Applying Continuous Formal Methods to Cardano (Experience Report)

James Chapman
Input Output
Glasgow, UK
james.chapman@iohk.io

Arnaud Bailly
Input Output
Nantes, France
arnaud.bailly@iohk.io

Polina Vinogradova
Input Output
Ottawa, Canada
polina.vinogradova@iohk.io

Abstract

Cardano is a Proof-of-Stake cryptocurrency with a market capitalisation in the tens of billions of USD and a daily volume of hundreds of millions of USD. In this paper we reflect on applying formal methods, functional architecture and Haskell to building Cardano. We describe our strategy, projects, lessons learned, the challenges we face, and how we propose to meet them.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Keywords: Agda, Formal Methods, Software Engineering, Cardano, Distributed systems verification

ACM Reference Format:
James Chapman, Arnaud Bailly, and Polina Vinogradova. 2024. Applying Continuous Formal Methods to Cardano (Experience Report). In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Software Architecture (FUNARCH '24), September 6, 2024, Milan, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3677998.3678222

1 Introduction

The Cardano network is a global decentralised network consisting of several thousand block-producing nodes and thousands more relay nodes, that are run by a diverse, decentralised group of individuals and entities on the public internet.

The environment is challenging and realistic testing is difficult. Moreover, mistakes are potentially very costly and hard to rectify. For this reason rigorous practises such as those suggested by formal methods are essential so that high quality design and engineering are built in from an early stage and carried through the development process. This approach has an exemplary period of stability for the system with 6 years of very little downtime. It is also notable that we have a small formal methods team - usually a single FM Engineer per development team of around 6. The work described here is the work of 6 teams.

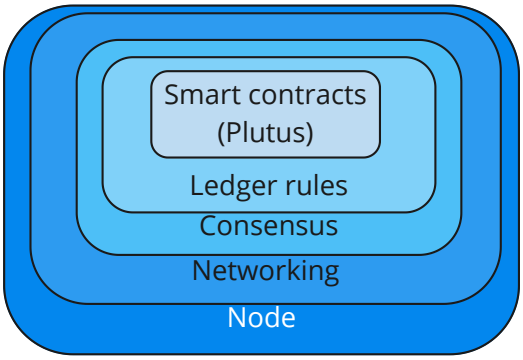

Figure 1. Cardano node layers

The software that runs on a Cardano node is written in Haskell and was developed by Input Output. The node is separated into a number of layers (see figure 1). The layers proceed outwards from a pure internal core to an impure and outward-facing outer layer.

The smart contract layer's behaviour is not only pure, but it is predictable from only a summary of the contents of the executing transaction, independent of the global ledger state. The ledger is pure, consensus involves concurrency, and networking has to deal with the complexity of unreliable communication.

Additional off-chain components have sufficiently robust formal, statistical, and incentives-based guarantees for the Cardano system to rely on them in its operation. This includes systems like Mithril, which provides a way to quickly bootstrap a node without replaying the entire chain, and Hydra, which is a secure transaction settlement mechanism updating an on-chain contract with summaries of recent (Hydra) transactions on a regular basis.

This structure supports a separation of concerns which pushes complexity to the outside and keeps the core as pure and simple as possible.

From the formal methods perspective we tune our approach to each component and apply heavier techniques

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FUNARCH '24, September 6, 2024, Milan, Italy
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1101-5/24/09
https://doi.org/10.1145/3677998.3678222

with a greater emphasis on verification to the more tractable inner components and more lightweight approach (type-safety, at minimum) with a greater emphasis on testing to the impure outer components. A key feature of the inner components is designing them with simplicity and building on familiar foundations, which makes them easy to reason about. A key feature of the outer layer is that it is designed with the goal of being easily testable in mind.

Cardano operates on an impressive scale with a high degree of reliability, but there is more to come. Future developments of Cardano will bring technical improvements to scalability and settlement time and bring technical challenges such as supporting greater degrees of decentralisation.

2 Formal Methods Strategy

We have described our earlier experience with the role of formal methods (FM) in Cardano development in a previous paper[9]. Since then, we have refined our strategy in several key ways: it was (i) mechanised, including Agda specifications and conformance testing, (ii) democratised, i.e. made more accessible to a broader audience, including the Cardano community and internal engineers, (iii) industrialised, i.e. has industry-like development practices and standards, and (iv) the scope of application of formal methods was broadened, e.g. to include cryptography.

Designing and implementing a system that handles large sums of money, which must support timely and predictable interaction by users, must run reliably forever, and be decentralised, is a challenging task. The system must be trustworthy and also able to change and evolve. At the same time, no functionality can ever be deprecated at the ledger or smart contract level, since it is required to re-apply blocks at the time of bootstrapping.

The formal methods strategy we follow requires that we get involved early, and continue to stay involved throughout the lifecycle of the project. It also requires us to develop formal artifacts that are tuned to the right level of formality, and are able to grow and change with the project. The *Formal Methods* team at IOG is therefore relatively small, about 6 people, and mostly works *embedded* within other teams.

At the early stage we may be involved in developing a simplified or prototype model. This was the case in integrating smart contract functionality in the ledger, where we developed a bare-bones extended UTxO (EUTxO) model [3] prior to the implementation of the full-scale ledger support for smart contracts. Another early stage activity is up-front performance modelling [7] which can be carried out before creating a prototype to rule out infeasibility early on. The aim here is to ensure that design has realistic performance constraints and does not, for example, require transatlantic network connections that are faster than the speed of light.

We do not consider formal modelling to be useful only during the design stage, or an intervention. Instead we fully embrace that the artifact (e.g. a formal specification) will be adopted and maintained by the engineering team as part of their quality assurance and documentation approach. To support this we follow the rule of thumb that there should be at least one formal methods engineer in each engineering team. We also train other engineers in formal methods so that they are able to understand and work with formal methods artifacts. Special attention must be paid to avoid possible scope creep of our models, as well as to ensure that unnecessary implementation details do not leak into the models. This makes it easier to maintain models and to train new team members on them.

It is possible a specification may, at some point, diverge from its implementation. Also, the specification itself may contain errors. Three artifacts are involved in ensuring correctness of the software: (1) the Agda code, which is verified, (2) the Haskell code generated from Agda, and (3) the hand-written Haskell production code. Equivalence between (1) and (2) is established by contruction, then (3) is verified by running conformance tests, which pass the same input to both the executable specification implementation and the production Haskell implementation, then check that the same output is produced. Test cases are generated using a random or exhaustive generator and also often using a set of handcrafted realistic examples. This ensures that bugs in either the specifications and implementations are likely to be found at the time a commit is made.

For many specifications, the CI also generates (from the literate Agda specification) a PDF document describing the specification, which includes the important data structures, functions, and properties but excludes proof details. This serves as a reference to stake holders, such as core developers, smart contract developers, auditors, and stake pool operators.

We also aim to ensure that the formal model remains relevant and up-to-date throughout development, so that it can be used to provide consistent high assurance guarantees about the project, and also so that any changes of direction or additions can be made with confidence and consistency.

3 The Cardano Node

3.1 Smart Contracts

The low level smart contract language of Cardano Plutus Core is based on System F with iso-recursive types and builtin types, constants, and functions for Cardano-specific features [4, 11]. To reduce transaction size, the type signatures of a Plutus Core program must be erased prior to its inclusion in a transaction. The semantics of the language are given by a small step reduction semantics, which are proven to be equivalent to a more performant (execution of a given contract happens in fewer steps) CEK machine that runs in the production code.

Plutus Tx is a higher-level language that is based on Haskell, and whose compilation target is Plutus Core. Other languages have been developed by the Cardano community that target Plutus Core, but are not based on Haskell.

The formal specification of Plutus Core is written in Agda. It consists of definitions of type checking, reduction, abstract machines (CC, CK, CEK), progress and preservation, and behavioural equivalence proofs. It is executable. This serves as a type-checked specification and also a reference implementation that is used for conformance testing. At the time of development, it also served as a prototype to investigate new proposed design changes. The conformance test suite provides a language-agnostic tool for implementors to guarantee the soundness of custom Plutus interpreters. This is particularly useful in the context of Cardano as it eases the development of alternative high-level languages. For example, language implementors can develop smart-contract testing tools that have strong guarantees and respect on-chain behaviour.

We have also used formal methods to define what it means to have guarantees of soundness of on-chain contract behaviour with respect to *its particular specification* [13]. At this time, only a prototype exists, meaning that the Adga ledger model for which we can reason about (Agda) contract specifications is greatly simplified as compared to the Cardano ledger. Work is in progress to establish a similar framework for Plutus contracts running on on the Cardano ledger, intended for community use.

3.2 Ledger

A block (in the blockchain) consists of a header and a body. The Cardano ledger is concerned with processing the block body, which is made up of a list of transactions. A small-step semantics specification style is used to do this, wherein applying a block to the ledger state is the atomic (small) step [10]. That is, given any ledger state and any block body, a transition specifies when that body is valid in that state, and the updated state to which the block advances the given state.

A block's transactions are applied sequentially, and each must be valid in the state to which it is applied in order for the block to be valid. One of the main components of a UTxO ledger is the UTxO set, which contains all *unspent transaction outputs*, i.e. outputs of previously applied transactions that have not yet been spent by other transactions. Transitions are defined for updating the individual components of the ledger state via transaction application, such as the records of delegation, update proposals, UTxO set, etc. A valid ledger update requires that each component is updated in a valid way by each transaction being applied. For example, to compute an updated UTxO set, UTxO entries spent by the updating transaction are removed from UTxO set, and outputs of that transaction are added.

In early versions of the system, the Ledger Specification was a LaTeX document. It was manually translated into a reference implementation written (and type-checked) in Haskell. The implementation and specification were also kept in sync manually, including inference rules structure, identifiers, comments, etc. An example of the inference rule for updating the UTxO set (as it appears in the LaTeX document) is given in listing 2.

In the latest version of the system, the specification is written in (literate) Agda, from which a LaTeX document is generated. Listing 1 gives one of the predicates in the UTxO rule as it appears in the resulting specification, and the same fragment in the Haskell implementation. A reference implementation in Haskell is generated from the Agda specification, which is then used for conformance testing against the actual (Haskell) implementation using QuickCheck properties.

```
Agda:
  ∀[ (_ , txout) ∈ ʰtxouts .₁proj ]
    inject (utxoEntrySize txout * minUTxOValue pp)
      ≤ᵗ ʰgetValue txout

Haskell:
  {- ∀(_ → (_, c)) ∈ txouts txb, c ≥ (minUTxOValue
     pp) -}
  runTest $ validateOutputTooSmallUTxO pp outputs
```

Listing 1. Code fragments

Interestingly, testing has also evolved alongside the system. Earlier versions generated whole blockchains from scratch. However, good coverage was hard to ensure, and obscure cases were difficult to explore. Newer versions of the generator generate individual ledger states and can be tailored to particular scenarios of interest more easily.

3.3 Consensus

While the ledger works at the level of the *block body*, the consensus layer deals with the block header. It implements

$$
\text{UTxO-inductive}\ \frac{
\begin{array}{c}
\mathit{txb} := \mathsf{txbody}\ \mathit{tx} \qquad \mathsf{txttl}\ \mathit{txb} \geq \mathit{slot} \\
\mathsf{txins}\ \mathit{txb} \neq \emptyset \qquad \mathsf{minfee}\ \mathit{pp}\ \mathit{tx} \leq \mathsf{txfee}\ \mathit{txb} \qquad \mathsf{txins}\ \mathit{txb} \subseteq \mathsf{dom}\ \mathit{utxo} \\
\mathsf{consumed}\ \mathit{pp}\ \mathit{utxo}\ \mathit{txb} = \mathsf{produced}\ \mathit{pp}\ \mathit{poolParams}\ \mathit{txb} \\
\\
\mathit{slot} \\
\mathit{pp} \vdash \mathit{ppup} \xrightarrow[\text{PPUP}]{\mathsf{txup}\ \mathit{tx}} \mathit{ppup}' \\
\mathit{genDelegs} \\
\forall(_ \mapsto (_, c)) \in \mathsf{txouts}\ \mathit{txb}, c \geq (\mathsf{minUTxOValue}\ \mathit{pp}) \\
\forall(_ \mapsto (a, _)) \in \mathsf{txouts}\ \mathit{txb}, a \in \mathsf{Addr}_{\mathsf{bootstrap}} \Rightarrow \mathsf{bootstrapAttrsSize}\ a \leq 64 \\
\forall(_ \mapsto (a, _)) \in \mathsf{txouts}\ \mathit{txb}, \mathsf{netId}\ a = \mathsf{NetworkId} \\
\forall(a \mapsto _) \in \mathsf{txwdrls}\ \mathit{txb}, \mathsf{netId}\ a = \mathsf{NetworkId} \\
\mathsf{txsize}\ \mathit{tx} \leq \mathsf{maxTxSize}\ \mathit{pp} \\
\\
\mathit{refunded} := \mathsf{keyRefunds}\ \mathit{pp}\ \mathit{txb} \\
\mathit{depositChange} := \mathsf{totalDeposits}\ \mathit{pp}\ \mathit{poolParams}\ (\mathsf{txcerts}\ \mathit{txb}) - \mathit{refunded}
\end{array}
}{
\begin{array}{r}
\mathit{slot} \\
\mathit{pp} \\
\mathit{poolParams} \\
\mathit{genDelegs}
\end{array}
\vdash
\begin{pmatrix}
\mathit{utxo} \\
\mathit{deposited} \\
\mathit{fees} \\
\mathit{ppup}
\end{pmatrix}
\xrightarrow[\text{UTXO}]{\mathit{tx}}
\begin{pmatrix}
(\mathsf{txins}\ \mathit{txb} \lhd \mathit{utxo}) \cup \mathsf{outs}\ \mathit{txb} \\
\mathit{deposited} + \mathit{depositChange} \\
\mathit{fees} + \mathsf{txfee}\ \mathit{txb} \\
\mathit{ppup}'
\end{pmatrix}
}
\tag{4}
$$

Figure 2. Ledger specification

Ouroboros Praos[6] consensus rules to determine how new blocks are formed, and whether or not a block can extend the current best chain, depending on the underlying ledger state. Following the successful work on the ledger layer, we are also developing a formal specification for the consensus layer, progressing from a LaTeX specification to a fully mechanized specification.

Like the ledger team, the consensus team has improved their testing approach. They are moving away from simulating a network of interoperating nodes towards an approach of testing a single node in a simulated environment. This enables them to more easily tailor tests to particular scenarios and ensure good coverage.

The consensus layer is based on a series of peer-reviewed consensus protocols. In the future, our plan is to establish a formal link between protocols defined in the research papers and the production implementations. Of course, the implementation only deviates from the papers when well-reasoned and negotiated design refinements are made, such as splitting the blocks into headers (relevant to consensus) and bodies (relevant to ledger).

3.4 Networking

The network layer is specifically designed to be robust against Byzantine adversarial behaviour. It is structured as a set of pull-based so-called *mini protocols* that are multiplexed over a single connection. The various mini protocols' message format and state machine are specified in LaTeX. The Haskell implementation leverages Haskell's type system through *session types* to statically ensure that the implementation follows these protocols precisely.

The networking layer, being critical, is subject to intensive property-based testing. However, because it is leaning heavily on concurrency for efficient operations, testing proved challenging. For that reason, the team has developed dedicated packages which provide an additional layer of abstraction on top of parts of the Haskell runtime system, that is essentially *software transactional memory* concurrency support. The same code can then be executed either by the Haskell RTS, or via a pure and fast implementation which deterministically *simulates* it, enabling excellent testing.

The ΔQ Software Development[7] methodology was used during the design and early development phase of the networking layer. This technique proved invaluable to rule out "naive" designs and investigate various options, providing evidence supporting some critical design decisions like the header-body split diffusion, or parallel download of blocks from upstream peers. The implementation is instrumented so that the behaviour of the system can be compared to the model.

4 Research & Development Projects

Cardano is a research based blockchain. The challenges to the engineering teams are (1) taking research artifacts and turning them into robust implementations; and (2) taking proven properties of protocols and turning them into guarantees about the code.

In the past this effort owed a lot to key individuals working closely with research and development teams. We now have a department dedicated to taking research ideas at an early (1 or 2) technology readiness level[14] and advancing them up to a point where they can be integrated by a production engineering team.

4.1 Babel Fees

Babel fees is an upcoming Cardano platform feature which is designed to enable users to pay transaction fees in multiple *tokens* (ie. crypto-currencies) without any changes to the consensus layer, or requiring users to interact with an on-chain contract. This feature has a number of additional use cases, such as swaps, or the construction of atomic transaction batches. The Babel fees design has gone through the first stages of the pipeline including a research paper [2], a mechanized specification, and a prototype implementation. Due to practical constraints, multiple design refinements had to be made to adjust the design proposed in the research paper to a viable mechanized specification.

This feature is supported by a framework for applying lists of underspecified transactions called *validation zones*. Babel fees supports the settlement of offers which are matched off-chain and placed into validation zones. Research is ongoing into whether validation zones, along with their formal guarantees, are suitable for processing a variety of intents, i.e. underspecified transactions.

4.2 Peras

Peras is a distributed consensus protocol that is an extension to the existing Praos protocol for the purpose of accelerating settlement. It differs significantly from the previous work on consensus due to the fact that prototyping and modelling work was started *before* the research was completed and the paper published. As was pioneered in the Plutus project, we moved from a sequential process whereby engineering work mostly starts after the project is handed over from research, to a more dynamic one where various activities overlap and can influence each other.

Figure 3 depicts the overall process followed by the project, highlighting the interactions between activities and how different artifacts and techniques are being used across the project.

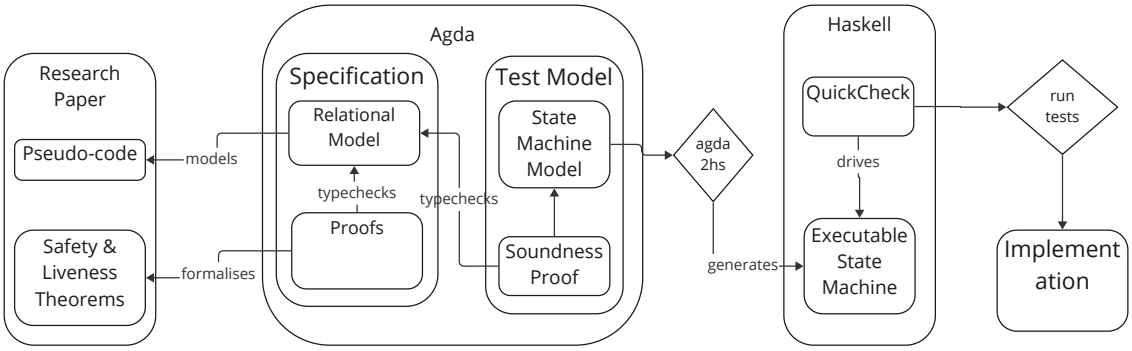

Figure 3. Conformance testing process

The Agda specification is considered as the "source of truth" about the protocol. At its core it is defined as a small-step semantics detailing how an honest node's state is affected by incoming messages. The various types and structures defined in the specification are being used to state and prove properties, using Agda as a pure theorem prover, and formalising proof from the research paper.

The executable specification part is provided by the test model, or rather several test models, which are also coded in Agda, but executed in Haskell after generation with agda2hs[5]. Those models are designed to be run by a QuickCheck-based library[8] with dedicated generators to provide a partial conformance test suite for implementors to base their work on. The soundness of each test model with respect to the core specification is proven using Agda.

ΔQSD is a formal methods technique that has proved quite useful in the Peras project: It allowed us to quickly rule out an initial version of the protocol that would have had too much impact on the timely diffusion of block headers, with minimal modeling effort.

5 Conclusion

Applying FM to cryptocurrency platforms is of particular interest because users trust the platform code with their assets, and once deployed, it is very slow and difficult to update this code. The strategy discussed here has provided essential guarantees about our code, and the peace of mind that comes with that. The successful, zero-downtime operation of the Cardano platform for over five years is a testament to this claim. Our strategy helped establish a common language for communication between researchers and practitioners, and provided us with a principled way of adding new features. It has provided valuable reference material, and taught us a number of important lessons along the way.

5.1 Refining and Interpreting the Strategy

Get involved at an early stage of development As early as possible but not too early is key. Ideally, we should have a coherent idea or relatively stable core to build on. We don't want to redefine carefully designed data structures or proofs too often in reaction to churn in the design.

Develop formal artifacts that can grow and change with the project It is good to start small and gradually expand scope and increase rigour. If we enforce too many of static guarantees, it will be harder to change things later.

Tune the level of formality to the project There is a balance to be struck between customising the approach and tools to the project and standardising in order to reduce duplication and improve knowledge and code sharing.

Lock formal methods into the development workflow via continuous integration/testing This is essential in the long run, but a having a type-checked specification early on is extremely valuable in its own right, even before any connection is made with the production code.

Stay involved during late stages of development and maintenance It is extremely important to do this in order to ensure the system remains consistently reliable across time. For example, as we all know, one should be prepared to throw prototypes away when they deviate significantly from changing requirements.

5.2 The challenge

Formal methods provide significant benefits outlined here, however, there are challenges as well (see [12] for a detailed academic survey).

Maintenance Formal specifications supporting research are usually written once and rarely touched again. On the contrary, software that is in use is subject to change over time as its applications, requirements, features, and technology evolve, and as errors and problems are discovered and fixed. *Legacy software* becomes harder to change over time if specific counter-measures, e.g. *refactoring*, are not put in place. Blockchain is unique in this respect in that all formal specifications of ledger implementations remain relevant in the bootstrapping process. However, we accept that FM tools and standards will change, and plan to further refine our strategy to adapt whenever necessary.

Integration with development practices Software development practices have evolved, and *agile methods* has put a heavy emphasis on both shortening the *feedback loop* and increasing the ability of software to adapt to a *rapidly changing environment*. Formal methods tools and processes can be quite computation- and human resources- intensive even for moderately complex systems. It is still unclear how formal methods can fully benefit from well-known faster feedback techniques like *eXtreme Programming*[1], and how this impacts development of more traditional kinds of software. However, there has been some progress made towards this goal, as described in existing work [9].

Skills shortage Finding and retaining talented Haskell engineers proves challenging, as Haskell still is a *niche* language. The problem is more acute in formal methods: (1) teaching is often not universal or compulsory in standard software engineering or computer science curricula; (2) the formal methods landscape is very fragmented, and previous knowledge is only partially transferable between environments, e.g. a Coq specialist will need time to retrain as an Agda engineer despite the same underlying technology (type theory); (3) people with both industrial and formal methods experience are in short supply; (4) training engineers with a "traditional" background takes time and resources, and most available training courses do not focus on industry, with Summer schools aimed at PhD students being currently the most prevalent.

Tool maturity and efficiency Formal methods tools like proof assistants, compilers, or editors, while usually of great quality, definitely do not benefit from a large enough audience to be on par with what *integrated development environments* provide for mainstream languages. Moreover, they are not as widely supported as other languages by tools and services on which software teams usually rely, like build or continuous integration services.

Silos and Specialists Delivering complex and reliable software systems faster requires whole team communication and collaboration. The aforementioned challenges combine to reduce the ability of every team member to be comfortable working with the formal methods part of the software being developed. This issue leads to silos and team fragmentation, the need for synchronisationwhich in turn leads to longer development cycles because of the synchronisation required. This is improving as we integrate mechanised specifications into the workflow and we improve training.

5.3 Meeting the Challenge

Formal methods is an essential ingredient in any future development for Cardano. As such, we need to ensure formal methods artifacts are considered as first-class citizens in all stages of the R&D process, by all parties involved. This has a number of implications for what we want to focus on next.

Upskilling and knowledge sharing We have already had a successful one-week internal training session in Agda in May 2023, attended by about 30 people from the company, including researchers and engineers. A key aim was to develop a common language between researchers and engineers. The course (which we recorded) is currently available online internally for self study. We want to turn this into a regular event, and make it a part of the onboarding process for all new engineers and researchers working on Cardano. This will be achieved, among other things, by: (1) generalising training courses and offering people from the ecosystem the opportunity to attend them; (2) improving the level of detail and reach of the available documentation and training material; (3) thoroughly documenting the concrete use cases we applied formal methods to, thus providing concrete examples on how to get started on that journey.

Involve researchers earlier in the development process Research within the Cardano ecosystem is very diverse, ranging from programming languages theory, to economics, cryptography, and security. Not all researchers are familiar with formal methods, although all of them share a strong mathematical background as mathematical formalisation is at the heart of most published research. Through training, early collaboration with engineers, and development of domain-specific languages, we want to ensure that machine-checked formalisation happens earlier in the innovation cycle.

Improve tooling Having highlighted how the fragmentation and lack of tool maturity was an impediment to more widespread adoption of formal methods, it is clear that much of our future effort needs to happens there. We have already contributed and will continue to support contributions to various tools: *agda2hs* and *agda2rust* compilers, ΔQ modeling library, *quickcheck-dynamic* model-based testing library, etc. We are also investigating new or improved tooling in areas such as cryptography, where tooling is sparse.

Expanding reach Cardano is an open ecosystem with many independent parties involved. As the ecosystems becomes more diversified and multiple node implementations are created, the significance of machine checkable specifications to ensure compatibility will become increasingly important. We are also working on developing frameworks intended for use by the community, e.g. for smart contract verification. This ties in with our plans to disseminate knowledge and increase the reach of formal methods.

Acknowledgments

In addition to our co-authors from our earlier paper we would also like to acknowledge Mauro Jaskelioff, Ramsay Taylor, Andre Knispel, William De Meo, Ali Hill, Alexey Kuleshivich, Tim Sheard, Damian Nadales, Alexendar Esgen, Nick Frisby, Javier Sagrado, Yves Hauser, Brian Bush, Ulf Norell and Phil Wadler.

References

[1] Kent Beck. 1999. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Publishing Company.

[2] Manuel M. T. "Chakravarty, Nikos Karayannidis, Aggelos Kiayias, Michael Peyton Jones, and Polina" Vinogradova. 2022. Babel Fees via Limited Liabilities. Springer-Verlag, Berlin, Heidelberg, 707–726. https://doi.org/10.1007/978-3-031-09234-3_35

[3] Chakravarty, Manuel and Chapman, James and MacKenzie, Kenneth and Melkonian, Orestis and Müller, Jann and Jones, Michael and Vinogradova, Polina and Wadler, Philip. 2020. *Native Custom Tokens in the Extended UTXO Model*. 89–111. https://doi.org/10.1007/978-3-030-61467-6_7

[4] James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. 2019. System F in Agda, for Fun and Profit. In *Mathematics of Program Construction: 13th International Conference, MPC 2019, Porto, Portugal, October 7–9, 2019, Proceedings* (Porto, Portugal). Springer-Verlag, Berlin, Heidelberg, 255–297. https://doi.org/10.1007/978-3-030-33636-3_10

[5] Jesper Cockx, Orestis Melkonian, Lucas Escot, James Chapman, and Ulf Norell. 2022. Reasonable Agda is correct Haskell: writing verified Haskell using agda2hs. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium* (Ljubljana, Slovenia) *(Haskell 2022)*. Association for Computing Machinery, New York, NY, USA, 108–122. https://doi.org/10.1145/3546189.3549920

[6] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. 2018. Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain. In *Advances in Cryptology – EUROCRYPT 2018*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 66–98.

[7] Seyed Hossein Haeri, Peter Thompson, Neil Davies, Peter Van Roy, Kevin Hammond, and James Chapman. 2022. Mind Your Outcomes: The ΔQSD Paradigm for Quality-Centric Systems Development and Its Application to a Blockchain Case Study. *Computers* 11, 3 (2022). https://doi.org/10.3390/computers11030045

[8] John Hughes. 2016. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 169–186. https://doi.org/10.1007/978-3-319-30936-1_9

[9] Philipp Kant, Kevin Hammond, Duncan Coutts, James Chapman, Nicholas Clarke, Jared Corduan, Neil Davies, Javier Díaz, Matthias Güdemann, Wolfgang Jeltsch, Marcin Szamotulski, and Polina Vinogradova. 2020. Flexible Formality Practical Experience with Agile Formal Methods. In *Trends in Functional Programming*, Aleksander Byrski and John Hughes (Eds.). Springer International Publishing, Cham, 94–120.

[10] Andre Knispel, Orestis Melkonian, James Chapman, Alasdair Hill, Joosep Jääger, William DeMeo, and Ulf Norell. 2024. Formal Specification of the Cardano Blockchain Ledger, Mechanized in Agda. In *5th International Workshop on Formal Methods for Blockchains (FMBC 2024) (Open Access Series in Informatics (OASIcs), Vol. 118)*, Bruno Bernardo and Diego Marmsoler (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:18. https://doi.org/10.4230/OASIcs.FMBC.2024.2

[11] Peyton Jones, Michael and Gkoumas, Vasilis and Kireev, Roman and MacKenzie, Kenneth and Nester, Chad and Wadler, Philip. 2019. Unraveling Recursion: Compiling an IR with Recursion to System F. In *Mathematics of Program Construction: 13th International Conference, MPC 2019, Porto, Portugal, October 7–9, 2019, Proceedings* (Porto, Portugal). Springer-Verlag, Berlin, Heidelberg, 414–443. https://doi.org/10.1007/978-3-030-33636-3_15

[12] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2020. QED at Large: A Survey of Engineering of Formally Verified Software. *CoRR* abs/2003.06458 (2020). arXiv:2003.06458 https://arxiv.org/abs/2003.06458

[13] Vinogradova, Polina and Melkonian, Orestis and Wadler, Philip and Chakravarty, Manuel and Krijnen, Jacco and Jones, Michael Peyton and Chapman, James and Ferariu, Tudor. 2024. Structured Contracts in the EUTxO Ledger Model. In *5th International Workshop on Formal Methods for Blockchains (FMBC 2024) (Open Access Series in Informatics (OASIcs), Vol. 118)*, Bruno Bernardo and Diego Marmsoler (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:19. https://doi.org/10.4230/OASIcs.FMBC.2024.10

[14] Wikipedia. 2024. Technology readiness level — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Technology%20readiness%20level&oldid=1224142038. [Online; accessed 13-June-2024].

Received 2024-06-03; accepted 2024-06-30