NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

RONALDO LUDGERO ABREU DA CORTE

Bachelor in Computer Science

SMART TYPES FOR SMART CONTRACTS VALIDATION

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon

⟨December⟩, ⟨2022⟩

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

SMART TYPES FOR SMART CONTRACTS VALIDATION

RONALDO LUDGERO ABREU DA CORTE

Bachelor in Computer Science

Adviser: António Ravara
Associate Professor, NOVA University Lisbon

Co-adviser: Mário Pereira
Researcher, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
⟨December⟩, ⟨2022⟩

Smart Types for Smart Contracts Validation

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.4) [29].

Acknowledgements

It has been a long journey and I would like to thank some people without whom this work would not be possible.

I would like to start by giving special thanks to my Professor António Ravara and Doctor Mário Pereira for all their help, moments and opportunities.

I thank my friends for all the moments, specially Henrique for every adventure since first year.

I would like to thank my mother and father for all the help and support. I would also like to thank my sister for the encouragement and help since day one.

Finally I would like to thank my girlfriend Carlota, who was always there for me and for making every little moment better.

Work partially supported by the EU H2020 programme under the Marie Skłodowska-Curie grant agreement 778233 (BehAPI).

Abstract

The notion of Smart Contracts consist in describing agreements between two or more parties that can be automatically enforced without a trusted intermediary. Smart Contracts run on a very specific network of peers called Blockchain, a a digitally distributed, decentralized, public ledger that exists across a network. Potential conflicts are resolved by the network's consensus protocol.

The Blockchain [26] is immutable, this means that once a Smart Contract is deployed on the Blockchain it cannot be amended. This immutability (despite being one important selling point of Smart Contracts) leave no room for mistakes in their implementation.

Many contracts are hard to implement correctly and bugs and vulnerabilities can be exploited for erroneous or even fraudulent behaviour.

The countless advantages and applications of Smart contracts are constantly increasing their popularity. This added to the fact that Smart Contracts manipulate resources with monetary value is bringing a lot of attention to attackers. There are a lot of infamous Smart Contracts attacks, the DAO Attack per example drained millions of dollars in Ether (cryptocurrency of Ethereum).

Mainstream tools used to develop distributed Smart Contracts do not address these requirements. Consequently, many vulnerabilities of these contracts are known and can be exploited.

In order to help developers to design safer contracts that follow their protocols and specifications we propose a language integrated with assertions and a static behavioural type system able of protecting resources and enforce usage protocols to ensure the safety and soundness in Smart Contracts execution.

Since proof assistants are too demanding for most developers, there is a need for automatic tools well integrated with programming languages. Therefore, we joined our language with a model-checker to discharge to it the quantitative assertions during the compilation process. In short, we provided a translation of the types and assertions to an automaton in the format of Cubicle's (model checker) input language and used this one to conduct Software Verification.

Keywords: Smart Contracts, Blockchain, Programming Languages, Safety, Soundness, Formal-Methods, Typechecking

Contents

List of Figures

List of Listings

1 Introduction

1.1 Motivation

A Blockchain is a distributed infrastructure which comprises of peer nodes. Each node stores information state of the network, a structure named ledger, that has the information about the order of the transactions that were made within the network. Every transaction executed in the Blockchain has to be validated by the network through a consensus protocol which assures that every node has the same ledger's copy.

Smart Contracts are critical pieces of software that automatically verify and enforce contractual agreements without the need of a trusted intermediary. These programs are deployed on the Blockchain.

Smart contracts are beneficial because once deployed, they cannot be changed (the Blockchain is immutable) and they do not require the authority of a third party to verify their authenticity. This allows the parties involved in the contract to save time and money, for a particular transaction.

Smart Contracts run in very demanding environments: the manipulated resources have monetary value and the programs implement often intricate stateful contracts, with several causal-dependent steps. Many Contracts are hard to implement correctly so the developer has no room for mistakes as bugs and vulnerabilities can be exploited.

Contracts should be correct so tools that can provide automation, correctness and safety guarantees during Smart Contracts development are highly appreciated. Formalising and verifying Blockchain systems is crucial to provide soundness guarantees. The effort is ongoing, but only starting. Works address particular systems and/or particular properties, lacking generality, and demand complex interactive human intervention, lacking automation. Many approaches focus on constructing models of aspects of Blockchain frameworks and proving them correct. This model-based approach is important but insufficient for the overall purpose. We believe one should not forget the code - in the end, the implementations must be correct or everything fails.

In short, the lack of comprehensive formal approaches providing, as automatically as possible, soundness guarantees not only for the infrastructure implementation (blocks

creation, and transaction semantics and consensus protocol to add blocks to the chain), but also for the code of the contracts leaves a lot of room for improvement and to develop tools that can provide these guarantees.

1.2 Problems

Because of the multiple advantages and applications of Smart Contracts, their popularity is constantly rising. As we mentioned above, Smart Contracts manipulate resources with monetary value. These are all big reasons to attackers try to exploit the vulnerabilities of Smart Contracts.

There is a lot of known attacks to Smart Contracts, per example the DAO Attack that drained million of dolars in Ether, cryptocurrency of Ethereum.

One of the strong and appealing aspects of Blockchain is immutability (as it prevents fraud), but it is also a big challenge. Once a contract is deployed he can't be amended what brings a lot of concern on Smart Contracts development because if a contract has some mistakes, after deployed it cannot be changed. However, the number of bugs reported and of attacks is staggering, with huge losses acknowledged. The Blockchain organisations recognise the problem and are gradually embracing formal verification techniques to validate their systems and support their evolution.

Smart Contracts are written in general purpose programming languages that do not enforce any type of specification of correctness. Additionally, many vulnerabilities of these contracts have been discovered and exploited, like reentrancy and incorrect behaviour order.

For these reasons many domain-specific languages are being developed to help programmers to avoid common mistakes, ensuring some safety guarantees. However, the majority of these languages cannot provide the needed guarantees.

The research proposals for languages handling the Smart Contracts states and states transitions explicitly do not statically guarantee protocol compliance and completion, resource protection, bounded computational cost and functional requirements altogether while considering a transactional semantics, which is the appropriate tool to describe contracts running on Blockchains governed by consensus protocols.

Thus, the research problem that we addressed focused on providing static correctness guarantees and preventing smart contracts execution errors.

1.3 Goals and Contributions

Our main goal for the project was to guarantee the correctness of Smart Contracts.

Deductive verification lacks of full automation and requires human intervention to creatively devise the soundness proof. Our aim was to find an automatic approach, still language-based (i.e., using the code of the program instead of a abstract model of it like in model-checking) and performed statically (i.e., at compile-time), that uses dependent

types. Instead of just using assertions as contracts on the code (pre and post-conditions, as well as invariants), one used assertions to limit the values of a type (like positive integers for example).

Cubicle [19] incorporates state-of-the-art language and verification techniques (e.g. ghost variables, backward reachability, parallelisation on multi-cores). Because of all these reasons, Cubicle was particularly relevant for our project:

1. The model language combines the benefit of a compact formal model with the power of a pushdown automaton that allows to describe open, array-based systems indexed by first-class process entities.

2. The logic embedded in the model language seemed suitable to describe functional properties of Smart Contracts as well as their interaction with the underlying Blockchain and consensus protocol.

Therefore, our contribution consisted in designing an assertion language integrated with a type-system, appropriated for expressing desired properties of smart contracts, more expressive than those used in dependent type systems yet amenable to automatic verification by being treatable by SMTs and BMC.

We also contributed with the report of some bugs found in Smart Contracts implementations from [13].

In short, our contribution is a DSL to program Smart Contracts equipped with:

- A quantitative behavioural type system to ensure statically (in a lightweight way), that well-typed digital contracts do not go wrong.

- An automatic assertion verification system integrated with the compilation process, supported by the automata-based model checker, Cubicle.

This contribution is available in a github online repository [20]. To a user be able of using this contribution, he must clone the repository and use the files *MiniFS*, *Parser-MiniCub* and *ParserCub* so he can write a MiniFS Smart Contract, typecheck it and finally, verify its specifications with a model checker.

2 Background

To provide the necessary context about this master plan, we present multiple concepts about Distributed Systems, Blockchain, Smart Contracts, Software Verification and finally, Type Systems.

2.1 Distributed Systems

In this section we present the basic concepts about Distributed Systems (DS), the CAP properties of a Distributed System and lastely, we present the general failures of these systems.

2.1.1 Distributed System

A Distributed System [21] is composed by a set of processes that are interconnected through some network where processes (that are located on different physical machines) seek to achieve some form of cooperation (by communicating/interacting with each other) to execute tasks and achieve a common goal.

In short, these processes help in sharing different resources and capabilities to provide users with a single and integrated coherent network.

2.1.2 CAP Properties of a Distributed System

The main goals of a Distributed Systems consists in satisfying some properties like Consistency, Availability and Partition-Tolerance.

However, the CAP Theorem provide a theoretical basis for the inability of a Distributed System to obtain consistency, availability, and partition tolerance simultaneously [25].

- Consistency: Consistency requires that no two nodes in a network provide a different state at any point in time and that no node returns a non-current state.
- Availability: Availability is the time a system remains functional to perform its required task in a specific period.

- Partition-tolerance: A partition describes the inability of two or more nodes of a network to communicate. Partition tolerance refers to a network's ability to continue to operate despite the presence of partitions.

Web services attempt to achieve consistency by relying on ACID properties [25]:

- Atomic: Either the operations are committed or fail in their entirety.
- Consistent: All transactions must result in consistent data.
- Isolated: Uncommitted transactions are isolated from each other.
- Durable: Once a transaction is committed it is permanent.

Likewise, web services are also expected to be always available and on a distributed network, if any component fails, it should still perform as expected.

In short, since a Distributed System cannot fully satisfy the CAP properties simultaneously, he must chose which ones are more important and helpful to achieve his goal.

2.1.3 Failures

Distributed Systems should be Fault-Tolerant (be able to continue operating uninterrupted despite the failure of one or more of its components) and so, it's important to address his failures and how to solve them.

Distributed systems can fail mainly because of two types of nodes failures.

- The first one is when a node halts/crashes or become unreachable i.e a crash fault.
- The second one is when there are nodes with malicious behavior where nodes cannot fully trust each other. Such faults are illustrated by the Byzantine Generals Problem.

2.1.3.1 Byzantine Generals Problem

The Byzantine Generals Problem [37] address the failure tolerance situation where a some network components try to reach a consensus under an attack of other malicious peers. Basically this problem describes how a reliable system should deal with faulty components that send messages to other components with the purpose of misleading them. This problem can only be solved if more than 2/3 of the components are honest and the follow properties are satisfied [28]:

- All honest components must decide the same plan while malicious ones can behave in anyway and the honest components must come up with a reasonable plan. For these conditions to be satisfied, the following must be true:

  1. Any two honest generals must get the same value as the other honest general.

2. If the general is honest, then the value that he sends must be used by any other honest general that receives it.

- A small number of traitors will not be able to cause the honest generals to follow the malicious plan. As the final decision is based on the majority of the votes, a small number of traitors can influence the decision of the honest ones but only if they are torn between two possible plans, in which case both plans are reasonable.

2.1.4 Consensus

A Distributed System is said to achieve consensus if it guarantees a uniform state that is consistent with the operations directed at any correct node. To a Distributed System be consistent and also to always take the most beneficial choices, achieving consensus is fundamental.

Because of malicious actors and faulty processes, reaching consensus in a network might be problematic.

To solve this problem there is some consensus protocols that enforces that in the end every network peer will have the same correct state.

2.1.4.1 Broadcast Consensus

Broadcast Consensus is one of the many examples of consensus protocols.

The consensus problem

- Each process has an initial value v that he proposes.
- Each correct process eventually decides a value.
- Some properties like Termination, Validation, Integrity and Agreement must be satisfied.

Broadcast Consensus Intuition This algorithm progresses in rounds and each process knows:

- In which round it is (variable round).
- If he already decided (variable decision).
- The set of correct processes in previous rounds (initially, at round zero, all processes are considered correct).
- The set of all proposals known in each round (initially at round zero, a process only knows its proposal).
- Algorithm starts at round 1.

- In each round, each process broadcasts the set of proposals that it got in the previous round.
- A round terminates when a process gets a message from every correct process in that round.
- A decision is made when a message is obtained from all correct processes and no new crashed process is detected during a round.
- The decision is obtained by applying a deterministic function over the set of all proposals.

Termination In the Termination property, each process every correct process eventually decides a value. At most in round n, all processes decide:

1. Processes that does not fail will keep moving from round to round (failures will be detected for sure).
2. In The worst case a processes fails in each round.
3. There are n processes in the system and only f processes can fail with $f < n$.

Validity: Follows from the algorithm.

1. To decide a value, that value has to be in the set of known proposals.
2. For the value to be in that set, then some process broadcasted that value in its set.
3. The only way to add a value to the set of proposals is by either receiving it from another process or if the value was proposed by that same process.

Integrity: Follows from the algorithm.

1. A decision can only be made if decided has a bottom value.
2. When a process decides, its decided state becomes the decided value (and hence no longer bottom).

Agreement By construction of the algorithm:

1. All processes that reach round n will have the same set of values in their local proposal-set.
2. Since the value to be decided is picked using a deterministic function, all process will decide the same value.

2.2 Blockchain

A Blockchain is a ledger of blocks of information (e.g., a group of transactions or agreements) that are stored sequentially across a network of computers [26].

In a Blockchain each block is chained together with the previous one by including its hash so we can consider a Blockchain an ordered linked list of blocks. This one is distributed and maintained by a network of nodes where each node stores his own local copy of the Blockchain.

The process of appending a new block is called mining. In this process a set of nodes (miners) try to build a new block to be appended to the Blockchain [26]. To avoid conflicts, a block can only be added to the Blockchain if it has been validated by a consensus protocol, and once a block is added to the chain it cannot be changed, making Blockchain's ledger immutable.

The goal of Blockchain is to allow digital information to be recorded and distributed, but not edited. In this way, a Blockchain is the foundation for immutable ledgers, or records of transactions that cannot be altered, deleted, or destroyed. This is why Blockchains are also known as a distributed ledger technology (DLT).

One of the most important properties of Blockchain is decentralization. Blockchain allows the data to be spread out among several network nodes at various locations. This not only creates redundancy but also maintains the fidelity of the data stored and if somebody tries to alter a record at one instance of the ledger, the other nodes would not be altered and thus would prevent a bad actor from doing so.

This system helps to establish an exact and transparent order of events. This way, no single node within the network can alter information held within it.

Because of this, the information and history (such as of transactions of a cryptocurrency) are irreversible. Such a record could be a list of transactions (such as with a cryptocurrency), but it also is possible for a Blockchain to hold a variety of other information like legal contracts, state identifications, or a company's product inventory.

Because of this property, Blockchain achieves decentralized security.

2.2.1 Types Of Blockchain

According to the way that new network participants join a network, Blockchains can be classified into three main categories [37]:

1. **Public Blockchain**: Anyone can join the Blockchain. In this one, new participans are unvetted and no one can exclude a node from the network. Anyone can participate in reading, writing, and verifying the Blockchain.

   Public Blockchains are open and transparent. Each node in the network can review any entry from any time.

   Two of the most famous examples of Public Blockchains are Bitcoin and Etheureum.

2. **Private Blockchain**: The reading permissions being open to anyone but the writing permissions are restricted to a single participant (or organization).

   The most common examples of Private Blockchains are Ripple (XRP) and Hyperledger.

3. **Consortium-controlled Blockchain**: Has some constraints when it comes to writing permissions, only pre-selected participants of the network can influence the consensus procedure, whilst reading permissions are given to any participant.

   Examples of Consortium Blockchains would be: Quorum and Corda.

2.2.2 CAP Properties in Blockchain

As a Distributed System, Blockchain also tries to achieve the CAP Properties [37]. In Blockchain these properties means:

- Consistency: All peer nodes have the same ledger with the most recent update.
- Availability: Any transaction made will be accepted by the ledger.
- Partition Tolerance: If any node fails, the network can still operate.

2.2.3 Blockchain Consensus Protocols

To reach a consensus in the network in order to validate new blocks added to the chain and to make sure that every peer has identical copies of the ledger, a consensus protocol is required [26]. To achieve the consensus in a network many different protocols (with different pros and cons) can be used, being between the most common ones the **Proof of work** and the **Proof of state** consensus protocol.

2.2.3.1 Proof of work

This protocol requires peers (miners) to spend a non-trivial amount of processing resources solving a puzzle to be eligible to mint a block [37] .

Miners must find some value n such that Hn || H(b) < k, for some predetermined k. In other words, they must iteratively test integers until a number is found that, when concatenated with the block b's hash and the subsequent result is fed into a hash function as input, produces a hash value that is smaller than some parameter k. The first peer to submit the block along with asuitable value n is selected as that block's minter.

In short, Proof of work is a consensus mechanism used to confirm that network participants, called miners, calculate valid alphanumeric codes called hashes to verify Bitcoin transactions and add the next block to the Blockchain. Bitcoin is one Blockchain technology that uses this consensus protocol.

2.2.3.2 Proof of stake

Unlike PoW, this protocol doesn't depend on incentives to guarantee security, it promotes penalty-based solutions. Only participants "who have locked up their capital as deposits" (stake) can be chosen to be miners or validators [37] .

Anyone can become a participant as long as they send "a special type of transaction to lock up a certain amount of their coins". The Blockchain maintains a record of the set of validators that have shown proof of stake.

To add a block in the chain, each validator will place a bet on the block in order to qualify as validator for the block. If the block gets added to the chain, then all the validators that bet on it will be rewarded.

So, unlike PoW in which security is achieved by rewarding the "burning of computational energy", in PoS security is ensured by penalizing the ones that cause economic losses.

The biggest proof-of-stake Blockchains by market capitalization at this momment are Cardano, Avalanche, Polkadot and Solana.

2.2.4 Real World Applications

Due to important properties of the Blockchain such as Immutability and Decentralization, this one has a lot of applications in different sectors (since payment methods to secure sharing of medical data) [37]. Some real word applications of Blockchain are:

- Currencies: Cryptocurrencies are the most prominent current applications of the Blockchain technology. Bitcoin and Etheureum are the most famous among these digital currencies. These currencies have also significantly contributed to popularizing Blockchain as a concept.

- Cross-border Transfers: Cross-border payments are another area in which Blockchain technology has been applied with notable early success.

  The most famous example of Blockchain Cross-border Payment is Ripple (a payment settlement asset exchange and remittance system which is used by banks and financial middlemen dealing across currencies).

- Tokenization: Tokenization describes the process of converting rights to an asset into digital tokens. Everledger is one example of a Blockchain application in this field.

- Asset Tracking: Provenance is the history of the ownership and transmission of an object. Experts are interested in the provenance of an item for several reasons, the most important of which is that well-documented provenance helps confirm that an item is authentic.

- Commodity Trading: Commodity trading is the buying, selling and trading of commodities. Omega Grid (Blockchain energy rewards platform) is one of the most famous Blockhain applications in this field

2.2.5 Advantages and Disadvantages of Blockchain

In this section we present the pros and cons of Blockchain.

2.2.5.1 Benefits

Accuracy of the Chain Transactions on the Blockchain network are approved by a network of thousands of computers. This removes almost all human involvement in the verification process, resulting in less human error and an accurate record of information. Even if a computer on the network were to make a computational mistake, the error would only be made to one copy of the Blockchain. For that error to spread to the rest of the Blockchain, it would need to be made by at least 51 percent of the network's peers.

Cost Reduction Blockchain eliminates the need for third-party verification, and consequently, their associated costs.

Decentralization By spreading that information across a network, rather than storing it in one central database, Blockchain becomes more difficult to tamper with. If a copy of the Blockchain fell into the hands of a hacker, only a single copy of the information, rather than the entire network, would be compromised.

Efficient Transactions In Blockchain, transactions can be completed in as little as 10 minutes and can be considered secure after just a few hours. This is particularly useful for cross-border trades, which usually take much longer because of time zone issues and the fact that all parties must confirm payment processing.

Private Transactions When a user makes a public transaction, their unique code—called a public key, as mentioned earlier—is recorded on the Blockchain. Their personal information is not. If a person has made a Bitcoin purchase on an exchange that requires identification, then the person's identity is still linked to their Blockchain address—but a transaction, even when tied to a person's name, does not reveal any personal information.

Secure Transactions Once a transaction is recorded, its authenticity must be verified by the entire Blockchain network.

After a computer has validated the transaction, it is added to the Blockchain block. Each block on the Blockchain contains its own unique hash, along with the unique hash of the block before it. When the information on a block is edited in any way, that block's hash code changes—however, the hash code on the block after it would not. This discrepancy

makes it extremely difficult for information on the Blockchain to be changed without notice.

2.2.5.2 Drawbacks

Illegal Activity While confidentiality on the Blockchain network protects users from hacks and preserves privacy, it also allows for illegal trading and activity on the Blockchain network.

Scalability Scalability refers to the time needed for propagating, processing, and validating transactions. The higher the number of nodes is, the more limiting network bandwidth, overall storage space, and power consumption become.

Power consumption The Blockchain power consumption is very demanding especially if this one has heavy consensus protocols (Proof of work per example).

2.3 Smart Contracts

A smart contract is a program whose code is stored in a Distributed Blockchain structure and that directly controls digital assets without relying on a third-party intermediary [26]. When these contracts are invoked, they check if the contract conditions are satisfied and if so, the contracts generate transactions that are recorded on the ledger.

The program determines autonomously if the contract conditions were met by retrieving external data using predefined application programming interfaces (APIs) as a source, and executes a contract (e.g., a value transfer) accordingly.

Smart contracts can facilitate or even fully automate insurance processes, financial instruments, and legal processes that are currently heavily paper-based, long-winded, and expensive.

These contracts run in very demanding environments: the manipulated resources have monetary value and the programs implement often intricate stateful contracts, with several causal-dependent steps.

The Ethereum smart contract platform has gained the most momentum so far. The Ethereum project runs a Public Blockchain platform that is similar to but separate from the Bitcoin Blockchain, with extensive, Turing-complete smart contract functionality. Ethereum smart contracts are generally written in Solidity, a JavaScript-like language that compiles to EVM (Ethereum Virtual Machine) bytecode, used for storing and execution.

In Etheureum smart contracts each instruction that the virtual machine executes has a price and the cost of running a smart contract code is measured in units called gas. Charging execution costs addresses the problem of infinite execution times as any infinite loop will eventually be terminated when the contract runs out of gas.

Because of this, when an originator launches a smart contract, he must always set the gas limit to indicate the maximum gas that the smart contract may consume during its execution.

Ideally, the program terminates before the gas limit is reached, and the originator pays exactly the gas the smart contract consumes. Gas is awarded to the miner who successfully packs the transaction into a block and executes its code.

2.3.1 Real World Applications

Smart contracts can facilitate or even fully automate insurance processes, financial instruments, and legal processes that are currently heavily paper-based, long-winded, and expensive.

One of the sectors that most benefits from the Blockchain technology is the financial and banking ones, as its immutable properties make the Blockchain perfect for storing financial information and records. Additionally, the settlement and clearing process must guarantee that the money was indeed transferred and that all parties involved in the exchange updated their respective accounts. This can be ensured by smart contracts as they enforce and verify if all participants respected the agreements in the contract. Besides, by doing this automatically, it reduces the risk of human error. Another area in which smart contracts are being used is healthcare, as they provide reliable and easy access to patients' data. Some applications, like Ethereum's MedRec that try to tackle the issue of data scatter throughout multiple organisations by having a single platform where you can keep and access every patient's record.

2.3.2 Problems

The constant rise of smart contracts popularity and the fact of these programs deals with valuable assets attracts a lot of interest in expoliting smart contracts vulnerabilities, and consequentely programmers must be extra wary of attacks that aim to steal or tamper with them, as these vulnerabilities cause a lot of money loss.

Most of the languages/tools used to develop smart contracts are not well prepared/designed to ensure safety and correctness in this ones. This only creates even more vulnerabilites to be exploited.

There's a lot of vulnerabilites like **Exception Disorder**, **Gasless Send**, **Reentrancy** between many others.

The **Reentrancy** vulnerabilty is one of the most popular and dangerous one as it was the one that led to the infamous **DAO Atack**.

2.3.2.1 Reentrancy

The atomicity and sequentiality of transactions may induce programmers to believe that, when a non-recursive function is invoked, it cannot be re-entered before its termination.

This may not be the case because the fallback mechanism may allow an attacker to re-enter the caller function. This may result in unexpected behaviours, and possibly also in loops of invocations which eventually consume all the gas.

For example lets assume that the contract Bob is already on the Blockchain when the Attacker publishes the Mallory contract.

```
1  contract Bob {
2      bool sent = false;
3
4      function ping(address c) {
5          if (!sent) {
6              c.call.value(2)();
7              sent = true;
8          }
9      }
10 }
```

Figure 2.1: Example of reentrancy vulnerability. Bob contract.

The function ping is used to send exactly 2 values to an address c, using a call with empty signature and no gas limits. Lets assume that ping has been invoked with Mallory's address. The call has the side effect of invoking Mallory 's fallback, which in turn invokes again ping.

Since variable sent has not already been set to true, Bob sends again 2 values to Mallory and invokes one more time her fallback starting a loop.

```
1  contract Bob {
2      function ping();
3  }
4
5  contract Mallory {
6      function() {
7          Bob(msg.sender).ping(this);
8      }
9  }
```

Figure 2.2: Example of reentrancy vulnerability. Mallory contract.

This loop end when the stack limit is reached or execution goes out-of-gas or when Bob has spent off all his ether. In all cases an exception is thrown: however, since call does not propagate the exception, only the effects of the last call are reverted, leaving all the previous transfers of ether valid.

This vulnerability resides in the fact that function ping may misbehave if invoked before its termination.

2.4 Software Verification

Software Verification has the main goal of finding the answer of the following question: **"Did we build the software correctly?"**. This question checks if the functionalities and requisites desired were achieved, and so if the software was correctly built.

There are two well established approaches to software verification: **Model Checking** and **Theorem Proving**.

2.4.1 Model Checking

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model [17]. Basically, the check is performed as an exhaustive state space search that is guaranteed to terminate since the model is finite. There are two general approaches to model checking:

- The first approach is called **Temporal Modal Checking:** In this approach specifications are expressed in a temporal logic and systems are modeled as finite state transition systems. In this one an efficient search procedure is used to check if a given finite state transition system is a model for the specification.

- In this approach the specification is given as an automaton and then the system (modeled as an automaton as well) is compared to the specification to determine if its behavior conforms to that of the specification.

Model Checking is completely automatic and it can be used to check partial specifications, and so, even if the system has not been completely specified, it can provide useful information about a system's correctness.

The main disadvantage of model checking is the state explosion problem (as the number of state variables in the system increases, the size of the system state space grows exponentially). Another disadvantage of model checking is that this one doesn't analyse the program source code.

2.4.2 Theorem Proving

In this technique, both the system and its desired properties are expressed as formulas in some mathematical logic [17]. This logic is defined by a formal system that is composed by a set of axioms and a set of inference rules.

Basically, theorem proving is the process of finding a proof of a property through the axioms of the system.

In contrast to model checking, theorem proving can deal directly with infinite state spaces.

There's a lot of tools and techniques that uses Theorem Proving. One example of it, is the Verity Verification tool.

This one can handle entire processor designs containing millions of transistors. Using this tool, the functional behavior of a hardware system at the register- transfer level, gate level, or transistor level is modeled as a Boolean state transition function. Algorithms based on BDDs are used to check the equivalence of the state transition functions for different design levels.

Theorem Proving have some disadvantages like developing the hints / proof by hand can be very demanding and It can also be very difficult to formalize correctness.

However, Theorem Proving can reduce the risk of mistakes, being a valuable approach to Software Verification

2.4.3 Static Analysis

Static program analysis is the analysis of software performed without executing any programs [15]. The term is usually applied to analysis performed by an automated tool, with human analysis typically being called "program understanding".

The uses of the information obtained from the analysis vary from highlighting possible coding errors to formal methods that mathematically prove properties about a given program (e.g., its behaviour matches that of its specification). The OMG (Object Management Group) describes three levels of software analysis:

- Unit Level: Analysis that takes place within a specific program or subroutine, without connecting to the context of that program.
- Technology Level: Analysis that takes into account interactions between unit programs to get a more holistic and semantic view of the overall program in order to find issues and avoid obvious false positives. For instance, it is possible to statically analyze the Android technology stack to find permission errors.
- System Level: Analysis that takes into account the interactions between unit programs, but without being limited to one specific technology or programming language. A further level of software analysis can be defined.
- Mission/Business Level: Analysis that takes into account the business/mission layer terms, rules and processes that are implemented within the software system for its operation as part of enterprise or program/mission layer activities. These elements are implemented without being limited to one specific technology or programming language and in many cases are distributed across multiple languages, but are statically extracted and analyzed for system understanding for mission assurance.

Static analysis can be defined as the set of algorithms and techniques applied on a representation of the program code, that we want to analyze, without being executed. Static analysis is an automatic process and allows the identification of errors at an early stage of development, which considerably reduces costs, since defect discovery is exponentially more expensive over time.

However, static analysis is not perfect, since some of the properties it tries to verify are undecidable. If we ignore the limitations of finite memory, we can consider that the programming languages used in software development are complete Turing languages.Complete Turing language is theoretically capable of expressing all tasks that computers can perform, that is, it has the same processing power as a Turing machine.

2.5 Type Systems

The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program .

The absence of execution errors is a nontrivial property. When such a property holds for all of the program runs that can be expressed within a programming language, we say that the language is type sound.

The notation commonly used for describing type systems consists of judgments, which are formal assertions about the typing of programs, type rules, which are implications between judgments, and derivations, which are deductions based on type rules.

2.5.1 Execution errors

An execution error occurs when a program is asked to do something that it cannot, resulting in a crash [14]. The most obvious symptom of an execution error is the occurrence of an unexpected software fault, such as an illegal instruction fault or an illegal memory reference fault. There are, however, more subtle kinds of execution errors that result in data corruption without any immediate symptoms. Moreover, there are software faults, such as divide by zero and dereferencing nil, that are not normally prevented by type systems.

2.5.2 Typed and untyped languages

A program variable can assume a range of values during the execution of a program [14]. An upper bound of such a range is called a type of the variable.

Languages where variables can be given (nontrivial) types are called typed languages.

Languages that do not restrict the range of variables are called untyped languages. This ones do not have types or, equivalently, have a single universal type that contains all values.

A type system of a typed language keeps track of the types of variables and, in general, of the types of all expressions in a program. Type systems are used to determine whether programs are well behaved. Only program sources that comply with a type system should be considered real programs of a typed language; the other sources should be discarded before they are run.

2.5.3 Execution errors and safety

We can distinguish two kinds of execution errors: the ones that cause the computation to stop immediately, and the ones that go unnoticed (for a while) and later cause arbitrary behavior [14].

The former are called trapped errors, whereas the latter are untrapped errors. A program fragment is safe if it does not cause untrapped errors to occur. Languages where all program fragments are safe are called safe languages.

Therefore, safe languages rule out the most insidious form of execution errors: the ones that may go unnoticed. Untyped languages may enforce safety by performing run time checks. Typed languages may enforce safety by statically rejecting all programs that are potentially unsafe. Typed languages may also use a mixture of run time and static checks.

2.5.4 Execution errors and well-behaved programs

A subset of the possible execution errors is called by forbidden errors. The forbidden errors include all untrapped errors plus a subset of the trapped errors [14]. A program is well behaved if it does not cause any forbidden error to occur, otherwise the program is ill behaved.

In particular, a well behaved fragment is safe. A language where all of the (legal) programs have good behavior is called strongly checked.

A strongly checked language guarantees to a type system the following properties:

- No untrapped errors occur (safety guarantee).
- None of the trapped errors designated as forbidden errors occur.
- Other trapped errors may occur. It is the programmer's responsibility to avoid them.

Statically checked languages can enforce good behavior through performing static checks (so unsafe and ill programs don't run). This checking process is called typechecking and the algorithm that performs this checking is called the typechecker.

A program that passes the typechecker is said to be well typed, otherwise, it is ill typed.

3

Related Work

3.1 Smart Contract Languages

In this section, we introduce some relevant tools and smart contract languages that have been proposed to help in the development of Smart Contracts and to ensure their correct and safe behavior.

According to the main Blockchain organisations the languages used to develop Smart Contracts are the mainstream ones [1].

Digital contracts need to secure valuable assets and these languages provide no means to do that, as many attacks have been exploited leading to important losses.

To overcome this problem, these organisations have developed Domain Specific Languages (DSLs) to support programming digital contracts.

3.1.1 Solidity

Solidity is the most advanced and the first object-oriented, high-level programming language for Ethereum.

Solidity is an imperative language that leverages JavaScript [2]. The Solidity code is executed in the Ethereum Virtual Machine, and developers need to set an amount of gas for the contract, which needs to be enough for the contract to execute completely, as each line needs some amount of gas to be executed.

In short, Solidity is a statically typed object-oriented language inspired by Python and JavaScript, that does not provide any guarantees of correct behaviour, suffering from several vulnerabilities.

3.1.2 Vyper

Vyper is a pythonic contract oriented programming language used to write Smart Contracts for Ethereum [3]. This language address resource usage and deliberately has less features (like recursive calls and class inheritance) than Solidity with the aim of making contracts more secure and easier to audit.

Despite this measures to improve safety, Vyper still suffers from re-entrance vulnerabilities.

3.1.3 Bamboo

Bamboo is a smart contract language targeting the Ethereum Virtual Machine with a sintax similar to Solidity and supporting all of the same default types [4]. This language aims to model state transitions explicitily. In Bamboo when a function returns it must explicitly declare the state of the current contract. States are modeled as multiple contracts with a discrete set of functions that are only available while a contract is in a given state.

In short, Bamboo avoids re-entrance by design, but has no mechanism to track assets stored in a contract.

3.1.4 Move

Move is a programming language based on Rust that was created by Facebook for developing customizable transaction logic and Smart Contracts for the Libra digital currency.

Move is an executable bytecode language used to implement custom transactions and Smart Contracts [5]. The key feature of Move is the ability to define custom resource types with semantics inspired by linear logic: a resource can never be copied or implicitly discarded, only moved between program storage locations. This language allows the definition of types to represent and control assets, but has no native support for contract protocols or gas usage.

3.1.5 Michelson

Michelson is a domain-specific language for Smart Contracts on Tezos [6]. This language is a low-level, stack-based and heavily typed programming language used to write Smart Contracts on the Tezos blockchain, designed to facilitate formal verification, allowing users to prove the properties of their contracts. Michelson is too low level and purely functional with no mechanisms to express linearity or protocols.

3.1.6 LIGO

LIGO is a statically typed, high-level language for Smart Contracts. The LIGO language is extensible and easy to use while promoting security. It also supports several syntaxes. While most examples are written in Pascal, it also supports OCaml.

LIGO also is developing a tool that will link to Cubicle, in order to model check the protocols.

LIGO is a simple smart contract language designed for developing longer contracts than one would naturally write in Michelson. It is an imperative language that despite compiling down to Michelson does not add correctness assurances.

3.1.7 Academic Languages

Very few of the Domain Specif Languages presented above provide soundness guarantees (none covers all the crucial properties, and most are quite low-level, thus difficult to use) and even fewer have a formal verified model of their semantics. To address these shortcomings, several academic languages were proposed.

BitML [12] supports the specification of contracts via assertions describing requirements and processes declaring the execution logic. BitML specifications are compiled to Bitcoin, but as the latter is less expressive than the former, part of the intended behaviour is "lost in the translation".

DeepSea is a functional programming language that puts security first, using Formal Verification to secure code [7]. This language targets Ethereum integrated with the Coq Proof Assistant.The compiler generates efficient executable code and models to be verified and so, in this one the developer is assumed to be both a programmer and a skilled user of theorem provers.

There are also experiments developing small, rigorously defined type-safe languages that ensures some valuable properties but in general, none of these languages support assertions, not allowing to fully prove contracts correct.

LiquidJava is a language based in Java, integrated with the usage of liquid types. This one is achieved by creating an additional type system with refinements on top of the existing Java type system.

LiquidJava does the Software Verification through SMT Solver tool.

Flint Flint is type-safe, contract-oriented programming language specifically designed for writing robust Smart Contracts on Ethereum [8]. It introduces protection blocks to prevent incorrect behaviour from the contract. Using typestate, these blocks restrict when the code can be executed, as states are checked statically for internal calls, and at runtime for external calls to the contract. In short, Flint provides typestates to define protocols but still, this language lacks several important aspects like assets protection or gas consumption control.

Featherweight Solidity is a calculus formalizing the core features of the Solidity language, thus providing a fundamental step to reason about safety properties of Smart Contracts' source code [22]. Like flint, this language still lacks a lot of important aspects to guarantee Smart Contracts safety and correctness guarantees.

FLINT-2 Flint-2 is a new Rust-based programming language which aims to help programmers lessen the chances of accidental code errors. It provides a development environment that ensures the correct behaviour of contracts before their deployment with

specific safety features, such as typestates and caller protections. This language is the newest iteration of Flint that captures the "safety-oriented" characteristics once introduced in Flint, being the protection blocks the main focus. Flint-2 also takes advantage of typestates and caller groups by restricting the function callings. These features help Flint-2 to ensure Smart Contracts safety.

3.2 Verification Tools

3.2.1 SMTChecker

The SMTChecker automatically tries to prove that a program satisfies the specification given by require and assert statements [9]. That is, it considers require statements as assumptions and tries to prove that the conditions inside assert statements are always true. If an assertion failure is found, a counterexample may be given to the user showing how the assertion can be violated. SMTChecker essentially converts a program into statements (or more specifically, into problems) and it then tries to prove whether the code can be broken or not by returning a value of true or false (also acknowledged as either being satisfiable/unsatisfiable in terms of SAT theory).

The idea behind the module is to basically catch bugs and prove the correctness of a program at compile-time, modules like this are often present as compiler optimization features because it is a mathematically proven way ensure behavioral consistency with respect to i.e. the translation process from high level to low level code.

Some of the advantages of this tool are:

- Running the SMTChecker can be considered easy.
- Since Solidity is the predominant language used for writing Smart Contracts, having a model checker built directly into Solidity will make formal methods more accessible to developers.
- Specifications are written inline using the Solidity language itself which means that while there is some extra work involved in writing a specification, it does not come with the usual overhead of having to install a new tool and learn a new language.
- Through assertions and require instructions, this tool can check at compile time some properties like:
    1. Arithmetic underflow and overflow.
    2. Division by zero.
    3. Trivial conditions and unreachable code.
    4. Popping an empty array.
    5. Out of bounds index access.

6. Insufficient funds for a transfer.

- Despite this tool still being a work in progress, a quick look at the Solidity GitHub repository shows promising and active development.
- When finding bugs, this tool doesn't produce false negatives, that is, if it determines that a verification target is safe, it is indeed safe,

On the other hand, this tool has still some drawbacks:

- Using this tool in a way that gives meaningful results can still be tricky. It is important to be aware of the model checker options, to tell the tool which specific contracts need to be verified, and to use the best engine. It is essential to try as many engines, solvers, and configurations as possible when dealing with complex problems and properties. It is often the case that one different parameter can cause the solver to go the right way and prove/disprove a property that other solvers and configurations may timeout or run out of memory.
- The SMTChecker implements abstractions in an incomplete and sound way: If a bug is reported, it might be a false positive introduced by abstractions (due to erasing knowledge or using a non-precise type).
- Therefore, several properties might not be solved or might lead to false positives for large contracts.

In conclusion, Solidity's model checker is currently known to be sound but incomplete. This means that the tool shouldn't miss any bugs without warning you that it cannot reason about something, though it may also produce false positives which will need to be sorted through.

As it was mentioned above, this is a promising tool that is still a work in progress. There are still some important properties that this tool still can't verify like unsafe reentrancy, unsafe timestamp dependence and denial of service.

3.2.2 ESBMC-Solidity

3.2.2.1 ESBMC

ESBMC (the Efficient SMT-based Bounded Model Checker) is a mature, permissively licensed open-source context-bounded model checker for verifying single and multi-threaded C/C++ programs [35].

This model checker can verify safety properties and user defined assertions automatically. ESBMC supports the Clang compiler as its C/C++ frontend, IEEE floating-point arithmetic for various SMT solvers, implements the Solidity grammar production rules as its Solidity frontend and state-of-the-art incremental BMC and k-induction algorithms.

ESBMC was initially devised as a C-language model checker but can also be extended to support different programming languages and target systems.

3.2.2.2 ESBMC-Solidity Overview

ESBMC-Solidity consists in a frontend that uses the model checker ESBMC to verify Solidity Smart Contracts via two steps. First, the Solidity Smart Contract is converted from JavaScript object notation (JSON) abstract syntax trees (AST) to the ESBMC's intermediate representation (IR). Second, the last one is integrated with ESBMC's infrastructure to reuse its existing SMT-based verification strategies (incremental and k-induction).

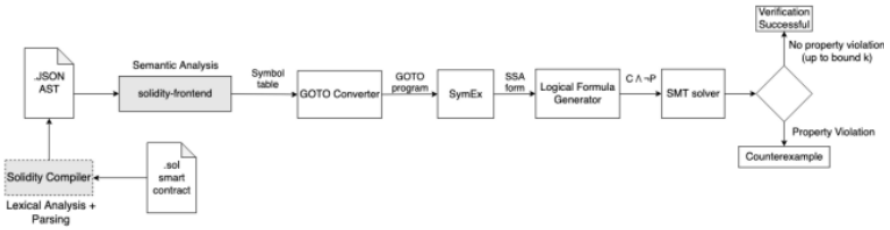

Figure 3.1: Architectural overview of ESBMC with its extension for verifying Solidity Smart Contracts. [34]

Fig 3.1 illustrates the architecture of ESBMC-Solidity. The gray box with solid border represents the new frontend and white boxes are the ESBMC's components. The gray box with a dashed border indicates an external element, the Solidity compiler, that is used for preprocessing Smart Contracts. It is used basically for lexical analysis and parsing, taking a smart contract as input and then transforming it into JSON AST.

This approach converts each node from the JSON AST nodes into an equivalent IR one, using the ESBMC's irept, a tree-structured IR that preserves a program's semantics. Then, each of the irept nodes is converted to a corresponding symbol and then added to a table, which is translated into a GOTO program. After this, the program is processed by the symbolic execution engine (SymEx) to generate its static single assignment (SSA) form, which is used to generate verification condition (VCs) C $\wedge\neg P$, where C represents constraints and P denotes a safety property. Lastly, ESBMC uses SMT solvers for verifying the satisfiability of those VCs.

When a property is satisfiable, an execution path leads to a bug in the Solidity Smart Contract. Then, when ESBMC detects it, a counterexample is provided, in the form of state traces.

Some of the advantages of this tool are:

- ESBMC-Solidity checks memory safety user-defined properties in Smart Contracts written in the Solidity programming language.
- It can detect some vulnerabilites like:
  1. Integer Overflow.
  2. Integer Underflow.
  3. Authorization through tx.origin.

4. Static array out-of-bounds.
5. Dynamic array out-of-bounds.

On the other hand, some of the drawbacks of this tool are:

- A bounded model checker can only analyze a finite number of program steps. With loops the model checker must also account for any number of loop iterations and the extra paths associated with them, which may become huge (state blowup). To mitigate this problem, Solidity's model checker currently only considers one iteration of a loop. This approach alone would lead to unsoundness, and consequently, false positives.
- Because of some atomicity assumptions, this tool can also produce false negatives [34].

3.2.3 RA: A Static Analysis Tool for Analyzing Re-Entrancy Attacks in Ethereum Smart Contracts

RA (Re-entrancy Analyser) is a combination of symbolic execution and equivalence checking by a satisfiability modulo theories solver to analyze vulnerability of Smart Contracts to re-entrancy attacks [16].

One of the most important features of this tool is that it supports the analysis of inter-contract behaviors by using only the Ethereum Virtual Machine bytecodes of target Smart Contracts, i.e., even without prior knowledge of attack patterns and without spending Ether. In addition, RA can detect vulnerabilities in Smart Contracts without having to execute them and it does not provide false positives and false negatives. As we mentioned above, this tool focuses on the analysis of EVM bytecodes of Smart Contracts instead of their corresponding source codes. The main reasons for that are:

1. The analysis of EVM bytecodes is independent of a high-level language that is periodically updated.
2. The analysis of a high-level language, such as Solidity, requires an analyst to input a copy of the actual source codes of the analysis targets.

3.2.3.1 RA Overview

The figure 3.2 illustrates RA's architecture.

RA consists of three modules, CFManager, VM and Verifier. CFManager and VM make the **symbolic re-entrancy emulation process** (emulation of re-entrancy attacks by connecting different contracts with each other) possible. The Verifier (assisted by VM) executes the vulnerability **verification process** (verification of vulnerabilities using Z3 SMT solver in the CFGs obtained from the symbolic re-entrancy emulation).

Some of the advantages of this tool are:

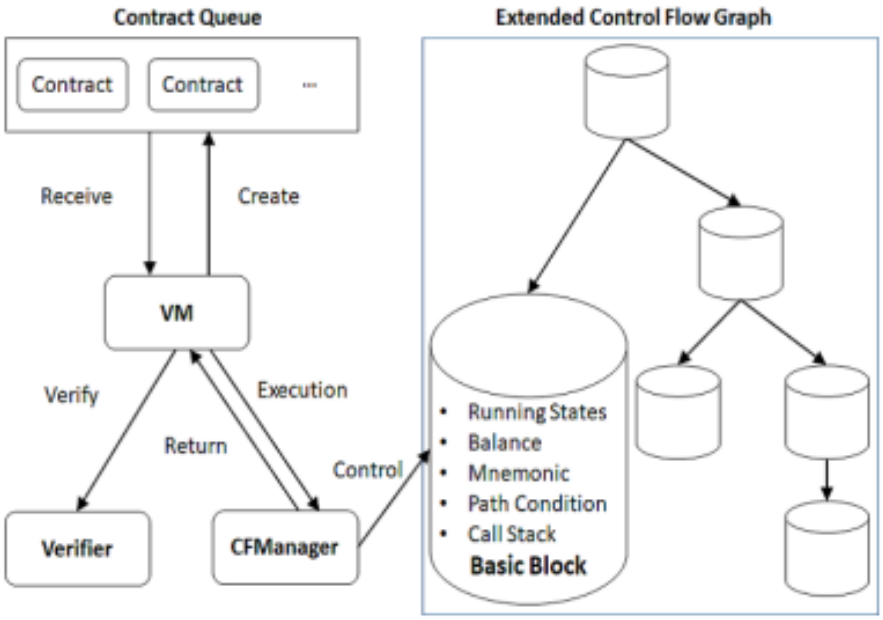

Figure 3.2: Overview of RA. [16]

- Through symbolic execution and equivalence checking by a satisfiability modulo theories solver and through the analysis of inter-contract behaviors by using only the Ethereum Virtual Machine bytecodes of target Smart Contracts, RA can effectively verify the existence of vulnerabilities to re-entrancy attacks without execution of Smart Contracts.
- It doesn´t produce false positives or false negatives.

However, this tool has still some limitations:

- The analysis of gas is not considered. Thus, contracts that restrict gas consumption may not be analyzed precisely.
- The case where multiple contracts are tightly coupled with each other, i.e., more than two contracts are strongly interdependent, is out of the scope of RA. Specifically, instructions that utilize other contracts as an external library are not implemented.
- Bytecodes of contracts cannot include symbolic values. Consequently, a case where created contracts are different for each execution is not considered.

3.2.4 MVP: Move Prover

The Move Prover [24] is an automated tool that allows developers to formally verify Smart Contracts written in the Move programming language.

This tool operates on the Move bytecode itself, avoiding potential compiler bugs from interfering with prover correctness. MVP has an expressive specification language, and

is fast and reliable enough that it can be run routinely by developers and in integration testing.

There are three main points that resulted in dramatic improvements in speed and reliability in MVP:

- An alias free memory model based on Move's semantics.
- Fine-grained invariant checking, which ensures that invariants hold at every state, except when developer explicitly suspends them.
- Monomorphization, which instantiates type parameters in Move's generic structures, functions, and specification properties.

3.2.4.1 Move Prover architecture

The figure 3.3 illustrates the MVP architecture.

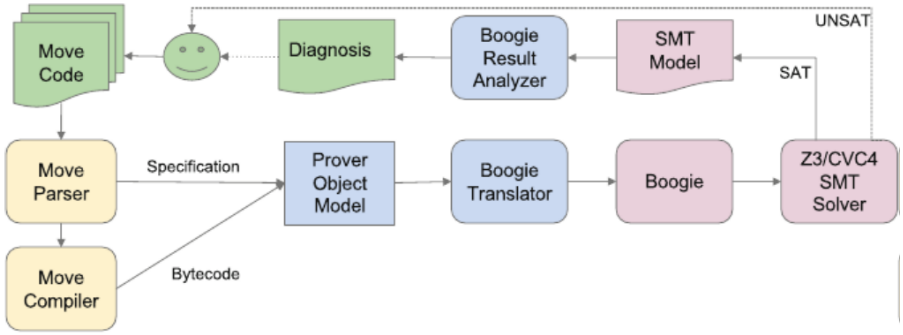

Figure 3.3: The Move Prover architecture. [38]

The Move Prover starts by receiving as an input a Move source file that contains the code annotated with specifications of the Smart Contract. Then, the Move Parser parse and adds these specifications to a Move prover object model. Then, the model is translated into an intermediate verification language called Boogie[10]. Then, the Boogie Verification system generates an SMT formula that if satisfiable it means that the specifications doesn't hold. To verify if the generated formula is unsat or satisfiable the input is passed to the automated theorem prover Z3 [36].

The Z3 prover then checks if the SMT formula is unsatisfiable. If yes, it means that the Smart Contract specifications hold. Contrarily, the tool generates a report with the model that satisfies the conditions. The diagnosis report is then reverted to a source-level error which parallels a standard compiler error.

Some of the advantages of this tool are:

- Move is minimal in comparison to most conventional programming languages. The only types besides records are primitives (Booleans, unsigned integers, addresses), vectors, and references [24].

- Its Verification-Friendly Design [39]. There are many factors that make this possible. First, to ensure deterministic execution, move has limited interaction with the environment i.e data can only read from the global blockchain state or the current transaction (no file or network I/O). Second, many features that are challenging for verification are excluded from Move: concurrency, higher-order functions, exceptions, sub-typing, and dynamic dispatch. The absence of the last feature is particularly notable because it avoids the famous re-entracy attacks.

- Move has built-in safe arithmetic: overflows and underflows are detected during execution and result in a transaction abort [11].

- Many common errors are prevented by the Move bytecode verifier, a static analyzer that checks each bytecode program before execution.

On the other hand:

- To prevent some general bugs move exclude features (like concurrency, sup-typing etc) [11].

- Move is a prototype, still being improved and yet its age means that some flaws are still to be discovered and proven.

3.2.5 Verification Tools Comparison

Software Verification Tools				
Verification Tool	Activity	False Negatives	False Positives	Arithmetic Bugs
SMTChecker	✓	✓	×	✓
ESBMC-Solidity	✓	×	×	✓
RA	×	✓	✓	×
MVP	✓	✓	?	✓
Cubicle	×	✓	?	✓

Table 3.1: Comparison of the approached software verification tools.

The table 3.1 shows a comparison of the multiple software verification tools. The first column "Activity" shows if this tool is not only being used but also updated and improved. The second column shows if the tool can produce false negatives (if it says that a contract doesn't have a specific bug when it actually has). The third column shows if the tool can produce false positives (if it states that a contract has a bug when it actually hasn't). Finally the last column states if this tool can check some arithmetic properties like overflows, underflows, division by zero etc. The question mark "?" states that no information was found about it.

In conclusion, all of these tools are useful in software verification and each one can solve specific different problems.

An important part of this dissertation focused in using a tool to automatically verify Smart Contracts. The main reason why we chose Cubicle was that the model language combines the benefit of a compact formal model with the power of a pushdown automaton that allows to describe open, array-based systems indexed by first-class process entities. These array-based systems are compatible with the specifications (protocols) defined with assertions in a state diagram.

Another Cubicle important feature is that this tool can verify if a Smart Contract matches its specifications with an arbitrary number of participants (where although each client interacts with its own instance of the contract, its state is shared). In short, Cubicle allows a programmer to write the specifications of a Smart Contract and then, with an arbitrary number of participants, tries to violate these specifications.

4

Solidity and Featherweight-Solidity

4.1 Solidity and Featherweight-Solidity Overview

4.1.1 Solidity

Solidity is an object-oriented programming language designed by Ethereum for expressing Smart Contracts on Blockchain platforms. This programming language is used to create Smart Contracts that implement business logic and generate a chain of transaction records in the blockchain system [33].

The Solidity code is executed in the EVM, and developers need to set an amount of gas for the contract, which needs to be enough for the contract to execute completely, as each line needs some amount of gas to be executed [30].

In conclusion, Solidity is an imperative language that has easy to learn sintax similiar to JavaScript, is statically typed and also supports complex types.

4.1.1.1 Smart Contract Bank written in Solidity

The example 4.1 shows a simple contract written in Solidity [32]. This Smart Contract represents a Bank that through a state variable *balances*, stores the accounts of every user.

A user can deposit money into his account via the *deposit* function. To do it, he must call this function with an amount of Wei, which will be the money to increment to his account. A user can also check his balance by calling the getter *getBalance* function. Through the function *withdraw* a user can withdraw money from his account and with the transfer function, he can also transfer money from his account to another.

This Smart Contract doesn't have any fallback function since the only way to send money to this Bank is through the deposit function. As a final note, when a user try to do something that he is not supposed to (per example, withdraw more money that he actually has in his account), there are no errors or exceptions thrown, the function simply does nothing.

Listing 4.1: Bank.sol

```
contract Bank {
mapping (address => uint) private balances;

constructor() public {}

function deposit() external payable {
    balances[msg.sender] += msg.value;
}

function transfer(address to, uint amount) external {
    if (balances[msg.sender] >= amount) {
    balances[msg.sender] -= amount;
    balances[to] += amount;
    }
}

function getBalance() external view returns (uint) {
    return balances[msg.sender];
}

function withdraw(uint amount) external {
    if (balances[msg.sender] >= amount) {
        balances[msg.sender] -= amount;
        msg.sender.transfer(amount);
    }
}
}
```

4.1.2 Featherweight-Solidity (FS)

Featherweight-Solidity (FS) [32] is a calculus that models the core features of Solidity. Its goal is to allow the study of Smart Contracts, such as contract deployment, interaction among contracts and money transfers, as it includes formalised operational semantics and a type-system. This type-system prevents incorrect behavior where a lot of errors, such as accesses to a non existing function or state variable, are only detected at runtime and cause interruption and rolling-back of transactions [23].

4.1.2.1 Smart Contract Bank written in Featherweight-Solidity

The Listing 4.2 shows the Featherweight-Solidity implementation of the Bank example. As we can observe, the are some differences between Solidity and Featherweight-Solidity. In Featherweight-Solidity we have to explicitly initialize every state variable. Furthermore, the syntax for defining the functions is a slightly different, more similar to java but here we use the type unit to represent void. In Featherweight-Solidity, function

bodies consist of a single expression, return e, where e represents the expression of the function body.

Listing 4.2: Bank.fs

```
contract Bank{

mapping(address -> unit) balances;

Bank(mapping(address -> unit) balances) {
    this.balances = balances;
}

unit deposit() {
    return
        this.balances = this.balances[msg.sender -> this.balances[msg.sender] + msg.value]; u
}
uint getBalance() {
    return this.balances[msg.sender]
}
unit transfer(address to, uint amount) {
    return
        if this.balances[msg.sender] >= amount then
            this.balances = this.balances[msg.sender -> this.balances[msg.sender] - amount];
            this.balances = this.balances[to -> this.balances[to] + amount];
            u
        else
            u
}
unit withdraw(uint amount) {
    return
        if this.balances[msg.sender] >= amount then
            this.balances = this.balances[msg.sender -> this.balances[msg.sender] - amount];
            msg.sender.transfer(amount);
            u
        else
            u
}

}
```

4.1.3 Featherweight Solidity Implementation

To write and typecheck Smart Contracts we opted by using this Featherweight-Solidity implementation [31] as a starting point.

In this section, we present this implementation and explain some examples of use. Then we will explain its limitations and modifications made to fully be able of writing well typed contracts.

4.1.3.1 Featherweight Solidity Grammar

In the Figure 4.1 we describe the grammar of the Featherweight Solidity implementation.

⟨*(Contract decl)*⟩	SC ::= contract C ($\widetilde{Ts}$) K $\widetilde{F}$
⟨*(Constructor decl)*⟩	K ::= C (($\widetilde{Tx}$))($\widetilde{this.s}$ = x))
⟨*(Function decl)*⟩	F ::= T f (($\widetilde{Tx}$)) { return e } | unit f b() { return e }
⟨*(Call Stack)*⟩	σ ::= β | $\sigma \cdot$ a
⟨*(Blockchain)*⟩	β ::= $\emptyset$ | $\beta \cdot$ [x -> v] | $\beta \cdot$ [(c,a) -> ((C, ($\widetilde{s:v}$) ,n),(x -> v))]
⟨*(Contract Table)*⟩	CT ::= $\emptyset$ | CT $\cdot$ [C -> SC]
⟨*(Expression)*⟩	e ::= v | x | b | this | this.f | msg.sender | msg.value | address(e) | e.s | e.transfer(e) | new C.value(e)(($\widetilde{x:e}$))(c a) | e e | T x = e | x = e | e.s = e | e[e] | e[e $\rightarrow$ e] | e.value(e)($\widetilde{x:e}$) | e.f .value(e)($\widetilde{x:e}$) | revert | e.f .value(e).sender(e)($\widetilde{x:e}$) | if e then e else e
⟨*(Values)*⟩	v ::= true | false | n | a | u | M | c vv ::= v|c.f
⟨*(Types)*⟩	T ::= $\widetilde{T} \rightarrow$ T | bool | uint | address | unit | mapping(T $\rightarrow$ T) | C

Figure 4.1: Featherweight Solidity Grammar [31]

In this grammar, a contract C is composed by a set typed state variables $\widetilde{Ts}$, a constructor K, and some function definitions $\widetilde{F}$. Each function has return type $\widetilde{T}$, a name f, some typed parameters $\widetilde{Tx}$ and and expression e (function's body). Basically, T f (($\widetilde{Tx}$)) return e defines a function named f that takes a sequence, possibly empty, of typed parameters (T1 x1, . . . , Tn xn) and whose body returns a value of type T.

The β represents the blockchain, and follows the formalisation of the Ethereum one. β maps pairs (c, a) to triples (C, $\widetilde{s:v}$, n), and variable identifiers x to values v. The unique pair of unique identifiers (c, a) represents the contracts reference and its address respectively, and it is associated to the name of the contract C, the contract's state variables ($\widetilde{s:v}$), and its balance n, and also to its instantiated variables.

The Call Stack σ stores the addresses of the contracts that performed function calls within the execution of a transaction [32].

Finally, the Contract Table maps the contract names to their declaration.

4.1.3.2 Execution of Bank example

As it was mentioned above, to summarize, the Bank contract contains a state variable *balances* that represents the amount in each Bank account. Function *deposit* increases the balance of the msg.sender account by the amount read form msg.value. Function *getBalance* returns the balance of the caller. In addition, functions *withdraw* and *transfer* allow the clients to transfer the money to their own account or to another client's account, respectively.

Listing 4.3: Bank's program typecheck

```
1  let ctBank =
2    (CT.add "Bank" ([((TMapping(TAddress, TNat)), "balances")],
3                  [(TUnit, "constuctor",[((TMapping(TAddress, TNat)),"balances")]);
4                  (TUnit, "deposit",[]);
5                  (TNat, "getBalance", []);
6                  (TUnit, "transfer", [(TAddress,"to");(TNat,"amount")]);
7                  (TUnit, "withdraw", [(TNat,"amount")])
8                  ]) CT.empty);;
9
10 let gammaBank = (Gamma.add "aC1" TAddress
11           (Gamma.add "Bank" (TContract("Bank"))
12           (Gamma.add "aBank" TAddress Gamma.empty)));;
13
14 (test_func gammaBank ctBank None (mk_eval
15 (TmDecl ((TContract("Bank")), "Bank",
16 (TmNew("Bank", (TmValue(VCons(500))), [(TmValue(VMapping([( (VAddress("aC1")) , VCons(550))]))
       ↪ )] )),
17 (TmSeq(
18 (TmCallTop((TmValue(VContract("Bank")), "constuctor", (TmValue(VCons(500))), (TmAddress(
       ↪ TmValue(VContract("Bank")))) ,[(TmValue(VMapping([( (VAddress("aC1")) , VCons(500))])))
       ↪ ] ))),
19 (TmSeq((TmCallTop((TmValue(VContract("Bank")), "deposit", (TmValue(VCons(100))), (TmAddress(
       ↪ TmValue(VContract("Bank")))) ,[] ))),
20 (TmSeq((TmCallTop((TmValue(VContract("Bank")), "getBalance", (TmValue(VCons(0))), (TmAddress(
       ↪ TmValue(VContract("Bank")))) ,[] ))),
21 (TmCallTop((TmValue(VContract("Bank")), "transfer", (TmValue(VCons(100))), (TmAddress(TmValue(
       ↪ VContract("Bank")))) ,[TmValue(VAddress("aC1")); TmValue(VCons(100))] )))
22 ))))))))))
```

In the Listing 4.3 we present an example of an execution program of the Smart Contract Bank, and how do we use the type system to check that it is well-typed. Firstly, we define the Contract Table with the contract Bank and its contract declaration i.e state variables, then the constructor and functions declaration. Finally, we define the gamma that stores the contract, its address and an address of a user.

In the line 14 we call a function *testfun* that checks if the program is well typed. This function receive four arguments. The first argument is the contract's gamma, the second

is the contract's contract table and the third one is the expected type of the expression (in this case). The fourth argument is an evaluation of the expression that returns it's type. In short, *testfunc* will see if the expected type is the expression's type. Since in this case, the expression is an execution of the program, the *eval* function will only typecheck the expression and return the *None* type.

In line 15 we declare the Bank contract and initialize its state variables. After that, 100 of balance is deposited in the account of a user (with the address "Ac1"). Then, he check his balance and finally, the Bank transfers 100 to his account though the transfer function. In 4.4 we show the partial output of the typechecking. In A.1 (*Appendix: Bank's program Typecheck Output*) we can check the whole output with the derivation trees.

Listing 4.4: Bank's program typecheck output

```
1  contract Bank {(mapping(address => uint) : balances)} unit constuctor ((mapping(address =>
       ↪ uint) : balances)) ; unit deposit () ; uint getBalance () ; unit transfer ((address :
       ↪ to)(uint : amount)) ; unit withdraw ((uint : amount))
2  , (Bank : Bank); (aBank : address); (aC1 : address); (aC2 : address); |- Bank Bank = new Bank.
       ↪ value(500)([(aC1 , 550)]) ;
3
4  Bank.constuctor.value(500).sender(address(Bank))([(aC1 , 500)]) ; Bank.deposit.value(100).
       ↪ sender(address(Bank))() ; Bank.getBalance.value(0).sender(address(Bank))() ; Bank.
       ↪ transfer.value(100).sender(address(Bank))(aC1, 100) : (DECL)
5
6  Bank(Bank : Bank); |- Bank.constuctor.value(500).sender(address(Bank))([(aC1 , 500)]) : (
       ↪ CALLTOPLEVEL)
7            Bank(Bank : Bank); |- address(Bank) : address
8
9  Bank(Bank : Bank); |- Bank.deposit.value(100).sender(address(Bank))() : (CALLTOPLEVEL)
10             Bank(Bank : Bank); |- address(Bank) : address
11
12
13 Bank(Bank : Bank); |- Bank.getBalance.value(0).sender(address(Bank))() : (CALLTOPLEVEL)
14               Bank(Bank : Bank); |- address(Bank) : address
15
16
17 Bank(Bank : Bank); |- Bank.transfer.value(100).sender(address(Bank))(aC1, 100) : (CALLTOPLEVEL
       ↪ )
18               Bank(Bank : Bank); |- address(Bank) : address
19
20 Success
```

4.2 Mini-FS

In the previous section, we showed how we can use the implementation of Featherweight-Solidity [31] to make sure that programs are well typed. Still, we want to typecheck a whole contract and not only its execution, i.e we also want to typecheck it's implementation code. To do so, we had to make some changes to it's grammar. In the next sections,

we will describe this changes and present some examples of use.

4.2.1 Mini-FS Grammar

Below, in 4.5 the MiniFS strtucture of a contract is presented. In B.1 (*Appendix: MiniFS's Grammar*) we can see the full grammar implemented in Ocaml. As we can observe, the grammar is similiar to the Featherweight Solidity one (described in 4.1), yet, some modifications were made:

- The new grammar now supports arithmetic operations, i.e. (addition, subtraction, multiplication and division).
- Boolean operations were added (and, or, not, greater, less, equals)
- The types *TString* and *TState* were added to *typ*. The type *TString* was added so the grammar could be more complete. *TState* was added because we need a type to represent the behavioral types (that we will explain later).
- A few extra fields were added to functions (requires and body). The 4.5 main goal was a well-typed execution of a program so the bodies of the functions were not considered. As our objectives cover the whole type checking of a Smart Contract implementation (and not just execution programs), function's bodies must also be typechecked.

  One of the big changes in Mini-FS was the addition of behavioral types. With behavioral types, a lot of execution errors can be prevented since they define how a participant interacts with a contract (which functions can he call, in which order can he call them, etc). As we will explain in the next sections, the requires modification facilitates the implementation of behavioral types.
- A contract now has a list of behavioral types, an *init* feature (which allows the programmer to set some initial conditions on variables) and a invariant feature. This new feature allows the developer to set some conditions to the contract that should always be met.

Listing 4.5: Mini-FS Grammar in ocaml

```
1
2  type contract_tc = {
3  name : string;
4  init : (params list) * term;
5  invariant : (params list) * term;
6  behavioral_types: term list;
7  vars: (term * typ) list;
8  cons : construtor_;
9  funcs: func list;
10 }
```

4.2.1.1 Type System

As we mentioned in the previous sections, MiniFS's grammar is composed by expressions, types *typ* (the expression's types), *values* (expression's values) and the commands (e.g. assign a value to a var).

To typecheck these expressions, MiniFS's typesystem has a function *typecheck* which checks if an expression has a specific type. This function evaluates the expression recursively (by branches) and checks if the expression type is the desired one.

In 4.6 listing, there's an example of an arithmetic operation "2 + 3 * 3". In this example, we build the expression *operation* with MiniFS's grammar and then we use the function *test_func* that checks if this expression has TNat type.

Listing 4.6: MiniFS arithmetic example

```
1  let operation = TmPlus(TmValue(VCons(2)), TmMult( TmValue(VCons(3)) , TmValue(VCons(3))));
2  test_func Gamma.empty CT.empty (Some TNat) (mk_eval operation)
```

In 4.7 we can check the recursive evaluation of operation branches and then we can observe the success message, indicating that this exppression is in fact, well-typed.

Listing 4.7: MiniFS arithmetic example output

```
1  |- 2 + 3 * 3 : uint
2  (Op Arit)
3       , |- 2 : uint
4  (NAT)
5       , |- 3 * 3 : uint
6  (Op Arit)
7         , |- 3 : uint
8  (NAT)
9         , |- 3 : uint
10 (NAT)
11 Success
12 - : unit = ()
```

In MiniFS there are a lot of situations where variables or parameters are used. In order to the MiniFS type system be able to type check expressions containing these variables/-parameters, it should store each var/parameter and its associated type in a structure called gamma.

Listing 4.8: MiniFS gamma arithmetic example

```
1  let gammaenv = (Gamma.add "x" (TString) Gamma.empty)
2   let operation = TmPlus(TmVar("x"), TmMult( TmValue(VCons(3)) , TmValue(VCons(3))));
3   test_func gammaenv CT.empty (Some TNat) (mk_eval operation)
```

In 4.8 there is the declaration of a var *x* of type *TString*. Then the expression operation is built as "x + 3 * 3". As it can be noticed, this expression is wrong since var *x* is a *TString* (string) and not *TNat* (uint). The listing 4.9 shows the evaluation of this example and finally, an error message stating that the expected type was uint but got string.

Listing 4.9: MiniFS gamma plus output

```
(x : string); |- x + 3 * 3 : uint
(Op Arit)
    , (x : string); |- x : uint
(VAR)
type mismatch : termected type uint but got type string
- : unit = ()
```

In MiniFS a contract is composed by:

- A **name** (string).
- **Constuctor**. In short, a constructor is a special function that initializes the contract's variables.
- **Init**. A complementary expression to the constructor. (The use of this feature will be better explained in further sections).
- **Invariant**. This feature is basically a condition (or set of conditions) that we always want our contract to achieve.
- **Behavioral types**. To enforce protocols in these contracts, we will use behavioral types protocols.
- **Variables**. A set of state variables.
- **Functions**. A set of functions. Each function has a name, a return type, a params list (parameters), body (expression or set of expressions) and finally, a requires feature (a condition that must be met to the current function execute).

To MiniFS's typesystem be able to type check a whole function, there is a function *type-check_contract* that recursively typechecks every feature defined above.

Listing 4.10: MiniFS typecheck_contract function

```

let rec do_all ct gamma lst =
  match lst with
  | [] -> ()
  | x :: xs -> Printf.printf "----------------FUNC:␣%s␣------------------\n" x.func_name;
               test_func gamma ct (Some TBool) (mk_eval x.requires);
               test_func gamma ct (Some x.return_type) (mk_eval x.body);
               Printf.printf "----------------Done:␣%s␣------------------\n" x.func_name;
               do_all ct gamma xs;;

let typecheck_contract ct gamma contract =
  Printf.printf "--------INIT--------------------------\n";
  let (_,init) = contract.init in
  let (_,invariant) = contract.invariant in
  test_func gamma ct (Some TBool) (mk_eval init);
```

```
16  Printf.printf "--------INVARIANT---------------------------\n";
17  test_func gamma ct (Some TBool) (mk_eval invariant);
18  Printf.printf "--------CONSTRUCTOR---------------------------\n";
19  let (_,cons_body) = contract.cons in
20  test_func gamma ct (Some TUnit) (mk_eval cons_body);
21  Printf.printf "--------FUNCS---------------------------------\n";
22  do_all ct gamma (contract.funcs) ;;
```

4.3 Mini-FS examples

In this section, we present some Smart Contract implementations (mainly taken from a Microsoft Azure repository [13]) with Mini-FS and show how this one helps the programmer to develop well-typed contracts.

4.3.1 Bank

As we explained in previous sections, Bank is a simple contract that stores user's accounts. This users can deposit, transfer and withdraw money from their accounts by interacting with this Smart Contract. We present this example with the purpose of showing that not all contracts in Mini-FS have to use the behavioural types features.

Lets consider the implementation of the function *withdraw* in 4.11. In short, A function is composed by a term (requires instruction), a return type, a name, a parameters list and finally, the body of the function (another term).

In 4.11 we are declaring a function called withdraw, with no behavioral types (hence the requires is just a true boolean), with return type *TUnit* (void), with a parameter *param_withdraw_amount* and finally with a body (expression).

Listing 4.11: Withdraw function in Mini-FS

```
1  let param_withdraw_amount : params = (TNat, "W_amount")
2  let param_withdraw_amount_val : term = TmGetParam("W_amount")
3  let withdraw : func = {requires = TmValue(VTrue); return_type = TUnit; func_name = "withdraw";
       ↪ params = [param_withdraw_amount]; body =
4  TmSeq( TmIf( GreaterOrEquals(TmMappSel(balances, sender), param_withdraw_amount_val),
5             TmSeq(TmMappAss(balances, sender, TmMinus( TmMappSel(balances,
                  ↪ param_withdraw_amount_val),param_withdraw_amount_val)),
6       TmTransfer( sender,param_withdraw_amount_val ) ),
7       TmReturn(TmValue(VUnit))),
8  TmReturn(TmValue(VUnit))) }
```

In 4.11 the body is composed by multiple seq terms. In short, there's an if condition checking if the sender (user that calls the function) has enough balance. If he has enough balance we can then withdraw his money, decrementing his balance, otherwise, the function does nothing. As a final note, we could also move the if condition to the requires feature and the body would be just the instructions decrementing sender's balance (body).

Listing 4.12: Withdraw's function typecheck

```
1  Bank |- if Balances[param sender] >= param W_amount then Balances[param sender -> Balances[
       ↪ param W_amount] - param W_amount] ; param sender.transfer(param W_amount) else return
       ↪ u ; return u : unit
2  (SEQ)
3    Bank |- if Balances[param sender] >= param W_amount then Balances[param sender -> Balances[
         ↪ param W_amount] - param W_amount] ; param sender.transfer(param W_amount) else return
         ↪  u : (IF)
4      Bank |- Balances[param sender] >= param W_amount : bool
5  (GreaterOrEquals)
6       Bank |- Balances[param sender] > param W_amount : (Cond )
7         Bank |- param W_amount : uint
8  (PARAM)
9  uint
10        Bank |- Balances[param sender] : uint
11 (MAPPSEL)
12          Bank |- Balances : (Var)
13 mapping(address => uint)
14          Bank |- param sender : address
15 (PARAM)
16     Bank |- Balances[param sender -> Balances[param W_amount] - param W_amount] ; param sender.
         ↪ transfer(param W_amount) : (SEQ)
17       Bank |- Balances[param sender -> Balances[param W_amount] - param W_amount] : (MAPPASS)
18         Bank |- param sender : (Param)
19 address
20        Bank |- Balances[param W_amount] - param W_amount : (Arit)
21          Bank |- Balances[param W_amount] : uint
22 (MAPPSEL)
23            Bank |- Balances : (Var)
24 mapping(address => uint)
25            Bank |- param W_amount : address
26 (PARAM)
27 type mismatch : termected type address but got type int
```

In the Listing 4.12 there is the output of withdraw's typechecking. We can observe that the body is evaluated by branches, and finally, in line 27 we can see that there's an error "type mismatch : termected type address but got type uint". In the implementation 4.11 in line 5, there's a type error. It should be "TmMappSel(balances, → sender)", since balances is a map of address * int and not int*int.

The listing C.1 (*Appendix: Bank MiniFS and its Typecheck*) is the fully and well-typed implementation of this example. We start by declaring the contract on the Contract Table *ctBank* (with the specification of the contract) 4.13.

Listing 4.13: Bank.MiniFS's Contract Table

```
1
2  let ctBank =
3    (CT.add "Bank" ([((TNat), "Msg_value"); (TMapping(TAddress, TNat), "Balances");
4                     (TMapping(TAddress, TNat), "balances_param");
5                     ((TNat), "amount"); ((TAddress), "to");
6                     ((TNat), "W_amount");
7                     ],
8                 [(TUnit, "Bank_constructor",[((TNat), "msg_value"); ((TAddress), "sender"); ((
                      ↪ TMapping(TAddress, TNat)), "balances_param");
9                 ((TAddress), "to");
10                ((TNat), "amount");((TNat), "W_amount");]);
11                (TUnit, "transfer",[((TNat), "amount");((TAddress), "to")]);
12                (TUnit, "withdraw",[((TNat), "W_amount")]);
13                (TNat, "deposit", []);
14                (TNat, "get_balance", []);
15                ]) CT.empty);;
```

Then declare the implementation of each function 4.14.

Listing 4.14: Bank.MiniFS's Functions

```
1
2  let param_balances : params = (TMapping(TAddress, TNat), "balances_param")
3  let param_balances_val : term = (TmGetParam("balances_param"))
4  let Bank_constructor : construtor_ = ([param_balances],TmSeq(TmAssign("Balances",
       ↪ param_balances_val), TmReturn(TmValue(VUnit)) ))
5
6  let deposit : func = { requires = TmValue(VTrue); return_type = TUnit; func_name = "deposit";
       ↪ params = []; body = TmSeq( TmMappAss(balances, sender, TmPlus( TmMappSel(balances,
       ↪ sender), msg_value)), TmReturn(TmValue(VUnit))) }
7
8  let get_balance : func = { requires = TmValue(VTrue); return_type = TNat; func_name = "
       ↪ get_balance"; params = []; body = TmReturn( TmMappSel(balances, sender) ) }
9
10 let param_to : params = (TAddress, "to")
11 let param_to_val : term = TmGetParam("to")
12 let param_amount :params = (TNat, "amount")
13 let param_amount_val : term = TmGetParam("amount")
14 let transfer : func = {requires = TmValue(VTrue); return_type = TUnit; func_name = "transfer";
       ↪  params = [param_to; param_amount];
15 body = TmSeq( TmIf( GreaterOrEquals(TmMappSel(balances, sender), param_amount_val),
16 TmSeq( TmMappAss(balances, sender, TmMinus( TmMappSel(balances, sender),param_amount_val)),
17 TmSeq(TmMappAss(balances, param_to_val, TmPlus( TmMappSel(balances, param_to_val),
       ↪ param_amount_val)),TmReturn(TmValue(VUnit)))),
18 TmReturn(TmValue(VUnit))), TmReturn(TmValue(VUnit))) }
19
20 let param_withdraw_amount : params = (TNat, "W_amount")
21 let param_withdraw_amount_val : term = TmGetParam("W_amount")
```

```
22 let withdraw : func = {requires = TmValue(VTrue); return_type = TUnit; func_name = "withdraw";
       ↪  params = [param_withdraw_amount]; body =TmSeq( TmIf( GreaterOrEquals(TmMappSel(
       ↪ balances, sender), param_withdraw_amount_val),
23 TmSeq(TmMappAss(balances, sender, TmMinus( TmMappSel(balances, sender),
       ↪ param_withdraw_amount_val)),
24       TmTransfer( sender,param_withdraw_amount_val ) ),
25       TmReturn(TmValue(VUnit))), TmReturn(TmValue(VUnit))) }
```

Once we have declared every function, we declare an instance of type *contract_tc* 4.15 which is basically the entire contract code.

Listing 4.15: Bank.MiniFS Contract

```
1
2  let contractBank : contract_tc = {
3    name = "Bank";
4    init = ( [((TAddress,"param_init"))] ,GreaterOrEquals(TmMappSel(TmVar("Balances"),
         ↪ TmGetParam("param_init") ), TmValue(VCons(0))));
5    invariant = ([(TAddress, "param_unsafe")],Less(TmMappSel(TmVar("Balances"), TmGetParam("
         ↪ param_unsafe") ), TmValue(VCons(0))));
6    behavioral_types = [];
7    vars = [(TmVar("Msg_value"), TNat);(TmVar("Balances"), TMapping(TAddress,TNat))];
8    cons = Bank_constructor;
9    funcs = [deposit; get_balance; transfer; withdraw];
10 };;
```

We then use this instance to typecheck the contract and in C.2 we can see that the contract is indeed well-typed.

The listing C.3 has the ocaml result of the final Bank contract implementation.

4.3.2 Marketplace

The Marketplace Contract expresses a workflow for a simple transaction between an owner and a buyer. The state transition diagram below shows the interactions among the states in this workflow. As we will observe in the next examples, almost all of them are characterized by a specification in form of a transition state diagram. Since we want our implementations to follow these specifications, we added to Mini-FS behavioral types, and hence, the requires feature, the type *TState* and the.

An instance of the Simple Marketplace application's workflow starts in the ItemAvailable state when an Owner makes an item available for sale by specifying its description and price. A buyer can then make an offer by specifying their price for the item. This action causes the state to change from ItemAvailable to OfferPlaced. Now, if the owner agrees to the buyer's offer, then owner calls the function to accept an offer, and the workflow reaches a successful conclusion state denoted by the Accepted state. If the owner, however, is not satisfied with the offer, then the owner can call the function to reject the offer. On rejection, the state changes to ItemAvailable indicating that the item is still

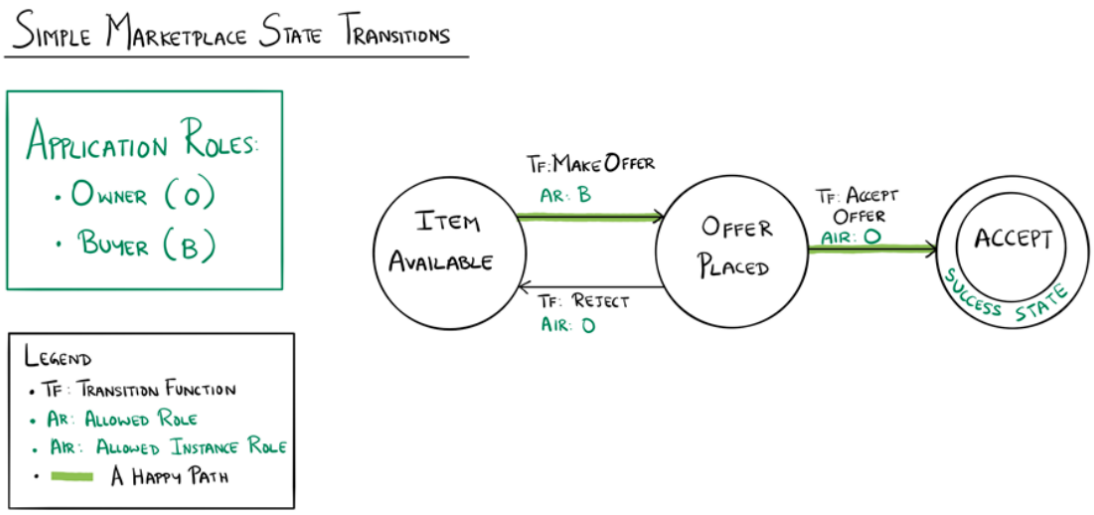

Figure 4.2: Marketplace

up for sale. The transitions between the ItemAvailable and the OfferPlaced states can continue until the owner is satisfied with the offer made.

A happy path shown in the transition diagram traces an owner making an item available, a buyer making an offer, and the owner accepting the offer.

4.3.2.1 Marketplace implementation

In Listing D.1 we can see the implementation of the Marketplace contract. We use behavioral types to enforces the protocol (transition states diagram) explained above.

We start by declaring the *init* function 5.2. The *init* function is used to help the programmer set conditions for the initial state of the contract. Here we set *CurrentState* (a variable that represents the current contract state) and we also state that we want that the value of the asking price is lower than the offer price (so every offer is valid).

Listing 4.16: Marketplace.MiniFS's init

```
1  init =
2   ([],
3    And (Equals (TmVar "CurrentState", TmGetState "ItemAvailable"),
4     And (Greater (TmVar "AskingPrice", TmValue (VCons 0)),
5      Greater (TmVar "OfferPrice", TmVar "AskingPrice"))));
```

Then, we define the invariant (condition that we do not want our contract to violate). In this example we want that in every instance of this contract, the offer's price is always higher than the asking price 4.17.

Listing 4.17: Marketplace.MiniFS's invariant

```
1  invariant = ([], Greater (TmVar "OfferPrice", TmVar "AskingPrice"));
```

Afterwards, we set the behavioral types list 4.18 (every state that our contract can be). Note that each state in the behavioral types list corresponds to a state in 4.2.

Listing 4.18: Marketplace.MiniFS's behavioral types

```
behavioral_types =
   [TmGetState "ItemAvailable"; TmGetState "OfferPlaced";
    TmGetState "Accept"];
```

Once the behavioral types are defined, we can declare each variable by setting its name and the correspondent type 4.19.

Listing 4.19: Marketplace.MiniFS's variables

```
vars =
   [(TmVar "CurrentState", TState); (TmVar "InstanceOwner", TAddress);
    (TmVar "Description", TString); (TmVar "AskingPrice", TNat);
    (TmVar "InstanceBuyer", TAddress); (TmVar "OfferPrice", TNat)];
```

After declaring the variables, the initial values of each variable are set 4.20.

Listing 4.20: Marketplace.MiniFS's constructor

```
  cons =
   ([(TString, "param_description"); (TNat, "param_price")],
    TmSeq
     (TmSeq (TmAssign ("CurrentState", TmGetState "ItemAvailable"),
       TmSeq (TmAssign ("InstanceOwner", TmGetParam "sender"),
        TmSeq (TmAssign ("AskingPrice", TmGetParam "param_price"),
         TmAssign ("Description", TmGetParam "param_description")))),
     TmReturn (TmValue VUnit)));
```

Then, we start implementing the contract's functions. In *make_offer* we start by setting the requires instruction 4.21. This one states that the instance buyer is the only participant able to call this function, that the contract must be in the state *ItemAvailable* and the offer amount must be higher than the item price.

Listing 4.21: Marketplace.MiniFS's make_offer requires

```
requires =
      And (Equals (TmGetParam "sender", TmVar "InstanceBuyer"),
       And (Equals (TmGetState "ItemAvailable", TmVar "CurrentState"),
        Greater (TmGetParam "param_offer", TmVar "AskingPrice")));
```

Then on the function's body 4.22 the new offer is recorded and the current state is set to *OfferPlaced* (hence the application transitions to this state).

Listing 4.22: Marketplace.MiniFS's make_offer transition

```
    body =
     TmSeq
      (TmSeq
        (TmIf (Equals (TmGetParam "param_offer", TmValue (VCons 0)),
```

```
5           TmRevert, TmReturn (TmValue VUnit)),
6         TmSeq
7          (TmIf (Equals (TmGetParam "sender", TmVar "InstanceOwner"),
8            TmRevert, TmReturn (TmValue VUnit)),
9          TmSeq (TmAssign ("InstanceBuyer", TmGetParam "sender"),
10          TmSeq (TmAssign ("OfferPrice", TmGetParam "param_offer"),
11           TmAssign ("CurrentState", TmGetState "OfferPlaced")))),
12       TmReturn (TmValue VUnit));
```

To the owner be able to call the function *accept* or *reject*, the contract must be in the *OfferPlaced* state. The requires defined in these functions enforces this behavior. Then, the current state is set accordingly i.e, if the onwer accepts the offer, the state is set to *Accept*, otherwise the application transitions back to *ItemAvailable*.

The Listing D.2 has the solidity implementation of this Smart Contract. As we can notice, the syntax is slightly different but there is a direct association between MiniFS and Solidity.

4.3.3 Telemetry

The Telemetry Smart Contract covers a provenance scenario with IoT monitoring. It can be described as a supply chain transport scenario where certain compliance rules must be met throughout the duration of the transportation process.

The initiating counter party specifies the temperature and humidity range the measurement must fall in to be compliant. At any point, if the device takes a temperature or humidity measurement that is out of range, the contract state will be updated to indicate that it is out of compliance.

All participants can consult the state and details of the contract at any point in time. The counter party doing the transportation can also specify the next counter party responsible, and the device will ingest temperature and humidity data, which gets written to the chain. This allows the Supply Chain Owner and Supply Chain Observer to pinpoint which counter party did not fulfill the compliance regulations if at any point in the process either the temperature or humidity requirements were not met.

Name	Description
InitiatingCounterParty	The first participant in the supply chain.
Counterparty	A party to whom responsibility for a product has been assigned. For example, a shipper
Device	A device used to monitor the temperature and humidity of the environment the good(s) are being shipped in.
Owner	The organization that owns the product being transported. For example, a manufacturer
Observer	The individual or organization monitoring the supply chain. For example, a government agency

Figure 4.3: Telemetry application roles

Name	Description
Created	Indicates that the contract has initiated and tracking is in progress.
InTransit	Indicates that a Counterparty currently is in possession and responsible for goods being transported.
Completed	Indicates the product has reached it's intended destination.
OutOfCompliance	Indicates that the agreed upon terms for temperature and humidity conditions were not met.

Figure 4.4: Telemetry states

The figure 4.3 describes the roles that a participant can have in this application and the figure 4.4 describes the states that the application can have.

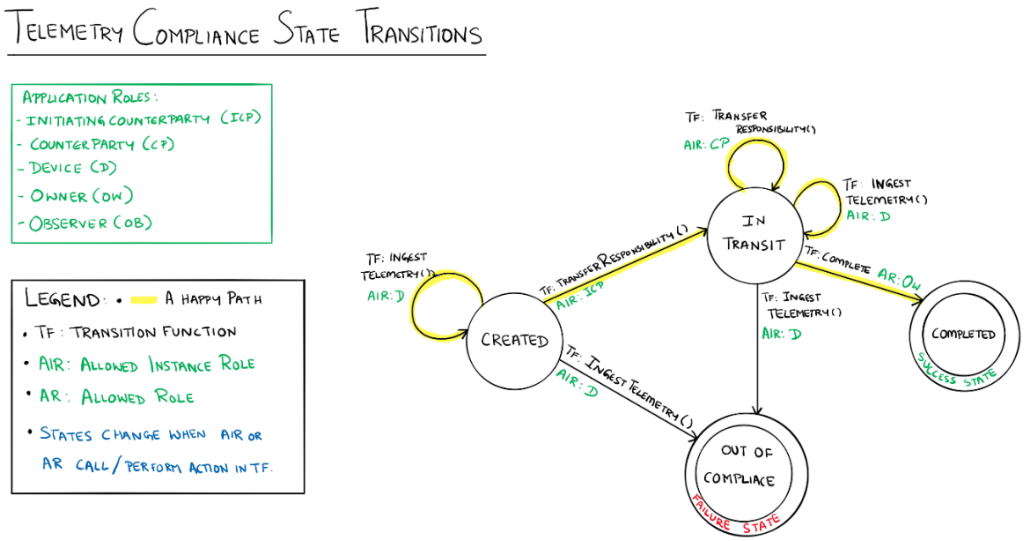

Figure 4.5: Telemetry

The state transition diagram 5.6 articulates the possible flows, and the multiple transition functions at each state that a participant can take. Each user is only allowed to take certain actions depending on the application role. Instance roles indicate that only the user with the application role assigned to the specific contract is able to take actions on the contract.

This contract demonstrates how to collect telemetry information and enforce the specification described in 5.6, essentially, receiving and evaluating temperature and humidity data against an agreed upon acceptable range.

If the device collected data identifies that the telemetry is out of the acceptable range, the contract will shift into an out of compliance state.

In the highlighted happy path, the device ingests readings, which are in compliance throughout the transportation process, while the involved counterparties transfer responsibility until the transportation is completed.

4.3.3.1 Telemetry implementation

In E.1 the implementation of Telemetry contract is described. In short, a party participant can collect data by calling the function *ingestTelemetry* which checks if the data values are inside the agreed range. A party participant can also calls the function *transfer_responsability* so a new party can now be responsible for the transportation. Finally, when the the transporation reaches to an end, the owner can call the function *complete* which ends the application.

In E.2 the solidity implementation of this contract is presented so we can compare it with the MiniFS one.

4.3.4 Digital Locker

The Digital Locker contract expresses a workflow of sharing digitally locked files where the owner of the files controls the access to these files. The state transition diagram below 5.5 shows the interactions among the states in this workflow.

Name	Description
Owner	The owner of the digital asset.
BankAgent	The keeper of the digital asset.
ThirdPartyRequestor	A person requesting access to the digital asset.
CurrentAuthorizedUser	A person authorized to access the digital asset.

Figure 4.6: Digital Locker application roles

Requested	Indicates owner's request to make the digital asset available.
DocumentReview	Indicates that the bank agent has reviewed the owner's request.
AvailableToShare	Indicates that the bank agent has uploaded the digital asset and the digital asset is available for sharing
SharingWithThirdParty	Indicates that the owner is reviewing a third party's request to access the digital asset.
Terminated	Indicates termination of sharing the digital asset.

Figure 4.7: Digital Locker transition states

Figure 4.6 describes the roles that a participant can have in this application. In short, a participant can be an Owner (owner of the document), a Bank agent (the entity that keeps the digital asset), a third party requester (an external user that asks for access to the digital asset) and finally, an authorized user (a user that already has access to the digital asset).

Figure 4.7 describes the states that an application can be.

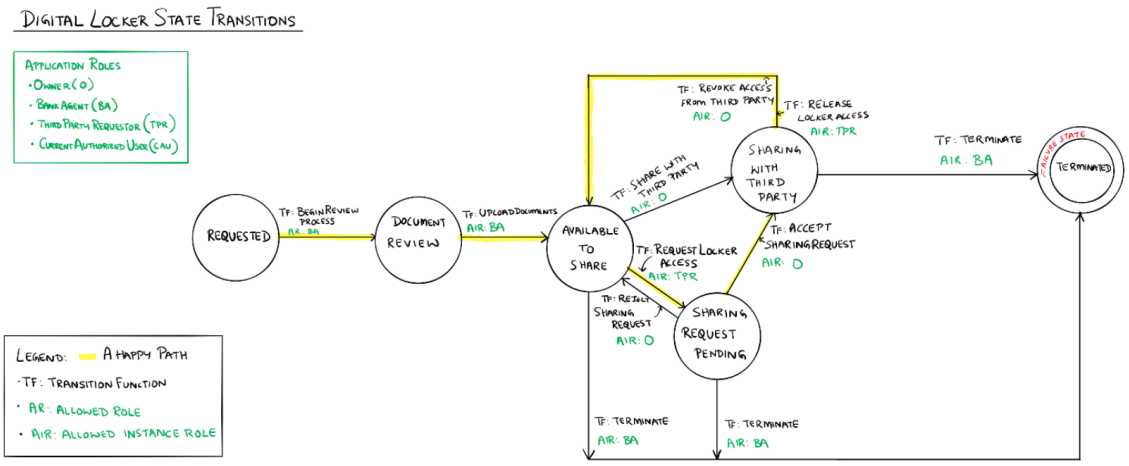

Figure 4.8: Digital Locker

An instance of the Digital Locker application's workflow starts in the Requested state with an Owner requesting a Bank to begin a process of sharing a document.

A BankAgent causes the state to transition to DocumentReview by calling the function BeginReviewProcess indicating that the process to review the request has begun.

Once the review is complete, the document becomes available (the Bank agent uploads the document).

Once the document is available to share (AvailabeToShare state), the document can be shared either with a third party that the owner has identified or any random third-party requester. If the owner specifies the third-party requester, then the state transitions from AvailableToShare to SharingWithThirdParty state. If a random third-party user needs access to the document, then that third-party requester first requests access to the document. At this point, the owner can either accept the request and grant access or reject the request. If the owner rejects the request then the state goes back to AvailableToShare state. If the owner accepts the request to allow the random third-party user to access the document, then the state transitions to SharingWithThirdParty.

Once the third-party user is done with the document, they can release the lock to the document and the state transitions to AvailableToShare. The owner can also cause the state to transition from SharingWithThirdParty to AvailableToShare when they revoke access from the third-party user.

Finally, at any time during these transitions the Bank agent can decide to terminate the sharing of the document once the document becomes available to share.

In F.1 we have the ocaml implementation of this example. This implementation follows the exact diagram specification thanks to the behavioral types features.

For comparison purposes, F.2 presents the solidity implementation of the Digital Locker Smart Contract.

5

Cubicle

5.1 Cubicle Overview

Smart Contracts are usually conceived for an unknown number of users. From a theoretical point of view, they can be seen as parameterized state machines with multiple entry points, shared variables, and a message passing mechanism [18]. Cubicle is a model checker that is used for very safe properties of array based systems. This is a syntactically restricted class of parameterized transition systems with states represented as arrays indexed by an arbitrary number of processes. The Smart Contract examples presented in the previous sections are described by transition states diagrams that can be described as an array system (set of states). Therefore, Cubicle is suitable to verify safe properties on this examples.

5.1.1 Cubicle model

Cubicle is an SMT-based model checker for parameterized transition systems. It allows the verification of safe properties in an automatic fashion way. We can describe a contract as a an array based system (with behavioral types protocols) and cubicle will run this contract and try to reach an unsafe state (following the protocols specification).

Therefore, Cubicle will detect if these unsafe states were reached, and if yes, it will show the trace that lead to the unsafe states. Cubicle has lot of features like:

- **Type, Variable and Array Declarations**. Cubicle has several built-in data types, among which are integers (int), booleans (bool), and process identifiers (proc). Additionally, the user can define enumerations.
- **Initial states**.The content of a system state is fully characterized by the value of its global variables and arrays. The initial states are defined by an init formula given as a universal conjunction of literals. Note the resemblance between this init formula and the one added in MiniFS.
- **Transitions**. The execution of a parameterized system is defined by a set of guard/action transitions. It consists of an infinite loop which non-deterministically triggers

at each iteration a transition whose guard is true and whose action is to update state variables. Each transition can take one or several process identifiers as arguments. A guard is a conjunction of literals and an action is a set of variable assignments or array updates.

- **Unsafe States**. The safety properties to be verified are expressed in their negated form and characterize unsafe states. They are given by existentially quantified formulas.

5.1.2 Cubicle Examples

In this section, we present the implementation of several examples (some defined in the previous chapter) with the purpose of showing the usefulness of Cubicle.

5.1.2.1 Marketplace

Let's recall the Marketplace example explained in the previous section. Marketplace

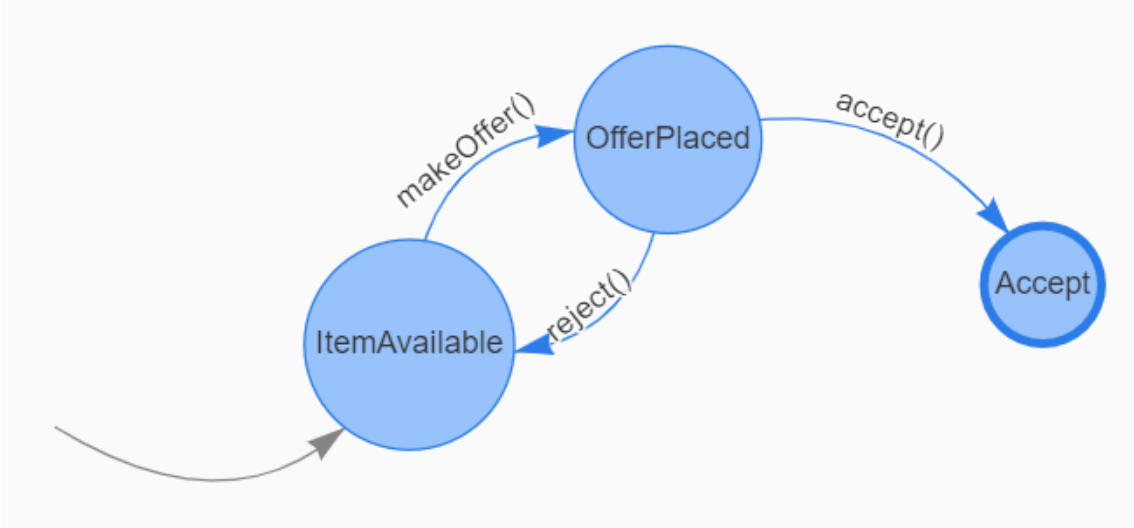

Figure 5.1: Marketplace protocol

has two types of participant roles, owner and buyer. In short, an owner has an item and a buyer can make an offer to it, that can be accepted or rejected. We used [27] to generate the actual state diagram that states the behavioral types. If we compare it with the states diagram 4.2 we can observe that this ones are similiar.

Let's start with defining the variables of this example.

Listing 5.1: Marketplace variables

```
1 type marketplace_state = ItemAvailable | OfferPlaced | Accept
2 var MarketplaceState : marketplace_state
3 var Owner : proc
4 var Buyer : proc
```

```
5 var AskingPrice : int
6 var OfferPrice : int
7 array Random_Offers [proc] : int
```

Listing 5.1 shows the variables that are needed to implement this example. In line 1, an enumerate is declared. This enumerate represents the behavioral types in 4.2 and *MarketplaceState* variable represents the current state that the application is in. Then, *Owner* and *Buyer* represents the participants owner and buyer. Here, the type proc represents a user that calls a function. Finally, there is the declaration of *AskingPrice*, *OfferPrice* and the array *Random_Offers*. In Cubicle, there isn't parameters of type int. So when a buyer tries to make an offer, we use the *random_offers* array to generate a random offer to the buyer.

After declaring the variables, the *init* state of the application is set via *init* function.

Listing 5.2: Marketplace init

```
1 init( i ) {Random_Offers[i] > 0 && MarketplaceState = ItemAvailable && AskingPrice > 0 }
```

Here, it is stated that every user makes offers bigger than zero, the asking price of an item is always positive and the initial state of the protocol is set to *ItemAvailable*.

Listing 5.3: Marketplace unsafe

```
1 unsafe (i ) { OfferPrice <= 0 && MarketplaceState = Accept }
```

In 5.3 the unsafe properties that we want to check are set. Basically, this means that if a current offer is negative or zero and the current state is "Accept" that the contract is unsafe. Therefore, Cubicle will try to achieve this state respecting its specifications.

In 5.4 there are the transitions of Marketplace.

- *Make_Offer* receives a parameter i (a potential buyer). The guard in requires states that this transitition can only be executed if the current state is *ItemAvailable*. In make_offer's body, the current offer *OfferPrice* is set and after, the current state is set to *OfferPlaced*.
- *Accept*. This transition states that the owner, if the current state is *OfferPlaced*, can accept an offer and consequently, set the state to *Accept*.
- *Reject*. This transition states that the owner, if the current state is *OfferPlaced*, can reject an offer and consequently, set the state to *ItemAvailable*.

Listing 5.4: Marketplace.cub

```
1
2 transition make_offer(i)
3 requires{MarketplaceState = ItemAvailable}
4 {
5     OfferPrice := Random_Offers[i];
6     MarketplaceState := OfferPlaced;
```

```
}

transition accept(i)
requires{i = Owner && MarketplaceState = OfferPlaced}
{
    MarketplaceState := Accept
}

transition reject(i)
requires{i = Owner && MarketplaceState = OfferPlaced}
{
    MarketplaceState := ItemAvailable
}
```

In this way, the specification defined in 4.2 is ensured. After running this contract, cubicle produces the following output.

```
$ cubicle Marketplace-azure.cub
node 1: unsafe[1]
                                                                                    1 (1+0) remaining

node 2: accept(#1) -> unsafe[1]
                                                                                    2 (1+1) remaining

node 3: make_offer(#1) -> accept(#1) -> unsafe[1]
                                                                                    2 (1+1) remaining

node 4: reject(#1) -> make_offer(#1) -> accept(#1) -> unsafe[1]
                                                                                    3 (2+1) remaining

node 5: make_offer(#2) -> accept(#1) -> unsafe[1]
                                                                                    1 (1+0) remaining

node 6: reject(#1) -> make_offer(#2) -> accept(#1) -> unsafe[1]
                                                                                    3 (3+0) remaining

================================================================================================
* STATS
------------------------------------------------------------------------------------------------
Number of visited nodes          : 6
Fixpoints                        : 5
Number of solver calls           : 10
Max Number of processes          : 2
Number of deleted nodes          : 0
Restarts                         : 0
================================================================================================


The system is SAFE
```

Figure 5.2: Cubicle's marketplace output

5.1.2.2 Markeptlace 2.0

Let's now consider an updated version of Markeplace. The specification 4.2 remains the same. The big difference is that in this example there's only one buyer and his balance is checked and updated.

Listing 5.5: Marketplace2.0 vars

```
type marketplace_state = ItemAvailable | OfferPlaced | Accept
var MarketplaceState : marketplace_state
var Owner : proc
var Buyer : proc
var OwnerBalance : int
var BuyerBalance : int
var Offer : int
var ItemValue : int
var IsSold : bool
array Random_Offers [proc] : int
```

Now, there are three extra variables, *OwnerBalance*, *BuyerBalance* and *IsSold*.

In 5.6, the inital state of Marketplace is set. The differences from the previous version are that here, owner's balance and buyer's balance are defined as positives and that the buyer should have enough balance to buy the item. There's also an extra flag *IsSold* stating if the item was already sold.

Listing 5.6: Marketplace2.0 init

```
init( i ) {Random_Offers[i] > 0 && OwnerBalance >= 0 && BuyerBalance >= 0 && BuyerBalance >
    ↪ ItemValue && MarketplaceState = ItemAvailable && IsSold = False }
```

In 5.7 the undesired unsafe properties are also set. This implementation is unsafe if:

1. Buyer's balance becomes negative.
2. Owner's balance becomes negative.
3. The state is "Accept" but the item hasn't been sold yet.

Listing 5.7: Marketplace2.0 unsafe

```
    unsafe (i) { BuyerBalance < 0 || OwnerBalance < 0 || (MarketplaceState = Accept && IsSold =
        ↪ False) }
```

Listing 5.8: Marketplace2.0 transitions

```

transition make_offer(i j)
requires{i = Buyer && MarketplaceState = ItemAvailable && BuyerBalance >= Random_Offers[j] &&
    ↪ Random_Offers[j] >= ItemValue }
{
    Offer := Random_Offers[j];
    MarketplaceState := OfferPlaced;
}

transition accept(i)
requires{i = Owner && MarketplaceState = OfferPlaced && IsSold = False}
{
    OwnerBalance := OwnerBalance + Offer;
    BuyerBalance := BuyerBalance - Offer;
```

```
14      IsSold := True;
15      MarketplaceState := Accept;
16  }
17
18  transition reject(i)
19  requires{i = Owner && MarketplaceState = OfferPlaced}
20  {
21      MarketplaceState := ItemAvailable;
22      IsSold := False;
23  }
```

In 5.8 there are the multiple transitions:

- *Make_offer*. In this transition, if the user i is Buyer, the item is available and the buyer has enough balance to make that offer, and the offer is valid (bigger than the Item value) the offer will be placed.
- *Accept*. If the user i is the Owner, the item hasn't been sold yet and the offer is placed, he can accepts the offer and update the balances.
- *Reject*. If the user i is the Owner he can then reject the offer.

If we run this example with Cubicle we will obtain the following output:

```
$ cubicle Marketplace.cub
node 1: unsafe[1]
                                                                    2 (2+0) remaining

node 2: unsafe[2]
                                                                    4 (3+1) remaining

node 3: unsafe[3]
                                                                    6 (4+2) remaining

node 4: accept(#1) -> unsafe[2]
                                                                    7 (2+5) remaining

node 5: accept(#1) -> unsafe[3]
                                                                    8 (0+8) remaining


Unsafe trace: make_offer(#2, #3) -> accept(#1) -> unsafe[3]
================================================================================================
* STATS
------------------------------------------------------------------------------------------------
Number of visited nodes          : 5
Fixpoints                        : 2
Number of solver calls           : 14
Max Number of processes          : 1
Number of deleted nodes          : 0
Restarts                         : 0
================================================================================================

Error trace: Init -> make_offer(#2, #3) -> accept(#1) -> unsafe[3]


UNSAFE !
```

Figure 5.3: Cubicle's marketplace2.0 output

After observing the output in 5.3, we can conclude that the trace " init -> make_offer(2,3) -> accept(3) " lead to an unsafe state ("2" and "3" are the users ids).

Transition *Accept* doesn't check some important conditions. With the current implementation, a Buyer can make a negative offer or he can also place an offer for which he does not have sufficient balance. Either way, this will lead to negative balances and consequently, unsafe states. Listing 5.9 shows the correct implementation that makes the whole contract safe.

Listing 5.9: Marketplace2.0 accept fix

```
1  transition accept(i)
2   requires{i = Owner && MarketplaceState = OfferPlaced && Offer > 0 && Offer < BuyerBalance
        ↪ && IsSold = False}
3   {
4       OwnerBalance := OwnerBalance + Offer;
5       BuyerBalance := BuyerBalance - Offer;
6       IsSold := True;
7       MarketplaceState := Accept;
8   }
```

The figure 5.4 shows the final output of the fixed version of this example.

```
$ cubicle Marketplace.cub
node 1: unsafe[1]
                                                                                2 (2+0) remaining

node 2: unsafe[2]
                                                                                4 (3+1) remaining

node 3: unsafe[3]
                                                                                6 (4+2) remaining

================================================================================================================
* STATS
----------------------------------------------------------------------------------------------------------------
Number of visited nodes          : 3
Fixpoints                        : 6
Number of solver calls           : 8
Max Number of processes          : 1
Number of deleted nodes          : 0
Restarts                         : 0
================================================================================================================


The system is SAFE
```

Figure 5.4: Cubicle's marketplace2.0 final output

5.1.3 Digital Locker

Let's recall the Digital Locker Smart Contract 5.5. In this example, basically a user (Owner) can trust to a another participant (Bank agent) to manage the sharing of a digital asset. The listing G.1 shows the Cubicle implementation of this contract.

We start by declaring an enumerate *state* 5.10 that represents every state that our Contract can be in. Note that every state in this enumerate corresponds to a state in 5.5.

Listing 5.10: DigitalLocker.cub state

```
1  type state = DocumentReview | Requested | AvailableToShare | SharingWithTp | SharingRP |
       ↪ Terminated
```

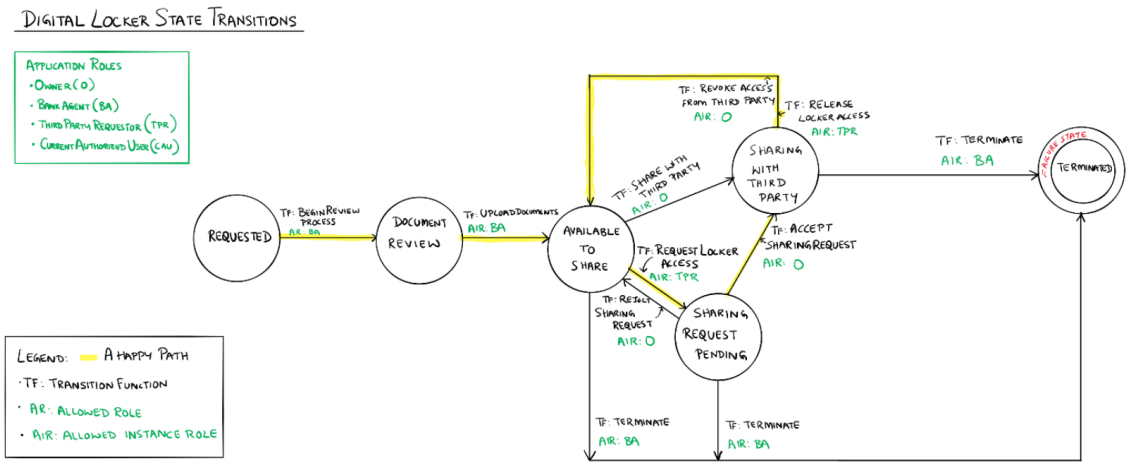

Figure 5.5: Digital Locker

Since Cubicle doesn't have strings, we use an enumerate *string* to represent every possible string that will be used in this Contract.

Listing 5.11: DigitalLocker.cub string

```
type string = Pending | Rejected | Shared | Approved | Available
```

Then, in a similar way to its MiniFS F.1 and Solidity F.2 implementation, we declare the remaining variables 5.12.

Listing 5.12: DigitalLocker.cub variables

```
var Owner : proc
var CurrentState : state
var BankAgent : proc
var LockerId : string
var CurrentAuthorizedUser : proc
var Exp_Date : int
var Image : string
var TPRequestor : proc
var LockerStatus : string
array GlobalBalances[proc] : int
array Random_param_image[proc] : string
array Random_param_locker_id[proc] : string
array Random_param_exp_date[proc] : int
var Revert : bool
```

In 5.13 we also declare the *init* state of the Contract and we also declare that a contract instance is *unsafe* if the Bank agent is sharing and item and its locker status is *Available*.

Listing 5.13: DigitalLocker.cub init and unsafe

```
init () { Requested = CurrentState}
unsafe (LockerStatus = Available && CurrentState = SharingWithThirdParty ) { not( true)}
```

The functions in F.2 corresponds to Cubicle's transitions 5.14, and in each function/-transition the requires instruction has the pre conditions that support the behavioral types protocols.

Listing 5.14: DigitalLocker.cub functions

```
transition begin_process_review(sender)
requires{ Owner = sender && Requested = CurrentState}
{
   BankAgent := sender;
  LockerStatus := Pending;
 CurrentState := DocumentReview
}

transition reject_application(sender)
requires{ BankAgent = sender && DocumentReview = CurrentState}
{
   BankAgent := sender;
  LockerStatus := Rejected;
 CurrentState := DocumentReview
}

transition upload_documents(sender)
requires{ BankAgent = sender && DocumentReview = CurrentState}
{
   LockerStatus := Approved;
  Image := Random_param_image[sender] ;
  LockerId := Random_param_locker_id[sender] ;
 CurrentState := AvailableToShare
}

transition share_with_tp(sender param_tp)
requires{ Owner = sender && AvailableToShare = CurrentState}
{
   TPRequestor := param_tp;
  LockerStatus := Shared;
  Exp_Date := Random_param_exp_date[sender] ;
  CurrentAuthorizedUser := param_tp;
 CurrentState := SharingWithTp
}

transition accept_s_request(sender)
requires{ Owner = sender && CurrentState = SharingRP}
{
   CurrentAuthorizedUser := TPRequestor;
 CurrentState := SharingWithTp
}

transition reject_s_request(sender)
requires{ Owner = sender && CurrentState = SharingRP}
{
```

```
46    LockerStatus := Available;
47  CurrentState := SharingWithTp
48 }
49
50 transition request_l_access(sender)
51 requires{ not( Owner = sender) && CurrentState = SharingRP}
52 {
53    TPRequestor := sender;
54  CurrentState := SharingRP
55 }
56
57 transition release_l_access(sender)
58 requires{ CurrentAuthorizedUser = sender && CurrentState = SharingWithTp}
59 {
60    LockerStatus := Available;
61  CurrentState := AvailableToShare
62 }
63
64 transition revoke_access(sender)
65 requires{ Owner = sender && CurrentState = SharingWithTp}
66 {
67    LockerStatus := Available;
68  CurrentState := AvailableToShare
69 }
70
71 transition terminate(sender)
72 requires{ not( CurrentState = DocumentReview) && not( CurrentState = Requested) && Owner =
       ↪ sender}
73 {
74    LockerStatus := Available;
75  CurrentState := Terminated
76 }
```

5.1.3.1 Telemetry.Cub vs Telemetry.MiniFS

Let's recall the Telemetry contract, explained in the previous chapter.

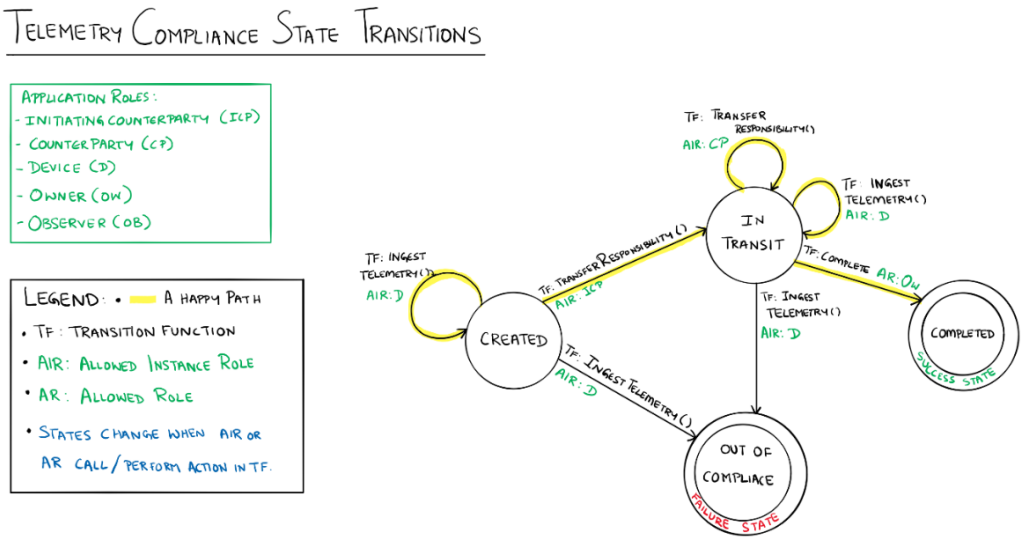

Figure 5.6: Telemetry

In this contract, a participant can essentially be an owner, a party (that carries the product) or a device that reads humidity and temperature data. As it was explained before, in this contract, a cargo is transported from a point to another by a current party that should make sure that the temperature/humidity values are inside an agreed range. Listing 5.15 has the full and safe implementation in cubicle of this contract.

Listing 5.15: Telemetry.cub

```
1  type ref_state = Created | InTransit | OutOfCompliance | Completed
2  var CurrentState : ref_state
3  var CurrentCounterParty : proc
4  var Owner : proc
5  var Device : proc
6  var MaxTemp : int
7  var MinTemp : int
8  var ComplianceStatus : bool
9  array RandomTemps[proc] : int
10
11 init( i ) { MaxTemp > MinTemp && CurrentState = Created && ComplianceStatus = True}
12
13 unsafe( i ) { (ComplianceStatus = False && CurrentState <> OutOfCompliance) }
14
15 transition ingest_telemetry(i)
16 requires{i = Device && (CurrentState = Created || CurrentState = InTransit) }
17 {
18     ComplianceStatus := case
19                         | (RandomTemps[i] > MaxTemp || RandomTemps[i] < MinTemp): False
```

```
20                      | _ : True;
21      CurrentState := case
22                      | (RandomTemps[i] > MaxTemp || RandomTemps[i] < MinTemp): OutOfCompliance
                            ↪ ;
23  }
24
25  transition transfer_responsability(i j)
26  requires{i = CurrentCounterParty && (CurrentState = Created || CurrentState = InTransit)}
27  {
28      CurrentCounterParty := j;
29      CurrentState := case
30                      | CurrentState = Created : InTransit;
31  }
32
33  transition complete(i)
34  requires{i = Owner && CurrentState = InTransit}
35  {
36      CurrentState := Completed;
37  }
```

If we compare this listing with Telemetry.MiniFS E.1 some similarities can be observed:

- **The *init* function**. Both languages have the *init* function that set the Contract initial state.
- **The unsafe**. In MiniFS there is the invariant feature. This feature make sure that some properties are guaranteed. In Cubicle there is the unsafe feature. Unsafe instruction make sure that if a cubicle program reaches a unsafe state, this program will be characterized as unsafe. Both instructions do essentially the same.
- **Var declarations**. Both languages have similar variables declarations.
- **Functions**. MiniFS's functions corresponds to Cubicle's transitions. Both have a name, a params list, a requires feature and finally, a body.

5.2 MiniCub

As noted above, Cubicle is extremely useful for checking safety properties. One of this dissertation goals it's to use this tool to typecheck Smart Contracts. Therefore, Smart Contracts implemented in MiniFS (after typechecked) have to be translated to Cubicle.

Since translating a contract directly from MiniFS to Cubicle is a complex process, we defined a language, MiniCub, which is an intermediary between Cubicle and MiniFS.

Smart Contracts written in MiniCub should be identical to their Cubicle version but should also resemble their MiniFS version.

5.2.1 MiniCub Grammar

The listing 5.16 shows the MiniCub contract's structure. In H.1 we can see the full MiniCub's grammar implemented in ocaml. MiniCub's grammar is essentially the same as the Cubicle one but it is in a format that simplifies the translation between MiniFS and Cubicle.

Listing 5.16: MiniCub's contract

```
type contract_cub = { (*meter init e unsafe*)
    name : string;
    init : (exp list) * exp;
    unsafe : (exp list) * exp;
    behavioral_types: behavioral_types;
    vars: cub_vars;
    funcs: cubfunc_ list
}
```

5.2.2 Marketplace contract in MiniCub

Let's consider the Marketplace 4.2 Contract. Listing I.1 has the MiniCub version of Marketplace example.

A contract in MiniCub is composed by:

- A name (string), in this case, *Markeptlace*

```
{name = "Marketplace";
```

- An init and an unsafe function (like Cubicle and MiniFS).

```
{ init =
   ([],
    C_And (C_Equals (C_GetVar "CurrentState", C_GetState "ItemAvailable"),
     C_And (C_Greater (C_GetVar "AskingPrice", C_Value (CCons 0)),
      C_Greater (C_GetVar "OfferPrice", C_GetVar "AskingPrice"))));
  unsafe =
   ([], C_Not (C_Greater (C_GetVar "OfferPrice", C_GetVar "AskingPrice")));
  behavioral_types =
   [C_GetState "ItemAvailable"; C_GetState "OfferPlaced";
    C_GetState "Accept"];
```

- A set of variables.

```
 vars =
   [(C_GetVar "CurrentState", State); (C_GetVar "InstanceOwner", Proc);
    (C_GetVar "Description", Proc); (C_GetVar "AskingPrice", Int);
    (C_GetVar "InstanceBuyer", Proc); (C_GetVar "OfferPrice", Int);
    (C_GetVar "GlobalBalances", CArr (Proc, Int));
    (C_GetVar "Random_param_offer", CArr (Proc, Int));
    (C_GetVar "Random_param_offer_r", CArr (Proc, Int));
```

```
8     (C_GetVar "Random_param_offer_a", CArr (Proc, Int))];
```

- A set of behavioral types.

```
1  behavioral_types =
2    [C_GetState "ItemAvailable"; C_GetState "OfferPlaced";
3     C_GetState "Accept"];
```

- A set of functions (where each function has a name, parameters list, requires feature and body).

```
1   funcs =
2    [{requires_ = (...)
3      name_ = "make_offer"
4      body_ = (...)
5     {requires_ = (...)
6      name_ = "reject"
7      body_ = (...)
8     {requires_ = (...)
9      name_ = "accept"
10     body_ = (...)
```

5.2.2.1 Marketplace.Cub vs Marketplace.MiniCub

After comparing Marketplace.MiniCub I.1 and Marketplace.cub 5.4, we can observe that the grammar between Cubicle and MiniCub is literally the same. The big difference here is that MiniCub's contracts have the same structure as the MiniFS's ones, which facilitates the translation between MiniFS and Cubicle.

5.3 MiniFs to MiniCub Translator

Since one of the main goals of this project is to use cubicle to model check smart contracts, a translator of smart contracts from MiniFS to MiniCub was implemented.

This translator has a function *parse_term* that receives a MiniFS term and produces a MiniCub exp. In short, this function recursively evaluates by branches the MiniFS term, and for reach instruction in MiniFS, a suitable exp in MiniCub is produced.

The main function this translator is called *parse_contract*. This one essentially receives a MiniFS contract and produces a MiniCub version of it. Listing 5.17 shows the ocaml implementation of this function.

Listing 5.17: Parse_Contract function

```
1  let parse_contract contract_name fsvars gamma funcs bt unsafe_fs init_fs =
2    let cubvars = ref [] in
3    let cubfuncs = ref[] in
4    let new_unsafe_params = ref[] in
```

```
5  let new_init_params = ref[] in
6  let (unsafe_params, unsafe_cond) = unsafe_fs in
7  let (init_params, init_cond) = init_fs in
8  parse_unsafe_or_init_params unsafe_params "unsafe" cubvars new_unsafe_params gamma;
9  parse_unsafe_or_init_params init_params "init" cubvars new_init_params gamma;
10 Printf.printf "----fsvars----------------------------------------------------Parsing␣vars
       ↪ ...-----------------------------------------------------------␣\n";
11 ignore(parse_vars fsvars cubvars gamma);
12 cubvars := !cubvars @ [(C_GetVar("GlobalBalances"),CArr(Proc,Int) )];
13 Printf.printf "-----------------------------------------------------Parsing␣funcs
       ↪ ...-----------------------------------------------------------␣\n";
14 process_fun funcs gamma cubvars cubfuncs;
15 let cub_bt = ref[] in
16 List.iter (fun(x) -> (cub_bt := !cub_bt @ [parse_term x gamma (ref[]) cubvars (ref[]) ""]))
       ↪ bt;
17 let new_contract : contract_cub = {
18   name = contract_name;
19   init = ((!new_init_params),(parse_term (init_cond) (gamma) (ref[]) cubvars (ref[]) ""));
20   unsafe = (!new_unsafe_params,((parse_term (Not(unsafe_cond)) (gamma) (ref[]) cubvars (ref
         ↪ []) "")));
21   behavioral_types = !cub_bt;
22   vars = !cubvars;
23   funcs = !cubfuncs;
24 } in
25 new_contract;;
```

This function receives the MiniFS contract info and then for each field (init, unsafe, vars, functions) produces a translation to the MiniCub suitable field. Finally it constructs and returns the new MiniCub contract.

5.3.1 Difficulties and Workarounds

As it was mentioned before, the contracts structure in MiniFS is essentially the same as the one in MiniCub. However, there are some big differences in their grammar and consequently, some solutions we're took to solve this dissimilarities.

- **Behavioral types**. To Cubicle's final contracts be able of follow the behavioral types protocols, some logic had to be implemented. When translating a contract, the behavioral types are converted to an enumerate *type*. Then the first var that is created is called *CurrentState* and has the type *type* (*CurrentState* can have one of the values of the enumerate *type*). This way, the *CurrentState* will symbolise the current transition state that the program is in.

- **Types**. There are some differences between MiniFS's types and MiniCub ones. For example, there aren't strings in Cubicle. To solve this problem, two approaches can be taken:

1. The developer change this strings to constants and uses Integers instead of strings.

2. The developer uses strings and the translator creates an enumerate with the possible strings that a variable can have. For example, Digital Locker contract 5.5 has two variables *LockerStatus* and *state* that can have the following values: *Pending*, *Rejected*, *Approved*, *Shared* and *Available*. This way an enumerate is created with this values, and the var *LockerStatus* can have the values of this enumerate. For example, in 5.15 the *ref_state* is the enumerate *type* and *CurrentState* is the variable that stores the current value of the Contract state.

- **Parameters**. Cubicle only allows parameters of type *proc* (*TAddress* in Minifs), on the other hand, MiniFS's parameters can have any type. To solve this problem, when a parameter doesn't have type *proc*, a random value it's generated by cubicle to replace it.

  For example, Marketplace's make offer receives a parameter *offerPrice* of type *TNat*. If we observe Marketplace's cubicle implementation I.1, this parameter doesn't exists and a random generated value is used instead. These random values can still be ruled by the *init* function.

- **Arithmetic operations with arrays**. Cubicle doesn't allow arithemtic operations with arrays, when this happens, a local var is created containing a copy of one of the arrays.

- **Sender**. Cubicle doesn't a msg.sender feature. In cubicle, the sender (caller of the function) is always received by parameter.

- **Balance**. In Cubicle generated contracts, there is always a var called *Global_Balances*. This var stores the balances of each sender (user). This way, the MiniFS operation *Balance(user)* can still be used.

- **If conditions**. In Cubicle, the associated instruction to *If* is the *Case* instruction. Each case is composed by a var (to be affected) and a set of conditions * var values. In a MiniFS *If* instruction when more than one var is changed, in MiniCub, a *Case* instruction is created to each var change. For example, the if condition present in the *ingest_telemetry* from the Telemetry example (E.2 line 75) is converted to the case instruction in 5.15 line 19.

- **Inter-Contract behavior**. Cubicle doesn't work with Contracts that interacts with other Contracts. To overcome this hurdle, the only solution is to write the multiple contract in one and to simluate its behavior (interactions) through behavioral types (states diagrams protocols).

- **Revert**. Cubicle doesn't have Soldity's feature *Revert*, to simulate it, every contract as a boolean var *revert* and every transition verifies (through the *requires*) if this boolean is true or false.

5.4 MiniCub to Cubicle Parser

After the translation from MiniFS to MiniCub is concluded, the parsing between MiniCub to Cubicle can finally happen.

This parser, has a function called *parse_exp* which receives an expression of MiniCub an write it to a buffer. This function recursively evaluates an expression and writes the appropriated info to the buffer.

The main function of this parser is called the *parse_contract* which receives a MiniCub contract as a parameter. Basically it creates a file.cub (the final cubicle contract), creates a buffer and then calls multiple functions.

Each function translates a contract field and adds the appropriated info to the buffer. Listing 5.18 shows parse_contract's implementation.

Listing 5.18: Parse contract

```
let rec parse_contract contract=
  let file = contract.name^"_parsed.cub" in
  let oc = open_out file in
  let buffer_vars = Buffer.create 32 in
  let s = begin_parsing_bts contract.behavioral_types "" ^ "␣\n" in
  Buffer.add_string (buffer_vars) (s);
  ignore(parse_vars contract.vars buffer_vars );
  parse_init contract.init buffer_vars contract.vars;
  parse_unsafe contract.unsafe buffer_vars contract.vars;
  List.iter ( fun(x) -> parse_fun x buffer_vars contract.vars) contract.funcs;
  Buffer.output_buffer oc buffer_vars;
  close_out oc;
```

Once the translation is finished, the buffer is dumped to the created .cub file and the process ends. Finally we have a parsed and typeckecked contract that can be model checked by cubicle.

5.5 Marketplace Full example

The following figure 5.7 shows a diagram that explains a full example of this dissertation contribution.

Let's consider the Marketplace contract. First, the programmer writes this contract in MiniFS D.1. This contract will be typechecked by MiniFS's typesystem and consequently, will be well typed. After this, the contract will be translated to MiniCub. Then, the parsing between MiniCub and Cubicle is done and finally, Cubicle executes this program to determine if it is safe or not i.e, if it matches its specifications.

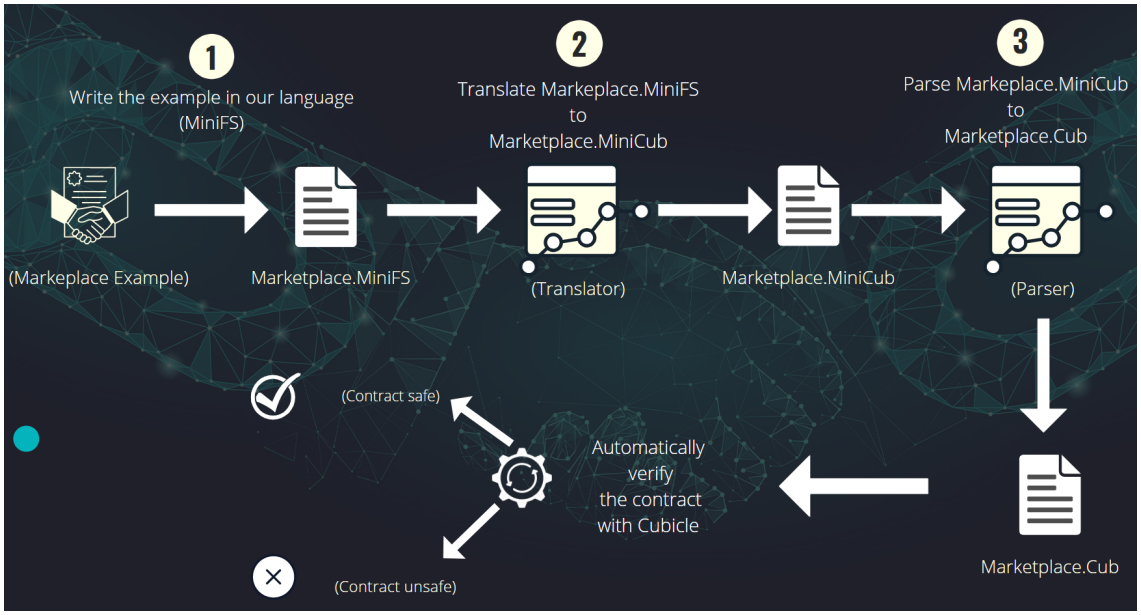

Figure 5.7: Marketplace complete example

5.5.0.1 Methodology

In this section we will explain how to use the tools developed in this project. First, we start by cloning the repository [20]. Then we write our MiniFS contract in the Minifs.ml file. After writing the contract we can typecheck it by using the function *typecheck_contract* that receives the contract implemented as an argument.

After the typechecking, we can use the function *parse_contract* (present in parser.ml) to translate the MiniFS Contract to a MiniCub contract. Following this conversion, we use the function *parse_contract* (from the file parsercub.ml) to convert the MiniCub Contract to a Cubicle contract. Finally if needed, we can add some extra conditions to the *unsafe* instruction (that we want to verify) and finally, we use Cubicle to automatically verify this contract.

6

Conclusions and Future Work

6.1 Conclusions

Since Smart Contracts after deployed on the Blockchain can't be ammended, it is important to ensure static safety (static type verification) so some execution errors can be prevented (prior to the deployment).

Therefore, it was concluded that the use of MiniFS and its Type System is essential to guarantees static correctness properties.

In this dissertation, it was also concluded that the use of behavioral types protocols prevents a lot of errors (e.g. a user invoking methods by an invalid order). Therefore, the addition of behavioral types and assertions to MiniFS was essential to prevent this errors and to ensure liquidity.

Finally, it was also concluded that model checking has also a big role in ensuring some safety properties in smart contracts, especially, in an automatic fashion way. Thus, the use of Software Verification Tool Cubicle is indispensable to our approach because it allows the programmer to verify that a contract has some desired properties and follows its specifications.

6.2 Future Work

In spite of our contribution being useful to detect/prevent bugs and unexpected behaviours in smart contracts, there is still work to be done.

Firstly, since Featherweight Solidity is a mainstream language, it would be also useful to implement a parser between Featherweight Solidity and MiniFS. We believe that this parser can be simple to implement in view of the fact that its grammars are similar.

Secondly, it would be also useful to automatically generate cubicle unsafe instructions (contract undesired properties). To this task, we suggest that some violations of behavioral types are automatically generated and used in the unsafe feature. (Note that here, the programmer could still manually add and change these instructions).

Lastly, our behavioral types protocol implementation is simple and can still be improved with a lot of features. In our solution, this behavioral types are checked in cubicle, we believe that checking this behavioral types on compilation time would be a better approach.

Bibliography

[1] URL: https://www.blockchain-council.org/blockchain/best-programming-languages-to-build-smart-contracts/ (cit. on p. 19).

[2] URL: https://docs.soliditylang.org/en/v0.8.12/ (cit. on p. 19).

[3] URL: https://vyper.readthedocs.io/en/stable/ (cit. on p. 19).

[4] URL: https://github.com/cornellblockchain/bamboo (cit. on p. 20).

[5] URL: https://developers.diem.com/docs/technical-papers/move-paper/ (cit. on p. 20).

[6] URL: https://wiki.tezosagora.org/learn/smartcontracts/michelson (cit. on p. 20).

[7] URL: https://www.certik.com/resources/blog/technology/an-introduction-to-deepsea (cit. on p. 21).

[8] URL: https://github.com/flintlang/flint (cit. on p. 21).

[9] URL: https://docs.soliditylang.org/en/v0.8.11/smtchecker.html (cit. on p. 22).

[10] S. L. Akash Lal Michal Moskal. *Boogie: An Intermediate Verification Language*. 2008. URL: https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language (cit. on p. 27).

[11] *An Introduction to Move*. 2022. URL: https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples (cit. on p. 28).

[12] N. Atzei et al. "Developing secure bitcoin contracts with BitML". In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by M. Dumas et al. ACM, 2019, pp. 1124–1128. DOI: 10.1145/3338906.3341173. URL: https://doi.org/10.1145/3338906.3341173 (cit. on p. 21).

[13] M. Azure. *Azure samples*. 2019. URL: https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples (cit. on pp. 3, 39).

[14] L. Cardelli. "Type Systems". In: *The Computer Science and Engineering Handbook*. Ed. by A. B. Tucker. CRC Press, 1997, pp. 2208–2236 (cit. on pp. 17, 18).

[15] Y. Chinen et al. "RA: A Static Analysis Tool for Analyzing Re-Entrancy Attacks in Ethereum Smart Contracts". In: *J. Inf. Process.* 29 (2021), pp. 537–547. DOI: 10.2197/ipsjjip.29.537. URL: https://doi.org/10.2197/ipsjjip.29.537 (cit. on p. 16).

[16] Y. Chinen et al. "RA: A Static Analysis Tool for Analyzing Re-Entrancy Attacks in Ethereum Smart Contracts". In: *J. Inf. Process.* 29 (2021), pp. 537–547. DOI: 10.2197/ipsjjip.29.537. URL: https://doi.org/10.2197/ipsjjip.29.537 (cit. on pp. 25, 26).

[17] E. M. Clarke and J. M. Wing. "Formal Methods: State of the Art and Future Directions". In: *ACM Comput. Surv.* 28.4 (1996), pp. 626–643. DOI: 10.1145/242223.242257. URL: https://doi.org/10.1145/242223.242257 (cit. on p. 15).

[18] S. Conchon, A. Korneva, and F. Zaidi. "Verifying Smart Contracts with Cubicle". In: *Formal Methods. FM 2019 International Workshops: Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part I*. Porto, Portugal: Springer-Verlag, 2019, pp. 312–324. ISBN: 978-3-030-54993-0. DOI: 10.1007/978-3-030-54994-7_23. URL: https://doi.org/10.1007/978-3-030-54994-7_23 (cit. on p. 49).

[19] S. Conchon, A. Korneva, and F. Zaidi. "Verifying Smart Contracts with Cubicle". In: *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part I*. Ed. by E. Sekerinski et al. Vol. 12232. Lecture Notes in Computer Science. Springer, 2019, pp. 312–324. DOI: 10.1007/978-3-030-54994-7_23. URL: https://doi.org/10.1007/978-3-030-54994-7%5C_23 (cit. on p. 3).

[20] R. Corte. *Smart Types For Smart Contracts*. 2022. URL: https://github.com/RonaldoCorte/Smart-Types-for-SmartContracts (cit. on pp. 3, 66).

[21] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems - concepts and designs (3. ed.)* International computer science series. Addison-Wesley-Longman, 2002. ISBN: 978-0-201-61918-8 (cit. on p. 4).

[22] S. Crafa, M. D. Pirro, and E. Zucca. "Is Solidity Solid Enough?" In: *Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*. Ed. by A. Bracciali et al. Vol. 11599. Lecture Notes in Computer Science. Springer, 2019, pp. 138–153. DOI: 10.1007/978-3-030-43725-1_11. URL: https://doi.org/10.1007/978-3-030-43725-1%5C_11 (cit. on p. 21).

[23] S. Crafa, M. D. Pirro, and E. Zucca. "Is Solidity Solid Enough?" In: *Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*. Ed. by A. Bracciali et al. Vol. 11599. Lecture Notes in Computer Science. Springer, 2019, pp. 138–153. DOI: 10.1007/978-3-030-43725-1_11. URL: https://doi.org/10.1007/978-3-030-43725-1%5C_11 (cit. on p. 31).

[24] D. L. Dill et al. "Fast and Reliable Formal Verification of Smart Contracts with the Move Prover". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by D. Fisman and G. Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 183–200. DOI: 10.1007/978-3-030-99524-9_10. URL: https://doi.org/10.1007/978-3-030-99524-9%5C_10 (cit. on pp. 26, 27).

[25] S. Gilbert and N. A. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2 (2002), pp. 51–59. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601 (cit. on pp. 4, 5).

[26] D. Hellwig, G. Karlic, and A. Huchzermeier. *Build Your Own Blockchain*. Management for Professionals. Springer International Publishing, 2021. ISBN: 978-3-030-40141-2 (cit. on pp. iv, 8, 9, 12).

[27] M. João. *TypeState-editor*. URL: https://typestate-editor.github.io/ (cit. on p. 50).

[28] L. Lamport, R. E. Shostak, and M. C. Pease. "The Byzantine generals problem". In: *Concurrency: the Works of Leslie Lamport*. Ed. by D. Malkhi. ACM, 2019, pp. 203–226. DOI: 10.1145/3335772.3335936. URL: https://doi.org/10.1145/3335772.3335936 (cit. on p. 5).

[29] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[30] R. Mitra. *Tezos VS Ethererum: [The Ultimate Comparison Guide]*. Aug. 2019. URL: https://blockgeeks.com/guides/tezos-vs-ethererum-the-ultimate-comparison-guide/ (cit. on p. 30).

[31] B. Moreira. *Formalization of Smart Contracts Languages*. 2021 (cit. on pp. 32, 33, 35).

[32] M. di Pirro. *How solid is Solidity? An in-dept study of solidity's type safety*. 2018 (cit. on pp. 30, 31, 33).

[33] Simplilearn. *What is Solidity Programming, its Data Types, Smart Contracts, and EVM in Ethereum?* June 2022. URL: https://www.simplilearn.com/tutorials/blockchain-tutorial/what-is-solidity-programming#what_is_solidity_programming (cit. on p. 30).

[34] K. Song et al. "ESBMC-Solidity: An SMT-Based Model Checker for Solidity Smart Contracts". In: *CoRR* abs/2111.13117 (2021). arXiv: 2111.13117. URL: https://arxiv.org/abs/2111.13117 (cit. on pp. 24, 25).

[35] *The ESBMC model checker*. URL: https://github.com/esbmc/esbmc (cit. on p. 23).

[36] *The Z3 Theorem Prover*. 2008. URL: https://github.com/Z3Prover/z3 (cit. on p. 27).

[37] R. Zhang, R. Xue, and L. Liu. "Security and Privacy on Blockchain". In: *ACM Comput. Surv.* 52.3 (2019), 51:1–51:34. DOI: 10.1145/3316481. URL: https://doi.org/10.1145/3316481 (cit. on pp. 5, 8–10).

[38] J. Zhong et al. "The Move Prover". In: July 2020, pp. 137–150. ISBN: 978-3-030-53287-1. DOI: 10.1007/978-3-030-53288-8_7 (cit. on p. 27).

[39] J. E. Zhong et al. "The Move Prover". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*. Ed. by S. K. Lahiri and C. Wang. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 137–150. DOI: 10.1007/978-3-030-53288-8_7. URL: https://doi.org/10.1007/978-3-030-53288-8%5C_7 (cit. on p. 28).

A

Appendix: Bank's Program Typecheck Output

Listing A.1: Bank's program typecheck output

```
1  contract Bank {(mapping(address => uint) : balances)} unit constuctor ((mapping(address =>
       ↪ uint) : balances)) ; unit deposit () ; uint getBalance () ; unit transfer ((address :
       ↪ to)(uint : amount)) ; unit withdraw ((uint : amount))
2  , (Bank : Bank); (aBank : address); (aC1 : address); (aC2 : address); |- Bank Bank = new Bank.
       ↪ value(500)([(aC1 , 550)]) ;
3
4  Bank.constuctor.value(500).sender(address(Bank))([(aC1 , 500)]) ; Bank.deposit.value(100).
       ↪ sender(address(Bank))() ; Bank.getBalance.value(0).sender(address(Bank))() ; Bank.
       ↪ transfer.value(100).sender(address(Bank))(aC1, 100) : (DECL)
5     Bank
6        Bank |- new Bank.value(500)([(aC1 , 550)]) : Bank
7     (NEW)
8           Bank |- 500 : uint
9     (NAT)
10          Bank |- [(aC1 , 550)] : mapping(address => uint)
11    (MAPPING)
12           Bank |- aC1 : address
13    (ADDRESS)
14           Bank |- 550 : uint
15    (NAT)
16
17
18 Bank(Bank : Bank); |- Bank.constuctor.value(500).sender(address(Bank))([(aC1 , 500)]) : (
       ↪ CALLTOPLEVEL)
19          Bank(Bank : Bank); |- address(Bank) : address
20    (ADDR)
21            Bank(Bank : Bank); |- Bank : Bank
22    (REF)
23          Bank(Bank : Bank); |- Bank.constuctor.value(500)([(aC1 , 500)]) : (CALL)
24            Bank(Bank : Bank); |- Bank : Bank
25    (REF)
26            Bank(Bank : Bank); |- 500 : uint
27    (NAT)
28            Bank(Bank : Bank); |- [(aC1 , 500)] : mapping(address => uint)
```

```
    (MAPPING)
              Bank(Bank : Bank); |- aC1 : address
    (ADDRESS)
              Bank(Bank : Bank); |- 500 : uint
    (NAT)


Bank(Bank : Bank); |- Bank.deposit.value(100).sender(address(Bank))() : (CALLTOPLEVEL)
            Bank(Bank : Bank); |- address(Bank) : address
    (ADDR)
              Bank(Bank : Bank); |- Bank : Bank
    (REF)
            Bank(Bank : Bank); |- Bank.deposit.value(100)() : (CALL)
              Bank(Bank : Bank); |- Bank : Bank
    (REF)
              Bank(Bank : Bank); |- 100 : uint
    (NAT)


Bank(Bank : Bank); |- Bank.getBalance.value(0).sender(address(Bank))() : (CALLTOPLEVEL)
              Bank(Bank : Bank); |- address(Bank) : address
    (ADDR)
                Bank(Bank : Bank); |- Bank : Bank
    (REF)
              Bank(Bank : Bank); |- Bank.getBalance.value(0)() : (CALL)
                Bank(Bank : Bank); |- Bank : Bank
    (REF)
                Bank(Bank : Bank); |- 0 : uint
    (NAT)


Bank(Bank : Bank); |- Bank.transfer.value(100).sender(address(Bank))(aC1, 100) : (CALLTOPLEVEL
    ↪ )
              Bank(Bank : Bank); |- address(Bank) : address
    (ADDR)
                Bank(Bank : Bank); |- Bank : Bank
    (REF)
              Bank(Bank : Bank); |- Bank.transfer.value(100)(aC1, 100) : (CALL)
                Bank(Bank : Bank); |- Bank : Bank
    (REF)
                Bank(Bank : Bank); |- 100 : uint
    (NAT)
                Bank(Bank : Bank); |- aC1 : address
    (ADDRESS)
                Bank(Bank : Bank); |- 100 : uint
    (NAT)

Success
```

B

Appendix: MiniFS's Grammar

Listing B.1: Mini-FS Grammar in ocaml

```
type typ =
| TBool
| TNat
| TUnit
| TAddress
| TContract of string
| TMapping of typ * typ
| TFun of typ list * typ
| TString
| TState

type values =
| VTrue
| VFalse
| VCons of int
| VAddress of string
| VContract of string
| VUnit
| VMapping of (values * values) list
| VString of string
| VState of string

type exp =
  | TmValue of values
  | TmVar of string
  | TmGetParam of string
  | TmGetState of string
  | TmBalance of exp
  | TmMappSel of exp * exp
  | TmStateSel of exp * string
  | TmPlus of exp * exp
  | TmMinus of exp * exp
  | TmMult of term * term
  | TmDiv of term * term
```

```
  | Equals of exp * exp
  | Less of exp * exp
  | Greater of exp * exp
  | Not of exp
  | And of exp * exp
  | Or of exp * exp
  | LessOrEquals of exp * exp
  | GreaterOrEquals of exp * exp
  | TmAddress of exp

type comm =
  | TmStateAss of exp * string * exp
  | TmTransfer of exp * exp
  | TmNew of string * exp * exp list
  | TmDecl of typ * string * exp * exp
  | TmAssign of string * exp
  | TmMappAss of exp * exp * exp
  | TmFun of exp * string
  | TmIf of exp * comm * comm
  | TmRevert
  | TmCall of exp * string * exp * exp list
  | TmCallTop of exp * string * exp * exp * exp list
  | TmSeq of comm * comm
  | TmReturn of exp

type params = typ * string
type construtor_ = params list * term
type func = {
    requires : term;
    return_type: typ;
    func_name: string;
    params: params list;
    body: term;
  }

type contract_tc = {
name : string;
init : (params list) * term;
invariant : (params list) * term;
behavioral_types: term list;
vars: (term * typ) list;
cons : construtor_;
funcs: func list;
}
```

C

Appendix: Bank MiniFS and its Typecheck

Listing C.1: Bank.MiniFS

```
1
2  let ctBank =
3    (CT.add "Bank" ([((TNat), "Msg_value"); (TMapping(TAddress, TNat), "Balances");
4                     (TMapping(TAddress, TNat), "balances_param");
5                     ((TNat), "amount"); ((TAddress), "to");
6                     ((TNat), "W_amount");
7                     ],
8                  [(TUnit, "Bank_constructor",[((TNat), "msg_value"); ((TAddress), "sender"); ((
                       ↪ TMapping(TAddress, TNat)), "balances_param");
9                  ((TAddress), "to");
10                 ((TNat), "amount");((TNat), "W_amount");]);
11                 (TUnit, "transfer",[((TNat), "amount");((TAddress), "to")]);
12                 (TUnit, "withdraw",[((TNat), "W_amount")]);
13                 (TNat, "deposit", []);
14                 (TNat, "get_balance", []);
15                 ]) CT.empty);;
16
17 let sender : term = TmGetParam("sender")
18 let msg_value : term = TmVar("Msg_value")
19 let balances = TmVar("Balances")
20
21 let param_balances : params = (TMapping(TAddress, TNat), "balances_param")
22 let param_balances_val : term = (TmGetParam("balances_param"))
23 let Bank_constructor : construtor_ = ([param_balances],TmSeq(TmAssign("Balances",
       ↪ param_balances_val), TmReturn(TmValue(VUnit)) ))
24
25 let deposit : func = { requires = TmValue(VTrue); return_type = TUnit; func_name = "deposit";
       ↪ params = []; body = TmSeq( TmMappAss(balances, sender, TmPlus( TmMappSel(balances,
       ↪ sender), msg_value)), TmReturn(TmValue(VUnit))) }
26
27 let get_balance : func = { requires = TmValue(VTrue); return_type = TNat; func_name = "
       ↪ get_balance"; params = []; body = TmReturn( TmMappSel(balances, sender) ) }
28
29 let param_to : params = (TAddress, "to")
```

```
let param_to_val : term = TmGetParam("to")
let param_amount :params = (TNat, "amount")
let param_amount_val : term = TmGetParam("amount")
let transfer : func = {requires = TmValue(VTrue); return_type = TUnit; func_name = "transfer";
    ↪  params = [param_to; param_amount];
body = TmSeq( TmIf( GreaterOrEquals(TmMappSel(balances, sender), param_amount_val),
TmSeq( TmMappAss(balances, sender, TmMinus( TmMappSel(balances, sender),param_amount_val)),
TmSeq(TmMappAss(balances, param_to_val, TmPlus( TmMappSel(balances, param_to_val),
    ↪ param_amount_val)),TmReturn(TmValue(VUnit)))),
TmReturn(TmValue(VUnit))), TmReturn(TmValue(VUnit))) }

let param_withdraw_amount : params = (TNat, "W_amount")
let param_withdraw_amount_val : term = TmGetParam("W_amount")
let withdraw : func = {requires = TmValue(VTrue); return_type = TUnit; func_name = "withdraw";
    ↪  params = [param_withdraw_amount]; body =TmSeq( TmIf( GreaterOrEquals(TmMappSel(
    ↪ balances, sender), param_withdraw_amount_val),
TmSeq(TmMappAss(balances, sender, TmMinus( TmMappSel(balances, sender),
    ↪ param_withdraw_amount_val)),
     TmTransfer( sender,param_withdraw_amount_val ) ),
     TmReturn(TmValue(VUnit))), TmReturn(TmValue(VUnit))) }

let contractBank : contract_tc = {
  name = "Bank";
  init = ( [((TAddress,"param_init"))] ,GreaterOrEquals(TmMappSel(TmVar("Balances"),
      ↪ TmGetParam("param_init") ), TmValue(VCons(0))));
  invariant = ([(TAddress, "param_unsafe")],Less(TmMappSel(TmVar("Balances"), TmGetParam("
      ↪ param_unsafe") ), TmValue(VCons(0))));
  behavioral_types = [];
  vars = [(TmVar("Msg_value"), TNat);(TmVar("Balances"), TMapping(TAddress,TNat))];
  cons = Bank_constructor;
  funcs = [deposit; get_balance; transfer; withdraw];
};;
```

Listing C.2: Bank typecheck

```

 contract Bank {(uint : Msg_value)(mapping(address => uint) : Balances)(mapping(address =>
     ↪ uint) : balances_param)(uint : amount)(address : to)(uint : W_amount)} unit
     ↪ Bank_constructor ((uint : msg_value)(address : sender)(mapping(address => uint) :
     ↪ balances_param)(address : to)(uint : amount)(uint : W_amount)) ; unit transfer ((uint
     ↪ : amount)(address : to)) ; unit withdraw ((uint : W_amount)) ; uint deposit () ; uint
     ↪ get_balance ()
 , (Balances : mapping(address => uint)); (Bank : Bank); (Msg_value : uint); (W_amount : uint);
     ↪  (aBank : address); (amount : uint); (balances_param : mapping(address => uint)); (
     ↪ param_init : address); (param_unsafe : address); (sender : address); (to : address);
     ↪ |- Balances[param param_init] >= 0 : bool
(Equals)
  Bank |- Balances[param param_init] > 0 : (Cond )
    Bank |- 0 : uint
(NAT)
```

```
uint
    Bank |- Balances[param param_init] : uint
(MAPPSEL)
     Bank |- Balances : (Var)
mapping(address => uint)
     Bank |- param param_init : address
(PARAM)
Success
Bank |- Balances[param param_unsafe] < 0 : bool
(Equals)
  Bank |- Balances[param param_unsafe] > 0 : (Cond )
    Bank |- 0 : uint
(NAT)
uint
    Bank |- Balances[param param_unsafe] : uint
(MAPPSEL)
     Bank |- Balances : (Var)
mapping(address => uint)
     Bank |- param param_unsafe : address
(PARAM)
Success
Bank |- Balances = param balances_param ; return u : unit
(SEQ)
  Bank |- Balances = param balances_param : (ASS)
    Bank |- param balances_param : (Param)
mapping(address => uint)
mapping(address => uint)
    Bank |- Balances : (Var)
mapping(address => uint)
  Bank |- return u : unit
(RETURN)
    Bank |- u : unit
(UNIT)
Success
Bank |- true : bool
(TRUE)
Success
Bank |- Balances[param sender -> Balances[param sender] + Msg_value] ; return u : unit
(SEQ)
  Bank |- Balances[param sender -> Balances[param sender] + Msg_value] : (MAPPASS)
    Bank |- param sender : (Param)
address
    Bank |- Balances[param sender] + Msg_value : (Arit)
      Bank |- Balances[param sender] : uint
(MAPPSEL)
       Bank |- Balances : (Var)
mapping(address => uint)
       Bank |- param sender : address
(PARAM)
uint
```

```
     Bank |- Msg_value : uint
(VAR)
mapping(address => uint)
    Bank |- Balances : mapping(address => uint)
(VAR)
  Bank |- return u : unit
(RETURN)
    Bank |- u : unit
(UNIT)
Success
Bank |- true : bool
(TRUE)
Success
Bank |- return Balances[param sender] : uint
(RETURN)
  Bank |- Balances[param sender] : uint
(MAPPSEL)
    Bank |- Balances : (Var)
mapping(address => uint)
    Bank |- param sender : address
(PARAM)
Success
Bank |- true : bool
(TRUE)
Success
Bank |- if Balances[param sender] >= param amount then Balances[param sender -> Balances[param
    ↪  sender] - param amount] ; Balances[param to -> Balances[param to] + param amount] ;
    ↪ return u else return u ; return u : unit
(SEQ)
  Bank |- if Balances[param sender] >= param amount then Balances[param sender -> Balances[
      ↪ param sender] - param amount] ; Balances[param to -> Balances[param to] + param
      ↪ amount] ; return u else return u : (IF)
    Bank |- Balances[param sender] >= param amount : bool
(Equals)
     Bank |- Balances[param sender] > param amount : (Cond )
       Bank |- param amount : uint
(PARAM)
uint
       Bank |- Balances[param sender] : uint
(MAPPSEL)
         Bank |- Balances : (Var)
mapping(address => uint)
         Bank |- param sender : address
(PARAM)
    Bank |- Balances[param sender -> Balances[param sender] - param amount] ; Balances[param to
        ↪  -> Balances[param to] + param amount] ; return u : (SEQ)
     Bank |- Balances[param sender -> Balances[param sender] - param amount] : (MAPPASS)
       Bank |- param sender : (Param)
address
       Bank |- Balances[param sender] - param amount : (Arit)
```

```
         Bank |- Balances[param sender] : uint
(MAPPSEL)
           Bank |- Balances : (Var)
mapping(address => uint)
           Bank |- param sender : address
(PARAM)
uint
         Bank |- param amount : uint
(PARAM)
mapping(address => uint)
       Bank |- Balances : mapping(address => uint)
(VAR)
     Bank |- Balances[param to -> Balances[param to] + param amount] ; return u : (SEQ)
       Bank |- Balances[param to -> Balances[param to] + param amount] : (MAPPASS)
         Bank |- param to : (Param)
address
         Bank |- Balances[param to] + param amount : (Arit)
           Bank |- Balances[param to] : uint
(MAPPSEL)
             Bank |- Balances : (Var)
mapping(address => uint)
             Bank |- param to : address
(PARAM)
uint
           Bank |- param amount : uint
(PARAM)
mapping(address => uint)
         Bank |- Balances : mapping(address => uint)
(VAR)
       Bank |- return u : (RETURN)
         Bank |- u : (UNIT)
unit
unit
    Bank |- return u : (RETURN)
      Bank |- u : (UNIT)
unit
unit
unit
  Bank |- return u : unit
(RETURN)
    Bank |- u : unit
(UNIT)
Success
Bank |- true : bool
(TRUE)
Success
Bank |- if Balances[param sender] >= param W_amount then Balances[param sender -> Balances[
    ↪ param sender] - param W_amount] ; param sender.transfer(param W_amount) else return u ;
    ↪  return u : unit
(SEQ)
```

```
 Bank |- if Balances[param sender] >= param W_amount then Balances[param sender -> Balances[
      ↪ param sender] - param W_amount] ; param sender.transfer(param W_amount) else return u
      ↪  : (IF)
   Bank |- Balances[param sender] >= param W_amount : bool
(Equals)
     Bank |- Balances[param sender] > param W_amount : (Cond )
       Bank |- param W_amount : uint
(PARAM)
uint
       Bank |- Balances[param sender] : uint
(MAPPSEL)
         Bank |- Balances : (Var)
mapping(address => uint)
         Bank |- param sender : address
(PARAM)
   Bank |- Balances[param sender -> Balances[param sender] - param W_amount] ; param sender.
        ↪ transfer(param W_amount) : (SEQ)
     Bank |- Balances[param sender -> Balances[param sender] - param W_amount] : (MAPPASS)
       Bank |- param sender : (Param)
address
       Bank |- Balances[param sender] - param W_amount : (Arit)
         Bank |- Balances[param sender] : uint
(MAPPSEL)
           Bank |- Balances : (Var)
mapping(address => uint)
           Bank |- param sender : address
(PARAM)
uint
         Bank |- param W_amount : uint
(PARAM)
mapping(address => uint)
       Bank |- Balances : mapping(address => uint)
(VAR)
     Bank |- param sender.transfer(param W_amount) : (TRANSFER)
       Bank |- param sender : address
(PARAM)
       Bank |- param W_amount : uint
(PARAM)
unit
   Bank |- return u : (RETURN)
     Bank |- u : (UNIT)
unit
unit
unit
 Bank |- return u : unit
(RETURN)
   Bank |- u : unit
(UNIT)
Success
```

Listing C.3: Bank-Ocaml.MiniFS

```
contractBank : contract_tc =
  {name = "Bank";
   init =
    ([(TAddress, "param_init")],
     GreaterOrEquals (TmMappSel (TmVar "Balances", TmGetParam "param_init"),
      TmValue (VCons 0)));
   invariant =
    ([(TAddress, "param_unsafe")],
     Less (TmMappSel (TmVar "Balances", TmGetParam "param_unsafe"),
      TmValue (VCons 0)));
   behavioral_types = [];
   vars =
    [(TmVar "Msg_value", TNat);
     (TmVar "Balances", TMapping (TAddress, TNat))];
   cons =
    ([(TMapping (TAddress, TNat), "balances_param")],
     TmSeq (TmAssign ("Balances", TmGetParam "balances_param"),
      TmReturn (TmValue VUnit)));
   funcs =
    [{requires = TmValue VTrue; return_type = TUnit; func_name = "deposit";
      params = [];
      body =
       TmSeq
        (TmMappAss (TmVar "Balances", TmGetParam "sender",
          TmPlus (TmMappSel (TmVar "Balances", TmGetParam "sender"),
           TmVar "Msg_value")),
        TmReturn (TmValue VUnit))};
     {requires = TmValue VTrue; return_type = TNat;
      func_name = "get_balance"; params = [];
      body = TmReturn (TmMappSel (TmVar "Balances", TmGetParam "sender"))};
     {requires = TmValue VTrue; return_type = TUnit; func_name = "transfer";
      params = [(TAddress, "to"); (TNat, "amount")];
      body =
       TmSeq
        (TmIf
          (GreaterOrEquals
            (TmMappSel (TmVar "Balances", TmGetParam "sender"),
            TmGetParam "amount"),
          TmSeq
           (TmMappAss (TmVar "Balances", TmGetParam "sender",
             TmMinus (TmMappSel (TmVar "Balances", TmGetParam "sender"),
              TmGetParam "amount")),
           TmSeq
            (TmMappAss (TmVar "Balances", TmGetParam "to",
              TmPlus (TmMappSel (TmVar "Balances", TmGetParam "to"),
               TmGetParam "amount")),
            TmReturn (TmValue VUnit))),
          TmReturn (TmValue VUnit)),
        TmReturn (TmValue VUnit))};
```

```
50      {requires = TmValue VTrue; return_type = TUnit; func_name = "withdraw";
51       params = [(TNat, "W_amount")];
52       body =
53        TmSeq
54         (TmIf
55           (GreaterOrEquals
56             (TmMappSel (TmVar "Balances", TmGetParam "sender"),
57              TmGetParam "W_amount"),
58            TmSeq
59             (TmMappAss (TmVar "Balances", TmGetParam "sender",
60               TmMinus (TmMappSel (TmVar "Balances", TmGetParam "sender"),
61                TmGetParam "W_amount")),
62              TmTransfer (TmGetParam "sender", TmGetParam "W_amount")),
63            TmReturn (TmValue VUnit)),
64          TmReturn (TmValue VUnit))}]}
```

D

Appendix: Marketplace.MiniFS and Marketplace.Sol

Listing D.1: Marketplace.minifs

```
contract_tc =
  {name = "Marketplace";
   init =
    ([],
     And (Equals (TmVar "CurrentState", TmGetState "ItemAvailable"),
      And (Greater (TmVar "AskingPrice", TmValue (VCons 0)),
       Greater (TmVar "OfferPrice", TmVar "AskingPrice"))));
   invariant = ([], Greater (TmVar "OfferPrice", TmVar "AskingPrice"));
   behavioral_types =
    [TmGetState "ItemAvailable"; TmGetState "OfferPlaced";
     TmGetState "Accept"];
   vars =
    [(TmVar "CurrentState", TState); (TmVar "InstanceOwner", TAddress);
     (TmVar "Description", TString); (TmVar "AskingPrice", TNat);
     (TmVar "InstanceBuyer", TAddress); (TmVar "OfferPrice", TNat)];
   cons =
    ([(TString, "param_description"); (TNat, "param_price")],
     TmSeq
      (TmSeq (TmAssign ("CurrentState", TmGetState "ItemAvailable"),
        TmSeq (TmAssign ("InstanceOwner", TmGetParam "sender"),
         TmSeq (TmAssign ("AskingPrice", TmGetParam "param_price"),
          TmAssign ("Description", TmGetParam "param_description")))),
      TmReturn (TmValue VUnit)));
   funcs =
    [{requires =
       And (Equals (TmGetParam "sender", TmVar "InstanceBuyer"),
        And (Equals (TmGetState "ItemAvailable", TmVar "CurrentState"),
         Greater (TmGetParam "param_offer", TmVar "AskingPrice")));
      return_type = TUnit; func_name = "make_offer";
      params = [(TNat, "param_offer")];
      body =
       TmSeq
        (TmSeq
          (TmIf (Equals (TmGetParam "param_offer", TmValue (VCons 0)),
```

```
          TmRevert, TmReturn (TmValue VUnit)),
        TmSeq
         (TmIf (Equals (TmGetParam "sender", TmVar "InstanceOwner"),
           TmRevert, TmReturn (TmValue VUnit)),
         TmSeq (TmAssign ("InstanceBuyer", TmGetParam "sender"),
          TmSeq (TmAssign ("OfferPrice", TmGetParam "param_offer"),
           TmAssign ("CurrentState", TmGetState "OfferPlaced")))),
      TmReturn (TmValue VUnit))};
    {requires =
      And (Equals (TmGetParam "sender", TmVar "InstanceOwner"),
       And (Equals (TmGetState "OfferPlaced", TmVar "CurrentState"),
        Not (Equals (TmGetParam "param_offer", TmValue (VCons 0)))));
     return_type = TUnit; func_name = "reject";
     params = [(TNat, "param_offer_r")];
     body =
      TmSeq (TmAssign ("InstanceBuyer", TmValue (VAddress "sender")),
       TmSeq (TmAssign ("CurrentState", TmGetState "ItemAvailable"),
        TmReturn (TmValue VUnit)))};
    {requires =
      And (Equals (TmGetState "OfferPlaced", TmVar "CurrentState"),
       Equals (TmGetParam "sender", TmVar "InstanceOwner"));
     return_type = TUnit; func_name = "accept";
     params = [(TNat, "param_offer_a")];
     body =
      TmSeq
       (TmIf (Equals (TmGetParam "param_offer", TmValue (VCons 0)),
         TmRevert, TmReturn (TmValue VUnit)),
       TmSeq (TmAssign ("CurrentState", TmGetState "Accept"),
        TmReturn (TmValue VUnit)))}]}
```

Listing D.2: Marketplace.sol

```
pragma solidity >=0.4.25 <0.6.0;

contract SimpleMarketplace
{
    enum StateType {
      ItemAvailable,
      OfferPlaced,
      Accepted
    }

    address public InstanceOwner;
    string public Description;
    int public AskingPrice;
    StateType public State;

    address public InstanceBuyer;
    int public OfferPrice;

```

```
19      constructor(string memory description, int price) public
20      {
21          InstanceOwner = msg.sender;
22          AskingPrice = price;
23          Description = description;
24          State = StateType.ItemAvailable;
25      }
26
27      function MakeOffer(int offerPrice) public
28      {
29          if (offerPrice == 0)
30          {
31              revert();
32          }
33
34          if (State != StateType.ItemAvailable)
35          {
36              revert();
37          }
38
39          if (InstanceOwner == msg.sender)
40          {
41              revert();
42          }
43
44          InstanceBuyer = msg.sender;
45          OfferPrice = offerPrice;
46          State = StateType.OfferPlaced;
47      }
48
49      function Reject() public
50      {
51          if ( State != StateType.OfferPlaced )
52          {
53              revert();
54          }
55
56          if (InstanceOwner != msg.sender)
57          {
58              revert();
59          }
60
61          InstanceBuyer = 0x0000000000000000000000000000000000000000;
62          State = StateType.ItemAvailable;
63      }
64
65      function AcceptOffer() public
66      {
67          if ( msg.sender != InstanceOwner )
68          {
```

```
69              revert();
70          }
71
72          State = StateType.Accepted;
73      }
74  }
```

E

Appendix: Telemetry.MiniFS and Telemetry.Sol

Listing E.1: Telemetry.minifs

```
val contractTelemetry : contract_tc =
  {name = "Telemetry";
   init =
    ([],
     And (Greater (TmVar "Max_Temperature", TmVar "Min_Temperature"),
      And (Greater (TmVar "Min_Temperature", TmValue (VCons 0)),
       And (Equals (TmVar "ComplianceStatus", TmValue VTrue),
        Equals (TmVar "CurrentState", TmGetState "Created")))));
   invariant =
    ([], Greater (TmVar "Max_Temperature", TmVar "Min_Temperature"));
   behavioral_types =
    [TmGetState "Created"; TmGetState "InTransit";
     TmGetState "OutOfCompliance"; TmGetState "Completed"];
   vars =
    [(TmVar "Owner", TAddress); (TmVar "CurrentState", TState);
     (TmVar "CounterParty", TAddress); (TmVar "Device", TAddress);
     (TmVar "ComplianceStatus", TBool); (TmVar "Max_Temperature", TNat);
     (TmVar "Min_Temperature", TNat)];
   cons =
    ([(TAddress, "param_device"); (TAddress, "param_max_temp");
      (TAddress, "param_min_temp")],
     TmSeq
      (TmSeq (TmAssign ("ComplianceStatus", TmValue VTrue),
        TmSeq (TmAssign ("CounterParty", TmGetParam "sender"),
         TmSeq (TmAssign ("Owner", TmGetParam "sender"),
          TmSeq (TmAssign ("Device", TmGetParam "param_device"),
           TmSeq (TmAssign ("Max_Temperature", TmGetParam "param_max_temp"),
            TmAssign ("Min_Temperature", TmGetParam "param_min_temp")))))),
      TmReturn (TmValue VUnit)));
   funcs =
    [{requires =
       And (Equals (TmGetParam "sender", TmVar "Device"),
        Or (Equals (TmVar "CurrentState", TmGetState "Created"),
         Equals (TmVar "CurrentState", TmGetState "InTransit")));
```

```
35      return_type = TUnit; func_name = "ingest_telemetry";
36      params = [(TNat, "param_temp")];
37      body =
38       TmSeq
39        (TmSeq
40          (TmIf
41            (Or (Greater (TmGetParam "param_temp", TmVar "Max_Temperature"),
42              Less (TmGetParam "param_temp", TmVar "Min_Temperature")),
43            TmAssign ("ComplianceStatus", TmValue VFalse),
44            TmAssign ("ComplianceStatus", TmValue VTrue)),
45          TmSeq
46           (TmIf (Not (Equals (TmGetParam "sender", TmVar "Device")),
47             TmRevert, TmReturn (TmValue VUnit)),
48           TmIf
49            (Or (Greater (TmGetParam "param_temp", TmVar "Max_Temperature"),
50              Less (TmGetParam "param_temp", TmVar "Min_Temperature")),
51            TmAssign ("ComplianceStatus", TmValue VFalse),
52            TmReturn (TmValue VUnit)))),
53        TmReturn (TmValue VUnit))};
54     {requires =
55       And (Equals (TmGetParam "sender", TmVar "CounterParty"),
56        Or (Equals (TmVar "CurrentState", TmGetState "Created"),
57         Equals (TmVar "CurrentState", TmGetState "InTransit")));
58      return_type = TUnit; func_name = "transfer_responsability";
59      params = [(TAddress, "param_new_counter_party")];
60      body =
61       TmSeq
62        (TmIf (Equals (TmVar "CurrentState", TmGetState "Created"),
63          TmAssign ("CurrentState", TmGetState "InTransit"),
64          TmAssign ("CurrentState", TmGetState "Created")),
65        TmSeq
66         (TmSeq
67           (TmIf
68             (And
69               (Not
70                 (Equals (TmGetParam "sender",
71                   TmGetParam "param_new_counter_party")),
72                Not (Equals (TmGetParam "sender", TmVar "CounterParty"))),
73              TmRevert, TmReturn (TmValue VUnit)),
74            TmSeq
75             (TmIf
76               (Equals (TmGetParam "param_new_counter_party", TmVar "Device"),
77               TmRevert, TmReturn (TmValue VUnit)),
78             TmAssign ("CounterParty",
79              TmGetParam
80               "param_new_counter_party")))),
81          TmReturn (TmValue VUnit)))};
82     {requires = Equals (TmGetParam "sender", TmVar "Owner");
83      return_type = TUnit; func_name = "complete"; params = [];
84      body =
```

```
      TmSeq
       (TmAssign ("CurrentState"),
         TmGetState "Complete"),
       TmReturn (TmValue VUnit))}]}
```

Listing E.2: Telemetry.sol

```
pragma solidity >=0.4.25 <0.6.0;

contract RefrigeratedTransportation
{
    //Set of States
    enum StateType { Created, InTransit, Completed, OutOfCompliance}
    enum SensorType { None, Humidity, Temperature }

    //List of properties
    StateType public State;
    address public Owner;
    address public InitiatingCounterparty;
    address public Counterparty;
    address public PreviousCounterparty;
    address public Device;
    address public SupplyChainOwner;
    address public SupplyChainObserver;
    int public MinHumidity;
    int public MaxHumidity;
    int public MinTemperature;
    int public MaxTemperature;
    SensorType public ComplianceSensorType;
    int public ComplianceSensorReading;
    bool public ComplianceStatus;
    string public ComplianceDetail;
    int public LastSensorUpdateTimestamp;

    constructor(address device, address supplyChainOwner, address supplyChainObserver, int
        ↪ minHumidity, int maxHumidity, int minTemperature, int maxTemperature) public
    {
        ComplianceStatus = true;
        ComplianceSensorReading = -1;
        InitiatingCounterparty = msg.sender;
        Owner = InitiatingCounterparty;
        Counterparty = InitiatingCounterparty;
        Device = device;
        SupplyChainOwner = supplyChainOwner;
        SupplyChainObserver = supplyChainObserver;
        MinHumidity = minHumidity;
        MaxHumidity = maxHumidity;
        MinTemperature = minTemperature;
        MaxTemperature = maxTemperature;
        State = StateType.Created;
```

```
43          ComplianceDetail = "N/A";
44      }
45
46      function IngestTelemetry(int humidity, int temperature, int timestamp) public
47      {
48          // Separately check for states and sender
49          // to avoid not checking for state when the sender is the device
50          // because of the logical OR
51          if ( State == StateType.Completed )
52          {
53              revert();
54          }
55
56          if ( State == StateType.OutOfCompliance )
57          {
58              revert();
59          }
60
61          if (Device != msg.sender)
62          {
63              revert();
64          }
65
66          LastSensorUpdateTimestamp = timestamp;
67
68          if (humidity > MaxHumidity || humidity < MinHumidity)
69          {
70              ComplianceSensorType = SensorType.Humidity;
71              ComplianceSensorReading = humidity;
72              ComplianceDetail = "Humidity value out of range.";
73              ComplianceStatus = false;
74          }
75          else if (temperature > MaxTemperature || temperature < MinTemperature)
76          {
77              ComplianceSensorType = SensorType.Temperature;
78              ComplianceSensorReading = temperature;
79              ComplianceDetail = "Temperature value out of range.";
80              ComplianceStatus = false;
81          }
82
83          if (ComplianceStatus == false)
84          {
85              State = StateType.OutOfCompliance;
86          }
87      }
88
89      function TransferResponsibility(address newCounterparty) public
90      {
91          // keep the state checking, message sender, and device checks separate
92          // to not get cloberred by the order of evaluation for logical OR
```

```
93          if ( State == StateType.Completed )
94          {
95              revert();
96          }
97
98          if ( State == StateType.OutOfCompliance )
99          {
100             revert();
101         }
102
103         if ( InitiatingCounterparty != msg.sender && Counterparty != msg.sender )
104         {
105             revert();
106         }
107
108         if ( newCounterparty == Device )
109         {
110             revert();
111         }
112
113         if (State == StateType.Created)
114         {
115             State = StateType.InTransit;
116         }
117
118         PreviousCounterparty = Counterparty;
119         Counterparty = newCounterparty;
120     }
121
122     function Complete() public
123     {
124         // keep the state checking, message sender, and device checks separate
125         // to not get cloberred by the order of evaluation for logical OR
126         if ( State == StateType.Completed )
127         {
128             revert();
129         }
130
131         if ( State == StateType.OutOfCompliance )
132         {
133             revert();
134         }
135
136         if (Owner != msg.sender && SupplyChainOwner != msg.sender)
137         {
138             revert();
139         }
140
141         State = StateType.Completed;
142         PreviousCounterparty = Counterparty;
```

```
143         Counterparty = 0x0000000000000000000000000000000000000000;
144     }
145 }
```

F

Appendix: DigitalLocker.MiniFS and DigitalLocker.Sol

Listing F.1: DigitalLocker.minifs

```
1
2  let sender : term = TmGetParam("sender")
3  let owner : term = TmVar("Owner")
4  let currentState : term = TmVar("CurrentState")
5  let Bank_agent : term = TmVar("BankAgent")
6  let lockerId : term = TmVar("LockerId")
7  let curentAuthorizedUser : term = TmVar("CurrentAuthorizedUser")
8  let exp_Date : term = TmVar("Exp_Date")
9  let image : term = TmVar("Image")
10 let tpRequestor : term = TmVar("TPRequestor")
11 let lockerStatus : term = TmVar("LockerStatus")
12
13
14 let param_Bank : params = (TAddress,"param_Bank_agent")
15 let param_Bank_val : term = TmGetParam("param_Bank_agent")
16 let dl_constructor = ([param_Bank], TmSeq(TmSeq(TmAssign("Owner", TmGetParam("sender")), TmSeq
       ↪ ( TmAssign("CurrentState",TmGetState("Requested")), TmAssign("BankAgent",
       ↪ param_Bank_val) )), TmReturn(TmValue(VUnit))))
17
18 let begin_process_review : func = {requires = And((Equals(owner,sender)),Equals(TmGetState("
       ↪ Requested"),currentState)); return_type = TUnit; func_name = "begin_process_review";
       ↪ params = [];
19 body =TmSeq( TmSeq( TmAssign("BankAgent", sender),TmSeq(TmAssign("LockerStatus",TmValue(
       ↪ VString("Pending"))),TmAssign("CurrentState", TmGetState("DocumentReview"))) ),
       ↪ TmReturn(TmValue(VUnit))) }
20
21
22 let reject_application : func = {requires = And((Equals(Bank_agent,sender)),Equals(TmGetState(
       ↪ "DocumentReview"),currentState)); return_type = TUnit; func_name = "reject_application
       ↪ "; params = [];
23 body =TmSeq( TmSeq( TmAssign("BankAgent", sender),TmSeq(TmAssign("LockerStatus",TmValue(
       ↪ VString("Rejected"))),TmAssign("CurrentState", TmGetState("DocumentReview"))) ),
       ↪ TmReturn(TmValue(VUnit))) }
24
```

```
25
26 let param_image : params = (TString,"param_image")
27 let param_image_val : term = TmGetParam("param_image")
28 let param_locker_id : params = (TString,"param_locker_id")
29 let param_locker_id_val : term = TmGetParam("param_locker_id")
30 let upload_documents : func = {requires = And((Equals(Bank_agent,sender)),Equals(TmGetState("
       ↪ DocumentReview"),currentState)); return_type = TUnit; func_name = "upload_documents";
       ↪ params = [param_image; param_locker_id];
31 body =TmSeq( TmSeq( TmAssign("LockerStatus",TmValue(VString("Approved"))),TmSeq(TmAssign("
       ↪ Image", param_image_val) , TmSeq(TmAssign("LockerId", param_locker_id_val) , TmAssign(
       ↪ "CurrentState", TmGetState("AvailableToShare"))))), TmReturn(TmValue(VUnit))) }
32
33 let param_tp : params = (TAddress,"param_tp")
34 let param_tp_val : term = TmGetParam("param_tp")
35 let param_exp_date : params = (TNat,"param_exp_date")
36 let param_exp_date_val : term = TmGetParam("param_exp_date")
37 let share_with_tp : func = {requires = And((Equals(owner,sender)),Equals(TmGetState("
       ↪ AvailableToShare"),currentState)); return_type = TUnit; func_name = "share_with_tp";
       ↪ params = [param_tp; param_exp_date];
38 body =TmSeq( TmSeq(TmAssign("TPRequestor", param_tp_val),TmSeq( TmAssign("LockerStatus",
       ↪ TmValue(VString("Shared"))),TmSeq(TmAssign("Exp_Date", param_exp_date_val) , TmSeq(
       ↪ TmAssign("CurrentAuthorizedUser", param_tp_val) , TmAssign("CurrentState", TmGetState(
       ↪ "SharingWithTp")))))), TmReturn(TmValue(VUnit))) }
39
40 let accept_s_request : func = {requires = And(Equals(owner,sender), Equals(currentState,
       ↪ TmGetState("SharingRP"))); return_type = TUnit; func_name = "accept_s_request"; params
       ↪  = [];
41 body =TmSeq( TmSeq(TmAssign("CurrentAuthorizedUser", tpRequestor),TmAssign("CurrentState",
       ↪ TmGetState("SharingWithTp"))), TmReturn(TmValue(VUnit))) }
42
43 let reject_s_request : func = {requires = And(Equals(owner,sender),Equals(currentState,
       ↪ TmGetState("SharingRP"))); return_type = TUnit; func_name = "reject_s_request"; params
       ↪  = [];
44 body =TmSeq( TmSeq(TmAssign("LockerStatus", TmValue(VString("Available"))),TmAssign("
       ↪ CurrentState", TmGetState("SharingWithTp"))), TmReturn(TmValue(VUnit))) }
45
46 let request_l_access : func = {requires = And(Not(Equals(owner,sender)),Equals(currentState,
       ↪ TmGetState("SharingRP"))); return_type = TUnit; func_name = "request_l_access"; params
       ↪  = [];
47 body =TmSeq( TmSeq(TmAssign("TPRequestor",sender),TmAssign("CurrentState", TmGetState("
       ↪ SharingRP"))), TmReturn(TmValue(VUnit))) }
48
49 let release_l_access : func = {requires = And(Equals(curentAuthorizedUser,sender), Equals(
       ↪ currentState, TmGetState("SharingWithTp"))); return_type = TUnit; func_name = "
       ↪ release_l_access"; params = [];
50 body =TmSeq( TmSeq(TmAssign("LockerStatus",TmValue(VString("Available"))),TmAssign("
       ↪ CurrentState", TmGetState("AvailableToShare"))), TmReturn(TmValue(VUnit))) }
51
```

```
let revoke_access : func = {requires = And(Equals(owner,sender),Equals(currentState,
    ↪ TmGetState("SharingWithTp"))); return_type = TUnit; func_name = "revoke_access";
    ↪ params = [];
body =TmSeq( TmSeq(TmAssign("LockerStatus",TmValue(VString("Available"))),TmAssign("
    ↪ CurrentState", TmGetState("AvailableToShare"))), TmReturn(TmValue(VUnit))) }


let terminate : func = {requires = And(And(Not(Equals(currentState, TmGetState("DocumentReview
    ↪ "))),Not(Equals(currentState, TmGetState("Requested")))),Equals(owner,sender));
    ↪ return_type = TUnit; func_name = "terminate"; params = [];
body =TmSeq( TmSeq(TmAssign("LockerStatus",TmValue(VString("Available"))),TmAssign("
    ↪ CurrentState", TmGetState("Terminated"))), TmReturn(TmValue(VUnit))) }




  let contractDL : contract_tc = {
  name = "DL";
  init = ([], Equals(TmGetState("Requested"), currentState));
  invariant = ([],TmValue(VFalse));
  behavioral_types = [TmGetState("DocumentReview"); TmGetState("Requested"); TmGetState("
      ↪ AvailableToShare");TmGetState("SharingWithTp");TmGetState("SharingRP");TmGetState("
      ↪ Terminated")];
  vars = [(TmVar("Owner"),TAddress); (TmVar("CurrentState"), TState); (TmVar("BankAgent"),
      ↪ TAddress); (TmVar("LockerId"), TString);
         (TmVar("CurrentAuthorizedUser"), TAddress); (TmVar("Exp_Date"), TNat); (TmVar("Image"
             ↪ ), TString);(TmVar("TPRequestor"), TAddress);(TmVar("LockerStatus"), TString)
             ↪ ];
  cons = dl_constructor;
  funcs = [begin_process_review; reject_application; upload_documents;share_with_tp;
      ↪ accept_s_request;reject_s_request;request_l_access;release_l_access;revoke_access;
      ↪ terminate];
};;
```

Listing F.2: DigitalLocker.sol

```
pragma solidity >=0.4.25 <0.6.0;

contract DigitalLocker
{
    enum StateType { Requested, DocumentReview, AvailableToShare, SharingRequestPending,
        ↪ SharingWithThirdParty, Terminated }
    address public Owner;
    address public BankAgent;
    string public LockerFriendlyName;
    string public LockerIdentifier;
    address public CurrentAuthorizedUser;
    string public ExpirationDate;
    string public Image;
```

```
14      address public ThirdPartyRequestor;
15      string public IntendedPurpose;
16      string public LockerStatus;
17      string public RejectionReason;
18      StateType public State;
19
20      constructor(string memory lockerFriendlyName, address bankAgent) public
21      {
22          Owner = msg.sender;
23          LockerFriendlyName = lockerFriendlyName;
24
25          State = StateType.DocumentReview;
26
27          BankAgent = bankAgent;
28      }
29
30      function BeginReviewProcess() public
31      {
32          /* Need to update, likely with registry to confirm sender is agent
33          Also need to add a function to re-assign the agent.
34          */
35          if (Owner == msg.sender)
36          {
37              revert();
38          }
39          BankAgent = msg.sender;
40
41          LockerStatus = "Pending";
42          State = StateType.DocumentReview;
43      }
44
45      function RejectApplication(string memory rejectionReason) public
46      {
47          if (BankAgent != msg.sender)
48          {
49              revert();
50          }
51
52          RejectionReason = rejectionReason;
53          LockerStatus = "Rejected";
54          State = StateType.DocumentReview;
55      }
56
57      function UploadDocuments(string memory lockerIdentifier, string memory image) public
58      {
59          if (BankAgent != msg.sender)
60          {
61              revert();
62          }
63          LockerStatus = "Approved";
```

```
64          Image = image;
65          LockerIdentifier = lockerIdentifier;
66          State = StateType.AvailableToShare;
67      }
68
69      function ShareWithThirdParty(address thirdPartyRequestor, string memory expirationDate,
            ↪ string memory intendedPurpose) public
70      {
71          if (Owner != msg.sender)
72          {
73              revert();
74          }
75
76          ThirdPartyRequestor = thirdPartyRequestor;
77          CurrentAuthorizedUser = ThirdPartyRequestor;
78
79          LockerStatus = "Shared";
80          IntendedPurpose = intendedPurpose;
81          ExpirationDate = expirationDate;
82          State = StateType.SharingWithThirdParty;
83      }
84
85      function AcceptSharingRequest() public
86      {
87          if (Owner != msg.sender)
88          {
89              revert();
90          }
91
92          CurrentAuthorizedUser = ThirdPartyRequestor;
93          State = StateType.SharingWithThirdParty;
94      }
95
96      function RejectSharingRequest() public
97      {
98          if (Owner != msg.sender)
99          {
100             revert();
101         }
102         LockerStatus = "Available";
103         CurrentAuthorizedUser = 0x0000000000000000000000000000000000000000;
104         State = StateType.AvailableToShare;
105     }
106
107     function RequestLockerAccess(string memory intendedPurpose) public
108     {
109         if (Owner == msg.sender)
110         {
111             revert();
112         }
```

```
113
114         ThirdPartyRequestor = msg.sender;
115         IntendedPurpose = intendedPurpose;
116         State = StateType.SharingRequestPending;
117     }
118
119     function ReleaseLockerAccess() public
120     {
121
122         if (CurrentAuthorizedUser != msg.sender)
123         {
124             revert();
125         }
126         LockerStatus = "Available";
127         ThirdPartyRequestor = 0x0000000000000000000000000000000000000000;
128         CurrentAuthorizedUser = 0x0000000000000000000000000000000000000000;
129         IntendedPurpose = "";
130         State = StateType.AvailableToShare;
131     }
132
133     function RevokeAccessFromThirdParty() public
134     {
135         if (Owner != msg.sender)
136         {
137             revert();
138         }
139         LockerStatus = "Available";
140         CurrentAuthorizedUser = 0x0000000000000000000000000000000000000000;
141         State = StateType.AvailableToShare;
142     }
143
144     function Terminate() public
145     {
146         if (Owner != msg.sender)
147         {
148             revert();
149         }
150         CurrentAuthorizedUser = 0x0000000000000000000000000000000000000000;
151         State = StateType.Terminated;
152     }
153 }
```

G

Appendix: DigitalLocker.cub

Listing G.1: DigitalLocker.cub

```
type state = DocumentReview | Requested | AvailableToShare | SharingWithTp | SharingRP |
    ↪ Terminated
type string = Pending | Rejected | Shared | Approved | Available
var Owner : proc
var CurrentState : state
var BankAgent : proc
var LockerId : string
var CurrentAuthorizedUser : proc
var Exp_Date : int
var Image : string
var TPRequestor : proc
var LockerStatus : string
array GlobalBalances[proc] : int
array Random_param_image[proc] : string
array Random_param_locker_id[proc] : string
array Random_param_exp_date[proc] : int
var Revert : bool
init () { Requested = CurrentState}
unsafe (LockerStatus = Available && CurrentState = SharingWithThirdParty ) { not( true)}

transition begin_process_review(sender)
requires{ Owner = sender && Requested = CurrentState}
{
   BankAgent := sender;
  LockerStatus := Pending;
 CurrentState := DocumentReview
}

transition reject_application(sender)
requires{ BankAgent = sender && DocumentReview = CurrentState}
{
   BankAgent := sender;
  LockerStatus := Rejected;
 CurrentState := DocumentReview
```

```
}

transition upload_documents(sender)
requires{ BankAgent = sender && DocumentReview = CurrentState}
{
   LockerStatus := Approved;
  Image := Random_param_image[sender] ;
  LockerId := Random_param_locker_id[sender] ;
 CurrentState := AvailableToShare
}

transition share_with_tp(sender param_tp)
requires{ Owner = sender && AvailableToShare = CurrentState}
{
   TPRequestor := param_tp;
  LockerStatus := Shared;
  Exp_Date := Random_param_exp_date[sender] ;
  CurrentAuthorizedUser := param_tp;
 CurrentState := SharingWithTp
}

transition accept_s_request(sender)
requires{ Owner = sender && CurrentState = SharingRP}
{
   CurrentAuthorizedUser := TPRequestor;
 CurrentState := SharingWithTp
}

transition reject_s_request(sender)
requires{ Owner = sender && CurrentState = SharingRP}
{
   LockerStatus := Available;
 CurrentState := SharingWithTp
}

transition request_l_access(sender)
requires{ not( Owner = sender) && CurrentState = SharingRP}
{
   TPRequestor := sender;
 CurrentState := SharingRP
}

transition release_l_access(sender)
requires{ CurrentAuthorizedUser = sender && CurrentState = SharingWithTp}
{
   LockerStatus := Available;
 CurrentState := AvailableToShare
}

transition revoke_access(sender)
```

```
84 requires{ Owner = sender && CurrentState = SharingWithTp}
85 {
86    LockerStatus := Available;
87  CurrentState := AvailableToShare
88 }
89
90 transition terminate(sender)
91 requires{ not( CurrentState = DocumentReview) && not( CurrentState = Requested) && Owner =
       ↪ sender}
92 {
93    LockerStatus := Available;
94  CurrentState := Terminated
95 }
```

H

Appendix: Minicub's Grammar

Listing H.1: MiniCub's grammar in ocaml

```
type cub_typ =
 | Int
 | Bool
 | Proc
 | CArr of cub_typ * cub_typ
 | State
 | Unit

 type cub_values =
 | CBool of bool
 | CCons of int
 | CProc of string
 | CUnit
 | CMapping of (values * values) list
 | CState of string

 type exp =
 | C_Address of exp
 | C_Value of cub_values
 | C_Plus of exp * exp
 | C_Minus of exp * exp
 | C_Div of exp * exp
 | C_Mult of exp * exp
 | C_Equals of exp * exp
 | C_Not of exp
 | C_Less of exp * exp
 | C_Greater of exp * exp
 | C_And of exp * exp
 | C_Or of exp * exp
 | C_LessOrEquals of exp * exp
 | C_GreaterOrEquals of exp * exp
 | C_GetI of exp * exp
 | C_Get_Param of string
 | C_GetState of string
```

```
35 | C_GetVar of string
36 | C_TS of string
37 | C_Assign of string * exp
38 | C_AssignArr of exp * exp * exp
39 | C_Seq of exp * exp
40 | C_Case of string * (exp * exp) list
41
42 type behavioral_types = exp list
43 type cub_vars = (exp * cub_typ) list
44 type cubfunc_ = {
45 requires_ : exp;
46 name_ : string;
47 params_ : exp list;
48 body_ : exp
49 }
50 type contract_cub = { (*meter init e unsafe*)
51 name : string;
52 init : (exp list) * exp;
53 unsafe : (exp list) * exp;
54 behavioral_types: behavioral_types;
55 vars: cub_vars;
56 funcs: cubfunc_ list
57 }
```

I

Appendix: Marketplace.MiniCub

Listing I.1: Marketplace.minicub

```
{name = "Marketplace";
  init =
   ([],
    C_And (C_Equals (C_GetVar "CurrentState", C_GetState "ItemAvailable"),
     C_And (C_Greater (C_GetVar "AskingPrice", C_Value (CCons 0)),
      C_Greater (C_GetVar "OfferPrice", C_GetVar "AskingPrice"))));
  unsafe =
   ([], C_Not (C_Greater (C_GetVar "OfferPrice", C_GetVar "AskingPrice")));
  behavioral_types =
   [C_GetState "ItemAvailable"; C_GetState "OfferPlaced";
    C_GetState "Accept"];
  vars =
   [(C_GetVar "CurrentState", State); (C_GetVar "InstanceOwner", Proc);
    (C_GetVar "Description", Proc); (C_GetVar "AskingPrice", Int);
    (C_GetVar "InstanceBuyer", Proc); (C_GetVar "OfferPrice", Int);
    (C_GetVar "GlobalBalances", CArr (Proc, Int));
    (C_GetVar "Random_param_offer", CArr (Proc, Int));
    (C_GetVar "Random_param_offer_r", CArr (Proc, Int));
    (C_GetVar "Random_param_offer_a", CArr (Proc, Int))];
  funcs =
   [{requires_ =
      C_And (C_Equals (C_GetState "ItemAvailable", C_GetVar "CurrentState"),
       C_Greater
        (C_GetI (C_Get_Param "Random_param_offer", C_Get_Param "sender"),
        C_GetVar "AskingPrice"));
     name_ = "make_offer"; params_ = [C_Get_Param "sender"];
     body_ =
      C_Seq
       (C_Seq
         (C_Seq (C_Value CUnit,
           C_Seq (C_Value CUnit,
            C_Seq (C_Assign ("InstanceBuyer", C_Get_Param "sender"),
             C_Seq
              (C_Assign ("OfferPrice",
```

```
              C_GetI (C_Get_Param "Random_param_offer",
               C_Get_Param "sender")),
            C_Assign ("CurrentState", C_GetState "OfferPlaced"))))),
       C_Value CUnit),
      C_Case ("Revert",
       [(C_Equals (C_Get_Param "sender", C_GetVar "InstanceOwner"),
         C_Assign ("Revert", C_Value (CBool true)));
        (C_Equals
          (C_GetI (C_Get_Param "Random_param_offer", C_Get_Param "sender"),
          C_Value (CCons 0)),
         C_Assign ("Revert", C_Value (CBool true)))])};
   {requires_ =
     C_And (C_Equals (C_GetState "OfferPlaced", C_GetVar "CurrentState"),
      C_Not
       (C_Equals
         (C_GetI (C_Get_Param "Random_param_offer", C_Get_Param "sender"),
         C_Value (CCons 0))));
    name_ = "reject"; params_ = [C_Get_Param "sender"];
    body_ =
     C_Seq (C_Assign ("InstanceBuyer", C_Value (CProc "sender")),
      C_Seq (C_Assign ("CurrentState", C_GetState "ItemAvailable"),
       C_Value CUnit))};
   {requires_ =
     C_And (C_Equals (C_GetState "OfferPlaced", C_GetVar "CurrentState"),
      C_Not (C_Equals (C_Get_Param "sender", C_GetVar "InstanceOwner")));
    name_ = "accept"; params_ = [C_Get_Param "sender"];
    body_ =
     C_Seq
      (C_Seq (C_Value CUnit,
        C_Seq (C_Assign ("CurrentState", C_GetState "Accept"),
         C_Value CUnit)),
      C_Case ("Revert",
       [(C_Equals
          (C_GetI (C_Get_Param "Random_param_offer", C_Get_Param "sender"),
          C_Value (CCons 0)),
         C_Assign ("Revert", C_Value (CBool true)))]))}]}
```

⟨2022⟩ SMART TYPES FOR SMART CONTRACTS VALIDATION

Ronaldo Corte

NOVA SCHOOL OF