https://doi.org/10.1145/3477355.3477365

[2] Buterin, V.: Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/whitepaper/ (2014)

[3] e Estonia: Ksi blockchain provides truth over trust (2022), https://e-estonia.com/ksi-blockchain-provides-truth-over-trust/, accessed: 2025-04-14

[4] Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 — A Modern Refinement Checker for CSP. In: TACAS. LNCS, vol. 8413, pp. 187–201 (2014)

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

[6] Goldsmith, M., Moffat, N., Roscoe, B., Whitworth, T., Zakiuddin, I.: Watchdog transformations for property-oriented model-checking. In: FME. pp. 600–616 (2003)

[7] Harel, D.: Statecharts: A visual formalism for complex systems. Science of computer programming **8**(3), 231–274 (1987)

[8] Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)

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

[10] Nakamoto, S., et al.: Bitcoin: a peer-to-peer electronic cash system (2008) (2008)

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

[12] Roscoe, A.W.: Understanding Concurrent Systems. Springer (2010)

[13] Roscoe, A.W., Antonino, P., Lawrence, J.: Abstracting and Verifying Decentralised Systems in CSP, pp. 172–202. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-67114-2_8, https://doi.org/10.1007/978-3-031-67114-2_8

[14] Roscoe, A.W., Broadfoot, P.J.: Proving security protocols with model checkers by data independence techniques. J. Comput. Secur. **7**(1), 147–190 (1999). https://doi.org/10.3233/JCS-1999-72-303, https://doi.org/10.3233/jcs-1999-72-303

[15] Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M.H., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In: Brinksma, E., Cleaveland, W.R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 133–152. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)

[16] Roscoe, A., Antonino, P.: The challenges and triumphs of csp based formal verification. In: Quarrie, L. (ed.) Proceedings of 2024 Concurrent Processes Architectures and Embedded Systems Hybrid Virtual Conference. Kalpa Publications in Computing, vol. 20, pp. 1–16. EasyChair (2024). https://doi.org/10.29007/bbq9, /publications/paper/SPpC

[17] Roscoe, A.W., Antonino, P.R.G.: Embedding reverse links in a blockchain. In: Embedding reverse links in a blockchain (2022), https://api.semanticscholar.org/CorpusID:254293448

[18] Roscoe, A. W.: Model-checking CSP, chap. 21. Prentice Hall (1994), http://www.cs.ox.ac.uk/people/bill.roscoe/publications/50.ps

[19] Walport, M.: Distributed ledger technology: Beyond blockchain. Tech. Rep. GS/16/1, UK Government Office for Science (2016)

[20] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)