Towards Contract Modules for the Tezos Blockchain

Thi Thu Ha Doan
University of Freiburg, Germany

Peter Thiemann
University of Freiburg, Germany

Abstract

Programmatic interaction with a blockchain is often clumsy. Many interfaces handle only loosely structured data, often in JSON format, that is inconvenient to handle and offers few guarantees.

Contract modules provide a statically checked interface to interact with contracts on the Tezos blockchain. A module specification provides all types as well as information about potential failure conditions of the contract. The specification is checked against the contract implementation using symbolic execution. An OCaml module is generated that contains a function for each entry point of the contract. The types of these functions fully describe the interface including the failure conditions and guarantee type-safe and sometimes fail-safe invocation of the contract on the blockchain.

2012 ACM Subject Classification Software and its engineering → Specification languages

Keywords and phrases contract API, modules, static checking

Digital Object Identifier 10.4230/OASIcs.FMBC.2021.5

Category Short Paper

Supplementary Material *Software (Source Code)*: `https://github.com/proglang/tezos-project/tree/master/papers/contract-modules/code`

Funding *Thi Thu Ha Doan*: supported by a grant from the Tezos foundation.

1 Introduction

Contracts on the blockchain rarely run in isolation. To be useful beyond shuffling tokens between user accounts, they need to interact with the outside world. On the other hand, the outside world also needs to interact by initiating transactions and starting contracts that feed information into the blockchain. One direction is addressed by oracles that watch certain events on the blockchain, create a response by calculation or gathering data, and then invoke a callback contract to inject this response into the chain. Trust is an essential aspect for an oracle.

The other direction is about automatizing certain processes in connection with the blockchain. For example, opening or closing an auction according to a schedule, programming a strategy for an auction, or creating an NFT. To this end, an interface is needed to invoke contracts safely. Existing interfaces are lacking because they are essentially untyped (string-based or JSON-based) and often low level because they require dealing directly with RPC interfaces. Trust is not needed because the process runs on behalf of a certain user.

We propose contract modules that provide a clean, language-integrated way to interact with a blockchain from a host language (OCaml in our case). They abstract over underlying string-based interfaces and details like fee handling. They provide a high-level typed interface which reduces a contract invocation to a function call in the host language.

The contract modules approach does not provide a fixed API, but rather generates a specific interface for each contract with one function for each entry point of the contract. This interface is statically checked against the contract implementation to ensure type safety and exception safety. That is, values passed to an interface function do not lead to type mismatches when invoking the underlying contract. Morever, every failure condition arising during contract execution is handled by proper error reporting according to the interface.

Listing 1 Simple auction contract (auction.tz).

```
parameter (or (unit %close) (unit %bid));
storage (pair (bool %bidding)
              (pair (address %owner)
                    (address %hi_bidder)));
```

Our work is situated in the context of the Tezos blockchain, which supports Michelson as its low-level contract language, and the host language OCaml, which comes with an expressive polymorphic type system as well as a powerful module system that we enhance with contract modules.

2 Context

Tezos is a third generation, account-based, self amendable blockchain [8]. It employs a proof-of-stake consensus protocol, which includes ways to evolve the protocol itself. The consensus protocol is executed by so-called bakers and their proposed blocks are checked by validators. They receive some compensation in the form of tokens (Tezzies) for their work. According to proof-of-stake, bakers and validators are nodes elected by the Tezos network according to their token balance.

Each Tezos contract is associated with an account as well as some storage. Contracts are pure functions of type $\mathit{parameter} \times \mathit{storage} \to \mathit{operation\ list} \times \mathit{storage}$, where the types *parameter* and *storage* are depend on the specific contract while the type *operation* is fixed by the Tezos system. When a contract is invoked with a parameter, the blockchain provides the current storage and updates it with the second, storage component of its return value. The first component of the return value is a list of blockchain operations (contract deployments, token transfers, contract invocations, and delegation of baking rights) that are executed transactionally after the first invocation terminates. Each invocation may be accompanied with an amount of tokens that are added to the current account balance of the callee contract.

Contracts are implemented in the language Michelson, a statically typed stack-based language. Each contract has fixed types for its parameter and for its storage. The storage is initialized when the contract is deployed. Besides primitive types like unit, int, bool, address, and string, there are pairs, sums, functions, lists, and maps along with a range of domain-specific types (operation, key, signature, timestamp, key_hash, contract, mutez – for tokens, and so on) most of which can serve as types for storage and parameters.

A Michelson contract has a single default entry point. However, the *parameter* type is typically a sum type and each component of the sum can serve as a subsidiary entry point.

3 An auction contract

As a concrete example, we consider a simple auction contract with the header shown in Listing 1. This contract has two entry points, `close` and `bid`, expressed by giving the single parameter a sum type. To call the entry point `close` we invoke the contract with parameter `Left ()` otherwise we use `Right ()`, where `()` is the sole value of type `unit`. The contract's storage is a nested pair which contains a boolean flag and two addresses.

The contract works as follows. It is deployed with storage `(true,(owner,owner))` which indicates that bidding is allowed and the contract owner is currently the highest bidder. On deployment the owner deposits an initial balance to indicate the minimum bid. Closing the

Listing 2 Example contract module.

```
contract type Auction = sig
  paid entrypoint bid ()
  raises "closed"                (** auction closed *)
       | "too low"               (** bid too low    *)

  entrypoint close ()
  raises "closed"                (** auction closed *)
       | "not owner"             (** caller cannot close *)
end
```

contract transfers the balance to the owner. Closing is restricted to the owner. Closing as well as bidding fails if the auction is closed. If bidding is open and the amount of tokens accompanying the bid exceeds the current highest bid, the current bidder replaces the previous highest bidder and the previous highest bidder is reimbursed. Otherwise, bidding fails.

To invoke this contract from an OCaml program, we generate an OCaml module, say `Auction`, from a specification of the contract. This module contains two functions `close` and `bid` corresponding to the entry points. The type of these entry points reflects further properties of these entry points as well as the ways in which an entry point might fail.

Besides the obvious, technology induced ways that a contract invocation might fail (insufficient gas price offered, insufficient gas to complete, timeout due to lack of connectivity, etc) a Michelson contract can fail due to a programmer induced condition caused by the instruction `FAILWITH`. It terminates contract execution with an error message which is reported back to the caller. This error message includes the top value on the stack.

We consider the technological failures like Java's unchecked exceptions, but we wish to deal with the explicit failures like checked exceptions [2]. Our generated code handles failures in a suitable error monad that makes the failures explicit in a custom datatype.[1]

Listing 2 shows a contract module for the auction contract. It declares the entry point `bid` as **`paid`**, i.e., it should be invoked with a non-zero amount of tokens, it states the pattern `()` for the input value of type `unit`, and it specifies two failure messages that we wish to deal with programmatically. The `close` entry point is similar, but it is not `pair`. This contract reflects the understanding of the programmer with the intention that the **`raises`** clauses cover all failures that can arise during execution of the respective entry point.

Listing 3 contains an OCaml module signature as it would be generated from the contract module. The module `Tezos` supposedly contains types and other low-level Tezos-specific definitions. The type `pukh` for public key hashes identifies contracts, the type `mutez` stands for Tezos tokens, the type `status` reflects the internal return status, and `monad` is an internal monad type. The signature declares a function and an error type for each entry point.

The error types mirror the raises clauses. The first argument of each function is the address of the contract, then an optional argument for the transaction fee, an argument for passing an amount of tokens (only for a paid entry point), the next argument would be for the parameter; it is omitted here because its type is `unit`. The return type refers to the specific error type.

[1] Alternatively, errors could be modeled using OCaml exceptions, but we choose to stay within the monadic framework that is used by existing Tezos APIs.

Listing 3 Generated signature.

```
type bid_errors =
   | bid_closed      (** auction closed*)
   | bid_too_low     (** bid received is too low *)

val bid
  : Tezos.pukh -> ?fee:Tezos.mutez -> amount:Tezos.mutez
  -> (Tezos.status, bid_errors) Tezos.monad

type close_errors =
   | close_closed    (** auction closed *)
   | close_not_owner (** caller cannot close the auction *)

val close
  : Tezos.pukh -> ?fee:Tezos.mutez
  -> (Tezos.status, close_errors) Tezos.monad
```

4 Simple Checking

We check the contract by symbolic execution against its specification in the contract module. Symbolic execution proceeds by calculating a symbolic stack at each transition from one Michelson instruction to the next. The symbolic interpreter is fully typed and rejects ill-typed Michelson programs, if the ill-typed part is reachable from the default entry point. The initial stack is calculated from the storage type, the parameter type, and the entry point. As we aim to keep symbolic values as concrete as possible, some instructions may turn out to be unreachable or only reachable under certain conditions. This way, we obtain, for each entry point, a set of final symbolic stacks along with a path condition indicating when this final state is reachable. Moreover, for each `FAILWITH` instruction we obtain a symbolic value for the reported message and a path condition indicating reachability of this instruction. We employ the SMT solver Z3 [7, 4] to check the feasibility of a path condition.

Here are some simple examples of checkable properties.

For each entry point, we collect the set of reachable instructions. For example, the `AMOUNT` instruction obtains the amount of tokens sent with a contract invocation. It should not be possible to reach that instruction from an unpaid entry point like `close`. This property is straightforward to check from the path condition generated for the `AMOUNT` instruction.

For each entry point, we collect the set of reachable `FAILWITH` instructions along with their path condition and their arguments. In most cases the symbolic interpreter finds a concrete argument for each `FAILWITH` instruction because the typical usage pattern is to push a concrete (string) value on the stack immediately preceding the `FAILWITH`. It remains to check that each argument to `FAILWITH` should be accounted for by one **`raises`** clause.

Checking the simple auction contract in this way already flags an omission in the contract module (Listing 2). The problem is that bidding returns the previous highest bid as part of the transaction where the highest bidder is stored as a value of type `address`. However, to receive a token transfer, this address must by cast (by the Michelson implementation of the contract) to an implicit contract of Michelson type **`(contract`** `unit`**`)`**. This cast is

Listing 4 Enhanced contract module.

```
contract type SaferAuction = sig
  storage (Pair (bidding : bool)
                (Pair (owner : address) (hi_bidder : address)))

  entrypoint close ()
  requires bidding raises "closed"        (** auction closed*)
  requires (SOURCE = owner) raises "not owner"
  ensures not bidding
  ensures (post.BALANCE = 0)
  ensures (TRANSFER_TOKENS unit BALANCE hi_bidder)

  paid entrypoint bid ()
  requires bidding raises "closed"        (** auction closed*)
  requires (AMOUNT > pre.BALANCE) raises "too low" (** low bid *)
  ensures bidding
  ensures (post.BALANCE = AMOUNT)
  ensures (post.hi_bidder = SOURCE)
  ensures (TRANSFER_TOKENS unit pre.BALANCE hi_bidder)

  invariant (post.owner = owner)
  invariant (post.bidding => bidding)
  invariant (post.hi_bidder = hi_bidder or post.hi_bidder = SOURCE)
end
```

unavoidable as a value of contract type cannot be stored. However, the cast may fail and its failure leads to an error condition that is reported with a `FAILWITH` instruction that is not covered by the contract module.[2] The entry point for closing has the same issue with the address of the owner, who is scheduled to receive the balance of the contract.

The best way to address this issue would be to assert that the addresses stored for the owner and the highest bidder always cast successly into the `(contract unit)` type so that the stated module type can be retained. Unfortunately, doing so requires control over the initialization of the storage, which is part of the deployment of the contract. At present, our design for contract modules does not include support for the deployment of a contract. Moreover, it is not clear whether it makes sense to support it as there is no guarantee that a contract will be deployed using the API generated from a contract module.

The quick fix is to add `raises` clauses for the `No entrypoing ...` message. With this fix, we successfully check the module description against the Michelson implementation of its contract. Another fix would be to rewrite the contract to enable the dynamic creation of auctions.

5 Advanced Checking

As it is expensive to invoke a contract just to find out that it fails, we propose to extend entry point specifications with preconditions along with some global invariants as shown in Listing 4. The idea is that the generated OCaml module tries to check the preconditions

[2] We generated the Michelson code for this example using the Liquidity compiler (`https://www.liquidity-lang.org/`). While the failures for `closed` and `too low` are explicit in the source program, the compiler inserts the casts into the contract type automatically using the error message `No entrypoint default with parameter type unit`.

off-chain before invoking the contract. To this end, the off-chain code needs to obtain properties like balance, storage etc of the contract, but this information is available from a Tezos node without a fee! Of course, such an off-chain check is prone to race conditions as concurrent contract invocations from other parties may interfere and change the data before our module gets a chance to invoke the contract.

Generally, a specification can refer to the values available to the contract. For example, the instructions `SOURCE`[3], `BALANCE`, and `AMOUNT` refer to the respective values. As the current `BALANCE` includes the `AMOUNT` sent with the transaction, we write `pre.BALANCE` (= `BALANCE` − `AMOUNT`) for the balance before the transaction starts and `post.BALANCE` for the balance after the transaction finishes. The latter is calculated by subtracting the amounts transferred from the current `BALANCE`. The existence of a token transfer in the returned operation list is indicated by the respective `TRANSFER_TOKENS` instruction. The components of the storage are referred to by name. We distinguish the outgoing storage by prepending `post.` as in `post.owner`.

For each clause **`requires`** Φ **`raises`** s, we take the path condition Θ for a `FAILWITH` instruction with argument matching the string s and check that $\Phi \wedge \Theta$ is not satisfiable in the context given by the initial stack for the entry point.

From all **`requires`** Φ_i and **`ensures`** Ψ_j clauses, we check that $\neg(\bigwedge \Phi_i \rightarrow \bigwedge \Psi_j)$ is unsatisfiable in the context given by the initial stack for the entry point and its corresponding final stack and path condition.

We discuss two of the preconditions to highlight the properties that need to be analyzed and where race conditions may interfere.

The precondition `SOURCE = owner` of `close` can be checked off-chain because the owner's address is part of the storage. However, it is in general unsound to perform such a test off-chain because the owner's address could change if an entry point changes that component of the storage. To safely check this precondition, all other entry points must preserve the `owner`'s address, which is indeed the case by the postcondition in line 17. This postcondition is verified as outlined above.

The situation is slightly more complex at the `bid` entry point. The failure `"closed"` is guarded by `bidding`. As the `bidding` component of the state can change, a precise prediction is not possible. A closer look reveals some subtlety. If `bidding` is true, then the flag may have changed by some interleaved call to `close`. However, if `bidding` is false, then there is no point in invoking the contract because `bidding` will never be reset to true. We address this situation by verifying the global invariant `post.bidding => bidding`.

For the failure `"to␣low"`, the analysis is very similar: we need to know that there is no successful execution of `bid` after an execution of `close`. Moreover, each invocation of `bid` raises the balance of the contract monotonically. Thus, if the off-chain check `AMOUNT > pre.BALANCE` fails, we can be sure that the contract invocation will also fail; either because someone closed the auction or because the balance is at least as high as in the off-chain sample. Checking `pre.BALANCE` off-chain is particularly simple, because it is the current balance of the account.

[3] The execution of a Michelson contract is part of a transaction, which can encompass several contract executions. The SOURCE of a Michelson contract is the originator of the entire transaction.

6 Symbolic interpretation of Michelson

Michelsym, our symbolic interpreter for Michelson, works in two stages. In the first stage, it calculates symbolic stacks between each pair of reachable instructions. The underlying symbolic domain comprises all concrete values, supports the type system, and generates a term representation for symbolic values. Michelsym collects a path predicate that is extended at each conditional, but which remains uninterpreted. If symbolic execution reaches certain instructions (most notably `FAILWITH`), Michelsym records the argument value and the path condition.

Presently, Michelsym works on Michelson files which result from compiling the examples provided with the Liquidity compiler. It generates human-readable output as well as output in the SMTlib format suitable for SMT-solvers like Z3. This output needs to be weaved together manually with the formulas generated from a contract module.

We plan to revise Michelsym so that it directly communicates with Z3 to directly check for unsatisfiable path conditions and to be better integrated with the contract module frontend.

7 Related work

Smart contract-based applications often require interaction between a smart contract on the blockchain and the outside world. However, smart contracts cannot connect to external sources on their own. This is where oracles [13, 5] come into play. Oracles act as a bridge between smart contracts and external sources. Namely, they collect and verify external information and make it available to smart contracts on the blockchain. Several research works have been conducted to provide oracle solutions for the Blockchain. Adler et al.[12] proposed a framework to provide developers with a guide for incorporating oracles into blockchain-based applications. Oracles may need to observe the state of the chain to determine what information to send. In addition, oracles transmit data from external sources to the blockchain. Therefore, they would need to have a programmatic interface to interact with the blockchain.

The basic idea of our advanced checking, namely precondition checking, is inspired by JML, the Java modeling language [11, 6], in which the behavior of program components is described as a contract between Java program and its clients. This contract specifies preconditions that must be satisfied by clients and postconditions that are guaranteed by the program. A precondition supplied with a client call must be verified before a function defined by the program is called, and the program guarantees that the postconditions are satisfied in return after the call. The original idea of using preconditions and postconditions dates back to Hoare's paper [10]. Software contracts have also been proposed for blockchain [3]. In our approach, the safe contract module in the OCaml language comes close to contracts in this sense. Several applications are based on JML [14]. Ahrendt et al. [1] propose the KeY framework for deductive software verification.

Our contract module specifies preconditions and then off-chain checks whether a user call satisfies those preconditions. Symbolic execution plays an important role in the preconditions checking in our method. A smart contract is verified against its specification in the contract module by symbolic execution. In a paper on symbolic execution [9], Hentschel et al. proposed the symbolic execution debugger (SED) platform, which is based on the KeY framework. The platform SED has a static symbolic execution engine for sequential programs.

8 Conclusion

Current blockchains often provide low-level interfaces to interact with smart contracts. These interfaces work with loosely structured without static guarantees. This paper presents ongoing research on the programmatic interaction with smart contracts on the Tezos blockchain that could benefit developers of mixed applications and oracles comprised of on-chain and off-chain parts. The approach does not provide a general API, but targets each individual smart contract by generating a specialized contract module that provides a typed high-level interface from a contract specification. In doing so, errors from contract calls are explicitly specified in a user-defined data type. A contract call is wrapped in a fully typed and integrated OCaml function. In addition, the wrapper can check preconditions before the actual call to reduce the waste of gas of a failed call.

While our conceptual approch is applicable and would be beneficial for other blockchains, the actual implementation is very much tied to the Tezos blockchain. The key asset here is the symbolic interpreter which is hardcoded for Michelson and adapted to the peculiarities of the Tezos blockchain. By targeting Michelson, our work is applicable to all languages running on Tezos, but a similar tool would have to be developed from scratch for another blockchain.

References

1 Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter Schmitt, and Mattias Ulbrich. *Deductive Software Verification – The KeY Book: From Theory to Practice*, volume 10001. Springer-Verlag, January 2016. doi:10.1007/978-3-319-49812-6.

2 Davide Ancona, Giovanni Lagorio, and Elena Zucca. A core calculus for Java exceptions. In Linda M. Northrop and John M. Vlissides, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14–18, 2001*, pages 16–30. ACM, 2001. doi:10.1145/504282.504284.

3 Massimo Bartoletti. Smart contracts contracts. *Frontiers Blockchain*, 3:27, 2020. doi:10.3389/fbloc.2020.00027.

4 Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph M. Wintersteiger. Programming Z3. In Jonathan P. Bowen, Zhiming Liu, and Zili Zhang, editors, *Engineering Trustworthy Software Systems – 4th International School, SETSS 2018, Chongqing, China, April 7-12, 2018, Tutorial Lectures*, volume 11430 of *Lecture Notes in Computer Science*, pages 148–201. Springer, 2018. doi:10.1007/978-3-030-17601-3_4.

5 Giulio Caldarelli. Understanding the blockchain oracle problem: A call for action. *Information*, 11(11), 2020.

6 Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, page 342–363, Berlin, Heidelberg, 2005. Springer-Verlag.

7 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

8 L. Goodman. Tezos – A self-amending crypto-ledger, 2014. URL: https://www.tezos.com/static/papers/white-paper.pdf.

9 Martin Hentschel, Richard Bubel, and Reiner Hähnle. The symbolic execution debugger (SED): a platform for interactive symbolic execution, debugging, verification and more. *International Journal on Software Tools for Technology Transfer*, 21, October 2019.

10 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

11 Gary Leavens and Yoonsik Cheon. Design by contract with JML, 2006. URL: `https://www.cs.ucf.edu/~leavens/JML/index.shtml`.

12 Kamran Mammadzada, Mubashar Iqbal, Fredrik Milani, Luciano García-Bañuelos, and Raimundas Matulevičius. *Blockchain Oracles: A Framework for Blockchain-Based Applications*, pages 19–34. Springer Verlag, September 2020.

13 Roman Mühlberger, Stefan Bachhofner, Eduardo Castelló Ferrer, Claudio Di Ciccio, Ingo Weber, Maximilian Wöhrer, and Uwe Zdun. Foundational oracle patterns: Connecting blockchain to the off-chain world. *Business Process Management: Blockchain and Robotic Process Automation Forum*, page 35–51, 2020.

14 Peter W. V. Tran-Jørgensen. Automated translation of VDM-SL to JML-annotated Java. *Technical Report Electronics and Computer Engineering*, 5(29), March 2017.