Advanced Research & Development Center

Project S014
Automatic Formal Verification of Smart-Contracts

FLeX System
Functional Formal Verification
Technical Report

Authors
Evgeny Shishkin
evgeny.shishkin@infotecs.ru

Moscow
December, 2021

Contents

1 Introduction

In this document, we present a technical report on the function formal verification of FLeX trading system.

1.1 Overview

During our verification effort, we have discovered 9 defects, some of them are critical.[1]

All found defects were reported to the developers; for each defect we opened a separate issue on GitHub [2]. All issues were confirmed. We describe all our findings in Section 8.4.

The work also contributes in the following ways:

1. We provide a concise mathematical model of the TON smart-contracts runtime. It might be useful both for better understanding and for building reasoning formalism specifically tailored for the TON blockchain 5

2. We deliver TVF - TON Verification Framework - a set of libraries for Dafny programming and verification language. The framework allows rapid verification of TON smart-contracts, both on functional-level and targeting safety properties 7

3. FLeX smart-contracts encoded in the TVF, equipped with specifications and annotations. Proofs are generated automatically in most cases. 8.2

4. We prove strict lower and upper bounds on the number of generated messages by the system core smart-contract Price. Building those proofs helped us to discover several critical issues.

To the best of our knowledge, this work is one of the first to apply formal verification as means to enhance reliability of a industrial-scale trading system. Our observations and conclusions regarding the work are put in Section 8.5 and 9.

[1] May lead to full stop of the system working without a chance for recovery.

[2] `https://github.com/tonlabs/flex/issues`

1.2 Acknowledgments

We would like to thank Andrey Zhogin (TON Labs) - the main developer behind FLeX - for delivering nearly instant feedback on found issues and for addressing our technical questions on TON C++. His help greatly improved our work.

We also would like to thank for Boris Ivanovskiy (TON Labs) for addressing our questions on TON node architecture and on TON Solidity specifics. It also helped a lot.

We would like to thank Leonid Merkin (Innopolis University) for providing insight on how most of the modern trading systems work, including matching engine specifics.

1.3 Artifacts

All verification-related files, as well as TVF framework is available at:
`https://bitbucket.org/unboxed_type/flexphase2/src/master/`

A high-level specification of FLeX system is available at:
`https://bitbucket.org/unboxed_type/flexphase2/src/master/FlexSpec/spec_eng.pdf`

We did our best to do this document relatively self-contained, however additional studying might be necessary to grasp the material in full.

1.4 Warning

We started this verification project nearly from scratch - there were no formal verification tools available targeting the TON specifics what so ever.

What is worse, there is no definite specification for the TON node (documents of N.Durov are not precise enough is most cases). The current Rust node keeps changing, and some of its behavioral aspects are still not clear.

So, please keep in mind that there may be some deviations in our models from the real TON blockchain. Some of them are already known and planned to be fixed in future releases of TVF.

2 FLeX System Overview

Before going deep on technical verification aspects, we suggest to do a quick recap on the Flex system: what is the system purpose, what are the parts, how it is implemented, what parts are the most critical for its reliability.

2.1 About FLeX

FLeX is a blockchain-based trading system. It allows users to buy, sell and exchange tokens. To overcome classic shortcomings of other blockchain-based trading systems, FLeX implementation relies on a different highly distributed architecture. This makes FLeX a highly distributed system with intensive message passing between its components.

The main system function is to match user orders of opposite direction (buy vs sell) into deals. The deal here refers to an act of tokens exchange between users without a chance of cancellation.

2.2 FLeX Architecture

There are several known architectures of how the trading system might work. FLeX uses an order book based architecture, but with distributed implementation. It means that user orders are stored not in one smart-contract, but spread among several smart-contracts.

For example, buy/sell orders for TKN token by the price 10000 and orders for TKN token by the price 12000 gets placed into different smart-contracts.

The overall order book is formed by scanning the blockchain for presence of specific smart-contracts that are placeholders for trading orders. It is done on the user side.

One more specifics of FLeX is in how the user tokens get operated by the trading system. It is done by granting the FLeX a temporary permission to do user wallet operations on the user's behalf. Those permissions are limited in time duration and allowed tokens amount.

For more specifics on FLeX, please refer to the *High Level FLeX Specification* document.

2.3 FLeX Smart-Contracts

The FLeX system consists of the following smart-contracts:

Price, PriceXchg. It is both a main orders storage and orders matching engine. Price contract allows to buy/sell tokens for native blockchain coins. PriceXchg contract allows to , well, exchange one type of token for another in a given ratio. There is a separate Price and PriceXchg created for every trading pair and a price level.

FlexClient. The main user entry point into the system. Using this contract, the user can create buy/sell/exchange orders, cancel orders and operate do some wallet management.

TradingPair, XchgPair. Those contracts store information regarding the tokens being traded. They are needed to correctly identify available tokens at trade and form the order book on the user-side.

Flex. This contract stores the main system operational values, like fees value and contract binary program code values. Besides storing options, it is used to distinguish between different FLeX instances that may run in parallel.

TONTokenWallet, RootTokenContract. Implementation of TIP3-compatible token. RootTokenContract is a main entry point for users willing to create an instance of TONTokenWallet. Such wallet is needed to trade tokens in FLeX.

FlexToken. A variant of the TONTokenWallet, specifically tailored for FLeX.

In the latest FLeX, there are some more contracts present, but they are out of scope of the current work, so we do not describe them.

2.4 FLeX Implementation

FLeX smart-contracts are implemented in TON C++ programming language `https://github.com/tonlabs/TON-Compiler`.

Each smart-contract described in a pair of files: a header file with .hpp extension, and an implementation file with .cpp extension.

In this work, the following files are taken into account:

- FlexClient.hpp, FlexClient.cpp
- TradingPair.hpp, TradingPair.cpp
- XchgPair.hpp, XchgPair.cpp
- Price.hpp, Price.cpp
- PriceXchg.hpp, PriceXchg.cpp
- Flex.hpp, Flex.cpp
- TONTokenWallet.hpp, TONTokenWallet.cpp
- RootTokenContract.hpp, RootTokenContract.cpp

The repository link we refer to in our work:
`https://github.com/tonlabs/flex/tree/5a83080941`

3 Formal Verification Methodology

In this section, we explain our approach to formal verification. We also state the problem we are solving and define criteria by which to evaluate trustworthiness of the overall verification attempt.

3.1 Methodology Overview

To do verification, we embed a TON C++ program together with the TON Node Runtime layer into Dafny [3] - the programming language with built-in formal verification capabilities.

Dafny is both the name of the programming language and the name of a tool that do verification on those programs.

[3] https://github.com/dafny-lang/dafny

To do verification, Dafny programs get equipped with annotations - statements that describe desired program behavior. Annotations are given in the form of fist-order logic predicates with theories, or functional programming constructs. The program code that gets annotated is written in a richly typed imperative programming language.

In Dafny, the proof of correspondence between program code and its annotation is done *automatically* most of the time. It is achieved by translating the program with annotations into SMT formulas and feeding it into powerful SMT-solver. The solver answers if the specification holds for the code, or it is unknown. In the latter case extra annotations may be needed.

The ability to generate proofs automatically greatly simply and enhance the work of formal verification engineer. For example, contrast it with the burden of building proofs manually, how it is done in some modern proof-assistants.

It turned out, not only Dafny is expressive enough to model most of the TON C++ language in one way or another, but it is also a nice tool to build *reasoning frameworks*, specifically tailored for your domain.

To both simply our current work and make possible future verification projects, we developed the TON Verification Framework (TVF). It is a set of libraries for Dafny that allows an engineer to rapidly verify TON smart-contracts against the TON node execution logic modeled after the TON Rust node executor.

From the engineer perspective, TVF removes necessity of implementing stereotyped things from scratch each time, like message processing logic, TON programming primitives, TON specific types by putting all of this into reusable extendable components.

The methodology is depicted on Fig.1.

Later in the document, we give a brief overview of each verification component used in the methodology :

- Dafny Tool 4

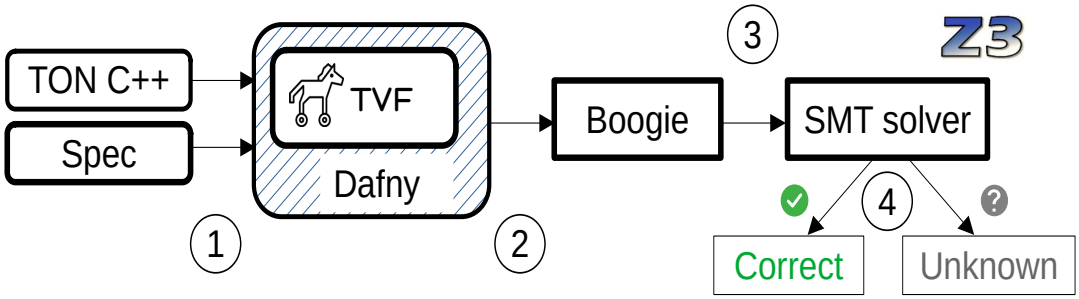

Figure 1: Verification in the methodology consists of 4 stages. **Stage 1** - the TON C++ program is translated into Dafny, using the TFV framework. The specifications for the program get translated into Dafny annotation language. **Stage 2** - Dafny tool translates the Dafny program into the Boogie program - the verification backend. **Stage 3** - Boogie program gets translated into the language of SMT-solver, Z3 in case. **Stage 4** - SMT-solver does the check and give an answer about the enquiry. After that, the answer propagated back to Dafny, and is displayed to the user. From a user perspective, stages 2, 3 and 4 are done automatically. They are shown for better clarity.

- TON smart-contracts Execution Environment 5
- TVF Framework 7
- TON C++ language overview 6

The procedure of encoding TON C++ program into TVF/Dafny in described Section 8.1.

3.2 Problem Statement

Now we would like narrow down what problem we are solving in the current work.

3.3 Background

Previously, we developed a high-level specification on FLeX. The document contains requirements for different aspects of the system behavior.

However, in the current work, by the contest rules, we limit our effort by doing formal verification of single functions, and not high-level

properties. The latter consist of modeling interactions between different smart-contracts and proving that property holds in any possible interaction scenario. Thus, it is related to safety properties class, and to be done on the next stages of formal verification.

The current work limits the effort on the function-level granularity. However, and even more importantly, we also prepare the infrastructure for doing system-wide safety and liveness proofs. Without this infrastructure, it is nearly impossible to conduct safety proofs in an efficient manner.

3.4 Function-level Predicates

What does the formal verification on a function-level granularity means in our specific case? We define it as follows.

For distinguished contract methods, provide the following:

1. Develop the Weakest Precondition predicate that, if holds, guarantees the success of the Compute phase execution

2. Develop the Post-condition predicate that describe the action queue contents after the method successfully executed the Compute phase

The point 1 is to derive a necessary and sufficient condition for the method to compute successfully. We latter check the condition with common sense understanding of the system intention. For some methods, this method is trivial (sometimes it is not even necessary to do anything at all!), and for others - very difficult (for example, Price.dfy, dealer::process_queue)
The point 2 is needed to evaluate the needed account balance value that would guarantee the success of Action phase.

Conditions 1 and 2, if put together, gives the precondition on the account, smart-contract and the message state that guarantees the successful execution through out the phases Compute and Action. If you add some trivial preconditions for Storage and Credit phase, you receive a predicate that, if holds, guarantees the success of message processing.

3.5 Methodology Soundness

Here we define a criteria that we use to judge the logical soundness of the presented methodology.

1. Smart-contract Execution Environment model is adequately reflects what happens in a real system. It may contain divergences, but they should not be that critical.
2. TVF framework precisely encodes the SmC Execution Environment logic (Section 7)
3. FLeX TON C++ implementation is correctly translated into the primitives of TVF/Dafny (Section 8.1).
4. Annotations for smart-contract methods makes sense relative to the defined problem statement, defined in Section 3.4.
5. Assumptions that are used to foster proofs looks reasonable and safe. (Section 8.3)
6. Annotations and proofs written in the supplied artifacts do not contain any insertions that may fool the tool into false conclusions (see Vacuous Proofs).

Actually, there are more: TON C++ compiler correctness, TON Node correctness, Dafny tool soundness/correctness, etc. We consider all of this to be reliable and trustworthy.

4 Dafny Tool

Here we give a very brief overview of what is Dafny.

Dafny - is a name of both a programming language with built-in formal verification capabilities, and the tool that does the verification checks. From now on, it should be obvious what exactly we refer to when use this name.

As an example, lets see the tiny annotated program written in Dafny.

```
1 method test(a:nat, b:nat) returns (c:nat)
2  ensures c == a + b
3 {
4  var v1 := a + b + a;
5  var v2 := b + b;
6  return v1 + v2 - b - b - a;
7 }
```

The test method adds two natural numbers, but does it in somewhat tricky way. The method annotation on the line 2 states that this method will always return a sum of two numbers. Dafny automatically proves that this indeed holds for every possible method input. If we try to change annotation for something different (and wrong), Dafny will discover this and show an error.

The statement on line 2 is a very simple theorem, and the proof for it gets built automatically by the tool. Using a classic predicate form, this theorem might be written as follows:

$$\forall a, b : test(a, b) = a + b$$

Put it another way, Dafny can be seen a automatic theorem proving tool that is specifically tailored for reasoning about richly typed imperative programs [4]

Dafny programming language supports a wide range of useful abstractions and mechanisms mimicking modern general-purpose programming language constructs.

- Algebraic Data Types
- Type parameters
- Exceptions using Failure-Compatible Types
- Traits, Classes and simple inheritance
- Heap-based object allocation
- Modules as an abstraction mechanism

[4] As opposed to general-purpose interactive theorem provers like Coq that allows to reason about broad range of mathematical objects, but at the expense of much less automation.

- Handful collection types, like: sequence, set, dictionary

And all of this is carefully adopted to be used with Dafny annotations.

There are also powerful proof encapsulation mechanisms like lemmas, axioms, ghost predicates, etc. We use it in our development, but do not discuss it here.

Truly, Dafny language and the tool leaves very pleasant impression!

5 Execution Environment Model

Smart-contracts do not exist on their own, they inhabit the blockchain, and the blockchain operates by its own rules. Blockchain node is an execution environment of smart-contracts. Hence, to reason about smart-contracts, we need to have a model of how the whole execution environment acts. The execution environment may not be dissected from the smart-contract without hurting the reasoning accuracy, and should be modeled properly.

Therefore, we describe our understanding of how the TON blockchain operates. The description is given in a high-level set-theoretic fashion, leaving a lot of details aside. We believe that if details were present, it would significantly hinder the understanding of the reader. For details, we advise to inspect TVF source code.

The whole blockchain may be described by the following entities:

$$Accounts, Messages$$

and the function *eval*

$$eval : Accounts \times Messages \mapsto Accounts \times 2^{Messages}$$

5.1 Accounts

Elements of the *Accounts* set are so-called accounts: they are basically placeholders for smart-contracts, equipped with some extra service

fields. The whole account is described by the following values: address, status, state ("state init") and the balance.

$$Accounts := Addr \times Status \times StateInit \times Balance$$

Addr is a unique identifier of the account among all existing accounts. Status describes an account mode of operation, and can be of 3 possible values:

$$Status := \{Uninit, Active, Frozen\}$$

Initially, all accounts have Uninit status. After the account state init is initialized with the code and data of a smart-contract, the status is switched to Active. If the balance of the account becomes insufficient to cover storage blockchain expenses, the account is switched into the Frozen state, and is unable to operate until it is switched back into Active state.

The smart-contract state contains its initial code and data.

$$StateInit = Code \times Data$$

Initial means that the state variables will change in future, but, at the time of initialization, they have the values supplied in the Data part of StateInit.

Elements of $StateInit$ set are datasets that have some meaning for the *eval* function. As with many other fields, we intentionally do not define those elements, leave them under-specified, because their nature is heavily depend on internals of *eval* function, and it is not defined here either.

A smart-contract gets executed when the blockchain node receives a correct message addressed to that smart-contract. In this case, it is said that the smart-contract executes the message. After the message is processed, the outcome is the updated account: new smart-contract state, balance and also a set of outbound messages generated by the smart-contract during the execution.

5.2 Messages

Messages between smart-contracts are called internal. Messages delivered from outside world into a smart-contract are called external.

Among internal messages, we define the following types: ordinary messages - messages that are sent by the smart-contract itself, during its execution; bounce message - a reply message that is sent to the originating smart-contract in case one of its messages failed to be processed properly on a receiver side.

Internal message is different to external message in that it has source address field, and it can carry coins within itself. External message carry no coins, and the smart-contract have to decide whether it should process such message, in runtime.

There is also a notion of event. Sometimes, being far fetched, events are called "external outbound messages". Conceptually, an event is just a record in the abstract events journal. This journal could be read by external users (but not by smart-contracts). To stay within the common terminology, we also relate events to messages.

Let us define those entities:

$$Message := OrdinaryMessage \times BouncedMessage \times EventMessage$$

The set $OrdinaryMessage$ is defined this way:

$$OrdinaryMessage := MsgHeader \times Option(MsgBody)$$
$$Option(T) := \{\varnothing\} \cup T$$

where $\varnothing$ is a specially distinguished element that denotes an undefined value.
Elements of $MsgHeader$ are service message headers. They may be of 2 kinds:

$$MsgHeader := IntMsgHdr \cup ExtMsgHdr$$

Here,

$$IntMsgHdr := Bounce \times Src \times Dest \times Value \times AnswerId \times AnswerAddr$$

Also $Bounce$ - is a boolean, denoting our intent to receive the notification message in case the destination contract fails to process our message, Src and $Dest$ - source and destination address, $Value$ - amount of coins attached, $AnswerId$ and $AnswerAddr$ - identifiers of

smart-contract and its method where the answer message should be addressed.

$$ExtMsgHdr := Dest \times Pubkey$$

Here $Dest$ - destination address, $Pubkey$ - sender public key. It is assumed, that the message is signed with a secret key that corresponds to the provided public key, the signature should match.
Let us now define the main carrier of the useful bytes, the message body.

$$MsgBody := Call \cup Deploy \cup DeployCall$$

The separation into 3 sets is by the following reasons: if the smart-contract is already deployed, the call into this contract has to be enveloped into a message from the set $Call$.
If a smart-contract is not yet deployed, it could be deployed in two different (and orthogonal) ways: a message of type $Deploy$ denotes a constructor call, and a message of type $DeployCall$ is an initialization of static contract variables with further method call (executed during the same message processing context).

Now, we define it further:

$$Call := ContractMethod$$

$$Deploy := StateInit \times ConstructorMethod$$

$$DeployCall := StateInit \times ConstructorMethod \times ContractMethod$$

Here, $ContractMethod$ elements are all possible method calls for the smart-contract, for example name of a method + argument values.

Elements of $StateInit$ are values that unambiguously define the smart-contract code and values of its state variables, $ConstructorMethod$ - a set of all possible constructor invocations.

Bounce notification messages are defined this way:

$$BouncedMessages := IntMsgHdr \times Option(MsgBody)$$

Here, the header may originate only from the $IntMsgHdr$ set, because bounce messages are generated only during the internal messages processing. The $MsgBody$ part may be absent (hence, the Option type), for example, if the message with some coins was sent into a contract, and the contract failed to process it (exception in fallback method).

An event is a pair of address and the event body:

$$EventMessage := SrcAddr \times EventBody$$

The *EventBody* set may differ for each smart-contract.

5.3 Message Processing

The message processing refers to a computation that takes place when the blockchain node receives a message addressed to some smart-contract. We describe it using the *eval* function.

The processing of a single message is the following :

$$eval : Account \times Message \rightarrow Account \times 2^{Message}$$

or

$$eval(account, msg) \mapsto (account', outMsgs)$$

Here, account refers to an account record that corresponds to the message destination address. The destination smart-contract inhabits this account.

Here, we describe the message processing logic in a superficial manner. A lot of details here takes place, most of them are not included in this description, not to hinder the general understanding.

1. *Credit* phase - The coins from the message are put on the balance of the account.

2. *Storage* phase - the node deducts so-called storage fee from the accounts balance. If the balance becomes insufficient to cover those expenses, the account becomes Frozen, and the execution stops here.

3. *Compute* phase - The smart-contract code gets invoked with the parameters from the message and from the account state. If the computation succeeds, the output of this phase is an updated contract state, and a sequence of actions to be performed on the next phase. If error happens, the execution continues on the Bounce phase.

4. *Action* phase - the actions generated on the previous phase starts to execute by the action handler. The order of execution depends on action type, so of them get postponed.

   We elaborate a bit further on what Actions are in the next section.

   The phase completes successfully if all actions are valid entities and there are enough coins on the account balance to cover all the actions fees. Successful processing of Send Message actions result in outbound messages put in the queue.

   If the phase completes with an error, then execution goes to Bounce phase. In this case, smart-contract state gets recovered to the initial state, that was before the execution of the Compute phase.

5. *Bounce* phase - If the incoming message was an internal message and had the bounce flag set in its header, then reply message of type BounceMessage is generated.

   It is put straight into the outbound message queue. The original coins of the incoming message minus some fees get attached to the bounce message. The balance of the account is updated accordingly.

5.4 Actions

Lets discuss entities called Actions. To the best of our knowledge, it is a unique concept among existing blockchains and may be unfamiliar for some experts.

During its execution, a TON smart-contract may perform a series of distinguished operations, like message sending, coins reservation, program code substitution, etc. Those distinguished operations are called *Actions*.

From a user perspective, those operations are nothing special. It is how they get processed inside the TVM/TON node that constitutes the most of its complexity.

Actions do not get executed at the place of its invocation, inside a smart-contract, but are put aside into a special queue - the action queue. Elements of this queue get processed after the computation phase of the contract is successfully finished.

The complexity that we are referring to arise due to following reasons:

- Because actions do not get executed immediately, but get queued, the contract, while executing, has no clue if he has enough balance to cover the fees of the next action it needs to do. The balance is not updated during the computing phase. Because of that, special logic has to be introduced to track such things.
- Some actions are highly customizable. Due to this, some actions are processed out-of-order, not to break common sense logic. This out-of-order rule greatly hinders ones ability to reason about the Action phase.
- Action customization leads to great number of possible outcomes. For example, you can send message with $2^6 = 64$ flag combinations, each with different outcome. Some flag combinations may conflict with other actions in non-trivial way.

This complexity creates a perfect environment for making logical and implementation errors. It also greatly complicates formal verification process.

6 TON C++ Language

Here we list some facts on TON C++ language and smart-contracts written in it, that might be handful during the work evaluation.

6.1 TON C++ Features

From C++17 standard, TON C++ language dialect differs considerably. Here are some of the most important differences:

- No memory management operations permitted.

- Only integer arithmetic is allowed. Any integer type, and also the char type, has a length of 257 bits.
- Exceptions mechanism is not supported. There is a built-in function *require(cond, code)*, that stops execution in case the condition *cond* does not hold at the time of function invocation.
- Function pointers are prohibited.
- Pointers on function arguments are prohibited, excluding functions with the attribute *__always_inline*
- Most of the STL containers are not supported (due to memory management issue)
- No multi-threading allowed

The TON C++ compiler is shipped with its own standard library that includes functions for message delivery, working with runtime, TVM-specific functions, TVM-specific container implementations, etc.

6.2 Header Files

In the header file with '.hpp' extension, the smart-contract interface gets declared. For each smart-contract method, several method attributes may be specified. The attributes semantics is described below:

- *internal* - the method may be called by an internal message. Please note, that this attribute may be combined with *external* attribute. In this case, the method is reachable using both internal and external message.
- *extenral* - the method may be called by an external message. May be combined with *internal*.
- For *external*-methods, *noaccept* attribute means that the method is willing to manually execute the *tvm.accept()* in case it is needed. Otherwise, the *tvm.accept()* is called automatically.

- *answer_id* attribute mean that the method is supplied with extra answer_id parameter value. This value will be used to coordinate the return message if any[5]

- *dyn_chain_parse* attribute signals that the serialization mechanism for the method have to be implemented in a way that favors gas usage over bytecode size.

6.3 TON C++ Runtime

The program code of TON C++ smart-contract is only a visible part of the code base that gets translated into TVM bytecode, and later put into the account StateInit.
Besides the program code itself, the contract has its own runtime included. It is implemented in the following directory:

`https://github.com/tonlabs/TON-Compiler/blob/master/llvm/projects/ton-compiler/cpp-sdk/tvm/`

We distinguish the following modules among others:

- **smart_contract_info.hpp** - System variables of the smart-contract, such as: the current block time (unix epoch), random seed and the current account balance. It is only copies of the values received from the actual account record, so mutating it will not effect the account values directly.

- **contract.hpp** - definition of tvm_transfer(), tvm_rawreserve(), tvm_accept(), tvm_hash() functions

- **contract_handle.hpp** - a wrapper that lets a developer send messages in a convenient way

- **default_support_functions.hpp** - definition of int_sender(), int_value(), set_return_func_id() functions

- **builder.hpp** - cell type marshalling logic (cell)

- **queue.hpp, small_dict.hpp** - queue and dict implementations

[5]In TON Solidity, such methods are called *responsible*

We provide this information by the following reason:

- All of this is used in FLeX code
- TVF models many of this functionality, so it is useful to know where to see the original implementation that we model.

6.4 Operator Return is a Message

In many FLeX methods, you may run into familiar **return** operator.

Keep in mind that in TON C++ this construct generates message that is sent to the contract with the address specified by the return_addr value, and put it in the function specified by return_func_id value.

It is allowed to specify the number of coins that is sent together with this message.

All of this can be customized by calling set_int_return_* functions family. Default values for those is set by the runtime.

Keep this in mind, because in TVF, we model the return operator by a special method call TON_CPP_return(), while the return keyword of Dafny is used for a different purpose.

7 TVF Framework

In this section, we give a brief overview of the TVF - TON Verification Framework, a set of libraries written to boost the verification engineer productivity in his verification attempts.

7.1 Domain-Specific Logic

The Dafny language allows to write imperative richly-typed code, cover this code with specifications and nearly automatically prove the correspondence between the code and the specifications, for all possible inputs.

However, the TON blockchain has its own domain specifics, such as:

- Domain specific functions, such as: message sending, coins reservation, some runtime parameters tuning, etc. The side-effect of those functions may affect on the further execution process.
- Smart-contracts get executed using *transactional semantics*: the method either succeeds, and the new state variables get stored in the state, or, if error happens, the state variables rolled back to its original value.
- The smart-contract can not be dissected from the account it belongs to, but this tight link is not visible in the code. The account entity is hidden from the developer, but its state may influence the outcome of the message processing, hence the call result.
- Node message processor that is called before running the SmC method, has rather complicated logic and defines the way the account state changes. It is highly undesirable to exclude it from the executing chain, otherwise the reasoning accuracy may suffer significantly.
- There are set of specific types typical for the TON blockchain, and operations on those types have its specifics.

From the above, it follows, that you can't just take the smart-contract code and rewrite it in pure Dafny. You will not gain anything useful this way.

Behold! This is where TVF comes in.

7.2 TVF. The Purpose.

The logic layer that is described above is represented by a big volume of code. It affects the outcome of message processing and can not be ignored, it has to be included in the final verification model.

As was stated before, the visible part of the code is only nearly a half of what is actually going on when the message is processed. There is Action phase that has to be considered.

```
1 {
2   require(msg_pubkey() == owner_,
3     error_code::message_sender_is_not_my_owner);
4   tvm_accept();
5   ext_wallet_code_ = ext_wallet_code;
6 }
```

Figure 2: Body of setExtWalletCode from FlexClient.cpp

The TVF framework hides most of this complexity into handful abstractions, allowing the verification engineer to build soundly reason about TON contract behavior without ignoring all the complexities of the protocol.

Besides, TVF tries to stay syntactically close to the original smart-contract program code: the form of the code should not change considerably, otherwise it becomes difficult to trace the correspondence between the two.

The TVF framework strives to give the following advantages to the user:

1. The translated code stay syntactically close to the original TON C++ program text.
2. Hide most of message handling protocol complexities from the engineer, leaving visible only those parts that are needed to build proofs.
3. Hide Compute and Action phase complexities.
4. Give an engineer a way to specify the outcome of different SmC execution phases.
5. Give the engineer all the TON execution environment related primitives, and types.

7.3 TVF Internals

Here, we briefly describe the framework device from the engineering user perspective.

```
1  {
2    :- require(msg_pubkey() == st.owner_,
3     error_code_message_sender_is_not_my_owner);
4   tvm_accept();
5   st.ext_wallet_code_ := Value(ext_wallet_code);
6   return Success;
7  }
```

Figure 3: Body setExtWalletCode from TVF

7.3.1 Package Contents

Currently, TVF consist of the following files:

- **BaseTypes.dfy** - common type definitions used in TON C++. Besides, types $option\langle T\rangle$ that corresponds to $std::optional\langle T\rangle$ and Status types for emulating exceptions.
- **Queue.dfy** - Queue container definition. Functions defined in a way to closely reflect what is used in FLeX.
- **TONTypes.dfy** - TON specific types for Messages and Actions.
- **TONContract.dfy** - The module that is refined by all smart-contracts. Contains all the message and phases execution logic. Besides, it contains all needed TON C++ runtime functions. It is a core of the framework.
- **ErrorCodes.dfy** - Numeric error codes. Right now, the values do not correspond to the real Rust-node error values.
- **MapToSeq.dfy** - Handful function that converts map into a set. Sometimes, it is needed for specifications and proofs.

7.3.2 Smart-Contract Programming

To embed the smart-contract into TVF/Dafny, the engineer has to do the following:

1. Define the module Module that *refines* the module TONModule. Refinement mechanism is not documented here, please refer to Dafny documentation [6].

[6]https://github.com/dafny-lang/dafny/blob/master/docs/DafnyRef/out/DafnyRef.pdf

2. Within the Module, refine the *ContractStateVariables* class, putting all the contract state variables inside it. They may use any types here, including TON-specific types defined in BaseTypes.dfy

3. Inside the *ContractStateVariables* class, refine the constructor. Constructor here set variable values to their default values, defined by the compiler or, if the type is an object, by its constructor, *even before the smart-contract constructor gets called.*

   You better be careful here, and relate the value initialization logic with your language. For example, in C++ it is common to have undefined values for variables. In Dafny, this is denoted with '*' symbol.

4. Inside the *ContractStateVariables* class, refine the initial_values() predicate. This predicate state what values are in the contract variables right after the constructor gets called. It is a constructor specification.

5. Within the *Module*, *refine* the *TONContract* class. Here, you start program your contract methods. We elaborate on it in Section 7.3.3.

6. Within the *TONContract*, *refine* methods

   - *execute_constructor()* - smart-contract constructor body goes here
   - *execute_external_method()* - external messages handler
   - *execute_internal_method()* - internal messages handler

7. If you need to reason about the overall message execution correctness, i.e. not only Compute phase, but also Action and Bounce phases, you will also need to refine the following:

   - *msg_dispatcher()* - main message entry point
   - *tvm_compute_phase()* - compute phase handler

Regarding points 6, 7. Please note, that you do not need to program anything inside those methods! You only need to put specifications regarding possible messages for your smart-contract and expected outcomes. Everything else is taken care by TVF.

```
__always_inline
void setFlexWalletCode(cell flex_wallet_code) {
  require(msg_pubkey() == owner_, error_code::
     message_sender_is_not_my_owner);
  tvm_accept();
  flex_wallet_code_ = flex_wallet_code;
}
```

Figure 4: FlexClient::setExtWalletCode method from FlexClient.cpp

```
// [[external, noaccept]]
method setExtWalletCode(ext_wallet_code: cell)
 returns (s:Status)
 requires external_message()
 modifies st
 ensures setExtWalletCode_pre() <==> s == Success
 ensures s == Success ==> setExtWalletCode_comp()
{
 :- require(msg_pubkey() == st.owner_,
    error_code_message_sender_is_not_my_owner);
 tvm_accept();
 st.ext_wallet_code_ := Value(ext_wallet_code);
 return Success;
}
```

Figure 5: setExtWalletCode method from FlexClient.dfy

7.3.3 Smart-contracts Methods Encoding

We now look at how the methods of a smart-contract gets encoded relatively to how it is done in TON C++. Lets look at the example.

On Fig.4 and Fig.5 the same method FlexClient::setExtWalletCode is encoded both in TON C++ and TVF/Dafny.

Return value. Note the difference between return types of two methods. In TON C++, the return type is *void*. But in TVF, the return type is specified as *Status*. Why it is done this way?

All methods that potentially may throw an exception have to be equipped with output parameter of failure-compatible type: it is the Status type in our case. In case of an exception, this return value signals the calling party that the method execution was aborted, returning value Failure(n:nat), or Success in case everything is fine. It

does not prevent us from returning other values if needed: Dafny supports returning multiple values in a single statement.

So, in TVF version, the method does the return Success at the end, while the TON C++ version contains no return operator at all (though, it implicitly presents there).

Method Attributes While it is not reflected in the method definition, the declaration of the setExtWalletCode in the header file FlexClient.hpp is tagged with the attribute external, noaccept. It means, this method may be called only by an external message.

In Dafny, the method attributes mean something different and can not be customized. That is why, in our case, attributes are encoded as pre-conditions for the method, located in the requires block. On line 3, it is required that method be executed only by en external message.

Operator require. On line 8, the require call is made using the failure propagation operator, like ":- require(cond, error);".
If the failure happens, the execution immediately returns with an error. Contrary to that, the tvm_accept() call on line 9 is done in a usual way. This is because tvm_accept() may not throw, by definition.

Contract State Variables Update. In TVF, smart-contract variables are stored within *st* object. This is why, all the updates must be done for fields inside this object. If the method itself or any method that gets called by the former mutate state variables, the extra *modifies st* annotation must present.
If a method do some TON-specific, action inducing operations, like message sending, modifies block must be extended accordingly.

Optional Type. On line 10, the value
Value(ext_wallet_code) is assigned to the *st.ext_wallet_code_* variable. In FlexClient.cpp, and in FlexClient.dfy this variable has the type $option\langle T\rangle$. But in C++ value packing inside the type $std{::}optional\langle T\rangle$ happens automatically, at type resolution phase. In Dafny, you have to explicitly specify the proper type constructor, that is Value(v) [7].

[7]There is a co-variance type mechanism in Dafny that allows to do things similar to what happens in C++, but we do not use it currently

Those discrepancies appear in some places. Let those who read the source code will not be afraid of it!

Ensures block. On line 5 and 6, the specification for *setExtWalletCode* is given.
The first one tells that if the method gets called from a state respecting the *setExtWalletCode_pre()* predicate, then the method will succeed ($r == Success$). And in the opposite direction: if the method succeeds, it must be the case that initial state respected the predicate. So, the predicate define necessary and sufficient condition for the method to succeed. We call such condition *the weakest precondition*[8]

The second one, on line 6, states that if method succeeds, then the predicate *setExtWalletCode_comp()* will hold. The predicate describe the expected state after the Compute phase succeeded.
Usually, we put a description of what contents has to be in the action_queue after the method succeeds, but for the method *setExtWalletCode* it is not that interesting - it generates no actions at all.

7.3.4 Writing Specs for Smart-Contracts

The TVF specifications may be put in several places, depending on what is the verification purpose.

- Annotations inside the *ensures* block, within the smart-contract method definition. The example was shown recently. This type of specification may only assert things about smart-contract state after the Compute phase.

- Specifications inside the *ensures* block of the *msg_dispatcher*. This type of spec is needed to prove final properties: properties that hold after all phases get executed.

  First and foremost, those proofs are needed when you reason about message exchange between several smart-contracts (see, PROP_SEC01.dfy, for example)

- Specifications inside the *ensures* block in *execute_external_method()*, *execute_internal_method()*, *tvm_compute_phase()*.

[8]This term was coined by E.W.Dijkstra in his famous "*A Discipline of Programming*" book

It is required to annotate those methods in case you do *msg_dispatcher* annotations.

At this point, we have to stop our clumsy shallow description of the TVF framework, with the hope that if a question arises then you may try to inspect sources by yourself, or ask the author directly.

8 FLeX Formal Verification

In this section, we give some details regarding our FLeX verification effort.

8.1 Mapping TON C++ into TVF/Dafny

Any imperative programming language, including TON C++, may be separated into two parts: effectful (create objects on the heap, mutate state variables, exceptions, message sending, etc) and pure (if/then/else, some arithmetic, basic types allocation, evaluating functions without side effects, etc).

Besides, from TON C++ language, we may extract even more narrow sub-language that is used in the FLeX development. From now on, we use TON C++ to denote this sub-language.

Pure fragment of TON C++ includes:

- Conditional operator *if/then/else* when conditional statement is evaluated.
- Comparison operators applied to some elementary types and some non-elementary types, for example uint_t and bool_t.
- Construction of objects *std::tuple* and *std::optional*, from the already evaluated values .
- Calculating the *std::min()*

Effectful part contains the following:

- All action-inducing calls, including message sending, coins reservation, etc.

- Integer arithmetic operations due to its overflow and underflow potential (remind you that TVM throws exception in this case). Increment also goes here.
- Working with queue and dictionary. It may throw in several places.
- Working with iterators
- Passing reference arguments into functions
- Calling TON-specific functions that may throw (require, for example)
- etc.

Precise semantics of C++ language is still not defined. The best thing we have is a description of the C++ standard [9]. We used it extensively through out our whole modeling adventure.
Modeling the pure part of TON C++ is relatively easy.
For modeling the effectful part, we did the following:

- For queues: we equip the methods that may throw with FC type. So we could track exceptions.
- For passing reference arguments into functions, we defined the type Ref¡T¿, that creates an object initialized with the passing value. In every place that reference this value, we use the value from this object.

```
1 var sell:Ref<OrderInfo> := new Ref(sellOrderInfo); //
     In C++, OrderInfo& sell = sellOrderInfo;
2 // ...
3   sell.st := sell.st.(account + v); // In C++, sell +=
        v;
4 // ...
5   make_deal(sell, buy);
6 // ...
```

- TON specific methods that may throw exception are equipped with FC type, all of them are called using the failure propagation

[9] https://en.cppreference.com/w/

operator ":-". The Dafny itself forces those methods to be called this way.

- Working with iterators - we decided not to model classic C++ iterator with all the methods like begin(), end(), next(), etc. Instead of that, we carefully rewritten the code without using iterators. In FLeX, there are only 2 such places - it is Price and PriceXchg, *cancel_order_impl()* method.

- Integer arithmetic - Dafny has the built-in overflow and underflow checks. In other words, each arith operation + / - / * / div has to be accompanied with the proof of overflow/underflow safety, or the assumption that it never happens. In several places, when we are sure that the overflow/underflow is highly improbably, we use overflow-resistant $add(X1, X2)$ and $addL([X1, X2, X3..])$ functions to sum several integers.

- There are some more of TON C++ nuances that we modeled in an ad-hoc way, because they appear only in single places throughout the project.

8.2 Properties and Proofs

Here, we list all properties and scenarios we addressed in our verification effort, relying on the formal apparatus and methodology described above.

- Arithmetic exception safety: all contracts

- Successful methods invocation on the Compute phase in case the initial state respects the weakest precondition predicate: *Price* (partially), *FlexClient*, *TradingPair*, *XchgPair*.

  The weakest precondition predicates are named using the scheme "method_name_pre".

- Successful method invocation on the Action phase in case the initial state respects the given predicate: *FlexClient*, *TradingPair*, *XchgPair*

- We implemented several interaction scenarios that arise in formalization of properties from the SEC section of FLeX specification. Those scenarios, together with its formalization, provided in the files:*PROP_SEC01.dfy*, *PROP_SEC02.dfy*, *PROP_SEC-03.dfy*, *PROP_SEC04.dfy*. We also started to prove scenario *PROP_ORD01.dfy*, but during building proofs, we ran into an error in the Price, and this is why we failed to prove it.

- The strict lower and upper bound on the number of generated messages for the method *Price::buyTip3* has been derived. Thanks for this proof, we were able to catch an issue with uncontrolled queue growth potentially leading to deadlock.

8.3 Assumptions

In our work, we relied on several assumptions to make the proof effort manageable.

- The marshalling functions *build(), parse()*, etc work correctly. We use high-level algebraic datatypes to encode structures, even if it is passed in a message.

- There are several places where they add several values. It is clear from the domain knowledge that those values are tiny relatively to datatype uint128. So we use exception-safe $add()$ and $addL()$ in those places.

- We assume that the queue size will never exceed MAX_UINT64. It is clearly the case, because the smart-contract is incapable of storing that much data.

- In several places, we rely on the fact that *tvm_balance()* is greater or equal that *int_value()*. It is indeed the case, because int_value() (message coins) get added to the balance before executing a method.

- We assume that timestamp value will never exceed *MAX_UINT32* value. This corresponds to year 2038. Before this date, we should not worry.

- Assumption regarding the

$$hash(smc.state\_init) == smc.account\_id$$

  It is by the definition of an SmC address.

8.4 Found Defects

In this section, we present our findings that we discovered during the verification process, we also explain how exactly the defect was found. We think that this experience may be very beneficial to other FV engineers.

1. In *Price.cpp, dealer::make_deal()*, arithmetic overflow may happen, leading to creation of an empty std::optional object. When dereferencing such object using operator *, the compiler may put an arbitrary value. In this case, we calculate the deal price, so, potentially, this may be very serious issue. `https://github.com/tonlabs/flex/issues/20`

   The error was found during inspecting the semantics of operator * for an empty optional type, while translating TON C++ into TFV/Dafny.

2. In *Price.cpp*, *dealer::process_queue()* the uncontrolled message queue growth may happen. It works this way, because, inside the cycle, the break operator is placed not quite accurate. Potentially, this could lead to a smart-contract deadlock. `https://github.com/tonlabs/flex/issues/22`

   The error was found during the building proofs on the message queue size bounds for the method *dealer::process_queue().*

3. In *Price.cpp*, extract_active_order(), the out-of-gas exception may happen. It works this way, because the iteration over cycle is not bounded, so if the collection has a significant size, the gas will be eventually exhausted. `https://github.com/tonlabs/flex/issues/23`

The error was found during building the proofs for action queue size bounds.

4. In *Price.cpp*, *dealer::process_queue()* the out-of-tons notification get sent with incorrect value of processed tokens. `https://github.com/tonlabs/flex/issues/25`

   The error was spotted by the FLeX developer himself while explaining some tricky question related to C++ references mechanics, and the code was near this place.

5. In *PriceXchg.cpp*, the function minor_cost() performs the call *builtin_tvm_muldivr(a,b,c)*. This function may throw in case c = 0.

   This case is not considered in the code, so, theoretically, is possible.

   `https://github.com/tonlabs/flex/issues/26`

   The error was found during building proof of successful method invocation on Compute phase.

6. In *PriceXchg.cpp*, cancelBuy() there is an error in how the value of the fee is calculated when buy orders get canceled.

   `https://github.com/tonlabs/flex/issues/27`

   The error was spotted due to familiarity with the *Price.cpp*, that contains very similar code, but without this error. The memory told us that there was a different logic in this place.

7. In *TONTokenWallet.cpp* the dictionary *lend_ownership_* may grow in uncontrolled fashion, leading to out-of-gas, and later to a deadlock.

   `https://github.com/tonlabs/flex/issues/28`

   The error was spotted analytically while we worked on some proofs for the code that calls this method.

8. In *RootTokenContract.cpp*, *deployWallet()* when the value wallet_code_ is not set, the exception may happen after the tvm.accept() was called, a scenario that leads to excessive coins loss due to charges applied by all validators running this code.

   `https://github.com/tonlabs/flex/issues/29`

   The error was spotted while deriving precondition for some method that is called within this method.

9. In *PriceXchg.cpp*, *dealer::process_queue()* there is a condition that contains a typo. Instead of checking the account value for buy order, it is checked for the sell order. `https://github.com/tonlabs/flex/issues/30`

   The error was found during manual translation of TON C++ into TVF/Dafny, with critical comprehension. We also noted that in similar code inside *Price.cpp*, the check looks different.

8.5 Statistics

Now, we have the following statistics: totally, from 9 found defects, 5 defects were found during proof building, 3 defects - by manual translation of the code, and 1 defect was spotted by the developer himself while answering some technical question of ours.

9 Conclusion

The TON blockchain gives provides rich smart-contract functionality capabilities.
From one hand, this increases the expressivity of programs. On the other hand, it increases program complexity that may lead to logic and implementation errors. So, we consider formal verification a viable way to ensure correctness of such programs.

A considerable amount of our time was spent on recovering details of message execution inside the TON node. It is not described anywhere,

and, still, some questions are not resolved[10]. To overcome this difficulty, we hope that from some time, the TVF may become a reference model for the TON node, at least in the message processing part.

While doing the verification of FLeX, we were able to find several non-trivial errors that could lead to a hazard situation. Previously, when doing manual code audit, we did not find those errors, but they were there.

In our opinion, the functional-level formal verification method of finding bugs demonstrated its efficiency.

A Artifacts Evaluation Instruction

1. Install Dafny version 3.3.0. Building and Installation manual is here:
   `https://github.com/dafny-lang/dafny/wiki/INSTALL`
2. Download the repo with Flex models and proofs: `https://bitbucket.org/unboxed_type/flexphase2/src/master/`
3. Run the check by executing *make* command
4. Depending on how powerful your CPU is, the check may take up to 5 minutes. If there are any timeout errors, the timeout value may be increased in the *Makefile*.

[10] see our issues in `https://github.com/tonlabs/ton-labs-executor/issues`