Acta Cybernetica **27** (2025) 197–219.

Smart Contract in the Loop: Fault Impact Assessment for Distributed Ledger Technologies*

Bertalan Zoltán Péter[ab], Zsófia Ádám[ac], Zoltán Micskei[ad], and Imre Kocsis[ae]

Abstract

Due to their decentralized and trustless nature, blockchain and distributed ledger technologies are increasingly used in several domains, including critical applications. The behavior of such blockchain-integrated systems is typically driven by smart contracts. However, smart contracts are application-specific software and may contain faults with severe system-level impacts. This is especially true in the case of the extensively used Hyperledger Fabric (HLF) platform, where smart contracts are written in general-purpose languages (Java, among others), and applications can go far beyond handling virtual-currency-like assets. In this work, we present a novel formal-verification-based approach to smart contract verification and a high-level empirical model of the HLF platform. Our *Smart Contract in the Loop (SCIL)* method uses a model checker (Java Pathfinder) to check whether specific error properties hold for a given smart contract, while a predefined combination of platform-level fault modes is active. We facilitate the checking of HLF smart contracts without modification and enable the propagation or non-propagation of platform faults through the smart contracts to the system failure level.

Keywords: distributed ledger technology, blockchain, formal verification, model checking, Java Pathfinder, Hyperledger Fabric

1 Introduction

Distributed ledger technologies (DLTs) – especially blockchains – provide high-integrity distributed databases without requiring a trusted party. Initially developed with financial applications in mind and powering cryptocurrencies, blockchain

*This paper was supported by multiple programs detailed in the Acknowledgments section.

[a]Critical Systems Research Group, Department of Artificial Intelligence and Systems Engineering, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics; Műegyetem rkp. 3, H-1111 Budapest, Hungary

[b]E-mail: bpeter@edu.bme.hu, ORCID: 0000-0002-5577-1369

[c]E-mail: adamzsofi@edu.bme.hu, ORCID: 0000-0003-2354-1750

[d]E-mail: micskei.zoltan@vik.bme.hu, ORCID: 0000-0003-1846-261X

[e]E-mail: kocsis.imre@vik.bme.hu, ORCID: 0000-0002-2792-3572

DOI: 10.14232/actacyb.312501

technology now has a variety of use cases, including supply chain management, healthcare, and telecommunication.

Blockchains & Smart Contracts Blockchains have powerful properties, such as immutability, distribution, decentralization, and high security that make them fit for cross-organizational (enterprise) applications. Where high integrity is paramount, they are already widely used, even in critical applications; e.g., in the nuclear [9] or the railway [14] industry (although, importantly, not in safety-critical functions). Typically, such use cases are backed by permissioned platforms, such as R3 Corda [12] or Hyperledger Fabric [1], but Ethereum [5] can also power permissioned networks. However, where other extra-functional properties, such as timeliness, age-of-information, dependability, or availability are also matters of concern, the system-level analysis of critical applications is still largely an open challenge.

Smart contracts, introduced with Ethereum [5], are akin to stored procedures and describe computations executed on the blockchain with effects that are persisted on-chain. They extend the original accounting "ledger" functionality of permissionless blockchains with rich, self-executing business logic. Smart contracts have since become ubiquitous and are widely used in most blockchain frameworks, enabling decentralized collaboration among the participants.

However, smart contracts are pieces of software and thus susceptible to faults with potentially devastating consequences. The beneficial properties of blockchains may also pose some issues; e.g., even if a bug is found, the ledger's immutability inherently prevents fault removal. Because of these risks, verifying smart contracts has been a central research topic in recent years, bringing about several approaches for fault removal and prevention [22]. These are mostly design-time methods, such as static analysis, and primarily target Ethereum and the Solidity programming language.

Hyperledger Fabric (HLF) [1] is a widely used, mature, enterprise-grade permissioned blockchain platform maintained by Linux Foundation Decentralized Trust (LFDT). It offers pluggable consensus mechanisms, identity management, flexible "subnetting" features, and privacy mechanisms. Fabric powers several projects in both development and production in various domains[1]. In HLF, the network *must* have smart contracts (called "chaincode") for any meaningful transactions to be able to occur.

Lack of Cross-Organizational V&V Support There is significantly less support for verifying enterprise smart contracts, even though recent developments show that the Ethereum Virtual Machine (EVM) is no longer the only available smart contract execution environment; known alternatives include:

- WebAssembly (WASM) [19] (used by Polkadot [23])
- the Berkeley Packet Filter (BPF) [15] VM (used by Solana [24])
- the Move [2] VM (used by the Aptos Blockchain [2])

[1] See the use case tracker at https://www.hyperledger.org/learn/use-case-tracker/.

Enterprise smart contracts necessitate developing different techniques from its public counterparts for several reasons, such as the usage of general-purpose programming languages instead of domain-specific ones to write smart contracts or additional variable features that have to be taken into account in the enterprise case, such as deployment. Further complications arise from the fact that enterprise solutions are often not openly available, lowering the number of available case studies and evaluations, thus hindering research efforts in this area.

One cannot follow the same methodologies for the verification and validation (V&V) of cross-organizational platforms and smart contracts that are already widely available in the literature. The main reason is that while in public platforms such as Ethereum, a canonical set of platform events and relevant attacks can be defined, there is much more variability in these aspects on consortial networks. We explain this notion in detail in Section 2.

Nevertheless, some formal verification approaches can be employed to verify smart contracts both in public and consortial applications [11, 3]. However, to our knowledge, no tooling enables the impact assessment of platform-level faults given an unmodified smart contract implementation, especially for enterprise platforms such as HLF – even though the differences in deployment and the platform greatly influence the possible fault modes. To take all of that into account, verification methods either need to divide all of these components into small parts and verify them separately, or they need to experiment with methods that can handle several of these layers together. We believe that the latter cannot be disregarded, as issues emerging from these systems as a whole must be considered.

While we have little information about cross-organizational smart contracts, as they are typically kept private, we can hypothesize that the majority of them are more or less direct translations of existing contracts from the public blockchain world. Furthermore, as the general-purpose languages used by most consortial platforms were not designed with smart contract development in mind, we also postulate that there are more types of faults to consider for these programs than for those written for the EVM. Unfortunately, we do not have a library of such common faults, as no suitable corpus of consortial contracts is available. At the same time, faults in cross-organizational smart contracts may be more consequential, as the potential damage is not limited to losses in financial assets.

Based on all this, we recognize a lack of V&V methods specialized to enterprise solutions, even though they are necessary based on both the use cases and the individual characteristics of the world of enterprise blockchain platforms.

Contributions & Paper Structure In this work, we propose the application of model checking to show whether a smart contract may develop errors in the presence of certain platform-level faults. To this end, we present a simplified model of the HLF blockchain platform with its primary components and configurable fault modes. This model implementation enables the user to define several aspects of deployment (e.g., the number of peers per organization or the channel's endorsement policy) and specify what faults or attacks can arise.

We demonstrate the viability of our approach with a Java-based prototype capable of simulating network faults in the context of a (hypothetical) safety-critical application. This prototype can be verified using the model checker Java Pathfinder (JPF) [16]. Our method provides the means for one to *plug in* their Java HLF smart contract to the framework and determine whether a predefined property holds while select platform-level faults are active. We dub this approach *Smart Contract in the Loop (SCIL)*. The prototype implementation and all other artifacts related to this paper are open-source and available online on GitHub[2].

In the next section, we briefly overview the application of formal verification to smart contracts and DLTs and what motivates this research. In Section 3, we present our model of the HLF platform, and we describe our Smart Contract in the Loop approach, followed by an overview of our prototype implementation in Section 4, and a worked out case study in Section 5. Finally, we conclude and discuss future work in Section 6.

2 V&V of Cross-Organizational Smart Contracts

Smart contracts are programs that run on blockchains, and as with any other software, they are prone to contain faults ("bugs"). Unfortunately, while traditional software can usually be patched and thus its faults can be removed, smart contracts are inherently immutable; i.e., platforms are typically unprepared to support patching or upgrading these programs because they follow an append-only paradigm. The public blockchain world quickly recognized the need for verification and validation (V&V) activities in the smart contracts development process to prevent these faults from making it into the deployed contracts, proposing several diverse approaches. That being said, cross-organizational (consortial) blockchain applications and smart contracts have unique aspects that warrant different V&V techniques. These different techniques are still largely unexplored.

Smart contract faults may result in the loss of (commonly financial) assets in permissionless systems and the cryptocurrency world (see, for example, the infamous DAO hack [7]). The potential effects are arguably far more devastating in the context of permissioned and especially critical applications. While smart contracts can be enhanced with various defenses (including runtime verification mechanisms or techniques such as n-version programming (NVP) [18]), faults of the platform itself may still induce unintended behavior.

2.1 An Overview of Smart Contract V&V Approaches

Since the initial release of Ethereum [5] and the quick recognition of the need for V&V techniques in smart contract development, hundreds of research papers have been published about various verification tools and approaches. [22] collected 202 papers that are concerned with blockchain V&V techniques in general, such as model checking, theorem proving, program verification, symbolic execution, and

[2] https://github.com/ftsrg/scil

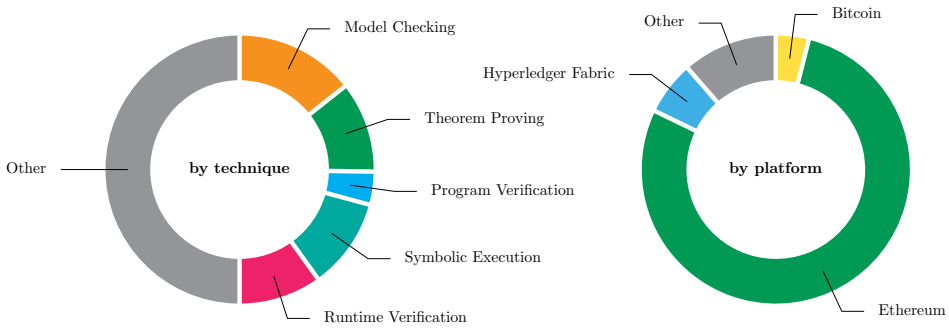

Figure 1: Distribution of V&V techniques in the underlying corpus of [22]

runtime verification. We have summarized the distribution of these techniques among the papers in Figure 1. The diagrams were created by filtering the papers listed at the website[3] created by the authors and counting the results.

It is clear from the results that there are significant research efforts towards smart contract V&V, but methods targeting Ethereum far outweigh those proposed for enterprise platforms. Indeed, Ethereum smart contracts are publicly available, and their common problems are already well-known. On the other hand, enterprise smart contracts are seldom made public, and therefore, we know much less about incidents or common faults in these programs. Furthermore, there are several key differences in cross-organizational blockchain applications that we outline in the following subsection.

2.2 Enterprise and Public Smart Contract V&V Differences

Although not immediately apparent, applications and smart contracts on cross-organizational distributed ledger technology (DLT) platforms may be radically different from their public platform counterparts. The fundamental difference is that while the relevant failure modes and effects in public platforms are fairly canonical, they are much more varied in cross-organizational DLTs. This subsection overviews the most important differences that highlight why the V&V of consortial DLTs forms a different, largely unsolved problem set.

Deployment The deployment model of a consortial DLT and a public blockchain differs. First of all, the infrastructure is typically given and available for smart contracts on permissionless blockchains. Its potential failure modes and their associated risks are known; e.g., selfish mining [10] on Ethereum [5] before The Merge[4], but similar attacks have been identified [17] for the current PoS consensus, too. Conversely, the deployment of a cross-organizational DLT is *application-*

[3] https://ntu-srslab.github.io/smart-contract-publications/

[4] Ethereum [5] switched from proof of work (PoW) consensus to proof of stake (PoS) on 2022-09-15.

dependent. It depends on the number of participating organizations, their relationships, the consensus protocol (especially the endorsement policy in Hyperledger Fabric (HLF)), and the underlying physical infrastructure, among others. Applications based on smart contracts may be affected by these parameters in unforeseeable ways.

Further, some consortial DLTs have capabilities that simply do not exist in the case of public networks. For instance, in HLF's model, smart contracts is installed independently to a number of peers in such a way that it is theoretically possible to have different implementations of the same smart contract specification installed onto different nodes within an organization or even across organizations. This idea is explained in further detail in [18].

Programming Model While there has been a recent, noticeable shift towards other programming languages and execution environments even in the public blockchain space (e.g., Rust in Solana [24], Move on Aptos [2], Python on Algorand [6], etc.), the vast majority of smart contracts have been written for the Ethereum Virtual Machine (EVM) [5] and in Solidity. The common Solidity vulnerabilities, weaknesses, and code smells are known; some examples include reentrancy, arithmetic over- and underflow, frontrunning, and access control [21]. On the other hand, smart contracts on consortial platforms are typically written in general-purpose programming languages. HLF [1], for example, currently supports Go, Java, and JavaScript. Corda [12] smart contracts are written in Java (or Kotlin).

The significance of this is twofold. First, as these languages were not developed with smart contracts in mind, we hypothesize that their usage may imply a more extensive yet unexplored set of potential software faults ("bugs"). As there is practically no publicly available corpus of enterprise smart contract written in these languages, we do not know the statistically most common problems (as opposed to contracts written in Solidity, where sizable corpora exist). Second, while Solidity is still a relatively new and unique language, extensive research has already been done regarding V&V techniques for pieces of software written in ubiquitous programming languages like Java.

Besides the language, the way the world state is stored often also differs; e.g., in HLF, the underlying database is a simple (versioned) key-value store. This greatly affects how smart contracts must be written, especially since serialization and key management issues also become matters of concern.

Execution Model Both consortial and public systems rely on consensus among participants to establish a world state agreed upon by all parties. However, the way this consensus is reached is radically different between the two types of systems.

Finality is a crucial difference; e.g., HLF offers immediate, absolute transaction finality, meaning that a transaction accepted by the network will deterministically end up in a block. Consensus mechanisms in public networks are different. In PoW, finality is *probabilistic:* as the block height grows, an accepted transaction is more and more likely to become final. In PoS, there is *economic* finality: a transaction is

final when "reversing" it would be financially infeasible due to the collateral losses of validators. As a corollary, temporary forks can form on these public platforms, but usually not on permissioned ones (like Fabric).

Another difference is how the smart contract halting problem is solved. In Ethereum [5] and its derivatives, *gas* is used for this purpose. In consortial DLTs, timeout mechanisms are employed since it often does not make sense to track money-like assets on the ledger.

Calls to external services from within smart contracts may be supported and even desired in consortial DLTs, while it is only possible through oracles in Ethereum.

Variability of Platform Events and Failure Effects Since deployment and configuration aspects need not be considered for individual applications, the expected platform-level events, attacks, and failure effects in public networks are fairly canonical and thus can be anticipated. All of these are variable regarding cross-organizational DLTs.

For example, in a HLF network, depending on the endorsement policy, the downtime of a peer may result in unintended behavior even when the smart contracts are entirely fault-free. Or, malicious behavior of a HLF channel's ordering service may also lead to issues regardless of smart contract quality. We detail an example in our case study in Section 5, where the ordering service reorders transactions (a kind of frontrunning attack), resulting in an incorrect final state of the ledger state and a real-world accident occurring as an effect.

Based on the above, we propose that due to the radically different models of execution, programming, and expected failure effects, the V&V of cross-organizational blockchain applications requires different, specialized approaches from those developed for public platforms. In fact, there are several traditional fault-tolerance techniques (such as NVP or runtime verification) that are not practically applicable to public networks (usually because of the added costs) but could be employed in consortial settings.

Because there are so many other "moving parts," one cannot rely on simple, direct V&V of smart contracts (like testing or formal, static analysis). Instead, we suggest a holistic approach where the smart contract is verifiable in the context of the entire target network, including deployment, configuration, higher-level applications (dApps) using on the smart contract as a backend, and any potential defenses. To this end, we have modeled the components of HLF to be able to run simulations. However, instead of requiring the smart contract to be modeled, we have developed a framework (for Java smart contracts) where the contract code can be plugged in *as is*. We elaborate on this framework in the Section 3.

2.3 Related Work

As shown in Subsection 2.1 there has been significantly more research on V&V techniques for public, permissionless platforms. Still, there are some papers aiming at the permissioned HLF platform, too.

In [3], the authors present an approach for the formal verification and deductive verification of HLF smart contracts using the KeY prover. They define the formal specification of Java smart contracts using Java Modeling Language (JML) that is translated into Java Dynamic Logic (JavaDL) [4] and then perform static analysis to ensure the specification's rules are fulfilled. The paper states explicitly that "verifying the correctness of the Fabric framework itself (e.g., communication between peers and orderer) [...] is not within the scope" of their work.

The BCVerifier [20] tool for HLF checks the integrity of the ledger itself to detect local modifications and to ensure transaction executions are valid. A Hyperledger Labs project[5] was created for the tool but has been since archived.

We were unable to find any such literature about other permissioned platforms, such as R3 Corda [12]. The only related research paper is [13], where the authors show a model-driven engineering (MDE) methodology that includes validation.

These existing solutions focus on specific elements of DLT-based applications, such as the smart contracts or the ledger state. The model-checking-based Smart Contract in the Loop (SCIL) approach presented in this paper is more comprehensive in the sense that it does not only verify smart contract correctness or state changes but also considers various platform-level events (faults) and potentially deployed defenses (e.g., smart contract NVP).

3 The Smart Contract in the Loop Approach

As explained in Subsection 2.2, we propose a holistic treatment of cross-organizational distributed ledger technology (DLT) systems for the verification and validation (V&V) of smart-contract-based consortial applications. Concretely, we have developed our Smart Contract in the Loop (SCIL) approach that performs *model checking* of a configurable Hyperledger Fabric (HLF) network model instance, given a smart contract and an error property to check for. We have visualized the core elements of our approach in Figure 2. At the core of our framework, there is an executable HLF model (written in Java), which we describe in the next subsection. In Subsection 3.2, we describe the further components of the SCIL framework for HLF.

3.1 Executable Model of Hyperledger Fabric

Hyperledger Fabric (HLF) [1] is a permissioned, highly configurable, modular enterprise blockchain platform maintained by Linux Foundation Decentralized Trust. Among R3's Corda [12] and Canton [8], it is among the few widely used consortial (cross-organizational) DLT platforms. Fabric is known to power several enterprise use cases[6].

Even with state-of-the-art verifiers, out-of-the-box model checking of a large project, such as the implementation of HLF, is still not feasible due to numerous

[5] https://github.com/hyperledger-labs/blockchain-verifier
[6] See footnote 1.

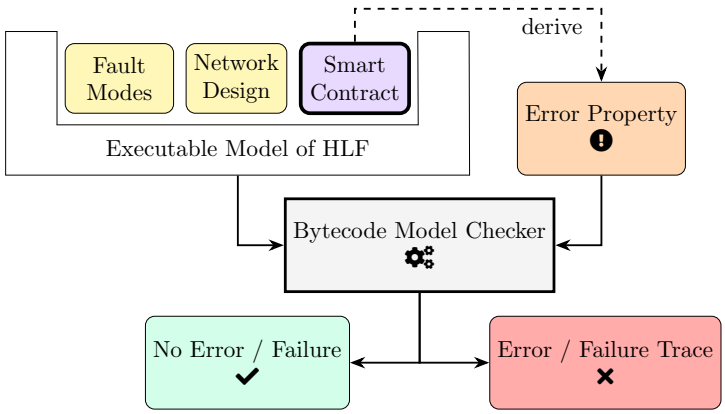

Figure 2: Verification Process

factors, such as scalability issues, libraries, and the distributed nature of the project. Therefore, we have instead created our simplified implementation-independent model of HLF with a level of abstraction that enables meaningful formal analysis but does not generate an overly complex state space.

The difficulty of this approach lies in the empirical nature of modeling the network – verification outcomes are hard to trust on an abstract model based on informal documentation and some code. We identified abstraction as a key point regarding the quality of the model; i.e., finding the right abstraction to catch all relevant aspects to the faults and the error property while keeping the model and its limitations clear.

Please refer to Figure 3 for a high-level overview of our model that incorporates both structure and dynamics (i.e., the messages between the components).

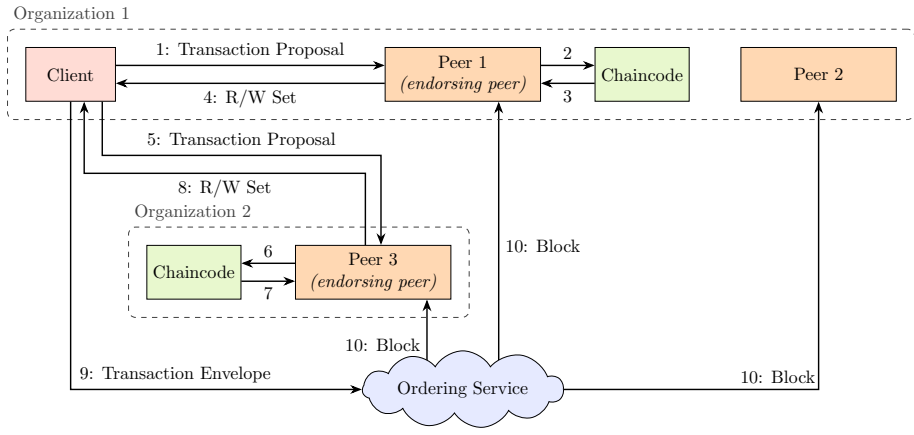

Figure 3: High-level overview of Fabric's architecture

3.1.1 Main Components

In the following, we elaborate on how the main components illustrated on Figure 3 have been modeled, including how they are mapped to the "real" HLF platform's components and their fault modes and interactions with other components.

Unlike common public blockchain platforms such as Ethereum [5], HLF is not designed to provide a de-facto network to be used by participants. Rather, a HLF network comprises several independent channels used by independent consortia. Figure 4 provides a high-level overview of this "consortial architecture."

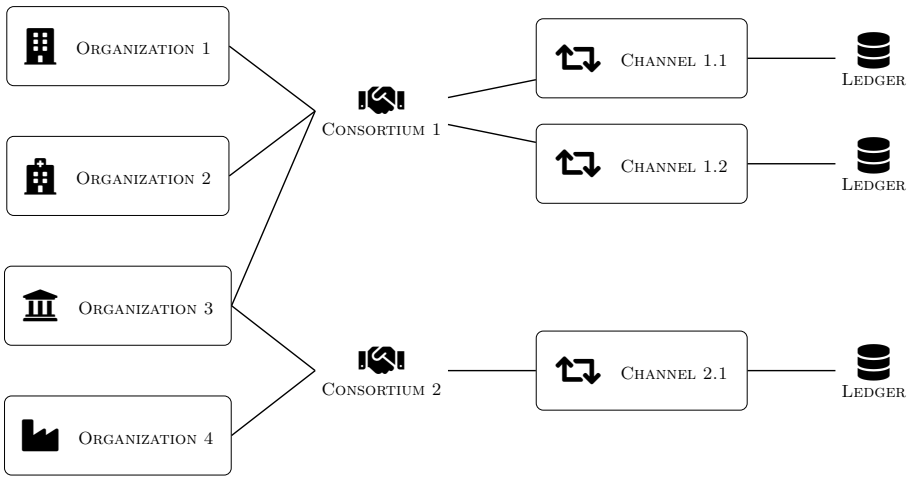

Figure 4: Overview of organizations, consortia, and channels in HLF

Organizations In HLF (and so-called "consortial" platforms and networks in general), the participants are collaborating *organizations.* Organizations form consortia, and each consortium maintains one or more channels.

Most other components, such as peers, orderers, and clients, belong to organizations. Although not explicitly modeled at this stage, it is important to mention that consortium member organizations agree on a per-channel so-called *endorsement policy* that defines the number of organizations required to *endorse* transactions for them to be accepted.

Ordering Service The ordering service is an abstraction formed by all *orderer* nodes in a channel, responsible for establishing a total order over the transactions and creating new blocks. Typically, an independent organization (or several independent organizations) provide ordering services.

We did not model individual orderer nodes at this phase, mainly due to the high complexity of the consensus mechanisms employed during ordering. We did model, however, the critical fault modes of ordering services:

1) **Dropping Transactions** An erroneously or maliciously behaving ordering service may occasionally ignore transactions, refusing their inclusion in new blocks.

2) **Reordering Transactions** Transaction reordering is usually done to perform a *frontrunning*-type attack; i.e., unfairly moving certain favored transactions ahead of others for some business advantage.

Peers Peers maintain the distributed ledgers for the channels they are in. Furthermore, peers receive and simulate client transaction requests and validate blocks published and broadcast by the ordering service.

As we have focused on ordering, we have not yet modeled fault modes of peers, but there are some ways they may misbehave – although the cause of these faulty behaviors would likely not be malicious intent. For instance, a peer may simply become unavailable, either due to issues with its physical host or network infrastructure problems. If the number of reachable peers is insufficient, clients will not be able to gather enough transaction endorsements, and desired state changes may be delayed.

Ledgers Each channel has its own *ledger,* where the world state is being stored. In HLF, the ledger is a simple key-value store. Accordingly, smart contracts (chaincode) are provided a *stub* through which they can read/write values from/to keys (but more complex operations, such as range queries, are also usually available).

Channels Channels group some peers to form a "subnet" in the HLF network with its own isolated and independent ledger. Newly created blocks are broadcast to the peers in the channel.

If endorsement policies were also modeled, they would significantly affect the behavior of the channels. Inappropriately chosen policies can have significant system-level effects – in fact, problems with endorsement policy configuration are fundamental fault modes of channels. However, in our current simplified implementation, there is no specific *endorsement* step; thus, we did not model the endorsement policy.

Application Clients Clients are the most user-facing components of the network, who submit transaction requests to peers. There may be additional logic embedded within clients, but in our current model's scope, clients can do nothing more than submit basic transactions (function names and arguments) to select peers.

Smart Contracts Smart contracts define the business logic of the cross-organizational collaboration a channel enables. In HLF 's terminology, smart contracts are typically referred to as *chaincode* – although more accurately, a piece of chaincode is a group of smart contracts.

Chaincode is installed on one or more peers in the channel. When clients submit transactions to peers, they in turn invoke the chaincode installed on them. The chaincode may read and write state (key-value pairs) from/to the ledger and return a so-called *read-write set* to the peer. The process is described in more detail in the Transaction Flow subsection.

Network The collection of all independent channels, all participating peers and orderers, the ledgers maintained by the peers, as well as the chaincode installed on them, form a HLF *network*.

3.1.2 Transaction Flow

A simplified version of HLF 's transaction flow is part of our model. Clients first submit transaction requests to endorsing peers, who respond with their endorsements and corresponding read/write sets (based on the results of the chaincode execution simulation). Then, the client sends the endorsed requests to the ordering service. Figures 3 and 5 visualize the process.

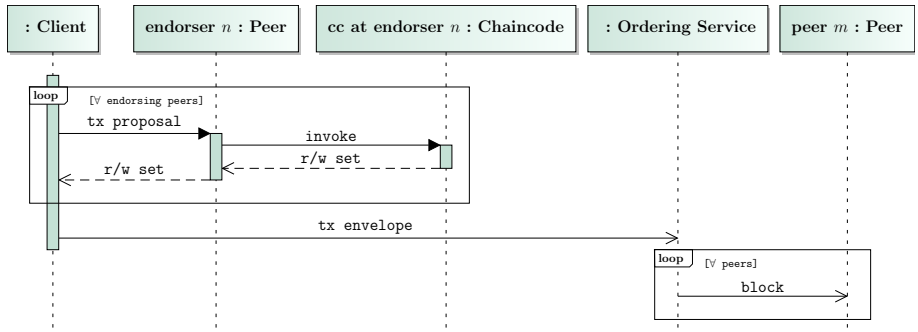

Figure 5: Transaction flow in our HLF model

3.2 Components of the SCIL Framework

We have already described our executable HLF model in detail in the previous subsection. In the SCIL framework, there are three configurable elements of the model:

1) Fault Modes

The fault modes of the individual components can be toggled before simulation. For example, in HLF, if a malicious ordering service intentionally reorders and selectively accepts (i.e., occasionally drops) transactions, ledger updates may not always reflect the expected world state. Platform-level faults in HLF include:

- malicious orderer behavior (transaction dropping, reordering)
- network faults (e.g., traffic congestion)
- host-level faults (e.g., a peer becomes unavailable)
- incorrect configuration (e.g., unsuitable endorsement policies)
- other malicious or unintentional behavior (e.g., client issues)

In a correctly configured network, Fabric protects against some of the potential faults. For example, endorsement policies can be designed to tolerate the downtime of some peers.

2) Network Design
The network design is an instantiation of the modeled components and describes the deployment of the network. This includes aspects such as how many organizations there are, how many peers do these organizations maintain, where are smart contracts installed, and what operation-time defenses have been employed.

3) Smart Contract
Our approach's core idea is to include the smart contract in the simulation as is. After defining the network and selecting the fault modes, one simply needs to *plug in* their existing smart contract code.

Error Property The error properties to consider during model checking can be derived from the smart contract and the application. This would usually be an undesired world state after a series of transactions or some erroneous results returned by the smart contract. We should note that while model checking can prove that a particular error property cannot be satisfied in a given configuration (an error state cannot be reached), it does not guarantee an utterly fault-free system.

Model Checking Given the network design, the enabled fault modes, and most importantly, the smart contract, a model checker can determine whether the specified error property can be satisfied. If so, the model checker also provides a failure trace – a list of events leading to the undesired state. If the property cannot be satisfied, we have formal proof that a certain error state is unreachable (if the initial model was correct).

The "In The Loop" Aspect We have dubbed our approach Smart Contract in the Loop because the smart contract source code is given to the model checker "as is." We do not expect smart contract developers to employ model-driven engineering (MDE) methods, and thus, a formalized model of the smart contract is likely not available. Furthermore, even if such a model exists, it is still worthwhile to test the concrete implementation in a simulation. The advantage of this approach is efficiency for the user: they simply need to provide the network configuration once, then plug in their existing smart contract implementation, and run the model checker.

4 Prototype Implementation

Our Java prototype implementation contains the Hyperledger Fabric (HLF) model described in Subsection 3.1, as well as a framework for model checking using the Smart Contract in the Loop (SCIL) approach. In the following, we describe the key elements of the prototype in more detail.

4.1 Implementation of the HLF Model

We have visually represented the most important classes of the model in Figure 6. A class exists for all major components, and there are also some additional utility classes; e.g., we package the invoked method and the arguments passed in `InvocationRequest` objects and the results returned from smart contracts (more accurately, chaincode) in `InvocationResult` instances. The latter includes the arbitrary result returned by the smart contract method and the read/write set resulting from the transaction simulation.

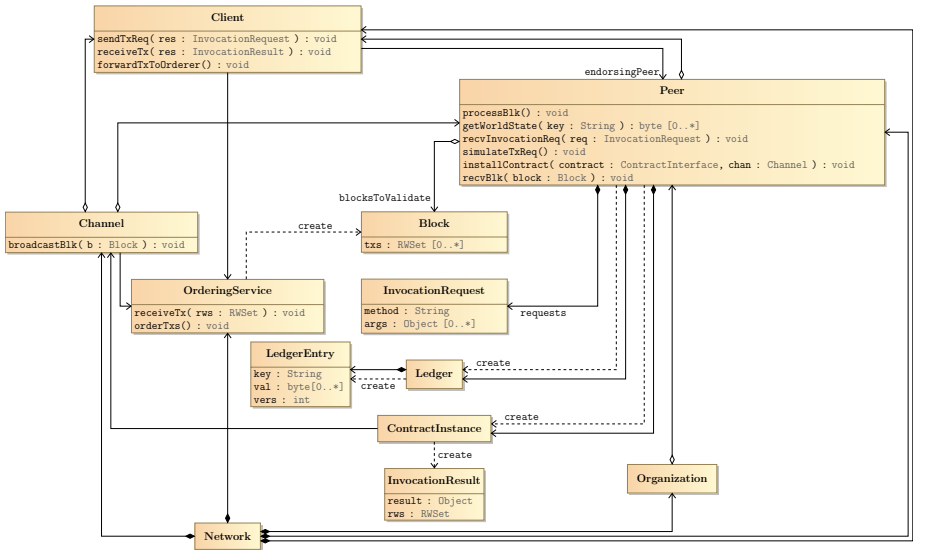

Figure 6: Simplified class diagram of the prototype

Most of Figure 6 follows from the conceptual HLF model presented in Subsection 3.1, but there are a few bespoke classes needed for network simulation and enabling smart contract "in the loop." For instance, the `ContractInstance` class does not, in fact, refer to a concrete instance of a specific smart contract class. Rather, it represents a smart contract installed at a peer, but it does have a reference to a `ContractInterface` object that is going to be an actual instance of the plugged smart contract class.

The ledger is stored in memory; the `Ledger` class contains a list of `LedgerEntry` objects, which are, in turn, versioned key-value plain old Java objects (POJOs). Peers have their local copy of the ledger and provide smart contracts with ledger data during transaction "simulation."

The `Client` class is not a concrete client implementation either, but an abstract, logical application client that can be used to send parameterized transaction requests (proposals) to the network.

There are several more classes present in the prototype that we have omitted for brevity and simplicity. Many of these facilitate the network simulation explained in the following subsection.

We should note that this model deliberately does not accurately reflect how the real HLF works. In the actual HLF implementation, transaction processing is much more complex as it uses shims, stubs, context providers, etc. We have done some simplifications and abstractions to improve model checking performance (by avoiding unnecessarily increasing the state space) and to keep our code concise and maintainable.

4.2 Network Simulation

To perform model checking, we simulate predefined transaction requests' execution on the user-configured abstract HLF network. The sequence diagram in Figure 7 concisely models how the framework simulates the network.

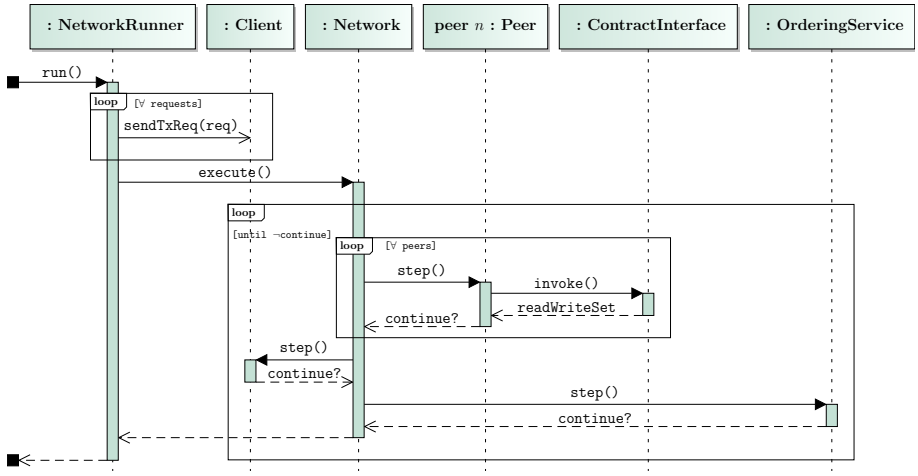

Figure 7: Network simulation

The primary logical entry point is the `NetworkRunner#run(String, OrderingService.FaultMode, List<InvocationRequest>)` method, which first instantiates all network components according to the supplied design and configuration. For now, network design is hardcoded into `NetworkRunner`'s source code,

but in a later iteration of the tool, we plan to offer a lightweight configuration language in which it can be specified at runtime. The method also *dynamically* instantiates the smart contract (chaincode) class based on a fully qualified class name passed as the first argument. We did not illustrate these instantiations in Figure 7 to make the diagram more readable.

The next step is sending what is essentially a call sequence to a client defined in the network. This call sequence is provided to the `run` method as its third argument. Each call represents an invocation with a method name and a list of arbitrary-type arguments.

The network is then ready for simulation. `NetworkRunner` calls `Network#execute`, which in turn begins a simulation loop. In each iteration, the `Network` class sequentially calls the `step` method of the individual components: peers first, then clients, then the ordering service. All components implement the `Participant` interface that requires them to define such a `step` method. During `Peer#step`, peers simulate an invocation of their local chaincode by calling `ContractInterface#invoke` – which will finally delegate the call to the actual smart contract implementation. The network simulation loop ends when no component indicates that there is still more to do.

At this point, all (virtual) ledger updates have occurred, and it is time to check the error property. For now, this is also hardcoded into `NetworkRunner#run` as a simple Java assertion.

4.3 Plugging in Smart Contracts

As explained in the previous subsection, our prototype framework dynamically instantiates chaincode classes based on fully qualified names passed as input arguments.

To make these pieces of chaincode work in the simulation without any additional modifications, we have developed a "shadowing," mock HLF Java package with stubbed versions of the classes required by smart contracts. These stubbed classes are in the `org.hyperledger.fabric` package and are intended to be found on the classpath before the classes in the *real* package supplied with HLF. Key mocked classes include `Context`, `ContractInterface`, `ChaincodeStub`, and annotations offered by HLF such as `@Transaction` and `@Contract`. This setup enables the seamless specification of existing pieces of chaincode to the framework.

4.4 Model Checking with JPF

Our framework uses Java Pathfinder (JPF) [16] (developed by National Aeronautics and Space Administration (NASA)) for model checking. JPF uses a properties file for configuration where the target main class, command line arguments, classpath, and other JPF-specific options. can be set. Listing 1 shows a partial example configuration where the ordering service's `CAN_DROP` fault mode is enabled (meaning it randomly ignores transactions), the "Smart Contract in the Loop" is a

minimal implementation of the train crossing example described in Section 5, and invocations are read from a `traincrossing.invocations` file.

```
target = hu.bme.mit.ftsrg.scil.Cli.CommandLineInterface
target.args = -f,CAN_DROP,-i,traincrossing.invocations,hu.[...].TrainCrossing
classpath = ./[...]/app-0.1.0-all.jar:./examples/train-crossing/[...]
cg.enumerate_random = true
+vm.assert = enable
```

Listing 1: Example (simplified) JPF configuration

5 Case Study

In the following, we demonstrate the viability of our approach in the context of a (hypothetical) safety-critical application where an autonomous vehicle may cross an unguarded railway intersection if, according to a (permissioned) decentralized application (dApp), it is safe to do so; i.e., no train is approaching. Figure 8 shows a simple schematic of this scenario.

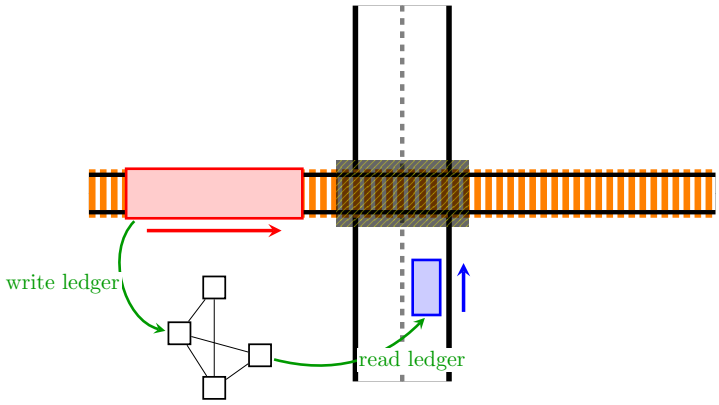

Figure 8: Visualization of the train crossing scenario

The precise operation of this illustrative system (implemented for Hyperledger Fabric (HLF)) is the following:

1. The two network participants are the railway company and the operators of the autonomous vehicles. For simplicity, let us assume there is only a single *train* and a single self-driving *car* – this is inconsequential from the application's perspective but simplifies the explanations in this paper.

2. There is a single world state entry that is updated and checked by the participants in any transaction: the value at the `canGo` key, which is either `"true"` or `"false"`.

3. The chaincode (smart contract) has a single `updateState` function that takes a boolean parameter and sets `canGo` to that parameter's value.

4. The train invokes `updateState` before entering the intersection (to set `canGo` to `"false"`). After the train has left the intersection, it again invokes `updateState` (this time parameterizing it with `"true"`).

5. Upon approaching the intersection, the car queries the ledger content and decides to cross or wait depending on the current latest value of `canGo`.

One way to phrase the fundamental safety-critical requirement in this elementary system: *the value of `canGo` MUST NOT be `"true"` when a train is in the intersection.*

The following describes how the individual elements first introduced in Section 3 are parameterized for the case study.

Network Design As mentioned in Section 4, the tool does not yet support dynamic network definitions at runtime; the network design must be specified programmatically before compiling to bytecode. For our case study, the network has the following components:

- two organizations R_1 and R_2
- a single channel C_1 with an ordering service O_1
- one peer at each organization (P_1 at R_1 and P_2 at R_2), both in C_1
- the smart contract is installed on P_1
- there is a single client in O_1 that connects to P_1

The programmatic setup of this network can be seen in Listing 2.

Fault Modes For illustration, let us assume that the O_1, the ordering service of the channel, behaves maliciously (or at least in a faulty manner): it randomly drops some transactions received. This behavior is enabled by passing the `-f CAN_DROP` option to the program, as seen in Listing 1.

Smart Contract We have attached the implementation of the train intersection smart contract in the Appendix, in Listing 3. It must be available on the classpath at runtime, and its fully qualified class name is specified as the first (and only) positional argument to the command line interface (CLI) of the framework.

Invocation Sequence We use an arbitrary sequence of a few transactions that update the state of `canGo`. The final invocation in the simulation invokes the smart contract with a `"false"` parameter, meaning there is a train entering the intersection, and it is not safe to cross.

Error Property Based on the safety-critical requirement and derived from the smart contract being checked (as well as the client-side invocations used for testing), we have specified the error property as an assertion of `canGo` being equal to `"false"` at the end of the simulation. A `"true"` value at the end represents a potential accident where the train has entered the intersection, but the ledger was not updated accordingly.

Model Checking To run the model checking, one needs to run the `jpf` binary on the properties file (Listing 1). Extensive logging is implemented throughout the framework so that, in case the error property is violated, there is a readily available execution trace. For example, due to the `CAN_DROP` fault mode of the ordering service in our example, it is possible to reach the error state when the orderer places a transaction with a `"true"` parameter last. See Listing 4 for the resulting execution trace leading up to the error.

6 Conclusion

In this paper, we have presented the novel Smart Contract in the Loop (SCIL) method and framework in detail that can be used for the comprehensive verification of a Hyperledger-Fabric-based [1] distributed ledger technology (DLT) application. In contrast to the few other verification and validation (V&V) tools available for permissioned (consortial) platforms, the approach presented in this paper does not only take the smart contracts or the ledger state into consideration but also aspects such as deployment and potential component-level faults.

SCIL contains a high-level model of HLF's key components and their interactions and performs model checking (based on Java Pathfinder (JPF) [16]) to determine whether a predefined error property can be fulfilled in the defined network and having the smart contract implementation (given to the tool "as is") installed. In other words, the tool can show how platform-level behavior can impact service-level behavior through smart contracts.

We have described our approach in theory and our prototype implementation created for the Hyperledger Fabric (HLF) platform and Java smart contracts. We have also presented a case study to demonstrate the viability of our approach on a theoretical safety-critical DLT application.

The framework can already be used, but there are still implementation efforts for future work, such as allowing the specification of the network's design at runtime (through the command line interface (CLI)) and implementing the possibility to define runtime *defenses* as well as faults. Furthermore, our model currently includes a limited set of fault modes for the ordering service component, while there are several other fault modes to consider.

We believe SCIL can also be extended to platforms other than HLF. Indeed, while Solidity smart contracts and the Ethereum Virtual Machine (EVM) can still be considered "common denominators" in the blockchain space, more and more new platforms support general-purpose programming languages and other domain-specific

languages (DSLs) for smart contract development – see, for example, Corda [12], which also supports Java, or Substrate[7], where so-called *pallets* are written in Rust. The key idea of reusing already existing, ready-to-use tools for analyzing smart contract bytecode also applies to other platforms wherever a virtual machine (VM) is used for execution. This includes Corda, as well as a few other platforms, such as Algorand [6].

Acknowledgments

The work of Bertalan Zoltán Péter, partially supported by the Doctoral Excellence Fellowship Programme (DCEP), is funded by the National Research Development and Innovation Fund of the Ministry of Culture and Innovation and the Budapest University of Technology and Economics under a grant agreement with the National Research, Development and Innovation Office.

The work of Bertalan Zoltán Péter was partially created under, and financed through, the Cooperation Agreement between the Hungarian National Bank (MNB) and the Budapest University of Technology and Economics (BME) in the Digitisation, artificial intelligence and data age workgroup.

The research of Zsófia Ádám was partially funded by the EKOP-24-3 New National Excellence Program under project number EKÖP-24-3-BME-288, and the Doctoral Excellence Fellowship Programme under project number 400434/ 2023; funded by the NRDI Fund of Hungary.

References

[1] Androulaki, E. et al. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. DOI: 10.1145/3190508.3190538.

[2] The Aptos Blockchain: Safe, scalable, and upgradeable Web3 infrastructure. Whitepaper, Aptos Foundation, 2020. URL: https://aptosfoundation.org/whitepaper/aptos-whitepaper_en.pdf.

[3] Beckert, B., Herda, M., Kirsten, M., and Schiffl, J. Formal specification and verification of a Hyperledger Fabric chaincode. In *3rd Symposium on Distributed Ledger Technology*, page 44–48. Institute for Integrated and Intelligent Systems, 2018. DOI: 10.5445/IR/1000092715.

[4] Beckert, B., Klebanov, V., and Weiß, B. *Dynamic Logic for Java*. In *Deductive Software Verification – The KeY Book – From Theory to Practice*, page 49–106. Springer, 2016. DOI: 10.1007/978-3-319-49812-6_3.

[7] https://substrate.io/

[5] Buterin, V. Ethereum: A next-generation smart contract and decentralized application platform. Whitepaper, Ethereum, 2014. URL: https://github.com/ethereum/wiki/wiki/White-Paper/.

[6] Chen, J. and Micali, S. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*, 777:155–183, 2019. DOI: https://doi.org/10.1016/j.tcs.2019.02.001.

[7] Dhillon, V., Metcalf, D., and Hooper, M. *The DAO Hacked*. In *Blockchain Enabled Applications: Understand the Blockchain Ecosystem and How to Make it Work for You*, page 67–78. Apress, Berkeley, CA, 2017. DOI: 10.1007/978-1-4842-3081-7_6.

[8] Digital Asset. Canton Network: A network of networks for smart contract applications. Whitepaper, Digital Asset, 2024. URL: https://www.digitalasset.com/hubfs/Canton/CantonNetwork-WhitePaper.pdf.

[9] Díaz, M., Soler, E., Llopis, L., and Trillo, J. Integrating blockchain in safety-critical systems: An application to the nuclear industry. *IEEE Access*, 8:190605–190619, 2020. DOI: 10.1109/ACCESS.2020.3032322.

[10] Feng, C. and Niu, J. Selfish mining in Ethereum. In *Proceedings of the 2019 IEEE 39th International Conference on Distributed Computing Systems*, page 1306–1316, 2019. DOI: 10.1109/ICDCS.2019.00131.

[11] Garfatta, I., Klai, K., Gaaloul, W., and Graiet, M. A survey on formal verification for Solidity smart contracts. In *Proceedings of the 2021 Australasian Computer Science Week Multiconference*, ACSW '21, New York, NY, USA, 2021. Association for Computing Machinery. DOI: 10.1145/3437378.3437879.

[12] Gendal Brown, R., Carlyle, J., Grigg, I., and Hearn, M. Corda: An introduction. Whitepaper, R3, 2016. URL: https://docs.r3.com/en/pdf/corda-introductory-whitepaper.pdf.

[13] Górski, T. and Bednarski, J. Applying model-driven engineering to distributed ledger deployment. *IEEE Access*, 8:118245–118261, 2020. DOI: 10.1109/ACCESS.2020.3005519.

[14] Kuperberg, M., Kindler, D., and Jeschke, S. Are smart contracts and blockchains suitable for decentralized railway control? *Ledger*, 5, 2020. DOI: 10.5195/ledger.2020.158.

[15] McCanne, S. and Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference*, USA, 1993. USENIX Association. DOI: 10.5555/1267303.1267305.

[16] National Aeronautics and Space Administration (NASA). Java Pathfinder, 2005. URL: https://github.com/javapathfinder/.

[17] Neu, J., Tas, E. N., and Tse, D. Two more attacks on proof-of-stake GHOST/Ethereum. In *Proceedings of the 2022 ACM Workshop on Developments in Consensus*, page 43–52, New York, NY, USA, 2022. Association for Computing Machinery. DOI: 10.1145/3560829.3563560.

[18] Péter, B. Z. and Kocsis, I. N-version programming as a mitigation for smart contract faults in execute-order-validate blockchain systems. In *Proceedings of the 30th Minisymposium of the Department of Measurement and Information Systems*, page 33–36, Budapest, Hungary, 2023. Budapest University of Technology and Economics. DOI: 10.3311/minisy2023-009.

[19] Rossberg, A. WebAssembly core specification. W3c recommendation, W3C, 2019. URL: https://www.w3.org/TR/wasm-core-1/.

[20] Shimosawa, T., Sato, T., and Oshima, S. BCVerifier: A tool to verify Hyperledger Fabric ledgers. In *Proceedings of the 2020 IEEE International Conference on Blockchain*, page 291–299, 2020. DOI: 10.1109/Blockchain50366.2020.00043.

[21] Soud, M., Liebel, G., and Hamdaqa, M. A fly in the ointment: an empirical study on the characteristics of Ethereum smart contract code weaknesses. *Empirical Software Engineering*, 29(1):13, 2024. DOI: 10.1007/S10664-023-10398-5.

[22] Tolmach, P., Li, Y., Lin, S.-W., Liu, Y., and Li, Z. A survey of smart contract formal specification and verification. *ACM Computing Survey*, 54(7):1–38, 2021. DOI: 10.1145/3464421.

[23] Wood, G. Polkadot: Vision for a heterogeneous multi-chain framework. Whitepaper, Ethereum & Parity, 2016. URL: https://whitepaper.io/document/596/polkadot-whitepaper.

[24] Yakovenko, A. Solana: A new architecture for a high performance blockchain. Whitepaper, Solana Foundation, 2017. https://solana.com/solana-whitepaper.pdf.

Appendix

```
Network network = Network.builder()
  .addOrganization("R1")
  .addOrganization("R2")
  .addPeer("P1", "R1")
  .addPeer("P2", "R2")
  .addOrderingService("O1", blockSize, faultMode)
  .addChannel("C1")
  .registerPeersToChannel(List.of("P1", "P2"), "C1")
  .installContract(contract /* <- SCIL */, "P1", "C1")
  .registerOrderingServiceToChannel("O1", "C1")
  .addClient("Client", "P1", "O1")
  .build();
```

Listing 2: Programmatic network design setup for the train crossing case study

```
package hu.bme.mit.ftsrg.chaincode.traincrossing;

import org.hyperledger.fabric.contract.Context;
import org.hyperledger.fabric.contract.ContractInterface;
import org.hyperledger.fabric.contract.annotation.*;
import org.hyperledger.fabric.shim.ChaincodeException;

@Contract(
  name = "TrainCrossing",
  info = @Info(/* contract metadata */))
public class TrainCrossing implements ContractInterface {

  @Transaction
  public void updateState(Context ctx, String value) {
    if (!(value.equals("true") || value.equals("false"))) {
      throw new ChaincodeException("Value must be 'true' or 'false'");
    }

    ctx.getStub().putStringState("canGo", value);
  }
}
```

Listing 3: Implementation of the train crossing scenario's chaincode

```
$ jpf model.jpf
JavaPathfinder core system v8.0 [...]

====================================================== system under test
hu.bme.mit.ftsrg.scil.cli.CommandLineInterface.main(
  "-v","-v","-v","-f","CAN_DROP",
  "-i","updateState false! updateState true! updateState false! updateState false! updateState
  ↪  true! updateState false",
  "hu.bme.mit.ftsrg.chaincode.traincrossing.TrainCrossing"
)

====================================================== search started: 9/30/24, 6:25 PM
[INFO | Peer#P1 @ 2024-09-30 18:25:03.164] simulating transaction request [...]
[INFO | Client#Client[connected to P1] @ 2024-09-30 18:25:03.335] forwarding transaction to
↪  orderer [...]
[INFO | OrderingService#O1 @ 2024-09-30 18:25:03.384] building a new block [...]
[INFO | Peer#P2 @ 2024-09-30 18:25:03.663] transaction applied to ledger, world state in peer
↪  updated
[INFO | network @ 2024-09-30 18:25:03.665] Network stopped

====================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.AssertionError: canGo should
↪  be false in..."

====================================================== statistics
elapsed time:       00:00:03
states:             new=10,visited=1,backtracked=2,end=3
search:             maxDepth=9,constraints=0
choice generators:  thread=3 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=2), data=6
heap:               new=5543,released=2117,maxLive=3316,gcCycles=10
instructions:       274880
max memory:         248MB
loaded code:        classes=324,methods=5455
====================================================== search finished: 9/30/24, 6:25 PM
```

Listing 4: Execution trace leading up to the violation of the error property