A tool for proving MICHELSON Smart Contracts in WHY3 *

Luís Pedro Arrojado da Horta, João Santos Reis, Simão Melo de Sousa
Release Lab., C4, DI, Universidade da Beira Interior, Portugal
NOVA-LINCS

Mário Pereira
NOVA-LINCS
DI, FCT, Universidade Nova de Lisboa, Portugal

***Abstract*—This paper introduces a deductive verification tool for smart contracts written in Michelson, which is the low-level language of the Tezos blockchain. Our tool accepts a formally specified Michelson contract and automatically translates it to an equivalent program written in WhyML, the programming and specification language of the Why3 framework. Smart contract instructions are mapped into a corresponding WhyML shallow-embedding of the their axiomatic semantics, which we also developed in the context of this work. One major advantage of this approach is that it allows an out-of-the-box integration with the Why3 framework, namely its VCGen and the backend support for several automated theorem provers. We also discuss the use of our tool to automatically prove the correctness of diverse annotated smart contracts.**

I. INTRODUCTION

Smart contracts are reactive programs that perform general-purpose computations within a blockchain and have been used to encode arbitrarily complex business logic of digital transactions. Since the use of smart contracts has been increasing significantly, and also since smart contracts cannot be changed once uploaded into a blockchain, it is of paramount importance to tackle the challenge of formally verifying their safety and correctness. The main focus of our work is the formal verification of smart contracts for the Tezos blockchain [14]. Moreover, we will lean towards the MICHELSON language and its formal specification [21].

Our approach is to make the verification process as automatic as possible. In order to do that, we chose the deductive program verification platform WHY3 [13] as the underlying proof framework tool in our smart contract verification tool. WHY3 is a framework aimed at automatic theorem proving through the use of external provers such as Alt-ergo [7], Z3 [11] or CVC4 [2]. Additionally, when a proof obligation can not be automatically discharged, WHY3 allows the user to call interactive theorem provers such as Coq or Isabelle.

This document is organised as follows: Section II discusses some of the related work in the field of formal verification of smart contracts. Section III discusses how we specified MICHELSON language in the WHY3 platform. Our axiomatic semantic will be described in Section IV. Section V explains how we generate WHYML code from MICHELSON. On Section VI we will detail two case studies. Finally, Section VII presents the main conclusions we gathered throughout the development of this work.

* Research supported by the Tezos Foundation through the "FRESCO - FoRmal vErification of Smart COntracts" project and by operation Centro-01-0145-FEDER-000019 - C4 - Centro de Competências em Cloud Computing, cofinanced by the European Regional Development Fund (ERDF) through the Programa Operacional Regional do Centro (Centro 2020), in the scope of the Sistema de Apoio à Investigação Científica e Tecnológica - Programas Integrados de ICDT

II. RELATED WORK

When it comes to formal verification of smart contracts, there are some efforts towards the design of verification platforms for said contracts. For instance, the work of *Nehai* and *Bobot* presented in [17] where they use Why3 to write smart contracts for the Ethereum blockchain [8]. Also *Bhargavan, K., et al.* developed a framework for analysis and verification of functional correctness of Ethereum (ETH) smart contracts by translation into F^* [6]. Moreover, for the same blockchain, in [3], *Abdellatif* and *Brousmiche* used the BIP framework for modeling and verifying said contracts using statistical model checking. Using the Coq Proof Assistant, *Zheng Yang* and *Hang Lei* combined symbolic execution with higher order theorem proving into a tool called FEther aimed at verifying Ethereum smart contracts [23]. The CertiK company has developed a commercial framework for formally verifying smart contracts and blockchain ecosystems [9]. *Marvidou* and *Laska* presented FSolidM in [16], a framework that allows its users to write more secure contracts for ETH using a graphical interface for designing finite state machines that will then be automatically translated into ETH smart contracts. In [20], *Sergey, I., et al.* describe SCILLA, an intermediate language for ETH smart contracts that is amenable to formal verification.

When it comes to MICHELSON formalisation, *Bernardo, B., et al.* specified a big-step semantic for Michelson using the Coq proof assistant [4] that serves as a base for a verification framework. This work differs from ours because we focus on the automation of the verification process. This fact relies on Why3 where the proof obligations are dispatched to external provers, where as in Coq the proof is made manually. The Archetype language [12] is a domain specific language that allows for formal specification of Tezos smart contracts, which in turn are translated to WHYML for use in WHY3, as a back-end. In this work, the contracts to be verified are Archetype contracts. Moreover, we chose

978-0-7381-0495-9/20/$31.00 ©2020 IEEE
DOI 10.1109/Blockchain50366.2020.00059

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on March 07,2021 at 16:04:11 UTC from IEEE Xplore. Restrictions apply.

MICHELSON as the object of our verification process, thus mitigating the need for the smart contract writer to learn yet another language for smart contract development.

III. MICHELSON SPECIFICATION IN WHY3

MICHELSON is a stack rewriting language for writing smart contracts for the Tezos blockchain. For a complete explanations of the MICHELSON language, we refer the reader to [21]. The relevant details of this language for this work will be introduced when needed.

In the MICHELSON language there are four primitive data types for constants, that we can name: `nat`, `int`, `string` and `bytes`. Additionally we have type `bool` for booleans and the optional type `Option` τ of type τ, (similar to the option type in OCaml). Some of these types are not primitive in WHY3 and thus, we had to make some choices on how to represent them.

In MICHELSON, both `int` (for integer constants) and `nat` (for natural number constants) have arbitrary precision, which means that computations with such constants are only limited by the Gas one is willing to pay. When it comes to WHY3, type `int` already has arbitrary precision, but we had to manually define type `nat` as shown in Figure 1. Namely, we model `nat` as a record type with a single field `value` add an invariant.

```
type nat = { value: int }
  invariant { value ≥0 }
```

Figure 1. Definition of type `nat` in WHY3.

WHY3 supports type `string` as built-in since Version 1.3.0. Given that a byte is a set of 8 bits, we chose to use BV8 (short for BitVector of size 8). In MICHELSON all data structures are immutable, and that property is still maintained with the corresponding types in WHY3.

In MICHELSON, comparisons between constants of the same type are possible. Figure 2 shows the definition of those comparable types in WHY3.

```
type comparable =
  | Int int | Nat Natural.nat
  | String string  | Bytes (seq Bytes.t)
  | Mutez int | Bool bool
  | Key_hash string  | Timestamp string
  | Address string
```

Figure 2. Definition of type `comparable` in WHY3.

Type `Mutez` represents *micro-tez* which is in fact the smallest unit of the Tezos blockchain token. Every operation involving Mutez is mandatory checked for over/underflows. Moreover this is one of the cases where the type system really helps, because it can assure us that we do not confuse *Mutez* for another numerical constant. The `Key_hash` type represents the hash value of a public key. Additionally, type `Timestamp` represents a date that can be written in a readable format according to RFC3339 [18], or in an optimised format, being the number of seconds since *Epoch*.

According to the specification in [21], comparison functions in MICHELSON for two given constants K_1 and K_2 must return a integer. Therefore in order to abide by the given specification we had to implement our own versions for said functions.

MICHELSON's execution stack contains only data or instructions, thus the type `data` is defined as depicted in Figure 3.

```
type data =
  | Comparable comparable
  | Key
  | Unit
  | Some_data data
  | None_data typ
  | List (list data) typ
  | Pair data data
  | Left data typ
  | Right data typ
  ...
 with instruction =
  | SEQ_I instruction instruction
  ...
```

Figure 3. Definition of type `data` in WHYML.

In order to ensure that all data is properly constructed, we defined the predicate `well_formed_data`. For the WHYML representation of the MICHELSON execution stack, we chose an immutable sequence (type `stack_t`) defined as follows:

```
type stack_t = seq well_formed_data.
```

Additionally we defined a function named `typ_infer` for determining the type of a specific element in the stack. This function gives us an extra assurance that the stack is well formed and well typed.

IV. AXIOMATIC SEMANTICS IN WHY3

In this section we present the reader with some of the more important details of our axiomatic semantics of MICHELSON in WHYML. Our approach is a shallow embedding of the MICHELSON language in WHYML. Furthermore opcodes such as `SEQ` do not need to be directly encoded given that one can take advantage of the WHYML language constructs e.g. `let ... in ...` or the sequence operator ';'.

Every MICHELSON opcode results in an abstract function in WHYML containing a set of annotations (i.e. rules). Moreover this set of rules defines the expected behaviour of that opcode and the effect it produces on the stack. All the opcodes take as input (at least) the stack and return a new stack.

As an example of such abstract function take the opcode `ADD` defined in [21] as the sum of the top two elements in the input stack, figure 4 depicts the corresponding WHYML code.

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on March 07,2021 at 16:04:11 UTC from IEEE Xplore. Restrictions apply.

```
val add (s: stack_t) (fuel: int) : stack_t
    ...
    ensures  add_post { match s[0].d, s[1].d with
        | Comparable (Int x), Comparable (Int y) →
          let res = Comparable (Int (x + y)) in
          result = (mk_wf_data res) :: s[2 ..]
        | Comparable (Int x), Comparable (Nat y) →
          let res = Comparable (Int (x + (eval_nat y))) in
          result = (mk_wf_data res) :: s[2 ..]
        | Comparable (Nat x), Comparable (Int y) →
          let res = Comparable (Int ((eval_nat x) + y)) in
          result = (mk_wf_data res) :: s[2 ..]
        | Comparable (Nat x), Comparable (Nat y) →
          let res = Comparable (Nat (add_nat x y)) in
          result = (mk_wf_data res) :: s[2 ..]
        | _ → false end }
```

Figure 4. Definition of ADD in WHYML.

For brevity, figure 4 only depicts the functional correctness post condition.

The limit of our formalisation.: In the present version of the axiomatic semantics, we have not formalised the internal details of the cryptographic operations. We have instead defined these instructions as abstract operations that follow the expected pre and post conditions.

Because the semantics of serialisation operations is not clear from the reference documentation, we also choose to abstract these operation the same way we handle cryptographic operations.

V. AUTOMATED TRANSLATION

In this section we present some of the most important details about the automatic translation from MICHELSON to WHYML. For a visual representation of the our tool structure, we refer the reader to figure 5. In order to obtain an abstract-syntax tree of a Michelson smart contract we implemented a parser in OCaml and Menhir [19]. This parser respects the syntax described on the Tezos documentation [21]. It allows us to obtain a data type that fully abstracts the syntax (with the exception of annotations) which we can then manipulate in order to generate WHYML. The automated translation to WHYML using the Why3 API is explained in subsection V-A. Additionally, a small example of a translated MICHELSON contract will be given in subsection V-B.

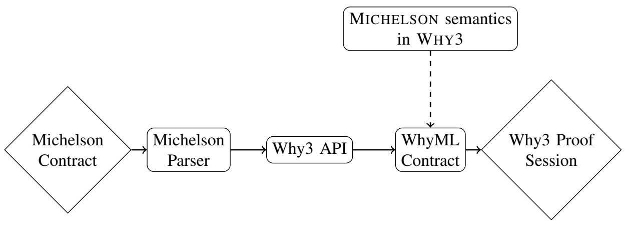

Figure 5. Visual Structure of the Implementation.

A. Why3 API

The core of our development is the translation of a MICHELSON contract into an equivalent WHYML program. Our purpose is to be able to feed the generated program to the WHY3 proof engine, in order to conduct formal verification on the original contract[1]. It is worth highlighting that our translation is completely done in-memory, *i.e.*, WHY3 *reads* the MICHELSON file and no intermediate WHY3 file is generated in order to contain the result of translation. This leads to a very smooth integration with the WHY3 framework.

The key insight of our translation mechanism is that we take the AST representation issued by the MICHELSON parser and, using the WHY3 source code as an OCaml library, we generate an AST of the WHYML language. We organise our translation code into several mutually-recursive functions, each one dealing with the translation of a different syntactic element of the MICHELSON language. Consider, for instance, the MICHELSON instruction ADD. For this instruction, our parser emits an AST containing the node I_add. To translate this add statement into this WHYML counterpart, we build the following homomorphic translation

```
let rec inst = function
  | I_add ->
     mk_expr (Eidapp (Qident (mk_id "add"), stack_fuel_args))
  ...
```

where mk_expr and mk_id are simply smart constructs for WHYML expressions and identifiers, respectively. The above OCaml code creates an application expression to our axiomatized add operation of Figure 4, where the arguments are the current stack and fuel amount. A more interesting example, and one that shows how we take advantage of underlying translation to WHYML, is the MICHELSON SEQ operation. For such MICHELSON statement, our parser issues a node of the form I_seq (i1, i2), where i1 and i2 are the two instructions composing the sequence. Our translation engines features the following code for this case:

```
| I_seq (i1, i2) -> mk_expr (Elet (mk_id "__stack__",
                   false,  Expr.RKnone,   inst i1, inst i2))
```

which builds the expression `let __stack__ = i1 in i2`. The Boolean constant `false` above tells WHY3 that this is a non-ghost expression, while the `Expr.RKnone` indicates that this is a simple locally-defined symbol, with no direct translation to a purely-logical symbol. Let us note that the tree-like data type produced by our translation corresponds to the AST issued by the WHYML parser, hence no typing or name resolution information is present at this point.

[1]The correctness of the generated WHYML program implies the correctness of the original MICHELSON contract. At the moment, such an argument is based on the informal reasoning that the semantics of MICHELSON is captured by the our axiomatic semantics developed in WHY3. A more rigorous rationale, which we plan to develop as future work, must provide mathematical and/or formal evidence that MICHELSON operational semantics conforms to our axiomatic encoding

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on March 07,2021 at 16:04:11 UTC from IEEE Xplore. Restrictions apply.

Having defined our MICHELSON to WHYML transformation function, we want to integrate it with the WHY3 framework in a completely transparent fashion for the end user. This means that we want to use the WHY3 proof engine over a MICHELSON contract, as if this was the native language of the framework. One can easily extend the WHY3 framework with the support for new input languages via its plugin capabilities. This is as simple as providing a parser and a translation function from the source language into one of the WHYML internal AST. Finally, in order to register the newly-developed plugin into the WHY3 configuration base, one simply states the extension of files that should be processed by the devised translation. In our particular case, we write the following:

```
let () =
  Env.register_format mlw_language "michelson" ["tz"]
    read_channel ~desc:"Michelson format"
```

Here, `mlw_language` indicates that the target of our translation is a WHYML program and `read_channel` is the function that calls the MICHELSON parser and feeds the produced AST to our transformation mechanism. With all this machinery in place, one can then call WHY3 directly on a `.tz` file. For instance, if one wishes to formally verify the contract contained in file `foo.tz`, using our plugin, the command line would be

```
$ why3 ide foo.tz
```

which opens the WHY3 graphical Integrated Development Environment over the result of the MICHELSON contract translation.

B. A Trivial Example

For a better understanding of this automated translation, we present the reader with a visual toy example. Consider the MICHELSON contract shown in figure 6.

```
parameter nat;
storage nat;
code { UNPAIR; ADD;
       NIL operation; PAIR };
```

Figure 6. Toy example of a MICHELSON contract.

This is a very simple contract, in fact it takes the `nat` it received as parameter and adds it to the `nat` in the storage. Basically it just adds two natural numbers. The MICHELSON contract does not contain any pre or post conditions, but our tool is able to directly infer four safety conditions, namely about the length and type of both the input and the output stacks. The WHYML code generated by our tool is shown in figure 7.

Using only the `split_vc` transformation this example generates 26 verification conditions that are quickly dispatched by Alt-ergo[7].

```
use axiomatic.AxiomaticSem
use dataTypes.DataTypes
use seq.Seq
use int.Int
let contract (__stack__: stack_t) (__fuel__: int) : stack_t
  requires { (length __stack__) = 1 }
  requires { __fuel__ > 0 }
  requires { (typ_infer (d (__stack__[0]))) =
        (Pair_t (Comparable_t Nat_t ) (Comparable_t Nat_t )) }
  ensures  { (length result) = 1 }
  ensures  { (typ_infer (d (result[0]))) =
        (Pair_t (List_t Operation_t ) (Comparable_t Nat_t )) } =
  let __stack__ =
    let __stack__ = unpair __stack__ __fuel__ in
    (let __stack__ = add __stack__ __fuel__ in
    (let __stack__ = nil_op __stack__ __fuel__ Operation_t  in
    (pair __stack__ __fuel__))) in
  __stack__
```

Figure 7. The WHYML generated code for the toy example.

VI. CASE STUDIES

In this section we discuss two case studies, namely the multisig and factorial smart contracts and explain how safety and functional correctness can be proved within our tool. For the sake of brevity we will elaborate the proof of safety for the multisig smart contract and on the functional correctness of the factorial smart contracts. We will detail the proof of correctness of the factorial smart contract since this contract highlight particularly well our purpose to show the advantages but also the drawbacks of our approach.

A. Multisig

There are several versions of the multisig contract, and the one we used can be found in [22]. We separated the multisig contract into three parts. The first one is the majority of the contract, the second one is the loop which iterates over the list of keys and optional signatures (`iter_multisig`) and finally, the third part (`outer_if_left`) is where the operation requested by the signers is produced. For the sake of brevity, these fuctions are not depicted here.

Figure 8 contains the part of the `multisig` function that represents the contract. We ask the reader to notice that it has only two pre conditions and two post conditions regarding the size and the type of the stack. In order to prove the last post condition, we had to equip some of the code in the contract with some additional typing information. An example of such typing information is the one below the `iter_multisig` line. Furthermore, this complement was necessary to help the SMTs check some of the pre conditions needed for the instructions in the middle.

This code generated a total of 758 VCs, 750 of which were proven by Alt-ergo [7], Z3 [11] and CVC4[2] proved 4 verification conditions each.

B. Factorial

The contract depicted in figure 9 is the MICHELSON version of the factorial calculation. This contract calculates the factorial of a given natural number interactively. The

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on March 07,2021 at 16:04:11 UTC from IEEE Xplore. Restrictions apply.

```
let multisig (in_stack: stack_t) (fuel: int) : stack_t
    requires { fuel > 0 }
    requires { length in_stack = 1 }
    requires { typ_infer in_stack[0].d =
            Pair_t parameter storage }
    ensures  { length result = 1 }
    ensures  { typ_infer result[0].d =
            Pair_t (List_t Operation_t) storage }
    raises { Failing }
  =
let s = unpair in_stack fuel in
let s = swap s fuel in
let s = dup s fuel in
...
let s = iter_multisig s fuel
ensures {
    typ_infer result[0].d = Comparable_t Nat_t ∧
    typ_infer result[1].d = List_t (Option_t Signature_t) ∧
    typ_infer result[2].d = Comparable_t Bytes_t ∧
    typ_infer result[3].d = Or_t
        (Pair_t (Comparable_t Mutez_t) (Contract_t Unit_t))
        (Or_t
            (Option_t (Comparable_t Key_hash_t))
            (Pair_t (Comparable_t Nat_t) (List_t Key_t))) ∧
    typ_infer result[4].d = storage
} in

...
```

Figure 8. Part of the multisig contract in WHYML.

contract receives as parameter the number whose factorial is going to be calculated and stores the result in the storage. It starts by dropping the previous storage and pushes an initial accumulator and iterator as the value 1. Then it compares the parameter value with 0 and if it's different, it enters the loop to calculate the factorial.

```
parameter nat;
storage nat;
code {  CAR; PUSH @index nat 1; DUP @acc;
        DIP 2 { DUP; PUSH nat 0; COMPARE; NEQ };
        DIG 2;
        LOOP { DIP { DUP;
                     DIP { PUSH nat 1; ADD @ipp } };
               MUL;
               DIP { DIP { DUP };
                     DUP;
                     DIP { SWAP };
                     COMPARE; LE };
               SWAP };
        DIP { DROP; DROP };
        NIL operation; PAIR };
```

Figure 9. Factorial MICHELSON contract.

Inside the loop *body*, the stack has size three, where the top element is the temporary result, the middle element is the index of the iteration and the bottom element is the input parameter. Since the *body* of the loop is where the computation actually happens, we will focus on the respective portion of WHYML code depicted in figure 10. For shortness we omitted typing information in between instructions as well as size and length pre and post conditions. The only specification that we left was the one regarding functional correctness.

The first pre condition assures us that the value stored at

```
let loop_body (s: stack_t) (fuel: int) : stack_t
    requires { match s[0].d,s[1].d with
        | Comparable(Nat res),Comparable(Nat n) →
                fact (n.value - 1) = res.value
        | _ → false end }
    ensures  { match s[0].d,s[1].d with
        | Comparable (Nat res_old), Comparable (Nat n_old)→
            fact n_old.value = n_old.value  * res_old.value
        | _ → false
         end }
    ensures  { match s[1].d, result[1].d with
        | Comparable (Nat i), Comparable (Nat b) →
                fact i.value  =  b.value
        | _ → false end }
=
 ...
  let s = mul s fuel in
  let s =
    let top = s[0] in let s = s[1..] in (* DIP *)
    let s =
      let top = s[0] in let s = s[1..] in (* DIP *)
      let s = dup s fuel in
    push s fuel top in
    let s = dup s fuel in
    let s =
      let top = s[0] in let s = s[1..] in  (* DIP *)
      let s = swap s fuel in
    push s fuel top in
    let s = compare_op s fuel in
    let s = le s fuel in
  push s fuel top in
  swap s fuel
```

Figure 10. Factorial WHYML contract.

the top of the input stack is in fact the value of factorial up to the previous iterations. The last post condition ensures that the value stored at the top of the result stack is the value of factorial up to the current iteration. This code generated 2890 VCs, of which 2671 were proven by Alt-ergo[7], and the remaining 219 by Z3[11].

VII. CONCLUSIONS

In this paper we presented a tool for automated formal verification of MICHELSON smart contracts. Moreover, it is the result of several implementations also described in this document, namely a MICHELSON parser, an axiomatic semantics and a translation function, all leading to a shallow embedding of MICHELSON in WHYML.

The first steps to the automatic proof in WHY3 of manually annotated MICHELSON smart contracts were done, since we were able to use our axiomatic semantics and our WHY3 plugin to successfully write and prove several MICHELSON contracts. Furthermore, the plugin development proved itself simple, due to the fact that the WHY3 platform exposes its API as an OCAML library.

In practice we found that some of the proof trees were bigger than expected and required user intervention, thus threatening our main purpose of automation. We are aware that this is a consequence of our encoding of MICHELSON types as tree-like data structures. One possible solution is to go even further in our shallow embedding and try to map as much as possible every MICHELSON type directly

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on March 07,2021 at 16:04:11 UTC from IEEE Xplore. Restrictions apply.

into WHYML native types. Moreover this would allow us to remove some algebraic constructors from our definition, leading us to a platform that allows the user to conduct formal verification of MICHELSON written smart contract with an elevated degree of automation.

As a final thought, we think that proving MICHELSON contracts has a certain advantage over proving some other formulation, since what is effectively executed is the MICHELSON smart contract and also because this approach can be used a back-end in developing reliable smart contract in any higher level language such a LIGO[1] or SmartPy[15].

Nevertheless, smart contracts developers will implement their smart contracts in a higher level language than MICHELSON. In this setting, it is also relevant to be able to formally prove these smart contracts at a level that programmers understand and be involved with. So an interesting long-term line of work to explore is to connect our tool with certifying-certified compilation techniques and platforms. For instance, we should evaluate how the integration of our tool with, *e.g.*, the Archetype platform [12], that also makes use of WHY3, can benefit the automatic proof of MICHELSON smart contracts. We should also evaluate how our tool could benefit from rigorously designed compilers as the one designed for Albert [5] to Mi-cho-coq [4]. For the WHY3 platform, such an endeavour could make use of the techniques introduced by *Clochard, M., et al.* in [10].

REFERENCES

[1] G. Alfour. LIGO. URL: https://ligolang.org/.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[3] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 3–12. Ieee, 2006.

[4] B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, and J. Tesson. Mi-cho-coq, a framework for certifying tezos smart contracts. In *FMBC'19: Workshop on Formal Methods for Blockchains*, 2019.

[5] B. Bernardo, R. Cauderlier, B. Pesin, and J. Tesson. Albert, an intermediate smart-contract language for the Tezos blockchain. *arXiv:2001.02630 [cs]*, Jan. 2020. http://arxiv.org/abs/2001.02630 arXiv:2001.02630.

[6] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96. ACM, 2016. https://doi.org/10.1145/2993600.2993611 doi:10.1145/2993600.2993611.

[7] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The alt-ergo automated theorem prover, 2008, 2013.

[8] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[9] CertiK. : Building fully trustworthy smart contracts and blockchain ecosystems, 2019. [Online; https://certik.org/docs/white_paper.pdf, accessed April-2020].

[10] M. Clochard, C. Marché, and A. Paskevich. Deductive verification with ghost monitors. 4(POPL):1–26, Jan. 2020. https://doi.org/10.1145/3371070 doi:10.1145/3371070.

[11] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 337–340, Berlin, Heidelberg, 2008. Springer. https://doi.org/10.1007/978-3-540-78800-3_24 doi:10.1007/978-3-540-78800-3_24.

[12] R. B. S. P. e. Duhamel, G. Archetype: a tezos smart contract development solution dedicated to contract quality insurance, 2019. [Online; https://docs.archetype-lang.org/ accessed 14-April-2020].

[13] J.-C. Filliâtre and A. Paskevich. Why3—where programs meet provers. In *European Symposium on Programming*, pages 125–128. Springer, 2013.

[14] L. Goodman. Tezos: A self-amending crypto-ledger position paper, 2014.

[15] F. Maurel and S. C. Arena. SmartPy. URL: https://smartpy.io/.

[16] A. Mavridou and A. Laszka. Designing secure ethereum smart contracts: A finite state machine based approach, 2017. http://arxiv.org/abs/1711.09327 arXiv:1711.09327.

[17] Z. Nehai and F. Bobot. Deductive proof of ethereum smart contracts using why3. In *FMBC'19: Workshop on Formal Methods for Blockchains*, 2019.

[18] C. Newman and G. Klyne. Date and Time on the Internet: Timestamps. RFC 3339, July 2002. https://doi.org/10.17487/RFC3339 doi:10.17487/RFC3339.

[19] F. Pottier and Y. Régis-Gianas. Menhir parser generator, 2018. [Online; http://gallium.inria.fr/ fpottier/menhir/, accessed April-2020].

[20] I. Sergey, A. Kumar, and A. Hobor. Scilla: a smart contract intermediate-level language, 2018. http://arxiv.org/abs/1801.00687 arXiv:1801.00687.

[21] Tezos Foundation. Michelson: the language of smart contracts in tezos, 2020. [Online; https://tezos.gitlab.io/whitedoc/michelson.html#language-semantics last accessed 15-April-2020].

[22] Tezos Foundation. Multisig conctract, 2020. [Online; https://tezos.gitlab.io/whitedoc/michelson.html#multisig-contract last accessed 15-April-2020].

[23] Z. Yang and H. Lei. Fether: An extensible definitional interpreter for smart-contract verifications in coq. *IEEE Access*, 7:37770–37791, 2018.

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on March 07,2021 at 16:04:11 UTC from IEEE Xplore. Restrictions apply.