Monitoring Smart Contracts with RTLola

Saarland University

Department of Computer Science

Bachelor's Thesis

submitted by

Frederik Scheerer

Saarbrücken, April 2021

Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Jana Hofmann, M.Sc.
	Jan Baumeister, M.Sc.
Reviewer:	Prof. Bernd Finkbeiner, Ph.D.
	Robert Künnemann, Ph.D.
Submission:	16 April, 2021

Abstract

Smart contracts are small computer programs, which are part of a blockchain and describe digital contracts. With them, one does not need to trust a third party, because the contract acts accordingly by itself. Since smart contracts cannot be altered after they are added to the blockchain, it is especially important that they behave correctly in all cases. They also often handle a huge amount of money, where an error would be fatal. Runtime monitoring checks the specification about the behavior of the program during the runtime of the program. In comparison to static methods, which have to take all possible executions into account, monitoring has the advantage that it only has to check the current execution. Because of that, it has no problems with scalability, and the monitor can directly act accordingly once it encounters a violation. One way to express these specifications is with the stream-based monitoring language RTLola by writing stream-equations. Input streams receive values from the state of the monitored program. Output streams are then computed based on current, but also earlier values from input streams and other output streams. A Trigger is a boolean expression, that indicates a violation of the specification. We implemented a translator, which receives a Solidity smart contract and a RTLola specification. It produces a new smart contract that behaves the same but where a monitor checks the specification during the runtime of the contract. Once a violation of the specification occurs, the monitor can directly react to the error by executing specified code.

Acknowledgements

I am very grateful to Prof. Bernd Finkbeiner for offering me this interesting and practical topic. I also wish to express my sincere thanks to my advisors Jana Hofmann and Jan Baumeister for their guidance, support, and patience throughout my thesis. Furthermore, I would like to thank Prof. Bernd Finkbeiner and Robert Künnemann for reviewing this thesis. Lastly, I would like to express gratitude to my family and friends for their ongoing support. Particularly, I want to thank Jasmin Gratz and Gideon Geier for proofreading this thesis and giving valuable feedback.

Contents

Chapter 1

Introduction

Smart contracts are digital contracts given as a computer program that is part of a blockchain. These digital contracts are used in various places: For example for managing supply chains, to protect copyrights or in insurance applications [1]. One aspect that a lot of these places have in common is that they often deal with a huge amount of money. Because of that, smart contracts are a great target for attackers. In 2016 hackers were able to hack the smart contract "The DAO" due to a bug in the contract. The hackers were able to steal cryptocurrency with a value of 60 Million Dollars [2]. Another incident took place in 2017, where cryptocurrency with an estimated value that corresponds to up to 300 Million Dollars was permanently frozen due to a bug in the Parity smart contract [3]. This bug was triggered accidentally by a developer searching for bugs in the contract. These events show that the correctness of smart contracts is very important since a mistake could easily cost millions of dollars. Besides that, another problem with smart contracts is that the developer can not fix the contract after it is published. Usually, a developer can provide a patched version when an error is found. This is not possible with smart contracts: Since smart contracts are part of a blockchain, they are immutable.

We propose a way to add runtime monitoring to smart contracts. We implemented a compiler, that takes a Solidity smart contract as an input together with a RTLola specification, and then produces another smart contract. This smart contract behaves exactly the same as the input but includes a monitor that checks the specification at runtime of the contract. Solidity [4] is a popular smart contract programming language for the cryptocurrency Ethereum [5]. RTLola [6] is a stream-based monitoring language. The monitor receives values from the state of the contract, the input streams. Based on these values, the monitor can compute other values, the output streams. The user can define triggers, that check for a violation of the specification. A trigger consists of a boolean expression, and when the expression becomes true the monitor executes user-specified code to handle the violation of the specification.

With runtime monitoring, even if the developer made a mistake in developing the contract, the monitor would catch the error and prevent most damage from happening. Compared to static analysis, monitoring has the advantage that it only has to analyze one trace, the one of the current execution, instead of all traces. This often allows checking more complex specifications. One disadvantage of monitoring is that it introduces additional overhead during runtime. We analyze the overhead of our monitoring approach with two example contracts. If the overhead introduced by the monitoring is reasonable, it might be acceptable if in return not millions of dollars are lost due to a bug in the contract.

This thesis is structured in the following way: In Chapter 3 we give all the background information needed for the thesis. In this chapter, we introduce smart contracts and the RTLola monitoring language. The chapter also explains how to analyze the RTLola specification to calculate the dependency graph, the memory bound, and the evaluation order. Chapter 4 deals with the idea behind our monitoring approach. We present how the RTLola specification is formulated for a given contract and how that specification is then transformed into a monitor in Solidity. Following that, we present some details about the implementation of the compiler in Rust in Chapter 5. We then evaluate our monitoring approach using two example contracts in Chapter 6. For the evaluation, we measure the gas usage of the examples and calculate the overhead that is introduced by the monitoring.

Chapter 2

Related Work

Closest to our approach is the runtime verification tool ContractLarva [7]. The tool takes a smart contract written in Solidity and a specification as an input and modifies the smart contract in a way, that the specification is checked during the runtime of the contract, just like the approach shown in this thesis. But instead of giving the specification in RTLola, ContractLarva takes a specification in the form of a dynamic event automaton. This type of automaton has an internal state, and transitions are annotated with conditions on specific events of the contract and conditions on the internal state of the monitor. When taking a transition, the monitor can modify the internal state. Some states of the contract are marked as bad states and when a bad state of the automaton is reached, the monitor executes predefined code. For example, reverting the transaction or bringing the contract into a specific state. ContractLarva extends the smart contract by a monitor that keeps track of the current state of the automaton and adds triggers to the contract such that the monitor is notified about specific events. Control-flow triggers activate before or after a function is executed and data-flow triggers activate when a global variable is assigned to. When describing the transition conditions, one can access the function's parameters, the global variables of the contract or monitor, and also the old value of an assigned variable. One advantage of giving the specification in RTLola stream-expressions instead of an automaton is that it most of the time feels more natural to formulate the specification in stream expressions than to think about how to represent the specification in states and transitions. Our approach currently does not support data-flow triggers like ContractLarva does, but allows accessing a functions return values, which is not possible with ContractLarva.

The RTLola language was already successfully used in different applications. The stream-based monitoring language Lola [8], the language RTLola was derived from, was extended with parameterized streams and used to analyze network traffic and detect web application fingerprinting attacks [9]. It was also shown how Lola and RTLola can be used to not only monitor unmanned aircrafts but also how to analyze their log files by calculating statistics afterwards [10, 11, 12]. Besides that, RTLola was successfully used to monitor web-based auction systems [13].

There are also other approaches for compilers that take a RTLola specification as an input. Baumeister et al. [14] presented a compiler, that translates specifications given in RTLola into synthesizable VHDL code. Hardware monitoring on an FPGA has the advantage that it is highly parallel and consumes less power than software-based monitoring. Finkbeiner et al. [15] presented a "verifying compiler", that translates specifications given in Lola into very efficient, parallel implementations in Rust. The generated Rust code contains annotations so that the functional correctness of the monitor can be automatically verified. Especially when using a monitor to guarantee functional correctness, it is essential that the monitor itself is correct. By using this compiler one can make sure that it is correct by automatically verifying the correctness of the monitor.

Instead of stream-based languages, the specifications for earlier approaches of runtime monitoring are often based on Linear Temporal Logic (LTL). The Temporal Rover [16] is a commercial tool, that is capable of taking C, C++, Java, Verilog, or VHDL code with an LTL based specification and modifies the code in a way, that it behaves exactly the same besides that the specification is checked during the runtime. The specification and the code that is executed once the specification is violated are written directly into the source code as comments.

Nasa's Java PathExplorer [17] also uses LTL-based specifications to automatically instrument java bytecode and monitor the program for functional correctness. This tool also adds deadlock and data race detection for concurrent programs and is mainly used for deeper insight into the program's execution during testing. It was successfully used to detect a deadlock in 90,000[1] lines of C++ code for a rover controller [17].

There also exist approaches for formal verification of smart contracts, so that the monitor does not have to handle the errors once they come up. Instead, it is assured that the contract is correct before committing it to the blockchain. Scilla [19] is an approach to verify Smart Contracts by introducing an intermediate language, that separates communication and computation of the contract. By doing that, properties can be verified with the proof assistant Coq [20]. Another approach of proving Smart Contract properties with a proof-assistant was presented in [21]. There the authors showed how to translate a subset of Solidity (excluding loops) or EVM bytecode to F* [22], a functional language designed for verification.

There are also approaches introducing new programming languages in which it is less likely for the programmer to make a mistake. An example of such a language is Bamboo [23]. Bamboo is a programming language for smart contracts that can be compiled to EVM bytecode. It is designed in a way that the programmer has to explicitly state which functions can be called in which order. This prevents the programmer from making a mistake when enforcing the order in which functions are allowed to be called manually. Other languages of this type are the python-based Vyper [24] and the Flint language [25].

[1] [18] states 35,000 lines

Other approaches verify smart contracts symbolically. Securify [26] is one of the most popular examples. It was used commercially and has verified over 18,000 contracts submitted by users. To do that, it analyzes the EVM bytecode and builds the dependency graph. This graph then is translated to Datalog [27], which a Datalog solver can analyze for predefined patterns. The results are interpreted and a security report is built, classifying each property as a violation, a warning (may be a false positive and has to be manually checked), or compliant. This is one of the things that lets Securify stand out in contrast to a lot of other symbolic analyzers. These combine the violation and the warning category, resulting in the user having to check a lot more potential errors. Securify also does well compared to other symbolic analyzers on how much unsafe behaviors are covered. [26]

Another example of a symbolic analyzer for smart contracts is Oyente [28], which analyzes the contract by symbolic execution. Oyente has the problems that it produces a lot of false positives, something that the ZEUS tool tries to improve. ZEUS [29] translates a Solidity smart contract to LLVM bitcode, which is then analyzed by an analyzer tool.

Chapter 3

Preliminaries

In this chapter, we give an introduction to smart contracts and RTLola. Solidity, a language to describe smart contracts for the Ethereum cryptocurrency, is introduced in Section 3.1.3 followed by an introduction to the RTLola monitoring language.

3.1. Smart Contracts

Since smart contracts are based on blockchains, we will introduce blockchains first. The section after that is about smart contracts in general. Finally, we introduce the smart contract programming language used in this thesis: Solidity.

3.1.1. Blockchains

Blockchains allow storing data immutable in a distributed system with no single point of failure [30]. In 1990 Haber and Stornetta [31] first introduced the concept of a blockchain as a way to time-stamp digital documents. Today, blockchains are used in various applications, for example as a music copyright system [32], for supply-chain tracing [33, 34], electronic voting [35], and even to help tackle the corona pandemic [36]. The most popular application was proposed by Satoshi Nakamoto in 2008 [37] for an electronic cash system, introducing the cryptocurrency Bitcoin.

A blockchain is a decentralized ledger. Decentralized in this context means, that every participant has a copy of the blockchain and is notified about updates to the blockchain through a peer-to-peer network. A blockchain is a chain of data, where each part of the chain, the blocks, contains the hash of the previous block (see Figure 3.1). The data that is stored in the blocks is different depending on what the blockchain is used for. In case of a cryptocurrency, the data contains transaction information. A blockchain also contains a way to reach a consensus about the correctness of the blocks. The most popular way to do that is to include a proof-of-work in every block. A proof-of-work is some piece of information that is hard to compute, but easy to check. This makes it difficult to add new blocks to the blockchain. One way to implement this proof-of-work

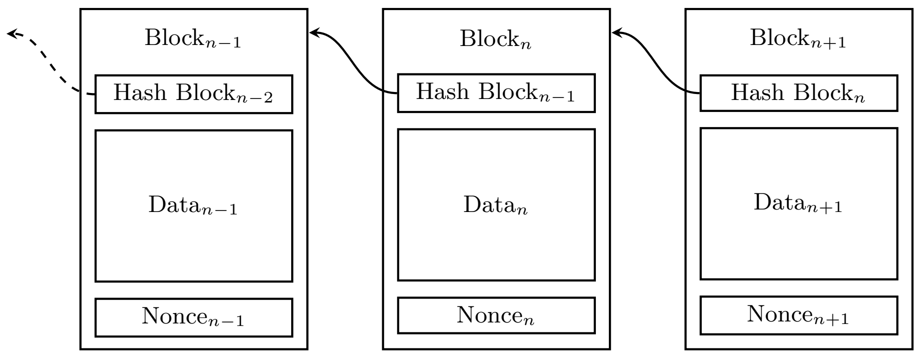


Figure 3.1.: A simplified version of three blocks from a blockchain. Each block contains the hash of the block before.

is to find a number, a nonce, so that the hash of the block with that nonce in it, starts with a defined number of zeros. This is hard to find as it is only possible by trying many different numbers, but it is easy to check, by computing the hash of the block. When someone wants to add new data to the blockchain, they broadcast this data in the network. Miners then collect this data and compete against other miners trying to solve the cryptographic puzzle of finding the proof-of-work first. After generating a valid block, the miner broadcasts the new block into the network and receives some amount of cryptocurrency as a reward for generating the block. Others in the network are able to check the validity of the block, by checking the hash of the previous block and validating that the proof-of-work is correct. [37]

The reason for introducing the proof-of-work in the blockchain is to make it very hard to modify a block. To modify a block, one would not only have to recompute all the following blocks but also do the proof-of-work for all these blocks all over again. Since everyone in the network has a copy of the blockchain, the attacker would need to do this on at least half of the copies of the blockchain in the network until others start to believe that the attackers version is the real version. [37]

There also exist other ways to verify the correctness of the blocks on the blockchain. One alternative is proof-of-stake, where a randomly chosen verifier verifies a new block. To become a verifier, one has to deposit some cryptocurrency that is locked during the time the verifier is active. The more cryptocurrency the verifier deposits, the more likely it is that they are chosen to verify a node. The verifier would not verify faulty blocks, because they would lose more of their deposited currency than they could gain. It has the advantage, that it does not require the huge amounts of energy that are required to mine new blocks by using proof-of-work [38]. The most famous cryptocurrencies, Bitcoin and Ethereum, burn about 1 Million Dollars of electricity and hardware costs every day for mining new blocks [38] and Bitcoin alone consumes each year about as much power as the country Belgium [39]. The cryptocurrency Ethereum plans to make the transition from proof-of-work to proof-of-stake in 2021/22 [40].

In summary, it can be said that blocks in a blockchain are immutable and irreversible. It is a distributed system with no single point of failure and no central authority that has to be trusted. [30]

3.1.2. Smart Contracts

A smart contract is a digital contract in form of a computer program that is part of a blockchain. The idea to formulate contracts digitally by embedding them in hardware or software was proposed by Nick Szabo in 1993 [41] and became popular with the introduction of the Ethereum cryptocurrency [5]. Instead of trusting the participants or a third party to act according to the contract, the smart contract enforces the contract itself by its own execution. When adding the smart contract to the blockchain, the contract becomes valid, and cannot be changed or retracted.

In comparison to traditional contracts, smart contracts have the advantage, that the code of a smart contract exactly defines what the contract is going to do transparently to everyone participating. There is no way to interpret a sentence in two different ways like it is the case with contracts written in natural language on paper. Another advantage is the speed at which smart contracts operate. It can take a person days to manually process a traditional contract, while smart contracts are much faster. Smart contracts also are really secure, since they use modern cryptography. [42]

3.1.3. Solidity

Solidity [4] is the most used programming language for smart contracts and is used by the cryptocurrency Ethereum [5]. A Solidity contract contains data and functions. Everyone can call these functions, and the functions can modify the data of the contract. Executing a smart contract consumes "gas", for which the participant calling a function has to pay. This prevents the contract from running too long.

We introduce the Solidity programming language with some examples. The following example contract is able to store a number:

```
contract example {
  int number;

  function setNumber(int num) public {
    number = num;
  }

  function getNumber() public view returns (int) {
    return number;
  }
}
```

The contract `example` contains two functions and one global variable `number` of type `int`. The `setNumber` function sets the number of the contract to the number given as an argument. The `getNumber` function returns the number stored in the contract at the moment when the function is called. Both functions are marked public, which means that everyone can execute them. Every new call to the function overwrites the old number. The `getNumber` function is marked as view because it does not change the state of the contract.

Most of the time it is not desirable for everyone to be able to call every function. To prevent this, it is possible to access the address of the party calling a function. If this address does not correspond to the person that is allowed to call the function, one can act accordingly, for example reverting all the changes made by that function call. This is possible because a function in a smart contract can either be executed completely or not at all. When calling `revert()` somewhere inside a function, the contract is brought back to the state it was in before that function was called.

revert

In the last example, everyone was able to change the number stored in the contract. We now modify the contract in a way, such that only the creator of the contract is able to change the number, but everyone is still able to read the number:

```
contract example {
  int number;

  address creator;

  constructor() public {
    creator = msg.sender;
  }

  function setNumber(int num) public {
    number = num;
    if(msg.sender != creator) revert();
  }

  function getNumber() public view returns (int) {
    return number;
  }
}
```

This example adds a constructor function to the contract. The `msg.sender` variable always holds the address of the one calling a function. These addresses identify all users and contracts. When the creator of the contract calls the constructor function, the contract stores the address of that person. This way, when executing the `setNumber` function, the contract checks that the person calling that function really is the same person that also called the constructor function. In the example above, this check is done after assigning the global variable. But since function calls are either executed completely or not at all, the variable does not change if the person calling the function is not the creator of the contract.

Each Solidity program has its own wallet and can also own and transfer Ether, the currency of Ethereum, to other accounts. It is possible to attach Ether to function calls, what is then transferred to the wallet of the contract. We now change the example contract in a way, that Ether has to be paid to the creator of the contract in order to change the number:

```
contract example {
  int number;
  address payable creator;

  uint COST = 5 ether;

  constructor() public {
    creator = msg.sender;
  }

  function setNumber(int num) public payable {
    number = num;
    require(msg.value == COST);
    creator.transfer(COST);
  }

  function getNumber() public view returns (int) {
    return number;
  }
}
```

The `setNumber` function is marked as `payable`, which means that the caller of that function is able to attach Ether to the function call. Also, the variable of the address of the creator has to be marked as payable so that the contract is able to send currency to that address. The `require()` function is a shortcut for the if-expression used in the example above: if the expression is not fulfilled, the transaction is reverted. The function call is only valid if the Ether that was sent with the function call is exactly the amount it costs to change the number. The attached Ether is now in the wallet of the contract. With the call to `creator.transfer()`, the contract sends that fee, which currently resides on the wallet of the contract, to the wallet of the creator of the contract. payable function

3.2. RTLola

RTLola [6] is a stream-based monitoring language. A RTLola specification consists of input streams, output streams and triggers. A *stream* is a finite sequence of values of a specific type. The input streams receive values describing the state of the monitored program. The output streams are defined by stream-expressions and computed based on current, but also previous values of the input streams and output streams (see Figure 3.2). A *trigger* is a boolean stream-expression, where the expression becoming true indicates a violation of the specification. stream trigger

In contrast to the stream-based monitoring language Lola [8], where RTLola was derived from, RTLola the ability to handle real-time streams. Lola, on the other hand, expects all streams to be synchronized by a global clock, something that is often not realistic in real-time applications.

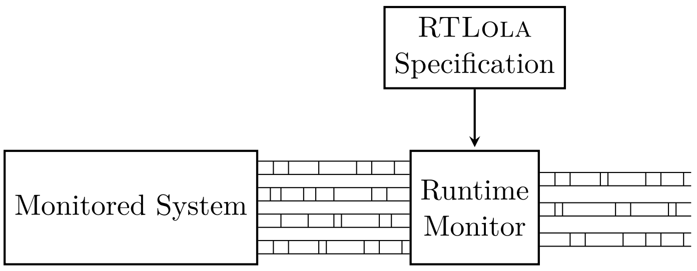


Figure 3.2.: The monitor constantly receives values from the monitored system, the input streams. Based on current and older values of these streams, the monitor then computes new values, the output streams.

The following example is a specification for a token transfer contract. The account of each user contains an amount of tokens. The user can then make a transaction, transferring a specific amount of tokens to another account.

```
input amount : Int64
output balance := balance.offset(by:-1).defaults(to:0) + amount
trigger balance < 0 "account can not get overdrawn"
```

The input stream `amount` receives a new value whenever a transaction takes place. If tokens are transferred to the account, the value in the `amount` stream is positive, if tokens are transferred from the account, the value is negative. To keep track of the current balance of the account, the specification defines an output stream `balance`. Whenever the `amount` stream receives a new value, the `balance` stream also calculates a new value, since the `balance` stream directly accesses the `amount` stream. Such access is called a *synchronous access*. The monitor calculates the new balance by taking the previous value of `balance`, i.e. the balance before the transaction, and adding the number of transferred tokens. Since this previous value potentially could not exist, e.g. if the monitoring was just started and there is no previous value, a default value is given. This default value is used when the accessed value does not exist. Because the default value is zero in this case, every account has a balance of zero when the monitoring starts. The trigger checks if the current balance is below zero. If the balance is below zero, the trigger-expression becomes true, which indicates a violation of the specification because the user is not allowed to overdraw their account.

synchronous access

activation condition

Activation conditions define when the monitor calculates a new value for an output stream. An activation condition consists of a boolean expression over stream names. Only if all streams of a conjunction in the activation condition receive a new value, that stream also generates a new value. Likewise for a disjunction: If any stream in the disjunction receives a new value, then the output stream generates a new value. In the example above, the activation condition is not given explicitly. In that case, the activation condition is calculated based on the synchronous access to the `amount` stream. But sometimes it is also necessary to give the activation condition explicitly:

```
output count @amount := count.offset(by:-1).defaults(to:0) + 1
```

The `count` stream is only incremented by one if the `amount` stream receives a new value. That stream, therefore, counts the number of transactions that took place.

A stream access can also be asynchronously. To demonstrate this, we define a `statements` stream. The monitor should add the current balance to the `statements` stream, whenever the input stream `create_statement` receives the value `true`:

```
input create_statement : Bool
output statements @create_statement :=
  if create_statement then
    balance.hold().defaults(to:0)
  else
    statements.offset(by:-1).defaults(to:0)
```

If the value in `create_statement` is `true`, the `statements` stream accesses the newest value of the `balance` stream *asynchronously* with the `hold()`-operator. The difference to a synchronous access is, that the asynchronous access does not require the balance stream to receive a new value in exactly that moment. Instead, the monitor simply accesses the newest value of that stream. Because this value potentially could also not exist, a default value is given here as well. If the `create_statement` stream receives a `false` value, the last balance in the `statements` stream is simply repeated.

asynchronous access

All the output streams shown so far in this chapter are *event-based*. The monitor calculates new values for these streams when the activation condition is satisfied. The other type of output stream is a *periodic stream*. That type of stream is evaluated in a fixed frequency:

event-based streams

periodic stream

```
output statements @1Hz :=
  balance.hold().defaults(to:0)
```

As an alternative to `statements` stream shown above, here the monitor does not add the current balance to the statements stream when a signal is given, but in a fixed frequency. Once a second, the current balance of the account is added to the statements stream.

RTLola can also aggregate over a *sliding window*. In the following example, the output stream `count_sec` contains the amount of transactions that took place in the last second:

sliding window

```
output count_sec @1Hz := amount.aggregate(over: 1s, using: count)
```

Once a second, the monitor evaluates this stream. Every time the stream is evaluated, the monitor counts all the values that were added to the `amount` stream in the last second. For this to work, the monitor has to know at what time the inputs arrive. RTLola supports two different modes to allow that: an online and an offline mode. In the online mode the monitor automatically adds the timestamp of the time when the data arrives. In the offline mode, the monitor analyzes a csv-file, where the timestamps are already included in the file.

3.2.1. Type system

Every stream in a specification has two types: The first type is the *value type* that defines which type of values are contained in the stream. These types include types

value type

such as `Bool`, `Int64` and `UInt64`. The user has to state the value types explicitly when defining an input stream. The value type is inferred automatically for an output stream.

stream type

The second type of a stream is the *stream type*. This type defines when the monitor calculates a new value for that stream. For event-based output streams, the stream type consists of a boolean expression over stream names. For such an event-based output stream, the stream type is either given explicitly with the activation condition, or is inferred automatically from the synchronously accessed streams. For a periodic stream, the stream type corresponds to the period in which the monitor computes a new value. [6]

3.2.2. Evaluating RTLola

This section presents the dependency graph of a RTLOLA specification and how that dependency graph can be used to determine in which order the monitor has to evaluate the streams. Our compiler does not support the full RTLOLA language. Because of that, the following definitions are simplified. We present the definitions for the subset of RTLOLA without sliding windows and periodic streams. The interested reader can find the definitions for the full RTLOLA language presented by Schwenger in [43].

Def. dependency graph

The *dependency graph* of a RTLOLA specification is a weighted and directed multi-graph, where every vertex corresponds to a stream from the specification. For a synchronous stream access from a stream s to s' with offset w, the graph contains an edge (s, w, s'). In the dependency graph, input streams never have outgoing edges, since they never depend on other streams. Triggers on the other hand never have incoming edges, since no stream can access a trigger.

Def. well-formed specification

For the monitor to be able to evaluate a RTLOLA specification, the specification has to be well-formed: A RTLOLA specification is *well-formed*, if there is no cycle in the dependency graph with a total weight of zero. If that would be the case, the streams in the cycle would influence the new value of each other, making it impossible for the monitor to evaluate them.

Example 3.2.1. To demonstrate the dependency graph, consider the following RTLOLA specification:

```
input a : Int64
output b := a + c
output c := b.offset(by:-2).defaults(to:0)
trigger c + a.offset(by:-1).defaults(to:0) > 0
```

Figure 3.3 depicts the dependency graph for this specification. Since the only cycle in the dependency graph b, c does not have a total weight of zero, this specification is well-formed. △

Def. evaluation order

The monitor has to update the streams in the correct order. Therefore, we define an evaluation order: The *evaluation order* $\prec$ of a RTLOLA specification with the dependency graph $DG = (V, E)$ is a partial order on streams. It satisfies the following rules:

1. $\forall i, j : in_i \prec out_j$: Every input stream is evaluated before any output stream.

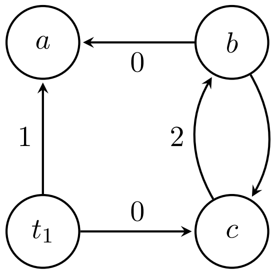


Figure 3.3.: The dependency graph for the specification in Example 3.2.1.

2. $(s, 0, s') \in E \Rightarrow s' \prec s$: If a stream s accesses a stream s' synchronously without an offset, then $s' \prec s$.

3. $s_i \prec s_j \wedge s_j \prec s_k \Rightarrow s_i \prec s_k$: ensures transitivity.

Next, we define the evaluation layers, which the compiler uses to determine in which order the monitor has to update the streams:

Definition 1 (Evaluation Layer [43])

The *evaluation layers* define the order in which the streams are evaluated. If a stream is in layer k, then all the streams that stream depends on are in a lower evaluation layer:

Def. evaluation layer

$$\mathit{Layer}(s_i) := 1 + \max\{\mathit{Layer}(s_j) \mid s_j \prec s_i\}$$

All input streams are in the lowest layer 0.

All streams that reside in the same activation layer do not depend on each other and could be evaluated in parallel. When a stream s is in a lower layer than another stream s', then s has to be evaluated earlier than s'.

Example 3.2.2. The evaluation order of the specification in Example 3.2.1 is the transitive closure of the relation $\prec$ with

$$a \prec b\ ,\ a \prec c\ ,\ a \prec t_1 \qquad \text{Rule 1}$$
$$a \prec b\ ,\ c \prec b\ ,\ c \prec t_1 \qquad \text{Rule 2}$$

and has the following evaluation layers:

$$\mathit{Layer}(a) = 0$$
$$\mathit{Layer}(c) = 1$$
$$\mathit{Layer}(b) = \mathit{Layer}(t_1) = 2.$$

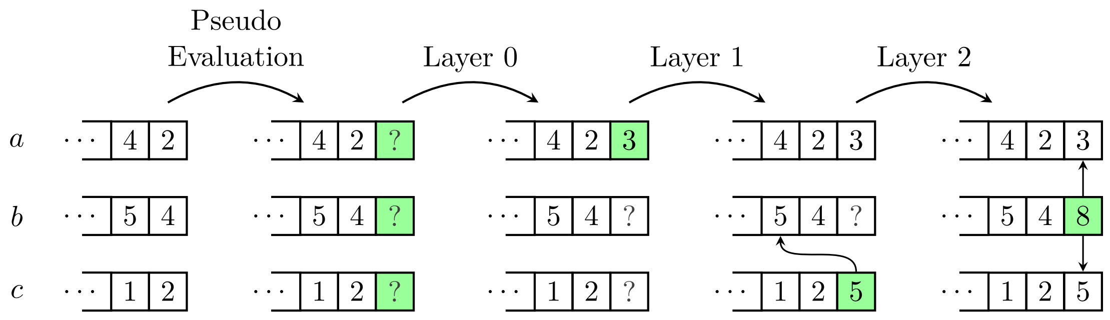


Figure 3.4.: Example for the evaluation of the specification in Example 3.2.1 when adding the value 3 to the input stream.

First, the monitor evaluates the input stream a, followed by the stream c. Followed by that, the monitor evaluates output stream b and the trigger. The order in which b and the trigger are evaluated does not matter, since they do not depend on each other. They can also be evaluated in parallel. △

The evaluation order only considers synchronous accesses without an offset. The order in which the monitor accesses previous values of streams does not matter, since all these values already exist. Nevertheless, for the accesses with an offset to access the correct value, a *pseudo evaluation phase* takes place before the monitor calculates the actual values. During that phase, all the streams that receive a new value, first receive a pseudo value. After that, the monitor evaluates the streams and replaces these pseudo values with the actual ones. The evaluation order ensures that a stream never accesses the pseudo value of another stream.

pseudo evaluation

Example 3.2.3. Figure 3.4 demonstrates the evaluation of the specification in Example 3.2.1 when adding the value 3 to the input stream.

First, all streams that receive a new value (in this case all the streams) receive a pseudo value. In the following steps, the monitor replaces these pseudo values with the correct values. First, the input stream a receives the value 3. In the next evaluation layer, the output stream c generates the new value, by accessing b with an offset of 2. Without the pseudo evaluation phase, the monitor would access b at the wrong position: It would not access the 5 but the value before that. Finally in layer 2, the monitor calculates the value for the output stream b by summing the newest values of a and c. In parallel to the update of b, the monitor checks the trigger condition of the trigger (not depicted in the figure). △

3.2.3. Storage Requirement

The storage requirement of a stream specifies how many values of that stream the monitor needs for a successful evaluation. This number is constant in the length of the trace, and can therefore be used by the compiler to determine how many values the monitor has to store for each stream.

Definition 2 (Storage Requirement [43])

Def. storage requirement

The *storage requirement* $\kappa(s)$ of a stream s in a RTLola specification with the dependency graph $DG = (V, E)$ defines how many values of that stream are needed for the evaluation:

$$\kappa(s) := \max\{w \mid (s', w, s) \in E\} + 1$$

The storage requirement of a stream is the maximal offset of an access to that stream. In addition, the monitor has to store the current value of every stream, so that other streams can access the current value. The storage requirement of a trigger is special. Since a trigger is never accessed, the monitor does not has to store any value of a trigger at all.

Example 3.2.4. The streams in the specification in Example 3.2.1 have the following storage requirements:

$$\kappa(a) = 2$$
$$\kappa(b) = 3$$
$$\kappa(c) = 1$$
$$\kappa(t_1) = 0$$

The monitor only has to store the last two values of the input stream a for a successful evaluation. For the output stream b it has to store three values, for c one value and no value at all for the trigger. △

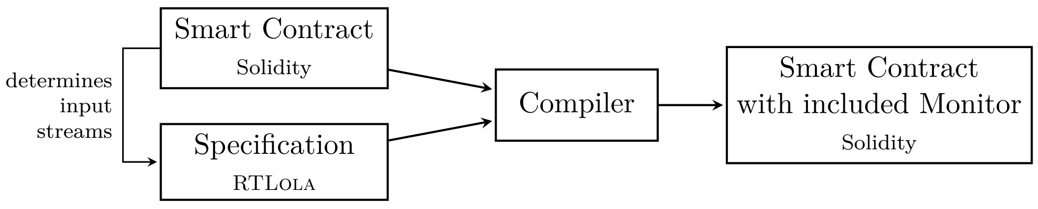
Chapter 4

General Overview

This chapter presents the compilation from a smart contract together with a specification into a safe smart contract. As long as the specification holds, the safe smart contract behaves the same as the input contract but is checked against the specification during its runtime. First, we give an overview of the compilation. Then, the details on how the combination of the input contract and the specification results in the output contract are presented. The details about the implementation of the compiler in Rust follow in the next chapter.

4.1. Smart Contract Specifications in RTlola

The compiler takes a smart contract in the form of a Solidity file, and a RTLola specification as an input. To be able to write a specification in RTLola for a given contract, the user has to construct input streams based on the contract:



In our setup, for each function in the contract, the user can include a set of corresponding input streams in the specification. Whenever the function is called, the monitor produces new values for the input streams based on that function call. The monitor observes every function in the input contract, that has at least one input stream associated with it.

We need a naming convention, to be able to conclude which input stream belongs to which part of the contract: Suppose there is a function called *⟨function-name⟩* in the input contract. Then, the user can include the following input streams in the specification:

- *⟨function-name⟩*`__`*⟨parameter-name⟩*: When the function is called, the new value of the stream is the value of the function argument *⟨parameter-name⟩*.
- *⟨function-name⟩*`__`*⟨return-value-name⟩*: When the function is called, the new value of the stream is the value *⟨return-value-name⟩* returned by that function call.
- *⟨function-name⟩*: This input stream of type `Bool` always receives the value `true` when the function is called. This stream never contains a value of `false`. When no one called that function, then no value is added to that stream.

artificial input streams

Besides that, it is possible to include *artificial input streams* in the specification. The values to these artificial input streams are automatically forwarded when someone calls a function:

- `current_time: UInt256`: The new value for this input stream is the current time at which the function was called.
- `msg_sender: UInt256`: The new value for this input stream is the address of the party calling the function.
- `attached_value: UInt256`: The new value for this input stream is the amount of money that the caller attached to a payable function call.
- `called_function: String`: The new value for this input stream is the name of the function that was called.

The difference between the *⟨function-name⟩* stream and the `called_function` stream is that the *⟨function-name⟩* stream only receives a value if someone actually called that function. This allows for the *⟨function-name⟩* stream to be included in the activation condition. Especially for streams without any arguments or return values, this is very useful. But we also need the `called_function` stream, when calls to different functions influence the new value of an output stream in different ways.

By using the presented input streams, the monitor is now able to observe the execution of the contract. But once a violation of the specification occurs, there has to be a way for the monitor to react to the error. To be able to do that, each trigger of the specification is associated with a *trigger function*. A trigger function is a function in the contract that the monitor executes when a trigger is checked and the trigger condition is satisfied. Each trigger needs a message, to indicate to which trigger function that trigger belongs. This message has to be either *⟨trigger-function⟩* or *⟨trigger-function⟩*: *⟨trigger-message⟩*. The monitor executes the function *⟨trigger-function⟩*, whenever the trigger activates. It is also possible for the same trigger function to be associated with multiple triggers. If the trigger function does not already exist in the input contract, the compiler creates

trigger function

```
contract example {

   int256 last_arg1;

   function example_function(int256 arg1, int32 arg2) public returns(bool equal,int256
       sum){
      equal = arg1 == last_arg1;
      last_arg1 = arg1;
      return (equal,arg1+int256(arg2));
   }
}
```

Listing 4.1: The example contract used for this chapter.

```
input example_function__arg1 : Int256
input example_function__arg2 : Int32
input example_function__equal : Bool
input example_function__sum : Int256

output sum := example_function__arg1 + cast(example_function__arg2)

trigger (example_function__arg1 == example_function__arg1.offset(by:-1).defaults(to:0))
    != example_function__equal "wrong_bool: the return boolean does not correspond to
    the equality of the argument and the previous argument"

trigger sum != example_function__sum "wrong_sum: the return value does not correspond
    to the sum of the arguments"
```

Listing 4.2: The specification for the `example_function` in Listing 4.1.

that function automatically. In that case, the trigger function contains a placeholder, which the user has to fill out afterward.

The monitor is capable of monitoring parameters and return values of type boolean, signed, and unsigned integer. The type of the stream for a Solidity type directly corresponds to the appropriate RTLola type. The one exception is the Solidity `address` type. The monitor converts the address to an unsigned integer, and it can therefore be used as an input to an `UInt256` input stream.

We use the contract in Listing 4.1 as an example contract for this chapter. This contract contains one function with two arguments. The global variable `last_arg1` contains the first argument to the function the last time the user called that function. The first return value is a boolean indicating whether the first argument is the same as it was the last time someone called that function. The second return value is just the sum of the two function arguments.

Listing 4.2 depicts the specification for that function. First, the specification defines input streams that capture the values of the function. In this example, there is one input stream for each argument and one input stream for each return value of that function. Next, the specification defines an output stream, which keeps track of the sum of the arguments to the function (line 6). The cast-operator adapts the type of the second argument to also be of type Int256. Whenever someone calls the example_function, the monitor adds the values of the arguments arg1 and arg2 to the corresponding input streams. Equally, the monitor adds the return values of the function to the two other input streams. When someone calls the function, the activation condition of the sum stream is satisfied, since the streams of both arguments receive a new value at the same time. Because of that, the monitor calculates the new value for the sum stream too. The first trigger in the specification (line 8) checks, that the equal return value is true iff the first function parameter is the same as the last time someone called that function. If the trigger condition is satisfied, which indicates a violation of the specification, the monitor executes the trigger function wrong_bool. The second trigger checks that the returned sum is equal to the sum of the function parameters (line 10). The monitor executes the wrong_sum function if this trigger activates.

4.2. Compiling the Specification to a Monitor

To compile the RTLola specification into the output contract, we require the following steps:

- Implement each stream and trigger in Solidity code.
- Add the corresponding values to the input streams whenever someone calls a function.
- Update the output streams according to the new values in the input streams and check the trigger conditions.

In this section, we present these steps in detail.

4.2.1. Implementing Streams

For every stream, the monitor has to store the last values of that stream. The number of values that the monitor has to keep is determined by the storage requirement of that stream (see Definition 2). The monitor stores these values in a ring buffer, implemented as an array of the type that corresponds to the type of the stream:

➔Def. 2, p. 17

```
int256[2] example_function__arg1_buffer;
int256 example_function__arg1_current;
bool[2] example_function__arg1_valid;
```

This buffer stores the last two values of the example_function__arg1 input stream, since the memory requirement of that stream is equal to two. Since the newest value does not always reside at the same index, the *_current variable stores the index of the newest element in the buffer. The *_valid buffer indicates, which elements of the array are valid entries. The monitor uses that information to determine whether to use the default value when accessing a value that does not always exist. When the monitoring starts, all entries are invalid.

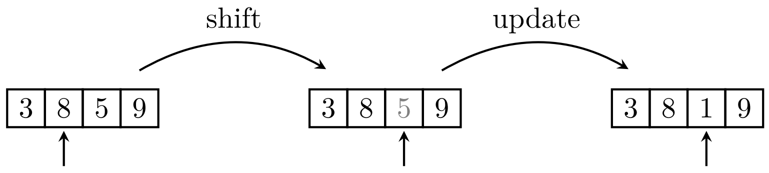


Figure 4.1.: The value 1 is added to a stream with a memory bound of 4. The last four values are 5,9,3,8 (with 8 as the newest value). Due to the evaluation order, the monitor never accesses the old value 5 before it is overwritten with the new value for the stream.

To make space for a new value, the compiler adds the `shift_*` function for every stream:

```
function shift_example_function__arg1() private {
    example_function__arg1_current =
        (example_function__arg1_current + 1) % 2;
}
```

This function increments the index of the current value in the buffer.

Depending on whether the stream is an input or an output stream, the way the monitor adds new values to a stream differs. For an input stream, the compiler generates an update function, which takes the new value for the stream as an argument and places it at the correct position in the buffer:

```
function update_example_function__arg1(int256 example_function__arg1) private {
    example_function__arg1_buffer[example_function__arg1_current] =
         example_function__arg1;
    example_function__arg1_valid[example_function__arg1_current] = true;
}
```

To add a new value to the stream, the monitor first has to shift the stream, and then call the update function with the new value (see Figure 4.1 for an example). The compiler separates the code for shifting the current index and adding a new value to the buffer so that a pseudo evaluation phase is possible, where the monitor first shifts all streams, and then adds the correct values.

For an output stream, the compiler also adds an update function. In contrast to that for an input stream, this update function does not take the new value as an argument. Instead, the monitor calculates the new value based on the stream expression defining that output stream:

```
function update_sum() private {
    sum_buffer[sum_current] =
        example_function__arg1_buffer[example_function__arg1_current]
        + int256(example_function__arg2_buffer[example_function__arg2_current]);
    sum_valid[sum_current] = true;
}
```

4.2.2. Accessing other Streams

The stream expressions of output streams access other streams in different ways. We now present how the compiler supports these different accesses.

For a synchronous access to the newest value of a stream, the monitor directly accesses the newest value on the buffer:

```
example_function__arg1_buffer[example_function__arg1_current]
```

This is not so easy for a synchronous access with an offset, since the accessed value potentially could not exist. Because of that, the compiler generates a helper function for accessing a stream with an offset:

```
function get_example_function__arg1_with_offset(uint256 offset, int256 default) private
     view returns (int256){
    int256 index =
       int256(example_function__arg1_current) - int256(offset);
    if (index < 0) { index += 2; }
    if (example_function__arg1_valid[uint256(index)]) {
       return example_function__arg1_buffer[uint256(index)];
    } else {
       return default;
    }
}
```

The compiler generates this function for every stream. The function takes the offset and the default value as an argument and returns the value at that offset if the value is valid, and returns the default value otherwise. The value of 2 that is used in this function corresponds to the storage requirement of the stream, which is equal to the size of the array.

To support the asynchronous access to a stream, the compiler generates another helper function:

```
function example_function__arg1_current_value(int256 default) private {
    if (example_function__arg1_valid[example_function__arg1_current]) {
       return example_function__arg1_buffer[example_function__arg1_current];
    } else {
       return default;
    }
}
```

This function returns the newest value of the stream if it is valid, and the default value if the newest value does not exist.

4.2.3. Translating Stream Expressions

When updating an output stream, the monitor calculates the new value with the stream expression of that stream. For the monitor to be able to do that, the compiler has to translate the stream expression to Solidity code. The compiler does this translation recursively. To translate a binary operator, the compiler translates both operands recursively and then combines them with the corresponding Solidity operator. The same is done for unary operators: The compiler translates the operand recursively and then prepends it with the corresponding Solidity operator. To make sure that the order of precedence stays intact, the compiler surrounds all operations with parentheses. A stream access is formatted differently, depending on the type of access. Either the monitor directly accesses the value, or calls the corresponding access function:

- synchronous access without offset: `accessed_stream[accessed_stream_current]`
- synchronous access with offset: `get_accessed_stream_with_offset(offset, default)`
- asynchronous access: `accessed_stream_current_value(default)`

The compiler directly translated a cast in a stream expression to the corresponding cast operator in Solidity as shown in the update function of the `sum` stream above (see page 23). There also exist some library functions supported by the compiler. The arguments to calls to these functions are translated to Solidity recursively. Then, the compiler generates a call to the corresponding equivalent Solidity function with these arguments. The compiler also adds the code implementing that function to the output contract. Currently supported are the functions `min`, `max` and `abs` from the math library.

4.2.4. Triggers

Since the monitor does not need to store any previous values of a trigger, the compiler simply adds a *trigger-check function* for every trigger to the output contract:

trigger-check function

```
function check_trigger1() private {
    if (sum_buffer[sum_current] !=
         example_function__return1_buffer[example_function__return1_current]) {
        wrong_sum();
    }
}
```

This function checks the trigger condition and executes the *trigger function* if the condition is satisfied. The trigger function is a private function, which takes no arguments. If the input contract already includes a function with the name of a trigger function, then the monitor uses that function. Otherwise, the compiler creates the trigger automatically. Nevertheless, the user has to decide what should happen if the trigger activates. The user could for example decide to bring the contract to the state it was before the function call that made the trigger activate:

trigger function

```
function wrong_sum() private {
    revert();
}
```

Something the user has to keep in mind in that case is, that when the trigger activates, everything that happened after the function call is reverted. This also includes updating the streams with the new values. This has the consequence that after the trigger activates, the streams are exactly the same as before the user called the function.

Another option for the user is, to bring the contract to a specific state. In this case for example, the user could set the global variable to a specific value:

```
function wrong_sum() private {
    previous_arg1 = 0;
}
```

4.2.5. Updating the Streams

Now, the streams are implemented in Solidity. What is missing is, that the monitor has to update the streams whenever someone calls a monitored function. To be able to do that, the compiler relocates the body of a monitored function into a helper function. This helper function has the same definition as the original, besides that it is marked as private, i.e. it can only be called from inside the contract. Otherwise, it would be possible to execute the body of the function without the monitor noticing. With the help of the helper function, it is possible for the monitor to access the return values after executing the function. The monitor could need these return values to update the corresponding input streams.

```
function _example_function(int256 arg1, int32 arg2) private returns(bool equal, int256
     sum){
    // original function body
}

function example_function(int256 arg1, int32 arg2) public returns(bool equal, int256 sum){
    (bool return0, int256 return1) = _example_function(arg1,arg2);
    // update the streams here
    return (return0, return1);
}
```

The modified function depicted above behaves exactly the same as the original function. Missing are the function calls to the stream-shift functions, stream-update functions, and the trigger-check functions.

First, the monitor shifts all streams that depend on that function call. This is equivalent to the pseudo-evaluation phase and ensures that the monitor accesses the values with the correct offset. The *dependent streams of a function* are all the streams, that the monitor has to update when that function is called. These include all the input streams that receive a value from that function and all the artificial input streams. They also include all the output streams, where the activation condition is satisfied when the function is called. The activation condition for the output stream `sum` for example consists of the conjunction of the input streams for `arg1` and `arg2`. Since both these input streams receive a value from that function, the `sum` stream also depends on that function.

dependent streams of a function

After all the streams are shifted, the monitor updates the dependent streams in the correct order, sorted by their evaluation layer (see Definition 1). Finally, the monitor checks the trigger conditions for all the triggers that depend on that function. For the specification in Listing 4.2, the following would be a valid sequence of updates:

➔Def. 1, p. 15

```
// stream shifts
shift_example_function__arg1();
shift_example_function__arg2();
shift_example_function_equal();
shift_example_function_sum();
shift_sum();
// input stream updates
update_example_function__arg1(arg1);
update_example_function__arg2(arg2);
update_example_function__equal(return0);
update_example_function__sum(return1);
// output stream updates
update_sum();
// trigger checks
check_trigger0();
check_trigger1();
```

4.2.6. Special Variables

The compiler has to handle the Solidity variables `msg.sender` and `msg.value` special in the output contract. The variable `msg.sender` always holds the address of the party calling a function, and `msg.value` holds the amount of money the caller attached to a payable function call. In the output contract, the user does not call the function that was originally intended to be called anymore. Instead the user calls another function which then calls the helper function with the original function body. Because of that, the values of `msg.sender` and `msg.value` are not what the user expects them to be. Since the monitor calls the helper function from inside the contract, the `msg.sender` variable always holds the address of the contract itself, while `msg.value` never is set since no money is attached to the call to the helper function.

The compiler fixes this problem, by renaming every usage of `msg.sender` to `msg_sender` as well as `msg.value` to `msg_value` in the original function body. The monitor passed the values for these variables as extra arguments to the helper function. We demonstrate this with the following example:

```
function example_function(uint32 example_arg) public payable {
    require(msg.sender == example_address);
    require(msg.value == 5 ether);
    // ...
}
```

This `example_function` accesses the variables `msg.sender` and `msg.value` to restrict the access to this function. When monitoring that function, the monitor passes these values as extra arguments:

```
function _example_function(uint32 example_arg, address msg_sender, uint256 msg_value)
     private {
    require(msg_sender == example_address);
    require(msg_value == 5 ether);
    // ...
}
```

```
function example_function(uint32 example_arg) public payable {
    _example_function(example_arg, msg.sender, msg.value);
    // stream updates etc.
}
```

Now, the `_example_function` works as expected by the user, since the values of the variables are taken from the function the user actually called.

4.3. Optimizations

This chapter presents a few optimizations that the compiler implements. The interested reader can find the code produced by the compiler for the example contract of this chapter with all the listed optimizations in Appendix A.1. →App. A.1, p. 55

4.3.1. Storage Requirement of 1

A lot of streams have a storage requirement of 1. This means that the monitor only has to store the current value of that stream. For all these streams, the compiler simplifies the implementation of the monitor:

```
int256[1] example_function__sum_buffer;
bool[1] example_function__sum_valid;

function update_example_function__sum(int256 example_function__sum)
    private {
    example_function__sum_buffer[0] = example_function__sum;
    example_function__sum_valid[0] = true;
}
```

When the monitor produces a new value for a stream, the last value can simply be overwritten since it is not needed anymore. Because of that, the monitor does not have to keep track of the index in the buffer, since there is only one value anyway. For the same reason, the monitor does not need a shift function for such a stream. The monitor needs the `*_valid` boolean nevertheless, to make sure that a value exists when doing an asynchronous access.

4.3.2. Streams of Type String

The artificial input stream `called_function` is the only supported stream of type `String`. Since all the strings in this stream encode a function, the monitor internally encodes these functions with a unique integer, rather than the name of the function. Whenever a string is used in any stream expression, the compiler translates that string to the correct identifier. Besides the increased performance for not having to do string comparisons, this also has the advantage that comparisons with non-existing function names trigger an error at compile time.

4.3.3. Unused Functions

Often a stream is never accessed asynchronously. In these cases, the monitor never uses the asynchronous access function generated for that stream. Because of that, this access function is omitted in the output contract. The same is the case for streams that the monitor never accesses with an offset. Here, the corresponding access function can be omitted as well. Just like that, only the library function implementations that the compiler actually uses have to be included in the output contract. Leaving out all the unused code in the output contract does not only make the output contract shorter and therefore easier to read, but also reduces the deployment cost of the contract.

Chapter 5

Implementation

This chapter presents the implementation of the compiler in Rust. We first show the parsing of the input contract, the RTLOLA Frontend, and templates. Finally, these elements are combined to implement the compiler.

5.1. Parsing the Contract

This section describes the parsing of the Solidity input file. First, we introduce the pest parser generator with a grammar for parsing the body of a function. Afterwards, we present the internal representation of smart contracts in the compiler.

5.1.1. Pest Parser-Generator

The compiler parses the input contract by using the pest [44] parser generator for Rust. A parser generator automatically generates efficient code for parsing a given grammar, instead of the programmer having to write a parser for that grammar by themself. The pest parser generator uses *parsing expression grammars* (PEG). We present these PEGs with a grammar for parsing the body of a function.

parsing expression grammar

Since the compiler does not need to know the details of a function body, there is no need for the compiler to parse the function bodies completely. Instead, the internal representation just stores the function body as a string. Nevertheless, the opening brace of the function body must be correctly matched with the corresponding closing brace. The parser has to find the correct closing brace since the body of the function most likely also contains opening and closing braces. On the other hand is it possible, that a string or comment contains a brace, which the parser should not consider when looking for the correct brace. The solution for this problem is to differentiate the function body into parts:

```
function = { "function" ~ function_name ~ ... ~ "{" ~ function_body* ~ "}" }

function_body = { (matching_braces | quoted_string | body_char)* }

matching_braces = { "{" ~ function_body ~ "}" }
quoted_string = @{ "\"" ~ (!"\"" ~ ANY)* ~ "\"" }
body_char = _{ !"}" ~ ANY }

COMMENT = _{ "/*" ~ (!"*/" ~ ANY)* ~ "*/" | "//" ~ (!"\n" ~ ANY)* }
```

The `|`-symbol is a choice operator, while the `*`-operator means zero or more occurrences of that expression. Therefore, the body of a function consists of a sequence of matching brace pairs, quoted strings, or any other character. Matching braces are allowed to contain anything that would be a valid function body inside them. This way, it is possible to nest brace pairs inside of each other. The `~`-symbol is a chain operator. A quoted string consists of an opening quotation mark, followed by an indefinite amount of characters other than a quotation mark, and is then completed with the closing quotation mark. The `!`-symbol is a negative lookahead. Instead of reading anything from the input, the parser just tests that the beginning of the input does not match the expression contained in the negative lookahead.

Pest natively supports the handling of comments. Whenever the grammar defines a `COMMENT` rule, then the content of the rule is optionally inserted at every `~`-operator. The `_` preceding the left curly brace of a rule marks that rule as silent: It is not included as an extra node in the parse tree. A comment is either a block comment consisting of the starting and ending symbols or a line comment, that is ended by a newline character. Comments however are not allowed to be parsed inside quoted strings. The grammar prevents this with the `@`-symbol preceding the left curly brace of the `quoted_string` rule. It tells pest to not allow comments inside that rule.

The grammar restricts the `body_char` rule to not include a closing brace so that the `function_body` does not eat up the closing brace the function body is enclosed in (or the closing brace of a nested block). This is necessary because a PEG does not do backtracking, like a context-free grammar (CFG) would do. The `*` operator consumes anything from the input that matches, and would not leave a closing brace to close the function body. On the other hand, the grammar does not have to exclude the quotation mark or opening brace in that rule, because the choice operator is ordered, it is tested from left to right. Only if one choice fails, the parser tests the next choice. This is another difference between CTGs and PEGs. Since the input contract is expected to be correct, which also means that there are no unmatched quotation marks or braces inside, `body_char` is only tested if the character at the beginning of the input is neither a quotation mark nor an opening brace.

5.1.2. Internal Contract Representation

The parser only parses the parts of the specification that are relevant for the compilation. The internal representation stores all the other parts, those which are not relevant for the compilation, just as strings. These irrelevant parts include function bodies, struct

Solidity type	internal representation
`int256`	`Type::Int(256)`
`uint128`	`Type::UInt(128)`
`bool`	`Type::Bool`
`address`	`Type::Address`
`int256[]`	`Type::Other("int256[]")`
`SomeStruct`	`Type::Other("SomeStruct")`

Table 5.1.: Examples of the internal representation of different types.

or enum definitions, and types that can not be used as an input for an input stream. Table 5.1 depicts various Solidity types and the way the internal representation stores them. The type `int256[]` is an array, and can therefore not be used as a value for an input stream. Since the compiler does not need to know the specifics about such a type, it just stores the string, to be able to include it in the output contract.

The internal representation of a smart contract consists, besides information like the name of the contract, of a list of contract elements. A contract element is either a function, a global variable, or a definition of a struct or enum. The internal representation stores these elements in a list, to keep the order of the elements in the output contract intact. Table 5.2 shows different kinds of contract elements and the way the internal representation represents them.

5.2. Parsing the Specification

The compiler uses the RTLola Frontend [45] to parse the RTLola specification into the internal representation. The Frontend also analyzes the specification, checks it for correctness, and calculates all the information relevant for the compilation. This information includes the inferred activation condition, the activation layer, and storage requirement for every stream. Listing 5.1 depicts the internal representation for the example contract in Listing 4.2 from the last chapter. It includes the storage requirement (line 19), evaluation layer (line 20) and activation condition (line 21) of every stream.

→Lst. 4.2, p. 21

5.3. Templates

The compiler generates the output contract with the template-engine tera [46]. Tera loads a text file with placeholders in it and then renders the text with values filled in the placeholders. The templates can not only contain simple placeholders for strings but also allow for control flow structures like loops and conditions. One example for

```
RTLolaIR {
    inputs: [
        InputStream {
            name: "example_function__arg1",
            ty: Int(I256),
            memory_bound: Bounded(2),
            // ...
        },
        // ...
    ],
    outputs: [
        OutputStream {
            name: "sum",
            ty: Int(I256),
            expr: Expression {
                kind: ArithLog(Add, [Expression { kind: StreamAccess(InRef(0), Sync), ty:
                     Int(I256) }, /*...*/ ], Int(I256))),
                ty: Int(I256)
            },
            memory_bound: Bounded(1),
            layer: 3,
            ac: Some(Conjunction([Stream(InRef(0)), Stream(InRef(1))])),
            // ...
        },
        OutputStream {
            name: "trigger_wrong_bool:...",
            ty: Bool,
            expr: // ...
            memory_bound: Bounded(0),
            layer: 3,
            ac: Some(Conjunction([Stream(InRef(0)), Stream(InRef(2))])),
            // ...
        },
        // ...
    ],
    triggers: [
        Trigger {
            message: "wrong_bool: the return boolean does not correspond to the equality
                 of the argument and the previous argument",
            reference: OutRef(1),
            trigger_idx: 0
        },
        // ...
    ]
}
```

Listing 5.1: The internal representation of the RTLola specification depicted in Listing 4.2. The listing does not show the parts of the internal representation that are not relevant for the compilation.

contract element	internal representation
`int256 globalVariable1 = 10;`	`ContractElement::GlobalVariable( (Type::Int(256), []), "globalVariable1", Some("10") )`
`address payable globalVariable2;`	`ContractElement::GlobalVariable( (Type::Address, [VariableModifier::Payable]), "globalVariable2", None )`
`function exampleFunction(int64 arg1) public returns (bool){ // function body }`	`ContractElement::Function{ name: "exampleFunction", parameter: [((Type::Int(64),[]), "arg1")], scope: Scope::Public, return_value: [(Type::Bool,[])], body: "// function body" }`

Table 5.2.: Examples of various contract elements.

such a template is depicted in Listing 5.2. This template generates the asynchronous access function for a stream (see Section 4.2.1). Tera replaces the placeholders with the corresponding name and type of the stream. This template also supports the optimizations shown in Section 4.3: You can see one optimization in the way the buffer access is generated: Depending on the memory bound of the stream, the way the monitor accesses the current value of the buffer is different. Another optimization shown in that template is to only generate this function if it is actually used by the monitor. Due to the outer if-block, tera only generates any code for that function, if the stream has an asynchronous access.

5.4. Output Contract Generation

Figure 5.3 depicts an overview of the compiler. After parsing the input contract as well as the specification into their corresponding internal representation, the compiler generates the output contract: First, the compiler iterates over all the elements of the input contract. Depending on the contract element, the compiler copies the element over to the output contract or handles it as part of the monitor. If there exists any stream in the specification that receives a value from a function in the input contract according to the naming convention, then that function is monitored. For this function, the compiler adds a helper function with the original function body to the output contract and also generates the stream updates for that function. During this progress, the compiler keeps track of all the streams it generated an update for, to warn the user about streams that never receive any updates.

```
{% if has_hold_access %}
function get_{{ stream_name }}_current_value({{ stream_type }} default) private view
    returns ({{ stream_type }}){
    {%- if size > 1 %}
    if ({{ stream_name }}_valid[{{ stream_name }}_current]){
        return {{ stream_name }}_buffer[{{ stream_name }}_current];
    {%- else %}
    if ({{ stream_name }}_valid[0]){
        return {{ stream_name }}_buffer[0];
    {%- endif %}
    } else {
        return default;
    }
}
{% endif %}
```

Listing 5.2: Template for the asynchronous access function.

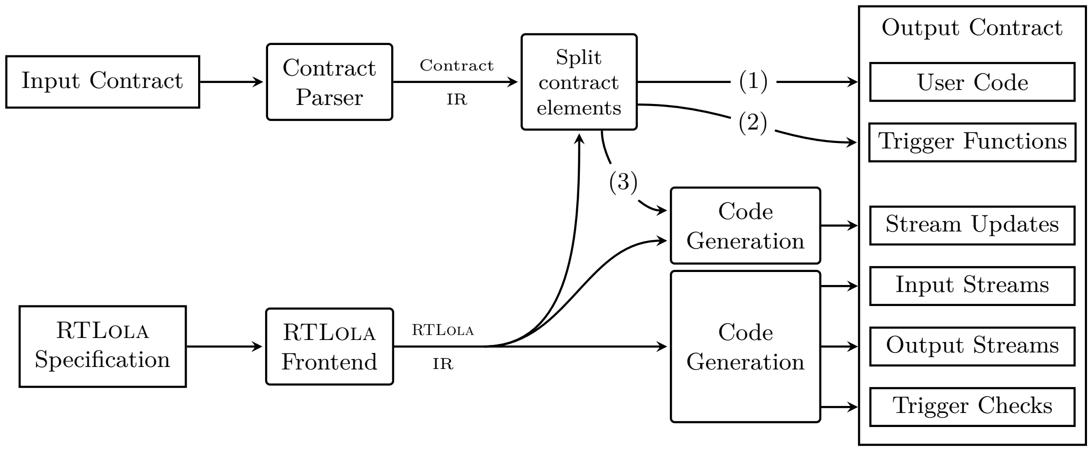


Figure 5.3.: Overview of the compiler implementation. The elements of the input contract are split into: (1) Global variables, enum/struct definitions, unmonitored functions, helper functions with the body of monitored functions. (2) Trigger functions (present in the input or automatically generated). (3) The definition of monitored functions to update the streams accordingly.

To make the output contract more readable, the contract elements are grouped into different sections. The first section contains all the user-defined code: every unmonitored function, other contract elements like global variables, and all the helper functions with the function body of a monitored function. The next section contains all the trigger functions, either copied from the input contract if provided by the user or automatically generated if they are not present in the input. When a trigger function is generated, the compiler inserts comments to the trigger function with all the trigger messages the function corresponds to. The compiler also adds a `revert("trigger_function")` statement in every generated trigger function. The message in the revert statement corresponds to the name of the trigger function and is used for debugging purposes. Next, the compiler adds all the stream implementations to the output contract. For that, the compiler translates all the stream expressions to Solidity first. The compiler does this separately from generating the streams so that it can keep track of which calls to access functions were generated. The compiler uses this information, to only generate the stream access functions that are actually used. As well as the calls to access functions, the compiler notes all calls to library functions, to only add the implementation for those library functions that are actually used. Then, the compiler adds the corresponding code for every stream in the specification to implement that stream in the output contract. This code is generated by using templates. The user can also instruct the compiler to add debugging functions. With these functions, the user can inspect the values of every stream, making debugging of the monitor a lot easier. In the end, the compiler adds the implementation for every used library function to the output contract.

Since the compiler did not take care of the correct formatting during compilation, it in the end formats the output contract correctly. For that, the compiler uses the prettier-solidity [47] plugin for the prettier [48] tool.

Chapter 6

Evaluation

We evaluate the implementation of the compiler with two example contracts. This chapter first presents these example contracts together with their specification in RTLola. In the next step, we evaluate the consumed gas by these contracts. Finally, we present the results of the evaluation.

6.1. Example Contracts

We evaluate the compiler with two example contracts already used by Azzopardi et al. [7]. The first contract is for an ordering system, where a buyer can order items from a seller. The second example is a token transfer contract, where the users can transfer tokens from one to another. This contract is special because the owner of the contract can change the implementation during the runtime of the contract. With runtime monitoring, it is ensured that every implementation follows the specification.

6.1.1. Ordering System

Azzopardi et al. [7] introduced their monitoring approach with an ordering system contract as an example. This contract is between two parties: a buyer and a seller. The Buyer can make orders for a specific item, which the buyer has to pay for once the seller delivered them. Listing 6.1 depicts the implementation of the contract that is used for the evaluation. This implementation is inspired by code shown by Azzopardi et al. [7].

```
contract OrderSystem {
    struct Order {
        uint256 timeOfDelivery;
        uint256 amount;
        bool delivered;
    }

    enum ContractStatus {UNOPENED, OPENED, ACCEPTED, CLOSED}
    ContractStatus status = ContractStatus.UNOPENED;
10
11      mapping(uint256 => Order) orders;
12
13      function openContract(uint256 endDate, uint256 price, uint256 minimumItems, uint256
             maximumItems) public payable {
14          status = ContractStatus.OPENED;
15      }
16
17      function acceptContract() public payable {
18          status = ContractStatus.ACCEPTED;
19      }
20
21      function placeOrder(uint256 orderId, uint256 itemsOrdered, uint256 timeOfDelivery)
             public {
22          orders[orderId] = Order(timeOfDelivery, itemsOrdered, false);
23      }
24
25      function deliveryMade(uint256 orderId) public payable {
26          orders[orderId].delivered = true;
27      }
28
29      function terminateContract() public {
30          status = ContractStatus.CLOSED;
31      }
32  }
```

Listing 6.1: The implementation of the ordering system in Solidity. The specification is not checked at all in this implementation.

The following paragraphs explain what the functions of the contract do. When the seller opens the contract by calling the `openContract` function (line 13), the seller can specify the earliest date the contract can be closed, the price for one item, and the minimum and the maximum number of items that the buyer has to order. After the seller opened the contract, the buyer has to accept the contract by calling the `acceptContract` function (line 17). When opening the contract, the seller has to deposit a performance guarantee, that is transferred to the wallet of the contract. The seller pays the performance guarantee, by attaching money to the call to the payable function `openContract`. The specification defines that the contract transfers the performance guarantee back to the seller if the seller complies with his side of the contract. Otherwise, the contract transfers the performance guarantee to the buyer. In the implementation shown in Listing 6.1, the contract never pays back the performance guarantee, it stays in the wallet of the contract. In the monitored version of this contract, the monitor will handle the payout of the performance guarantee with a trigger function. Similarly, the buyer has to deposit an escrow, to ensure that he pays for his delivered items. The buyer makes this deposit with the call to the `acceptContract` function.

When making an order with the `placeOrder` function (line 21), the buyer has to give a unique order id, the number of items ordered, and the time when the order has to be delivered by the seller. With the unique order id given when placing an order, the buyer has to mark the order as delivered as soon as it is. To do that, the buyer calls

1. This contract is between *⟨buyer-name⟩*, henceforth referred to as 'the buyer' and *⟨seller-name⟩*, henceforth referred to as 'the seller'. The contract will hold until either party requests its termination.
2. The buyer is obliged to order at least *⟨minimum-items⟩*, but no more than *⟨maximum-items⟩* for a fixed price *⟨price⟩* before the termination of this contract.
3. Notwithstanding clause 1, no request for termination will be accepted before *⟨contract-end-date⟩*. Furthermore, the seller may not terminate the contract as long as there are pending orders.
4. Upon enactment of this contract, the buyer is obliged to place the cost of the minimum number of items to be ordered in escrow.
5. Upon accepting this contract, the seller is obliged to place the amount of *⟨performance-guarantee⟩* in escrow.
6. Upon termination of the contract, the seller is guaranteed to have received payment covering the cost of the minimum number of items to be ordered unless less than this amount is delivered, in which case the cost of the undelivered items is not guaranteed.
7. The Buyer has the right to place an order for an amount of items and a specified time-frame as long as (i) the running number of items ordered does not exceed the maximum stipulated in clause 2; and (ii) the time-frame must be of at least 24 hours, but may not extend beyond the contract end date specified in clause 2.
8. Upon placing an order, the buyer is obliged to ensure that there is enough money in escrow to cover pending orders.
9. Upon delivery, the seller receives payment of the order.
10. Upon termination of the contract, any undelivered orders are automatically canceled, and the seller loses the right to receive payment for these orders.
11. Upon termination of the contract, if either any orders were undelivered or more than 25% of the orders were delivered late, the buyer has the right to receive the performance guarantee placed in escrow according to clause 5. Otherwise, it is released back to the seller.

Figure 6.1.: The natural language specification for the ordering system from the ContractLarva paper [7].

the `deliveryMade` function (line 25), and has to pay for the order at the same time, by attaching the payment to that function call. Either party can request the termination of the contract, by calling the `terminateContract` function (line 29).

Figure 6.1 depicts the specification for this contract in natural language. This specification is translated to RTLola next. We present some interesting examples of the specification in this section. If you are interested in the translation in detail, you can find the full specification for the contract in Appendix A.2.

➔App. A.2, p. 59

Our example needs the following input streams:

```
input current_time : UInt256
input sender_address : UInt256

input openContract : Bool
input openContract__endDate : UInt256
input placeOrder__orderId : UInt256
input deliveryMade__orderId : UInt256
input terminateContract : Bool
```

When the seller opens the contract by calling the `openContract` function, two input streams receive new values: the end date specified by the seller is placed in the `openContract__endDate` input stream, and the value `true` in the `openContract` stream. For this example, we assume that the buyer does not has to accept the contract. When the buyer places an order with the `placeOrder` function, the monitor inserts the order id in the placeOrder__orderId stream. Likewise, the order id is placed in the deliveryMade__orderId stream, when the buyer confirms the delivery of the order. If either participant requests termination of the contract by calling the terminateContract function, the value true is placed in the terminateContract input stream. To differentiate which party requested termination of the contract, we add the sender_address stream to the specification. Every time a party calls a function, this input stream receives the address of that party. The current_time input stream receives the current time, whenever someone calls a function.

The first specification we take a closer look at is the first part of §3: "*no request for termination will be accepted before (contract-end-date)*". In RTLola, we can express this as a trigger. When the trigger condition becomes true, a violation of the specification occurred and the associated trigger function example1 is executed:

```
trigger terminateContract && current_time < openContract__endDate.hold().defaults(to:0)
    "example1: the contract can only get closed after contract-end-date"
```

When a party calls the terminateContract function, but the current_time is less than the time specified when opening the contract, that is a violation of the specification. The trigger accesses the stream openContract__endDate asynchronously, since it is not expected that the contract is opened at the same time as it is closed. Instead, the value that the seller specified when opening the contract should be used. Since it is not possible to call two functions at the same time, it is not possible that input streams that correspond to different functions receive new values at the same time. Therefore, each output stream or trigger can only access the input streams of one function synchronously. Otherwise, that output stream would never calculate new values since the activation condition is never satisfied.

With the trigger shown above, it is however possible to terminate the contract before someone ever called the openContract function. The reason for that is, that if the openContract function was never called, the default value 0 is used instead of the end date. To prevent that, the specification defines another trigger:

```
trigger terminateContract && !openContract.hold().defaults(to:false) "example2: the
    contract can not get closed before it got opened"
```

If the terminateContract stream receives the true value because a party requested the termination of the contract, but the openContract stream has never received a true value before, the termination of the contract was requested before the contract ever being opened.

Now we want to look at the second part of §3: "*Furthermore, the seller may not terminate the contract if there are pending orders.*" This restriction only applies to the seller. Because of that, the monitor has to be able to distinguish which party called the terminateContract function. For that, the monitor stores the address of the seller when the seller opens the contract, to compare that address with the address of the party requesting the termination:

```
output seller_address @(openContract) := sender_address.hold().defaults(to:0)
```

The stream expression has to include the default value for the `sender_address` stream so that the type checking succeeds. Since artificial input streams always receive a new value, the monitor never uses this default value. We could also always include artificial input streams in the activation condition and then access them synchronously.

After that, the monitor has to remember which orders are pending and which are not. Now a problem occurs: For every possible order id, the monitor has to remember if that order was already ordered and delivered. For the RTLola language, this does not pose a problem. The RTLola language does support parametrized streams, which could handle that problem. The problem at the current time is, that these parametrized streams are not yet implemented in the RTLola Frontend. As a workaround, we use the following: We restrict the number of possible order ids to a limited set and create a separate stream for every possible order id.

For every order, the monitor stores if that order was already ordered and if that order was already delivered (here shown for order id 1):

```
output order1_ordered := placeOrder__orderId == 1 ||
    order1_ordered.offset(by:-1).defaults(to:false)
output order1_delivered := deliveryMade__orderId == 1 ||
    order1_delivered.offset(by:-1).defaults(to:false)
```

If the buyer places an order, and the order id is equal to 1, the stream is set to true. If the buyer places an order, but the order id is not equal 1, the last value of the stream is repeated. The truth value stays unchanged. The output stream that takes care of the delivery of order id 1 works in the same way.

Now, we can express the specification as a trigger:

```
1 trigger sender_address == seller_address.hold().defaults(to:0) && terminateContract
2     && order1_ordered.hold().defaults(to:false) &&
          !order1_delivered.hold().defaults(to:false)
```

If the seller requested the termination of the contract (line 1), but order 1 was ordered and not delivered (line 2), the specification is violated. Again, this trigger has to be extended for every supported order id.

6.1.2. ERC20 Token Interface

The second example Azzopardi et al. [7] use is the ERC20 Token Interface. This interface specifies a standard for Fungible Tokens. Fungible means, that the tokens are mutually interchangeable i.e. every token is the same. What exactly these tokens represent is arbitrary. They can for example represent the skills of a character in a game or financial assets. [49]

Listing 6.2 shows the functions that are required in the ERC20 Token Interface. The `totalSupply` function returns the total number of tokens that are in the system. A user can query the balance of an account with the `balanceOf` function. When calling the `transfer` function, one can transfer tokens from one's own account to someone else's account. With the `transferFrom` function, it is also possible to transfer tokens that are not on your own account. The restriction is that the participants are only able to

```
function totalSupply() public view returns (uint256)
function balanceOf(address owner) public view returns (uint256 balance)
function transfer(address to, uint256 value) public returns (bool success)
function transferFrom(address from, address to, uint256 value) public returns (bool
    success)
function approve(address spender, uint256 value) public returns (bool success)
function allowance(address owner, address spender) public view returns (uint256 remaining)
```

Listing 6.2: The required functions from the ERC20 Token Interface as specified in [50].

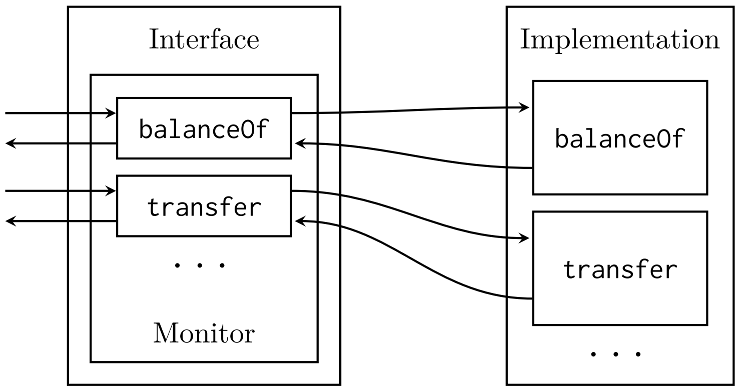


Figure 6.2.: The interface passes all calls through to the implementation. The monitor in the interface checks, that the implementation behaves correctly.

handle that amount of tokens of another account, that the other account allowed them to handle. This can be done with the `approve` function: When calling the `approve` function, one can allow another account to spend a specific amount of tokens from their account. With the `allowance` function, one can query how much one account allowed another account to spend.

The idea Azzopardi et al. [7, 51] proposed is to monitor this interface, but with the ability to exchange the implementation afterward. This first sounds counterintuitive since smart contracts are immutable. The trick that is used here, is that it is possible to call functions in other contracts. Instead of storing the whole implementation in the contract, the contract just stores the address of another contract, which contains the actual code. When someone calls a function in the contracts, the contract passes the call to the external function behind the stored address (see Figure 6.2). Since this address can be changed even after publishing the contract, it is possible to exchange the code that the functions execute.

```
interface ERC20TokenImplementation {
    function totalSupply () external returns (uint);
    function balanceOf (address tokenOwner) external returns (uint balance);
    function transfer (address caller, address to, uint tokens) external returns (bool
        success);
    // ...
}
```

```
contract ERC20Interface{

    ERC20TokenImplementation impl;
    address owner;

    constructor(ERC20TokenImplementation _impl, address _owner) public{
        impl = _impl;
        owner = _owner;
    }

    function updateImplementation(address newImpl) public{
        require(msg.sender == owner);
        impl = ERC20TokenImplementation(newImpl);
    }

    function totalSupply() public returns (uint){
        return impl.totalSupply();
    }

    function balanceOf(address tokenOwner) public returns (uint balance){
        return impl.balanceOf(tokenOwner);
    }

    function transfer(address to, uint tokens) public returns (bool success){
        return impl.transfer(msg.sender, to, tokens);
    }

    // ...
}
```

Listing 6.3: The code for the mutable ERC20 interface ([52], slightly modified and shortened).

See Listing 6.3 for the implementation of the interface from the ContractLarva Github repository [52]. Azzopardi et al. had to modify the interface slightly (i.e. compare `transfer` in Listing 6.3 to Listing 6.2) for passing the sender address of the function as an extra argument. When executing the constructor function, the contract stores two values. The first is the address of the contract, which contains the initial implementation. The constructor also stores the address of the one who is allowed to change the implementation, the owner. This owner can call the `updateImplementation` function, which updates the global variable storing the address to the implementation. Whenever someone calls a function, the call is just forwarded to the corresponding function in the current implementation.

However, it is undesirable that there is no control over what the implementation actually does since there is no guarantee to the users of the contract that the owner does not change the implementation to their disadvantage. To make sure that, independent from the implementation used, the functions still do what they are supposed to do, we add runtime monitoring to the interface. For that, the monitor has to store the current balance of every account. Here, the same problem occurs that we already saw with the ordering system since the number of accounts is not limited. And again, the workaround we use is that the monitor limits the number of accounts to a smaller set. To be able to do that, we have to write the addresses of all supported accounts directly into the specification.

➔ App. A.3, p. 62

Below, we present the most interesting parts of the specification with two accounts. For the full specification, we refer the interested reader to Appendix A.3. This example uses the following input streams:

```
input called_function : String
input sender_address : UInt256

input transfer : Bool
input transfer__to : UInt256
input transfer__tokens : UInt256
input transfer__success : Bool

input transferFrom : Bool
input transferFrom__from : UInt256
input transferFrom__to : UInt256
input transferFrom__tokens : UInt256
input transferFrom__success : Bool

input balanceOf__tokenOwner : UInt256
input balanceOf__balance : UInt256

input approve : Bool
input approve__spender : UInt256
input approve__tokens : UInt256
input approve__success : Bool
```

When the `transfer` function is executed, the monitor stores the arguments `to`, `value` and the return value `success` in the corresponding input streams. The same goes for the `transferFrom` function. When the balance of a user is queried by calling the `balanceOf` function, the monitor stores the address of the queried user in the `balanceOf__owner` input stream, and the returned balance in the `balanceOf__balance` input stream. When allowing another user to handle parts of your tokens by calling the `approve` function, then the monitor adds the arguments and the return value `success` in the corresponding input streams. Since multiple functions can be responsible for the change of a balance, we also need the `called_function` stream, to differentiate how the balance should change. Since a lot of the functions depend on the user that called the function, the specification also includes the `sender_address` input stream. Additionally, the specification contains all the addresses of the supported accounts:

```
output ADDRESS1 := 619237891...
output ADDRESS2 := 718931273...
```

These output streams are constant, they always have the same value.

The monitor needs to keep track of the current balance of every account (here shown for the first address):

```
1  output balance1 @(transfer || transferFrom) :=
2     balance1.offset(by:-1).defaults(to:0) +
3     if called_function.hold().defaults(to:"") == "transfer" &&
          transfer__success.hold().defaults(to:false) then
4        if transfer__to.hold().defaults(to:0) == ADDRESS1.hold().defaults(to:0) then
5           transfer__tokens.hold().defaults(to:0)
6        else if sender_address.hold().defaults(to:0) == ADDRESS1.hold().defaults(to:0)
             then
7           -transfer__tokens.hold().defaults(to:0)
8        else 0
9     else if called_function.defaults(to:"") == "transferFrom" &&
          transferFrom__success.hold().defaults(to:false) then
10       if transferFrom__from.hold().defaults(to:0) == ADDRESS1.hold().defaults(to:0)
             then
11          -transferFrom__tokens.hold().defaults(to:0)
12       else if sender_address.hold().defaults(to:0) == ADDRESS1.hold().defaults(to:0)
             then
13          transferFrom__tokens.hold().defaults(to:0)
14       else 0
15    else 0
```

Due to the activation condition, this stream always receives a new value when someone calls the `transfer` or `transferFrom` function. If the stream is updated because the `transfer` function was executed successfully (line 3), the monitor evaluates the upper block (line 4 to 8). If the transfer is towards account 1, then the monitor adds the transferred tokens to the current balance (line 5). If the transfer is from account 1, because account 1 is the one calling the function, then the monitor subtracts the tokens from the balance of account 1 (line 7). If the stream is updated because the `transferFrom` function was executed successfully (line 9), the monitor evaluates the lower block (line 10 to 14). If none of this is the case, for example, because the function was not successful and returned `false`, the balance does not change at all (line 15).

Besides that, the monitor has to record the current allowance from every account to every other account (here shown for the amount of tokens that account 1 allowed account 2 to handle):

```
output allowance1_to_2 :=
   if sender_address == ADDRESS1 && approve__spender == ADDRESS2 && approve__success then
      approve__tokens
   else
      allowance1_to_2.offset(by:-1).defaults(to:0)
```

If the account with `ADDRESS1` successfully called the `approve` function with `spender` being `ADDRESS2`, then the monitor sets the `approve_tokens` as the new value. Otherwise, the `approve` call does not influence the allowance from account 1 to account 2 and the monitor repeats the previous value.

Next, the specification defines a trigger to make sure that `balanceOf` always returns the correct balance:

```
trigger balanceOf__owner == ADDRESS1 && balanceOf__balance !=
    balance1.hold().defaults(to:0) "trigger1: wrong balance for account 1"
```

If the queried account's address is `ADDRESS1`, but the returned value does not equal to the current `balance1`, that is a violation of the specification. Also, `transferFrom` can only be used if the allowance is high enough:

```
trigger transferFrom__from == ADDRESS1 && sender_address == ADDRESS2 &&
    allowance1_to_2.hold().defaults(to:0) < transferFrom__tokens "trigger2: allowance
    from 1 to 2 is not high enough for the transfer"
```

If `ADDRESS2` tries to transfer tokens from `ADDRESS1`, but the allowance is not high enough, then that is also a violation of the specification.

6.2. Evaluation of the Gas Usage

We measure the gas usage of the example contracts by writing unit tests and then measuring the gas every function call uses. The unit tests are written in Javascript and executed with Truffle [53]. Truffle is a set of tools for developing and testing Solidity smart contracts. We use Ganache-time-traveler [54] to allow testing the time-relevant parts of the specification. The unit tests check whether the contracts fulfill the specification.

Listing 6.4 shows a test checking that the order contract can not be terminated by the buyer before the minimum number of items have been ordered. The buyer's request for termination of the contract (line 15) is a violation of the specification since the minimal amount of items that have to be ordered is 2, but the buyer did not order any at all. Because of that, an exception is expected since the monitor should call the trigger function which executes `revert()`. If the monitor does not raise the expected exception, the test fails with `assert(false)` (line 19).

To measure the gas usage, we use the eth-gas-reporter [55]. This tool measures the gas usage of every function called in the tests and produces a report in the end (see Figure 6.3). We are interested in the average gas usage of every function, to compare that with the gas usage of the monitored contracts.

We evaluate the gas usage of the ordering system with two different contracts and compare the gas usage with the monitored versions of these. The first one (Version 1) is the contract shown in Listing 6.1, where the contract does not check the specification at all. This contract does of course not pass the unit tests, but we can measure the gas usage nevertheless. In the second contract (Version 2), we modified Version 1 by hand in a way, so that the contract also checks all of the specifications. In the contract of Version 1 with included monitor (Monitored 1), we leave checking the specification completely to the monitor. In the monitored Version 2 (Monitored 2), the monitor verifies that the specification is correctly implemented in the contract. In case there is an error in the hand-implemented specification in Version 2, the monitor finds and handles that error.

➔Lst. 6.1, p. 39

Adding the monitoring to Version 1 (see Table 6.4) introduces a lot of overhead. The reason for that is that the monitor does all of the actual work of the contract. The unmonitored version just provides the functions to call but does not check the

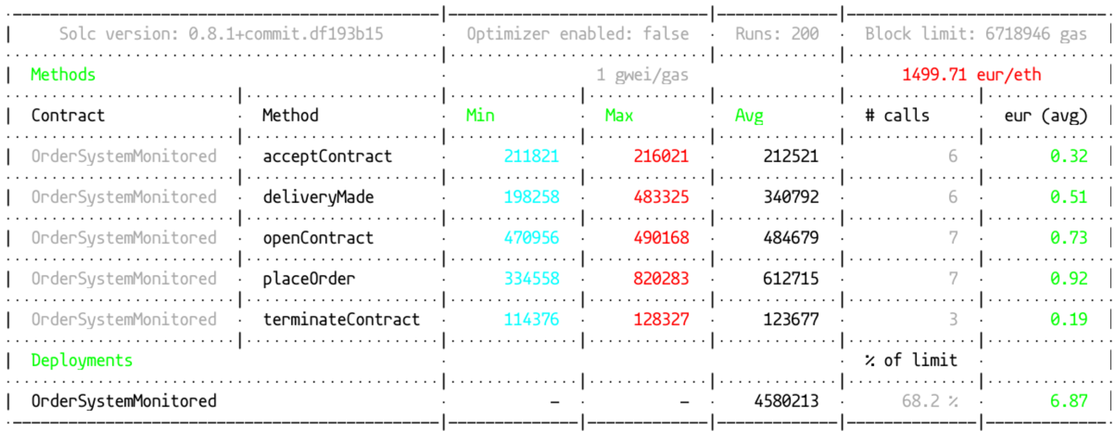
```
it('can not get closed by buyer before minimum Items', async () => {
    orderSystemInstance = await OrderSystem.deployed();
    let seller = accounts[0];
    let buyer = accounts[1];
    let minItems = 2;
    let price = ...

    await orderSystemInstance.openContract(endDate,price,minItems,maxItems, {from:
         seller, value: performance_guarantee});
    await orderSystemInstance.acceptContract({from: buyer, value: escrow});

    // three days later after end date
    timeMachine.advanceTimeAndBlock(3*24*60*60);

    try {
        await orderSystemInstance.terminateContract({from: buyer});
    } catch (e) {
        return
    }
    assert(false);
});
```

Listing 6.4: A unit test for the ordering system testing that the buyer can not terminate the contract before the minimum number of items have been ordered.



```
·------------------------------------------|---------------------------|-------------|---------------------------·
|    Solc version: 0.8.1+commit.df193b15   ·  Optimizer enabled: false ·  Runs: 200  ·  Block limit: 6718946 gas │
···········································|···························|·············|····························
|  Methods                                 ·               1 gwei/gas                ·      1499.71 eur/eth      │
······················|····················|·············|·············|·············|··············|·············
|  Contract           ·  Method            ·  Min        ·  Max        ·  Avg        ·  # calls     ·  eur (avg) │
······················|····················|·············|·············|·············|··············|·············
|  OrderSystemMonitored  ·  acceptContract     ·     211821  ·     216021  ·     212521  ·           6  ·       0.32 │
······················|····················|·············|·············|·············|··············|·············
|  OrderSystemMonitored  ·  deliveryMade       ·     198258  ·     483325  ·     340792  ·           6  ·       0.51 │
······················|····················|·············|·············|·············|··············|·············
|  OrderSystemMonitored  ·  openContract       ·     470956  ·     490168  ·     484679  ·           7  ·       0.73 │
······················|····················|·············|·············|·············|··············|·············
|  OrderSystemMonitored  ·  placeOrder         ·     334558  ·     820283  ·     612715  ·           7  ·       0.92 │
······················|····················|·············|·············|·············|··············|·············
|  OrderSystemMonitored  ·  terminateContract  ·     114376  ·     128327  ·     123677  ·           3  ·       0.19 │
······················|····················|·············|·············|·············|··············|·············
|  Deployments                             ·                                         ·  % of limit  ·            │
···········································|·············|·············|·············|··············|·············
|  OrderSystemMonitored                    ·          -  ·          -  ·    4580213  ·      68.2 %  ·       6.87 │
·------------------------------------------|-------------|-------------|-------------|--------------|-------------·
```

Figure 6.3.: Report generated by eth-gas-reporter. For each function in the contract that was called in any test, the report prints the minimal, maximal, and average gas usage. The report also includes the deployment costs of the contracts.

Function	Version 1	Monitored 1	Overhead	
Deployment	258359	4487272	4228913	1636,84%
`openContract`	64292	454150	410724	945,80%
`acceptContract`	47981	181220	154105	568,34%
`placeOrder`	87981	584117	519959	810,44%
`deliveryMade`	43505	328345	285748	670,82%
`terminateContract`	28025	133887	106770	393,74%

Table 6.4.: Gas usage overhead of the ordering system. Version 1 is the implementation shown in Listing 6.1 where the contract does not check the specification at all.

Function	Version 2	Monitored 2	Overhead	
Deployment	879075	5062544	4183469	475,89%
`openContract`	140589	551326	410737	292,15%
`acceptContract`	70737	224853	154116	217,87%
`placeOrder`	108176	628736	520560	481,22%
`deliveryMade`	63602	348650	285048	448,17%
`terminateContract`	59657	166432	106775	178,98%

Table 6.5.: Gas usage of the ordering system where the specification is also implemented in the original contract and the monitoring is just used in case of errors in the implementation.

Function	Original	Monitored	Overhead		Larva	Overhead	
Deployment	2051548	6693611	4642063	226,27%	2525349	473801	23,09%
`updateImplementation`	43404	43404	0	0,00%	44365	961	2,21%
`totalSupply`	25830	25830	0	0,00%	26741	911	3,53%
`balanceOf`	25508	123527	98019	384,27%	26419	911	3,57%
`allowance`	26962	203105	176143	653,30%	27873	911	3,38%
`approve`	46161	639802	593641	1286,02%	56356	10195	22,09%
`transfer`	45062	210205	165143	366,48%	51311	6249	13,87%
`transferFrom`	46511	310096	263585	566,72%	71570	25059	53,88%

Table 6.6.: Gas usage of the mutable ERC20 interface. The original contract is the interface without any monitoring. The monitored contract adds our monitoring approach to the interface with support for four accounts. The third contract adds the monitoring approach from ContractLarva to the original contract.

specification at all. The difference is visible when comparing the gas usage of Version 1 to the gas usage of Version 2 (see Table 6.5). There the gas usage also increases, even if a lot less as through adding the monitoring. Since the gas usage of Version 2 is higher than that of Version 1, the percentage overhead of introducing monitoring to Version 2 is much smaller than introducing monitoring to Version 1.

For the evaluation of the monitorable ERC20 interface, we use the interface and implementation used by Azzopardi et al. [56] for evaluating ContractLarva (updated for a newer version of the Solidity compiler). We compare the gas usage of the interface without any monitoring to the gas usage of the interface with our monitoring approach and the gas usage of the interface with the ContractLarva monitoring approach. All interfaces use the same implementation. Table 6.6 depicts the gas usage of the different contracts. The deployment costs in the table all include the 1419669 gas for deploying the implementation. The RTLola specification used in this evaluation allows for four different accounts. To support these four accounts, the specification needs a lot of output streams. Especially with the allowance streams, where every account has to have an allowance for every other account. This results in a lot of duplicated code, that would not be necessary with parametrized streams. Because of that, we also measured the gas usage of the monitor with only one supported stream (see Table 6.7). These gas usages are a lot smaller and may be more comparable to that of a monitor with parametrized streams.

The way our approach and ContractLarva check the specification is different. In our approach, the monitor keeps track of what the correct balance should be. If someone then queries the current balance, the returned value is compared with the expected balance. ContractLarva on the other hand does not store the expected balance in the monitor.

Function	Monitored	Overhead	
Deployment	3023494	971946	47,38%
`updateImplementation`	43404	0	0,00%
`totalSupply`	25830	0	0,00%
`balanceOf`	106263	80755	316,59%
`allowance`	159541	132579	491,73%
`approve`	145787	99626	215,82%
`transfer`	99059	53997	119,83%
`transferFrom`	158896	112385	241,63%

Table 6.7.: The gas usage of the monitored ERC20 Token Interface with only one account.

There, the monitor queries the balance directly after a transaction and checks that the balance changed accordingly. This difference is also visible in the gas measurements. ContractLarva does not really have any overhead for any of the getter functions, since everything is checked directly after a transfer. One problem with this approach is that a malicious owner of the contract could change the implementation in a way, that the value that `balanceOf` returns is not consistent. When the monitor calls `balanceOf` after a transfer, the malicious implementation could return the expected balance, but when the user calls the `balanceOf` function, the implementation could return a wrong balance. With ContractLarva, this behavior would not trigger a violation of the specification.

Chapter 7

Conclusion

This thesis implements runtime monitoring for smart contracts by using the stream-based monitoring language RTLola. We presented a compiler that takes a Solidity smart contract as an input together with a RTLola specification, and then produces a smart contract with an included monitor. As long as the specification is not violated, the output contract behaves exactly the same as the input contract, except that a monitor checks the specification during the runtime of the contract. Once a violation of the specification occurs, the monitor executes user-defined code to handle the error. We chose this design to accommodate for the fact that the monitor has to be able to handle the error by itself since smart contracts are immutable and the developer can not fix the errors after the contract is added to the blockchain. To handle a violation of the specification, the monitor reverts the faulty function call or brings the contract into a specific state.

First, we showed how to implement input streams, output streams, and triggers in Solidity. We also presented how the monitor adds new values to these streams whenever someone calls a function in the contract. The monitor has to know how many values of each stream it has to keep. This number is given by the storage requirement of each stream. Likewise, the monitor has to know which streams it has to update when a specific function was called and in which order these streams have to be evaluated. This information can be retrieved from the dependency graph of the specification and the activation conditions of each stream during the compilation. Debugging of the output contract produced by the compiler was challenging. We handled this by providing debugging messages from the compiler, including debugging functions in the output contract, and by writing unit tests for the output contracts. In the evaluation we found that we need parametrized streams to express the specifications. Since parametrized streams are not yet supported by the RTLola Frontend, we used a workaround. Therefore, we limited the number of possible parameters and added a separate stream for each parameter.

We evaluated our monitoring approach with two example contracts by measuring the gas usage of every function. Our monitoring approach introduced a non-negligible overhead on the gas usage. Nevertheless, one may be willing to pay that overhead instead of losing even more money due to a bug in the contract. The gas usage overhead of ContractLarva is a lot less than that of our approach. On the other hand is it more natural to write a specification as stream-expressions, instead of having to think of states and transitions. This also makes it less likely to introduce an error in the specification itself. Besides that, our approach allows monitoring of function's return values, which is not possible with ContractLarva.

Future Work

In the evaluation, we saw that the compiler would benefit from parametrized streams [9, 57]. To implement a parametrized stream in Solidity, instead of using a simple array for storing the values of a stream, we could use a mapping to arrays. The update function, as well as the access functions for a parametrized stream would then in addition also receive the parameter of the stream. It would be interesting to see how the gas usage overhead behaves after implementing parametrized streams. This could reduce the gas usage overhead introduced by the monitoring a lot.

Currently, not the full RTLola language is supported by the compiler. To add the real-time aspects of RTLola to our approach, every value in a buffer would also has to have a timestamp associated with it. Every time the monitor adds a new value to a stream, it would also add the current time to the value. That the monitor is only able to update streams when someone called a function poses a challenge. Because of that, the monitor has to evaluate periodic streams and sliding windows retrospectively, whenever a function is called.

Besides supporting the full RTLola language, it may also be an option to differentiate more between the user code and the monitor. Currently the monitored functions the user calls are part of the monitor. It may be more desirable to completely separate the monitor from the contract and only interact with it through an API.

Appendix A

Appendix

A.1. Idea Example Output Contract

The code produced by the compiler for the contract and the specification in Listings 4.1 and 4.2:

```
contract example {
   // --- CONTRACT ---------------

   int256 last_arg1;

   function _example_function(int256 arg1, int32 arg2)
      private
      returns (bool equal, int256 sum)
   {
      equal = arg1 == last_arg1;
      last_arg1 = arg1;
      return (equal, arg1 + int256(arg2));
   }

   function example_function(int256 arg1, int32 arg2)
      public
      returns (bool equal, int256 sum)
   {
      (bool return0, int256 return1) = _example_function(arg1, arg2);

      // input stream shifts
      shift_example_function__arg1();

      // input stream updates
      update_example_function__arg1(arg1);
      update_example_function__arg2(arg2);
      update_example_function__equal(return0);
      update_example_function__sum(return1);

      // output stream updates
```

```
        update_sum();

        // trigger checks
        check_trigger0();
        check_trigger1();

        return (return0, return1);
    }

    // --- TRIGGER ----------------

    function wrong_bool() private {
        // Trigger #0: "the return boolean does not correspond to the equality of the
            argument and the previous argument"
        // !! -- fill with custom code -- !!
        revert("wrong_bool");
    }

    function wrong_sum() private {
        // Trigger #1: "the return value does not correspond to the sum of the arguments"
        // !! -- fill with custom code -- !!
        revert("wrong_sum");
    }

    // --- MONITOR ----------------

    // -- input streams ----------

    int256[2] example_function__arg1_buffer;
    bool[2] example_function__arg1_valid;
    uint256 example_function__arg1_current;

    function shift_example_function__arg1() private {
        example_function__arg1_current =
            (example_function__arg1_current + 1) %
            2;
    }

    function get_example_function__arg1_with_offset(uint256 offset, int256 def)
        private
        view
        returns (int256)
    {
        int256 index =
            (int256(example_function__arg1_current) - int256(offset)) % 2;
        if (index < 0) {
            index += 2;
        }
        if (example_function__arg1_valid[uint256(index)]) {
            return example_function__arg1_buffer[uint256(index)];
        } else {
            return def;
        }
```

```
}

function update_example_function__arg1(int256 example_function__arg1)
    private
{
    example_function__arg1_buffer[
        example_function__arg1_current
    ] = example_function__arg1;
    example_function__arg1_valid[example_function__arg1_current] = true;
}

// ------

int32[1] example_function__arg2_buffer;
bool[1] example_function__arg2_valid;

function update_example_function__arg2(int32 example_function__arg2)
    private
{
    example_function__arg2_buffer[0] = example_function__arg2;
    example_function__arg2_valid[0] = true;
}

// ------

bool[1] example_function__equal_buffer;
bool[1] example_function__equal_valid;

function update_example_function__equal(bool example_function__equal)
    private
{
    example_function__equal_buffer[0] = example_function__equal;
    example_function__equal_valid[0] = true;
}

// ------

int256[1] example_function__sum_buffer;
bool[1] example_function__sum_valid;

function update_example_function__sum(int256 example_function__sum)
    private
{
    example_function__sum_buffer[0] = example_function__sum;
    example_function__sum_valid[0] = true;
}

// -- output streams ---------

int256[1] sum_buffer;
bool[1] sum_valid;

function update_sum() private {
        sum_buffer[0] = (example_function__arg1_buffer[
            example_function__arg1_current
        ] + (int256(example_function__arg2_buffer[0])));
        sum_valid[0] = true;
    }

    // -- trigger checks --------

    function check_trigger0() private {
        if (
            ((example_function__arg1_buffer[example_function__arg1_current] ==
                get_example_function__arg1_with_offset(1, (int256(0)))) !=
                example_function__equal_buffer[0])
        ) {
            wrong_bool();
        }
    }

    // ------

    function check_trigger1() private {
        if ((sum_buffer[0] != example_function__sum_buffer[0])) {
            wrong_sum();
        }
    }
}
```

A.2. Ordering System Specification

The full RTLola specification for the ordering system depicted as natural language in Figure 6.1 with one supported order id:

```
input sender_address : UInt256
input current_time : UInt256
input attached_value : UInt256

input openContract : Bool
input openContract__endDate : UInt256
input openContract__price : UInt256
input openContract__minimumItems : UInt256
input openContract__maximumItems : UInt256

input acceptContract : Bool

input placeOrder : Bool
input placeOrder__orderId : UInt256
input placeOrder__itemsOrdered : UInt256
input placeOrder__timeOfDelivery : UInt256

input deliveryMade : Bool
input deliveryMade__orderId : UInt256

input terminateContract : Bool

output PERFORMANCE_GUARANTEE := 1000

output seller_address @(openContract && sender_address) := sender_address
output buyer_address @(acceptContract && sender_address) := sender_address

output buyer_escrow @(acceptContract && attached_value) := attached_value

output order1_amount :=
    if placeOrder__orderId == 1 then placeOrder__itemsOrdered
    else order1_amount.offset(by:-1).defaults(to:0)

output order1_timeOfDelivery :=
    if placeOrder__orderId == 1 then placeOrder__timeOfDelivery
    else order1_timeOfDelivery.offset(by:-1).defaults(to:0)

output order1_ordered := placeOrder__orderId == 1 ||
     order1_ordered.offset(by:-1).defaults(to:false)

output order1_delivered := deliveryMade__orderId == 1 ||
     order1_delivered.offset(by:-1).defaults(to:false)

output orders_ordered @(placeOrder) := orders_ordered.offset(by:-1).defaults(to:0) + 1
output orders_delivered @(deliveryMade) := orders_delivered.offset(by:-1).defaults(to:0)
     + 1
```

```
output items_ordered := items_ordered.offset(by:-1).defaults(to:0) +
     placeOrder__itemsOrdered
output items_delivered := items_delivered.offset(by:-1).defaults(to:0) +
    if deliveryMade__orderId == 1 then order1_amount.hold().defaults(to:0)
    else 0

output items_pending @(placeOrder || deliveryMade):= items_ordered.hold().defaults(to:0)
     - items_delivered.hold().defaults(to:0)
output orders_pending @(placeOrder || deliveryMade) :=
    if order1_ordered.hold().defaults(to:false) &&
        !order1_delivered.hold().defaults(to:false) then 1 else 0

output orders_delivered_late :=
    if deliveryMade__orderId == 1 && current_time >
        order1_timeOfDelivery.hold().defaults(to:0) then
       orders_delivered_late.offset(by:-1).defaults(to:0) + 1
    else
       orders_delivered_late.offset(by:-1).defaults(to:0)

// 1
trigger openContract && terminateContract.hold().defaults(to:false) "trigger1: once
     closed, the contract can not get opened again"
trigger acceptContract && !openContract.hold().defaults(to:false) "trigger1: the contract
     can only get accepted after being opened"
trigger placeOrder && !acceptContract.hold().defaults(to:false) "trigger1: no orders are
     possible before opening the contract"

// 2
trigger terminateContract && sender_address == buyer_address.hold().defaults(to:0)
    && items_ordered.hold().defaults(to:0) <
        openContract__minimumItems.hold().defaults(to:0) "trigger1: the contract can only
        get terminated by the buyer if the minimum amount of items were ordered"
trigger items_ordered > openContract__maximumItems.hold().defaults(to:0) "trigger1: the
     buyer can order at most maximumItems"

// 3
trigger terminateContract && openContract__endDate.hold().defaults(to:0) > current_time
     "trigger1: the contract can only get terminated after endDate"
trigger terminateContract && sender_address == seller_address.hold().defaults(to:0) &&
     orders_pending.hold().defaults(to:0) > 0 "trigger1: contract can not get closed by
     seller if there are pending orders"

// 4
trigger acceptContract && attached_value < openContract__price.hold().defaults(to:0) *
     openContract__minimumItems.hold().defaults(to:0) "trigger1: buyer has to pay escrow
     for the minimum items when accepting the contract"

// 5
trigger openContract && attached_value != PERFORMANCE_GUARANTEE "trigger1: seller has to
     pay performance guarantee when opening the contract"

// 7
```

```
trigger placeOrder__timeOfDelivery < current_time + 86400 "trigger1: time of delivery has
    to be at least 24 hours later"

// 8
trigger items_pending * openContract__price.hold().defaults(to:0) >
    buyer_escrow.hold().defaults(to:0) "trigger1: buyer can only place orders if enough
    money is in escrow"

// 9
trigger deliveryMade__orderId == 1 && attached_value !=
    openContract__price.hold().defaults(to:0) * order1_amount.hold().defaults(to:0)
    "trigger1: upon delivery of order 1 the seller receives payment of the order"

// 11
output performance_guarantee_return @(terminateContract) :=
   orders_ordered.hold().defaults(to:0) < orders_delivered.hold().defaults(to:0)
   || orders_delivered_late.hold().defaults(to:0) * 4 >
       orders_delivered.hold().defaults(to:0)
trigger performance_guarantee_return "pay_buyer: buyer gets performance guarantee"
trigger !performance_guarantee_return "pay_seller: seller gets performance guarantee"
```

A.3. ERC20 Token System Specification

The full RTLola specification for the ERC20 token interface depicted in Listing 6.3 with two supported accounts:

```
input sender_address : UInt256
input called_function : String

input transfer : Bool
input transfer__to : UInt256
input transfer__tokens : UInt256
input transfer__success : Bool

input transferFrom : Bool
input transferFrom__from : UInt256
input transferFrom__to : UInt256
input transferFrom__tokens : UInt256
input transferFrom__success : Bool

input balanceOf__tokenOwner : UInt256
input balanceOf__balance : UInt256

input approve : Bool
input approve__spender : UInt256
input approve__tokens : UInt256
input approve__return0 : Bool

input allowance__tokenOwner : UInt256
input allowance__spender : UInt256
input allowance__return0 : UInt256

output ADDRESS1 := 588645899264321211530317558446108028729426462147
output ADDRESS2 := 704938353656261341379910710004063475570337743756

output balance1 @(transfer || transferFrom):=
    balance1.offset(by:-1).defaults(to:0) +
    if called_function.hold().defaults(to:"") == "transfer" &&
        transfer__success.hold().defaults(to:false) then
       if transfer__to.hold().defaults(to:0) == ADDRESS1.hold().defaults(to:0) then
           transfer__tokens.hold().defaults(to:0)
       else if sender_address.hold().defaults(to:0) == ADDRESS1.hold().defaults(to:0) then
           -transfer__tokens.hold().defaults(to:0)
       else 0
    else if called_function.hold().defaults(to:"") == "transferFrom" &&
        transferFrom__success.hold().defaults(to:false) then
       if transferFrom__from.hold().defaults(to:0) == ADDRESS1.hold().defaults(to:0) then
           -transferFrom__tokens.hold().defaults(to:0)
       else if sender_address.hold().defaults(to:0) == ADDRESS1.hold().defaults(to:0) then
           transferFrom__tokens.hold().defaults(to:0)
       else 0
    else 0

output balance2 @(transfer || transferFrom) :=
    balance2.offset(by:-1).defaults(to:0) +
    if called_function.hold().defaults(to:"") == "transfer" &&
         transfer__success.hold().defaults(to:false) then
        if transfer__to.hold().defaults(to:0) == ADDRESS2.hold().defaults(to:0) then
           transfer__tokens.hold().defaults(to:0)
        else if sender_address.hold().defaults(to:0) == ADDRESS2.hold().defaults(to:0) then
           -transfer__tokens.hold().defaults(to:0)
        else 0
    else if called_function.hold().defaults(to:"") == "transferFrom" &&
         transferFrom__success.hold().defaults(to:false) then
        if transferFrom__from.hold().defaults(to:0) == ADDRESS2.hold().defaults(to:0) then
           -transferFrom__tokens.hold().defaults(to:0)
        else if sender_address.hold().defaults(to:0) == ADDRESS2.hold().defaults(to:0) then
           transferFrom__tokens.hold().defaults(to:0)
        else 0
    else 0

output allowance1_to_2 :=
    if sender_address == ADDRESS1 && approve__spender == ADDRESS2 && approve__return0 then
        approve__tokens
    else
        allowance1_to_2.offset(by:-1).defaults(to:0)

output allowance2_to_1 :=
    if sender_address == ADDRESS2 && approve__spender == ADDRESS1 && approve__return0 then
        approve__tokens
    else
        allowance2_to_1.offset(by:-1).defaults(to:0)

trigger balanceOf__tokenOwner == ADDRESS1.hold().defaults(to:0) && balanceOf__balance !=
     balance1.hold().defaults(to:0) "trigger1: balance of 1 is wrong"
trigger balanceOf__tokenOwner == ADDRESS2.hold().defaults(to:0) && balanceOf__balance !=
     balance2.hold().defaults(to:0) "trigger1: balance of 2 is wrong"

trigger transferFrom__from == ADDRESS1 && transferFrom__to == ADDRESS2 &&
     allowance1_to_2.hold().defaults(to:0) < transferFrom__tokens "trigger1:
     allowance1_to_2 not high enough"
trigger transferFrom__from == ADDRESS2 && transferFrom__to == ADDRESS1 &&
     allowance2_to_1.hold().defaults(to:0) < transferFrom__tokens "trigger1:
     allowance2_to_1 not high enough"

trigger allowance__tokenOwner == ADDRESS1 && allowance__spender == ADDRESS2 &&
     allowance__return0 != allowance1_to_2.hold().defaults(to:0) "trigger1:
     allowance1_to_2 wrong"
trigger allowance__tokenOwner == ADDRESS2 && allowance__spender == ADDRESS1 &&
     allowance__return0 != allowance2_to_1.hold().defaults(to:0) "trigger1:
     allowance2_to_1 wrong"
```

Bibliography

[1] Bhabendu Kumar Mohanta, Soumyashree S Panda, and Debasish Jena. 2018. An overview of smart contract and use cases in blockchain technology. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. IEEE, 1–4.

[2] Michael del Castillo. 2016. The DAO attacked: Code Issue Leads to $60 Million Ether Theft. [Accessed: 24.11.20]. `https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft`.

[3] 2017. Code bug freezes $150m of Ethereum crypto-cash. [Accessed: 08.03.21]. `https://www.bbc.com/news/technology-41928147`.

[4] Ethereum Foundation. 2020. Solidity v7.5 documentation. `https://docs.soliditylang.org/en/v0.7.5/`.

[5] Vitalik Buterin. 2013. Ethereum Whitepaper. `https://ethereum.org/en/whitepaper`.

[6] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. 2019. Streamlab: Stream-based monitoring of cyber-physical systems. In *International Conference on Computer Aided Verification*. Springer, 421–431.

[7] Shaun Azzopardi, Joshua Ellul, and Gordon J Pace. 2018. Monitoring smart contracts: Contractlarva and open challenges beyond. In *International Conference on Runtime Verification*. Springer, 113–137.

[8] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. Lola: Runtime Monitoring of Synchronous Systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*. IEEE Computer Society Press, Burlington, Vermont, (June 2005), 166–174.

[9] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. 2016. A Stream-Based Specification Language for Network Monitoring. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings* (Lecture Notes in Computer Science). Yliès Falcone and César Sánchez, editors. Volume 10012. Springer, 152–168. DOI: `10.1007/978-3-319-46982-9_10`. `http://dx.doi.org/10.1007/978-3-319-46982-9_10`.

[10] Jan Baumeister, Bernd Finkbeiner, Sebastian Schirmer, Maximilian Schwenger, and Christoph Torens. 2020. RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft. *arXiv preprint arXiv:2004.06488*.

[11] Christoph Torens, Florian Adolf, Peter Faymonville, and Sebastian Schirmer. 2017. Towards Intelligent System Health Management using Runtime Monitoring. In *AIAA Information Systems-AIAA Infotech @ Aerospace*. American Institute of Aeronautics and Astronautics (AIAA), (January 2017). DOI: `10.2514/6.2017-0419`. `https://doi.org/10.2514%2F6.2017-0419`.

[12] Florian-Michael Adolf, Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Christoph Torens. 2017. Stream runtime monitoring on UAS. In *International Conference on Runtime Verification*. Springer, 33–49.

[13] Marvin Hofman. 2018. *Runtime Verification of Critical Web-based Systems with Lola*. Bachelor Thesis. Saarland University.

[14] Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. 2019. FPGA Stream-Monitoring of Real-Time Properties. *ACM Trans. Embed. Comput. Syst.*, 18, 5s, (October 2019). ISSN: 1539-9087. DOI: `10.1145/3358220`. `https://doi.org/10.1145/3358220`.

[15] Bernd Finkbeiner, Stefan Oswald, Noemi Passing, and Maximilian Schwenger. 2020. Verified rust monitors for lola specifications. In *International Conference on Runtime Verification*. Springer, 431–450.

[16] Doron Drusinsky. 3000. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*. Klaus Havelund, John Penix, and Willem Visser, editors. Springer Berlin Heidelberg, Berlin, Heidelberg, 323–330. ISBN: 978-3-540-45297-3.

[17] Klaus Havelund and Grigore Roşu. 2001. Monitoring java programs with java pathexplorer. *Electronic Notes in Theoretical Computer Science*, 55, 2, 200–217.

[18] Klaus Havelund and Grigore Roşu. 2004. An overview of the runtime verification tool Java PathExplorer. *Formal methods in system design*, 24, 2, 189–215.

[19] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Scilla: a smart contract intermediate-level language. *arXiv preprint arXiv:1801.00687*.

[20] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. 1997. The Coq proof assistant reference manual: Version 6.1. `https://coq.inria.fr/`.

[21] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, 91–96.

[22] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 256–270.

[23] Yoichi Hirai. 2018. Bamboo: a language for morphing smart contracts. `https://github.com/pirapira/bamboo`.

[24] Ethereum Foundation. 2020. Vyper documentation. `https://vyper.readthedocs.io/`.

[25] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. 2018. Writing safe smart contracts in Flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, 218–219.

[26] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 67–82.

[27] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. 1989. What you always wanted to know about Datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1, 1, 146–166.

[28] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 254–269.

[29] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *NDSS*.

[30] Rishav Chatterjee and Rajdeep Chatterjee. 2017. An overview of the emerging technology: Blockchain. In *2017 3rd International Conference on Computational Intelligence and Networks (CINE)*. IEEE, 126–127.

[31] Stuart Haber and W Scott Stornetta. 1990. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography*. Springer, 437–455.

[32] Music Buisness Worldwide. 2017. ASCAP, PRS and SACEM join forces for blockchain copyright system. Retrieved 03/17/2021 from `https://www.musicbusinessworldwide.com/ascap-prs-sacem-join-forces-blockchain-copyright-system/`.

[33] Kim S. Nash. 2018. Walmart Requires Lettuce, Spinach Suppliers to Join Blockchain. Retrieved 03/17/2021 from https://blogs.wsj.com/cio/2018/09/24/walmart-requires-lettuce-spinach-suppliers-to-join-blockchain/.

[34] Martin Westerkamp, Friedhelm Victor, and Axel Küpper. 2020. Tracing manufacturing processes using blockchain-based token compositions. *Digital Communications and Networks*, 6, 2, 167–176.

[35] Friðrik Þ Hjálmarsson, Gunnlaugur K Hreiðarsson, Mohammad Hamdaqa, and Gísli Hjálmtysson. 2018. Blockchain-based e-voting system. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 983–986.

[36] Sara Castellanos. 2020. A cryptocurrency Technology Finds New Use Tackling Coronavirus. Retrieved 03/17/2021 from https://www.wsj.com/articles/a-cryptocurrency-technology-finds-new-use-tackling-coronavirus-11587675966.

[37] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system.

[38] EthHub. 2019. Ethereum roadmap - proof of stakes. https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/proof-of-stake.

[39] Alex de Vries. 2020. Bitcoin's energy consumption is underestimated: A market dynamics approach. *Energy Research & Social Science*, 70, 101721. ISSN: 2214-6296. DOI: https://doi.org/10.1016/j.erss.2020.101721. http://www.sciencedirect.com/science/article/pii/S2214629620302966.

[40] Ethereum Foundation. 2020. Upgrading Ethereum to radical new heights. https://ethereum.org/eth2.

[41] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First Monday*.

[42] ChainTrade. 2017. 10 Advantages of Using Smart Contracts. Retrieved 03/03/2021 from https://medium.com/@ChainTrade/10-advantages-of-using-smart-contracts-bc29c508691a.

[43] Maximilian Schwenger. 2019. *Let's not Trust Experience Blindly: Formal Monitoring of Humans and other CPS*. Master Thesis. Saarland University.

[44] pest-parser. 2021. Pest. the Elegant Parser. https://pest.rs.

[45] Jan Baumeister, Florian Kohn, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, and Leander Tentrup. 2020. RTLola Frontend. https://crates.io/crates/rtlola-frontend.

[46] Vincent Prouillet. 2021. A powerful, easy to use template engine for rust. https://tera.netlify.app.

[47] prettier-solidity. 2021. Prettier-plugin-solidity. https://github.com/prettier-solidity/prettier-plugin-solidity.

[48] prettier. 2021. Opinionated code formatter. https://github.com/prettier/prettier.

[49] 2021. Erc-20 token standard. https://ethereum.org/en/developers/docs/standards/tokens/erc-20/.

[50] 2015. Eip-20: erc-20 token standard. (November 2015). https://eips.ethereum.org/EIPS/eip-20.

[51] Christian Colombo, Joshua Ellul, and Gordon J Pace. 2018. Contracts over smart contracts: Recovering from violations dynamically. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 300–315.

[52] J. Pace Gordon, Joshua Ellul, and Shaun Azzopardi. 2020. Contractlarva: Runtime Verification of Solidity smart Contracts. https://github.com/gordonpace/contractLarva.

[53] ConsenSys Software Inc. 2021. Truffle. https://www.trufflesuite.com/.

[54] Ethan Wessel. 2020. Ganache-time-traveler. https://github.com/ejwessel/GanacheTimeTraveler.

[55] Christopher Gewecke. 2021. Eth-gas-reporter. https://github.com/cgewecke/eth-gas-reporter.

[56] Shaun Azzopardi. 2019. Safely-mutable-erc-20-interface. https://github.com/shaunazzopardi/safely-mutable-ERC-20-interface.

[57] Florian Kohn. 2019. *A Stream-based Approach to Network Intrusion Detection*. Bachelor Thesis. Saarland University.