Model Checking Smart Contracts for Ethereum

Thomas Osterland[a], Thomas Rose[a,b]

[a]*Fraunhofer Institute for Applied Information Technology FIT, Schloss Birlinghoven, 53754 Sankt Augustin, Germany*
[b]*RWTH Aachen, Ahornstraße 55, 52074 Aachen, Germany*

Abstract

The striking characteristics of smart contracts is that no entity can manipulate their execution. However, in case there is an error in the implementation of the smart contract this beauty turns into a beast. Updating a once instantiated smart contract is complex and almost impossible. Thus, when signing or initially instantiating a smart contract in the blockchain one has to be absolutely sure that the program code works as expected. Especially in cases with risk of major financial losses.

We are especially interested in checking whether a smart contract meets its formal specification and how to proof the consistency of interactions among smart contracts. In this paper we formalize the syntax and semantics of the Ethereum smart contract programming language Solidity and propose a translation function that allows the generation of PROMELA models from Solidity programs. Then we can employ the SPIN model checker for checking program properties.

Keywords: Model Checking, Blockchain, Ethereum, Smart Contracts, SPIN

1. Introduction

The basic idea of smart contracts is to digitize conventional contracts such that those can be automatically and independently enforced once they have

*Corresponding author. Tel.: +49 2241 14 3618; fax +49 2241 14 2080
Email addresses: thomas.osterland@fit.fraunhofer.de (Thomas Osterland), thomas.rose@fit.fraunhofer.de (Thomas Rose)

been specified. In addition, a smart contract standardizes the representation of contracts and fosters the transparency of its contents. However, there is still an unbiased intermediary required [1] for testing the compliance or the correct execution of contracts. The need for intermediaries change once blockchain technology is considered. Basically the blockchain becomes the intermediary. Every interaction with a smart contract, whether originating from an external actor or another smart contract, results in a transparent and immutable transaction in the blockchain. Thus the execution of smart contracts is transparent and comprehensible. Furthermore, because of the nature of the blockchain, no single entity can tamper the execution and manipulate the process.

Smart contracts open a variety of opportunities for different application scenarios. For one thing, smart contracts can be used to digitize and automate business processes. In a car sharing scenario a smart contract can automatically unlock an IoT enabled car after receiving the required funds and checking that the driver has a valid driving license. But also machine-to-machine economy can be elevated to new application spheres, when using smart contracts for automating business transactions. The autonomous car of the future refuels and pays by sending money from its wallet to the smart contract of the petrol station. Additionally smart contracts allow the autonomous or semi-autonomous participation at strongly automated distributed market places, as for instance, in the case of energy markets, where every consumer is also a producer of energy. Hence, smart contracts are a key ingredient for implementing flexible business networks.

A popular blockchain technology that supports smart contracts is *Ethereum*. Since Ethereum is a public blockchain, every person can read the information stored in the blockchain or can interact with smart contracts residing in the blockchain. Ethereum needs a mechanism to ensure that no smart contract stalls the blockchain by executing an infinite loop for instance. Therefore, Ethereum requests a fee for any blockchain transaction. It uses the Ethereum currency *ETH* for this clearing. So when interacting with a smart contract a transaction is created that holds a certain amount of money to pay the execution fees.

35 When the money is depleted before the smart contract execution ends it will be cancelled.

Smart contracts in Ethereum are written in special purpose languages, as e.g., *Solidity* or *Serpens*. These programs will be translated into *Ethereum Virtual Machine* (EVM) byte-code and than executed by a stack machine. Cur-
40 rently Solidity is the most popular and most advanced language for development Ethereum smart contracts.

Despite the immutability of program execution due to blockchain technology there is the danger that this most important feature turns involuntarily into a disadvantage: Nobody can change a smart contract once instantiated. Since
45 smart contracts are written in program code, they face comparable problems as any software engineering project: Ensuring the correct behaviour of programs. So when instantiating or signing a smart contract in the blockchain one need to be absolutely sure that the program code works as expected and that there are no cases in which other parties can gain an advantage.

50 This challenge is further compounded by increasing the complexity of smart contracts. Decentralized Autonomous Organizations *DAO* are built-upon smart contracts. The entire organizational structure and the processes for managing the transactions across business partners are specified with smart contracts. Hence, the correctness of implementation is vital for organizations specified as
55 DAO, i.e. to enable broad usage in situations where different persons need to collaborate without actually knowing or trusting each other and with the risk of high financial losses.

This paper is organized as follows. First, we situate the general notion of correctness in software engineering and relate major lines of research in Section
60 2. Section 3 then focuses on existing approaches to verify the correctness of smart contracts. We analyze those approaches and demarcate our approach, as depicted in Figure 1. Basically, we start with a specification of syntactically well-formed program code via its operational semantics towards a logic-based model of program flow that can be checked by a model checker. In Section 4
65 we formalize the subset of the Solidity syntax that we intend to translate into

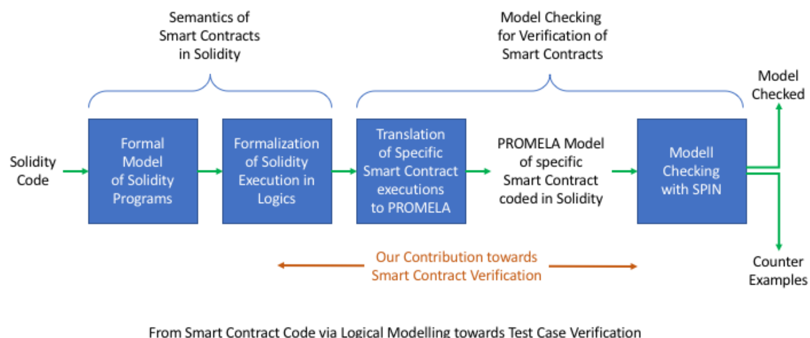

Figure 1: Schematic Overview of the Verification Process

PROMELA and provide semantics to determine our understanding of what the execution of a certain Solidity expression means. We additionally introduce the PROMELA syntax. In Section 5 we provide the problem statement that we are going to solve and describe our translation function to convert Solidity programs into PROMELA models. We implemented our approach what is described in Section 6 and did a small evaluation in Section 7. We conclude the paper in Section 8. To just get an overview on the general idea of the approach, the well-disposed reader can skip the more formal sections: Section 4 and Section 5.

2. Assessing the Correctness

Software Engineering has witnessed the invention of a variety of methods for assuring the correctness of programs ranging from dedicated requirements engineering approaches towards quality assurance of coding. In this context, we focus on coding.

Finding bugs in program code is certainly not a trivial task. Even the *methodical testing* of small programs can be a challenge. This is proven by the fact that everybody experiences software in her daily life that has bugs and

does not perform as expected. In software engineering one major approach to improve the quality of software, that means to reduce the number of bugs, is the application of exhaustive software tests. Thereby aspects of the software on different levels are tested. For instance on a low level every single method is tested with unit tests, while on a high integration level the interaction of different components is tested. The problem of software tests is that only those bugs can be found for which tests are created. That means it is only possible to find those bugs that are within the test coverage, but it is not always possible to anticipate every single case that can lead to a software failure.

Alternatively *formal verification* aims to verify the correctness of software based on a formal specification. It allows to proof program correctness, i.e. the absence of bugs, with respect to a formal specification. At first glance this seems to be a much better approach, but there is still the possibility to make errors in the specification and applying formal verification commonly requires much more effort than employing software tests.

In general there exists two different formal verification approaches: There is the class of *deductive verification* methods which use logical calculi to formalize the desired behavior of a program and apply theorem provers to test whether a program meets its specification. The disadvantage of this approach is that the proving process can become very elaborate and a person who applies deductive verification needs to deeply understand the program.

Another approach is *model checking*. The idea behind model checking is to analyze the program states that a program can reach. A program state represents a mapping of program variables to certain values of the corresponding data type domain. So for instance a program that consists of only one integer variable has a potential state space depending on the number of values the integer variables can obtain. Model checking is a potential one-click solution that allows to test a model that describes the behavior of the program and is used to derive the program states while testing those with respect to a set of properties [2, 3]. In case it finds a violation of the properties it provides a counter example that can be used to debug the model. Disadvantages of the

model checking approach is that the state space to be analyzed can become extremely large even for small programs. This phenomenon is also known as the *state space explosion problem.*

However, the characteristics of smart contracts running on the Ethereum blockchain as mentioned in Section 1 seem to encourage the use of model checking. Moreover, the execution of smart contracts is limited by the execution fees. This can be used to reduce the state space that must be considered during verification.

Applying formal verification entails in general an investment of time and money as well as an extension of the development process of a software project. It makes only sense in cases where either human lives are at risk or when high financial investments justify prolonged development periods. Since smart contract can be used to automate business processes that are instantiated several thousand times a day or that handle a large amount of value, it appears quite feasible to invest a part of the development budget into formal verification of the smart contract to ensure its correctness.

One example of a renowned model checker is *SPIN* [4], that was used to verify the software of the mars vehicle *Curiosity* [5]. It is developed by Gerard Holzmann and was initially designed to verify the correctness of communication protocols. Since then it is used in many different applications in research, teaching and industry [6]. Models to be checked are represented in the language *PROMELA* and can than be tested against temporal logic expressions that either represent the expected behavior of the program or behavior that must never occur.

Using the SPIN model checker for our purposes, we first have to specify the formal semantics of Solidity code and its execution on the Ethereum blockchain. Then, this formal model has to be translated into a PROMELA model before model checking can start. Moreover, the conditions that limit the correct execution boundaries of the smart contract have to be formalized in temporal logic such that we can test the correctness of the smart contract.

3. Related Work

145 Besides the problem of existing bugs in the program code that might be voluntarily or involuntary exploited by signers of the contract there is also the need to protect against malicious attackers that try to exploit the faulty implementation to either prevent its correct execution or to gain an advantage in the process. One popular example where an attacker was able to exploit the incor-
150 rect implementation of a smart contract is known as the DAO Hack [7]. The attacker was able to exploit a faulty splitting mechanism with a stack reentry attack and thereby captured Ether valued at several million USD. This problem could have been detected by applying formal verification.

However, the semantics of Solidity - the most prominent programming lan-
155 guage to implement Ethereum smart contracts and also the language in which the popular DAO was implemented - leads programmers to make a certain class of errors, especially regarding the handling of errors and exceptions [8]. An overview of the security vulnerabilities of the Solidity syntax and resulting problems is given in [9]. Future versions of Solidity might improve these aspects
160 to support the developer in implementing more secure smart contracts, but the deliberate application of formal verification will increase the security of smart contracts and thus reduce the risk of their usage.

Formal verification of software in general is not a new subject of research [2], however, due to the novelty of the blockchain technology - it was introduced in
165 2008 [10] - the specialized formal verification of smart contracts is relatively new and many different approaches are subject of current research: [11] introduced *Oyente* an interpreter that analyzes the byte-code of smart contracts based on symbolic execution. The research results are promising, but at the moment it does not support the complete EVM language and instead covers a restricted
170 subset, called *EtherLite*. Since the approach is not sound its application can lead to false alarms as mentioned by [12].

[13] follow an approach to verify smart contracts by translating a restricted set of Solidity and byte-code expressions into the $F*$ language that is designed

to be easily verifiable. To check that an F* program follows its specifications, 175 SMT (Satisfiability Modulo Theories) solving and manual proofs are required. Although the potential is substantially, checking Solidity smart contracts can require manual proofs and some important language constructs are currently not supported, e.g., loops.

A similar approach utilizes *Why3*, a deductive program verifier that inter- 180 prets a special purpose language *WhyML* to verify whether a program meets its specifications [14]. A Why3 backend that is capable to handle Solidity code is implemented in the Solidity Web IDE and can be used to verify contracts, although at the moment there is only a small subset of Solidity expressions supported.

185 [15] proposes two different approaches to verify Ethereum smart contracts. Path *a* tries to verify byte-code by utilizing the theorem prover *Isabelle* and expressing the Ethereum virtual machine and smart contracts in Higher Order Logic (HOL). Path *b* tries to verify Solidity programs. However, this second path is not yet researched and also path *a* is at a very early development state 190 with for our best knowledge no practical results yet.

[12] propose *ZEUS* a program that utilizes abstract interpretation and symbolic model checking for verifying the correctness of smart contracts. It supports the user in creating fairness criteria and employs static analysis to map the fairness expressions to certain states in the program. Solidity is translated into 195 LLVM (Low Level Virtual Machine) byte-code and automatically injects assertions based on policies provided. [12] claim that their approach is sound that means there are no false negatives and that it outperforms *Oyente*.

Different approaches exist to improve the security and simplify the verification of smart contracts by relying on special languages. [16] propose to use logic 200 based languages to formulate smart contracts, while [17] introduce *SCILLA* an "intermediate-level" programming language. The idea in both cases is to utilize the characteristics of a language that is easy to verify and to understand and simplify the development of secure smart contracts. This approach is also supported by the fundamental work of [18] to fully formalize the semantic behavior

of the Ethereum virtual machine.

Manticore [19] is a symbolic execution tool that supports in its recent version the analysis of Ethereum Virtual Machine Code. Although the features are limited yet, it also provides an interface to an EVM disassembler that allows the reverse engineering of Solidity smart contracts. The tool can be used to identify problems in Ethereum smart contracts.

Another tool for analyzing the security of smart contracts is *Mythril* [20]. It utilizes concolic analysis, taint analysis and control flow checking on the byte-code level of Ethereum smart contracts. The tool *Solgraph* visualizes the control flow of Solidity smart contracts [21]. It generates a dot file from Solidity files and uses color coding for identifying potential problems as calling external addresses or payable methods. However these last tools are rather for visualizing and supporting the manual analysis process of smart contracts, but cannot be considered as full-fledged formal verification approaches.

Since a great number of smart contracts dispose of substantial amounts of Ether, which corresponds to a substantial amount of USD the necessity to research methods and approaches to ensure the security and correctness of smart contracts is clearly there. The aforementioned approaches employ symbolic model checking and deductive program verification for checking the correctness of smart contracts. They are specialized on detecting problematic code patterns and provide considerable success, what is remarkable demonstrated by applying their verification approaches on a large set of smart contract stored in the Ethereum blockchain. However, no approach uses model checking for the verification and we are interested in researching how this approach performs on this tasks. We additionally concentrate on deriving the specified semantics of a smart contract from its supposed behavior and try to engineer an approach that tests a smart contract in the context of other interacting smart contracts. So we are additionally interested in the consistency of interacting smart contracts. Although most of the intended functionality is not yet implemented this paper provides the theoretical base of our approach and hopefully encourages rich discussions.

4. Preliminaries

Pursuing the aim to translate Solidity smart contracts into PROMELA models, we start by formally defining the syntax and semantics of the subset of Solidity expressions that are supported by our implementation. This determines our understanding of the Solidity semantics and provides a solid base for further research. We also introduce the syntax of PROMELA and elaborate on its semantics to explain our intentions regarding the design of the translation function and how the semantics of Solidity and the Ethereum blockchain that executes the smart contracts is equivalently translated into the PROMELA semantics. We introduce a number of helper function that are basically syntactic sugar for clarification and to improve readability of the definitions.

4.1. Syntax Definition of the Solidity Language

We define the syntax of a subset of the Solidity language as a context-free grammar. Thereby, as many other approaches we support only a subset of the Solidity expressions. So we do not support inheritance or multiple contracts. Also structs and strings are not covered in this first version.

A valid solidity program starts with a pragma definition that specifies the used Solidity version and consists of one contract that includes an arbitrary number of variable declarations, mapping declarations, event definitions and functions. The complete program logic of a Solidity program such as branching, assignments and loops is encoded in functions. Currently the supported data types are restricted to the types *bool, unsigned integer, integer* and *address.*

Definition 1 (Solidity Syntax)**.** *Let* $G_{Sol} = (N_{Sol}, \Sigma_{Sol}, P_{Sol}, S_{Sol})$ *be the context-free grammar describing the partial syntax of a Solidity program as follows:*

$$
\begin{aligned}
&S_{Sol} \rightarrow \textit{pragma solidity } z_1 . z_2 . z_3 \ ; \ C \\
&C \rightarrow \textit{contract } \alpha \ \{ \ G \ \} \\
&G \rightarrow M\ G| \ \ E\ G| \ \ F\ G| \ \ A\ G| \ \ \varepsilon \\
&M \rightarrow \textit{mapping } (\ T \Rightarrow T \ ) \ \alpha \, ; \\
&E \rightarrow \textit{event } \alpha \, (P) \, ; \\
\\
&F \rightarrow \textit{function } \alpha \, (P) \ F_2 \ \{ \ O \ \} \\
&F_2 \rightarrow \textit{returns } (R) \ | \ \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
O \rightarrow\ & \textit{if}\ (\ B\ )\ \{\ O\ \}\ |\ O;\ O \\
& |\ \textit{while}\ (B)\ \{\ O\ \}\ |\ T\ \alpha \\
& |\ A\ |\ \textit{return}\ B\ |\ \alpha\ () \\
\\
A \rightarrow\ & A_2\ =\ B\ |\ A_2\ \mathrel{+}=\ B \\
& |\ A_2\ \mathrel{-}=\ B\ |\ A_2\ *=\ B \\
& |\ A_2\ /=\ B\ |\ A_2{+}{+}\ |\ A_2{-}{-} \\
& |\ {+}{+}A_2\ |\ {-}{-}A_2 \\
A_2 \rightarrow\ & T\ \alpha\ |\ V \\
\\
B \rightarrow\ & \textit{true}\ |\ \textit{false}\ |\ Z\ |\ B\ \&\&\ B\ |\ B\ ||\ B \\
& |\ B == B\ |\ B \neq B \\
& |\ Z < Z\ |\ Z > Z \\
& |\ Z \leq Z\ |\ Z \geq Z \\
\\
Z \rightarrow\ & V\ |\ z\ |\ Z + Z\ |\ Z - Z\ |\ Z * Z \\
& |\ Z \setminus Z\ |\ Z\ \%\ Z \\
\\
V \rightarrow\ & \alpha\ |\ \alpha\ [\ Z\ ] \\
R \rightarrow\ & T\ |\ T\ \alpha \\
P \rightarrow\ & P,\ P\ |\ T\ \alpha\ |\ \varepsilon \\
\\
T \rightarrow\ & \textit{uint}\ |\ \textit{int}\ |\ \textit{address}\ |\ \textit{bool}
\end{aligned}
$$

with $z, z_1, z_2, z_3 \in \mathbb{Z}$, α *an arbitrary label of the form* $x_1 x_2 \cdots x_n$, $x_1 \in \{a - zA - Z\}$ *and* $x_i \in \{a - zA - Z\} \cup \mathbb{N}$ *for* $n \geq i > 1$ *and* $n \in \mathbb{N}$*.*

In general comments introduced by // or /···*/ can be placed at arbitrary positions in the program code and will be ignored.*

Based on the semantics of context-free grammars we define the set of variables Var_p for a program p holding all the variables that are declared in the entire program. The set GlobVar_p holds those variables that are declared on contract level, while the set of local variables LocVar_p contains the variables that are declared within the functions. Thereby a global variable is accessible in functions but a local variable declared in one function is not accessible from within another function. However, it is possible to define two variables with equal name in two different functions.

We additionally define the set of mappings Map_p as the set containing the mappings declared in the program p. A mapping must be declared on contract level. So there exists no *local* mappings.

Definition 2 (Solidity Variable and Mapping)**.** *Let $G_{Sol} = (N_{Sol}, \Sigma_{Sol}, P_{Sol}, S_{Sol})$ be the context-free grammar of solidity programs, $\Rightarrow \subseteq X^+ \times X^*$ the derivation relation following the semantics of context free grammars and $X = \Sigma_{Sol} \cup N_{Sol}$, while $*/+$ represent the Kleene closure. For $T \in P_{Sol}$, $p \in L(G_{Sol})$ we define the set of variables:*

$$Var_p = \{\alpha \in \Sigma^*_{Sol} \mid \exists \gamma_1, \gamma_2 \in X^* : S_{Sol} \Rightarrow^* \gamma_1 T \alpha \gamma_2 \Rightarrow^* p\}$$

For $A, G, T \in P_{Sol}$ we define the set of global variables:

$$GlobVar_p = \{\alpha \in \Sigma^*_{Sol} \mid \exists \gamma_1, \gamma_2, \gamma_3 \in X^* : S_{Sol} \Rightarrow^* \gamma_1 A G \gamma_2 \Rightarrow^* p \wedge (A \Rightarrow T\alpha\gamma_3 \vee A \Rightarrow \alpha\gamma_3)\} \quad (1)$$

The set of local variables is then given by:

$$LocVar_p = Var_p \setminus GlobVar_p \quad (2)$$

For $M, G, T \in P_{Sol}$ the set of mappings is defined by:

$$Map_p = \{\alpha \in \Sigma^*_{Sol} \mid \exists \gamma_1, \gamma_2 \in X^* : S_{Sol} \Rightarrow^* \gamma_1 M G \gamma_2 \Rightarrow^* p \wedge (M \Rightarrow \texttt{mapping}\ (T \Rightarrow T)\ \alpha;)\} \quad (3)$$

The set of functions Func_p holds all the functions declared in the program, while the set Const_p refers to the constructor of the smart contract that is executed at initial instantiation. Within the syntax we do not distinguish between a function and a constructor.

Note that in recent Solidity versions a constructor is not longer identified by having the same name as the contract itself, but by being introduced with the keyword *constructor*. In the remainder of the paper we stick to the older variant. However, there are only minor adjustments necessary to support also the new variant.

Definition 3 (Function and Constructor)**.** *Let* $G_{Sol} = (N_{Sol}, \Sigma_{Sol}, P_{Sol}, S_{Sol})$ *be the context-free grammar of solidity programs,* $\Rightarrow \subseteq X^+ \times X^*$ *the derivation relation following the semantics of context free grammars with* $X = \Sigma_{Sol} \cup N_{Sol}$*, while* $*/+$ *represent the Kleene closure. For* $F, F_2, G, O, P \in P_{Sol}$ *we define the set of functions:*

$$\begin{aligned} Func_p = \{\alpha \in \Sigma^*_{Sol} \mid \exists \gamma_1, \gamma_2 \in X^* : \\ S_{Sol} \Rightarrow^* \gamma_1 F G \gamma_2 \Rightarrow^* p \\ \wedge (F \Rightarrow \texttt{function}\, \alpha(P)\ F_2\ \{O\})\} \end{aligned} \tag{4}$$

For $G \in P_{Sol}$ *the constructor is defined by:*

$$\begin{aligned} Const_p = \{\alpha \in Func_p \mid \exists \gamma_1, \gamma_2 \in X^* : \\ S_{Sol} \Rightarrow^* \gamma_1 \texttt{contract}\, \alpha\ \{G\} \gamma_2 \Rightarrow^* p\} \end{aligned} \tag{5}$$

To improve the presentation of the following definitions we introduce a number of helper functions:

We define the function *type* that returns the type of a variable. A mapping
320 $a \mapsto b$ is a dictionary that maps a value a to a key b. The function *to* returns the data type of the value b of a mapping.

The function *of* allows us to recover the function in which a local variable was defined. As a quick reminder, while global variables are declared on contract level, that means on an equal level as the functions, local variables are (in our
325 *understanding*) always declared within the body of a function.

Finally we introduce the function *body* that returns the code from within a function definition. All these helper functions improve the readability of the Solidity semantics that we introduce in the following.

Definition 4 (Helper Functions)**.** *For a variable declaration* $Q \rightarrow T\ \alpha, Q \in \{A_2, O, P, R\} \subset P_{Sol}$ *and a program* $p \in L(G_{Sol})$ *with* $\alpha \in Var_p$ *we define* $type : Var_p \rightarrow \{uint, int, address, bool\}$

$$type : \alpha \mapsto \gamma, Q \Rightarrow T\ \alpha \wedge T \Rightarrow \gamma$$

For a mapping declaration $M \rightarrow mapping\ (T \Rightarrow T)\ \alpha; \in P_{Sol}$ *and a program* $p \in L(G_{Sol})$ *with* $\alpha \in Map_p$ *we define* $to : Map_p \rightarrow \{uint, int, address, bool\}$

$$to : \alpha \mapsto \gamma, M \Rightarrow mapping\ (T_1 \Rightarrow T_2)\ \alpha \wedge T_2 \Rightarrow \gamma$$

For a local variable declaration $Q \rightarrow T\ \alpha, Q \in \{A_2, O, P, R\} \subset P_{Sol}$ *and a program* $p \in L(G_{Sol})$ *with* $\alpha \in LocVar_p$, $G_{Sol} = (N_{Sol}, \Sigma_{Sol}, P_{Sol}, S_{Sol})$ *and* $X = N_{Sol} \cup \Sigma_{Sol}$ *we define* $of : LocVar_p \rightarrow Func_p$

$$of : \alpha \mapsto \zeta, \exists\ \gamma_1, \gamma_2, \gamma_3, \gamma_4 \in X^* :$$
$$S_{Sol} \Rightarrow^* \gamma_1 function\ \zeta(P)\ F_2\ \{O\} \wedge O \Rightarrow^* \gamma_3 T\ \alpha\gamma_4$$

For a function declaration $F \rightarrow function\ \alpha(P)\ F_2\ \{O\} \in P_{Sol}$ *and a program* $p \in L(G_{Sol})$ *with* $\alpha \in Func_p$, $G_{Sol} = (N_{Sol}, \Sigma_{Sol}, P_{Sol}, S_{Sol})$ *we define* $body : Func_p \rightarrow \Sigma^*$

$$body : \alpha \mapsto \gamma, F \Rightarrow function\ \alpha(P)\ F_2\ \{\gamma\}$$

Relying on the introduced context-free grammar of Solidity programs, we
330 are able to derive programs that are not valid Solidity programs. Starting the compiling process would result in errors. So we need to constrain the set of Solidity programs that can be derived from the context-free grammar of Definition **??** to those programs that are *well-defined*. In other words a Solidity program is well defined, if there exists only one constructor, there is a consistency in the
335 declaration of function return variables, variables and mappings are declared before they are referenced and names of variables, mappings and functions are assigned only once. Definition **??** formalizes these requirements.

Definition 5 (Well-defined Solidity Program)**.** *Let* $p \in L(G_{Sol})$ *where* $G_{Sol} = (N_{Sol}, \Sigma_{Sol}, P_{Sol}, S_{Sol})$ *be a solidity program and* $X = N_{Sol} \cup \Sigma_{Sol}$*. We consider*
340 *it well-defined, iff it meets the following conditions:*

- *There is exactly one constructor:* $|Const_p| = 1$
- *The constructor has no return variables:* $\forall\ \alpha \in Const_p, \forall\ \gamma_1, \gamma_2 \in X^*, F_2,$

$O, P \in P_{Sol}$

$$(S_{Sol} \Rightarrow^* \gamma_1 \textbf{\textit{function}}\, \alpha\ (P)\ F_2\ \{O\}\gamma_2) \Rightarrow (F_2 \Rightarrow \varepsilon)$$

- *Declaration of return values of a function either all in the form* $\langle type \rangle$ $\langle name \rangle$ *or anonymous only the* $\langle type \rangle$[1] $\forall\ \alpha \in Func_p, \forall\ \gamma_1, \gamma_2 \in X^*, F_2, P, T, O \in P_{Sol}$*:*

$$\begin{aligned}(S_{Sol} \Rightarrow^* \gamma_1 function\ \alpha(P)\ F_2\ \{O\}\gamma_2\\ \wedge\ F_2 \Rightarrow^*\ returns\ (p_1, \cdots, p_n),\\ p_i \in X^*, 1 \leq i \leq n, n \in \mathbb{N})\\ \Rightarrow\\ (\exists p_i, 1 \leq i \leq n, \alpha \in Var_p : p_i = T\ \alpha\\ \Rightarrow \nexists\ p_j, 1 \leq j \leq n : p_j = T)\\ \vee\\ (\exists p_i, 1 \leq i \leq n : p_i = T\\ \Rightarrow \nexists\ p_j, 1 \leq j \leq n, \alpha \in Var_p : p_j = T\ \alpha)\end{aligned}$$

- *Variables and mappings must be declared before those are used:* $\forall\ \alpha \in Var_p, \forall\ \gamma_1, \gamma_2 \in X^*, T, V, Z \in P_{Sol}:$

$$\begin{aligned}(S_{Sol} \Rightarrow^* \gamma_1 V \gamma_2 \wedge (V \Rightarrow^* \alpha \vee V \Rightarrow^* \alpha[Z]))\\ \Rightarrow (\exists \gamma_3, \gamma_4 \in X^* : S_{Sol} \Rightarrow^* \gamma_3 T\ \alpha\gamma_4 \Rightarrow^* \gamma_1 V \gamma_2)\\ \vee\\ (S_{Sol} \Rightarrow^* \gamma_3 \textbf{\textit{mapping}}(T \Rightarrow T)\ \alpha; \gamma_4 \Rightarrow^* \gamma_1 V \gamma_2)\end{aligned}$$

---

[1] In Solidity return parameters must be defined in the function header: `function` $\langle$name$\rangle$ ($\langle$parameters$\rangle$) `returns` ($\langle$return parameters$\rangle$)$\{\cdots\}$. Thereby it is possible to specify these return parameters only by type, which means to return a comma separated list of values or as named variables. These variables exists in the context of the function and values can be assigned to it.

- *There exists no two different functions with equal names:* $\forall\, \gamma_1, \gamma_2, \gamma_3, \gamma_4 \in X^*, \alpha \in Func_p$:

$$(S_{Sol} \Rightarrow^* \gamma_1 \textbf{\texttt{function}}\, \alpha(P)\ F_2\ \{O\}\gamma_3 \wedge\ S_{Sol} \Rightarrow^* \gamma_2 \textbf{\texttt{function}}\, \alpha(P)\ F_2\ \{O\})\gamma_4) \Rightarrow (\gamma_1 = \gamma_2 \wedge \gamma_3 = \gamma_4)$$

- *There exists no two different variables with equal names:* $\forall\, \gamma_1, \gamma_2, \gamma_3, \gamma_4 \in X^*, \alpha \in Var_p$:

$$(S_{Sol} \Rightarrow^* \gamma_1 T\ \alpha\gamma_3 \wedge\ S_{Sol} \Rightarrow^* \gamma_2 T\ \alpha\gamma_4) \Rightarrow (\gamma_1 = \gamma_2 \wedge \gamma_3 = \gamma_4)$$

- *There exists no two different mappings with equal names:* $\forall\, \gamma_1, \gamma_2, \gamma_3, \gamma_4 \in X^*, \alpha \in Map_p$:

$$(S_{Sol} \Rightarrow^* \gamma_1 \textbf{\texttt{mapping}}\ (T \Rightarrow T)\ \alpha; \gamma_3 \wedge S_{Sol} \Rightarrow^* \gamma_2 \textbf{\texttt{mapping}}\ (T \Rightarrow T)\ \alpha; \gamma_4) \Rightarrow (\gamma_1 = \gamma_2 \wedge \gamma_3 = \gamma_4)$$

- *There exist no internal call of functions:* $\nexists\, \alpha \in Func_p, \nexists\, \gamma_1, \gamma_2 \in X^*$

$$S_{Sol} \Rightarrow^* \gamma_1 \alpha() \gamma_2$$

For the rest of this paper we consider every program to be well-defined.

Based on the syntax of a Solidity program we are now able to define the semantic. A semantic definition on the EVM byte-code level - that is the language 345 interpreted by the Ethereum blockchain and the language Solidity is compiled to - is provided in [18]. Since for applying model checking the size of the considered state space decides about the solubility of the problem, we consider a Solidity program on an higher abstraction level.

At execution time of smart contracts, we need to consider two different 350 types of variables. Those that are persistently stored in the blockchain and volatile variables that are only available at runtime. In the following we consider

variables and mappings that are declared on contract level as persistent values that are stored in the blockchain and local variables that are declared on function levels as volatile variables (cfg. Definition **??**).

Definition 6 (Solidity State Space)**.** *We define the set of persistent values for a program* $p \in L(G_{Sol})$ *where* $GlobVar$ *and* Map *are derived from* p *as described in Definition* **??** *following the grammar definition* G_{Sol}*:*

$$\sigma_B : GlobVar \cup (Map \times \mathbb{N}) \rightarrow \mathbb{Z} \tag{6}$$

and the set of volatile memory variables with $LocVar$ *from Definition* **??** *by*

$$\sigma_M : LocVar \cup \{ret\} \rightarrow \mathbb{Z} \tag{7}$$

355 *Initially every variable has the value* 0*.*

The operational semantic defined in Definition **??** declares how the Solidity program expressions of Definition **??** can manipulate the states of the program state space.

The first two derivation rules of Definition **??** explain the semantic of the if
360 statement. In case the condition B evaluates to true the body of the if statement will be executed, otherwise it will be omitted.

For the concatenation of two program expressions we need to consider two cases: Either the first program translates into another step, then this new step will be executed next, or the first expression *terminates* represented by the $\downarrow$.
365 In this case the second program expression will be executed next.

As for the branching we need to consider two cases for the loop expression with the difference that we check the loop condition again after executing the loop body.

In case of a function return we assign the returned value to the special
370 variable *ret*. While at a function call, we first determine the function argument based on the current program (persistent and volatile) state and then execute the function body.

The next section covers the handling of assignments. Depending whether the considered variable is a local or a global variable we assign the value to the

375 corresponding state. Although we only provided the regular assignment it is quite obvious how to define the operational semantics of the $+=, -=, *=, /=$, increment and decrement operations.

Definition 7 (Solidity Operational Semantic)**.** *Let σ_M, σ_B the volatile and persistent memory as in Definition* **??**. *We use $\downarrow$ to imply that there is no succeeding*
380 *operation left and body$(\cdots)$ as in Definition* **??**. *The operational semantic for controlling the program flow is defined as follows:*

$$\frac{\langle B, \sigma_B, \sigma_M \rangle \to \mathit{True}}{\langle if(B)\{O\}, \sigma_B, \sigma_M \rangle \to \langle O, \sigma_B, \sigma_M \rangle} \qquad \frac{\langle B, \sigma_B, \sigma_M \rangle \to \mathit{True}}{\langle \mathit{while}(B)\{O\}, \sigma_B, \sigma_M \rangle \to \langle O; \mathit{while}(B)\{O\}, \sigma_B, \sigma_M \rangle}$$

$$\frac{\langle B, \sigma_B, \sigma_M \rangle \to \mathit{False}}{\langle if(B)\{O\}, \sigma_B, \sigma_M \rangle \to \langle \downarrow, \sigma_B, \sigma_M \rangle} \qquad \frac{\langle B, \sigma_B, \sigma_M \rangle \to \mathit{False}}{\langle \mathit{while}(B)\{O\}, \sigma_B, \sigma_M \rangle \to \langle \downarrow, \sigma_B, \sigma_M \rangle}$$

$$\frac{\langle O, \sigma_B, \sigma_M \rangle \to \langle O', \sigma_B, \sigma_M \rangle, O' \neq \downarrow}{\langle O; O, \sigma_B, \sigma_M \rangle \to \langle O'; O, \sigma_B, \sigma_M \rangle} \qquad \frac{\langle B, \sigma_B, \sigma_M \rangle \to b}{\langle \mathit{return}\ B, \sigma_B, \sigma_M \rangle \to \langle \downarrow, \sigma_B, \sigma_M[\mathit{ret} \mapsto b] \rangle}$$

$$\frac{\langle O, \sigma_B, \sigma_M \rangle \to \langle O', \sigma_B, \sigma_M \rangle, O' = \downarrow}{\langle O; O, \sigma_B, \sigma_M \rangle \to \langle O, \sigma_B, \sigma_M \rangle} \qquad \frac{\alpha \in \mathit{Func}_p \wedge O = \mathit{body}(\alpha) \wedge \sigma'_M = \sigma_M[q_1 \mapsto p_1, \cdots, q_n \mapsto p_n]}{\langle \alpha(p_1, \cdots, p_n), \sigma_B, \sigma_M \rangle \to \langle O, \sigma_B, \sigma'_M \rangle}$$

The operational semantic for assignments is defined as follows

$$\frac{\langle A_2, \sigma_B, \sigma_M \rangle \to \alpha \qquad \langle B, \sigma_B, \sigma_M \rangle \to b \qquad \alpha \in \mathit{GlobVar}}{\langle A_2 = B, \sigma_B, \sigma_M \rangle \to \langle \downarrow, \sigma_B[\alpha \mapsto b], \sigma_M \rangle}$$

385

$$\frac{\langle A_2, \sigma_B, \sigma_M \rangle \to \alpha \qquad \langle B, \sigma_B, \sigma_M \rangle \to b \qquad \alpha \in \mathit{LocVar}}{\langle A_2 = B, \sigma_B, \sigma_M \rangle \to \langle \downarrow, \sigma_B, \sigma_M[\alpha \mapsto b] \rangle}$$

The operational semantic for algebraic operations is given by

$$\frac{\langle \alpha, \sigma_B, \sigma_M \rangle \to \sigma_B[\alpha] \qquad \alpha \in \mathit{GlobVar}_p}{\langle \alpha, \sigma_B, \sigma_M \rangle \to \langle \sigma_B[\alpha], \sigma_B, \sigma_M \rangle}$$

$$\frac{\langle \alpha[Z], \sigma_B, \sigma_M \rangle \to \sigma_B[\alpha_z] \qquad \langle Z, \sigma_B, \sigma_M \rangle \to z \qquad \alpha \in \mathit{Map}_p}{\langle \alpha[Z], \sigma_B, \sigma_M \rangle \to \langle \sigma_B[\alpha_z], \sigma_B, \sigma_M \rangle}$$

$$\frac{\langle \alpha, \sigma_B, \sigma_M \rangle \to \sigma_M[\alpha] \qquad \alpha \in \mathit{LocVar}_p}{\langle \alpha, \sigma_B, \sigma_M \rangle \to \langle \sigma_M[\alpha], \sigma_B, \sigma_M \rangle}$$

$$\frac{\langle Z_1, \sigma_B, \sigma_M \rangle \to z_1 \qquad \langle Z_2, \sigma_B, \sigma_M \rangle \to z_2}{\langle Z_1 + Z_2, \sigma_B, \sigma_M \rangle \to \langle z_1 + z_2, \sigma_B, \sigma_M \rangle}$$

We use $\to^*$ with $\langle A, \sigma_B, \sigma_M \rangle \to^* \langle B, \sigma'_B, \sigma'_M \rangle = \langle A, \sigma_B, \sigma_M \rangle \to \langle A_1, \sigma^1_B, \sigma^1_M \rangle$

$\rightarrow \cdots \rightarrow \langle B, \sigma'_B, \sigma'_M \rangle$ *to indicate a transition that comprises multiple semantic*
390 *steps.*

This concludes the definition of the operational semantic of a Solidity program. Next we introduce the semantics that comes from executing Solidity programs in an Ethereum blockchain environment. Note that every interaction with a smart contract in the Ethereum blockchain results in the creation of a
395 transaction. That means manipulating a value in the persistent memory of a smart contract consequently creates a transaction that changes this value. Due to the nature of blockchain there can be always a partial order established on the transactions. So the state of smart contract depends on the order in which transactions were issued, but in particular it will never be the case that two
400 transactions and by that two instances of the same smart contract are executed in parallel.

A transaction is the basic entity of a blockchain and always originates from an address. We start by defining the address space in Definition 1.

Definition 8 (Address Space)**.** *We introduce* $Addr = \mathbb{Z}$ *to denote the set of ad-*
405 *dresses and introduce the address balance by* $\sigma_B[a]$ *for* σ_B *the persistent program state and* $a \in Addr$*.*

A transaction then triggers the execution of a function of the smart contract. Thereby it is possible to provide arguments to the transaction and in particular the instantiation transaction calls the constructor of a smart contract and thus
410 initializes it on the blockchain. Here we omitted the fact that in Ethereum the smart contract code itself is also stored on the blockchain.

Definition 9 (Transaction)**.** *Let* $p \in L(G_{Sol})$ *a well-defined solidity program and* $\alpha \in Func_p$ *a function,* $x_1, \cdots, x_n \in \mathbb{Z}$ *parameters,* $a \in Addr$ *an address and* σ_B, σ_M *the program state. A transaction* $tx_{a,\alpha}^{x_1,\cdots,x_n}$ *is defined by:*

$$tx_{a,\alpha}^{x_1,\cdots,x_n} : \langle \alpha(x_1, \cdots, x_n), \sigma_B, \sigma_M, a \rangle \rightarrow \langle \downarrow, \sigma'_B, \sigma'_M, a \rangle \quad (8)$$

We denote the set of all transactions with TX*.*

For an address $tx_{a,\alpha}^{x_1,\cdots,x_n}$ *the address a represents the sender of the transaction. We omit the address a and write* $tx_{\alpha}^{x_1,\cdots,x_n}$ *in case the origin address of*
415 *the transaction is not important.*

A transaction $tx_{a,\alpha}^{x_1,\cdots,x_n}$ *with* $\alpha \in Const_p$ *is called instantiation transaction and is also denoted by* $tx_{a,origin}^{x_1,\cdots,x_n}$*. We write* tx_α *in case that the function* $\alpha \in Func_p$ *has no parameters.*

As mentioned before the transactions that were added to the blockchain
420 establish a partial order. The transactions that were added to the blockchain and target the same smart contract establish a *total* order. We introduce the *step* function that allows the consecutive execution of two transactions and extends this function to execute an arbitrary number of transactions. We finally define the transaction history that represents a totally ordered set of consecutively
425 executed transactions.

In the context of the real Ethereum blockchain we need to recapitulate that there exists not only one smart contract in the blockchain but millions of different contracts and a multitude of transactions are issued every minute that call these smart contracts. Although those transactions can be totally ordered, only
430 those transactions that target the corresponding smart contract are important to compute the contract state. So in our case the transaction history represents all the transactions that lead to a certain smart contract state.

Definition 10 (Transaction Step and Transaction History)**.** *Let TX be the set of transactions,* $\alpha, \alpha' \in Func_p$*,* $p_1 = x_1, \cdots, x_m \in \mathbb{Z}$*,* $p_2 = y_1, \cdots, y_n \in \mathbb{Z}$ *parameters for* $m, n \in \mathbb{N}$*,* $a_1, a_2 \in Addr$ *addresses and* σ_B, σ_M *the persistent and volatile memory state. The* $step_{a_1,\alpha(p_1),a_2,\alpha'(p_2)} : TX \to TX$ *function is then defined by:*

$$\begin{aligned} & step_{a_1,\alpha(p_1),a_2,\alpha'(p_2)} : \\ & \langle \alpha(p_1), \sigma_B, \sigma_M, a_1 \rangle \to \langle \downarrow, \sigma_B^{\alpha}, \sigma_M', a_1 \rangle \\ & \mapsto \langle \alpha`(p_2), \sigma_B^{\alpha}, \sigma_M, a_2 \rangle \to \langle \downarrow, \sigma_B^{\alpha'}, \sigma_M``, a_2 \rangle \end{aligned} \tag{9}$$

For set of parameters $p_i = x_1^i, \cdots, x_{n_i}^i \in \mathbb{Z}, m, n_i \in \mathbb{N}, i \in \{1, \cdots, m\},,$

addresses $a_1, \cdots, a_m \in Addr$ *and functions* $\alpha_1, \cdots, \alpha_m \in Func_p$ *we define the* 435 *function call sequence* $\Omega = a_1, \alpha_1(p_1), \cdots, a_m, \alpha_m(p_m)$ *and the set of all sequences with Seq. We extend step to step′:*

$$step'_{a_1,\alpha_1(p_1),a_2,\alpha_2(p_2),\cdots,a_m,\alpha_m(p_m)} = \tag{10}$$

$$step_{a_{m-1},\alpha_{m-1}(p_{m-1}),a_m,\alpha_m(p_m)}(\cdots step_{a_1,\alpha_1(p_1),a_2,\alpha_2(p_2)}(tx)) \tag{11}$$

$\langle tx^{p_1}_{a_1,\alpha_1}, \cdots, tx^{p_m}_{p_m,\alpha_m}\rangle$ *is called a history iff* $\forall\ i, j \in \{1, \cdots, m\} : i < j \Leftrightarrow \exists\, \Omega \in Seq:\ tx^{p_j}_{a_j,\alpha_j} = step'_{\Omega}(tx^{p_i}_{a_i,\alpha_i})$

For $tx^{p_1}_{a_1,\alpha_1} = tx^{p_1}_{a_1,origin}$ *the address* $a_1 \in Addr$ *is denoted with tx.origin.* 440 *By executing the code* α *of a transaction* $tx^p_{a,\alpha}$ *the address a is denoted with tx.sender.*

The introduced syntax and semantics of the considered restricted Solidity version represents our understanding of the Solidity semantics. It is possible that this differs in some cases from the actual semantic. We work hard on 445 aligning our model with the reality. The target is to translate the semantics of a Solidity program in equivalent semantics of a PROMELA model to enable its formal verification. Therefore we need to introduce the PROMELA syntax next.

4.2. Syntax Definition of PROMELA

450 In general we follow the propositions of defining the syntax and semantics of PROMELA from [3]. Thereby we omitted expressions for channel communication, since there will be no parallel executed smart contracts. Recall Section 4.1, there exists always a total order between transactions and a smart contract will never be simultaneously executed by two transactions. However, changing 455 the order of the transactions, or removing certain transactions means to alter the resulting contract state.

Definition 11 (PROMELA Syntax)**.** *Let* $G_{Prom} = (N_{Prom}, \Sigma_{Prom}, P_{Prom}, S_{Prom})$ *be the context-free grammar describing the syntax of a PROMELA program as follows:*

460 $S_{Prom} \rightarrow P\ S \mid I\ S \mid A;\ S$

$I \rightarrow$ *init* { C }
$P \rightarrow$ *proctype* α() { C }
$A \rightarrow T\ \alpha = Z$

465

$C \rightarrow$ *run* α() | $C;C$ | A
| *atomic* { Q } | *d_step* { C }
| *skip* | *break*
| *if* G *fi* | *do* G *od*
470 | *assert*(B)

$Q \rightarrow A;\ Q \mid \varepsilon$

$G \rightarrow\ ::B \rightarrow C\ G \mid\ ::\ else \rightarrow C \mid \varepsilon$

475

$B \rightarrow Z < Z \mid Z \leq Z \mid Z > Z \mid Z \geq Z$
| $B = B$ | $B \neq B$
| B && B | B || B
| *true* | *false*
480 $Z \rightarrow z \mid Z + Z \mid Z - Z \mid Z * Z \mid Z\ /\ Z$

$T \rightarrow$ *byte* | *int*

with $z \in \mathbb{Z}$ *and* α *an arbitrary label of the form* $x_1 x_2 \cdots x_n$, $x_1 \in \{a-zA-Z\}$ *and* $x_i \in \{a-zA-Z\} \cup \mathbb{N}$ *for* $n \geq i > 1$ *and* $n \in \mathbb{N}$.

485 ### *4.3. Validation in SPIN*

Originally SPIN was designed to verify the correctness of communication protocols. Communication between different processes is based on communication channels, where one process writes a value into the channel at a certain position in code, while other processes listen on the channel. Processes are defined as
490 *proctypes* and although it is possible to spawn new processes dynamically at execution time there exists an *init* section to specify some initial instructions and to spawn one or several processes. It is also possible to annotate a proctype with an *active* flag to mark it for automatic start-up at the beginning.

SPIN models the execution of multiple processes in interleaving semantics,
495 that means single expressions are executed atomically, but every possible permutation of process expressions of interleaving processes is considered.

To tell SPIN that a certain code section is considered to be executed atomically it provides the keywords *atomic* and *d_step*. Thereby we must carefully differentiate between those commands. While the atomic expression allows to execute a code section indivisibly, as long as there is no blocking expression, such as a blocking listening on a channel, within the atomic section, d_step acts similarly, but *goto* jumps that lead into or out of the section are prohibited. Additionally the whole section is handled deterministically. Non-determinism is resolved by always selecting the first guard that evaluates to true and channel operations can not be used, since if their execution leads to an error it blocks the execution of the section.

When designing communication protocols a certain kind of non-determinism is necessary, since messages are received by coincidence. SPIN allows the modeling of non-determinism by introducing guarded expressions for branches and loops. A guarded statement consists of a boolean expression that is the guard and a body, which contains program expressions. Program expressions are executed, if the guard evaluates to true. An if statement and a loop comprise a set of these guarded statements and it is absolutely possible that multiple guards evaluate to true at the same time. In this case SPIN considers every possible program flow. While in the if statement the guards are only checked once, in the loop expression those are checked repeatedly. To leave a loop a *break* statement is necessary.

This explains how SPIN creates the state space of a PROMELA model. To recapitulate: A state can be considered as a function that maps to every variable that exists in the model a value of the corresponding variable data type domain. Obviously the state space can become very large even for small programs, if for instance the whole domain of several integer variables must be considered. A complete state space holds every possible program flow and every situation of a program. Interesting for the model checking is now to identify states or paths in the state space that either must be reached, or even repeatedly reached (liveness criteria) or must never be reached, since they represent an error state.

SPIN basically provides three mechanisms to specify the correctness of pro-

grams:

- Assertions
- Deadlock detection
- Checking Liveness properties with LTL

Assertion are boolean expressions prefixed by the keyword *assert*. It can be used to test properties of variables, as e.g., whether a variable is within certain limits. SPIN then checks every possible path and provides a counterexample in case it finds a path that allows to exceed these bounds.

SPIN is able to detect deadlocks, that means situations in which a program can not make a step to another program state anymore. An example is when two programs simultaneously wait for an answer of the other process. So process A waits for a message of process B, but process B can not send this message, since it is waiting for a message of process A. There is no way that either process can move.

One advantage of applying model checking is the possibility to check for liveness criteria. Liveness criteria express states that must be visited during regular program execution. So for instance a web server should always listen for web clients that request a certain web content. If the server does not longer listens for clients, it does not longer fulfill its purpose and thus crashed. Liveness properties can be used to find paths that lead to situations in which a program does not longer works as expected. Liveness properties in SPIN are expressed in linear temporal logic (LTL).

For all these methods SPIN provides counterexamples in case problematic states are identified. Those counterexamples can be used to fix the model and understand the actual problem of the current implementation.

SPIN provides an excellent base for all kinds of different verification methods, however in this first version we concentrate on the evaluation of assertions.

5. The Approach

We motivated the necessity of formal verification for creating smart contracts in Section 1 and Section 2. A particularity of smart contracts is the problem that many different parties from very different contexts must be able to understand the contracts. In reality only people who can read and understand program code are able to understand what a smart contract really does. Our aim is to provide a tool that allows to increase the safety and quality of smart contracts and that is easily applicable without extensive previous training.

SPIN is a popular model checker with a good reputation earned in a long history of application in industry and research. Therefore we decided to start by using SPIN as a model checker. The whole process is than given by: First the Solidity program must be translated in an equivalent PROMELA model. Thereby we must consider the operational semantics of Solidity, as well as the semantics induced by running Solidity programs on the Ethereum blockchain. Second there must be a possibility to provide assertions that are evaluated and placed into the PROMELA model. As a final step the generated model must be executed and in case of a detected error a counterexample must be provided.

In the following we start by extending the Solidity syntax to handle SPIN assertion. We than formally define the problem that we are going to solve and introduce the translation function that converts a Solidity program into a SPIN model.

5.1. Extending the Solidity Syntax

One aspect in the extension of the Solidity syntax is that we do not want to break the Solidity compilation process. Therefore a PROMELA assertion is considered as a comment by the Solidity compiler. On the semantic level we introduce a *fail state*, which is a state that must not be reachable through any program path.

Definition 12 (Extended Solidity Grammar)**.** *We extend the context-free grammar G_{Sol} from Definition* **??** *by the following rule to G^*_{Sol}:*

```
O → // assert(B)
```

We define the operational semantics of the assertion statement during a transaction as follows:

$$\frac{\langle B, \sigma_B, \sigma_M \rangle \to \mathit{True}}{\langle //\mathit{assert}(B), \sigma_B, \sigma_M \rangle \to \langle \downarrow, \sigma_B, \sigma_M \rangle} \tag{12}$$

$$\frac{\langle B, \sigma_B, \sigma_M \rangle \to \mathit{False}}{\langle //\mathit{assert}(B), \sigma_B, \sigma_M \rangle \to \langle \bot, \sigma_B, \sigma_M \rangle} \tag{13}$$

585 *where $\bot$ marks a fail state.*

We further introduce the failed transaction which is simply a transaction that leads to a fail state.

Definition 13 (Failed Transaction)**.** *Let $\alpha \in \mathit{Func}$ and $p = x_1, \cdots, x_m \in \mathbb{Z}, m \in \mathbb{N}$ parameters and $\sigma_B, \sigma'_B, \sigma_M, \sigma'_M$ program states. We write $tx^p_\alpha \vdash \bot$ for*
590 *$\langle \alpha(p), \sigma_B, \sigma_M \rangle \to^* \langle \bot, \sigma'_B, \sigma'_M \rangle$*

5.2. The Problem Statement

Based on these definitions we are now able to formally define the problem that we are going to solve:

For an arbitrary smart contract, can we find a sequence of transactions that
595 *leads this smart contract into a fail state?*

Definition 14 (The Problem)**.** *Let $n > 0, n \in \mathbb{N}$ and $\{tx_{origin}, tx_1, \cdots, tx_n\}$ be a set of transactions. $\exists\, \Pi : \{1, \cdots, n\} \to \{1, \cdots, n\}$ with $\langle tx_{origin}, tx_{\Pi(1)}, \cdots, tx_{\Pi(n)} \rangle$ a history, such that $\exists\, i \in \{1, \cdots, n\} : tx_i \vdash \bot$*

5.3. Translating Solidity Code into a SPIN Model

600 To translate a Solidity program into an equivalent PROMELA model we define a function that takes a well-defined Solidity program and the non-terminal symbols of the context-free grammar that describes the syntactic structure of

a Solidity program as arguments and outputs a PROMELA model. Parts of the Solidity program are just consumed, while others are directly or indirectly translated into corresponding PROMELA statements. To improve readability of the translation function (Equation 1), we split the definition into different categories.

$$tr : (L(G_{\mathrm{Sol}}) \times N_{\mathrm{Sol}}) \rightarrow (L(G_{\mathrm{Prom}}) \cup \{\varepsilon\}) \tag{14}$$

$\gamma, \gamma_1, \gamma_2 \in L(G_{\mathrm{Sol}})$ and α a label following Definition **??**.

5.3.1. Program Structure

For the definition of the Solidity version at the beginning of a Solidity program and a contract, there exists no equivalent expressions in PROMELA. Therefore we just consume them. Since we handle the declaration of variables in an additional step we just remove them from the Solidity code and events are not yet supported and therefore omitted. Since the translation function is based on the context-free grammar we need to handle specific concatenation of code differently, what is done in the following four mappings.

- $tr : (\texttt{program solidity}\ z_1.z_2.z_3; \gamma, S_{\mathrm{Sol}}) \mapsto tr(\gamma, C)$
- $tr : (\texttt{contract}\ \alpha\{\gamma\}, C) \mapsto tr(\gamma, G)$
- $tr : (\texttt{event}\ \alpha(\gamma);, E) \mapsto \varepsilon$
- $tr : (\texttt{mapping}\ (T \Rightarrow T)\ \alpha;, M) \mapsto \varepsilon$
- $tr : (\gamma_1\gamma_2, G) \mapsto tr(\gamma_1, M)tr(\gamma_2, G)$
  for $G \Rightarrow MG \wedge M \Rightarrow^* \gamma_1 \wedge G \Rightarrow^* \gamma_2$
- $tr : (\gamma_1\gamma_2, G) \mapsto tr(\gamma_1, E)tr(\gamma_2, G)$
  for $G \Rightarrow EG \wedge E \Rightarrow^* \gamma_1 \wedge G \Rightarrow^* \gamma_2$
- $tr : (\gamma_1\gamma_2, G) \mapsto tr(\gamma_1, F)tr(\gamma_2, G)$
  for $G \Rightarrow FG \wedge F \Rightarrow^* \gamma_1 \wedge G \Rightarrow^* \gamma_2$

- $tr : (\gamma_1\gamma_2, G) \mapsto tr(\gamma_1, A)tr(\gamma_2, G)$
  for $G \Rightarrow AG \wedge A \Rightarrow^* \gamma_1 \wedge G \Rightarrow^* \gamma_2$

As described in Section 4 and explained in the problem statement in Definition 3 is the aim to find sequences of transactions that lead to an error state in the smart contract. Every transaction represents the execution of a smart contract function, where the constructor is executed at smart contract instantiation on the blockchain. To test a smart contract means to evaluate every possible combination of function calls with every possible combination of parameters. Therefore a Solidity function is translated into a process and, since in the blockchain the transaction order is total and a smart contract is never executed by two transactions simultaneously, the function body is considered to be an atomic sequence. At the end of the translated function code we add a label that is used to jump to the end of the execution in case of a *return* in the Solidity code. That prevents that there is code executed after an occurring return statement.

- $tr : (\texttt{function}\ \alpha(\gamma_1)\ F_2\ \{\gamma_2\}, F) \mapsto$

  $\texttt{proctype}\ \alpha\ \{$

  $\quad tr(\gamma_1, P)$

  $\quad \texttt{d_step}\ \{$

  $\quad\quad tr(\gamma_2, O)$

  $\quad \}$

  $\quad \texttt{return_label_}\alpha :$

  $\}$

The constructor of a Solidity program is translated into an init section of the PROMELA model. This section is initially executed once and corresponds to the one time execution of the constructor body. Subsequently the functions of the smart contracts are non-deterministically spawned. This allows SPIN to check every possible permutation of function executions and since it is done in a loop it also allows to check the repeated execution of smart contract functions.

We use a counter *pc_count* to control the number of spawned processes. This is in particular necessary, since SPIN only supports a finite number of parallely spawned processes.

- tr : (`function` $\alpha(\gamma_1)$ F_2 $\{\gamma_2\}, F) \mapsto$

```
660  init {
         d_step {
             tr(γ1, P)
             tr(γ2, O)
         }
665      #define MIN_PC_COUNT s_min
         #define MAX_PC_COUNT s_max
         byte pc_count = MIN_PC_COUNT;
         do
             :: pc_count < MAX_PC_COUNT → pc_count + +; run α1()
670          ⋮
             :: pc_count < MAX_PC_COUNT → pc_count + +; run αn()
             :: pc_count >= MAX_PC_COUNT → break
         od
     }
```

675 for $\alpha \in \text{Const}_p$, $F_2 \Rightarrow^* \varepsilon$, $s_{min}, s_{max} \in \mathbb{N}$ and $\{\alpha_1, \cdots, \alpha_n\} = \text{Func}_p \backslash \text{Const}_p$

5.3.2. Program Flow

To translate the bodies of Solidity functions we need to recapitulate the definition of the operational semantics provided in Definition **??**. As mentioned in Section 4.2 branching and loop statements in PROMELA are handled non-
680 deterministically in case that there are several guards evaluating to true. By translating a Solidity if statement into a PROMELA guarded if statement we consider the boolean expression of the if statement as one guard leading to the corresponding statement body. We need to add an *else* guard that is true, if there is no other guard that is true. This guard allows us to jump over the
685 otherwise blocking branching statement.

- $tr : (\texttt{if}(\gamma_1)\{\gamma_2\}, O) \mapsto$

  `if`

  $:: tr(\gamma_1, B) \to tr(\gamma_2, O)$

  $:: \texttt{else} \to \texttt{skip}$

690 `fi`

The same holds for the translation of the looping statement. There exists one guard that tests the loop condition and accordingly executes the loop body, but another *else* guard allows us to jump out of the loop in case the loop condition is not longer satisfied.

695
- $tr : (\texttt{while}(\gamma_1)\{\gamma_2\}, O) \mapsto$

  `do`

  $:: tr(\gamma_1, B) \to tr(\gamma_2, O)$

  $:: \texttt{else} \to \texttt{break}$

  `od`

700 As before we omit variable declarations, since those are handled in an additional step. Finally we translate the function return statement into an assignment of the returned value to a special variable that is created for every value returning function. This variable can be used to test the return value of functions with assertions. Additionally we introduce a jump to the end of the
705 process the prevent the execution of any code after reaching a return statement.

- $tr : (\gamma_1; \gamma_2, O) \mapsto tr(\gamma_1, O); tr(\gamma_2, O)$
  for $O \Rightarrow O; O \Rightarrow^* \gamma_1; \gamma_2$

- $tr : (T\ \alpha, O) \mapsto \varepsilon$

- $tr : (\gamma, O) \mapsto tr(\gamma, A)$ with $O \Rightarrow A \Rightarrow^* \gamma$

710
- $tr : (\texttt{return}\ \gamma, O) \mapsto \text{ret}_\alpha = tr(\gamma, B); \text{goto return_label_}\alpha$
  for $S_{\text{Sol}} \Rightarrow^* \texttt{function}\ \alpha(P)\ F_2\ \{O\}$ and $O \Rightarrow^* \texttt{return}\ \gamma$

5.3.3. Memory Operations

Assignments also those that include algebraic operations, as for instance, increment and decrement operations are handled equally in Solidity and PROMELA
715 and can be translated accordingly.

- $tr : (\gamma_1 \delta \gamma_2, A) \mapsto tr(\gamma_1, A_2)\delta tr(\gamma_2, B)$, for $\delta \in \{=, + =, - =, * =, / =\}$ and $A \Rightarrow A_2 \delta B \Rightarrow^* \gamma_1 \delta \gamma_2$
- $tr : (\gamma\delta, A) \mapsto tr(\gamma, A_2)\delta$, for $A \Rightarrow A_2\delta \Rightarrow^* \gamma\delta$ and $\delta \in \{++, --\}$
- $tr : (\delta\gamma, A) \mapsto \delta tr(\gamma, A_2)$, for $A \Rightarrow \delta A_2 \Rightarrow^* \delta\gamma$ and $\delta \in \{++, --\}$

720 An expression that declares a variable and assigns directly a value to it is replaced by the plane assignment, since declarations are handled in an additional step.

- $tr : (T\alpha, A_2) \mapsto \alpha$
- $tr : (\gamma, A_2) \mapsto tr(\gamma, V)$ with $A_2 \Rightarrow V \Rightarrow^* \gamma$

725 Mappings are special dictionary type Solidity data structures that assign values to keys. In Solidity those can theoretically grow infinitely large, however, we deal with mappings by introducing a roll-out parameter that defines how many different key value pairs are supported. As a consequence the translation function assigns the value to the variable that represents the corresponding key
730 of the mapping in the PROMELA model.

- $tr : (\alpha, V) \mapsto \alpha$
- $tr : (\alpha[\gamma], V) \mapsto \alpha_{tr(\gamma, Z)}$

This first version does not support all the data types of Solidity. The supported data types are translated into the corresponding data types of the
735 PROMELA model. Note that to reduce the state space we aim to preferably use data types with small domains.

- $tr : (\gamma, T) \mapsto \texttt{byte}$ for $\gamma \in \{\texttt{address}, \texttt{byte}\}$

- $tr : (\gamma, T) \mapsto$ `int` for $\gamma \in \{\texttt{uint}, \texttt{int}\}$

The handling of function parameters is another important part. As mentioned before, we need to consider every possible combination of function calls and every possible combination of function parameter valuations. So for every function parameter we introduce a corresponding variable that we increment non-deterministically in a loop. In this way every parameter valuation is considered. Depending on the considered data type we use different lower and upper bounds.

- $tr : (\gamma_1\ \alpha_1, \cdots, \gamma_n\ \alpha_n, P) \mapsto$

  $tr(\gamma_1, T)\ \alpha_1 = \min_{\gamma_1};$

  `do`

      $::\ \alpha_1 < \max_{\gamma_1} \rightarrow \alpha_1 = \alpha_1 + 1;$

      $::$ `else` $\rightarrow$ `break`;

  `od`

      $\vdots$

  $tr(\gamma_n, T)\ \alpha_n = \min_{\gamma_n};$

  `do`

      $::\ \alpha_n < \max_{\gamma_n} \rightarrow \alpha_n = \alpha_n + 1;$

      $::$ `else` $\rightarrow$ `break`;

  `od`

  for $\min_{\gamma_i}, \max_{\gamma_i}$ the lower and upper bound of the data type $\gamma_i = \text{type}(\alpha_i)$ of variable $\alpha_i \in \text{Var}_p$.

5.3.4. Boolean and Algebraic Operations

Boolean and algebraic operations are handle equally in Solidity and PROMELA.

- $tr : (\texttt{true}, B) \mapsto$ `true`

- $tr : (\texttt{false}, B) \mapsto$ `false`

- $tr : (\gamma, B) \mapsto tr(\gamma, Z),\ B \Rightarrow Z \Rightarrow^* \gamma$

- $tr : (\gamma_1 \delta \gamma_2, B) \mapsto tr(\gamma_1, B) \delta tr(\gamma_2, B)$ for $B \Rightarrow B \delta B \Rightarrow^* \gamma_1 \delta \gamma_2$ and $\delta \in \{\&\&, ||, ==, \neq, <, >, \leq, \geq\}$

- $tr : (\gamma, Z) \mapsto tr(\gamma, V)$, for $Z \Rightarrow V \Rightarrow^* \gamma$

- $tr : (z, Z) \mapsto z$, for $z \in \mathbb{Z}$

- $tr : (\gamma_1 \delta \gamma_2, Z) \mapsto tr(\gamma_1, Z) \delta tr(\gamma_2, Z)$, for $\delta \in \{+, -, *, /, \%\}$ and $Z \Rightarrow Z \delta Z \Rightarrow^* \gamma_1 \delta \gamma_2$

5.3.5. Address Translation

Solidity provides a number of special variables that are assigned by the runtime environment. As introduced in Section 4 every transaction originates from an address. Thereby *tx.origin* refers to the address that was used to initially instantiating the smart contract in the blockchain, while *tx.sender* refers the address of the current sender of the transaction that must not be necessarily the same person that created the smart contract.

We reserve address 0 for the smart contract creator and assign it accordingly. The sender of a transaction can differ but is always represented by a natural number.

- $tr : (tx.origin, V) \mapsto 0$ We consider the smart contract creator address as 0.

- $tr : (tx.sender, V) \mapsto a$ For $\mathrm{tx}^{p}_{a,\alpha}$ where *tx.sender* is executed in α.

This concludes the translation function. To derive a PROMELA model from a Solidity program we concatenate the necessary variable declarations with the output value of the translation function.

In PROMELA there exists no local variables. Every variable is globally accessible. Therefore we create for every global Solidity variable a corresponding PROMELA variable, but for the local variables we create variables that are prefixed with the unique function name.

For mappings we create a number of variables, where every variable corresponds to a key in the mapping. A roll-out parameter determines the number of created variables.

Since Solidity functions can return values we create for every function that
795 is not the constructor one return variable.

Definition 15 (Model creation)**.** *Let* $p \in L(G_{Sol})$ *be a program with* $G_{Sol} = (N_{Sol}, \Sigma_{Sol}, P_{Sol}, S_{Sol})$ *and* $type(\cdots)$, $to(\cdots)$, $of(\cdots)$ *be functions from Definition* **??** *and* $n_{unroll} \in \mathbb{N}$ *a predefined constant that determines the number of values a mapping can maximally contain.*

We define the following sets of PROMELA variable declarations:

$$D_{Glob} = \{\gamma\ \alpha = 0 \mid \alpha \in GlobVar_p, \gamma = tr(type(\alpha), T)\}$$

$$D_{Loc} = \{\gamma\ f_\alpha = 0 \mid \alpha \in LocVar_p, \gamma = tr(type(\alpha), T), f = of(\alpha)\}$$

$$D_{Map} = \{\gamma\ \alpha_i = 0 \mid \alpha \in Map_p, i \in \{1, \cdots, n_{unroll}\}, \gamma = tr(to(\alpha), T)\}$$

$$D_{Ret} = \{\gamma\ ret_\alpha = 0 \mid \alpha \in Func_p \backslash Const_p\}$$

$$D = D_{Glob} \cup D_{Loc} \cup D_{Map} \cup D_{Ret}$$

800 *We create the PROMELA model by concatenating* D *with* $tr(p, S_{Sol})$.

6. The Implementation

We use Python 3 for the implementation of our tool chain, since it is a platform independent programming language that does not require much boilerplate code. It has a clear structure and there exists a great number of libraries that
805 can be included to extend functionality and unitize the implementation.

In general the program structure can be divided into three parts (Figure **??**). The first part contains the Solidity code parser. It reads the Solidity code from a file, parses it and generates an abstract syntax tree. For creating the

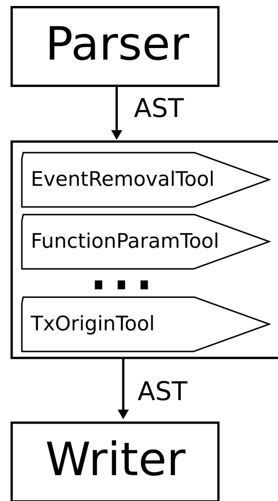

Figure 2: Schematic Overview of the Program Architecture

parser we used the *Toy Parser Generator* [2] that allows the specification of the
810 grammar within the python code and provides a convenient API for parsing. This abstract syntax tree is processed in the second part of the program, which is referred to as the *tool chain.* The tool chain contains a number of tools that are applied on the abstract syntax tree. A tool always requires an abstract syntax tree as input, manipulates it according to its purpose and returns the modified
815 abstract syntax tree. Then it is passed on to the next tool. Using the pattern of the tool chain it is very easy to encapsulate different functionalities and to activate or deactivate them if required. This is invaluable during debugging.

In the following we present an overview of the implemented tools:

- *EventRemovalTool*: As mentioned in Section 5 events are currently not
820 supported and will be removed from the abstract syntax tree.
- *FunctionParameterTool*: This tool handles function parameters. That means it creates the non-deterministic loops to search the complete state space of function parameter value combinations.

[2]http://cdsoft.fr/tpg/

- *FunctionReturnTool*: It translates the return statement into an assignment to the special return variables.
- *IfToGuardedIfTool*: It converts regular Solidity branching statements into the PROMELA guarded branching notation.
- *MappingTool*: It translates the declaration and usage of mappings to the corresponding roll-out mapping variables.
- *MsgSenderTool*: It translates the special Solidity variable *msg.sender* into an address.
- *TxOriginTool*: It replaces every use of the special Solidity variable *tx.origin* by 0.

The tool chain applies all the registered tools on the abstract syntax tree created by the parser and then translates it into PROMELA code. That is the third part. The fourth and final part is an programmatic interface that handles the execution of the spin model checker.

The current implementation consists of 24 python files that contain 4,338 lines of code of which 1,009 lines are comments.

7. The Evaluation

We demonstrate the implementation of the introduced methodology by introducing a rather artificial example of a smart contract [3]. The smart contract (cf. Listing 1) implements a coin that can be send from one account to another depending on the available account balance. The function *sendCoin* (line 14) takes as arguments a receiving address and an amount of tokens that must be transferred. It returns a boolean value that indicates whether the account has the necessary balance.

[3] This is a slightly modified of a popular contract used in tutorials.

A second function *getBalance* (line 24) allows querying the balance of a given account. The corresponding function parameter is the account address.

The constructor (line 8) initializes the balance of account 0 with 1000 coins.

We use three SPIN assertions to test and ensure the correct behavior of the coin contract. In the constructor at line 10 we check whether the initial balance was correctly set, while in line 16, after the check whether the current user holds the necessary balance on her account (line 15, we check if the transferred amount is smaller or equal to the account balance. This assertion is never reached in case of insufficient funding, since the branching statement in line 15 returns and quits the function. Finally we check in line 19 whether the send money is now on the account of the receiver.

```
pragma solidity ^0.4.4;

// This is just a simple example of a coin like contract.
// It is not standards compatible and cannot be expected to talk to other
// coin/token contracts. If you want to create a standards compliant
// token, see: https://github.com/ConsenSys/Tokens. Cheers!

contract MetaCoin {
        mapping (address => uint) balances;

        event Transfer(address _from, address _to, uint _value);

        function MetaCoin() {
                balances[tx.origin] = 1000;
                // spin_assert(balances[tx.origin] == 1000)
        }

        function sendCoin(address receiver, uint amount)
                                        returns(bool sufficient) {
                if (balances[msg.sender] < amount) return false;
                // spin_assert(balances[msg.sender] >= amount)
                balances[msg.sender]  = amount;
                balances[receiver] += amount;
                // spin_assert(balances[receiver] >= amount)
                Transfer(msg.sender, receiver, amount);
                return true;
        }

        function getBalance(address addr) returns(uint) {
                return balances[addr];
        }
}
```

Listing 1: Example Solidity coin contract

The *MetaCoin* example that we use to evaluate the implementation is very simple with a minor level of complexity. The advantage is that we can easily

determine its correctness by just looking. To provoke an error that we can find using the SPIN model checker we assume the unlikely scenario that a programmer forgot to check whether the sending account has the necessary balance. So we omit the balance check in line 15.

When starting the translation program with the MetaCoin program it is parsed into an abstract syntax tree and modified by the tool chain as introduced in Section 6. The modified abstract syntax tree is finally translated into a PROMELA model as shown in Listing 2.

Following the formally described translation process the PROMELA model comprises of three parts, where two parts represent the contract methods *send-Coint* (line 6) and *getBalance* (line 40), while the third part is the *init* section (line 44) holding the logic contained in the constructor of the smart contract and responsible for the spawning of the processes.

```
1  int return_0_getBalance;
2  bool return_0_sendCoin;
3  byte msg_sender;
4  int balances[255];
5
6  proctype sendCoin(){
7          msg_sender = 0;
8
9          #define LOW_sendCoin_amount 0
10         #define HIGH_sendCoin_amount 2147483647
11         int amount;
12         amount = LOW_sendCoin_amount;
13         do
14                         :: amount < HIGH_sendCoin_amount → amount++
15                         :: break
16         od
17
18         #define LOW_sendCoin_receiver 0
19         #define HIGH_sendCoin_receiver 254
20         byte receiver;
21         receiver = LOW_sendCoin_receiver;
22         do
23                         :: receiver < HIGH_sendCoin_receiver → receiver++
24                         :: break
25         od
26
27         assert(balances[msg_sender] >= amount);
28
29         d_step {
30                 balances[msg_sender] = balances[msg_sender]    amount;
31                 balances[receiver] = balances[receiver] + amount;
32                 assert(balances[receiver] >= amount);
33                 return_0_sendCoin = true;
34                 goto return_label_sendCoin
35                 return_label_sendCoin:
```

```
36          }
37
38  }
39
40  proctype getBalance(){
41          #define LOW_getBalance_addr 0
42          #define HIGH_getBalance_addr 254
43          byte addr;
44          addr = LOW_getBalance_addr;
45          do
46                          :: addr < HIGH_getBalance_addr → addr++
47                          :: break
48          od
49
50          assert(addr > LOW_getBalance_addr)
51
52          d_step {
53                  return_0_getBalance = balances[addr];
54                  goto return_label_getBalance
55                  return_label_getBalance:
56          }
57  }
58
59  init {
60          atomic {
61                  balances[0] = 1000;
62                  assert(balances[0] == 1000);
63          }
64
65          #define MIN_PC_COUNT 0
66          #define MAX_PC_COUNT 10
67          byte pc_count;
68          pc_count = MIN_PC_COUNT;
69          do
70                          :: pc_count < MAX_PC_COUNT → pc_count++; run sendCoin();
71                          :: pc_count < MAX_PC_COUNT → pc_count++; run getBalance();
72                          :: pc_count >= MAX_PC_COUNT → break
73          od
74
75  }
```

Listing 2: Translated PROMELA model

The logic in the smart contract functions that manipulates data is translated into equal PROMELA statements. Note that we omitted the balance check in the *sendCoin* function (line 6), but still use the assertion that tests whether this check was done in line 27.

When applying the SPIN model checker on the model in Listing 2 it determines that the assertion in line 27 was violated (Figure 2). It generates a *trail* files that shows the path through the program that leads to the violation of the assertion.

By analyzing the (lengthy) trail file it becomes obvious that we forgot to

```
error: max search depth too small
pan:1: assertion violated (balances[msg_sender]>=amount) (at depth 9999)
pan: wrote paper_faulty.prom.trail

(Spin Version 6.4.6 -- 2 December 2016)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 1128 byte, depth reached 9999, errors: 1
    10004 states, stored
       39 states, matched
    10043 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
   11.029       equivalent memory usage for states (stored*(State-vector + overhead))
   37.074       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
    8.998       memory lost to fragmentation
  156.610       total actual memory usage



pan: elapsed time 0.18 seconds
pan: rate 55577.778 states/second
```

Figure 3: Output of the SPIN model checker

implement the balance check. Although this is a very artificial and simple example it demonstrates the basic idea of applying model checking for verifying the correctness of smart contracts. By extending the supported syntax and utilizing the functionality provided by SPIN the range of application can be even more extended.

8. Conclusion

This paper presents smart contracts as a means for the automation of collaboration and business logics. As such, smart contracts are instrumental for the automation and its implementation of organizational structures and their processes described by DAO. The correctness of these implementations is decisive due to the immutability of any blockchain, hence once initiated smart contracts run for ever unless adaptations are agreed upon.

This paper developed a tool chain for translating chain code modelled in Solidity via its operational semantics in logics towards an automata-based rep-

resentation of the original chain that can be verified by a model checker. The last step of this tool chain generates a code representation in PROMELA, that can be checked by model checker such as SPIN. Model checking results in a confirmation that the code behaves as specified or counterexamples unveiling coding errors. We used a rather common currency example to illustrate missing specifications in this smart contract code, i.e. availability of coins even for negative balances.

The tool chain itself is founded in formal models describing the semantics of Solidity code as well as the transformation calculus towards PROMELA models serving as input for the model checker. We have decided to base the semantic interpretation of Solidity code on operational semantics. Our language scope focuses on core elements. Some data types such as strings or some functional elements such remote calls of other contracts have to be included in the future. However, the core computational elements are in place.

Another extension revolves around the limitation of the search space for model checking. Our claim is to reduce the search space to be explored by the model checker. One way to limit the search space is the necessity to spend gas for each transaction. Hence, limiting the availability of gas, will reduce the search space to be explored due to its lessened complexity.

A third extension addresses the readability of smart contracts. Imagine the specification of a complex organizational structure and its processes in terms of script-oriented languages such as Solidity. The level of precision convinces from a computer scientists point of view. But a domain expert is potentially overwhelmed by the mere complexity of the code. More abstract languages are required to represent organizational structure and their processes. Languages and tools for process management as well as business modelling appear attractive, because they allow for a domain-oriented modelling while also having sufficient formality to be translated to script- or Petri Net-oriented process specifications.

9. Acknowledgment

The authors would like to thank Matthias Volk from the RWTH Aachen University for reviewing the paper and providing valuable comments.

References

[1] N. Szabo, Formalizing and securing relationships on public networks, First Monday 2 (9). doi:10.5210/fm.v2i9.548.
URL http://ojphi.org/ojs/index.php/fm/article/view/548

[2] E. A. Emerson, 25 years of model checking, Springer-Verlag, Berlin, Heidelberg, 2008, Ch. The Beginning of Model Checking: A Personal Perspective, pp. 27–45. doi:10.1007/978-3-540-69850-0_2.
URL http://dx.doi.org/10.1007/978-3-540-69850-0_2

[3] C. Baier, J. Katoen, Principles of Model Checking, MIT Press, 2008.
URL https://books.google.de/books?id=nDQiAQAAIAAJ

[4] G. J. Holzmann, The model checker spin, IEEE Transactions on Software Engineering 23 (1997) 279–295.

[5] G. J. Holzmann, Mars code, Commun. ACM 57 (2) (2014) 64–73. doi:10.1145/2560217.2560218.
URL http://doi.acm.org/10.1145/2560217.2560218

[6] M. Ben-Ari, Principles of the Spin Model Checker, Springer London, 2008.
URL https://books.google.de/books?id=eVTN8UanIGcC

[7] P. Daian, Analysis of the dao exploit, hackingdistributed.comHttp://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[8] K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, E. Shi, Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab, IACR Cryptology ePrint Archive 2015 (2015) 460.

[9] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on ethereum smart contracts sok, in: Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204, Springer-Verlag New York, Inc., New York, NY, USA, 2017, pp. 164–186. doi:10.1007/978-3-662-54455-6_8.
URL https://doi.org/10.1007/978-3-662-54455-6_8

[10] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf (2008).

[11] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, ACM, New York, NY, USA, 2016, pp. 254–269. doi:10.1145/2976749.2978309.
URL http://doi.acm.org/10.1145/2976749.2978309

[12] M. Dhawan, Analyzing safety of smart contracts, IBM Research.

[13] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, S. Zanella-Béguelin, Formal verification of smart contracts: Short paper, in: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS '16, ACM, New York, NY, USA, 2016, pp. 91–96. doi:10.1145/2993600.2993611.
URL http://doi.acm.org/10.1145/2993600.2993611

[14] J.-C. Fillitre, A. Paskevich, Why3 - where programs meet provers., in: M. Felleisen, P. Gardner (Eds.), ESOP, Vol. 7792 of Lecture Notes in Computer Science, Springer, 2013, pp. 125–128.
URL http://dblp.uni-trier.de/db/conf/esop/esop2013.html#FilliatreP13

[15] Y. Hiray, Formal verification of ethereum contracts (yoichi's attempts), github.comHttps://github.com/pirapira/ethereum-formal-verification-overview/blob/master/README.md.

[16] F. Idelberger, G. Governatori, R. Riveret, G. Sartor, Evaluation of logic-based smart contracts for blockchain systems, in: RuleML, 2016.

1090 [17] I. Sergey, A. Kumar, A. Hobor, Scilla: a smart contract intermediate-level language, CoRR abs/1801.00687.

[18] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, B. Moore, Y. Zhang, D. Park, G. Roşu, Kevm: A complete semantics of the ethereum virtual machine, in: Computer Security Foundations Symposium,
1095 2018.

[19] TrailOfBits, Manticore: Symbolic execution for humans, trailofbits.comHttps://blog.trailofbits.com/2017/04/27/manticore-symbolic-execution-for-humans/.

[20] ConsenSys, Mythril - security analysis tool for ethereum smart contracts,
1100 github.comHttps://github.com/ConsenSys/mythril.

[21] R. Revere, Solgraph - visualize solidity control flow for smart contract security analysis, github.comHttps://github.com/raineorshine/solgraph.