Robust CBDC applications

László Gönczy, Pál Varga, Imre Kocsis

July 2021

Contents

1 Introduction **2**

2 Notions of robustness **2**

3 Robustness of DLT platforms **4**
- 3.1 Common vulnerabilities of DLT platforms 4
- 3.2 Consensus and transaction execution in Hyperledger Fabric . . . 5

4 Robust smart contracts **6**
- 4.1 Fault model 6
- 4.2 Protection mechanisms 7
- 4.3 Formal analysis of smart contracts 8
- 4.4 Validation and verification 9

5 Vulnerability assessment **12**
- 5.1 Applicability of general purpose vulnerability databases 12
- 5.2 Tools for minimizing smart contract vulnerabilities 13
  - 5.2.1 Mythril 13
  - 5.2.2 Slither 14
- 5.3 Dependability assessment by fault injection 14

6 Robust DLT-assisted workflow execution **16**
- 6.1 Describing and executing workflows or business processes 16
- 6.2 Workflow execution - requirements and dynamics 17
- 6.3 Workflows at enterprise level 18
- 6.4 Modeling parallel workflows at production level 19
- 6.5 Loose coupling and Late binding of stakeholder systems 20
  - 6.5.1 A brief overview of the main Arrowhead concepts 21
  - 6.5.2 The Arrowhead Framework to support DP 23
- 6.6 DLT-assisted workflow execution 23
- 6.7 Techniques for making DLT-based workflows robust 25
  - 6.7.1 Smart Contract Design Patterns in the Ethereum Ecosystem 25
  - 6.7.2 Further Design Patterns for Solidity 26

6.7.3 Creational, Structural, and Behavioral Design Patterns . 26
6.7.4 Proxy Patterns for adding modification features to Smart Contract 26

7 Summary: possible impacts of regulatory control enablement 27

1 Introduction

This documents introduces the general concept of (technical) robustness and dependability, focusing on the execution of smart contracts over blockchain. While smart contract code is considered as the business logic of applications, ledger content, supporting blockchain mechanisms such as consensus and ordering, and even client workload influences the observable robustness of a blockchain application.

We first present the general concept of dependability, then introduce its special applications and considerations in an enterprise blockchain. Information sources and tools supporting vulnerability assessment and a method for systematic evaluation by fault injection is also presented. When it comes to actual execution of smart contract, we approach the demonstration from industrial usage point of view, where workflows are executed both at the enterprise level and at the production level. Although such workflows can be described through various ways from Business Process Modelling Notation (BPMN) to Coloured Petri Nets (CPN), these can also be mapped to smart contracts. Being able to describing the workflows themselves error-free is an initial step towards having robust smart contracts based on those workflow descriptions. Describing the necessary precautions on smart-contract-based workflow execution is also part of the aims here.

2 Notions of robustness

First, the general concepts of dependability are introduced, in order to define the context of technical robustness of smart contracts, and in general, blockchain-based applications. Methods of fault tolerant computing traditionally covers modeling, analysis and improvement of computer-based systems. While some decades ago, the target domain of such methods were primarily mission-critical systems, where the consequence of a fault in the system may lead to catastrophic effects, such methods are now being used in general purpose systems as well, most cases hidden from the end-user or even from the application developer.

Operation and maintenance of large-scale computer systems providing resources for cloud services (see e.g. Site Reliability Engineering concept of Google, [5]) or design time static verification of computer drivers in desktop operating systems are applications of these long existing (and yet developing) methods in everyday life. Moreover, the core ideas of blockchain based systems, including mechanisms for consensus and ordering of transactions rely on methods which are originated in fault tolerant computing.

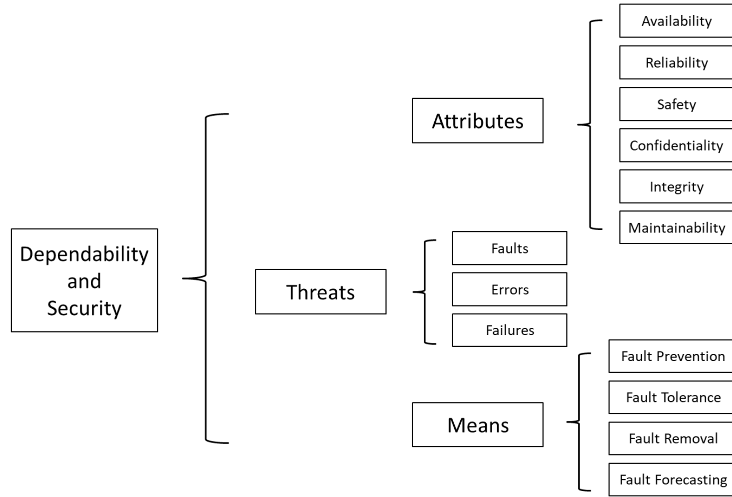

Figure 1: Dependability and security key concepts, based on [3]

Extending these apects, performability in general refers to the performance of a system in the presence of faults. In traditional software systems, design of performability refers to the control and parametrization of fault tolerant mechanisms (e.g., the waiting time between two message transfer, the semantics of communication in terms of filtering duplicate messages and ensuring integrity, etc). so that the system will meet its performance requirements under a given assumption of fault characteristics.

Talking about robustness, one of the key aspects is the fault propagation chain. Following the taxonomy defined in [3], we introduce the following:

- A *fault* is the core deviation causing a change in the state of a service/software. In the case of blockchain applications, an example can be a wrong instruction in a smart contract code (e.g. using the strict inequality operator "<" instead of "<=" in a parameter check). A fault can be active or passive, i.e.,; there can exist many faults which will never cause any deviation from the desired operation. In our example, if the parameters of the smart contract will never be equal, the fault will be dormant.

- An *error* is a change in the system state caused by a fault. This is still internal to the system, i.e., the wrong value of an internal variable, which does not necessarily lead to an observable wrong behavior.

- A *failure* is a user-perceivable deviation from the desired operation. In our case, if the parameter check is a pre-condition of the execution of a withdrawal transaction from an account, the result of a specific transaction execution will be different from the specification.

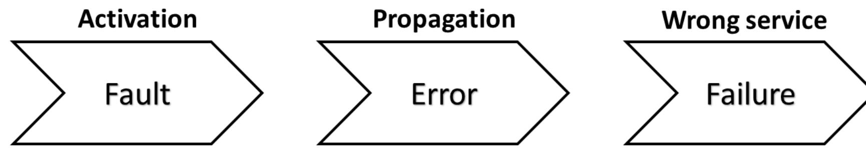

Figure 2: The fault-error-failure chain

Even this small example highlights the difficulty of finding potential software faults ("bugs") by traditional testing: finding the above deviation would need test cases where it is ensured that the parameters of the transaction are equal, and given the context and result of other computational steps, the return value of the transaction (the success of the withdrawal or the balance on the accounts after execution) is different from the specification.

3 Robustness of DLT platforms

When associated with systems, the term *robustness* means the tolerance against perturbations that may affect the systems's functions. The robustness of a system can be challenged by faults in the surrounding environment or threats of unintentional misuse or deliberate attacks. Analyzing the vulnerabilities of a system is a common way to evaluate its robustness.

3.1 Common vulnerabilities of DLT platforms

There are a number of research papers available which describe certain characteristics of blockchain systems which can be exploited by targeted attacks. Although the most well-known such attack is the "51%attack", a simple brute-force attack against the majority-base consensus mechanism, other threat models include partitioning of the network or injecting transactions resulting in double-spending, with a combination of low -level network attacks and suspicious participants (nodes) in the network. Such attacks target the basic operation of blockchain-based applications, exploiting the distributed nature of these systems. However, without introducing sophisticated mechanisms to separate or poison the nodes (which also might need significant computational power), it is also possible to exploit the vulnerabilities of the applications running over blockchain. This also might be performed at multiple levels:

- Attacking the protection/AAA mechanisms of the client application: if a blockchain may accept transactions from client applications (e.g., wallets) which have an inappropriate level of protection, then syntactically correct, properly authorized yet malicious transactions can be injected. Although this may lead to significant amount of loss, we do not focus on such attacks here, since these can be prevented by using standard protection mechanisms (like 2FA, OTP, etc). Wallet theft and cryptographic attacks may be considered as part of this category as well.

- The blockchain network can be attacked at the network level in order to split the network and therefore create multiple distinct subsets of nodes. Potential attacks target the overall consensus mechanism by creating orphan blocks or multiple forks.
- The P2P-based mechanisms ensuring communication among nodes in the network have many known attack vectors: delaying the consensus mechanism, manipulating the time synchronization, attacking the network services supporting routing across nodes (e.g., DNS, BGP), and, in the case of PoW networks, attacks which corrupt the evaluation and reward of mining are some typcial examples.
- Another, more important kind of attacks include application-level attacks by malicious execution of smart contract code. This can be prevented by design time analysis/verification and runtime monitoring using self-checking and invariant-based mechanisms.

3.2 Consensus and transaction execution in Hyperledger Fabric

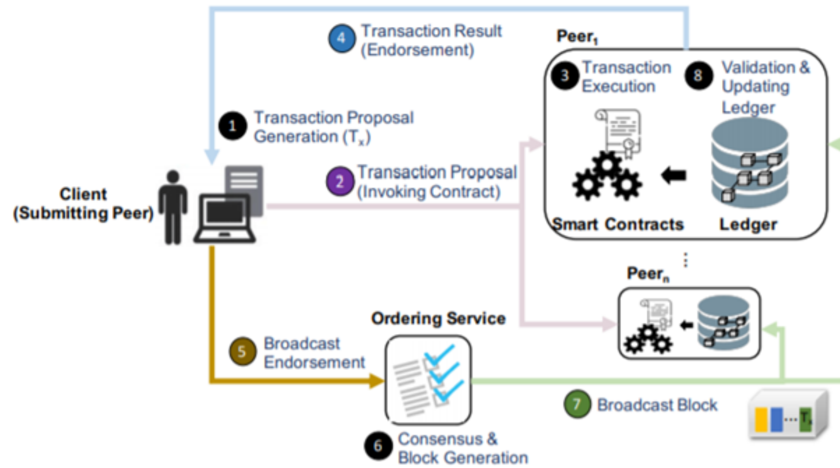

Figure 3: Transaction execution in Hyperledger Fabric

The client sends a digitally signed transaction proposal to one or more of the endorsing peers in the network. Execute: first, all endorser nodes (peers) execute the transactions locally. This means a simulated run of the transaction, and generation of Read-Write sets, i.e.,determining the data (technically in the form of key-value pairs) which will be read/written from/to the ledger if the transaction is accepted.

Answers from endorsement peers are returned to the client (digitally signed) which uploads these to the ordering service.

Depending on the result of the consensus mechanism, transactions are either accepted or rejected and blocks will be created to execute the valid transactions (note that this point the ledger status is unchanged). After the endorsement phase, transactions are validated, which includes the check format, the authorization of the client, signature check, etc. Transactions which fail these checks will be rejected. Valid transaction will modify the ledger state. If the transaction (as a function call) has a return value, this will be actually generated in this phase as well.

Main parameters of this mechanism include the number of transactions per block, and the requested number of nodes to agree during the consensus. In a consortial blockchain environment, an " n out of n" scheme may be followed, requiring all nodes to agree on the transaction execution. However, there is a possibility to give different schemes as well (e.g., to ensure that one compromised node cannot block the entire network).

4 Robust smart contracts

In order to define end evaluate robustness of smart contracts, or more precisely, smart contract based systems, possible fault categories and protection mechanisms for fault prevention/removal have to be defined.

4.1 Fault model

An initial fault model which describes the effect of deficiencies in a smart contract, concentrating on observability, was presented in [14]. We assumed that there is a reference implementation returning the correct value, so the result of each transaction can be evaluated and unequivocally labelled as good/faulty. Technically this can be achieved e.g., by executing carefully checked, correct version of smart contracts and then experimenting by injecting faults as briefly described in section 5.3. Note that the following categories do not represent distinct sets of errors: the same contract (executing multiple transactions) may belong to more than one of these.

Reliability failure indicates the possibility of transactions where the return value of a contract is incorrect wrt. the specification. These are cases where the execution returns with failure; however, since the fault categorization considers observable behavior, it might be possible that latent errors are also present in the same application, considering the ledger state as well.

Latent Ledger Integrity error refers to contract execution where, without any deviation of the observable behavior, the ledger content is different from that of the reference execution. Such errors are not discovered during the execution of the transactions; one of the key task of designing for fault tolerance is to either eliminate/prevent such errors or "force" all latent errors to (even implicitly) lead to a failure.

Abort failure represents a category where the transaction is aborted during the endorsement phase, although it should have been accepted by the set of

peers.

Integrity failure refers to faults which result in a ledger modification which violates the domain/business logic requirements (e.g., a transaction is committed which should have been aborted or the WriteSet is different from the specification).

In the context of a CBDC workflow, such failures would correspond to transactions not following the prescribed business logic. Latent here refers tot he possibility of immediate detection, and takes us tot he determination of error confinement region (e.g., in a transaction graph): all transactions which may be affected by the result of an integrity violation.

4.2 Protection mechanisms

The effect of the above fault categories can be prevented/mitigated different protection mechanisms, applied at several phases at development lifecycle.

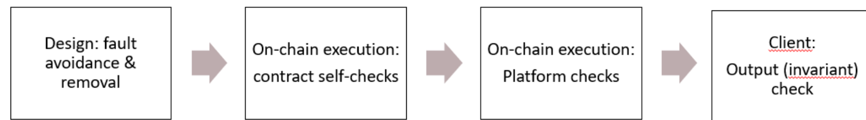

Figure 4: Protection mechanisms for blockchain smart contracts

Design time protection includes manual and automated checks during development. Such checks include the use (and avoidance) of certain code patterns, manual code inspection and audit, testing and the use of automated formal verification methods (see in more details in Sec.4.4).

Contract Self-Check are defensive programming constructs which can be inserted by the contract developed in order to locally evaluate the correctness. In the case of Solidity, for instance, the *require* construct refers to the check of preconditions (like values of input parameters, state variables or return value of external function calls) while *assert* is used to validate invariants and catch internal (logical) errors. Such mechanisms serve as a controlled "commit-rollback" for smart contract execution.

Runtime Platform Checks are mechanisms which are enforced by the runtime environment and not bound to specific contract logic. In the case of Ethereum, these are *Contract Self-Check* enforced by the Ethereum Virtual Machine (EVM) like check of gas limit, array indexing, address format, etc. while in the case of Hyperledger Fabric timeout detection result check over endorsing peers are such mechanisms. While these techniques work without explicit contribution of the smart contract programmer, they have the drawback that in some cases they only catch failures without explicit reference to the occurred faults. Depleting gas, for instance, can be a consequence of a bad program code.

Output invariant check refers to the possibility that the client (application) checks the return value of a transaction. Here it is important to note that the *return value* and the *WriteSet* of the transaction are different: a return

value in typical smart contracts refers to the final statues of execution (e..g, transaction successful) while the WriteSet contains the actual content to be written to the ledger. Therefore, it is an important design decision how to reflect the business logic, and therefore, guarantee the observability of the important attributes of the WriteSet to the client. (Explicit query execution on ledger content facilitates observability but leaves the control to the client application). Client side invariant check can not only be a tool for failure detection but in some cases (if the client is participating in the consensus mechanism, like in Hyperledger Fabric) it can serve fault prevention purposes as well.

4.3 Formal analysis of smart contracts

As mentioned earlier, one of the most effective techniques to improve the robustness of a system is formal verification. This refers to a set of mathematically well-funded methods, where the model of a system is evaluated against a precise specification to find possible situations where a requirement is violated. The main advantage of such methods is that an exhaustive check guarantees that the system is free of faults.

However, such methods need a precise mathematical model and a corresponding mathematical specification of the required properties. Moreover, applying formal methods directly needs significant background knowledge. To overcome these barriers, model-driven dependability analysis methods support the definition of formal requirements at the level of the engineering model, and derive mathematical constructs (like logic formulae defined over a set of constructs which refer to the structure of the engineering model, finite state automata describing the allowed communication protocol between two parties, etc) from such high level models.

[44] and [2] collect current state of the art on the application of formal methods in the field of smart contract analysis. According to their categorization, most of the currently available methods consider the following elements:

- User of the system: users are generally modeled as external actors, with the possibility to define behavior in form of sequences or processes executed by a given class of users. This allows to distinguish between malicious and normal users, and also model users with different strategies.
- (Smart) contracts: the business logic of transaction execution is captured by smart contracts. In formal verification, the concrete programming language is often mapped to a constructs like set and logic operators. It is also important to note that concrete numerical values are often abstracted into (ordered) intervals representing so-called equivalence classes of concrete values. E.g., in the case of a decision about a transaction confirmation, small, medium and large amounts can be handled differently. This approach is called qualitative abstraction and is often used in computer-aided verification, since it helps to create compact models (more precisely, create compact state representations of a system). Several fur-

ther techniques exists to facilitate the effective modeling and evaluation of dynamic models, the interested reader is advised to consult REF.

- Ledger state: the system state in most cases is modeled by the ledger state. This relies on the implicit assumption that the ledger stores the same information for all nodes. There are multiple already existing and emerging technologies to support information hiding, and therefore, the observable ledger state might be different at different nodes in the system (by using, e.g., private data channels supported by Hyperledger Fabric); this leads to an abstraction relation between the entire (private) and public ledger data. Such abstraction relations are widely used in formal data modeling and in general may lead to over-approximations (e.g., by labelling some states of the system suspicious or undecidable instead of correct), but this does not constraint the applicability of formal data modeling methods.

- Requirements for the system express assumptions on the behavior of the user (e.g. input sequence), the contract and the ledger state together.

Main classes of such requirements are the following:

- *Liveliness*: such requirements guarantee that the system will not reach a state during its operation where there are no further possible actions enabled (of course, with the exception of the final state of the execution). In broader terms, liveliness requirements cover behavior that will happen in the system. In the context of transaction execution, an example requirement can be: "All transaction executions will return a value of the Success, Failure set."

- *Safety*: safety requirements define forbidden behavior (what should b avoided). Such behavior is often expressed by a sequence of events/actions or a subset of the state space of the system. An example for a safety requirement can be: "There should be no token transfer where the token disappears during the transactions".

- *Correctness*: Such requirements correspond to application/domain specific criteria, but can be handled with similar mathematical constructs. An example from the financial transaction processing application: after all transaction execution, the balance of a client (end-user) must not be lower than the threshold set to the account.

  Besides describing the general requirements on smart contracts, there are domain-specific extensions to capture requirement fragments (expressed often as "clauses") typical to a certain application domain. The advantage of such reusable fragments is that these can be mapped to reusable verification criteria and program code as well.

4.4 Validation and verification

Smart contracts, just like any other piece of software, require validation and verification in order to ensure requirement compliance. Validation is the process of

checking whether the design of the software meets high level requirements (i.e., whether the software requirements follow the system requirements) while verification is the process of inspecting the software and evaluating its compliance to requirements.

Validation is hard to automate in general, as it needs a precise requirement definition language with proper expression power. [11], for instance, introduces SLCML (smart legal contract markup language) with an illustration from the automotive domain, following a layered approach to support Business Network Model definition, with the main focus on legal requirements (obligations and conditions) and conditional transaction rules.

Verification of smart contracts follows a similar approach to general purpose software verification. Static verification methods work on a model derived from the source code. In most cases, these rely on the Abstract Syntax Tree of the program, which is a typed graph capturing the structure of a software code. Node of an AST correspond to program elements like variables, operators, function calls, etc. where two elements in the graph are connected if there is a containment between the elements (e.g., a variable is referred in the condition evaluation of a loop). While an AST preserves the order of the statements in the program, it does not contain elements which are important only for readability of the code. AST is typically the basis of creating models of a software which are processable automatically in order to find certain patterns. Control Flow Graph (CFG) for instance, is a directed graph, capturing the dependencies among statements of a program. CFG is often used as a basis for metrics calculation, which estimate the difficulty of a software (e.g. cyclomatic complexity as defined by McCabe). Such metrics are not only important for the general quality reports, but can also concentrate the manual verification/code inspection work as described in [37].

Here we do not want to give a deep and detailed overview on all smart contract verification methods (which are emerging); instead we give an overview on the goals and benefits of these. In general, such methods aim at finding typical flaws in a program, like EtherTrust, SmartCheck, Slither, Symboleo (REF).

Verification methods, in general, systematically check a formal model of the system (derived from the abstract representation) and the fulfillment of several requirements. System behavior and requirements are both mapped into dynamic models (often captured as state machines). E.g., if a requirement states that *a transaction must be preceded by a confirmation of a third party*, than the status of the transaction and the confirmation together are part of the requirement specification. Verification mechanisms are performing an exhaustive search looking for situations where the composite state, combined of the behavior of the system and the requirement, shows a possible property violation (e.g., according to the the model of the system, the transaction is enabled without previous confirmation). Note that since requirements are mapped to model (state machine) fragments, verification methods are limited only by the expressive power of the requirement modeling, but not bound to concrete application domain. In case of verification failure, a counterexample is returned, typically

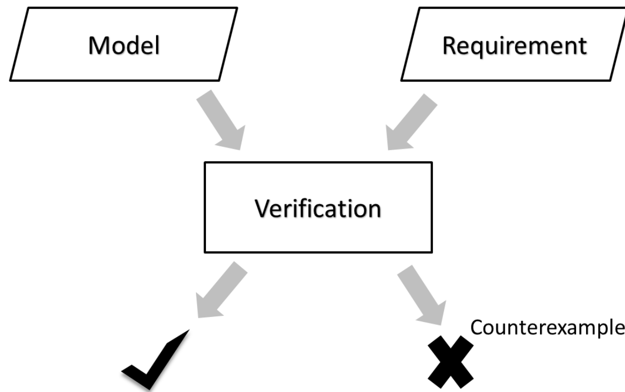

Figure 5: Basic principle of verification methods

in the form of event or action trace, leading to the requirement violation.

The possible size of the state space derived from system model may be a challenge, however, several techniques exist (from symbolic execution to iterative, heuristics-driven methods or counterexample-guided refinement) to facilitate the evaluation of requirement without the check of all possible states. In the previous example, if there is a large loop in contract execution which does not contain actions referring to the confirmation or the execution of a transaction, then the entire loop can be neglected when checking the sample requirement.

Note that verification methods need only the smart contract code (or a model which described an equivalent behavior) and a model of the requirements, but not an implementation or concrete blockchain network. The result of verification (assuming that these models are correct, i.e., there is no possible behavior in the implementation which is not captured by the model) covers all possible cases, therefore can be considered more precise than the result of testing. On the other hand, due to the over-approximation of models, there can be false positive results, where verification identifies a trace which cannot happen in the real implementation (e.g., because of the qualitative abstraction methods, a combination of qualitative values cannot be mapped to a valid combination of concrete values in the implementation.). Such result need an expert review and a possible refinement of the models.

However, there are several reasons for testing applications even if their model passed a successful verification:

- Integration and configuration issues: As verification is performed over abstract models, configuration problems (e.g. timeout, data conversion, network-related problems) must be checked by testing the final application deployed on the blockchain network as well.
- Model correctness: depending on the development method, the behavior final application (including the interface to client applications) may differ from that captured by the model.

5 Vulnerability assessment

After defining the basic concepts of dependability and the possible fault categories, we describe how smart contracts (and in general, blockchain applications) can be evaluated in order to assess their robustness. Until now, we defined logical faults and fault categories, here we present some sources of information which help to turn such logical faults to concrete defects at the level of the technologies applied.

5.1 Applicability of general purpose vulnerability databases

As blockchain-based applications are typically embedded into existing enterprise infrastructure, their robustness is effected by the environment they are being used, including the protection mechanisms of client applications. Therefore, besides the evaluation of smart contracts themselves, a systematic check of the "surrounding" environment is needed.

Common Weakness Enumeration (CWE, available at MITRE page) is a database containing general weaknesses, together with measurement/detection, mitigation and prevention mechanisms. The description of a weakness contains a description and a hierarchy of weaknesses which can be considered as specialization (see e.g., the description of improper input validation for an illustration). Besides the description of the weakness, a collection of related attack vectors is also presented to define how a series of malicious requests can exploit the vulnerability. While CWE is not specific to smart contracts, it serves a starting point for robustness analysis.

CVE (Common Vulnerability Enumeration, MITRE page) is also a collection of cybersecurity vulnerabilities, containing technological details as well. For instance, the keyword "smart contract" returns more than 520 vulnerabilities specific to smart contracts (as of June 2021). A lot of these are directly related to improper handling of token functions at different ERC20 compatible token implementations (e.g., a potential overflow of an integer value could result in setting an improper balance).

CAPEC (Common Attack Pattern Enumeration and Classification, MITRE page) is a collection of attack patterns, described at high level, which can then mapped to concrete technologies. E.g., the generic scheme of invalid input data is specialized in the case of web-based system to the attack type where system files are retrieved by manipulated client requests. Although CAPEC does not focus directly on smart contract execution, a number of possible attacks are collected which might be used to exploit vulnerabilities of the underlying cryptography services. CAPEC also enumerates mitigation and solutions techniques (which also have to be mapped to the concrete technology).

Looking for specific vulnerabilities of Hyperledger Fabric applications, one can mention Chaincode Scanner (previously Hyperchecker, EF), reviveCC ().

5.2 Tools for minimizing smart contract vulnerabilities

It is crucial for a smart contract to be as safe and as bug-free as possible, since deployed contracts cannot be updated. This means that the developers have no way to patch the already deployed smart contracts like they can release security fixes for traditional software, so the code will contain the bug forever, leading to the possibility of exploits, that could cause serious losses, just like it happened in the case of the infamous TheDAO attack [29].

The definition of smart contracts can go wrong in various different ways – the underlying reason is mostly lack of technical understanding. In order to avoid such defects, creators of smart contracts should follow certain best practices and rigorously validate and verify the behavior of the smart contract before unleashing it in the wild. A good reference on the possible smart contract defects can be found in [7], where the authors provide ideas on preventing such defects, as well as describing the effects of such errors in Ethereum-based blockchains. When discussing smart contracts on Ethereum, further description of best practices to avoid security vulnerabilities can be found in [49] and in [31].

A further reference material in this topic is [27], in which the authors – beside describing vulnerabilities and their exploits – present their tool called Oyente, aiming to discover vulnerabilities in smart contract codes. The ContactFuzzer [20] is a tool with similar aims, complementing the Oyente capabilities in vulnerability analysis.

In our case, to mitigate the risk of the code containing serious vulnerabilities, the code of the company contract was checked by tools that detect vulnerabilities in Solidity contracts, including, but not limited to reentrancy vulnerabilities, unchecked tokens transfers and functions allowing anyone to destruct the contract. The tools used are Mythril and Slither.

5.2.1 Mythril

Mythril is a security analysis tool that uses symbolic execution, SMT solving and taint analysis to detect security vulnerabilities in smart contracts (more specifically, in the EVM bytecode of the contract) [33]. The full list of vulnerabilities detected by Mythril can be found in its module listing [34]. By default, Mythril uses 22 as the recursion depth for the symbolic execution engine. To increase the number of explored states, therefore lowering the possibility of uncovered states and bugs remaining in the code, Mythril was run with double recursion depth compared to the default value.

Regarding our use case demonstrator, Mythril did not find any vulnerabilites in the company smart contract with 44 recursion depth. The command and its output was the following:

```
sudo docker run -v ~/contracts:/tmp mythril/myth analyze /tmp/Company.sol
--max-depth 44
```

```
The analysis was completed successfully. No issues were detected.
```

5.2.2 Slither

Slither [41] is a static analysis framework for Solidity. It converts Solidity smart contracts into an intermediate representation, therefore it is able to preserve semantic information that would be lost in transforming Solidity to bytecode [12]. It complements the use of dynamic analysis tools, such as Mythril, it is able to find more/different vulnerabilities, and it can highlight code optimization opportunities. The list of vulnerabilities detected by Slither – along with the information about them and the detectors – can be found on its GitHub page [41], as well. GitHub page.

Regarding the company contract in our use case demonstrator, Slither only reported informational warnings about the used compiler version being too recent (0.8.4). Other warnings were about the ERC20 and asset contracts.

5.3 Dependability assessment by fault injection

Besides evaluating the robustness of a concrete contract, it is important to estimate the effectiveness of different protection mechanisms, in order to give a general suggestion on improving the quality of blockchain applications. In critical systems, this effectiveness is often evaluated by applying *fault injection* techniques. This approach takes a fault-free system and a fault model as input, and tries to systematically insert faults, finding all possible occurrences, and then evaluates the behavior of the artificially "spoilt" system, comparing it the the fault-free reference. This technique was first applied in hardware testing where the effect of physical phenomena (heat, vibration, background radiation, etc.) is evaluated on test devices. In case of software, fault injection refers to the mechanism where typically programming faults are inserted into a correct program code, and the effect of these faults are checked. This can be considered as a practical evaluation of the fault-error-failure chain at a system given configuration.

Faulty version of a software are called *mutants*, referring to the *mutation operators* which represent typical faults. Such an operator can be e.g. comparing two objects by reference instead of value; this comparison will return "false" to a the comparison of logically equivalent objects stored at different memory addresses. Another typical faults are the usage of wrong arithmetical operator, not handling a possibly zero value of a divider, etc.

In order to apply such mutation operators systematically, their potential occurrences (mutant candidates) are labelled according to the categorization model of ODC (Orthogonal Defect Classification). ODC defines severity and potential impact of faults, as well as a logical category for the mutant (e.g., whether a wrong operator in a concrete comparison corresponds to a *wrong input validation* or a *wrong arithmetic expression used in variable assignment*). To find such places in the program code, it first needed to be analyzed using the Abstract Syntax Tree representation.

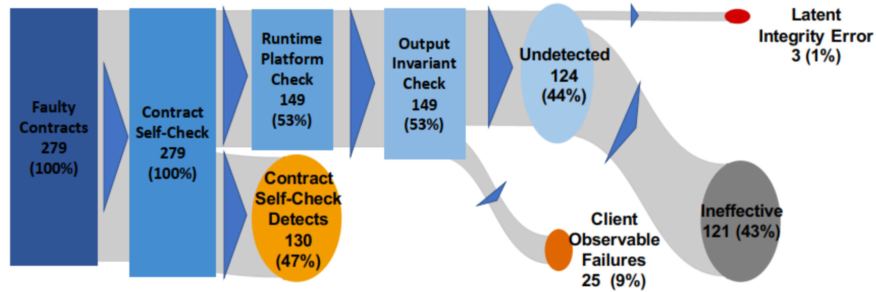

Figure 6: Fault detection example: contracts with built-in protection, without formal methods, [14]

After finding the mutant candidates, a check is performed in order to filter duplicates (where the same mutation operator covers multiple fault categories) and, by using qualitative abstraction, the representative mutants are generated. While this correspond to the fault injection phase, evaluation is performed by running the contract with a pre-defined workload, which is representative to the contract family (e.g., in the case of a transaction handling contract, the workload contains both successful and failed requests, etc.).

The same mutant was evaluated under different configuration, as the purpose was to evaluate protection mechanisms in the presence of faults. There were two orthogonal dimensions of protection: whether formal verification was applied or not and what was the level of protection assumed from the contract developed. The latter was modeled by creating three variant from each contract: a base contract which was an original publicly available contract, a 'stripped' version where all protection constructs which are not necessary for the execution of the core business logic were removed (representing the "quick and dirty" development of proof of concept applications) and a "protected" version where protection mechanisms (like revert and asserts) were inserted to all potential places in the source code (representing the careful contract developed following all best practices).

As an example, we introduce the evaluation of protection methods, and in general, possible development processes, we present the result of two concrete parameter combinations of the experiment space: where contracts are well protected by the developer, first without formal verification, then assuming that a design time formal verification precedes the deployment.

As Fig.6 shows, the self-detection mechanism of the smart contracts catch most faults, preventing them from causing a user-perceivable failure. Platform mechanisms, due to their more general nature, did not improve fault detection (for the sample test set of contracts), since these would have caught faults which are already covered by specific checks in revert and assert constructs. In a small number of cases, some faults cause failures, which, however, can be detected by the client application as the return value of the transition are different from

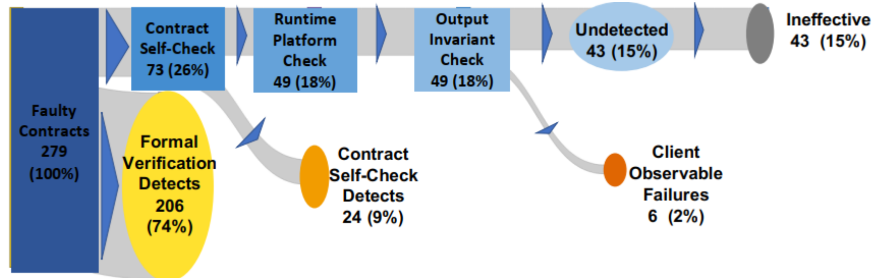

Figure 7: Fault detection example: contracts with built-in protection, with formal methods,[14]

that of the reference implementation. In a small number of contracts, latent errors (which cannot be detected at the time of execution) will occur which may clearly undermine the robustness of the application, as it may later result in faulty operation when reading such data. Note that percentages on the figure represent ratios across contracts and not individual transactions; therefore, with other types of workload these may be lower as well.

Fig.7 illustrates results for contracts where formal verification is also used: in this case, most of the mutants will be detected at design time and no mutant can cause latent errors. Note that the number of ineffective mutants (causing no difference from reference implementation and not detected by protection mechanisms) is lower as well: this reflects the fact that verification is rigorous and will find cases which would not be detected by testing.

6 Robust DLT-assisted workflow execution

6.1 Describing and executing workflows or business processes

Workflows appear in various levels (from physical devices to abstract business structures working together towards a goal) and in various horizontal domains (from production to healthcare). The description of these workflows are often based on a domain-specific language or on a set of rules that does not even qualify a language. In other cases the description is based on standardized concepts such as BPMN (Business Process Modeling Notation) or CPN (Coloured Petri-Nets).

Recently, the description, execution and analysis of workflow processes have become a popular thanks to the Industry 4.0 initiatives. The following paragraphs are an extract of our previous work on related literature study [22].

Krishna et al. [25] analyzed business processes that describe the production of goods or services as a set of local tasks and inter-organization exchanges. In their view it seems that primary business process modeling notations have a workflow perspective of business processes. A thorough state of the art study

is presented by Brouns et al. [6], which identifies certain differences of business process design and implementation when it comes to the digital world and IoT use-cases. They highlight the importance of presenting IoT system of systems in business process models in a unified way. They also suggest to incorporate the various concepts of IoT and cyber-physical systems, focusing on the connection between digital systems and physical agents.

Modeling languages do not fully cover concepts such us availability and mobility of resources or process context information. Among the alternatives for modeling analyzed, such as the BPMN 2.0, EPC (Event-driven Process Chains), and UML (Unified Modeling Language), their paper pinpoints BPMN as the best position to represent IoT; although some improvements are necessary. BPMN 2.0 is an ISO standardized notation for modeling business processes that can be made executable either using process engines (e.g., Activiti, Bonita BPM, or jBPM) or using model transformations into executable languages (e.g., BPEL – Business Process Execution Language) [22].

Mass et al. compares different workflow execution types: an embedded workflow engine and a coding program representing the workflow [28]. The conclusion is that executing business processes on a workflow engine cannot be justified in cases where system resources are sparse, and the model is not reused. In cases where the same process is executed multiple times, and lots of memory is available, a workflow engine is a viable solution. The same applies for distributed DLT-based smart contract execution. It is often unfeasible for IoT endpoints to do anything more than "logging" transactions to the BC, due to resource constraints.

6.2 Workflow execution - requirements and dynamics

When mapping workflow execution to smart contracts, it is important to eliminate the problems first at the workflow descriptions.

The correctness of the workflow processes and their changes can be analyzed in various ways, where the aim is the preciseness of task sequences or finding out structural errors [13, 1, 43]. Regarding the handling of fast-changing business environments, Du et al. [10] found that it is so challenging for the workflow execution that they propose an off-line approach using temporal constraints that impacts during the design stage. They identified that systems must adapt and respond dynamically to changes throughout the life cycle. Therefore, the ultimate goal is to provide a solution that can operate in real-time circumstances and can adapt to changes on-the-fly, like the substitution of a given resource (occupied) by another available. This research is independent from the service oriented architecture-based IoT production workflow execution, but points towards the same requirements and probably, solutions as well.

When creating a framework to manage these processes and resources, we have set the following requirements in [22]:

1. Requirements at the enterprise level:

- To provide a high-level graphical vision of workflows for easy understanding and monitoring;
- To provide a universal language for easy communication;
- The technique modeling processes must be easily implemented and deployed in real scenarios.

2. Requirements at production level:
    - To represent complex processes involving distributed systems and production flows;
    - To accommodate timing constraints and enable hierarchical composition.

3. Requirements at the framework level:
    - To accommodate standard modeling techniques;
    - To provide a single framework for the implementation, execution, and monitoring of processes;
    - To maintain the hierarchy for enterprise and production levels;
    - To provide tools for the agile and dynamic construction of workflows;
    - To enable the automatic deployment of workflows;
    - To enable business process changes during the whole lifecycle;
    - To enable the sharing of resources (reallocation and substitution) in real-time;
    - To enable interoperability and integrability for heterogeneous systems;
    - To enable a service-oriented driven architecture guaranteeing adaptable, loosely coupled, and late-bound services.

These requirements are partially covered by the Arrowhead Framework already [21] – and since part of the framework is still growing, the plan is to cover all the above requirements at a certain stage.

6.3 Workflows at enterprise level

This section appear as is in our published work, [22].

Several business process modeling tools and languages have been proposed to describe, analyze, and evaluate business processes [15]. The BPMN [48] is one of the most widely used and standardized modeling languages that can be used to describe workflow structure, organizational level tasks, helping to manage process-related resources effectively. Consequently, BPMN serves as a universal language that bridges the communication gap that often occurs between business process planning and implementation. The primary goal with BPMN is to support business process management and provide a structure

that is easy to understand for every participant during the production. The BPMN specification also provides a graphical description and has the following components:

- Events: start, intermediate, end;
- Activities: (i) Tasks: service, user, script, mail, receive, business; (ii) Multiple instances; (iii) Sub-processes; and (iv) Loop.
- Gateways: exclusive, inclusive, parallel, event-based and complex;
- Data and Flows: data object, association, sequence-, default- and message flow.

There are several possibilities for creating and executing BPMN workflows, depending on the solution preferred by the designer. The most common process engines are [4] Activiti, jBPM, or Bonita BPM. Model transformation to some executable language such as BPEL is also possible [25].

6.4 Modeling parallel workflows at production level

This section is an extended version of a similar section in our related, published work [22].

Modeling languages – such as BPMN or BPEL – are relatively easy to interpret, but they are not entirely suited to describe very complex production cases. In contrast, based on a previous comparison [47], Petri nets are useful for describing complex logic that can represent distributed systems and production flows, as well. The Petri net [32, 19] is a widely-known and used graphical-mathematical model-description language. Briefly described, it is a directed bipartite graph consisting of places and transitions. Directed arcs carry "firing" conditions between places and transitions. This language is suitable for modeling distributed systems; this is one of the reasons why it is widespread in the industrial field.

Although the language satisfied the industrial expectations in the past, the requirements of Industry 4.0 already set higher demands on modeling languages in general, which traditional Petri net can no longer satisfy in the sense of, e.g., timing or hierarchy. However, along with the recent digital industrial revolution, the Petri net has also undergone many iterations – resulting in the so-called High-Level Petri nets, which can be used to describe more complex industrial processes. One of the languages is resulting among others the Coloured Petri net (CPN) [8]. This extension adds the ability to carry more complex information in the tokens to Petri nets, becoming the token "colored". It also allows the use of time as a parameter and supports a hierarchical composition. For the efficient modeling of CPN, several tools have been developed, which can model the CPN approach, and the best known is the CPN Tools [17]. CPN Tools is based on the Standard Meta Language (SML) – which is a functional programming language –but the CPN Tools also extends SML with functions such as color sets and constructs for declaring variables, multisets, and related operators

and functions. SML enables simulation, state-space analysis, and performance analysis [18]. The next section puts our proposal in context with the related work.

6.5 Loose coupling and Late binding of stakeholder systems

Multi-stakeholder industrial ecosystems have dynamic connections with each other, and their processes are interwoven in complex ways. It is a long-term challenge for industrial ecosystems to handle the three dimensions of Digital Production (DP), Product Lifecycle Management (PLM) and Supply Chain Management (SCM) together. Although until recently these were addressed by separate models, methodologies and tools, it would be advantageous to handle them together. In order to reach such an integrated state, we must address the burden of inter-domain communication issues among the experts of the *digital quadruplets*. This means that mechanical and electrical engineers, computer scientists have to work together with experts of the law and the economics domain in order to provide an optimal path to have a harmonised, converged Industry 4.0 playground – not only in the engineering levels but all the four areas of the digital quadruplets: (i) engineering the physical world, (ii) engineering the cyber counterpart of cyber-physical systems (CPS), (iii) handling the regulatory and legal issues of CPS, as well as (iv) taking care of the financial economic relations of the CPS approach.

Let us consider the service oriented architecture abstraction here, in the overall ecosystem. In other words, we should think of the stakeholders (let they be physical devices or enterprises) as they aim for exchanging services, and every one of them is either a service provider, a service consumer, or both. "Services" in this sense are abstract entities, and could be anything that a partner could exchange with another: information, solution, or even a physical asset.

Loose coupling of stakeholder systems mean that the involved devices – or even enterprises – depend on each other to the least possible extent when exchanging services. They can connect to each other "run-time" only for exchanging the given service and then decouple just as easily – no strings attached, they do not have to know anything else about each other.

This leads us to the other important aspect called late binding, which allows not only pre-defined connections "planning-time" or "deployment-time", but the stakeholder elements can bind each other "run-time" without knowing preliminarily who the other counterpart will be.

This function can be made possible through service discovery (or lookup) in the service oriented architecture.

Even when staying within the engineering domains, the optimal, homogeneous solution requires a framework that can dynamically automate processes while taking into account various issues, including the

- creation of new System-of-Systems,
- integration of brand new as well as legacy elements,

- dynamic interoperability between elements,
- seamlessly scalable information sharing among participants – even within different stakeholder domains and security levels,
- real-time requirements,
- reliability and scalability issues,
- Quality of Service issues,
- status handling of – or even the control over – resources,
- ease of engineering processes – in design-time, deployment-time and operational-time,
- safety and security issues.

6.5.1 A brief overview of the main Arrowhead concepts

The current version of this overview is based on our survey [23] on DP, SCM and PLM.

The Arrowhead Framework has originally come to life [46] to cover interoperability and integration issues for the IIoT world. It supports the collaboration of newly built as well as legacy CPS architectures based on the principles of Service-Oriented Architectures (SOA) through applying the System-of-Systems (SoS) approach. One part of the above listed issues are tackled through the Local Cloud concept empowered by inter-cloud communication capabilities. The Arrowhead Framework defines mandatory core systems for the local clouds, which provide the necessary core systems. Further, supportive core systems provide general services that are often needed in System of Systems, so integrators do not have to implement their solutions for such common services. The Application Systems are distinct elements of the SoS, these provide (and in fact, consume) the various application services – in a discoverable, late-bound, loosely coupled way that is defined by the SOA. Figure 8 describes the current, main systems of the Arrowhead Framework.

The mandatory core services are Orchestration System (mainly service discovery and late binding), Service Registry (so services providers can announce their active services), and Authorisation System (to provide Authorisation and Authentication). Further, supporting core services are provided by the Gateway and Gatekeeper Systems for inter-cloud communication (data and control plane, respectively), the Workflow Choreographer (to trigger the next step in the process execution), the Event Handler (to circulate status and event information), and the Plant Description System (to keep track of SoS- or Plant-related metadata), among others. Each stakeholder has their local cloud(s), working as an SoS, their systems implement either intra- or inter-cloud information sharing, as well as security- and other policies.

When taking a closer look at the above-listed requirements, the corresponding citations actually "hide" scientific papers related to Arrowhead Framework.

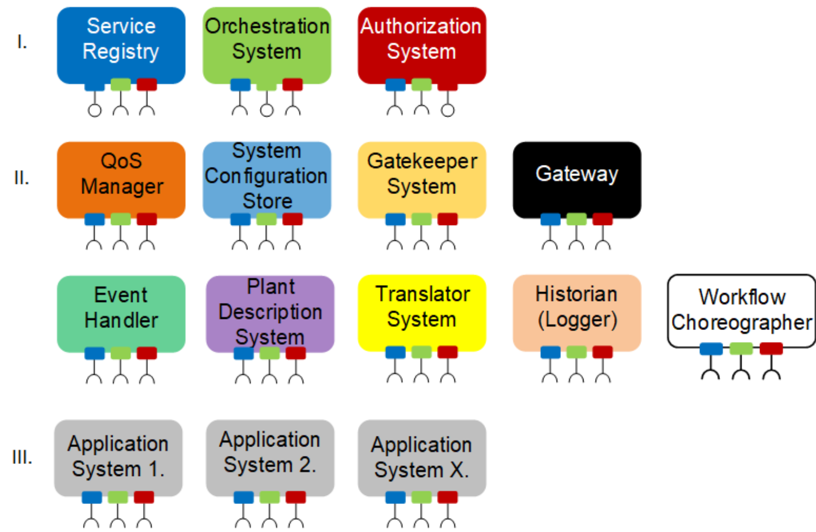

Figure 8: Core Systems of the Arrowhead Framework

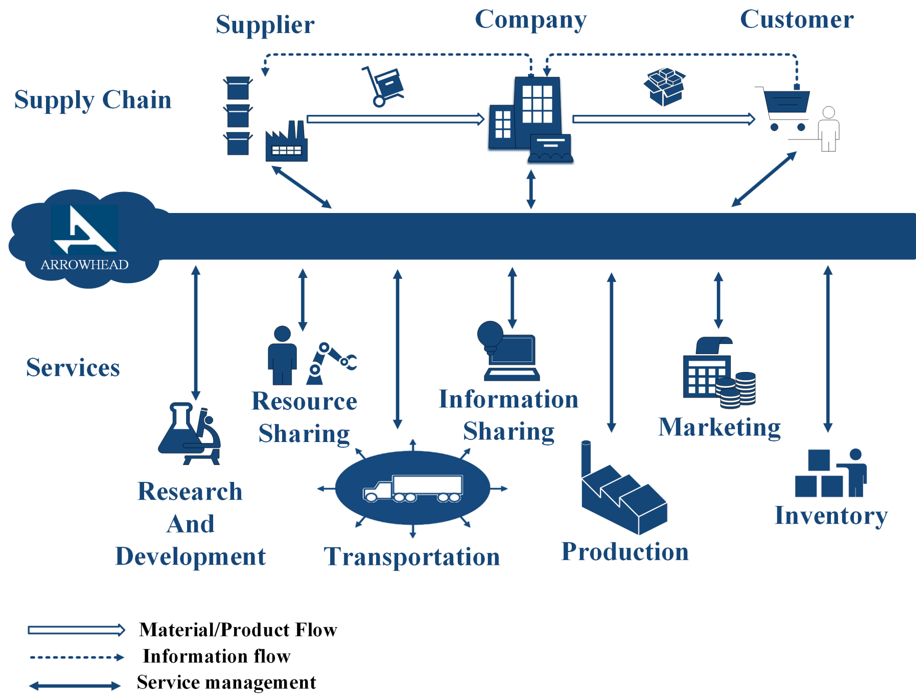

Figure 9: SCM, PLM and DP managed by Arrowhead Framework

The papers surveyed in [23] are not only requirement specifications, but original conceptual descriptions, and reports on implemented solutions.

6.5.2 The Arrowhead Framework to support DP

Digital Production is hidden in Figure 9 – these can be within the Supplier's premises or the Company's premises. Production is listed as one of the services in the figure (which is true for the SCM hierarchy), although there are many systems on the production floor that exchange services between each other – these could be, e.g., fitting, welding, screw-driving, cutting, painting, filling, etc. This infrastructure is by default supported through the Arrowhead Framework – including its Workflow Choreography service –; various application examples have been demonstrated already [22].

One such example is illustrated by Figure 10, showing how the Enterprise level workflows (described through BPMN) and the production-level workflow details (described through CPNs) can get integrated. The production-level CPN execution is controlled by an Arrowhead Workflow Choreographer. It executes the CPN-described recipe by using the Orchestrator to find out which service provider is available and fit for a given task at the given place and time. This loop of steps for choosing the provider at a given execution step is very straightforward, as shown by Figure 11.

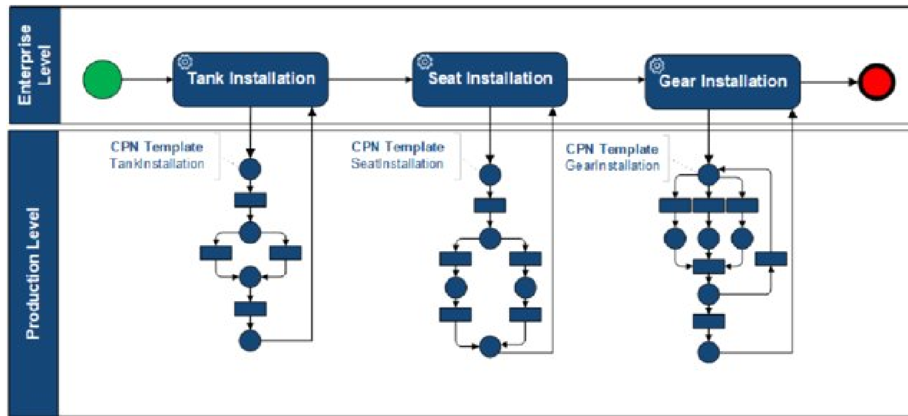

Figure 10: Integration of the BPMN model on enterprise level and the CPN templates on production [22]

6.6 DLT-assisted workflow execution

After providing an overview of workflow modeling, workflow descriptions, dynamic execution of workflows based on the available executor devices, let us take a further step and see how Distributed Ledger Technologies can assist workflow execution.

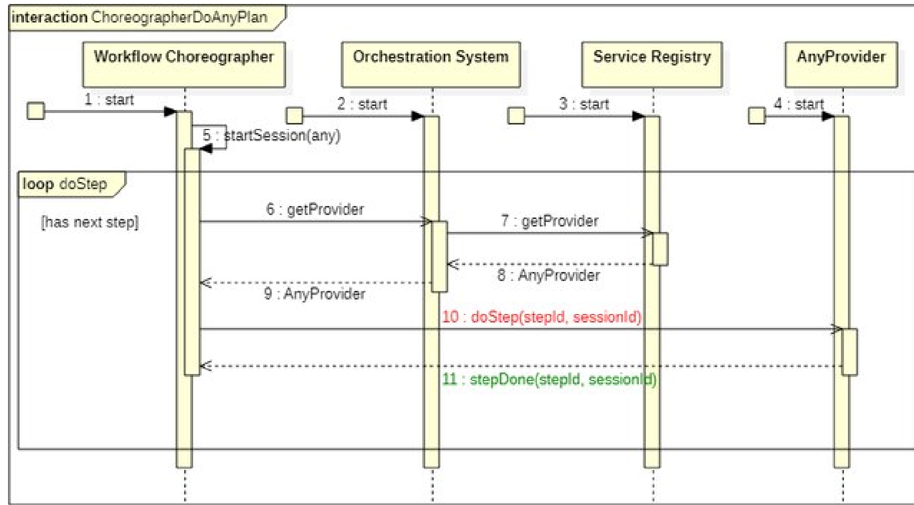

Figure 11: Sequence diagram of executing any production plan by using the Workflow Choreographer of Arrowhead [24]

We are practically talking about workflows described as smart contracts, and smart contracts executed through DLT.

Smart contracts can be abstracted from workflow or process descriptions with additional restrictions or features. An early representation of such an abstraction is described in [40].

A more recent study from J. Mendling et.al. [30] describes the challenges and opportunities that blockchain technologies provide for Business Process Management. They discuss blockchain in relation to the BPM lifecycle, reflection on all its phases including identification, discovery, analysis, redesign, implementation, execution, monitoring, as well and adaptation and evolution. This is a very important overview, since the common focus in on implementation and execution – although all the other BPM phases must be taken into account when using blockchains for either smart contracts or workflow and process control purposes.

Besides discussing BPM capabilities – such as strategy, governance, IT, people and culture – in the light of blockchain technologies, the authors of [30] list seven future research directions on the domain. These are

1. developing execution and monitoring systems,
2. devising new methods for analysis and engineering business process based BCT,
3. redesigning processes to benefit the opportunities provided by BCT,
4. defining methods for evolution and adaptation,

5. developing techniques for the BPM phases to adopt BCT,
6. understanding the impact on strategy and governance of BCT regarding business and governance models,
7. investigating the culture shift towards openness in the management and execution of processes.

When discussing Ethereum blockchains in particular, Hsain et.al. [16] surveyed Model Driven Engineering (MDE) methods for constructing and developing smart contracts. The survey compares the various efforts on using BPMN, DMN (Decision Model and Notation), dependency graphs, finite state machines, object-event tables, smart contracts as transition system, contract action diagrams and many more. Each of the analyzed methods have corresponding Solidity or Hyperledger code which is beneficial for comprehension, although not all the surveyed papers implemented proper security measures, which could be a warning sign: robustness should be addressed at the modeling phase as well.

A well-documented example of the BPMN approach applied to Hyperledger-based smart contacts is described by Panduwinata and Yugopuspito [36], where a reservation-based parking system case is handled through smart contracts but also visually described as a multi-level BPMN process.

6.7 Techniques for making DLT-based workflows robust

There are some already established design patterns that help making DLT-based workflows robust – in this case: in the format of smart contracts. Domain experts keep improving the related patterns, methods and best practices. The following subsections provide a current overview of the state-of-the-art design patters, and although many are targeted to the Ethereum platform, their findings can often be applied universally.

6.7.1 Smart Contract Design Patterns in the Ethereum Ecosystem

In their paper, Wöhrer and Zdun [50] identify and categorize the most widely applied design patterns used in Solidity smart contracts based on the logical role of the individual patterns. These groups are *Action and Control*, *Authorization*, *Lifecycle*, *Maintenance*, and *Security*.

The description of patterns range from the very basic *Ownership* pattern, used by almost every smart contract in some way, that allows for the restriction of access of certain function, to the more advanced *Oracle* or *Commit and Reveal* patterns. The authors describe every pattern through a short overview and demonstrate their usage by a sample contract code, along with the problem that the particular pattern provides a solution to. Overall, this paper [50] identifies 18 design patterns coming from various sources, although only 12 is discussed in detail, covering a variety of use cases and real-world applications. All sample code – for Solidity smart contracts – can be found in [9], together with a brief description of every pattern they identified.

The newly proposed Maintenance Patterns help developers to write smart contracts that can be updated on the blockchain without the need to deploy a whole new contract whenever there is a small part or functionality that is added or changed.

6.7.2 Further Design Patterns for Solidity

The GitHub repository for Solidity Patterns [42] contains very similar practices are described before, and some additional ones, as well. One such example is the *Randomness* pattern, that is the core of many games and gambling applications, although it is very hard to achieve a sufficient level of randomness on the blockchain, due to its intrinsic nature of being a deterministic and transparent environment. The author of [42] identified four groups, namely *Behavioral Patterns*, *Security Patterns*, *Upgradeability Patterns*, and *Economic Patterns*. The latter mentioned one is one about efficiency, containing three patterns that are designed to optimise the gas usage of methods. This aspect is not often discussed, and sometimes overlooked, but it is also a quality of robust smart contracts.

6.7.3 Creational, Structural, and Behavioral Design Patterns

The authors of [26] took yet another approach to the taxonomy of design patterns. They identified 8 design patterns and categorized them into groups called *Creational Patterns*, *Structural Patterns*, *Inter-Behavioral Patterns* and *Intra-Behavioral Patterns*. Five patterns have actually been applied to a real-world, blockchain-based traceability system called originChain. Beside presenting patterns that work together and form a complex structure of smart contracts, the authors of [26] also proposed a related traceability system, as well.

6.7.4 Proxy Patterns for adding modification features to Smart Contract

OpenZeppelin [35] proposed a method to solve the problem of upgrading (or correting) already deployed smart contracts. Note that this is a contradiction, since the immutability feature of the blockchain would naturally prevent the modification of smart contracts. Although deployed contracts still remain immutable; but changing certain variables, writing special functions, and separating logic and storage can enable developers to upgrade features or even whole smart contracts if needed.

The blog post [38] authors proposed several ways to achieve this on OpenZeppelin – they called their set of methods *Proxy Patterns*. The method for *separating storage* makes it possible to update the underlying logic without losing data or needing costly rewrites. The main idea is to use a permanent proxy contract, which is an intermediary between the caller of the called contract, and the called contract itself. This contract does nothing but forward calls to the logic contract and the result back to the caller. It does not change at all, its

address remains the same so the callers do not notice any change when updates happen to the logic contract. When developers need to update or add some functions, they deploy a new logic contract that contains the updated code, and the proxy contract will then communicate with this new contract. This way, the changing of the logic is completely transparent to the callers.

There are three ways proposed to address the related problem. These approaches are called *Inherited Storage*, *Eternal Storage*, and *Unstructured Storage*. Each of them solves the problem, but they have some characteristics which make one preferred over the other, depending on the circumstances. Namely, the structure of the *Unstructured Storage* makes it the best choice in the majority of use cases, because it is fairly easy to implement, and the existing contracts do not need to be modified at all – they can be used with the proxy without any changes. (This approach is very similar to the *Contract Relay Pattern* found in [50], and the *Proxy Delegate* pattern found in [9], however, the *Contract Relay Pattern* is an outdated version that is not recommended to use due to its inability to return result values.) Further details and some important things about the *Unstructured Storage*, and sample contracts can be found in [39] and [45].

7 Summary: possible impacts of regulatory control enablement

To summarize, we conclude with an overview of the potential impact of enabling the regulatory control by distributed ledger technologies, with an emphasis on fault tolerance.

Fault removal and mitigation: Continuous audit In the case of traditional audit processes, faults can be detected only long time after their occurrence, therefore minimizing their impact can be done only by cumbersome compensation actions. Although such actions can also be expressed as special workflows (as supported by e.g., the BPEL and BPMN standards), these are rarely used since the difficulty of designing a correct and effective compensation flow. Therefore, the promise of executing a "continous audit process" can be a strong argument for the application of distributed ledgers.

Fault prevention: Checking / enabling Transactions before they get active or live. Fault prevention can be implemented by including the regulatory body or further participants of the financial infrastructure in the execution of transactions. As there is a possibility to share transactions without revealing the origin/target of the transaction (e.g., based on digital identities and external oracles), suspicious transactions and transaction patterns can be analyzed quasi realtime, which can drastically decease the chance of executing a fraudulent transaction.

Fault prevention: Whitelisting and Blacklisting functionalities can be directly integrated to the business logic. As execution of smart contract steps can be bound to specific roles, the supervisor can act as a distinguished actor to "enable or disable" certain transactions. Smart contract: how to include?

Whether and how different participants of the financial infrastructure should be part of the business logic or used as information source, and exactly what kind of data should be stored on the ledger and managed by smart contracts (including the possibility to offer query to third parties) is an important further research question.

Value-added services on top of CBDC smart contracts. If the smart contract handle different aspects of transactions (and also different "colors" of tokens), a number of value added services can be created as a layer on top of the "flow of money". In the case of industrial IoT, for example, applying certain methods in manufacturing (like a low-emission machinery step) can result "bonuses" (e.g., reduction of environmental tax). The conditions for such services can be regularly updated and the effect of this "upper layer" of smart contracts can be evaluated and compared with the original goal of the policy maker.

Platform: whether and how to include the regulator and/or central bank The level and method of including the regulatory in the blockchain network is an important design decision. A possible design pattern can be to share only aggregated information, when even without the detailed view, the regulatory can follow and monitor the transaction flow. On the other hand, performance itself should not be an obstacle; depending on the resolution of information and the definition of "transaction" in the smart contracts (chaincode), current technologies promise a potential throughput which even supports the central bank in participating in a number of blockchain networks.

References

[1] Naveed Ahmad, David C Wynn, and P John Clarkson. "Change impact on a product and its redesign process: a tool for knowledge capture and reuse". In: *Research in Engineering Design* 24.3 (2013), pp. 219–244.

[2] Mouhamad Almakhour et al. "Verification of smart contracts: A survey". In: *Pervasive and Mobile Computing* (2020), p. 101227.

[3] Algirdas Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.

[4] Karim Baına and Salah Baına. "User experience-based evaluation of open source workflow systems: The cases of Bonita, Activiti, jBPM, and Intalio". In: *2013 3rd International Symposium ISKO-Maghreb*. IEEE. 2013.

[5] Betsy Beyer et al. *Site reliability engineering: How Google runs production systems.* " O'Reilly Media, Inc.", 2016.

[6] Nadja Brouns et al. *Modeling IoT-aware Business Processes - A State of the Art Report*. 2018.

[7] Jiachi Chen et al. "Defining Smart Contract Defects on Ethereum". In: *IEEE Transactions on Software Engineering* (2020). DOI: 10.1109/TSE.2020.2989002.

[8] Søren Christensen, Lars Michael Kristensen, and Thomas Mailund. “Condensed state spaces for timed Petri nets”. In: *International Conference on Application and Theory of Petri Nets*. Springer. 2001, pp. 101–120.

[9] *Design Patterns*. `https://github.com/maxwoe/solidity_patterns`. Accessed: 2021-07-28.

[10] Yanhua Du, Benyuan Yang, and Hesuan Hu. “Incremental Analysis of Temporal Constraints for Concurrent Workflow Processes With Dynamic Changes”. In: *IEEE Transactions on Industrial Informatics, Vol 15, NO. 05, Pp 2617-2627 (2019)* 15 (2019), p. 10. DOI: `10.1109/TII.2018.2868810`.

[11] Vimal Dwivedi et al. “A Formal Specification Smart-Contract Language for Legally Binding Decentralized Autonomous Organizations”. In: *IEEE Access* 9 (2021), pp. 76069–76082.

[12] Josselin Feist, Gustavo Grieco, and Alex Groce. “Slither: A Static Analysis Framework for Smart Contracts”. In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)* (May 2019). DOI: `10.1109/wetseb.2019.00008`.

[13] D. Habhouba, S. Cherkaoui, and A. Desrochers. “Decision-Making Assistance in Engineering-Change Management Process”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 41.3 (May 2011), pp. 344–349.

[14] Ákos Hajdu et al. “Using Fault Injection to Assess Blockchain Systems in Presence of Faulty Smart Contracts”. In: *IEEE Access* 8 (2020), pp. 190760–190783.

[15] Michael Havey. *Essential business process modeling*. O’Reilly Media, Inc., 2005.

[16] Yassine Ait Hsain, Naziha Laaz, and Samir Mbarki. “Ethereum’s Smart Contracts Construction and Development using Model Driven Engineering Technologies: a Review”. In: *Procedia Computer Science* 184 (2021), pp. 785–790.

[17] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. “Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems”. In: *International Journal on Software Tools for Technology Transfer* 9.3-4 (2007), pp. 213–254.

[18] Kurt Jensen and Grzegorz Rozenberg. *High-level Petri nets: theory and application*. Springer Science & Business Media, 2012.

[19] Kurt Jensen et al. *Transactions on Petri Nets and Other Models of Concurrency VII*. Vol. 7480. Springer, 2013.

[20] Bo Jiang, Ye Liu, and W. K. Chan. “ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Association for Computing Machinery, 2018, pp. 259–269. ISBN: 9781450359375. DOI: `10.1145/3238147.3238177`.

[21] Dániel Kozma, Pál Varga, and Felix Larrinaga. "Data-driven Workflow Management by utilising BPMN and CPN in IIoT Systems with the Arrowhead Framework". In: *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2019, pp. 385–392.

[22] Dániel Kozma, Pál Varga, and Felix Larrinaga. "Dynamic multilevel workflow management concept for industrial iot systems". In: *IEEE Transactions on Automation Science and Engineering* (2020).

[23] Dániel Kozma, Pál Varga, and Gábor Soós. "Supporting digital production, product lifecycle and supply chain management in industry 4.0 by the arrowhead framework–a survey". In: *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*. Vol. 1. IEEE. 2019, pp. 126–131.

[24] Dániel Kozma, Pál Varga, and Kristóf Szabó. "Achieving Flexible Digital Production with the Arrowhead Workflow Choreographer". In: *IECON 2020 The 46th Annual Conference of the IEEE Industrial Electronics Society*. 2020, pp. 4503–4510. DOI: 10.1109/IECON43393.2020.9254404.

[25] Ajay Krishna, Pascal Poizat, and Gwen Salaün. "Checking business process evolution". In: *Science of Computer Programming* 170 (2019), pp. 1–26.

[26] Yue Liu et al. "Applying Design Patterns in Smart Contracts". In: June 2018, pp. 92–106. ISBN: 978-3-319-94477-7. DOI: 10.1007/978-3-319-94478-4_7.

[27] Loi Luu et al. "Making Smart Contracts Smarter". In: CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 254–269. ISBN: 9781450341394. DOI: 10.1145/2976749.2978309.

[28] Jakob Mass, Chii Chang, and Satish N. Srirama. "Workflow Model Distribution or Code Distribution? Ideal Approach for Service Composition of the Internet of Things". In: *Proceedings - 2016 IEEE International Conference on Services Computing, SCC 2016*. 2016. ISBN: 9781509026289. DOI: 10.1109/SCC.2016.90.

[29] Izhar Mehar et al. "Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack". In: *Journal of Cases on Information Technology* 21 (Jan. 2019), pp. 19–32. DOI: 10.4018/JCIT.2019010102.

[30] Jan Mendling et al. "Blockchains for business process management-challenges and opportunities". In: *ACM Transactions on Management Information Systems (TMIS)* 9.1 (2018), pp. 1–16.

[31] Alexander Mense and Markus Flatscher. "Security Vulnerabilities in Ethereum Smart Contracts". In: *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications & Services*. iiWAS2018. Association for Computing Machinery, 2018, pp. 375–380. ISBN: 9781450364799. DOI: 10.1145/3282373.3282419.

[32] Tadao Murata. "Petri nets: Properties, analysis and applications". In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580.

[33] *Mythril*. URL: https://github.com/ConsenSys/mythril. (accessed: 05.07.2021).

[34] *Mythril modules for vulnerability analysis*. https://mythril-classic.readthedocs.io/en/master/module-list.html. accessed: 06.07.2021.

[35] *OpenZeppelin*. https://openzeppelin.com/. Accessed: 2021-07-30.

[36] Frans Panduwinata and Pujianto Yugopuspito. "BPMN Approach in Blockchain with Hyperledger Composer and Smart Contract: Reservation-Based Parking System". In: *2019 5th International Conference on New Media Studies (CONMEDIA)*. 2019, pp. 89–93. DOI: 10.1109/CONMEDIA46929.2019.8981845.

[37] András Pataricza et al. "Cost Estimation for Independent Systems Verification and Validation". In: *Certifications of Critical Systems-The CECRIS Experience. River Publishers* (2017), p. 117.

[38] *Proxy Patterns*. https://blog.openzeppelin.com/proxy-patterns/. Accessed: 2021-07-30.

[39] *Proxy Upgrade Pattern*. https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies. Accessed: 2021-07-30.

[40] Holger Schmidt. "Service contracts based on workflow modeling". In: *International Workshop on Distributed Systems: Operations and Management*. Springer. 2000, pp. 132–145.

[41] *Slither, the Solidity source analyzer*. URL: https://github.com/crytic/slither. (accessed: 05.07.2021).

[42] *Solidity Patterns*. https://github.com/fravoll/solidity-patterns. Accessed: 2021-07-28.

[43] Ping Sun and Changjun Jiang. "Analysis of workflow dynamic changes based on Petri net". In: *Information and Software Technology* 51.2 (2009), pp. 284–292.

[44] Palina Tolmach et al. "A survey of smart contract formal specification and verification". In: *arXiv preprint arXiv:2008.02712* (2020).

[45] *Upgradeability using Unstructured Storage*. https://blog.openzeppelin.com/upgradeability-using-unstructured-storage/. Accessed: 2021-07-30.

[46] Pal Varga et al. "Making system of systems interoperable–The core components of the arrowhead framework". In: *Journal of Network and Computer Applications* 81 (2017), pp. 85–95.

[47] Pál Varga, Dániel Kozma, and Csaba Hegedűs. "Data-Driven Workflow Execution in Service Oriented IoT Architectures". In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. IEEE. 2018, pp. 203–210.

[48] Stephen A White. "Introduction to BPMN". In: *IBM Cooperation* (2004).

[49] Maximilian Wohrer and Uwe Zdun. "Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity". In: *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. 2018, pp. 2–8. DOI: 10.1109/IWBOSE.2018.8327565.

[50] Maximilian Wöhrer and Uwe Zdun. "Design Patterns for Smart Contracts in the Ethereum Ecosystem". In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 2018, pp. 1513–1520. DOI: 10.1109/Cybermatics_2018.2018.00255.