Modeling and Verification of Solidity Smart Contracts with the B Method

Faycal Baba, Amel Mammar, Marc Frappier, Régine Laleau

▶ **To cite this version:**

Faycal Baba, Amel Mammar, Marc Frappier, Régine Laleau. Modeling and Verification of Solidity Smart Contracts with the B Method. Engineering of Complex Computer Systems, Jun 2024, Limassol (Chypre), Cyprus. hal-04850532

HAL Id: hal-04850532

https://hal.science/hal-04850532v1

Submitted on 20 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Modeling and Verification of Solidity Smart Contracts with the B Method

Faycal Baba
University Paris-Est Crétei
Créteil, France
Université de Sherbrooke
Sherbrooke, Canada
faycal.baba@usherbrooke.ca

Amel Mammar
Télécom SudParis
Every, France
amel.mammar@telecom-sudparis.eu

Marc Frappier
Université de Sherbrooke
Sherbrooke, Canada
marc.frappier@usherbrooke.ca

Régine Laleau
University Paris-Est Créteil
Créteil, France
laleau@u-pec.fr

***ABSTRACT*—Smart contracts written using the SOLIDITY programming language of the ETHEREUM platform are well-known to be subject to bugs and vulnerabilities, which already have led to the loss of millions of dollars worth of assets. Since smart contract code cannot be updated to patch security flaws, reasoning about smart contract correctness to ensure the absence of vulnerabilities before their deployment is of the utmost importance. In this paper, we present a formal approach for generating correct smart contracts from B specification that verify safety properties. Our approach consists of two phases: first a smart contract and its properties are specified and verified in B, then a set of rules we defined are applied to generate the correct smart contract code in SOLIDITY. The approach is implemented in a tool that can generate SOLIDITY contract from a proven B project. The whole approach is demonstrated by a case study on the ERC-20 (ETHEREUM Request for Comments 20) Wrapped Ether (WETH) contract, which is abstractly specified in B, with invariants stating correctness properties, modeled checked with PROB for temporal properties, implemented in B0, proven correct, and automatically translated into a Solidity contract.**

***Index Terms*—Smart contracts, Solidity, Blockchain, Formal modeling & verification, B Method, Refinement.**

I. INTRODUCTION

In recent years, blockchain technology has garnered significant attention from both industry and academia due to its potential for decentralization, security, and transparency. Initially introduced through the creation of Bitcoin, blockchain serves as a distributed and immutable ledger that securely records transactions in a transparent manner [1]. Unlike conventional systems reliant on trusted intermediaries such as banks, blockchain employs consensus among nodes to validate and record cryptocurrency transactions. Ethereum, often regarded as blockchain 2.0, stands out as a pioneering platform that not only features a cryptocurrency called ether, but also enables the execution of self-executing programs known as smart contracts [2].

Smart contracts are self-executing agreements that can be executed on a blockchain when predetermined conditions are met. For example, they can automate agreements among multiple parties, enabling one party to trigger a state change that, in turn, allows another party to access ether stored within the smart contract. To create smart contracts, developers primarily utilize SOLIDITY [3], a Turing-complete high-level language similar to JavaScript. This SOLIDITY code is then compiled into Bytecode, a lower-level code, and deployed on the Ethereum blockchain.

Like any software program, smart contracts are susceptible to bugs and vulnerabilities, with potentially grave financial implications [4]. Compounded by the immutability of blockchain, rectifying post-deployment issues becomes impossible. Ethereum has encountered various attacks and vulnerabilities resulting in significant losses, such as the infamous DAO smart contract attack [5], where an attacker exploited a vulnerability (later known as the reentrancy vulnerability) within the contract code, stealing over $50 million worth of ether.

Given these challenges, ensuring the safety and security of smart contracts before deployment to the blockchain is of paramount importance. This paper presents an approach to the formal modeling and verification of SOLIDITY smart contracts using the formal B method [6]. We adopt a classical approach, commencing with the definition of an abstract B model. This abstraction allows for a simpler expression and verification of smart contract properties and functionalities. The verification process leverages a range of techniques, including specification animation, theorem proving and model checking within the existing B Method toolset. Following successful verification, the B refinement process is employed to transform the abstract specification into a more concrete representation, aligning closely with SOLIDITY data and control structures. Then, through established translation rules, the entire model is subsequently translated into SOLIDITY code. The translation rules are implemented in a tool we developed using *ANTLR*[1] and *Jetbrains MPS*[2].

The remainder of this paper is organized as follows: In Section II, we begin with an overview of the literature related to our contribution in the verification of SOLIDITY smart contracts. Then, in Section III, we present the basic concepts of the B method and the SOLIDITY language. Section IV describes our approach to modeling, verifying and implementing

[1] https://www.ANTLR.org/
[2] https://www.jetbrains.com/mps/concepts/

SOLIDITY smart contracts, up to the automatic translation of B0 code (the implementation language of B) into SOLIDITY. The entire approach is illustrated by a case study concerning the WETH token smart contract. Section V concludes and presents some future work.

II. RELATED WORK

Several works precede ours in the analysis, verification, and modeling of SOLIDITY smart contracts, reflecting diverse approaches, techniques, and tools. In this section, we provide an overview of relevant works categorized by the formal modeling and verification techniques they employ: i) SOLIDITY code/bytecode analysis, ii) SOLIDITY code verification, and iii) correct-by-construction SOLIDITY code generation.

In the first category, researchers focus on detecting common code patterns leading to known vulnerabilities (e.g., reentrancy, transaction order dependence, timestamp dependence) using static analysis techniques that examine program artifacts (e.g., Control Flow Graph, Abstract Syntax Tree). Prominent examples include the works [7]–[10], which rely on Symbolic Execution. The tool *smartcheck* in [11] identifies problematic patterns using XPath queries, and fuzz testing in [12]–[14] detects various SOLIDITY vulnerabilities. These approaches may not explore all program paths, potentially leading to false negatives. Purely syntactic techniques might not fully account for operational semantics or execution environment intricacies [15], compromising the analysis soundness and completeness.

In the second category, researchers translate SOLIDITY code into formal models and verify properties using various tools and techniques. Model checking was used in [16], where the authors use *NUSMV* to model blockchain applications and to verify temporal properties in CTL. In [17], Coloured Petri Nets are used to model and verify smart contract specifications using LTL properties. Although model checking is employed in various approaches, its limitation are well known, and approaches that rely on it, such as [16], [17], often face the problem of state explosion when dealing with complex or infinite models [15]. F* is used in [18] to formally verifies models of smart contracts, analyzing correctness and gas consumption with tools like Solidity* and EVM*. For now, this work only considers a restricted subset of SOLIDITY. It is also worth noting that F* program verification faces challenges including complexity, and steep learning curve. In [19], an EVENT-B model generated from SOLIDITY code is used to formalize smart contracts, the model is then simulated and verified using the tool PROB[3]. Notably, the translation process only considers subset of the SOLIDITY language

In the third category, the focus is on creating correct-by-construct smart contracts. For instance, in [20], smart contracts are modeled as FSMs and verified with LTL properties, later generating SOLIDITY code. The authors of [21] employ EVENT-B and present a top-down verification and refinement approach that theoretically generates SOLIDITY programs from EVENT-B models, although they did not provide a concrete implementation or practical demonstration of their solution. The authors of [22] further explores this approach, introducing EB2Sol, a tool for modeling, verifying, and generating SOLIDITY smart contracts. It is worth noting that this work considers a restricted subset of SOLIDITY and can only generate sequential programs consisting solely of assignments statements, without support for conditional (if) or iterative (while) statements.

Our work falls within the third category. We use the B method [6] to model and verify SOLIDITY smart contracts. The B method supports animation and model checking (LTL and CTL) using ProB[4], and theorem proving using Atelier B[5]. These tools have an industrial strength and have been routinely used in safety critical industrial projects for over 25 years [23]. These tools allow one to express and verify complex models and properties. Developers can perform comprehensive checks to ensure contract correctness, logical consistency, and avoid potential deadlocks. Our approach encompasses a broader subset of the SOLIDITY language, facilitating the modeling and verification process compared to [22], which does not cover conditional (if) or iterative (while) statements.

III. BACKGROUND

In this section, we present the main concepts of the B method and the SOLIDITY language that are relevant for this paper.

A. The B Method

The B method [6] is a formal method for modeling and verifying software systems. It is based on set theory and first-order logic, allowing for the creation of precise and unambiguous system descriptions. At the most abstract level, the system is represented by an abstract machine that includes both the structural aspect, describing system states, and the behavioral aspect, which describes state evolution.

The structural aspect may define enumerated or abstract user types in the 'SETS' clause. It can also declare variables (resp. constants) in the 'VARIABLES' (resp. 'CONSTANTS') clause, with type specifications in the 'PROPERTIES' clause for constants and the 'INVARIANT' clause for variables, using first order predicates. Additional properties, like safety or integrity of the machine state, can be specified in the 'INVARIANT' clause.

The behavioral aspect comprises the 'INITIALISATION' clause, which assigns initial values to variables, and the 'OPERATIONS' clause, used to describe operations that can modify variable values through simultaneous and possibly nondeterministic substitutions. An abstract machine can reference other machines through various clauses, with different access rights. The relevant clauses for our purposes are the 'SEES' and 'INCLUDES' clauses. The 'SEES' clause provides read-only access, whereas 'INCLUDES' grants additional access, including the ability to invoke operations. Proof obligations

[3]https://prob.hhu.de/

[4]https://prob.hhu.de

[5]https://www.atelierb.eu/

must be discharged to verify invariant preservation through the execution of operations in the abstract model.

One of the strengths of the B method is its refinement process that consists in defining successive refinements in a B model development, each of them adding more detail and reducing abstraction of the initial abstract machine. The final stage, B0 implementation, represents a concrete model that can be automatically translated into C or Ada, and is subject to constraints (e.g., it can only use concrete data types and deterministic substitutions), ensuring compatibility with programming languages like C or ADA (and in our case, SOLIDITY). An implementation component largely shares the same clauses as an abstract machine, with some differences. 'VARIABLES' (resp. 'CONSTANTS') are replaced by 'CONCRETE_VARIABLES' (resp. 'CONCRETE_CONSTANTS'), and an additional 'VALUES' clause is used to assign values to implementation constants. The implementation component can use 'SEES' to reference an abstract machine, and the 'INCLUDES' clause is replaced by the 'IMPORTS' clause, which disallows read access on variables. The refinement process generates proof obligations that must be fulfilled to ensure that the implementation component preserves the properties of the abstract model.

Table I presents the B elements used in our approach to specify and generate SOLIDITY programs. x and E denote a variable and an expression respectively, T and T' denote *substitutions*, P, G, I are predicates, S and S' are sets. In the B method, a substitution is a construct that allows for the specification of operations.

TABLE I
SUBSET OF RELEVANT B ELEMENTS

B element	**B Syntax**
Enumerated set	$< enum_name >$ = $\{Id_1, Id_2,..,Id_n\}$
Total function type	$S \rightarrow S'$
Total injective function	$S \rightarrowtail S'$
Array type	$(n..n') \rightarrow S$
Inclusion	$S' \subseteq S$
Belongs	$x \in S$
Lambda Expression	$\lambda\ x.\ (\ P \mid E\ )$
Variable declaration	**VARIABLES** v **INVARIANT** $v \in S$
Constant declaration	**CONSTANTS** c **PROPERTIES** $c \in S$ **VALUES** $c = E$
B structure type variable declaration	$x \in$ **struct**$(m_1 \in S, \ldots, m_n \in S')$
Assignment substitution	$x := E$
Simultaneous substitution	$T \| T'$
Sequential substitution	$T; T'$
Conditional statement	**IF** P **THEN** T **ELSE** T' **END**
Precondition substitution	**PRE** G **THEN** T **END**
While substitution	**WHILE** P **DO** T **INVARIANT** I **VARIANT** E **END**
Var statement : Declares local variable(s) in an operation body	**VAR** $Identifier^+$ **IN** T **END**

B. Smart Contract and Solidity

The ETHEREUM blockchain defines two primary types of accounts: External Owned Accounts (EOAs) and Smart Contract Accounts (SCAs). EOAs are under the control of blockchain users, each associated with a unique address and a private key for initiating transactions. In contrast, SCAs are governed by the immutable code of deployed smart contracts. Both account types can receive, hold, and send ether. However, only EOAs can initiate transactions, which can be simple ether transfers or a call to a smart contract function which can transfer ether among accounts and also invoke functions of other smart contracts.

SOLIDITY is the most popular programming language for ETHEREUM smart contracts. SOLIDITY is a high-level object-oriented language, based on C++ and JavaScript and is statically typed. The language supports inheritance, libraries and complex user-defined types among other features [3]. To be deployed on the ETHEREUM computing platform EVM (ETHEREUM Virtual Machine), SOLIDITY code needs to be compiled into Bytecode. Once the Bytecode is deployed on the platform, a new smart contract is created and its state (i.e., its variables and balance of ether) is recorded on the blockchain. In SOLIDITY, aside from the constructor which is executed once at the contract deployment, all other functions can serve as entry points. A function can be called by users or other smart contracts, and can read and update the smart contract state. To ensure that the execution of a function terminates, the initiator of a call pays a fee for its execution, measured in units of *gas*.

In our approach, we consider a subset of SOLIDITY which contains the main components of the language and is general enough to express useful contracts:

- State variables and constants definitions
- Enumeration type declaration
- Structure type declaration
- Constructor declaration
- Function declaration

We consider the following SOLIDITY types: int (integer), $uint$ (unsigned integer), $bool$ (boolean), $address$, $bytes$, $structtype$, $mapping$ (A mapping is a set of pairs $(key, value)$), enumeration type, and $arrays$ type. There are types not supported yet, including the $contract$ type, the $function$ type, and the $fixedpointnumbers$ type.

The type *address payable*, an extension of the type *address*, offers additional methods to a smart contract to transfer ether to a variable of *address payable* type. The keyword *payable* can also be added to a function signature and denotes that the function can receive ether from the account of the caller of the function.

A SOLIDITY function is always invoked and executed within a context that contains the invocation message (i.e., the function called with its parameters), denoted by msg, and the last block of the blockchain, denoted by $block$. These elements are stored in the global name space of SOLIDITY. In

our subset, we consider the following elements of this global name space:

- $msg.sender$: denotes the caller of a function, which can be a user or another smart contract.
- $msg.value$: denotes the value of ether sent with the message.
- $block.timestamp$: denotes the current timestamp in the blockchain.
- $address(this)$: refers to the address of the current smart contract denoted by $this$.
- $x.balance$: returns the balance of ether of address x.

In a function body, we can find standard programming statements (Assignment, If and While statements). Additionally, to initiate ether transaction, we consider the SOLIDITY function *a.transfer(m)*, which sends an amount m of ether from the contract balance to address a. SOLIDITY also provides mechanisms for handling conditions and errors inside a smart contract function. Function *revert()* stops contract execution and reverts state changes to its value before the call to the function. Function *require(c)* is a conditional revert; it evaluates a condition c; if c is true, then the function proceeds normally, otherwise, a revert is executed.

Listing 1 shows a simple code of a SOLIDITY contract that represents an electronic wallet. The constructor function sets the owner of the wallet as the sender of the message $msg.sender$ when the contract is created. The deposit function allows the owner of the wallet to deposit funds into the wallet, thanks to its keyword *payable* which suffices to denote contract balance update, and thus its body is empty. The withdraw function allows the owner of the wallet to withdraw an amount of money, provided that the owner is the sender of the message and that the amount requested does not exceed the balance of the wallet.

```
1  contract SimpleWallet {
2      address payable owner;
3      constructor() payable{owner = msg.sender;}
4      function deposit() public payable {}
5      function withdraw(uint _amount) public {
6          require(msg.sender == owner);
7          require(address(this).balance >= _amount
           );
8          owner.transfer(_amount);}
9  }
```

Listing 1. Example of a SOLIDITY program

IV. MODELING AND VERIFICATION OF SOLIDITY SMART CONTRACTS

In our approach, illustrated in Figure 1, a SOLIDITY smart contract is modeled as a B project, where the components of the smart contract are modeled first as a B abstract machine, and contract specific properties are expressed and verified using the B method toolset. The abstract model is then refined to obtain a B0 implementation. This refined model bridges the gap between the abstract B representation and the specific data and control structures of the SOLIDITY programming language. The final step of our approach is the generation of a SOLIDITY implementation using a set of transformation rules that accurately transforms the refined B0 model. The implementation of these rules is achieved through a dedicated tool that uses *ANTLR* for parsing B specifications and *JetBrains MPS* for model-driven development and code generation from B0 to SOLIDITY. The generated SOLIDITY code inherits the correctness and reliability established through the B method formal verification process, providing a solid base ensuring the correctness of a contract.

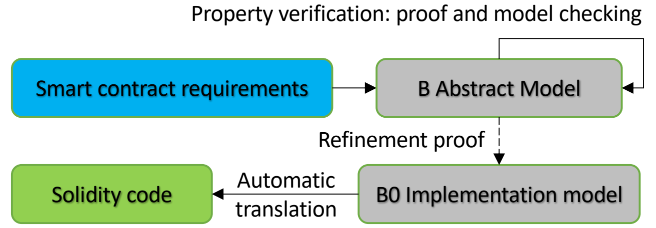

Fig. 1. Process for modeling and verifying a SOLIDITY smart contract

The following sections present a more detailed explanation of our approach, illustrated by a case study concerning the WETH Token smart contract.

A. Case Study Presentation

Wrapped ether (WETH) is a standardized digital representation of ETHEREUM native cryptocurrency, ether, implemented as an ERC-20 token standard[6]. This tokenization process enhances compatibility and ease of use within various ETHEREUM smart contracts. A WETH contract allows users to deposit ether in a contract. The users can then exchange ether (represented as WETH tokens) with each other, without the need of actually transferring ether between their ETHEREUM accounts. The contract keeps track of the balance of each user, called the WETH balance, in a similar way as a bank allows its customers to exchange money between their bank accounts, without actually moving money around. A user can withdraw ether from his WETH account. Thus, transfers between WETH accounts incur lower ETHEREUM transfer gas fees and offer faster processing times.

The main components of the WETH smart contract are:

- The *account* variable: variable of type *mapping*; keeps track of the WETH balance of all addresses.
- The *allowance* variable: variable of type *mapping*; keeps track of allowances; an allowance is when an address a allows an address b to transfer an amount of WETH from a account.
- The *deposit* function: deposit ether in a user account.
- The *withdraw* function: withdraw ether from a user account.
- The *transfer* function: transfer WETH to another account.
- The *approve* function: a allows b to transfer WETH up to a certain limit c (i.e., his allowance) from a account.

[6]https://ethereum.org/en/developers/docs/standards/tokens/erc-20/

- The *transferFrom* function: b transfers from a account to an account c, up to an amount previously allowed to him by a.

To further demonstrate the adaptability and practicality of our formal verification approach, we propose an extension to this contract, where the owner has implemented a new behavior: a reward system for early adopters. Specifically, the contract owner offers one `WETH` token as a reward to each of the first *100* users who successfully accumulate a total balance of *100* tokens in their accounts. The purpose of this new behavior is to demonstrate the ease and flexibility of implementing functionalities that rely on data control structures such as *IF-THEN-ELSE* and *WHILE LOOP* within our formal verification framework using the B method. This demonstration serves as a direct comparison to alternative approaches, notably those employing EVENT-B, where the implementation of such behaviors have not been addressed. To implement this behavior, a new function *rewardTopDepositors* is added. This function can only be executed by the manager of the smart contract, and only once the hundred depositors are selected. The manager must also deposit an amount of ether equal to the amount that is distributed to each of the hundred rewarded depositor.

B. The B Model Architecture of SOLIDITY *Smart Contracts*

1) Types: Firstly, we define the mapping between SOLIDITY types and B sets. Among the SOLIDITY types defined in III-B, some are directly supported in B: $uint \equiv NAT$ or $NAT1$, $int \equiv INT$, SOLIDITY $arrays \equiv$ B $arrays$ (defined as a function), *bool* (boolean) $\equiv BOOL$, and SOLIDITY structure type as B structure type. A SOLIDITY enumeration type is modeled as a B enumeration set.

For the types not supported in B, we define the following rules: The SOLIDITY types *address*, *bytes*, and *string* are modeled as B sets, and defined in a machine called *Solidity_Types*. While *bytes* and *string* are modeled as abstract sets, *address* is modeled as an enumerated set containing the values *THIS*, representing the modeled smart contract address value, and *addr_0*, representing the *null* address value. The *Solidity_Types* also defines the constant *USERS*, which represents a subset of the set *address* excluding the values *THIS* and *addr_0*. The machine *Solidity_Types* will be referenced by other components using *SEES* links. Lastly, for each variable of type *mapping*, a B abstract machine is defined and included/imported in the B abstract/implementation model of a smart contract. Such a machine represents the mapping variable, and it defines a variable of type total function, whose domain represents the keys, and range represents the values. The machine also contains the standard operations of the *mapping* type for inserting and reading pairs.

2) Contract Function Specification: The constructor of a contract is modeled as the initialisation substitution of a B machine, and functions are modeled as operations. Implicit function parameters *msg.sender*, *msg. value*, and *block.timestamp* are modeled as input parameters of a B operation. These parameters are appropriately typed and included among other potential input parameters in the B precondition substitution. To specify the behavior of a contract function, we use a B substitution **IF** C **THEN** T **END**. Condition C is the condition under which the contract function successfully terminates. Substitution T defines the state changes. In the translation process to *Solidity*, we add an **ELSE** *revert()* **END** statement within this top-level **IF** statement to ensure that the state is reverted to its value before the call when C is not satisfied. Thus, a contract function $f(T_1\ p_1, \dots, T_n\ p_n)$ is specified in the following form:

$f(msg_sender, msg_value, p_1, \dots, p_n) =$
PRE $msg_sender \in USERS \wedge msg_value \in NAT1 \wedge$
$p_1 \in T_1 \wedge \dots \wedge p_n \in T_n$ **THEN**
IF C **THEN** T **END**
END

Note that in the B world, this is known as a *defensive* operation specification style, which differs from the traditional *offensive* style of the form $f(\vec{p})$ = **PRE** *Typing Condition of* $\vec{p} \wedge C$ **THEN** S **END**. In B, when an operation f is called, the calling machine must prove that C is satisfied, thus preventing from having to check C in the implementation of f. In our case, there is no other machine that calls our B specification of a contract. Thus, we must establish with the specification of f what happens when C is not satisfied. At the abstract level, when C is not satisfied on a call, an implicit **SKIP** substitution is executed, which leaves the state unchanged. We model this in the implementation by using a call to the $revert$ function.

3) ETHEREUM *Platform Modeling:* To model the *transfer* and *x.balance* SOLIDITY platform functions, which can be used in a number of machines (or implementations) in a B project, we have introduced an abstract machine named *Platform*. This machine can be included (resp. imported) in abstract machines (resp. implementations) of the B project, leveraging B specification modularization. Within *Platform*, the solidity expression $x.balance$ is defined by the variable *balanceOf*, a total function associating each address with its ether balance. *Platform* also defines the *transfer* operation with three input parameters: $sender$ address, $receiver$ address, and transfer $amount$, along with two essential preconditions. The preconditions validate (1) the existence and distinction of sender and receiver addresses and (2) that the sender balance is greater or equal to the transfer amount. Additionally, it also defines the *get_balanceOf(x)* operation, which retrieves an address ether balance.

The B project modeling the `WETH` smart contract contains initially 3 machines: (1) The *Solidity_Types* machine, (2) the *Platform* machine and (3) the abstract machine representing the `WETH` smart contract. To model the two variables $account$ and $allowance$ of type `mapping`, two additional machines $account$ and $allowance$ are created and included in the abstract model of the `WETH` smart contract. Each of these two machines contains a variable of type *total function*, named (resp.) $accountOf$ and $allowanceOf$, where $accountOf$ is typed as *ADDRESS* $\rightarrow$ *NAT*, and $allowanceOf$ is typed as *ADDRESS* $\rightarrow$ *(ADDRESS* $\rightarrow$ *NAT)*. Each ma-

chine also contains two operations; one to insert new values: *set_accountOf* and *set_allowanceOf*, and another to read values: *get_accountOf* and *get_allowanceOf*. We use naming rules *get_*⟨variable_name⟩ and *set_*⟨variable_name⟩ to define mapping machines operations that read (resp. insert) a pair (k, v) in the mapping. Similarly, another operation *set_accountOf_abstract* inserts a set of pairs (k, v) instead of a single pair.

4) B Abstract Specification of the WETH Contract: We specify the operations *deposit*, *withdraw*, *transfer*, *approve*, and *transferFrom*. Each operation changes the state of the machine by modifying the values of the variables *account* and *allowance*. They also define conditions for their execution expressed as a B *IF* substitution that encompasses the body of the operation. For example, one of the conditions to trigger the *withdraw* operation is that the sender (input parameter msg_sender) account must own an amount of tokens greater than or equal to the amount being withdrawn. Additionally, the B method imposes the definition of well-definedness conditions. These conditions ensure, for example, that when an arithmetic expression is used, it must be proven that the operands and the result belong to the defined domain of the operator in B.

To implement the rewarding system, three additional variables have been introduced. (1) *depositors*: This is a subset of the address set, containing users who qualify for the reward system. Users are added to this subset if their total token balance exceeds one hundred. (2) *manager*: This variable represents the contract manager address. (3) *donated*: A Boolean indicating whether the reward has been distributed or not. The *rewardTopDepositors* operation can be invoked by the manager once the specified conditions are met. This operation distributes rewards among the address values stored in set *depositors*.

Listing 2 shows an excerpt of the abstract machine `WETH`

```
1  MACHINE Weth
2  SEES Solidity_Types
3  INCLUDES Platform, account, allowance
4  CONSTANTS threshhold
5  PROPERTIES threshold ∈ NAT
6  VARIABLES manager, depositors, donated
7  INVARIANT
8    depositors ⊆ ADDRESS ∧
9    manager ∈ USERS ∧
10   donated ∈ BOOL ∧
11   balanceOf(THIS) ≥
12     (Σ(ct).(ct ∈ dom(accountOf)| accountOf(ct))) ∧
13   card(depositors) ≤ threshold
14 INITIALISATION
15     ...
16 OPERATIONS
17 deposit(msg_sender, msg_value) =
18 PRE
19   msg_sender ∈ USERS ∧ msg_value ∈ NAT1
20 THEN
21   IF balanceOf(msg_sender) - msg_value ∈ NAT ∧
22     accountOf(msg_sender) + msg_value ∈ NAT ∧
23     balanceOf(THIS) + msg_value ∈ NAT
24   THEN
25     transfer(msg_sender, THIS, msg_value) ||
26     set_accountOf(msg_sender, accountOf(msg_sender)+
           msg_value) ||
27     IF accountOf(msg_sender) + msg_value
28        ≥ threshold ∧
29       msg_sender ∉ depositors ∧
30       card(depositors) < threshold
31     THEN
32       depositors := depositors ∪ {msg_sender}
33 END END END
34 ;
35 rewardTopDepositors(msg_sender, msg_value) =
36 PRE
37   msg_sender ∈ USERS ∧ msg_value ∈ NAT
38 THEN
39   IF msg_value = threshold ∧
40     msg_sender = manager ∧
41     card(depositors) = threshold ∧
42     donated = FALSE ∧
43     balanceOf(THIS) + msg_value ∈ NAT ∧
44     balanceOf(manager) - msg_value ∈ NAT ∧
45     ∀xx.(xx ∈ depositors => accountOf(xx) + 1 ∈ NAT)
46   THEN
47     transfer(manager, THIS, msg_value) ||
48     set_accountOf_abstract(λ xx. (xx ∈ depositors |
           accountOf(xx) + 1)) ||
49     donated := TRUE
50 END END;
51  ...
52 END
```

Listing 2. Excerpt of abstract machine WETH_AM

C. Verification of B Models of Smart Contracts

Formal verification is an important step of our approach. It ensures that the B model captures all the desired behavioral aspects and satisfies its safety properties ensuring its correctness, and this using the formal verification techniques of the B method. For our case study, it means to make sure that the abstract modeling of the `WETH` token captures all the desired behaviors such as token issuance, transfers, and ownership management. For this purpose, the ATELIER B [24] theorem prover can be used to discharge proof obligations associated with invariant preservation. The PROB model checker[7] can be used to animate/model check B model. It can exhibit some missing preconditions related to operation calls or discover problems, such as invariant violations or deadlocks. PROB also allows for the verification of temporal properties expressed as LTL or CTL formulae.

In our model, a property we want to verify is that the ether balance of the model (represented by *balanceOf(THIS)*) remains greater or equal to the total supply of tokens (sum of all values in the mapping variable *accountOf*) during conversions between ether cryptocurrencies and `WETH` Tokens:

$$\text{balanceOf(THIS)} \geq \sum_{ct \in \text{dom(accountOf)}} \text{accountOf(ct)}$$

Proving the abstract B machines with ATELIER B generates 121 proof obligations, of which 111 are automatically proved by the ATELIER B provers, and 10 had to be proved interactively.

Another property that we checked is that the contract doesn't reach a deadlock state, in which case users would be prevented from depositing and withdrawing their ether. This

[7] https://prob.hhu.de/

property cannot be verified using ATELIER B; we use instead the PROB model checker. We can also use PROB to simulate the model. The simulation starts by defining an initial state. Then, we can execute the operations if their preconditions are satisfied, and validate whether the model behavior is consistent with the requirements.

Temporal properties can also be expressed and verified using PROB. For instance, we can verify that at all times, if an address has a token balance greater than 0, then the operations *withdraw* and *transferTo* are enabled. This property is formally expressed as the following PROB LTL formula:

$$G(\{\exists(x).(x \in dom(accountOf) \wedge accountOf(x) > 0))\} \Rightarrow e(withdraw) \wedge \mathrm{e}(transferTo))$$

where the curly brackets {...} are used to express a B predicate, and $e(Op)$ denotes that the operation Op is enabled.

Furthermore, beyond the scope of contract-specific properties verification, our approach also implements precautionary measure to address the reentrancy attack. This type of attack occurs when a *victim contract* initiates a call or ether transfer to a *malicious contract*, which subsequently triggers repeated and recursive calls back to a function within the *victim contract* before the initial call completes, which may cause the different invocations to interact in destructive ways with the *victim contract*. To preempt such detrimental behavior, our strategy involves the modeling and translation of the ether exchange function, specifically by opting to employ the SOLIDITY *transfer* function. Notably, the *transfer* function is adept at mitigating the risk of reentrancy due to its inherently limited allocation of *gas*. This allocation, sufficient for only a single call, effectively blocks potential recursive invocations and protects the contract against reentrancy-based exploits.

D. Refinement of the B Models

The B refinement process is used to gradually transform an abstract specification into a more concrete one, closer to SOLIDITY data and control structures. Each step of the process generates proof obligations, that need to be discharged to ensure the preservation of properties.

For our case study, we have developed an implementation machine, shown in Listing 3, that refines the B abstract model of the `WETH` token. This implementation machine, named `WETH_i`, imports the machines: *Platform*, *account*, and *allowance*. Within `WETH_i`, we redefine the operations established in the abstract model. These re-definitions must adhere to our current B translatable subset, which includes the set of expressions and conditions of the B implementation language subsets, along with the assignment, the conditional, the loop and the *VAR-IN* instructions.

At the implementation level, variables and constants of the abstract model are replaced with new concrete ones, and a gluing invariant is added to define a relationship between the abstract variables/constants and their concrete counterparts. For example, the set *depositors* of the abstract machine is replaced by two variables: (1) array *depositors_i* which contains the depositors; (2) variable *index* of type NAT, which denotes the number of depositors currently stored in array *depositors_i*. Consequently, the invariant '$index = card(depositors)$' (line 16) equates the value of the implementation variable *index* to the size of the set *depositors* in the abstract machine, and the invariant '$depositors_i[0..index-1] = depositors$' (line 17) defines the relation between the content of this set *depositors* and the range of the concrete array *depositors_i*. Variable *depositedOver100*, declared in machine *depositedOver100*, is a mapping from *address* to *boolean*; it serves as an indicator to track whether a qualifying user *address* has already been added to the *depositors_i* array, preventing duplicate entries. It avoids looping over array *depositors_i* in operation *deposit* to find out if a depositor has already been added. It implements the condition of line 29 in Listing 2. Since it is not possible to loop over a SOLIDITY mapping, the array *depositors_i* is still needed in function *rewardTopDepositors* to loop over depositors to transfer funds to them. This example shows how consistency between dependent variables can be proved in the implementation. Here, variables *depositors_i* and *depositedOver100* contain the same information (lines 17 and 19), and the implementation invariant ensures that it is consistent with the abstract variable *depositors* (line 19).

Some expressions of the B abstract machine language cannot be used in B0, like the condition $\forall x.x \in \text{depositors} \Rightarrow \text{accountOf}(x) + 1 \in NAT$ in the abstract operation *rewardTopDepositors*, because it uses a universal quantifier $\forall$. They must be implemented using concrete constructs like the loop from lines 61 to 72 of Listing 3.

```
1 IMPLEMENTATION B_weth_i
2 REFINES B_weth
3 SEES Solidity_Types
4 IMPORTS Platform, account, allowance,
    depositedOver100
5 CONCRETE_CONSTANTS threshold_i
6 PROPERTIES threshold_i ∈ NAT ∧ threshold_i =
    threshold
7 VALUES threshold_i = 100
8 CONCRETE_VARIABLES manager_i, depositors_i, index,
    donated_i
9 INVARIANT
10  // TYPING INV
11  index ∈ NAT ∧ index>=0 ∧
12  donated_i ∈ BOOL ∧
13  depositors_i ∈ 0..threshold_i → ADDRESS ∧
14  manager_i ∈ USERS ∧ manager_i = manager ∧
     donated_i = donated ∧
15  // GLUING INV
16  index = card(depositors) ∧
17  depositors_i[0..index-1] = depositors ∧
18  (0..index-1) <| depositors_i : 0..index-1 ↣
     depositors ∧
19  depositedOver_100~[{TRUE}] = depositors
20 INITIALISATION
21    index := 0;
22    depositors_i := (0..threshold_i) * {addr_0};
23    ...
24 OPERATIONS
25 deposit(msg_sender, msg_value) =
26 BEGIN
27  VAR senderBalance, senderAccount, thisBalance IN
28   senderAccount <-- get_accountOf(msg_sender);
29   senderBalance <-- get_balanceOf(msg_sender);
30   thisBalance <-- get_balanceOf(THIS);
31   IF thisBalance + msg_value ≤ MAXINT ∧
```

```
      senderBalance - msg_value ≥ 0 ∧ senderAccount
       + msg_value ≤ MAXINT
32   THEN
33    set_accountOf(msg_sender, senderAccount +
         msg_value);
34    transfer(msg_sender, THIS, msg_value);
35    VAR distinct IN
36     distinct <-- get_depositedOver_100(msg_sender);
37      IF senderAccount + msg_value ≥ threshold_i ∧
          distinct = FALSE ∧ index < threshold_i
38      THEN
39       depositors_i(index) := msg_sender;
40       set_depositedOver_100(msg_sender, TRUE);
41       index := index +1
42 END END END END END
43 ;
44 rewardTopDepositors(msg_sender, msg_value) =
45  BEGIN
46   VAR thisBalance, managerBalance IN
47    thisBalance <-- get_balanceOf(THIS);
48    managerBalance <-- get_balanceOf(manager_i);
49    IF msg_value = threshold_i ∧
50     msg_sender = manager_i ∧
51     index = threshold_i ∧
52     donated_i = FALSE ∧
53     thisBalance + msg_value ≤ MAXINT ∧
54     managerBalance - msg_value ≥ 0
55    THEN
56     VAR jj, safe IN
57      //* jj ∈ NAT
58      //* safe ∈ BOOL
59      jj := 0;
60      safe := TRUE;
61      WHILE jj < index ∧ safe = TRUE DO
62       VAR depositorBalance IN
63        depositorBalance <-- get_accountOf(
            depositors_i(jj));
64        safe := bool(depositorBalance + 1 ≤ MAXINT);
65        jj := jj+ 1
66       END
67      INVARIANT 0≤index ∧ jj≤index ∧ jj≥0 ∧
68       safe = bool(∀xx.(xx ∈ ran((0..jj-1) <|
           depositors_i) => accountOf(xx) + 1 ∈ NAT))
            ∧
69       donated_i= FALSE  ∧
70       ∀xx.(xx ∈ ran((0..jj-2) <| depositors_i) =>
           accountOf(xx) + 1 ∈ NAT)
71      VARIANT index - jj
72     END;
73     IF (safe=TRUE) THEN
74      transfer(msg_sender, THIS, msg_value);
75      donated_i := TRUE;
76      VAR ii, depositorBalance IN
77      //* ii ∈ NAT
78      //* depositorBalance ∈ NAT
79      ii := 0;
80      WHILE ii < index DO
81       depositorBalance <-- get_accountOf(
           depositors_i(ii));
82       set_accountOf(depositors_i (ii),
           depositorBalance + 1);
83       ii := ii+ 1
84      INVARIANT ii=threshold_i or ii∈ dom(
          depositors_i) ∧
85       accountOf =
86       accountOf$0<+(λxx. (xx ∈ depositors_i[0..(ii
          -1)] | accountOf$0(xx) + 1)) ∧
87       threshold_i = threshold ∧
88       donated_i = TRUE ∧ safe= TRUE ∧
89       depositors_i[0..(ii-1)]<∈ depositors ∧
90       jj=index ∧
91       ∀xx.(xx ∈ ran((ii+1..index-1) <| depositors_i)
            => accountOf(xx) + 1 ∈ NAT)
92      VARIANT index - ii
93 END END END END END END END
94 ...
95 END
```

Listing 3. Excerpt of the Implementation component WETH_IM

E. B to Solidity Translation

The final part of our approach is the transformation of the B project into SOLIDITY code. For that purpose, we defined a set of translation rules, which have been implemented in a dedicated tool created with *JetBrains MPS*. The translation rules take into account both the implementation component and the initial abstract model, since some necessary information are inherited by refinement, and they must be retrieved from the abstract machine, or some of its refinements. For instance, in the abstract model, the precondition substitution '*PRE* G *THEN* T' of an operation serves to define typing predicates of operation parameters. This substitution is discarded in the implementation component because the caller of an operation (i.e., another B machine) must prove that the precondition of the called operation is satisfied when it is called. Thus, the translation rules retrieve the typing predicates from the abstract model to generate input parameters of functions. Other substitutions of operations (i.e. assignment, conditional and loop substitutions) are directly translated from the implementation component into their equivalent SOLIDITY statement. Table II shows an excerpt of the translation rules. The other B0 constructs are easily translated into their equivalent SOLIDITY constructs (assignment, if-then-else, loop, variable declaration).

TABLE II
EXCERPT OF THE TRANSLATION RULES B TO SOLIDITY.

B construct	**Solidity Language**
Implementation M	**contract** $M\{\ \}$
CONCRETE_CONSTANTS c **PROPERTIES** $c \in S$ **VALUES** $c := E$	$[\![S]\!]$ **constant** $c = E$
CONCRETE_VARIABLES v **INVARIANT** $v \in S$	$[\![S]\!]\ v$;
INITIALISATION T	**constructor**() $\{\ [\![T]\!]\ \}$
op_name (input_params) = **BEGIN** T **END**	**function** *fun_name* (input_params) **public** $\{\ [\![T]\!]\ \}$

The SOLIDITY *payable* keyword is not explicitly represented in the B architecture; it is generated during the translation. If an operation receives ether by calling the operation *transfer* of machine *Platform*, the equivalent function generated in the translation will have the keyword *payable* added in its signature. Also, variables that are used in the *to* parameter of a transfer operation call are casted as *address payable*. For example, if the B model initiates a transfer of an amount m to an address variable dst, the variable will be translated with a quick cast to the type *address payable*: *payable(dst).transfer(m)* in order to use the *transfer* function of SOLIDITY.

Any imported machine within the implementation component, with the exception of *Platform*, corresponds to a *mapping*

variable and is translated as such in SOLIDITY. The types of both the domain and range of the mapping contained within the machine are translated to their respective SOLIDITY types, as described in Section IV-B. Listing 4 shows an excerpt of the SOLIDITY code generated.

```
1 contract Weth
2 {
3 mapping (address => uint) private accountOf;
4 mapping (address => mapping (address => uint))
      private allowanceOf;
5 mapping (address => bool) private depositedOver_100;
6 address[] private depositors;
7 ...
8
9 constructor (){ manager = msg.sender; donated =
      false; index = 0;}
10
11 function deposit ( ) payable public {
12   uint senderAccount = accountOf[msg.sender];
13   uint senderBalance = msg.sender.balance;
14   uint thisBalance = address(this).balance;
15   if ( thisBalance+msg.value<=type(uint).max &&
16        senderAccount+msg.value<=type(uint).max &&
17        senderBalance >= msg.value)
18     {
19     accountOf[msg.sender] = senderAccount+msg.value;
20     bool distinct = depositedOver_100[msg.sender];
21     if ( senderAccount+msg.value>=threshold&&
          distinct==false&&index<threshold ){
22       depositors.push(msg.sender);
23       depositedOver_100[msg.sender] = true;
24       index = index+1;
25     }
26   }
27   else { revert();}
28 }
29 function rewardTopDepositors ( ) payable public {
30   uint thisBalance = address(this).balance;
31   uint managerBalance = manager.balance;
32   if ( msg.value==threshold&&msg.sender==manager &&
33       index==threshold&&donated==false &&
34       thisBalance+msg.value<=type(uint).max &&
35       managerBalance-msg.value>=0 )
36     {
37     uint jj; bool safe; jj = 0; safe = true;
38     while(jj<index&&safe==true){
39       uint depositorBalance = accountOf[depositors[
            jj]];
40       safe = depositorBalance+1<=type(uint).max;
41       jj = jj+1;
42     }
43     if ( ( safe==true ) ){
44       donated = true; uint ii; ii = 0;
45       while(ii<index){
46         uint depositorBalance =
47           accountOf[depositors[ii]];
48         accountOf[depositors[ii]] = depositorBalance
              +1;
49         ii = ii+1;
50     }}}
51   else { revert();}
52 }
53 ...
54 }
```

Listing 4. Excerpt of SOLIDITY code generated from the B WETH token model project

The Solidity code generated may include conditions within the main *IF* clause and other code snippets that are implicitly checked by the ETHEREUM virtual machine. In the B abstract machine, these conditions are needed to prove the preservation of the machine invariant, to ensure safety properties. They must be preserved by the B0 implementation and thus appear in the B0 code. For instance, verifying the sender cryptocurrency balance for an Ether transfer and preventing overflow or underflow in integer calculations are implicitly checked in ETHEREUM. As an example, consider line 17 in Listing 4, which checks that the sender balance is greater than or equal to the amount sent. This line is superfluous because the ETHEREUM platform disallows transfers with insufficient funds. These conditions and statements could be eliminated using optimization rules, which we have not done yet.

Currently, our tool lacks a built-in type checker to evaluate the types of certain components in a B implementation when their types are not explicitly defined. To address this limitation, we have introduced a temporary rule in which we explicitly define the types of these components by adding a comment with the following syntax: `//* variable_name : type`. This approach is a temporary solution. We are actively working on the development of a comprehensive type checker for future tool enhancements.

We can as a final step manually check that the same scenarios executed on the B model, using PROB, and the generated SOLIDITY program give the same traces. We use Remix IDE [25], an open source web application that offers an environment for simulating the ETHEREUM blockchain, and used for the development and testing of SOLIDITY smart contracts. Through Remix, we can compile, deploy the generated SOLIDITY code, and simulate execution scenarios of smart contracts. The translation tool and the B proven project associated to this case study can be found in [26]. The implementation generated 131 proof obligations, of which 46 had to be proved interactively.

V. CONCLUSION AND FUTURE WORK

This paper proposes a comprehensive approach to develop formally verified SOLIDITY smart contracts using the B formal method. We propose a specific B modeling for a SOLIDITY smart contract taking into account a subset of SOLIDITY and EVM domain variables and functions. We adopt a classical approach, starting from an abstract model to specify the behavior of a contract and its safety properties that needs to be preserved during its execution. The model can be verified, animated and validated using the whole range of tools that support the B method. Finally, after refinement of the abstract machine, the model can be translated into SOLIDITY using the rules we defined and implemented in a tool using *Jetbrains MPS*. The correctness of our translation rules from B0 to SOLIDITY is straightforward to establish, since we are using simple constructs and simple types whose semantics are the same in B0 and in SOLIDITY. In future works, we plan to extend our framework in multiple directions: first by expanding the SOLIDITY subset considered, for instance, function calls between contracts, and use our approach in the other direction, that is, generate a B abstract machine from a SOLIDITY program to prove properties about it.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Dec 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[3] Ethereum, "Solidity documentation. release 0.7.0." [Online]. Available: https://docs.soliditylang.org/en/v0.7.0/

[4] N. Atzei *et al.*, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust - 6th International Conference, POST 2017*, ser. LNCS. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6_8

[5] V. Buterin, "Critical update re: Dao vulnerability." [Online]. Available: https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/

[6] J. Abrial, *The B-book - assigning programs to meanings*. Cambridge University Press, 1996.

[7] L. Luu *et al.*, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 2016*. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[8] I. Nikolic *et al.*, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC*, 2018. [Online]. Available: https://doi.org/10.1145/3274694.3274743

[9] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*.

[10] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC*, 2018. [Online]. Available: https://doi.org/10.1145/3274694.3274737

[11] S. Tikhomirov *et al.*, "Smartcheck: Static analysis of ethereum smart contracts." [Online]. Available: https://ieeexplore.ieee.org/document/8445052

[12] G. Grieco *et al.*, "Echidna: effective, usable, and fast fuzzing for smart contracts." [Online]. Available: https://doi.org/10.1145/3395363.3404366

[13] T. D. Nguyen *et al.*, "sfuzz: an efficient adaptive fuzzer for solidity smart contracts," 2020.

[14] C. F. Torres *et al.*, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts." [Online]. Available: https://doi.org/10.1109/EuroSP51992.2021.00018

[15] P. Tolmach *et al.*, "A survey of smart contract formal specification and verification," 2022. [Online]. Available: https://doi.org/10.1145/3464421

[16] Z. Nehai, P. Piriou, and F. F. Daumas, "Model-checking of smart contracts," in *IEEE International Conference on Internet of Things 2018, Halifax, NS, Canada, July 2018*. [Online]. Available: https://doi.org/10.1109/Cybermatics_2018.2018.00185

[17] I. Garfatta, K. Klai, M. Graïet, and W. Gaaloul, "Model checking of vulnerabilities in smart contracts: a solidity-to-cpn approach," in *SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022*.

[18] K. Bhargavan *et al.*, "Formal verification of smart contracts: Short paper," 2016. [Online]. Available: https://doi.org/10.1145/2993600.2993611

[19] J. Zhu *et al.*, "Formal simulation and verification of solidity contracts in event-b," in *IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC 2021*, 2021, pp. 1309–1314.

[20] D. Suvorov and V. Ulyantsev, "Smart contract design meets state machine synthesis: Case studies," *CoRR*.

[21] R. Banach, "Verification-led smart contracts," in *Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING 18-22, 2019*, ser. LNCS, 2019, pp. 106–121.

[22] N. K. Singh *et al.*, "Chapter 8 - formal verification and code generation for solidity smart contracts," in *Distributed Computing to Blockchain*, 2023, pp. 125–144. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780323961462000280

[23] M. J. Butler *et al.*, "The first twenty-five years of industrial use of the b-method." Springer.

[24] ClearSy, "Tool atelier b." [Online]. Available: https://www.clearsy.com/en/tools/atelier-b/

[25] "Remix web ide. the native ide for web3 development." [Online]. Available: https://remix.ethereum.org/

[26] "B2sol translation tool." [Online]. Available: https://github.com/ICECCS2024/B2SolPrototype