Flexible Formality
Practical Experience with Agile Formal Methods

Philipp Kant$^{1[0000-0001-9331-1462]}$, Kevin Hammond$^{1[0000-0002-4326-4562]}$, Duncan Coutts$^{1,2[0000-0001-7450-4429]}$, James Chapman$^{1[0000-0001-9036-8252]}$, Nicholas Clarke$^{1,4[0000-0002-8912-4530]}$, Jared Corduan$^{1[0000-0003-3838-3038]}$, Neil Davies$^{1,3[0000-0001-9462-8584]}$, Javier Díaz$^{1,6[0000-0003-3588-7468]}$, Matthias Güdemann$^{1,5[0000-0002-1002-6023]}$, Wolfgang Jeltsch$^{1,2[0000-0002-8068-8401]}$, Marcin Szamotulski$^{1[0000-0002-9902-8315]}$, and Polina Vinogradova$^{1[0000-0003-3271-3841]}$

[1] IOHK `firstname.lastname@iohk.io`
[2] Well-Typed `firstname@well-typed.com`
[3] PNSol `neil.davies@pnsol.com`
[4] Tweag `nicholas.clarke@tweag.io`
[5] University of Applied Sciences Munich `matthias.guedemann@hm.edu`
[6] Atix Labs `jdiaz@atixlabs.com`

Abstract. Agile software development and Formal Methods are traditionally seen as being in conflict. From an *Agile* perspective, there is pressure to deliver *quickly*, building vertical prototypes and doing many iterations / sprints, refining the requirements; from a *Formal Methods* perspective, there is pressure to deliver *correctly* and any change in requirements often necessitates changes in the formal specification and might even impact all arguments of correctness.

Over the years, the need to "be agile" has become a kind of mantra in software development management, and there is a prevalent prejudice that using formal methods was an impediment to being agile. In this paper, we contribute to the refutation of this stereotype, by providing a real-world example of using good practices from formal methods and agile software engineering to deliver software that is simultaneously reliable, effective, testable, and that can also be iterated and delivered rapidly. We thus present how a lightweight software engineering methodology, drawing from appropriate formal methods techniques and providing the benefits of agile software development, can look like. Our methodology is informed and motivated by practical experience. We have devised and adapted it in the light of experience in delivering a large-scale software system that needs to meet complex real-world requirements: the Cardano blockchain and its cryptocurrency ada.

The cryptocurrency domain is a rather new application area for which no clear engineering habit exists, so it is fitting well for agile methods. At the same time, there is a lot of real monetary value at stake, making it a good fit for using formal methods to ensure high quality and correctness. This paper reports on the issues that have been faced and overcome, and provides a number of real-world lessons that can be used to leverage the benefits of both agile and formal methods in other situations.

1 Introduction

There has long been a tension between Software Engineering and Formal Methods. From a *software engineer's* perspective, there is pressure to deliver *quickly*; from a *formal methods* perspective, it is essential to deliver *correctly.* In this paper, we argue that rather than fueling this tension, formal methods not only can, but should, be *fused* with agile software engineering methods. The goal is to promote a *flexible* software engineering methodology that aims to combine the best aspects of both agile and formal methods to deliver properly engineered and correct software solutions quickly and effectively. We illustrate how such a methodology can look like by referring to our experience at IOHK, a company that is using strongly typed and functional programming (specifically Haskell) to deliver a new cryptocurrency.

1.1 Formality versus Agility

Agile software development [BBvB+01] has, since its inception at the turn of the century, risen to become one of the most prevalent software development methodologies. Agile methodologies are attractive because they promise rapid delivery, and fit normal development approaches. When done well, with a focus on what *needs* to be delivered, rather than what is *easily* delivered, agile techniques allow effort to be focused towards the most important goals, and away from unimportant goals. However, if they are to be used successfully, *discipline is essential* and *management must exercise strong control.*

Agile techniques can appeal to undisciplined developers precisely because they can deflect attention from what *needs* to be done (which is often hard) towards what can *quickly* be done. This allows an *illusion of progress* to be maintained. Management is then happy because they can apparently observe progress, and the software is close to product, or only needs a few more small adaptations; and software developers feel valued because they are producing code that is apparently appreciated, and there are continual exciting challenges that they must overcome. Unfortunately, the software may have little real utility, may be hard to maintain, and may also be unreliable. When this happens, "agile" methods are both costly and ineffective: the precise opposite of the motivation for adopting them.

In contrast, classical formal methods require careful thought and design. It is necessary to first carefully specify a system, then to laboriously translate this into an implementation, and finally to verify the result against some complex and often hard-to-understand semantics. Since a large fraction of the overall time and work is spent on writing specifications, it can be hard to demonstrate progress unless the specifications are accessible to management. Furthermore, changes to the software product require changes in the specification, code, and verification, which can act as a barrier to accepting changes in requirements.

For this reason, commercial product teams can be very wary of adapting formal methods, and startups can feel that they cannot afford the costs. This is a pity, since modern formal methods do not have to suffer from these drawbacks.

For example, using executable specifications are a great tool to demonstrate progress, and automated tools like QuickCheck can be used to check correctness of software in a way that is stable against local changes. We hope that by providing our own positive experience, we can help reducing the bad reputation of formal methods being too slow and inflexible for practical things, and ultimately encourage more practitioners to consider using some formal techniques.

1.2 Our Contribution

In this paper, we argue from our own experience that the perceived dichotomy between "being agile" and "being formal" is mostly a consequence of an oudated view on the landscape of formal methods[7], and that using modern formal techniques not only does not contradict the goals of quickly delivering software in the presence of changing requirements, but that they are indeed rather helpful. This paper makes the following contributions:

- We describe the motivation that led to the real-world adoption of formal methods techniques and functional programming technologies within an advanced technology company (IOHK);
- We provide examples of the real-world use of lightweight formal methods and functional programming as part of a large software development process;
- We consider the positive and negative aspects of both formal and agile techniques in the light of experience with both approaches, as well as the gap in left between the methodologies;
- Based on this analysis we outline a *flexible formal* software engineering methodology that provides the most significant benefits of both agile and formal software development;
- We discuss the advantages of functional programming for *flexible formal* software development.

Moreover, there are relatively few reports of real-world experiences of using functional programming technologies as an intrinsic part of large-scale, distributed software development (exceptions include e.g. reports on Erlang). This paper provides another addition to this corpus.

2 Cardano: a Proof-of-Stake Cryptocurrency

Cardano (https://www.cardano.org) is a novel decentralised blockchain and cryptocurrency that is being developed by IOHK. cryptocurrencies are distributed systems that contain a public shared transaction *ledger*, which allows participants

[7] In fact, we would go even further and say that the the picture of a waterfall-style development, with a strictly linear succession of gathering requirements, writing specifications, writing code, and proving correctness against the specification, was always more of a caricature of a bad approach than an accurate description of how people were using formal techniques in practice.

to track and send funds in a virtual currency. The striking feature is that these systems are permissionless and decentralised, in the sense that anyone can run a node and take part in maintaining the *ledger* without needing to be registered with a central authority,

This poses an immediate problem: since there is no central authority, it is necessary to reach *consensus* on how to progress the construction of the *blockchain*. The consensus algorithm has to be resistant to a malicious actor setting up any number of nodes with the aim of taking over the decision finding process (a so-called *Sybil attack* [Dou02]). Bitcoin [Nak09], the first cryptocurrency, achieves this using a *Proof-of-Work* (PoW) mechanism, where taking part in the consensus requires computational resources that are proportional to the total amount of computational resources in the system. This renders a Sybil attack highly expensive. The cost is in making the whole system ridiculously inefficient: Bitcoin is now at a stage where it consumes as much electrical power as a mid-sized nation state, but can only enter a handful of transactions into its ledger per second. Were it not for the computational cost of the PoW Sybil protection, this could be easily achieved using a single commodity laptop or other small device.

In contrast, Cardano uses an alternative *Proof-of-Stake* mechanism (PoS). Under PoS, the price of participating in the consensus algorithm is not paid in computational power, but instead by having to own some of the virtual currency in the system. The larger your share of the total funds (the higher your stake), the greater is the probability of your being elected as the leader in the next consensus round.

While PoS has many advantages over PoW – it is ecologically sustainable, and automatically incentivises powerful parties in the consensus to behave honestly (since large stakeholders have a lot to lose if the system is found to be manipulated) – it is hard to get right. For this reason, IOHK committed itself to base Cardano on a solid foundation of original peer-reviewed research, and to using formal methods in the development process.

There are already a lot of moving parts to the Cardano cryptocurrency system. In time, it will additionally become a smart contracts platform, running the languages Plutus[8] and Marlowe [LST18], which have been specifically designed to be used on Cardano.

3 Formal and Agile Development of Cardano

While IOHK has always been devoted to getting things right, building upon sound academic research and robust, reliable engineering, the company is also aware of commercial realities, such as the importance of *time-to-market* in a relatively young and quickly evolving sector. For this reason, it set out on a two-pronged approach for Cardano: a team A of energetic developers would quickly develop, in an Agile manner, a Minimal Viable Product (MVP) to release to market. Meanwhile, a second team B would aim for a high-assurance version,

[8] `https://github.com/input-output-hk/plutus`

using formal methods, that would, once ready, replace the first implementation. Team A would deliver swiftly, and Team B would use the experience from having a working system in production to guide their design and development. Both implementations were done in Haskell.

Some time after releasing the MVP, it became clear that maintaining it and adding new features was much harder than anticipated. The organically grown code, which had been developed under time pressure in an agile style, lacked a proper separation of concerns or good documentation of the design. This resulted in poor testability and extensibility for the codebase. Crucially, the implementation of some key features (namely, a proper system for stake delegation) had been delayed until the very end, and by that time, design choices that had been made while implementing other, simpler, functionality, had made that task more complicated than it would have needed to be. As a consequence, estimated development times for the missing features, as well as for future features, were much longer than they needed to be.

At the same time, team B had achieved a first success, in successfully implementing a wallet[9] for Cardano based on a semi-formal specification. A decision was thus made to pivot, cutting back development effort on the existing implementation to a bare minimum. Team B would scale up and accelerate their efforts, and the next features on the roadmap would be implemented exclusively in the follow-on to the MVP. At this point, Team B faced a number of challenges:

- Since team A was no longer adding new features, they had to accelerate their pace in order to quickly get to a point where the new implementation could be used to deliver new features.
- Compromising on the quality and robustness, or future maintenance costs, was not an option; Cardano has to safely manage and secure large-scale financial transactions, and needs to be fit for that purpose.
- They had to ensure backwards compatibility with the already released code. The lack of good documentation meant that they had to write a specification based on the existing code. Writing specifications and code adhering to them is like time travel, in that one direction is significantly easier than the other.
- As the research and design for the new features were still somewhat in flux, they would need to be flexible to adjust to changing requirements.

To overcome those challenges, the team chose a pragmatic approach – with a well-dosed, non-dogmatic use of *lightweight* formal methods, and a focus on rapid delivery – that we will describe in this paper.

4 "Flexible Formal Development": a Fusion of Formal Methods and Agile Software Engineering

Both agile software development and formal methods aim at helping their practitioners to become "better" at producing software, but they focus on different aspects:

[9] A cryptocurrency wallet is a piece of software that allows users to track their balance in the system and submit transactions.

agile is all about speed and flexibility; formal methods is all about correctness and method. This is not helped by the number of books, papers and experts that promote specific methods (whether formal or agile) as a complete solution. Examples include Agile Scrum Methodology [SB01]; Lean Software Development [PP03]; Kanban [Bre15]; Extreme Programming (XP) [Bec00]; Feature Driven Development (FDD) [PF01]; Model Checking [CGP99]; Abstract Interpretation [CC77]; Type-Driven Development [Bra16] etc. In this section, we will explore the broad differences, similarities, and potential synergies between formal and agile approaches and aim to understand how their fusion can ensure software that is both high-assurance and reasonably time- and cost-effective to produce.

4.1 What do we need?

Fundamentally, software development needs are quite simple. In general, we need to produce software that does what it is supposed to do; is produced quickly; costs no more to produce than is necessary; can be easily maintained, at reasonable cost; and doesn't require expensive support. Other issues are generally secondary or specific to particular domains (e.g. telecommunications applications may have real-time constraints, aerospace applications may have overriding safety concerns, autonomous vehicles may have regulatory concerns, etc). The basic criteria for a successful methodology which is shared by many software development domains is presented in Table 1.

Table 1: Criteria for software engineering methodologies, along with stereotypical expectations of whether agile or formal methodologies satisfy them. This is to be taken with a grain of salt, as there is a large variety of both agile and formal techniques.

Issue	**Agile**	**Formal**
Identify the requirements for the software	Y?	Y
Ensure that the software meets these requirements	Y?	Y
Provide usable prototypes rapidly	Y	Y?
Minimise the costs of development	Y?	N?
Ensure that code is high quality	N	Y
Ensure that software is easy to use	N	N
Ensure that changes can be made easily	Y	N
Be easily applied without extensive training	N	N

The details of this table can be argued, of course, mainly because there are many different agile techniques and many different formal methods. Different development teams may also have different levels of experience and be more or less familiar with specific techniques and technologies. They will also have different competencies in terms of e.g. mathematical backgrounds or training in specific development techniques. Effective deployment of either technology, however, needs extensive specific training and practice. We will consider each of the issues from the table in detail, considering how well they are met by agile and formal development techniques.

Identify Requirements. Here, the key issue is to have a strong product vision. Ideally, there should be a dialogue between the *product manager* and the *software developers.* **Agile** developers should then interact with the *product manager* to deliver the capabilities in the software that is needed, and the *product manager* should adapt the capability requirements of the product to make it easier to implement/maintain, without compromising on essential features. In practice, there may be no distinct *product manager*, meaning that the *development team* acts as the designers. This can create a number of problems, including failure to deliver a successful product, repeated non-converging iterations, missing essential features, and included non-essential features. Requirements gathering and design is done on the fly. Because it is easy to change requirements, the software design and implementation will frequently change direction. The final solution will then have no clear design pathway. **Formal methods techniques** on the other hand often require detailed and careful analysis of alternatives, followed by months of painstaking work to laboriously craft out possible solutions, prove that they are sound with respect to some formal model or semantics, and then to verify that the software matches those requirements. Even small changes may require major alterations to the formal specification, and significant effort to re-prove, re-verify and then re-implement the software. In this approach, it is therefore essential for the product owner to be involved in the requirements analysis and problem specification. Unfortunately, they will often lack the technical/mathematical knowledge to be able to understand the implications of the design decision.

Meet Requirements. Since **formal methods** use mathematical techniques to specify requirements, provided that they are properly captured and the process is followed correctly, then the software will always meet these requirements. This is a major strength of a formal approach. When using **agile methods**, on the other hand, the *product owner* – and also users, where early delivery is used – can easily see the current version of the software, identify any mistakes or misunderstandings and feed corrections into the development process.

Provide Prototypes. Good **agile methods** will always ensure that a prototype is available. By using *continuous integration* and *continuous testing*, a non-breaking version will always be available for deployment. Non-breaking means, of course, that the code will compile and that none of the tests have failed, not that the code works perfectly. However, it is easy to observe change, and therefore to measure (real or apparent) progress. Some **formal methods** also allow the production of prototypes. For example, where a modelling approach is used, an *executable specification* might be produced, or where a *refinement process* is used, then successive refinements will produce gradually more detailed prototypes. However, this is not a feature of all formal methods techniques. Because it is usually necessary to formally prove software correctness, there may be long periods when no new software versions are produced. Since there is no observable change, it is difficult to measure progress during such periods.

Minimise Development Cost. A key goal of **agile** (especially *lean*) software development is to minimise software costs by producing precisely the minimal product that is required, and by focusing attention on the most important features. By avoiding implementing unnecessary features or by delaying less useful features, the software can be brought to market more quickly, and at an adequate cost. In practice, achieving this requires strong discipline. It is easy to focus attention instead on short-term, but less important bug fixes, on easy-to-implement features, or on features that are nice-to-have. While daily "stand-up" meetings allow good team communication, they need to be properly organised if a priority task list is to be produced and followed. By using *continuous testing*, software is not accepted that does not pass regression tests, so fewer bugs will enter the code base. However, this same process can also act as a barrier to major change – completely new tests will then be necessary. In contrast, reducing development cost is not usually a major goal of **formal methods** development. If correctness is paramount, then spending effort to ensure correctness is always the right thing to do. Although there has been major progress in e.g. automated proof assistants and model checking, most formal methods tooling is not well integrated into the usual software development process.

Ensure High Quality. The primary aim of **formal methods** is to produce very high quality, high reliability, high assurance software. This is, however, rarely an explicit goal of **agile methods**.

Maximise Ease of Use. Ease of use is not a primary goal for either agile software development or when using formal methods. Rather, it must be layered as an additional concern.

Enable Change. Software is notoriously hard to change. While **agile methods** allow design changes to be incorporated during development, as discussed above, they do not encourage major design changes: any significant change will break not only the existing code, but also testing, documentation, etc. Similarly, traditional **formal methods** do not provide any assistance with major design changes. While small changes can usually be incorporated without major work, large changes will often require significant and laborious specification, verification, proof or other work. In both cases, it is often easier to start with a blank canvas and produce a completely new design. This can also be cheaper and quicker than adapting an existing design. However, it means that significant effort has been wasted.

Do Not Require Extensive Training. There is a major software skills shortage. As evidenced by e.g. salary levels, good software developers are rare and in high demand. It is not cost effective to require them to learn to use new tools and techniques on a regular basis. While they may be highly effective once mastered, mathematical techniques may also require extensive study and practice, which is also costly. Unfortunately, much of the available tooling to support **both agile software development and formal methods** is special-purpose and requires extensive time to learn to use effectively. This creates stickiness: better tooling is not used because it takes time to learn to use (or sometimes, especially in smaller companies, because it costs money). It also means that few people have experience with both kinds of tools or the expertise to move easily between them.

Our Goal: Flexible Formal Software Engineering Based on the analysis above, we argue for a *flexible formal* approach. Our goal is to combine the best elements of agile and formal software engineering so that we can produce software that meets all of the criteria above. In particular, it should be high quality, quick and cost effective to produce, easy to change, clearly meet the requirements and not require extensive training to develop. This is naturally highly ambitious, and in this paper we will only be able to report on the initial steps that we have taken. However, it is important that the software development community does not simply settle for the *status quo* but strives to achieve these goals. In this way, we will be able to deliver software that is better, less costly, and easier to adapt both *by design* and *by construction*. Modern functional programming is key to helping us achieve this.

5 Key Messages and Lessons

5.1 Approach(es) taken at IOHK

When rebuilding Cardano, we separated concerns into layers, as is common when dealing with larger projects. This allowed us to parallelise work, test things in isolation, and will allow us to swap out individual components when needed, to produce customised variants. It turns out that there is sufficient difference in nature between the components to make each amenable to a different approach in designing and implementing them. In the following, we will briefly describe each layer, and explain the methodology chosen for each, and why.

Ledger Layer The ledger layer contains the main logic of the cryptocurrency. It is where all the data is kept, and has to ensure that users' balances are recorded correctly, that money can not be arbitrarily created or destroyed, that no one can spend funds they do not own (or spend their funds twice), etc. Correctness of the ledger is thus of utmost importance to the integrity of the system.

The Cardano ledger is of moderate complexity. It does not have to deal with any concurrency issues – those are contained in the consensus and networking layers – but it is more than just a simple book-keeping device. In addition to listing and ordering transactions, and keeping balances, it has to also keep track of state that is important for the operation of the system itself. Parameters of operation (such as the frequency with which new blocks[10] are created) can be adjusted during operation, by announcing the new value on the ledger. Similarly, new versions of the software itself can be announced via an update mechanism. Another aspect of the ledger is *delegation*: while every stakeholder has the *right* to participate in the consensus algorithm, it is unlikely that each and every user of the system would want to continuously run and maintain a node in the system. In Cardano, users can chose to delegate their stake to people who do run a node, forming a *stake pool*. Rewards that the system pays out for maintaining consensus are automatically shared between operators and participants of such pools.

All of this lead to a rather voluminous design; the informal document describing the mechanisms of delegation and incentives alone [SL-D1] runs at roughly 60 pages, and builds upon two papers of original research conducted for Cardano [KKL18,BKKS18]. While none of the individual parts are rocket science, they can interact in subtle ways. Since the ledger is where the value is being held, correctness has to be on the top of the list of priorities of the development methodology chosen. However, we also needed a flexible approach: commercial reality required us to start work on the implementation before the design and research of the whole ledger was truly finished, so choosing an approach where small changes in the design would require massive amounts of work to be done had to be ruled out.

We decided to simplify the ledger design by defering all stateful operations – in particular data storage and issues related to eventual consistency – to the consensus layer. This allows us to express the whole ledger logic in a purely functional way, in terms of a set of valid state transitions and transition rules. The approach we followed in defining operational semantics is called *small-step semantics* (see [FM-TR-2018-01]). We can use these operational semantics to define valid state transitions in a deterministic way, e.g. what sequences of transactions forms a valid ledger. We will not discuss here the language and rules itself, but instead summarize the principles we follow in constructing rules and types with this approach:

- There should always be a unique way to represent every state transition as a sequence of these ”small steps”. E.g., to apply a block, the sequence of intermediate states must contain each of the states resulting from applying every transaction in the block individually to the ledger ledger state resulting from applying the preceeding one.
- There should not be any partial state transition rules or unnecessary data dependencies between state transitions. E.g., we do not want to make separate state transition rules for processing the inputs and the outputs of a transaction.

The first principle requires us to define rules with high granulariy, so that we don’t miss any intermediate steps. The second principle discourages us from having unnecessary intermediate steps, during which some invariants we expect from the system may not hold.

[10] Cryptocurrencies are built on a data structure called blockchain, which are essentially linked lists, where each block contains a page of a transaction ledger

We call the transition types, together with the transition rules we defined in this way, a *semi-formal* specification, since it is formal, but not machine checked. Translating this specification into valid Haskell code is straightforward and mostly mechanical: every transition rule corresponds to a pure Haskell function, with some pre- and postconditions. This gives us an *executable* specification.

This is a very powerful tool: the typechecker is very good at finding subtle self-consistency errors. Since it is executable, this specification can serve as a prototype and demonstrate tangible progress to stakeholders. You can also start running tests with the executable specification. Lastly, it can either serve as a basis for developing a production implementation through a series of refinement steps, or as a testing oracle for the production implementation (we chose the former here).

We did not, initially, produce any proofs about the emergent properties of the ledger (such as conservation of value, delegating stake properly modifying the stake of a pool, etc.). Instead, we got reasonable confidence by having the executable specification pass the type checker (we got the plumbing right), and by writing the desired properties as QuickCheck properties. Not performing proofs at this stage allowed us to move quickly, and react to changes in the design. Having the type checker and QuickCheck properties allowed us to do so with confidence that the changes were not breaking parts of the system. In that way, the approach combines elements from formal methods and agile practices like test-driven development. As things became more stable, we also started proving a subset of the properties, most of them in a traditional, pen and paper style, and some also formally in Isabelle.

This approach requires two techniques that are not stock items in the repertoire of software engineers: formal specifications, and efficient use of property based testing[11]. We organised a one-week intensive on-site training course in those techniques for our engineers to make up for that, run by Well-Typed, QuviQ, and the IOHK education department. The course was very well-received, and our engineers report that programming from executable specifications was a very pleasant experience.

Here is a list of the things that we found worked well, or not so well:

+ The language of transition rules in a small-step operational semantics formed a lingua franca to talk about the ledger within the company. While it might look intimidating when unfamiliar, we found that after a little bit of introduction to the framework, we could use it to communicate not only with engineers, but also other stakeholders within the company (researchers, product management, and the CEO). Subtle questions from the researchers were easier to answer by looking at the formal spec than by looking at code. Additionally, we received a lot of very helpful feedback from our auditors, concerning details in the specification.

[11] While the use of property based testing has surged in recent years, with QuickCheck clones available in most languages, experience in *efficient* use, including writing good generators and shrinkers, is not common.

- + The simple mathematical style of the small-step operational semantics translated extremely well to Haskell. Comparing the two specs side-by-side is very easy to do, therefore strengthening our trust in the translation from paper to machine.
- + In most cases, small-step semantics combined nicely with agile methodology. Adhering to the two principles we stated above encouraged us to maintain the right granularity in our rule definitions. That is, in response to a requirement change, the scope of the type-level changes and associated semantic rule changes was often contained to a single transition type and an (often singular) relevant rule, or at least easily traceable over several rules. Because of this, implementing incremental local changes could be done in a rather a non-disruptive and tractable way. Note, however, that even with this approach, not all small local changes can be made to be non-breaking.
- + Flexibility with confidence, through the type-checker and QuickCheck.
- + Extensibility: even before the first version of the ledger was finished, we had one team member work on integrating the next feature, integration of the smart contract language Plutus, on the level of the specification. This required adding some new types, some new transition rules, and modifications to a few existing rules. We expect a massive reduction in lead times for future features.
- + The formal spec made the job of estimating the work required to implement new features much easier than it would have been with code alone: the spec provided a view on the system that had enough detail to see which parts would have to be changed in order to implement a new feature, while still being concise enough to quickly analyse the impact on the whole system. Similarly, when integration issues made us consider the impact of refactoring, the formal spec was valuable for choosing the path forward.
- - We had to keep two versions of essentially the same document – the semi-formal and executable specification – in sync. Performing formal proofs in Isabelle required yet another version of the same specification. Ideally, we would like to derive all of those documents from one single source. While there are some tools available (such as lhs2tex[12], Ott [SZNO+10], and Lem [OBZNS11,MOG+14]), we chose to do this manually on the first project, for pragmatic reasons: we did not have enough time to research the existing tools sufficiently to convince ourselves that we would not run into limitations along the way. We intend to improve this, by evaluating existing tools, and possibly modifying one, or even writing our own.

Consensus Layer The consensus layer determines who is allowed to produce a block at which point in time. It is based upon Ouroboros [KRDO17], the first provably secure PoS protocol, and variants [DGKR17,BGK+18]. Ouroboros guarantees – as long as more than half of the participants (weighted by their stake) behave according to the protocol – that transactions submitted to the network will be included in the ledger, and that the ledger stabilises, so that

[12] http://hackage.haskell.org/package/lhs2tex

transactions can not be dropped after they have been in the ledger for a certain amount of time. Having those guarantees for Cardano requires a faithful implementation of the consensus protocol.

Unavoidably, the consensus protocol inherently involves concurrency, which is notoriously hard to get correctness guarantees about. While we do want to ultimately get a high-assurance implementation of Ouroboros, we decided that going for that right away was too risky in terms of development time.

So again, we chose an approach of two development streams, with different speeds and levels of formality. But we took a lesson from the past, and asked very experienced and disciplined engineers to do the initial implementation. They would produce code that was well documented, designed with testability in mind, modular, and solid. They would use prototyping to make informed design decisions. Rigorous code review, direct communication with the Ouroboros authors, and extensive property based testing would ensure that the resulting code was of high quality. Extensive use of polymorphism and Haskell type classes was essential in achieving a flexible and testable design (more on that in *Integration* below).

To eventually get the extra bit of assurance that comes with a formal model and proofs, a second group of people is following their traces, and is modelling the resulting design formally in Isabelle, using a process calculus. They should then be able to provide machine-checked proofs about the correctness of aspects of the implementation, or providing a basis for re-implementing parts of the consensus to build on the fully formal core. As a first step towards this goal, we have developed a custom process calculus [Jel19] and proved some properties relating message relaying and broadcast based on it.

The advantage of our approach is that we do not have to make an up-front decision about the final level of formality, but can defer this decision to a point where we have a better understanding of the complexity of the endeavour. The code that we do have is robust enough stay in production for the lifetime of the system. Every step that we go on the formal side increases our confidence in the design, and thus is not wasted, regardless of whether we will go to an actual implementation derived from the formal model. We achieve this incremental confidence by first performing proofs about the *design* that we followed in the implementation. Those are relevant for the implementation even if the code is not derived from the formal model underlying the proofs. The next step is modelling the actual implementation formally and proving more elaborate global properties. This is still much less work than an implementation based on a formal model, since we can abstract over many details (in particular, interactions with other layers). The option of ultimately replacing the implementation with code that builds upon the formal model is still open, but even if we decide not to do that, the confidence we gained during the previous steps is not lost.

For instance, one of the proofs that we did concerns the way that chains of blocks are distributed amongst nodes in the system. In the research paper, there is an abstract and perfect notion of a network where every node can broadcast their chain to every other node, and then each node will pick the "best" one according to certain rules. The proofs of the security of the protocol assume this perfect broadcast, but it is not feasible to directly implement this in a real world system; for one, nodes will already agree on a long prefix of the correct chain, so they should only interchange the latest blocks. Also, in a large network, the abstract broadcast will be implemented in terms of communications of each node with a limited number of peers. We have been able to prove that our design for relaying blocks through the network is a refinement of the abstract whole chain broadcast functionality in the paper.

Networking Layer A PoS blockchain cryptocurrency like Cardano is very demanding on the networking side. Ouroboros divides time into discrete slots, and elects slot leaders for the consensus in a pseudorandom manner. For this to work, the next block in the blockchain has to traverse the network from one elected leader to the next leader within the available time, and it must do so successfully in the vast majority of cases. This places a hard real-time constraint on the networking layer. At the same time, the network should be decentralised and permissionless, allowing anyone to join the network. Not only is this in tension with ensuring performance, it also increases the attack surface. Nodes in the system must interact with other potentially adversarial nodes, and the design of this interaction has to enable honest nodes to avoid asymmetric resource attacks, which is not simple in PoS designs[13].

The networking design for Cardano consists of nodes engaging in one-to-one protocols. To reduce complexity, this communication is divided into separate concurrent "mini-protocols", each with a narrow focus[14]. The protocols are designed to ensure that honest nodes can work in bounded resources; they all use consumer-driven control flow for example. The construction of the peer-to-peer network aims to ensure rapid dispersion of information across the network, and limiting an attacker's ability to spam the network, or slow down the network by intentionally delaying replies. We used simulations to verify that our peer selection algorithm, which takes decisions locally, leads to suitable network topologies globally. The peer selection takes into account both the number of hops to disperse information and the network distance of each hop, relying on local measurements of the network distance to available peers in the ΔQ framework [Ree03,DHT99b,LGPC+16,DHT99a,DHST99].

Networking protocols are hard to get right. Reducing complexity by having dedicated mini-protocols for specific tasks was already very helpful, but we also

[13] In PoW systems, there is a distinct computational cost advantage for the honest nodes, in that validating a block is very cheap (just hashing the block) but producing a block requires an enormous amount of computational work by an adversary. In PoS, the computational costs are much more finely balanced and the validation checks require the full ledger state, and thus a closer coupling of the networking layer with the rest of the application.

[14] For efficiency and to aid with network resource management complexity, we use multiplexing to just use one network connection for all protocols between a pair of nodes.

wanted to reason formally about those protocols. To do that, we used *session types*, modeling the communication between two nodes as state machines. We intentionally restricted the admissible communication patterns, so that in each state, one of the nodes could send a message, and the other had to expect and handle any message by the other node. That restriction ensures that there can be no deadlocks (since it there is no state in which both nodes are expecting a message), and also no race conditions (since there is no state where two nodes send messages at the same time). And those powerful guarantees do not have to be proven manually, but are enforced by the Haskell type checker!

Both the network and consensus layers make significant use of concurrency which is notoriously hard to get right and to test. We use Software Transactional Memory (STM) to manage the internal state of a node. While STM makes it much easier to write correct concurrent code, it is of course still possible to get wrong, which leads to intermittent failures that are hard to reproduce and debug.

In order to reliably test our code for such concurrency bugs, we wrote a simulator that can execute the concurrent code with both timing determinism and giving global observability, producing execution traces. This enables us to write property tests that can use the execution traces, and to run the tests in a deterministic way so that any failures are always reproducible.

The use of the mini-protocol design pattern, the encoding of protocol interactions in session types, and the use of a timing reproducible simulation, has yielded several advantages:

+ Adding new protocols (for new functionality) with strong assurance that they will not interact adversly with existing functionality and/or performance consistency.
+ Consistent approaches (re-usable design approaches) to issues of latency hiding, intra mini-protocol flow control and timeouts / progress criteria.
+ Performance consistent protocol layer abstraction / subsitution: construct real world realistic timing for operation without complexity of simulating all the underlying layer protocol complexity. This helps designs / development to maintain performance target awareness during development.
+ Consistent error propagation and mitigation (mini protocols to a peer live/die together) removing issues of resource lifetime management away from mini-protocol designers / implementors.

Integration Having broken the design into components allowed us to parallelise work, which was crucial to reduce development time. Unless done carefully, however, this can lead to a situation where after each component is finished and working in isolation, integration of the components becomes unexpectedly painful and time intensive.

A common way to avoid that situation is to fix, up front, the interfaces between the components, and ensure that every team works against those unyielding interfaces. But this goes against our goal of flexibility: during the design and development process, we might discover that the interfaces we put in place were not ideal, forcing one or more team to work around those imperfections, making their component(s) clunkier, and the whole system more brittle and inefficient than necessary. Conversely, a laissez-faire attitude to the interfaces is asking for trouble during the integration phase. But we can find a middle ground.

For us, the key to avoiding problems with late integration was to perform large parts of the integration at a very early stage, before any of the components was actually finished.

For the consensus/ledger integration, our design puts the consensus in control. It will access functions provided by the ledger layer for things like transaction validation, evolving the ledger state, or querying the distribution of stake between actors in the system (which is relevant for the consensus itself in a PoS system). To achieve an early integration, the consensus layer is developed against a Haskell type class representing an arbitrary ledger, that provides exactly the functions that consensus needs. The result is a consensus implementation that is polymorphic in the ledger.

When we noticed during development that we needed to change that type class, the team was free to do so – after talking to the ledger team to ensure that there would be nothing preventing writing an instance of the new type class for the real ledger.

The benefits of this approach go well beyond avoiding integration pains, though. Being able to swap components proved to be very useful for running demos, and for testing. The ability to demonstrate continuous progress to stakeholders is a key goal emphasised by agile techniques. Performing demo sessions where we could show working code in different stages of readiness – from a mock implementation, to an executable specification/prototype, through refinements of these, up to the final production code – let us achieve this goal.

We used the same technique to improve the testability of our code. Not only could we run tests for the consensus layer before the ledger was ready, by using a mock ledger. We also wrote a mock implementation for the cryptography layer, that would not perform cryptographic signatures, for testing purposes. Not only are tests using the mock cryptographic layer faster and produce test output that is easier to analyse; it also simplified the process of generating and shrinking test cases in property based testing.

To test resilience of the storage layer against file corruption, we wrote a mock implementation that would simulate a file system. Not only did that allow us to run those tests consistently and reproducibly, it also allowed us to increase the frequency of file system errors during tests, to find bugs during testing that would occur only after years of running in production otherwise.

Finally, being polymorphic in the ledger allows IOHK to reuse the codebase for other blockchain-based products.

+ Avoids both late integration pains and the inflexibility that comes with setting interfaces in stone up front.
+ Better testability: tests can be performed independently of other components. That allows us to run them before those components are ready, can make tests run faster, and test output easier to understand.

+ Continuously assessing progress: we could run an early demo session using mock components, use an executable specification (that would already have the real logic, but might not be efficient, not feature persistence, etc) in another demo, and plug in the production implementation when ready.
+ Facilitates code reuse in other projects.

Upcoming Features: Smart Contracts Languages Plutus and Marlowe In IOHKs forthcoming smart contract offering Plutus, formal methods have been involved from the outset. Aspects of the design have been prototyped first in Agda before implementation in Haskell [PJGK+19]. This is because the Agda type system and its interactive programming environment provide greater assistance to the programmer that help speed up development on certain tasks. Building on the methodology described in this paper, Plutus Core (the compilation target for the Plutus language) has an executable specification written in Agda [CKNW19]. Plutus is a general purpose language for designing smart contracts that is closely related to Haskell. It is complemented by the Marlowe [LST18] language, a domain specific language specifically targetted at financial smart contracts. In Marlowe, formal methods also play a crucial role; Marlowe programmers can use builtin support for static analysis when programming [IOH]. This functionality makes use of the Z3 SMT solver [DMB08].

5.2 Lessons

We have learned several lessons from our experience.

Lesson 1: Flexibility. One key lesson is about flexibility. By adopting an agile mentality and by using *suitably lightweight* formal methods – most notably, executable specifications in a functional language with a strong type system, and property based testing via QuickCheck, we have been able to quickly and effectively incorporate design changes, even at a late stage in the implementation process, without either breaking code or restarting the development process. Using the type system to bank the consensus between teams - type classes being especially useful in this respect - proved to be an efficient technique for retaining flexibility in a large scale project.

Lesson 2: Communication. A second major lesson that we have learned is about communication. Agile methods are effective partly because they are designed to ensure good internal communication within a team (this may break down in practice, of course), but also, less obviously, because they naturally improve external communication. Agile methods are effective precisely because the results of the development process are visible externally: there should always be a workable fallback once the MVP is produced, and it is easy to evaluate the differences between the current status of the product and what is wanted/needed (the feature list).

In contrast, not all formal methods are suitable for continuously communicating progress. Formal methods development may suffer from a lack of transparency for several reasons. The dense and difficult to parse (for a human) proofs and specifications result, internally, in uncertainty about the amount of effort required to bring them to completion. This is reflected in external communication as well, as it is more difficult to communicate the current state of the formalization to those without a formal methods background.

By enforcing better communication (both internal and external), including by providing regular measurable results, it is possible to bring software projects to a quicker, more successful conclusion, without compromising on software quality. We found prototypes and demonstrations, based on executable specifications, and a refinement approach to development, to be very helpful here.

Lesson 3: No "Big Bang". A third, related, lesson is about iteration. Rather than saving results until a formal process is finished, it is important to share intermediate results, even if they are not fully worked out. This has the key benefit of demonstrating progress, but also has the advantage that it is possible to obtain constructive feedback, that can then be incorporated into new designs and implementations. Sometimes, this reveals that some planned work is not actually necessary, or that some part of the design or implementation can be eliminated, because it is no longer required, or of reduced interest. This is, of course, part of a good agile approach. Refinement-based or gradual approaches, where abstractions are made increasingly concrete, can be highly effective. An advantage is that refinement can be stopped and restarted at any point. By connecting the formal refinement process with software equivalents, high assurance prototypes or demonstrators can be produced, with details left to be implemented at a later date.

Lesson 4: Ensure Consistency. A fourth lesson relates to testing and verification. By using a formal approach, it is easy to demonstrate consistency between the design and the implementation. Formal properties can be derived, either manually or directly from a specification, that can then be used as part of a methodical property-based testing approach, e.g. QuickCheck [CH00] or Hedgehog (`https://hackage.haskell.org/package/hedgehog`). At IOHK, we manually translated the required formal properties into property-based tests. The same properties can be used to support formal proofs, to drive a model checker or some other formal verification technique. It is not necessary to use multiple techniques to verify the same property, but this can give higher assurance. For example, a property can be manually proved to be sound, an automated proof can be produced based on this, and assertions can be introduced into the code. Since properties are derived systematically from the specification, effort can be focused on the most important issues.

Lesson 5: Maintain Progress. A final lesson relates to diversion of effort. By maintaining focus on the end goal of the software development process, as required by good agile development methodologies, we can avoid diverting effort to short-term fixes that have no long-term benefit. For example, by prioritising the properties that need to be proved or tested, we can avoid wasting effort and so maintain progress towards the most important goals.

5.3 Flexible Formal Software Engineering

Our flexible software engineering methodology is made of up the following essential activities which together comprise a full development cycle. These may apply at different levels of the classical software lifecycle (requirements, design, implementation, testing, deployment etc.).

Gather Informal Requirements. Start with a good understanding of the problem, and describe the solution in an informal but unambiguous way. Note however, that it is not always the case that *all* requirements can be captured beforehand. It is permissible to add requirements iteratively (see *Iterate* and *Redesign* below).

Isolate and Abstract. Consider how the functionality can be made modular. Divide the problem into non-overlapping but interacting parts, figure out what is required from each of them.

Generate Semi-Formal Specification. For each component, develop the informal requirements into a semi-formal specification with an appropriate choice of denotational or operational semantics.

Identify Properties. Identify important properties that the software should have. State these precisely and formally. Prove the most important properties. Other properties can be used either as the basis for formal verification, for property-based testing, or for normal unit testing etc.

Build the Executable Specification. Produce an executable specification. By writing our implementation in Haskell, it was possible to maintain a high degree of consistency between the design and implementation.

Iterate. Work iteratively. Refine the system design to add more detail, verifying that these details do not violate the required properties. By using an executable specification approach, it is possible to ensure that a working prototype is always available.

Redesign. Maintain design flexibility. Use suitable levels of abstraction (e.g. in Haskell, type classes or polymorphic types), so that alternative implementations can be produced. Feed new or changing requirements into the design and implementation process.

Prove, Test and Verify. Apply the right technology (manual and automated proof, automated testing, etc.) to obtain the required assurances in the correct operation of the software.

Communicate. Hold regular meetings to discuss progress, focus design and implementation effort, discuss technical issues, and ensure that the team is aware of each other's activities. Encourage all team members to express concerns, suggest ideas, or to ask for technical help. Hold regular detailed technical seminars to discuss new techniques or to investigate specific issues in detail. Make sure that results are communicated throughout the organisation (it may be necessary to use different techniques for this – senior management is unlikely to read detailed soundness proofs, for example) and that input is taken.

By combining the best features of both agile and formal software development, we can obtain significant advantages over either approach used independently. Functional programming technology is, of course, critical to achieving this. Functional programming naturally supports many lightweight formal methods, including advanced type mechanisms such as dependent types, session types etc. Higher-order functions provide excellent abstraction mechanisms, and enable flexible design and implementation. Formal proofs are much easier to relate to implementations in a functional style. High levels of abstraction mean that it is easy to maintain consistency between the design and implementation. Properties are easy to relate to software, and there are good property-based testing systems. Software is concise, can often be executed interactively, prototypes can easily be produced and demonstrated. Effects can be isolated and contained using well-understood structuring mechanisms. The semantic gap between specification and implementation languages is typically much smaller when using functional languages.

Issue	**Agile**	**Formal**	**Flexible**
Identify the requirements for the software	Y?	Y	**Y**
Ensure that the software meets these requirements	Y?	Y	**Y**
Provide usable prototypes rapidly	Y	Y?	**Y**
Minimise the costs of development	Y?	N?	**Y**
Ensure that code is high quality	N	Y	**Y**
Ensure that software is easy to use	N	N	**N**
Ensure that changes can be made easily	Y	N	**Y**
Be easily applied without extensive training	N	N	**?**

In short, we have found that we can obtain major practical benefits from our approach in terms of both the speed of development and the quality of code that is produced.

6 Related Work

There is a vast literature on software development, and an equally vast literature on formal methods. The potential for interaction between the two has not gone unnoticed: the annual *Formal Methods for Software Engineering* conference publishes a regular collection of the latest formal methods techniques and suggests how they might be deployed in practice. Software engineering has moved away from classical "Waterfall" development towards "Agile" development. This means a move away from a rigid specification-design-implement-test-debug-deploy cycle towards a more flexible approach where phases are intermingled and a software development team can work in a less hierarchical way.

In many ways, combining the best of both approaches is more of a philosophy. It reflects how actual software engineering has always been practiced, but encourages better internal and external communication, earlier product release, and ideally responsiveness. *Continuous testing* [AD14] using automated frameworks is a key part of the corpus: no software should be committed without being tested against the recognised test suite. *Continuous integration*, where changes are continually applied to a master version, is also key to the success of an agile approach, ensuring that fixes and improvements can quickly be made available to end-users. In the most ambitious projects, this can result in daily, or even more frequent, software releases.

"Lean" software development [PP03] is one of the more extreme forms of agile development. Here, the focus is on strong product design and minimising wasted effort. The goal is to produce a "Minimal Viable Product" as quickly as possible. This requires very high levels of discipline: it is necessary to avoid deviating from the most important goals, to avoid adding unnecessary features, to test adequately but not excessively, and to quickly adapt to changing goals.

A noteworthy body of research is the work on *Cleanroom Software Engineering* [HLT94] from the 80-90s, an incremental engineering approach, which makes use of specifications and refinements, and a sophisticated statistical model-based testing approach.

What is less common is the recognition that *functional programming* techniques can play a key part in agile software engineering. They are the glue that holds together the *flexible* software engineering methodology that we have described above, and that enables us to quickly incorporate appropriate lightweight formal methods, while maintaining high levels of flexibility. By building on well-understood, malleable and abstract functional components, we can quickly and easily refine designs, use existing components as part of a new design, and the discipline that is imposed by strong type systems means that we can have a high degree of confidence in the correctness of any software that is released.

Safety critical systems are the more traditional application area of formal methods, as errors in software for these systems can have grave impact, potentially causing accidents and hurting or even killing people. At the same time there is a strong pressure to realize more and more functionality in software which makes agile development approaches attractive for critical systems.

One research project in critical systems was the openETCS[15] project from the rail domain. The project developed a toolchain for ETCS (European train control system) which supports agile development combined with formal methods [openetcs-miv07]. Another research project which investigated this was the Open-DO[16] project from the avionics domain. In the hi-lite[17] subproject, there was considerable tool development for making the use of formal methods easier [hilite-L5.3], in particular by automating large parts of the formal proof effort. Increased automation allows for more frequent changes by reducing the required work on the formal model and proof part.

Even for interactive theorem provers, this now allows for proof-replay, automated proof-finding [CK18] and counterexample detection [BN10]. There have also been considerable formal verification efforts at the operating system level [KEH$^+$09].

[15] `www.openetcs.org`

[16] `www.open-do.org`

[17] `http://www.open-do.org/projects/hi-lite/`

The microkernel seL4 is both high-performance and extremely thorough in the depth of its formal verification, which includes the compiler, assembly code, and hardware. Moreover, it follows a similar iterative cycle of prototyping, formal specification, verification, requirements adjustment, all the while reflecting the changes in the actual implementation. The actual implementation, however, is manually derived from the prototype and specifications. This is potentially a source of incongruety between the specification and implementation, which is different from our approach of the specification and implementation being one in the same. These examples of formal specification approaches are quite specific to their application domains.

Within the last few years, there have been calls to action to devise a methodology that combines agile and formal methods approaches in a general way (see [Ghe18]), but the specifics of the methodology of producing such a piece of software are not well-documented. The approach we present is universal. As there is currently no standard or regulation for development of cryptocurrencies, there is more freedom in our domain. Regardless, the approaches do share automation as a common topic.

7 Conclusions

This paper has described the approach to complex software engineering that has been successfully deployed at IOHK for the construction of a new distributed blockchain. The Cardano system is designed to support large-scale, verifiable transactions in a decentralised way, without requiring the inefficient PoW consensus mechanism that is used by e.g. BitCoin. The flexible formal software development approach that we describe in this paper combines the speed and visibility advantages of agile software development with the correctness advantages of formal methods development, while also delivering additional new advantages in terms of the ease of design change. This approach codifies our own experience, as well as that of others at the many companies that are using functional programming and formal methods as part of an integrated software development approach.

The key take-aways from our experience honing a software development methodology that fuses formal and agile approaches in a flexible way are:

(i) it is imperative to maintain a disciplined and structured approach to prototyping, implementation, verification, and testing, as the foundation of the development (no ad-hoc solutions!)

(ii) maintaining transparecy (e.g. decision tracking), explainability (both external and internal), clarity of requirements, and good communication during the development process as well as the deployment cycle are key to the success of our approach

(iii) if we faithfully adhere to these principles, we will be rewarded with the benefits of both formal methods and the agile approach to engineering, which means high-assurance software that is fast to deliver and amenable to changing requirements

7.1 Possible Improvements

We will continute to evolve our methodology, based on our experience in developing Cardano and future projects. Below, we list some concrete improvements we will be pursuing. Firstly, in certain places, we failed to use the right abstractions in our code. Refactoring the code to change properties on Haskell type classes was time-consuming, for example. In hindsight, greater abstraction would have allowed more flexibility and saved overall development time. Secondly, we could and should have produced more prototypes and demonstrators. There was a tendency for the team to hold back until software was correct rather than when it was working, which could be perceived as a lack of progress. We could also have achieved better visibility of our results both internally and externally (for example, some documents could be hard to find, more blog posts could have been written, more interviews given, etc.).

Thirdly, we produced our executable specification and tests manually from the formal specification. It would have been more efficient and provided greater confidence in their consistency if we had instead produced the executable specification and property-based tests directly from the formal specification. We are not aware of suitably robust tooling that would allow us to do this, unfortunately, but we would welcome any suggestions and future developments.

7.2 Acknowledgements

We like to thank to Charles Hoskinson for his committment to using a formal approach for the flagship product of IOHK. This is a bold step in a young and fast moving industry, and we are grateful for his confidence. We would also like to thank our friends and colleagues at IOHK and elsewhere who have contributed ideas and suggestions for this paper but who are not listed as authors. This includes several software developers from other organisations, who have commented on their own experiences, but who wish to remain anonymous.

References

AD14. W. Ariola and C. Dunlop. *Continuous Testing*. CreateSpace Independent Publishing Platform, USA, 2014.

BBvB$^+$01. K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for agile software development, 2001. URL http://www.agilemanifesto.org/.

Bec00. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

BGK$^+$18. C. Badertscher, P. Gazi, A. Kiayias, A. Russell, and V. Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. Cryptology ePrint Archive, Report 2018/378, 2018. https://eprint.iacr.org/2018/378.

BKKS18. L. Bruenjes, A. Kiayias, E. Koutsoupias, and A.-P. Stouka. Reward sharing schemes for stake pools. Computer Science and Game Theory (cs.GT) arXiv:1807.11218, 2018.

BN10. J. C. Blanchette and T. Nipkow. Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder/. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving*, pages 131–146, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

Bra16. E. Brady. *Type-driven Development With Idris*. Manning, 2016. URL http://www.worldcat.org/isbn/9781617293023.

Bre15. E. Brechner. *Agile Project Management with Kanban*. Microsoft Press, Redmond, WA, USA, 1st edition, 2015.

CC77. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. https://doi.org/10.1145/512950.512973.

CGP99. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

CH00. K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. https://doi.org/10.1145/351240.351266.

CK18. Ł. Czajka and C. Kaliszyk. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning*, 61(1):423–453, Jun 2018. https://doi.org/10.1007/s10817-018-9458-4.

CKNW19. J. Chapman, R. Kireev, C. Nester, and P. Wadler. System F in Agda, for Fun and Profit. In G. Hutton, editor, *Mathematics of Program Construction*, volume 11825 of *Lecture Notes in Computer Science*, pages 255–297, Cham, 2019. Springer International Publishing.

DGKR17. B. M. David, P. Gazi, A. Kiayias, and A. Russell. Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. *IACR Cryptology ePrint Archive*, 2017:573, 2017.

DHST99. N. Davies, J. Holyer, A. Stephens, and P. Thompson. Generating service level agreements from user requirements. In *The Management and Design of ATM Networks*, volume 5, pages 4/1 – 4/9, 12 1999.

DHT99a. N. Davies, J. Holyer, and P. Thompson. End-to-end management of mixed applications across networks. In *IEEE Workshop on Internet Applications*, pages 12 – 19. IEEE, 09 1999.

DHT99b. N. Davies, J. Holyer, and P. Thompson. An operational model to control loss and delay of traffic at a network switch. In *The Management and Design of ATM Networks*, volume 5, pages 20/1 – 20/14, 12 1999.

DMB08. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages

337–340, Berlin, Heidelberg, 2008. Springer-Verlag. URL http://dl.acm.org/citation.cfm?id=1792734.1792766.

Dou02. J. R. Douceur. The sybil attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260, London, UK, UK, 2002. Springer-Verlag. URL http://dl.acm.org/citation.cfm?id=646334.687813.

FM-TR-2018-01. IOHK Formal Methods Team. Small Step Semantics for Cardano, IOHK Technical Report FM-TR-2018-01, 2018. URL https://github.com/input-output-hk/cardano-chain/blob/master/specs/semantics/latex/small-step-semantics.tex.

Ghe18. C. Ghezzi. *Formal Methods and Agile Development: Towards a Happy Marriage*, pages 25–36. Springer International Publishing, Cham, 2018. https://doi.org/10.1007/978-3-319-73897-0_2.

hilite-L5.3. Hi-Lite Team. Hi-Lite - Simplifying the use of formal methods, 2013. URL http://www.open-do.org/wp-content/uploads/2013/05/hilite-L5.3.pdf.

HLT94. P. A. Hausler, R. C. Linger, and C. J. Trammell. Adopting cleanroom software engineering with a phased approach. *IBM Syst. J.*, 33:89–109, 1994.

IOH. IOHK Marlowe Team. URL https://testnet.iohkdev.io/en/marlowe/tools/marlowe-playground/.

Jel19. W. Jeltsch. A process calculus for formally verifying blockchain consensus protocols. To appear, Nov. 2019. URL https://github.com/jeltsch/wflp-2019/tree/master/Paper.

KEH$^+$09. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery. https://doi.org/10.1145/1629575.1629596.

KKL18. D. Karakostas, A. Kiayias, and M. Larangeira. Account management and stake pools in proof of stake ledgers, 2018.

KRDO17. A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Advances in Cryptology – CRYPTO 2017*, volume 10401 of *Security and Cryptology*. Springer International Publishing, 2017. https://doi.org/10.1007/978-3-319-63688-7.

LGPC$^+$16. S. Leon Gaixas, J. Perello, D. Careglio, E. Gras, M. Tarzan, N. Davies, and P. Thompson. Assuring qos guarantees for heterogeneous services in rina networks with Δq. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 584–589. IEEE, 12 2016.

LST18. P. Lamela Seijas and S. Thompson. Marlowe: Financial contracts on blockchain. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, volume 11247 of *Lecture Notes in Computer Science*, pages 356–375, Cham, 2018. Springer International Publishing.

MOG$^+$14. D. Mulligan, S. Owens, K. Gray, T. Ridge, and P. Sewell. Lem: Reusable engineering of real-world semantics. *ACM SIGPLAN Notices*, 49, 08 2014.

Nak09. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

OBZNS11. S. Owens, P. Böhm, F. Zappa Nardelli, and P. Sewell. Lem: A lightweight tool for heavyweight semantics. volume 6898, pages 363–369, 08 2011.

openetcs-miv07. Klaus-Rüdiger Hase. openETCS: model-based, agile and open-source, 2016. URL http://www.schienenfahrzeugtagung.at/download/PDF2016/MiV07_Hase.pdf.

PF01. S. R. Palmer and M. Felsing. *A Practical Guide to Feature-Driven Development*. Pearson Education, 1st edition, 2001.

PJGK$^+$19. M. Peyton Jones, V. Gkoumas, R. Kireev, K. MacKenzie, C. Nester, and P. Wadler. Unraveling recursion: Compiling an ir with recursion to system f. In G. Hutton, editor, *Mathematics of Program Construction*, volume 11825 of *Lecture Notes in Computer Science*, pages 414–443, Cham, 2019. Springer International Publishing.

PP03. M. Poppendieck and T. Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

Ree03. D. C. Reeve. *A New Blueprint for Network QoS*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK, August 2003. URL http://www.cs.kent.ac.uk/pubs/2003/1892.

SB01. K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

SL-D1. IOHK Formal Methods Team. Design Specification for Delegation and Incentives in Cardano, IOHK Deliverable SL-D1, 2018. URL https://github.com/input-output-hk/cardano-ledger-specs/tree/master/docs/delegation_design_spec.

SZNO$^+$10. P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20:71–122, 01 2010.