Formal Verification and Code Generation for Solidity Smart Contracts

Neeraj Kumar Singh, Akshay M Fajge, Raju Halder, Md. Imran Alam

Formal Verification and Code Generation for Solidity Smart Contracts

Neeraj Kumar Singh[a,*], Akshay M. Fajge[b], Raju Halder[b,*], Md. Imran Alam[b,c]

[a]*INPT-ENSEEIHT/IRIT, University of Toulouse, France*
[b]*Indian Institute of Technology Patna, Patna, India*
[c]*Università Ca' Foscari Venezia, Italy*

Abstract

Blockchain technology has gained widespread acceptance in industries such as e-commerce, energy trading, healthcare services, and asset management. Ethereum is an open-source blockchain computing platform with contract functionality. To manage digital assets, it executes bytecode on a simple Solidity stack machine, which is difficult due to Ethereum's openness that allows both programs and anonymous users to call into the public methods of other programs. In such cases, combining trusted and untrusted code for large applications can be risky and lead to catastrophic failure. For example, TheDAO is hacked by an attacker by examining EVM semantics to transfer 50 million USD in Ether. In this chapter, we outline a framework for analysing, verifying, and implementing smart contracts while maintaining functional correctness and required safety properties in Event-B using a correct by construction approach. The refinement approach is used to reduce the complexity of smart contract verification. The Rodin tool is used to develop formal models of smart contract specifications. Furthermore, the developed Event-B models are used to generate smart contracts in Solidity using our developed tool EB2Sol. For prototype development, the development architecture and code generation rules are described in detail. Finally, we demonstrate the scalability and effectiveness of our proposed framework through a case study.

Keywords: Formal methods, Refinement and Proofs, Blockchain, Smart Contracts, Solidity, Code generation, Event-B

1. Introduction

Ethereum [1] is an open-source computing platform, also known as the Ethereum Virtual Machine (EVM), for blockchain technology with contract functionality. The blockchain technology has been adopted successfully in different business sectors, such as e-commerce, banking, finance, insurance, energy trading, healthcare services, and asset management. Manipulation of critical transactions, as well as management of digital assets, making them attractive targets for security

*Corresponding author
Email addresses: nsingh@enseeiht.fr (Neeraj Kumar Singh), fajge_1921cs12@iitp.ac.in (Akshay M. Fajge), halder@iitp.ac.in (Raju Halder), imran.pcs16@iitp.ac.in (Md. Imran Alam)

threats and attacks, which may result in financial losses and data leakage. EVM, on the other hand, executes bytecode of smart contracts written in Solidity [2, 3], a JavaScript-like language, on a simple stack machine to handle and transfer digital assets, which is extremely difficult due to Ethereum's openness, allowing both programs and anonymous users to call into the public methods of other programs. In such cases, the use of trusted and untrusted code together in a large and complex application, particularly one involving financial management or privacy data, can be dangerous. For example, TheDAO is hacked by an attacker by examining EVM semantics to transfer 50 million USD in Ether [4].

Since software plays an important role in the blockchain technology, we need effective ways to evaluate smart contracts in order to certify and ensure safe transactions. The blockchain community and industrial partners are looking for better technology and methods to provide safe and secure transactions. Furthermore, smart contracts must meet safety and security standards. We believe that formal methods have the potential to develop safe and secure systems that are also certifiable with required features that can be used to verify and certify smart contracts.

This chapter contributes a framework for analysing, verifying, and implementing smart contracts while maintaining functional correctness and required safety properties through incremental refinement in Event-B [5]. We use the Rodin [6] tool to develop formal models of smart contract specifications. The primary use of this formal development is to assist in the construction, clarification, and validation of the smart contract requirements. Furthermore, the developed Event-B models are used to generate smart contracts in Solidity using our developed tool EB2Sol. This developed tool covers a subset of the Event-B modelling language in order to generate a Solidity subset. In this chapter, we will go over technical details about implementation as well as design decisions made during prototype development. Finally, we use a case study to demonstrate the scalability and effectiveness of our proposed framework.

In summary, the main contributions of this chapter are :

1. a framework for using formal methods to develop smart contracts from requirements analysis to code generation;
2. smart contracts (in Solidity) are generated from the proven Event-B formal models.
3. EB2Sol is developed to generate Solidity code.
4. the proposed framework is used for developing several case studies in order to formalise the development of smart contracts.

The rest of the chapter is as follows. Section 2 reviews related work. Section 3 briefly overviews key elements of Event-B modelling language, including refinement, and Solidity smart contract. A framework for developing formal Solidity smart contract in Event-B is presented in Section 4. Section 5 outlines the translation principle and development architecture of EB2Sol. Section 6 describes a case study to demonstrate the applicability of the proposed framework, including implementation of Solidity smart contract. In Section 7, we provide an assessment and Section 8 concludes the paper with future work.

2. Related work

Solidity smart contracts are widely used on the Ethereum platform for blockchain technology. Following several attacks [7, 4] in recent years, formal methods are now regarded as first-class

citizens for mitigating potential risks through formal reasoning on defined contracts. Several approaches based on formal methods for the development of smart contracts have been proposed in recent years.

Palina et al. [8] provided a comprehensive overview of formal models and smart contract specifications. They also highlighted some of the identified challenges and gaps in order to guide future research in the area of formal methods for developing trustworthy smart contracts.

In [9], the authors proposed EVM semantics in the K Framework based on the ERC20 Standard Token for formalising and analysing smart contracts. Hirai et al. [10] defined EVM in Lem language that can be translated into many standard interactive theorem provers. In particular, they prove interesting safety properties of Ethereum smart contracts in Isabelle/HOL. The ConCert framework [11] was developed for verifying smart contracts in Coq in order to detect vulnerability.

In [12], the authors proposed a lazy approach to determining input conditions under which the contract terminates or not by statically proving conditional termination and non-termination of a smart contract. This is accomplished by ensuring in advance that both the current state and the contract's input satisfy the termination conditions.

Wang et al.[13] presented VERISOL, a formal verification tool for smart contracts verification based on semantic conformance of smart contracts against a state machine model with access-control policy. They discovered some previously unknown bugs in the published smart contracts, then fixed the bugs and would be able to perform model checking-based verification with VERISOL. In [14], the authors described an SMT-based formal verification module integrated with the Solidity compiler for identifying potential bugs during the compile time, such as arithmetic overflow/underflow, unreachable code, trivial conditions, and assertion fails.

In [15], the authors presented a framework for analysing and veryfing the runtime safety properties as well as functional correctness of Ethereum contracts using the F* functional programming language. In [16], the authors showed small-step semantics of EVM bytecode in F*. They validated the executable code against the Ethereum test suite. Furthermore, they identified some bugs and defined several security properties to prevent them, such as call integrity and atomicity. In a similar vein, [17] presented a mechanism for translating Solidity contracts to Event-B models by defining transfer functions covering a subset of the Solidity language. Further, the produced Event-B model can be refined at different abstraction levels to verify properties associated with Solidity contracts using Rodin [6].

A structured approach to smart contracts verification based on refinement in the Event-B modelling language proposed in [18]. Our work is also in this vein, as we propose a framework based on Event-B formal methods for specifying, analysing, verifying, and implementing smart contracts through refinement by preserving the required safety properties. In addition, we have developed a prototype tool, EB2Sol, to generate Solidity smart contracts from verified Event-B models. As far as we know, this is the first tool for translating Event-B models into Solidity smart contracts.

3. Background

3.1. Event-B Modelling Framework

This section summarises the core components of the Event-B modelling language [19], including modelling concepts. The Event-B language has two main components, which are described in Table 1: *context* and *machine*. A *context* describes the static structure of a system, including *carrier sets* s and *constants* c, as well as *axioms* $A(s, c)$ and *theorems* $T_c(s, c)$ that state their properties. A *machine* defines a system's dynamic structure, which includes *variables* v, *invariants* $I(s, c, v)$, *theorems* $T_m(s, c, v)$, *variants* $V(s, c, v)$, and *events* evt. Terms like *refines, extends*, and *sees* are used to describe the relation between components of Event-B models. In a *machine*, *events* are used to modify state variables by providing appropriate *guards*.

CONTEXT	**MACHINE**
ctxt_id_2	*machine_id_2*
EXTENDS	**REFINES**
ctxt_id_1	*machine_id_1*
SETS	**SEES**
s	*ctxt_id_2*
CONSTANTS	**VARIABLES**
c	v
AXIOMS	**INVARIANTS**
$A(s, c)$	$I(s, c, v)$
THEOREMS	**THEOREMS**
$T_c(s, c)$	$T_m(s, c, v)$
END	**VARIANT**
	$V(s, c, v)$
	EVENTS
	Event evt
	any x
	where $G(s, c, v, x)$
	then
	$v :\mid BA(e)(s, c, v, x, v')$
	end
	END

Table 1: Model structure

Theorems	$A(s, c) \Rightarrow T_c(s, c)$ $A(s, c) \wedge I(s, c, v) \Rightarrow T_m(s, c, v)$
Invariant preservation	$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x) \wedge BAP(s, c, v, x, v')$ $\Rightarrow I(s, c, v')$
Event feasibility	$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x)$ $\Rightarrow \exists v'.BAP(s, c, v, x, v')$
Variant progress	$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x) \wedge BAP(s, c, v, x, v')$ $\Rightarrow V(s, c, v') < V(s, c, v)$

Table 2: Some of the associated proof obligations

3.1.1. Modelling Actions over States

An Event-B model is characterised by a set of state variables that can be modified by a set of events. An invariant $I(s, c, v)$ expresses the required safety properties that the variable v must satisfy during event activation. An event is a state transition in a dynamic system that contains *guard(s)* and *action(s)*. A *guard*, predicate based on state variables, is required for an event to be enabled. A *action*, also known as a before-after predicate, is a generalised substitution that describes how the occurrence of an event changes one or more state variables. An event evt can be defined in three ways: the first is BEGIN $v : |BAP(s, c, v, v')$ END, in which the *action*, $BAP(s, c, v, v')$, is not guarded and is always enabled; the second is WHEN $G(s, c, v)$ THEN $v : |BAP(s, c, v, v'))$ END, in which the *action*, $BAP(s, c, v, v')$, is guarded by $G(s, c, v)$, and the *guard* must be satisfied to enable the *action*; and the final is ANY x WHERE $G(s, c, v, x)$ THEN $v : |BAP(s, c, v, x, v')$ END, where the *action*, $BAP(s, c, v, x, v')$, is guarded by $G(s, c, v, x)$ and depends on the local state variable x to describe non-deterministic events.

The Rodin platform generates proof obligations (POs) [20]. Event-B supports a variety of POs (see Table 2), such as invariant preservation, non-deterministic action feasibility, variant, well-definedness, and so on. Invariant preservation ensures that each invariant is preserved by each event; non-deterministic action feasibility shows the feasibility of the event e with respect to the invariant I; variant ensures that each convergent event decreases the proposed numeric variant; and well-definedness ensures that each axiom, theorem, invariant, guard, action, and variant is well-defined.

3.1.2. Refinement

To model a complex system, the Event-B modelling language supports a stepwise refinement technique. The refinements allow us to gradually model a system and provide a way to strengthen invariants, introducing more detailed system behaviour. By modifying the state description, this refinement method converts an abstract model into a more concrete version. The refinement process adds new events to a list of state variables by refining each abstract event to a concrete version, or by simply refining each abstract event to a concrete version. These refinements preserve the relationship between an abstract model and its concrete model while introducing new events and variables to specify more concrete system behaviour. *Gluing invariants* connect the abstract and concrete state variables. Each abstract event is correctly refined by its concrete version, thanks to the generated POs. Table 3 shows some of the important POs associated to refinement, such as event simulation, guard strengthening and invariant preservation. The simulation ensures that each action in a concrete event simulates the corresponding abstract action; the guard strengthening in a refinement ensures that the concrete guards in the refining event are stronger than the abstract ones; and the invariant preservation ensures that each invariant is preserved by each refined event. These POs contain axiom $A(s, c)$ invariant $I(s, c, v)$, gluing invariant $J(s, c, v, w)$, guard $H(s, c, y, w)$, witness $W(s, c, x, w, y, w')$ and before-after predicate $BAP(s, c, w, y, w')$.

A set of new events introduced in a refinement step is referred to as hidden events because they are not visible to the environment of the system being modelled. These introduced events are beyond the environment's control. *Skip* is refined by new events that are not visible in the abstract model. Any number of executions of an internal action may occur between each execution of a visible action. By strengthening the guards and/or predicates, the refined model reduces the degree

Event simulation	$A(s,c) \wedge I(s,c,v) \wedge \ \wedge J(s,c,v,w) \wedge H(s,c,y,w) \wedge W(s,c,x,w,y,w')$ $\wedge BAP(s,c,w,y,w') \Rightarrow \ BAP(s,c,v,x,v')$
Guard strengthing	$A(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge \ H(s,c,y,w) \wedge W(s,c,x,w,y)$ $\Rightarrow G(s,c,v,x)$
Invariant preservation	$A(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge \ H(s,c,y,w) \wedge W(s,c,y,w,v',w') \wedge$ $BAP(s,c,w,y,w') \Rightarrow \ I(s,c,v',w')$

Table 3: Refinement POs

of nondeterminism. The refinement of an event e by an event f implies that the event f simulates the event e, ensuring that the set of traces of the refined model includes (up to stuttering) the traces of the resulting model.

3.1.3. Rodin

Rodin [20] is an open source Eclipse-based integrated development environment (IDE) for developing Event-B models. Rodin is a foundational collection of plug-ins for project management, formal development, syntactic analysis, proof assistance, and proof-based verification. Furthermore, it supports extension points for a variety of additional plugins that provide various functionalities and features related to model checking, animation, code generation, additional proof capabilities using SMTs and external theorem provers (i.e., Why3, Isabelle), UML-B, Theory plug-ins, composition and decomposition, refactoring framework, and model editors.

3.1.4. Applications

The Event-B method has been used successfully to design critical systems for a variety of applications, including the control system for the Meteor line 14 in Paris and the VAL shuttle for Paris CDG airport [21, 22, 23], as well as medical devices [24, 25, 26], autonomous systems [27, 28], security protocols [29, 30], control-command systems [31, 32, 33], and distributed protocols [34, 35, 36, 37, 38]. More information can be found in [39].

3.2. Solidity

Solidity is a programming language designed specifically for developing smart contracts that compiled into byte-code executable by Ethereum platform's execution engine, often known as Ethereum Virtual Machine (EVM). Smart Contracts are programs that execute on a decentralized network without the intervention of a central authority. Solidity enables developers to create self-enforcing business logic via smart contracts, resulting in a trustworthy and authoritative record of transactions. The syntax of Solidity is quite similar to those of scripting languages like JavaScript, and it is heavily influenced by C++, Python. Solidity extensively utilizes programming techniques derived from other languages. It features variables, static typing, functions, libraries, and interfaces. In addition, it offers a range of control structures such as for, while, do-while, and if-else. In case of an object-oriented programming language like Java, programmers work with classes, whereas Solidity programmers deals with contracts. Each contract can define various Solidity constructs such as state variables, functions, function modifiers, events, errors, structure types, and enum types. This subsection outlines the major Solidity constructs that are relevant to our goal. More information about Solidity programming language can be found in [2].

3.2.1. Solidity Types, Special Functions and Variables

Solidity is a statically typed programming language, which means that variable types are declared explicitly and thus determined at compile time. In Solidity, numerous elementary types exist that can be combined to form more sophisticated types. Solidity offers an extensive range of types, notably value types, reference types such as arrays and structures, mapping types, user-defined types and it also supports elementary type conversion. Solidity does not support undefined or null values, newly declared variables always have a default value based on their type. Table 4 summarises the most frequently used value types.

Value Types	**Keyword**	**Description**
Boolean Type	$bool$	possible values $true$ and $false$
Integer Types	int**X**	signed integers where **X** varies from 8 to 256 in steps of 8
	$uint$**X**	unsigned integers where **X** varies from 8 to 256 in steps of 8
Address Types	$address$	Ethereum address of 20 bytes
	$address\ payable$	additional members like $transfer$ and $send$
Byte Arrays (Fixed Size)	$bytes$**X**	**X** varies from 1 to 32
Byte Arrays (Dynamic Size)	$bytes$	similar to $bytes1[]$ but skips padding
	$string$	similar to $bytes$, but don't allow $length$ or index access
Enumerated Type	$enum$	default value is the first member

Table 4: Solidity Value Types

In the global namespace, special variables and functions exist at all times and are primarily used to relay information about the blockchain or to perform general-purpose utility operations. Table 5 lists some of the most frequently used variables and functions from the global namespace of Solidity. Additionally, solidity supports various denomination of Ethereum cryptocurrency such as wei, $gwei$, and $ether$, as a suffix to number literals.

Type	**Variable/Function**	**Description**
$address$	$msg.sender$	sender of the message (current call)
$uint$	$msg.value$	number of wei sent with the message
-	$assert(bool)$	used for internal errors
-	$require(bool, [message])$	used for checking condition on input
-	$revert([message])$	abort execution and revert state changes
$uint$	$< address > .balance$	balance of the $address$ in wei
-	$< address\ payable > .transfer(uint)$	send given amount of wei to Address
$bool$	$< address\ payable > .send(uint)$	send wei to $address$, returns $false$ on failure
Contract	$this$	refers to the current contract
$uint$	now	current block timestamp
$address\ payable$	$tx.origin$	sender of the transaction

Table 5: Frequently Used Special Variables and Functions

Errors are handled in Solidity using state-reverting exceptions. This exception nullifies any state changes made during the execution of the code and notifies the caller of an error. The assert and require functions enable programmers to check for conditions and throw exceptions if they are not met. The assert function should be preferred exclusively to check for internal errors and invariants. The require function supports optional error message and should be used to ensure the existence of valid conditions that must be identified during contract execution. This includes conditions on input values or the return values of external contract calls. The revert function is another mechanism available in the global namespace for reporting errors and undoing state

changes during contract code execution. Additionally, this function accepts and returns an optional message containing details about the error to the caller.

3.2.2. Function Modifiers

Modifiers in Solidity are analogous to the Object-Oriented Programming decorator patterns. A modifier controls how a function behaves at run-time. In the example provided in Code Snippet 1, the modifier $validate(int)$ restricts the execution of the $increment$ function if the parameter value of increment function is less than one.

```
pragma solidity >=0.6.0;

contract ModifierExample {

    int public number;

    modifier validate(int value){
        require(value>0, "value␣of␣incrementBy␣must␣be␣greater␣than␣0");
        _;
    }

    function increment(int incrementBy) validate(incrementBy) public{
        number += incrementBy;
    }

}
```

Code Snippet 1: Modifier Example

Multiple modifiers can be applied to a function or constructor by specifying them in a whitespace-separated list and are evaluated from left to right. Modifiers are inheritable properties of contracts and may be overridden by derived contracts. The symbol $'\_;'$ in modifier body returns the flow of execution to the original function code. It applies to various contexts, such as providing an easy-to-understand approach to express certain guards, confirming specific conditions after the function execution.

4. Formal framework for Solidity smart contracts

Fig. 1 depicts a modelling framework for formalising and implementing Solidity smart contracts that support the *correct by construction* approach. In this proposed framework, there are three main important process blocks: (1) Event-B model, (2) formal verification, and (3) Solidity implementation. The Event-B model block deals with complex modelling of smart contracts, and building a formal model of smart contracts may require several iterations. Starting from an initial model, it is gradually enhanced by adding design decisions and handling requirements. Note that the progressive safe enrichment allows to add the required safety properties and low level system requirements related to smart contracts. These refinements preserve the relationships between an abstract model and its corresponding concrete model while introducing low-level details and new properties to specify more concrete behaviour of a system. This incremental development ensures the correctness of concrete behaviour of a smart contract system in relation to the abstract model.

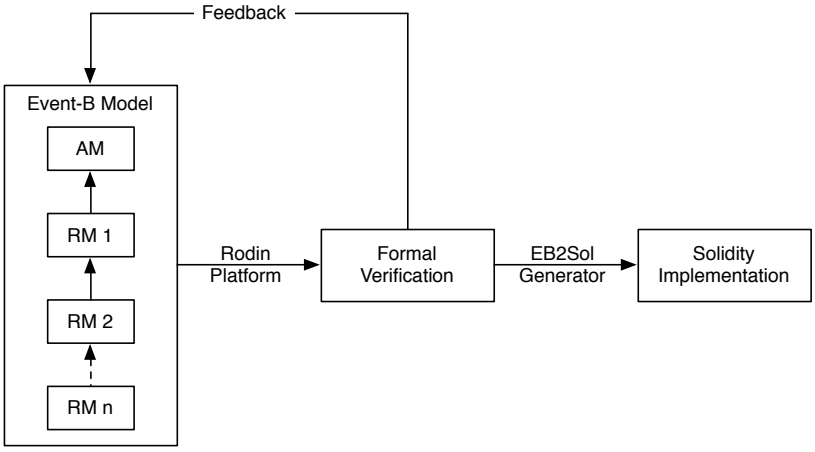

Figure 1: A framework for developing formal Solidity smart contracts.

Rodin tools are used to verify the abstract and refined Event-B models. The analysed models provide feedback to the original Event-B model, which is depicted in Fig. 1.

The second block of Fig. 1 is concerned with formal verification, which is an important step in proving and disproving the correctness of a formalized system's intended behavior under the given safety properties and assumptions. In the process of system verification, we can use an automated theorem prover to check the consistency and correctness of the generated formal model. Using mathematical operators, a formal notation can be analyzed and manipulated, and mathematical proof procedures can be used to test (and prove) the internal consistency (including data conservation) and syntactical correctness of the specifications. The formal verification ensures that the model is correctly designed, that the developed formal specification has the required properties, and that the generated models are free of errors, oversights, and bugs. The Rodin development framework can be used for project management as well as model verification via syntactical and refinement checking. Furthermore, this tool can be used to discharge the generated proof obligations. If any inconsistencies in the model are discovered during the system verification process, we can modify the model obtained in the previous step of the development process. We can iterate until we get a correct model.

The final block in our framework is related to Solidity implementation using our developed tool EB2Sol. This is the last stage of our framework proposal. From the formalised, proved, and validated formal models, it generates Solidity language code for system implementation. This phase of development includes the major executable components, concrete data structure definitions, and some auxiliary structures or functions that may be assumed in a design. However, it is also critical to rigorously review the system implementation in relation to the specified system requirements and selected design. Automatic code generation has become standard practice in developing verified code in a target language corresponding to the given specification, required configurations, and platform information without any human intervention in the new age of software development and automation of the development life-cycle.

In our proposed framework, we use our developed tool EB2Sol, an extension of EB2ALL [40, 41], to generate source code in Soldity language from the verified formal specification. Section 5

describes the basic architecture of this tool as well as the development process. This automation process has several advantages. For example, the automatic code generation tool produces executable software with far fewer implementation errors than a human programmer. Manually implementing the formal specification can be time-consuming and error-prone. In automatic code generation, code maintenance and successive changes are simple to handle, and the consistency between the generated code and the given formal specification is always maintained. However, there are some drawbacks to automatic code generation. For example, automatically generated codes can be difficult to understand, excessive code can be generated, and incorrect code is still a possibility.

5. EB2Sol: Event-B to Solidity

In this section, we describe the development architecture for our developed tool EB2Sol, which is an extension of EB2ALL [41], developed as a new plugin for generating Solidity smart contracts from Event-B models. Fig. 2 depicts the overall architecture of the EB2Sol automatic code generation tool. The given boxes show different steps of the code generation process. The first block is associated with the Event-B model, which serves as an input to the code generation tool.

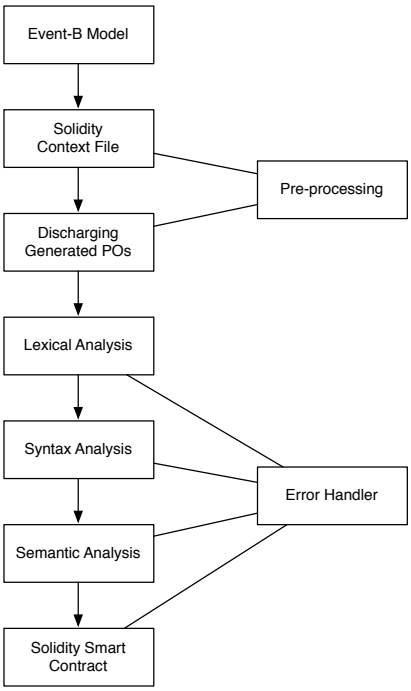

Figure 2: Development architecture of EB2Sol

5.1. Preprocessing and generated POs

Most of the time, systems fail due to a run-time error. Overflow and underflow of bounded integers, for example, are types of run-time errors that should be checked before producing the Solidity code. There is a pre-processing step in the code generation process that allows for the

Event-B type	Formal Range	Solidity type
tl_int8	$-2^7..2^7-1$	int8
tl_int16	$-2^{15}..2^{15}-1$	int16
tl_int24	$-2^{23}..2^{23}-1$	int24
...	...	...
tl_int256	$-2^{255}..2^{255}-1$	int256
tl_uint8	$0..2^8-1$	uint8
tl_uint16	$0..2^{16}-1$	uint16
tl_uint24	$0..2^{24}-1$	uint24
...	...	...
tl_int256	$0..2^{256}-1$	uint256

Table 6: Signed and Unsigned Integers

introduction of a context file based on the Solidity language to provide a deterministic range for data types in order to make the Event-B model deterministic [42]. To obtain the deterministic model, we can use vertical refinement to refine the previous concrete model, which provides deterministic definitions of constants and variables. For this, we can introduce a *Solidity context file* which contains bounded integer data types. Table 6 shows a map between Event-B and Solidity types for signed and unsigned integers. Note that the Solidity context file is an Event-B context file, which is required to generate correct code. Adding a new context file may generate new set of POs that must be proved before generating the Solidity code as well as to verify the specification in order to ensure the system's consistency.

5.2. Translation of Event-B to Solidity

The main objective is to translate the Event-B model into a *semantically observationally equivalent* standard Solidity smart contracts. The developed translator is based on a set of transformation rules that map between the Event-B and Solidity constructs. In fact, the transformation functions allow to produce code in target language applying a set of rules when matching the inputs from the source language. We define a set of transformation functions to generate Solidity code from the Event-B model. These transformation rules are given in Table 7.

T_{decl}	:	Datatype and Constant declaration translation function
T_{st}	:	State variables translation function
T_{axm}	:	Axioms translation function
T_{pred}	:	Predicate translation function
T_{thm}	:	Theorem translation function
T_{evt}	:	Events translation function
T_{grd}	:	Event's guards translation function
T_{act}	:	Event's actions translation function
T_{exp}	:	Expressions translation function

Table 7: Transformation function

A set of supported symbols of EB2Sol tool is given in Table-8. This table shows a subset of Event-B syntax to equivalent Solidity smart contracts. All constants defined in a model's context must be replaced with their literal values. We consider Event-B formal notations available at [5]

and capture the Event-B grammar in a Abstract Syntax Tree (AST). This translation tool accepts conditional, arithmetic, and logical formal model expressions. Event-B model that contains machines and contexts are translated to Solidity contracts. Below we provide the translation process for dealing with context and machine models.

Event-B	Solidity	Comment
const_x $\in \mathbb{N} \wedge$ cons_x=120	int256 constant const_x = 120	Constant declaration
x $\in \mathbb{Z}$	int256 x	Signed integer variable declaration
x $\in \mathbb{N}$	uint256 x	Unsigned integer variable declaration
x $\in$ tl_int16	int16 x;	Signed integer variable declaration
b $\in$ BOOL	bool b;	Boolean variable declaration
x $\in$ n..m $\rightarrow \mathbb{Z}$	int [m+1] x;	Array declaration
x = y	if(x==y) { ... }	Conditional statement
x $\neq$ y	if(x!=y) { ... }	Conditional statement
x < y	if(x<y) { ... }	Conditional statement
x $\leq$ y	if(x<=y) { ... }	Conditional statement
x > y	if(x>y) { ... }	Conditional statement
x $\geq$ y	if(x>=y) { ... }	Conditional statement
(x>y) $\wedge$ (x$\geq$z)	if ((x>y) && (x>=z) { ... }	Conditional statement
(x>y) $\vee$ (x$\geq$z)	if ((x>y) || (x>=z) { ... }	Conditional statement
$\neg$x<y	if(!(x<y)){ ... }	Logical not
X$\Rightarrow$Y	if(!X || Y){ ... }	Logical Implication
X$\Leftrightarrow$Y	if((!X || Y) && (!Y || X)){ ... }	Logical Equivalence
x := y + z	x = y + z;	Arithmetic assignment
x := y - z	x = y - z;	Arithmetic assignment
x := y * z	x = y * z;	Arithmetic assignment
x := y $\div$ z	x = y / z;	Arithmetic assignment
x := a(y)	x = a[y];	Array assignment
x := y	x = y;	Scalar action
a := a $\Leftarrow$ {x$\mapsto$y}	a[x] = y;	Array action
a := a $\Leftarrow$ {x$\mapsto$y} $\Leftarrow$ {i$\mapsto$j}	a[x]=y; a[i]=j;	Array action
fun $\in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$	function fun_name(uint256 $arg1$, uint256 $arg2$) public returns(arg) { ... }	Function definition

Table 8: Event-B to Solidity

5.2.1. Context models

The context of an Event-B model consists of *sets*, *enumerated sets*, *constants*, *arrays* and *functions*, all of which are associated with their respective type. For translation purposes, the translation tool supports all types of context components. The observational equivalence is based on the equivalence of Event-B values and the values of Solidity smart contracts. This equivalence on values is naturally extended to context instances. In Table-9, the observational equivalence between Event-B sets and Solidity smart contracts types is given.

Event-B types	Solidity language
Enumerated sets	Enumerated types
Basic integer sets	Predefined integer types
Event-B array types	Solidity array type
Function	Solidity function structure

Table 9: Equivalence between Event-B and Solidity smart contracts

Constants, sets and enumerated sets of the Event-B model are translated into constants, type declaration and enumerated sets of the Solidity contracts using the translation function T_{decl} and

T_{axm}. An Event-B enumerated sets are semantically equivalent to Solidity contracts enumerated types, thus it is simple to translate.

For the efficiency of the generated code and the correctness of the translation, the link between Event-B and Solidity contracts for integer values have been regarded significant. We recommend to use preprocessing step by introducing Solidity context file. Similarly, the Event-B constant are also directly translatable due to direct typing correspondence between Event-B integer types and Solidity language integer types. For example, if Event-B use a data type given in Table 6.

The translation for the declaration of Event-B array type and the Solidity array type is not straightforward. In Event-B, an array can be defined as a total function, whereas in Solidity, they relate to a contiguous memory zone (coded as the beginning address of the array and its size). The semantical correspondence between an array element *arr(i)* in Event-B and the value at the position *arr[i]* in Solidity, can be easily translate.

The translation for the Event-B function into Solidity function is also very complex. Only the Event-B total function is supported by the current prototype tool. Solidity function input and output arguments can be easily identified by looking at the left and right sides of the total function symbol ($\rightarrow$) in the Event-B function specification. The Event-B function definition can be translated in a Solidity function structure.

The context model elements are declared global in the generated code. The type information for context elements is derived from the context axioms used for type definition, such as it can be used to express as integer ranges, specifically supported bit-map types, or arrays of the defined mapping functions.

5.2.2. Machine models

A machine model consists of *variables*, *invariants*, *events*. The Event-B variables can be used to generate Solidity attributes, and the Event-B invariants can be used to extract typing information. All the generated variables or attributes have default property *public*. Note that the Event-B machine may also contain function and array declaration that can also translate similarly to translation rule for function and array given in context model. The required typing information can be extracted from the Event-B invariants.

There are two types of variables in the Event-B specification: global variables and local variables. Global variables are produced directly from the variable declarations, and all of these variables have global scope. Local variables are derived from any clause of an event and are completely local to the corresponding event. All local variable declarations are placed as a list of input arguments of the function when the function structure is generated.

The translation tool uses a recursive method to generate Solidity contracts for each event of the Event-B specification. The translation tool always checks for the *'null'* event (i.e. the guard of a false condition), never generates the source code for such event, and inserts a relevant note into the generated code for traceability. For example, if an event has a single guard with a $false$ condition, the Solidity code does not produce for that event. This automatic reduction occurs to avoid the production of inaccessible run-time code.

The initialization event of Event-B machine is translated as a constructor, and all variables are initialized with default values in the constructor body which are directly derived from the action predicates of the Event-B initialization event.

Guard handling in Event-B is extremely ambiguous due to different meanings, such as local variable type definition, assignment of a value to a local variable, condition statements using negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), and equivalence ($\Leftrightarrow$) operators. We build a recursive technique for parsing and identifying different elements of the Event-B guard for translation purposes. For example, an implication ($\Rightarrow$) and equivalence ($\Leftrightarrow$) operator, the translator tool automatically rewrites the predicate in an equivalent form using conjunction ($land$), disjunction ($\vee$), and negation ($\neg$) operators, an equal relation may signify an assignment or equality comparison, and the precise meaning (and thus the resulting translation) deduced from the type and scope of implication ($\Rightarrow$).

Another interesting point to discuss is the definition of a functional-image relation, which can be used to represent a data array or an external function. Once the guards of an event have been classified, the guards that confer local variable type information are utilized to generate variable declarations in the function, while the remaining guards are used to generate local assignment and conditional statements. In addition, local variable type information is derived in the same way as global variables from guard information.

Event-B events are translated into functions, and event parameters are passed as function and modifier arguments. In order to effectively call each function, modifiers are added to each generated function. These associated modifiers are also defined as a function using required/assert statements derived from Event-B guard predicates.

Actions are triggered concurrently in Event-B, and any state modification in the actions is only valid in the whole event post-condition. As a result, dependency checks must be conducted to guarantee that no state variable used as an action assignee has been updated to its post-condition before to usage. As a guard statement, a similar kind of parsing is used on the Event-B action statement. At last, the event actions are directly translated in the form of assignment statements in the function body. Assignments to scalar variables, override statements acting on array-type variables, and arithmetic complicated expressions are all supported through an action translation.

6. Case Study

To assess the proposed methodology and developed tool EB2Sol, we use several small and large examples. For example, we developed the formal specification of arithmetic operations (increment and decrement), cardiac pacemaker and smart purchase. In this section, we apply our proposed methodology for developing smart purchase case study, and discuss its formal development, including code generation in Solidity.

6.1. Informal description of smart purchase

Purchasing goods over the internet necessitates the trust of several parties. A basic example with seller and buyer is available in [2]. However, it becomes more complicated when a courier is included in the system, as it is challenging to detect an item's delivery to its intended recipient. Fig. 3 illustrates a process of purchasing goods. In this example, we have included a seller, courier, and buyer with the ability to return the product within a return window. The seller can execute actions such as listing an item for sale, revoking it, processing an order, and initiating settlement. On the other hand, the buyer can purchase the item, confirm delivery, and request a return. Services like pickup and pickup return are available for a courier.

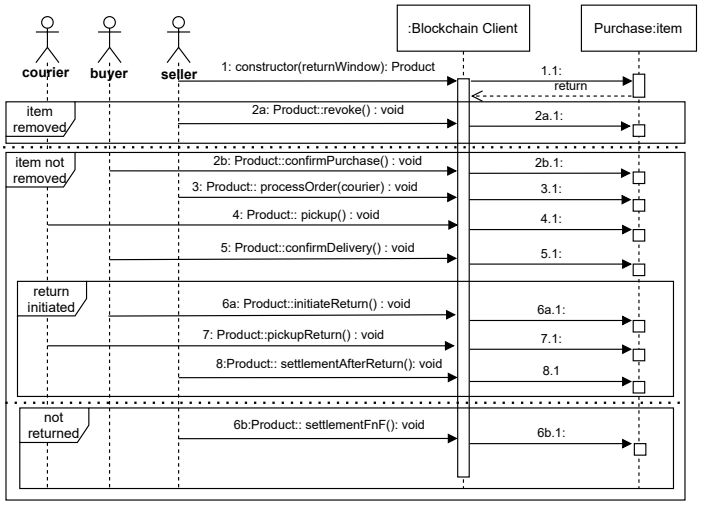

Figure 3: Smart Purchase Example

6.2. Formal Development

For developing a formal model of the smart purchase, we develop a context model to express static behaviour, properties of the system, while the dynamic behaviour and properties are modelled in a machine model. Below we describe both the context and machine models.

6.2.1. Context model

In the context model of smart purchase, we declare a set $ADDRESS$ to represent address type that is unique in Solidity contracts. An enumerated set $ProductState$ is defined to encode all possible states of different processes in $axm1$. A new constant $\_timeInSec$ is defined in $axm2$ to represent time duration. The next two axioms ($axm3$ and $axm4$) are used to define $balance$ and now constants to represent initial balance and time value. The last axiom defines a constant $this$ as an address type to indicate one's own address.

$$
\begin{aligned}
&axm1 : partition(ProductState, \{InStock\}, \{Booked\}, \{Packed\}, \{Transite\}, \\
&\qquad \{ReturnReq\}, \{ReturnAck\}, \{Delivered\}, \{Closed\}) \\
&axm2 : \_timeInSec \in \mathbb{N} \\
&axm3 : balance \in \mathbb{N}_1 \\
&axm4 : now \in \mathbb{N} \\
&axm5 : this \in ADDRESS
\end{aligned}
$$

6.2.2. Machine model

A machine model is developed to specify the dynamic behaviour of smart purchase including all possible operations. In this model, we introduce 8 state variables ($inv1$-$inv8$). A variable $address\_list$ is defined to record all the addresses who call the smart purchase contract. The next three variables $seller$, $buyer$ and $courier$ are defined as address types belonging to $address\_list$. A transfer function is defined in $inv5$. The variable $state$ represents possible product states when purchasing goods. The next variable $price$ is used to define the product's cost and the last variable $returnWindow$ is used to define product's return time duration.

$inv1 : address\_list \subseteq ADDRESS$
$inv2 : seller \in address\_list$
$inv3 : buyer \in address\_list$
$inv4 : courier \in address\_list$
$inv5 : transfer \in ADDRESS \pfun \mathbb{N}$
$inv6 : state \in ProductState$
$inv7 : price \in \mathbb{N}$
$inv8 : returnWindow \in \mathbb{N}$

In this machine, we introduce 12 events, including INITIALISATION event. The INITIALISATION event is used to set the initial value for each declared variable.

EVENT INITIALISATION
BEGIN
act1: $address\_list := \{this\}$
act2: $seller := this$
act3: $buyer := this$
act4: $courier := this$
act5: $transfer := \varnothing$
act6: $state := InStock$
act7: $price := 0$
act8: $returnWindow := \_timeInSec$
END

Other events are: $initiateSeller$- to initiate seller of contract; $revoke$- to revoke the product and reclaim the deposited ethers; $confirmPurchase$- to confirm the purchase; $processOrder$- to be called by the seller after the product has been booked; $pickup$- to be called by the courier at the time of pickup; $confirmDelivery$- to confirm delivery of the item; $initiateReturn$- to initiate the process of product return; $pickupReturnedProduct$- to pay the product price to the seller; $FnFSettlement - AfterDelivery$- to confirm the buyer's full and final settlement for the purchased item; $FnFSettlementAfterReturnAccepted$- to confirm full and final settlement for the purchased item when the buyer returns the purchased item; and $new\_account$- to add a new address that invokes the smart purchase contract.

In the event $initiateSeller$, we define three parameters and five guards. The first two guards define the local parameters msg_value and msg_sender. The following guard defines $value$ and ensures that it equals $msg\_value/3$. The $grd4$ states that msg_value is three times $value$, and the last guard states that $seller$ is not equal to msg_sender. These guards keep the required condition for initiating seller. In this event, two new actions are introduced, and these actions are used to initiate the $seller$ and $price$ variables based on the selected address msg_sender and msg_value, respectively.

EVENT initiateSeller
ANY $value, msg\_value, msg\_sender$
WHERE
grd1: $msg\_value \in \mathbb{N}_1$
grd2: $msg\_sender \in address\_list \setminus \{this\}$
grd3: $value \in \mathbb{N} \land value = msg\_value/3$
grd4: $msg\_value = 3 * value$
grd5: $seller \neq msg\_sender$
THEN
act1: $seller := msg\_sender$
act2: $price := msg\_value/3$
END

Similarly, another event $confirmPurchase$ is defined to confirm the purchase, which can only be called by the buyer if the product is in stock. In this event, we define two parameters and four guards. The first two guards define the local parameters msg_value and msg_sender. The next guard ($grd3$) states that the $state$ is $InStock$. The last guard shows that the msg_value is three times the $price$. These guards keep the required condition for confirming purchase. Two new actions are introduced in this event, and these actions are used to set the $state$ as $Booked$ and the $buyer$ is updated by msg_sender.

EVENT confirmPurchase
ANY msg_value, msg_sender
WHERE
grd1: $msg\_value \in \mathbb{N}_1$
grd2: $msg\_sender \in address\_list \setminus \{this\}$
grd3: $state = InStock$
grd4: $msg\_value = 3 * price$
THEN
act1: $state := Booked$
act2: $buyer := msg\_sender$
END

Other remaining events are formalised in a similar way. A complete formal development of the smart purchase case study is available at [1].

6.2.3. Model validation and analysis

Formal modelling and verification are carried out using the open source integrated development framework Rodin. In this section, we summarise the generated proof obligations generated by the Rodin prover. The generated proof obligations are related to well-definedness, feasibility and invariant. The generated proof obligations ensure consistency checking and it guarantees that the modelled events always preserve the given invariants. The formal development of smart purchase is presented in a single machine, but it can be developed incrementally through an abstract model and a series of refinement models. This development results in 22 proof obligations (POs), in

[1] `http://singh.perso.enseeiht.fr/BAC-2021/SmartPurchaseModel.pdf`

which 19 (87%) are automatically proved and the remaining 3 (13%) are proved interactively using the Rodin prover and other associated tools, such as SMT solvers (CVC4, Z3, and veriT). By achieving the required functional behavior for the smart purchase model, we have successfully discharged all of the generated proof obligations that ensure consistency checking.

The model analysis is carried out with the help of the ProB [43] model checker, which can be used to explore traces of Event-B models. The ProB tool can detect potential deadlocks and supports *automated consistency checking* and *constraint-based checking*. In this work, we use the ProB tool as a model checker to show the absence of errors (no counterexample exists) and the absence of deadlock. It is worth noting that the ProB employs all of the described safety properties during the model checking process in order to report any violations of safety properties against the formalized system behavior. We also use the ProB tool for animating the models to validate the developed smart purchase model. This validation approach entails gaining confidence that the developed models are in accordance with the requirements.

6.3. Code Generation in Solidity

In this section, we will use our developed tool EB2Sol to generate Solidity smart contracts from the formal Event-B model of smart purchase. EB2Sol is an Eclipse-based plug-in for code generation in the Solidity language for the Rodin platform. A detailed description of the automatic code generation is given in section 5.

A screen shot of the EB2Sol in the Rodin environment is shown in Fig. 4. After installing this plug-in successfully, the menu *Translator/EB2Sol* and a tool button on the toolbar will appear. To generate Solidity source code for any formal model, a user can select it from the EB2Sol menu or tool button, and a dialog box will appear. This dialog box displays a list of currently active projects. Any project can be chosen by the user to generate Solidity contracts, including a log file containing information about the code generation process.

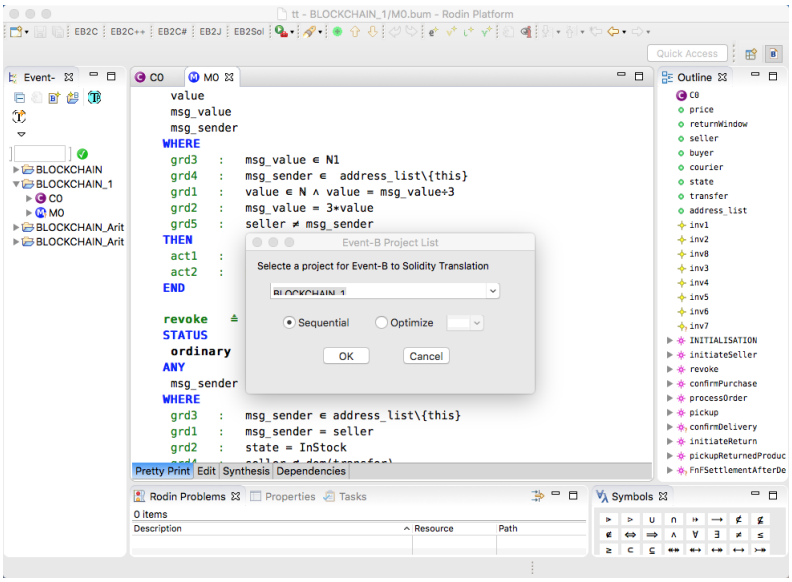

Figure 4: EB2Sol plug-in in the Rodin IDE

In our case study, we generate Solidity source code from the proven smart purchase model using the EB2Sol plug-in. Before using this tool, we refine our concrete model by introducing a new context containing Solidity type definitions (see Table 6) and removing the abstract operations and non-supported symbols via data refinement. By defining some glueing invariants, this refinement makes the smart purchase model deterministic. All new generated POs must be discharged before generating smart contracts.

Smart contract generation from an Event-B model is straightforward. The EB2Sol tool generates Solidity smart contract files from the concrete model. Constants, type definitions, variables, modifiers, and functions are all included in the generated smart contracts that are extracted from the smart purchase model. The translated constants, type definitions, and variables are taken from the generated code shown in code snippet 2.

```
 enum ProductState {InStock,Booked,Packed,Transite,ReturnReq,
 ReturnAck,Delivered,Closed} // Enumerated  definition

 int256 constant _timeInSec = 10;
//Solidity datatype declaration when constant is given
 int256 constant balance = 500;
//Solidity datatype declaration when constant is given
 ...
 ...
 uint256 price;
//Solidity datatype declaration when variable is given
 uint256 returnWindow;
//Solidity datatype declaration when variable is given
 ...
```

Code Snippet 2: Generated type declaration

The formalised smart purchase model yields a set of functions. These functions are generated from events by analyzing various elements such as local parameters, guards, and actions. Event-B events are converted into functions, and event parameters are used as function arguments. Modifiers are added to each generated function in order to effectively call it. These associated modifiers are defined as a function as well, using required/assert statements derived from Event-B guard predicates.

The event actions are directly translated equivalent to Solidity assignment expressions. To execute a set of actions in any function, all of the given modifiers must be TRUE. If the given modifiers do not satisfy, the function body statements are skipped. The only excerpt from the generated code equivalent to the given events is given in code snippet 3.

```
...
modifier MOD_initiateSeller {
        uint256 value = msg.value/3;
        require((msg.value==3*value)&&(seller!=msg.sender));
        _;
}
modifier MOD_confirmPurchase{
        require((state==InStock)&&(msg.value==3*price));
        _;
}
```

```
...
...
function initiateSeller() MOD_initiateSeller public{
    // Actions
    seller = msg.sender;
    price = msg.value / 3;
    returnWindow = _timeInSec;
}
function confirmPurchase() MOD_confirmPurchase public{
    // Actions
    state = ProductState.Booked;
    buyer = msg.sender;
}
...
```

Code Snippet 3: Generated modifiers and functions

7. Discussion

This chapter has presented a framework for analysing, verifying and implementing the Solidity smart contracts based on formal methods. Furthermore, the case study that we have presented demonstrated the viability of our framework. This section discusses the main benefits and limitations of our approach.

7.1. Benefits

Our proposed framework has the potential to improve blockchain system safety and quality, including assisting in the system certification process for developing smart Solidity contracts. To support the proposed framework, we use the Event-B modelling language to specify smart contracts requirements. The main advantages of this approach are that it improves the process of abstractly specifying smart contract requirements, formal development using correct by construction, and detecting potential flaws through rigorous analysis. This may enable us to analyse and validate the important properties of consistency, deadlock-freedom and safety of complex transections. The main advantages of our approach are described in the following section.

7.1.1. Progressive development of smart contracts

For a long time, stepwise refinement has been used to model complex systems. Our proposed framework adopted the key concepts of refinement for designing complex smart contracts. This gradual development enables us to introduce concrete details and the required safety properties associated with blockchain technology. The use of refinement reduces the complexities of design and proof efforts by increasing proof automation.

7.1.2. Improving smart contract error detection

Our proposed framework included modelling and verification approaches for rigorous analysis to ensure the functional behaviour of the designed smart contracts, as well as their refinement steps. We can detect inconsistencies in the given requirements and refinement relations using formal verification, where we must discharge all generated proof obligations. In addition, we may

use the ProB model checker to validate the modelled contracts and ensure the absence of any counter examples.

7.1.3. Minimising smart contract development cost

Errors are always less expensive to identify earlier in the development phase. Our framework enables to analyse complex operations in order to define smart contracts. The use of formal reasoning in the design of complex smart contacts always results in a lower overall cost. Moreover, the use of design patterns and refinement can be applied to the development of other similar smart contracts.

7.2. Limitations

The proposed framework consists of formal modelling, verification and implementation. We used the Event-B modelling language for modelling, Rodin tools for verification, and EB2Sol for generating Solidity smart contacts. Here, we will go over some of the limitations of our approach and the tools we have chosen.

7.2.1. Need careful analysis in modelling and implementation

We have used Rodin tools for verification and EB2Sol code generation in our work. In fact, we must ensure the consistency of the designed smart contracts as well as the generation of the Solidity smart contracts, as we use Solidity context file to derive a deterministic model manually. All newly generated POs must be discharged before code generation. Currently, a user may need to provide a Solidity context file and refined the model, including proof process, but this can be automated in the future.

7.2.2. Need powerful theorem provers for smart contracts

In the current work, we discharged the generated POs using Rodin-supported proof tools. Most proofs can be discharged automatically, but human interaction is inevitable when proofs are complex. We need powerful theorem provers to automate the proof process when dealing with complex and large smart contract requirements.

7.2.3. No standard in the generated smart contracts

The Solidity smart contracts must be expressive and optimised in order to use the least amount of gas. Currently, the prototype tool EB2Sol does not take into account the required standard for producing Solidity smart contracts. We need to improve our developed tool so that it can meet the required standard for reducing gas consumption. Otherwise, Solidity smart contracts can be generated manually from the last deterministic model in accordance with the preferred standards.

8. Conclusion

In this chapter, we presented a framework for analysing, verifying and implementing Solidity smart contracts using the correct by construction approach in Event-B. The framework relies on incremental development, which allows for the abstract modelling of Solidity contracts and then further it can be refined to introduce concrete details, including the required safety and security

properties. The refinement enables for the gradual modelling of complex transaction scenarios, and the required properties can be added in a step-by-step approach.

For the implementation of Solidity smart contracts, we developed the prototype tool EB2Sol, which generates source code automatically from verified and proven Event-B models. EB2Sol is a new plugin for the Rodin platform for generating Solidity contracts. Development architecture and a set of transformation functions are defined. The translation tool has been meticulously developed with safety properties preservation. The results detailed in this chapte are an architecture of the translation process, to generate Solidity code from Event-B models following syntax directed translations. Despite its limited syntax, it already covers the most of numerical applications and supports powerful static-analysis methods. The current prototype tool has covered a subset of the Event-B modelling language in order to generate Solidity subset.

We used a case study to better demonstrate the practicability of our proposed framework from abstract modelling to implementation. The selected case study is successfully developed and proven in the Rodin platform, and finally Solidity smart contracts are produced using our tool EB2Sol.

Our current approach is based on first-order logic and refinement in order to develop a safe and secure Solidity contracts that allow us to verify complex properties. Our developed tool currently supports a subset of the Event-B modelling language. Due to the richness of the Event-B modelling language, we intend to extend the current subset of Event-B modelling language for EB2Sol in order to provide modeling and reasoning flexility as well as the ability to generate Solidity contracts from any refinement level. Furthermore, we intend to expand our EB2Sol tool to cover the remaining Solidity language constructs. Another important future task will be to validate the defined translation principals for EB2Sol. We plan to develop a meta model in HOL to check the correctness of translation rules. It may aid to certify the generated Solidity smart contracts.

[1] G. Wood, Ethereum: A secure decentralised generalised transaction ledger EIP-150 REVISION, http://gavwood.com/paper.pdf (2014).

[2] Solidity Documentation, https://solidity.readthedocs.io.

[3] Solidity Github, https://github.com/ethereum/solidity.

[4] X. Zhao, Z. Chen, X. Chen, Y. Wang, C. Tang, The DAO attack paradoxes in propositional logic, in: 2017 4th International Conference on Systems and Informatics (ICSAI), 2017, pp. 1743–1746. doi:10.1109/ICSAI.2017.8248566.

[5] J.-R. Abrial, Modeling in Event-B: System and Software Engineering, 1st Edition, Cambridge University Press, New York, NY, USA, 2010.

[6] Project RODIN, Rigorous Open Development Environment for Complex Systems, http://rodin-b-sharp.sourceforge.net/ (2004).

[7] 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, Berlin, Heidelberg, 2017, p. 164?186. doi:10.1007/978-3-662-54455-6_8.
URL https://doi.org/10.1007/978-3-662-54455-6_8

[8] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, Z. Li, A survey of smart contract formal specification and verification, ACM Comput. Surv. 54 (7). doi:10.1145/3464421.
URL https://doi.org/10.1145/3464421

[9] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, G. Rosu, Kevm: A complete formal semantics of the ethereum virtual machine, in: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), 2018, pp. 204–217. doi:10.1109/CSF.2018.00022.

[10] Y. Hirai, Defining the ethereum virtual machine for interactive theorem provers, in: M. Brenner, K. Rohloff,

J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, M. Jakobsson (Eds.), Financial Cryptography and Data Security, Springer International Publishing, Cham, 2017, pp. 520–535.

[11] D. Annenkov, M. Milo, J. B. Nielsen, B. Spitters, Extracting smart contracts tested and verified in coq, in: Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 105?121. doi:10.1145/3437992.3439934.
URL https://doi.org/10.1145/3437992.3439934

[12] T. C. Le, L. Xu, L. Chen, W. Shi, Proving conditional termination for smart contracts, in: Proceedings of the 2nd ACM Workshop on Blockchains, Cryptocurrencies, and Contracts, BCC '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 57?59. doi:10.1145/3205230.3205239.
URL https://doi.org/10.1145/3205230.3205239

[13] S. K. Lahiri, S. Chen, Y. Wang, I. Dillig, Formal specification and verification of smart contracts for azure blockchain, CoRR abs/1812.08829. arXiv:1812.08829.
URL http://arxiv.org/abs/1812.08829

[14] L. Alt, C. Reitwiessner, Smt-based verification of solidity smart contracts, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice, Springer International Publishing, Cham, 2018, pp. 376–388.

[15] 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, Association for Computing Machinery, New York, NY, USA, 2016, p. 91?96. doi:10.1145/2993600.2993611.
URL https://doi.org/10.1145/2993600.2993611

[16] I. Grishchenko, M. Maffei, C. Schneidewind, A semantic framework for the security analysis of ethereum smart contracts, CoRR abs/1802.08660. arXiv:1802.08660.
URL http://arxiv.org/abs/1802.08660

[17] J. Zhu, K. Hu, M. Filali, J. Bodeveix, J. Talpin, Formal Verification of Solidity contracts in Event-B, CoRR abs/2005.01261. arXiv:2005.01261.
URL https://arxiv.org/abs/2005.01261

[18] R. Banach, Verification-led smart contracts, in: A. Bracciali, J. Clark, F. Pintore, P. B. Rønne, M. Sala (Eds.), Financial Cryptography and Data Security, Springer International Publishing, Cham, 2020, pp. 106–121.

[19] J.-R. Abrial, Modeling in Event-B: System and Software Engineering, 1st Edition, Cambridge University Press, New York, NY, USA, 2010.

[20] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin, Rodin: An Open Toolset for Modelling and Reasoning in Event-B, Int. J. Softw. Tools Technol. Transf. 12 (6) (2010) 447–466.

[21] P. Behm, P. Benoit, A. Faivre, J.-M. Meynadier, Météor: A Successful Application of B in a Large Project, Springer Berlin, 1999, pp. 369–387.

[22] F. Badeau, A. Amelot, Using B as a High Level Programming Language in an Industrial Project: Roissy VAL, in: H. Treharne, S. King, M. Henson, S. Schneider (Eds.), ZB 2005: Formal Specification and Development in Z and B, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 334–354.

[23] D. Essamé, D. Dollé, B in Large-Scale Projects: The Canarsie Line CBTC Experience, in: J. Julliand, O. Kouchnarenko (Eds.), B 2007: Formal Specification and Development in B, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 252–254.

[24] N. K. Singh, Using Event-B for Critical Device Software Systems, Springer, 2013.

[25] N. K. Singh, H. Wang, M. Lawford, T. S. E. Maibaum, A. Wassyng, Stepwise formal modelling and reasoning of insulin infusion pump requirements, in: Digital Human Modeling - Applications in Health, Safety, Ergonomics and Risk Management: Ergonomics and Health - 6th International Conference, DHM 2015, Held as Part of HCI International 2015, Los Angeles, CA, USA, August 2-7, 2015, Proceedings, Part II, 2015.

[26] R. Banach, Hemodialysis Machine in Hybrid Event-B, in: M. Butler, K.-D. Schewe, A. Mashkoor, M. Biro (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2016, pp. 376–393.

[27] N. K. Singh, Y. Aït Ameur, M. Pantel, A. Dieumegard, E. Jenn, Stepwise Formal Modeling and Verification of Self-Adaptive Systems with Event-B. The Automatic Rover Protection Case Study, in: 21st International Conference on Eng. of Complex Computer Systems, ICECCS, 2016, pp. 43–52.
[28] W. Su, J.-R. Abrial, Aircraft Landing Gear System: Approaches with Event-B to the Modeling of an Industrial System, Int. J. Softw. Tools Technol. Transf. 19 (2) (2017) 141166.
[29] N. Benaissa, D. Méry, Cryptographic protocols analysis in event b, in: A. Pnueli, I. Virbitskaite, A. Voronkov (Eds.), Perspectives of Systems Informatics, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 282–293.
[30] L. Laibinis, E. Troubitsyna, I. Pereverzeva, I. Oliver, S. Holtmanns, A formal approach to identifying security vulnerabilities in telecommunication networks, in: K. Ogata, M. Lawford, S. Liu (Eds.), Formal Methods and Software Engineering, Springer International Publishing, Cham, 2016, pp. 141–158.
[31] G. Dupont, Y. A. Ameur, N. K. Singh, M. Pantel, Event-B Hybridation: A Proof and Refinement-based Framework for Modelling Hybrid Systems, ACM Trans. Embed. Comput. Syst. 20 (4) (2021) 35:1–35:37. doi:10.1145/3448270.
URL https://doi.org/10.1145/3448270
[32] R. Banach, M. Butler, S. Qin, N. Verma, H. Zhu, Core Hybrid Event-B I: Single Hybrid Event-B machines, Science of Computer Programming.
[33] G. Dupont, Y. Aït-Ameur, M. Pantel, N. K. Singh, Proof-Based Approach to Hybrid Systems Development: Dynamic Logic and Event-B, in: M. J. Butler, A. Raschke, T. S. Hoang, K. Reichl (Eds.), 6th International Conference, ABZ 2018, Proc., Vol. 10817 of LNCS, Springer, 2018, pp. 155–170.
[34] P. Stankaitis, A. Iliasov, Y. Ait-Ameur, T. Kobayashi, F. Ishikawa, A. Romanovsky, A refinement based method for developing distributed protocols, in: 2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE), 2019, pp. 90–97.
[35] D. Méry, N. K. Singh, Analysis of DSR Protocol in Event-B, SSS'11, Springer-Verlag, Berlin, Heidelberg, 2011, p. 401415.
[36] M. B. Andriamiarina, D. Méry, N. K. Singh, Integrating proved state-based models for constructing correct distributed algorithms, in: 10th International Conference, IFM, Vol. 7940 of LNCS, 2013, pp. 268–284.
[37] F. Zoubeyr, Y. Aït Ameur, M. Ouederni, A. Tari, A correct-by-construction model for asynchronously communicating systems, Int. J. Softw. Tools Technol. Transf. 19 (4) (2017) 465–485.
[38] S. Benyagoub, Y. Aït Ameur, M. Ouederni, A. Mashkoor, A. Medeghri, Formal design of scalable conversation protocols using Event-B: Validation, experiments, and benchmarks, J. Softw. Evol. Process. 32 (2).
[39] M. J. Butler, P. Körner, S. Krings, T. Lecomte, M. Leuschel, L. Mejia, L. Voisin, The first twenty-five years of industrial use of the b-method, in: M. H. ter Beek, D. Nickovic (Eds.), 25th International Conference, FMICS, Vol. 12327 of LNCS, Springer, 2020, pp. 189–209.
[40] D. Méry, N. K. Singh, Automatic code generation from Event-B models, in: Proceedings of the 2011 Symposium on Information and Communication Technology, SoICT 2011, Hanoi, Viet Nam, October 13-14, 2011, 2011, pp. 179–188. doi:10.1145/2069216.2069252.
URL https://doi.org/10.1145/2069216.2069252
[41] EB2ALL, An automatic code generation tool from Event-B, http://eb2all.loria.fr/ (2011).
[42] R. L. Sites, Clean Termination of Computer Programs, Ph.D. dissertation, Stanford University, Stanford, California (June 1974).
[43] M. Leuschel, M. Butler, ProB: A Model Checker for B, LNCS, Springer, 2003, pp. 855–874.