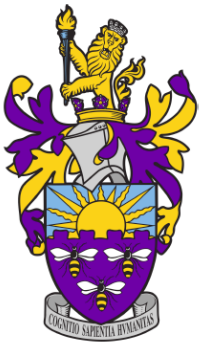


SMT-Based Bounded Model Checking for Solidity Smart Contracts

A Dissertation Submitted to The University of Manchester
For The Degree of Master of Science
In The Faculty of Science and Engineering

2021

Kunjian Song
10264348

Department of Computer Science

Contents

Word Count: 17831

List of Figures

List of Tables

Abstract

Apart from Bitcoin, Ethereum is another distributed ledger that uses blockchain technology. Smart contracts are autonomous programs that automatically control Ether's transactions in the distributive environment of the Ethereum blockchain. A vulnerable smart contract allows the hackers to perform unauthorized withdraw. Since a smart contract is immutable after its deployment on the Ethereum blockchain, which does not allow the owner to fix bugs, it becomes critical to make sure the smart contract is safe prior to deployment. Solidity is the most widely used programming language to create such contracts. There is a great deal of interest from academia and industry in formal verification for Solidity smart contracts.

The SMT-Based BMC has been successfully used to verify software programs written in general programming languages. ESBMC is a state-of-the-art SMT-based bounded model checker to verify C and C++ software. This project uses ESBMC as the vehicle to explore the opportunity to apply SMT-Based BMC for Solidity verification. However, Solidity is a domain-specific language for writing smart contracts. To extend ESBMC to verify Solidity smart contract, a detailed study of syntax, semantics and grammar rules of Solidity language was conducted. Two type checking methods were proposed to convert Solidity AST into ESBMC intermediate representation: *Tracker-Based Hybrid Conversion* and *Grammar-Based Hybrid Conversion*.

The *Grammar-Based Hybrid Conversion* method was found to have better extendibility and maintainability. As a result, a new Solidity frontend was developed to extend ESBMC to verify Solidity smart contacts. Additionally, a test suite that contains vulnerably smart contracts was developed due to the lack of a standard benchmark for Solidity. The test results confirmed the correctness of the new Solidity frontend that enables ESBMC to verify Solidity smart contracts. ESBMC was compared with other state-of-the-art Solidity verification tools by running the same test suite against other tools. The results show that ESBMC is the only tool that successfully detected all vulnerabilities in each test case and provided the corresponding counterexamples for each type of vulnerability. The other tools are only able to reveal the vulnerabilities in the test suite partially.

Keywords: programming language theory, compiler, bounded model checking, SMT, Solidity

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor Dr. Lucas Cordeiro for his guidance, encouragement, and advice he has provided throughout this project. ESBMC opened up a new world for me - a world in automated reasoning, decision procedures, and program verification.

1 Introduction

1.1 Motivation

The blockchain is a distributed ledger technology that forms the main mechanism behind Bitcoin, Ethereum, and alternative cryptocurrencies [1]. Blockchain can be considered as a data structure that contains a linked list of blocks, each of which contains a list of unmodifiable transactions [2]. Smart contracts are autonomous programs that run on Ethereum blockchain [3].

Solidity is a domain-specific language (DSL) for writing smart contracts [4]. Once deployed on Ethereum blockchain, there is no way to update the smart contract except deleting it completely and re-deploying a new one. Even the smart contract author cannot modify the program code or fix bugs after deployment [5]. Because the smart contracts are compiled into EVM (Ethereum Virtual Machine) assembly instructions for deployment on the blockchain [6]. Due to the nature of such immutability, it is critical to ensure the security of the smart contract before deploying it on the Ethereum blockchain. But the reality is the opposite. The deployed smart contracts often suffer from software vulnerabilities. These vulnerabilities have been exploited by malicious attackers, which leads to monetary losses. For example, the DAO attack that happened in 2016 results in a large monetary loss of $60 million, which eventually forces the Ethereum blockchain to be hard forked to roll back to a previous state [7, 8]. There is growing demand for the verification of Solidity smart contracts. For example, the *0xproject* offers up to $100,000 US dollars to detect of critical vulnerability in Solidity smart contract [9, 10].

1.2 Research Question, Aim and Objectives

Research Question.
Solidity is a DSL for writing smart contracts to be deployed on Ethereum blockchain. Hence, a natural question would be: Can SMT-based bounded model checking be used to verify DSL?

Aim.
This project aims to answer this question by using ESBMC as the vehicle for research. The goal is to develop a new type checking methodology to transform the Solidity program into ESBMC intermediate representation (IR), and ultimately to use the existing SMT-encoding schemes of ESBMC to verify the original Solidity program.

Objectives.
The objectives of this project are outlined as follows:

- [OBJECTIVE-1] Investigate the programming paradigm of Solidity and understand Solidity language syntax, semantics, grammar, and special features.
- [OBJECTIVE-2] Investigate and analyse ESBMC to identify:
    a. The architecture and the intermediate representations used in ESBMC.
    b. The IR data structures in ESBMC

c. The existing encoding schemes for verifying the language constructs in general programming languages like C and C++.

- [OBJECTIVE-3] Based on the outcomes of [OBJECTIVE-2], extend ESBMC to verify Solidity, a DSL for writing smart contracts.

1.3 Deliverables

The deliverables of this project are listed in the following:

- [Deliverable-1] Develop a new type checking methodology to enable ESBMC to verify Solidity smart contracts using the existing SMT encoding schemes.

- [Deliverable-2] Due to the lack of a standard benchmark for Solidity, develop a test suite that contains vulnerable smart contracts to evaluate the new frontend in *Deliverable-1*.

- [Deliverable-3] Final dissertation to summarize the work done for *Deliverable-1* and *Deliverable-2*.

1.4 Contribution

Over the last few years, several Solidity frameworks were proposed. Except for Mythril [11], the other frameworks outlined in Section 2.4 do not encode Solidity programs in logic formulae or use SMT solvers to verify Solidity smart contracts. Instead, some of them just use SMT solvers to find the satisfiability of path conditions after symbolically executing the program. Although Mythril uses SMT solver to verify the Solidity smart contracts, it does not always generate a counterexample. The test results in Section 4 shows that Mythril did not detect the vulnerability of arithmetic underflow in the test case developed in this project.

The contribution of this project is that it successfully used SMT-based bounded model checking technique to verify Solidity smart contracts. None of the state-of-the-art Solidity verification tools uses such technique. The test results in Section 4 show that ESBMC with the new Solidity frontend detected the vulnerability in all test cases and provided a counterexample in each case. ESBMC outperforms all other tools.

1.5 Dissertation Structure

This dissertation contains five chapters including the introduction. The layout of the remaining chapters is as follows. Chapter 2 presents the relevant background knowledge and theories in programming language theory, compiler design, software verification, bounded model checking, and Satisfiability Modulo Theories (SMT), which is used to aid in understanding the design rationale behind the new type checking methodologies in Chapter 3. There are two type checking methodologies proposed in Chapter 3. These methodologies guide the implementation of the new Solidity frontend. Chapter 4 discusses the test suite design, and explains the test results. In addition, Chapter 4 also compares ESBMC to other state-of-the-art Solidity verification frameworks. Chapter 5 concludes this project and identifies further work extending ESBMC to cover all Solidity language features.

2 Background and Theory

This chapter will lay the theoretical foundation that will become useful in later chapters. Each subsection contains an answer to a question "How is … related to this project?", which explains the intention of the corresponding literature review.

The architecture of an SMT-based bounded model checker consists of two parts: the frontend and the backend. This chapter starts with introducing topics in programming language theory and compiler theory that will help the reader understand the frontend of an SMT-based bounded model checker. Next, it uses an illustrative example to explain the verification flow of an SMT-based bounded model checker. This chapter then gives a survey of state-of-the-art verification frameworks for Solidity. It ends by discussing the existing clang-based frontend of ESBMC.

2.1 Programming Language Theory

This subsection aims to explain some key concepts in programming language theory, which lay the theoretical foundation for understanding the clang-based frontend in ESBMC, and designing the new Solidity frontend.

2.1.1 Syntax and Grammar

The syntax of a programming language can be described precisely using formal grammar [12]. The method to formally describe a formal grammar is known as Backus-Naur Form (BNF). The origin of BNF is the paper published by computer scientists John Backus and Peter Naur in 1960 [13]. BNF is referred to as a metalanguage used to describe another programming language. A metalanguage is a language used to describe another language [14]. Hence, the syntax of a programming language can be described by context-free grammar written in BNF. The context-free grammar contains production is shown as follows:

```
<Expr> ::= <Identifier> <BinaryOperator> < Identifier >
          | <UnaryOperator> <Identifier>

<Identifier> ::= a | b | c | d

<BinaryOperator> := + | -

<UnaryOperator> := ++
```

The simple grammar above is written in BNF style. <Expr> is known as a non-terminal symbol because it can be replaced by other symbols. <Identifier>, <BinaryOperator> and <UnaryOperator> are also non-terminal symbols. 'a', 'b', 'c' and 'd' are terminal symbols, so are the operators '+', '-' and '++'.

The above grammar can be used to check the validity of the syntax in the following expressions:

$$a + b \quad (2.1)$$
$$++a \quad (2.2)$$
$$a++ \quad (2.3)$$
$$--a \quad (2.4)$$
$$a * b \quad (2.5)$$

The syntax of expression (2.1) is valid, because the sequence of symbols in this expression can be generated using the production rules as follows:

<Expr> → <Identifier> <BinaryOperator> <Identifier>
<Identifier> → a
<BinaryOperator> → +
<Identifier> → b

A production rule is in the form of "<Identifier> → x". It means that the non-terminal symbol <Identifier> can be replaced by another symbol "x" as specified by the BNF-style grammar, where "x" could be a terminal symbol (e.g. "a") or a sequence of non-terminal symbols (e.g. as in the rule "<Expr>").

The syntax of expression (2.2) is valid, because the sequence of symbols in this expression can be generated using the production rules as follows:

<Expr> → <UnaryOperator> <Identifier>
<UnaryOperator> → ++
<BinaryOperator> → a

The syntax of expressions (2.3), (2.4) and (2.5) is invalid for the following reasons:

- There is no production rule to replace <Expr> with < Identifier ><UnaryOperator> in expression "a++".
- There is no production rule to replace <UnaryOperator> with a terminal symbol "--" used in the expression "--a".
- There is no production rule to replace <BinaryOperator> with a terminal symbol "*" used in the expression "a * b".

How is it related to this project?
The concept discussed in this subsection will become useful to understand the formal grammar of the Solidity language [16]. The production rules in Solidity grammar will guide the development of *Grammar-Based Hybrid Conversion Method*.

2.1.2 Operational Semantics and Type Checking

While the syntax of a programming language describes the structure of expressions, statements and language constructs, emantics is the meaning of those expressions, statements, and language constructs [14]. Type specifies the range of values that a variable can represent and the set of operations that are defined for these values. Type puts constraints on the operands and the operator to ensure they fit together properly [15].

Type and semantics are not independent of each other. They can be shown in the same picture of the processing phases of a programming language. There are two processing phases of a programming language: one is the static phase of processing, and the other one is the dynamic phase of processing [15]. There are two goals in the static phase of processing to ensure the program is well-formed:

1. Make sure the structure is correct. For example, a binary operator "+" expects two operands in the form of *(a + b)*, which is a well-formed expression. The expression *(a +)* is ill-formed because the right-hand-side operand is missing.

2. Make sure the type is compatible. For example, a binary operator "*" expects two operands of numerical types. *(1 + 2)* is a well-formed expression. The expression *(1 + "hello")* is ill-formed because the second operand is of the type *string* and it does not make sense to add an integer to a string.

The dynamic phase of processing refers to the execution of a well-formed program based on the semantics. One of the formal ways to describe semantics is to use operational semantics. There are two types of operational semantics based on the levels of interest [12, 14]:

1. If the interest is in the final result of the execution of a well-formed program, then it is called natural operational semantics, also known as big-step semantics.

2. If the interest is in the sequence of state changes during the execution, then it can be described by structural operational semantics, also known as small-step semantics.

An example of structural operational semantics is shown in Figure 1 [12]:

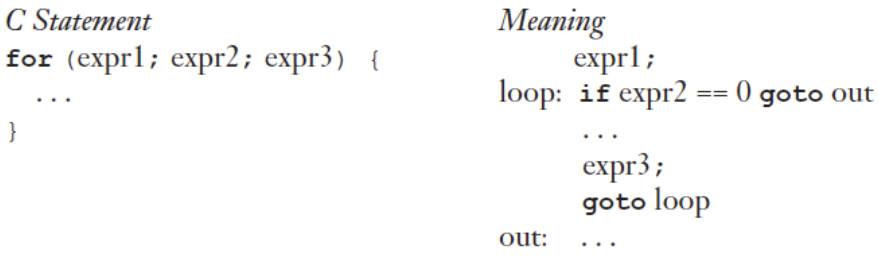


Figure 1: Structural operational semantics example.

As shown in Figure 1, the structural operational semantics of a *for* loop can be described using the sequential flow that consists of three terms: *if* statement, *goto* statement and the corresponding labels. The same approach can also be used to describe other non-sequential control flows, such as *do-while* loop or *while* loop. If it's a function call, the call expression can be replaced by the body of the function to make it sequential.

The type compatibility is enforced by typing judgements. The typing judgements are rules described in a similar way to the natural deduction [17]. An example of such judgement is described below:

First, let τ represent the types, and let e represent expressions defined in Figure 2 [15]:

Typ	τ ::=	`num`	`num`	numbers
		`str`	`str`	strings
Exp	e ::=	x	x	variable
		`num`$[n]$	n	numeral
		`str`$[s]$	$"s"$	literal
		`plus`$(e_1;e_2)$	$e_1 + e_2$	addition
		`times`$(e_1;e_2)$	$e_1 * e_2$	multiplication
		`cat`$(e_1;e_2)$	$e_1 \hat{} e_2$	concatenation
		`len`(e)	$\|e\|$	length
		`let`$(e_1;x.e_2)$	`let` x `be` e_1 `in` e_2	definition

Figure 2: τ and e definitions.

Next, a typing environment Γ is defined as a set of (e, τ). This pair means e is of the type τ, which is often denoted by $e : \tau$. Γ is sometimes called typing context. Using these notations, we can define a ternary relation:

$$\Gamma \vdash e : \tau$$

which means that e is of the type τ in the typing environment defined by Γ.

Then, the typing rules can be defined as follows [15]:

$$\frac{\Gamma \vdash a : num \qquad \Gamma \vdash b : num}{\Gamma \vdash plus(a, b) : str}$$

$$\frac{\Gamma \vdash a : str \qquad \Gamma \vdash b : str}{\Gamma \vdash cat(a, b) : str}$$

$$\frac{\Gamma \vdash a : str}{\Gamma \vdash len(a) : num}$$

The relations above the line are premises, and the relation below the line is the conclusion. The first rule states that if variable a is of the type *num* in the typing environment Γ, and variable b is of the type *num* in the typing environment Γ, then the *plus* function that operates on these variables should also be of the type *num* in the typing environment Γ. Analogously, the second rule enforces the type compatibility of the concatenation expression, and the third rule enforces the type compatibility of the length expression.

A type checker is essentially an algorithm that implements the typing rules given above.

How are structural operational semantics and type checker related to this project?
ESBMC uses GOTO program as the intermediate representation of the original program. The GOTO program uses only guarded *goto* and *assume* statements to model the control flow [18]. The GOTO program essentially describes the structural operational semantics. The frontend of ESBMC has a type checking phase. In clang-based frontend of ESBMC this type checking phase is implemented as the *clang_c_converter* class. This class checks the type of each clang AST node and transforms it into the equivalent tree-structured *irept* node.

2.1.3 Taxonomy

Programming paradigm is the style of programming, which can be used to classify programming languages. A program written in an imperative language consists of a sequence of commands which modify the memory (or state) [19]. The end of a sequential command is indicated by the semicolon. The type system imposes constraints on the formation of expressions [15]. A type system consists of the predefined types and the typing rules as discussed in the previous section. A program written in a strongly typed (or type safe) language cannot violate the distinctions between types defined in that language [19]. Object-oriented programming (OOP) is a programming paradigm with three fundamental features: Encapsulation, Inheritance and Polymorphism.

How is taxonomy related to this project?
The goal of this project is to verify smart contracts written in Solidity. It would be useful to know what type Solidity language is. In Solidity a *contract* is like a *class* in OOP which encapsulates the attributes to indicate the state of the contract and methods that defines the way how a smart contract can be interacted with. The statement (or command) in Solidity is also ended by a semicolon. Hence, Solidity is:

- An imperative programming language.
- An OOP language.
- A strongly typed language.

2.2 Compiler Theory

A compiler translates a source program to a target program. The translation process consists of five phases [20]:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Optimization
5. Code generation

Lexical Analysis.
In this phase, the characters are aggregated to form a word. This is done by a lexical analyzer, or sometimes called a scanner, which applies a set of rules to check the validity of the word. If it is valid, a token will be generated.

Syntax Analysis.
The input to the parser is a stream of tokens (or words). The purpose of the parser is to understand the structure of the program. The parser verifies the input tokens based on the formal grammar of the source programming language. The output of the parser is a tree-structured representation, usually a parse tree or abstract syntax tree (AST) [21]. Parse trees contain more information than AST. The parse tree usually includes a record of rules used to recognise the input [22]. In a parse tree a node might be the name of the production rule used. In an AST the tree is simplified by removing the nodes representing the name of the production rule. An AST is more concise than the parse tree [20]. The syntax analysis phase ensures a program to be well-formed (c.f. Section 1.1.2). Both parse tree and AST are considered as syntax tree. A syntax tree is a tree-structured intermediate representation (IR).

Semantic Analysis.
Semantic analysis is to check the meaning of the program. A semantic analyser uses syntrax tree (usually AST) as the input. First, the semantic analyser generates a symbol table that contains the type and scope information of each declaration node in the syntax tree. Next, the semantic analyser uses the syntax tree and the symbol table to ensure the source program is semantically consistent with the scopes and language type system [21]. Hence, the semantic analysis phase contains a type checker (c.f. Section 1.1.2).

How are these concepts related to this project?
Phases 4 and 5 are not related to this project because the goal of this project is not to improve the performance of a specific compiler or generate code for a specific target. Phases 1, 2 and 3 are related to this project, because the goal of ESBMC frontend is to generate the symbol table. The type checker traverses the AST and converts each declaration node into ESBM *irept* node. The *irept* parse tree is a tree-structured IR used to represent the syntax structure of the original program in ESBMC.

2.3 SMT-Based Bounded Model Checking

ESBMC is one of the most powerful SMT-based bounded model checkers to verify software programs written in C and C++ [23]. ESBMC has won various awards in previous SV-COMP competitions [24]. The overview of ESBMC is shown in Figure 3.

A finite-state transition system can be modelled by a Kripke structure M which has a set of states $S = \{s_0, s_1, \dots, s_{k-1}\}$, where $s_0 \in S_0$ and S_0 represents the set of initial states. A transition relation R is a subset (not necessarily a proper subset) of the Cartesian product S and S, i.e. $R \subseteq S \times S$. A state transition from s_j to s_{j+1} is denoted by $R(s_j,\ s_{j+1})$. It means that the program counter moves forward while taking some actions to update the state. Such actions can be evaluating an expression and making new assignment to a variable, modifying the element in a container data structure, or changing the flow of execution by jumping to another block of statements. Given a Kripke structure M that models a state transition system, Bounded Model Checking aims to build the verification condition (VC) as the following formula [25]:

$$\psi_k = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} R(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{2.6}$$

where I is the set of initial states, s_i is state variable and k represents the bound limit, e.g., the number of loop iterations BMC unwinds. In Eq (2.6), $I(s_0) \wedge \bigwedge_{j=0}^{i-1} R(s_j, s_{j+1})$ means the

execution trace of length i. $\neg\phi(s_i)$ represents violation of a property in state s_i. If the VC is satisfiable, it means that there exists a state s_i that violates the safety property. The violation can be arithmetic over- or underflow, divide by zero, accessing a null pointer, double frees, etc. The counterexample will be represented by a sequence of states $\{s_0, s_1, \ldots, s_k\}$ and the corresponding transitions $R(s_i, s_{i+1})$ where i is bounded by $0 \leq i < k$ [25].

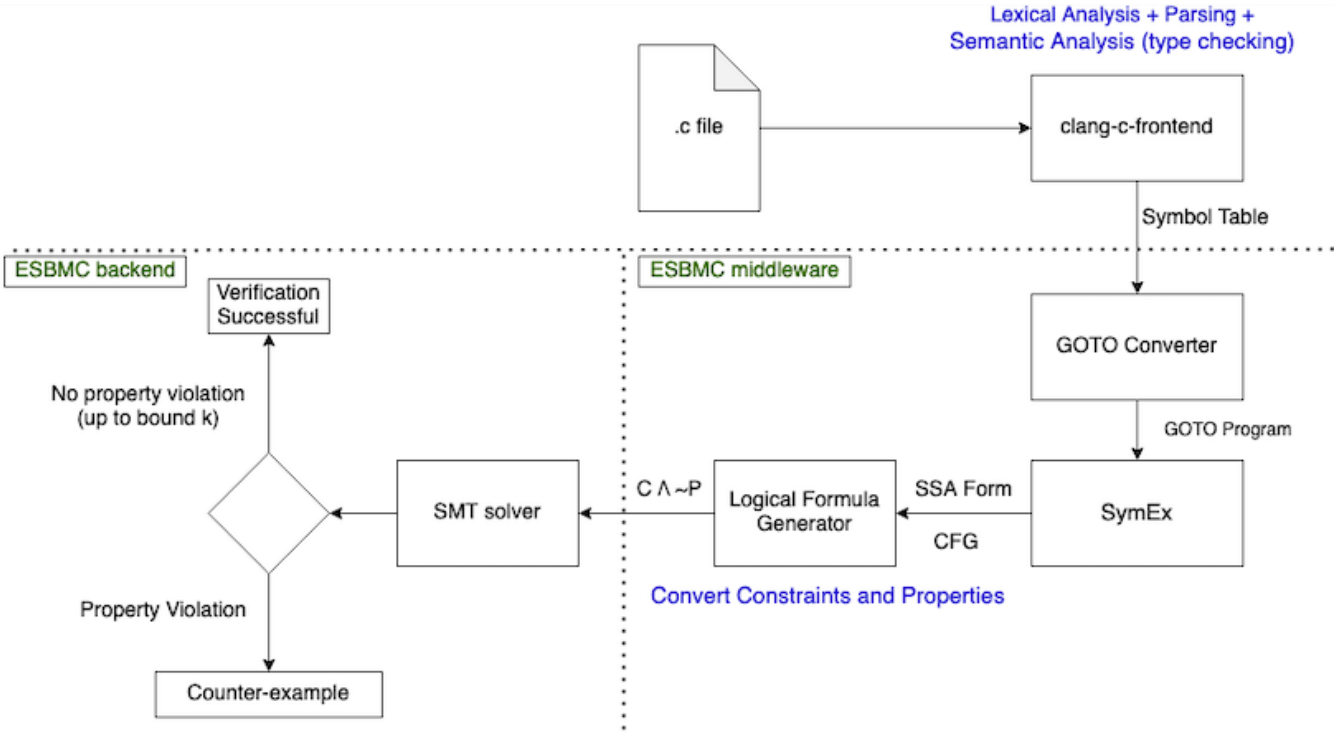


Figure 3: Overview of ESBMC.

In Eq. (2.6), $\neg\phi(s_i)$ corresponds to the property part, "~P" as in "C ∧ ~P" in Figure 3. The rest of the equation corresponds the constraint part, "C". If the verification condition ψ is satisfiable then it means that there exists a counterexample that violates the property up to a given bound k. However, BMC is incomplete if k is not high enough. It is only able to find a logic error (also called "falsification") up to k steps [26]. Calculating the completeness threshold (CT) of BMC is found to be as hard as the BMC itself [25].

```
1  int main() {
2
3    int a[2];
4    unsigned n = 2, i = 0;
5    while (i <= n)
6    {
7      a[i] = 100;
8      ++i;
9    }
10   assert(a[0] + 1 == 101);
11
12   return 0;
13 }
```

Figure 4: Example code to be verified.

The example code in Figure 4 contains an error of array out-of-bound access. To verify the code in Figure 4, the clang-c frontend generates symbol table, which is used by the GOTO converter to generate the equivalent GOTO program. Then the program is symbolically executed to generate the SSA form shown in Figure 5. Although the code in Figure 4 has no syntax error and can be compiled using GCC or Clang, it contains a run-time error of array out-of-bound access in line 7. The *while* loop is unrolled three times according to the value of *n*. The operational semantics of the statement in Line 9 in Figure 5 is as following:

```
if g0 == true:
then:
        a1 = (a0 WITH [0:=100]) ;
else:
        a1 = a0;
```

```
1  i0, i1, i2: bitvector(64)
2  i0 = 0;
3  n, N: bitvector(64)
4  n = 2;
5  N = 2; // array bound
6
7  g0 = (i0 <= n);
8  assert(i0 + 1 < N); // array bound check
9  a1 = g0? (a0 WITH [0:=100]) : a0;
10 i1 = g0? (i0+1) : i0
11
12 g1 = (i1 <= n);
13 assert(i1 + 1 < N); // array bound check
14 a2 = g1? (a1 WITH [1:=100]) : a1;
15 i2 = g1? (i1+1) : i1
16
17 g2 = (i2 <= n);
18 assert(i2 + 1 < N); // array bound check
19 a3 = g2? (a2 WITH [2:=100]) : a2;
20 i3 = g2? (i2+1) : i2
21
22 assert(a3[0] + 1 == 101)
```

Figure 5: SSA form of the code in Figure 4.

The array equality "a1 = a0" is defined as [25, 27, 28]:

$$a = b \Leftarrow \forall i \cdot select(a, i) = select(b, i)$$
$$a \neq b \Rightarrow \exists i \cdot select(a, i) \neq select(b, i)$$

The *if-then-else* statement is represented by the *ite* operation in Figure 6. The array theory of SMT solver is based on McCarthy axiom [26, 30]. The semantics are as follows:

- store(a, i, v) means to write the value of v in position i of array a. This expression returns the updated array.
- select(a, i) means to read the value at position i of array a. This expression returns the value at position i in that array.

```
C = [
  i0=1 /\ N=2 /\ n=2
  /\ g0=(i0 <= n) /\ a1=ite(g0, store(a0,0,100),a0) /\ i1=ite(g0,(i0+1),i0)
  /\ g1=(i1 <= n) /\ a2=ite(g1, store(a1,1,100),a1) /\ i2=ite(g1,(i1+1),i1)
  /\ g2=(i2 <= n) /\ a3=ite(g2, store(a2,2,100),a2) /\ i3=ite(g2,(i2+1),i2)
]

P = [
  (i0 + 1) < N
  /\ (i0 + 1) < N
  /\ (i0 + 1) < N
  /\ (select(a3,0) + 1) = 101
]
```

Figure 6: C and P formulae of the SSA form in Figure 5.

The *store(a, i, v)* function can be represented by the WITH operator. *"(a0 WITH [0:=100])*" means to write 100 in position 0 of array a0 and return the updated array. The select(a, i) function can be represented by the equivalent array subscript expression, *a[i]* [31], [32]. Note that in Figure 5, the initialization of array a0 is not shown, because arrays are unbounded in the array theory and we just use a symbolic representation "a0" to denote the initial array. Compared to the previous work on array encoding [33], Cordeiro .et. al [25] proposed a new method to check array out-of-bound by adding additional bound checks in each unrolled block to check the array index against the array bound. Note that ESBMC also applies reduction on the formulae shown in Figure 6. Therefore, the actual set of formulae solved by the SMT solver is simpler than the one derived manually in Figure 6.

2.4 State-of-The-Art Solidity Verification Frameworks

This section aims to outline five verification frameworks for Solidity smart contracts – SolAnalyser, Slither, Oyente, Smartcheck, and Mythril. Additionally, this chapter also discusses the Remix, most popular Solidity IDE, which also has some functions to assist the developers to verify a smart contract before deployment on the Ethereum *mainnet* [34].

SolAnalyser.
SolAnalyser is an automated verification framework for Solidity smart contracts. It uses both static and dynamic analysis. SolAnalyser framework relies on another code instrumentation tool called *Solidity Instrumentation Framework* (SIF). The responsibilities of SIF are [35]:

- Statically analyse the code for vulnerability detection.
- Inject assertions in the source code to specify property checks.
- Generate contract mutants by injecting a single hard-coded vulnerability into the original smart contract.

To inject assertions, SIF gathers information of each AST node and inject pre- or post-conditions into the original contracts based on the operands and operators. For example, the pre-condition for the vulnerability of division by zero of the expression "*a = b / c*" would be "c != 0". The post-condition for the vulnerability of unsigned underflow of the expression "*a = b + c*" would be "*a >= c && a >= b*" [35].

SolAnalyser uses mutation-based blackbox fuzzing as its strategy for dynamic analysis. A contract mutant generated by SIF will be compiled into EVM bytecode, which contains the

Abstract Binary Interface (ABI) of the mutant. SolAnalyser interacts with the ABI of the mutant and applies fuzz testing [34]. When the test is complete, SolAnalyser searches the test logs for a sequence of specific keywords and events that indicate a violation of the property checks [53].

Slither.
Unlike SolAnalyser, Slither is a verification framework that uses only static analysis [29]. The static analysis technique used in Slither is taint tracking (not to be confused with taint checking). First Slither transforms Solidity AST into an intermediate representation called SlithIR and converts the IR into SSA form, and symbolically execute the SSA form. Next, Slither tracks the data dependency using tainted tracking. If the data is tainted, it means that the data cannot be trusted. A tainted variable is an untrusted variable [36]. Slither marks a variable as tainted if the variable can be influenced by the user. For example, if a variable depends on another user-controlled variable, Slither will also mark it as tainted [29]. If a protected function depends on the tainted variable, then a potential vulnerability might be detected. For example, if the parameter of a function is tainted, the usage of this variable in the function body might be vulnerable. Slither uses a group of pre-defined bug detectors to make the final verdict. Apart from detecting vulnerabilities, Slither is also able to suggest code optimizations.

Oyente.
Due to the non-determinism and complexity in Ethereum blockchain, it requires much more effort to simulate the execution environment of such distributed system input-by-input using dynamic analysis techniques [37]. Unlike SolAnalyser and Slither that work on Solidity source code, Oyente works on EVM assembly code to follow the execution model of a smart contract. As a by-product, the CFG of EVM assembly code can be generated by Oyente [37]. Z3 solver is used to find the satisfiability of the branch condition for a path, which is explored using Depth First Search (DFS). When a smart contract makes a function call, Oyente collects the path condition of the caller and checks the updated states before the callee finishes. If the updated states still satisfy the path condition for the caller, then it is possible for the callee to re-enter the caller, and hence, re-executing the caller. This refers to a *Re-entrancy* vulnerability [38, 39].

Smartcheck.
Similar to Slither, Smartcheck converts the Solidity source code into XML-based intermediate representation (IR) [40]. Instead of using symbolic execution as in Slither, Smartcheck uses XPath queries on the IR to detect vulnerability patterns [41]. Another difference to Slither is that Smartcheck performs lexical and syntactical analysis on Solidity source code instead of using the Solidity, while Slither takes the Solidity JSON AST as input generated by Solidity compiler.

Mythril.
Mythril is a verification tool that works on EVM bytecode [42]. Unlike the other tools proposed in the research, Mythril is a verification tool developed by the company ConsenSys [43]. Mythril forms part of the security analysis platform MythX [44]. Mythril uses various techniques for software verification– symbolic execution, SMT solving and taint analysis.

Table 1 summarizes the verification strategies used in these tools:

Tools		SolAnalyser	Slither	Oyente	Smartcheck	Mythril
Input	Source code	✓			✓	
	JSON AST		✓			
	EBM bytecode			✓		✓
Techniques	Convert to IR		✓ (SlithIR)		✓ (XML-based IR)	
	Symbolic Execution		✓	✓	✓	✓
	Taint analysis		✓			✓
	SMT solver			✓		✓
	Fuzzing	✓				
	Code Instrumentation Or Other queries	✓ (Code instrumentation)			✓ (XPath queries)	

Table 1: Summary of Solidity verification tools.

How are these frameworks related to this project?
A test suite will be designed in this project to evaluate the new Solidity frontend in ESBMC, and each of the test case in this suite will be a vulnerable contract. To compare ESBMC with other state-of-the-art verification tools, the same test suite will be run again all tools discussed in this section.

2.5 Clang-Based Frontend in ESBMC

A detailed study of the clang-based frontend is conducted to figure out the usage of ESBMC *irept* data structures. The clang-based frontend traverses the AST of the input C program and generates a symbol table. First, the clang AST is converted into another IR called *irept*. Then the type checker annotates the *irept* node based on the information in clang AST node and generates the corresponding symbol, which is then added to the symbol table.

2.5.1 Clang AST Context

Since this frontend uses a mix of external APIs of the clang compiler infrastructure to facilitate the conversion of clang AST node into the equivalent *IRep* tree node in ESBMC, the conversion process may seem quite intricate to a reader without knowing the specifics of clang.

Clang is the official LLVM frontend for C, C++, Objective-C and Objective-C++ [45]. As of version 5.0, ESBMC started to use the clang-based frontend [46]. This frontend uses two clang components: *clang::tooling* and *clang::ASTUnit*. The *clang::tool* class provides utility functions to perform frontend actions, such as getting the current file name, build AST from the current file, .etc. The *clang::ASTUnit* class is also a utility class that provides APIs to generate the AST context and retrieve each AST node from that context. The AST context can be obtained using the getter API *getASTContext()*. A full list of APIs can be found in the *clang::tool* and *clang::ASTUnit* reference manuals, respectively [47, 48].

In a clang AST context, the root is the translation unit declaration shown in Figure 7. The child nodes of the translation unit represent the user-defined declarations, e.g. variable declaration nodes or function declaration nodes. Figure 8 shows the text representation of clang AST context, which is printed by the *dump()* function. The translation unit declaration is shown as the root of the tree in line 1 in Figure 8.

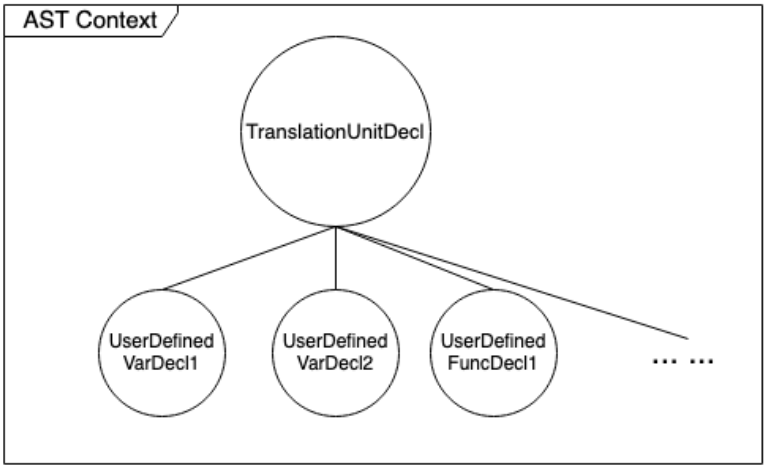


Figure 7: Clang AST context structure

Each declaration is represented by different types derived from the declaration based class in *clang::Decl*. Table 2 shows the class representations of each declaration node in clang.

The top-level translation unit declaration is represented *clang::Decl::TranslationUnit,* which is not a standard language construct but a clang internal data structure to facilitate the compiling process. This translation unit can be obtained using the getter API *getTranslationUnitDecl()*.

The type of a declaration node can be obtained using the getter API *getKind()*. For each type of clang declaration in *clang::Decl,* there exists a corresponding conversion function that converts a clang AST node into *irept* node in ESBMC, which preserves the semantics of the original clang AST node.

C Language Construct	clang Declaration Type	clang Class
Label	*clang::Decl::Label*	*clang::LabelDecl*
Var	*clang::Decl::Var*	*clang::VarDecl*
Function	*clang::Decl::Function*	*clang::FunctionDecl*
Field	*clang::Decl::Field*	*clang::FieldDecl*
TypeDef	clang::Decl::Typedef	*clang::TypedefDecl*

Table 2: Clang declaration class

```
TranslationUnitDecl 0x7fd37a80a608 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x7fd37a80aec8 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x7fd37a80aba0 '__int128'
|-TypedefDecl 0x7fd37a80af38 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x7fd37a80abc0 'unsigned __int128'
|-TypedefDecl 0x7fd37a80b240 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
| `-RecordType 0x7fd37a80b010 'struct __NSConstantString_tag'
|   `-Record 0x7fd37a80af90 '__NSConstantString_tag'
|-TypedefDecl 0x7fd37a80b2e8 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x7fd37a80b2a0 'char *'
|   `-BuiltinType 0x7fd37a80a6a0 'char'
|-TypedefDecl 0x7fd38a80c000 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag [1]'
| `-ConstantArrayType 0x7fd37a80b580 'struct __va_list_tag [1]' 1
|   `-RecordType 0x7fd37a80b3c0 'struct __va_list_tag'
|     `-Record 0x7fd37a80b340 '__va_list_tag'
|-FunctionDecl 0x7fd38a80c130 <esbmc_intrinsics.h:1:1, col:26> col:6 __ESBMC_assume 'void (_Bool)'
| `-ParmVarDecl 0x7fd38a80c068 <col:21> col:26 '_Bool'
|-FunctionDecl 0x7fd38a80c390 <line:2:1, col:40> col:6 __ESBMC_assert 'void (_Bool, const char *)'
| |-ParmVarDecl 0x7fd38a80c230 <col:21> col:26 '_Bool'
| `-ParmVarDecl 0x7fd38a80c2b0 <col:28, col:39> col:40 'const char *'
|-FunctionDecl 0x7fd38a80c5e0 <line:3:1, col:53> col:7 __ESBMC_same_object '_Bool (const void *, const void *)'
| |-ParmVarDecl 0x7fd38a80c488 <col:27, col:38> col:39 'const void *'
| `-ParmVarDecl 0x7fd38a80c508 <col:41, col:52> col:53 'const void *'
|-FunctionDecl 0x7fd38a80c6e0 <line:4:1, col:27> col:6 __ESBMC_atomic_begin 'void ()'
|-FunctionDecl 0x7fd38a80c7a0 <line:5:1, col:25> col:6 __ESBMC_atomic_end 'void ()'
|-FunctionDecl 0x7fd38a80c928 <line:7:1, col:20> col:5 __ESBMC_abs 'int (int)'
| `-ParmVarDecl 0x7fd38a80c858 <col:17> col:20 'int'
|-FunctionDecl 0x7fd38a80cab8 <line:8:1, col:31> col:10 __ESBMC_labs 'long (long)'
| `-ParmVarDecl 0x7fd38a80c9e8 <col:23, col:28> col:31 'long'
```

Figure 8: Clang AST context in text format

As an example, we consider a variable declaration node shown in Figure 9.

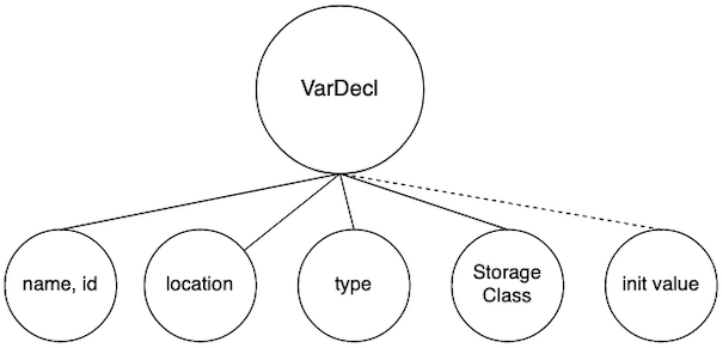


Figure 9: The parse tree of a variable declaration node

The edge between the initial value node and its parent node is represented by a dashed line because the initial value is optional. A variable declaration without initial value is legal in most of the languages. If an initial value is provided, e.g. as in 'int a = 1;', this variable declaration node will become an initialisation node. An initialisation node is a variable declaration node with an additional child representing the initial value.

Therefore, a variable declaration can be represented by a tree that consists of a parent node denoting the variable declaration and a group of child nodes defining the semantic of the parent node:

- Name and ID of the variable
- Location of the variable declaration in the source file, e.g. the name of the source file and the line number in that source file
- Type of the variable
- Storage class that tells the type of variable, e.g. static, global, extern, or local variable within a function.
- Initial value

A variable declaration node is represented by *clang::VarDecl* class. Its child nodes are also represented by the corresponding clang classes. Each child node can be extracted using the corresponding getter API of the parent node class, *clang::Decl.* Table 3 summarizes the clang classes for each type of the child node along with their getter APIs.

Node Type	clang class	Getter
Name, ID	*clang::IdentifierInfo*	*getIdentifier()* *ID.getName()*
Location	*clang::SourceLocation*	*getSourceRange().getBegin()*
Type	*clang::QualType*	*getTypePtrOrNull()*
Storage Class	*clang::StorageClass*	*getStorageClass()*
Init Value	*clang::Expr*	*getInit()*

Table 3: Clang classes for semantics of the child nodes.

2.5.2 ESBMC Intermediate Representation: *irept*

The purpose of the clang-based frontend is to traverse the AST and generate the symbol table. This task is performed by the *clang_c_converter* class located in the "*src/clang-c-frontend/*" directory. The converter starts from the translation unit declaration, the root node shown in Figure 7. When traversing the AST, there is no need to implement the traversal algorithm. Because clang already provides the APIs to traverse AST tree. Clang provides the *decls()* API to return an range expression, *llvm::iterator_range<decl_iterator>*. All the declarations are contained in a container data structure bounded by this range iterator. To visit each declaration node one by one, all we need to do is executing a range-based *for* loop over that range expression. The *decl_iterator* can be dereferenced to the pointer that points to a declaration node represented by the *clang::Decl* class.

Now we have the pointer to each declaration node. The goal is to convert each declaration node into the equivalent *irep* node and hence the corresponding symbol. However, the declaration nodes extracted from the range expression might be of different types (cf. Table 2). The type information of a *clang::Decl* can be determined using the getter API *getKind()*.The converter class provides different functions to process different types of declarations as shown in Table 4.

C Language Construct	Declaration Conversion Function
Label	*bool **get_decl**(const clang::Decl &, exprt &)*
Var	*bool **get_var**(const clang::VarDecl &, exprt &)*
Function	*bool **get_function**(const clang::FunctionDecl &, exprt &)*
Field	*bool **get_decl**(const clang::Decl &, exprt &)*
TypeDef	No conversion needed

Table 4: Conversion functions for declaration node.

Unlike a variable or function declaration, *TypeDef* is not considered as an "identifier". The converter ignores *TypeDef* declaration because clang will always give the underlying type defined by the *typedef*. The conversion function takes different types of the declaration node as the first parameter, but the types pointed by the *decl_iterator* are different.

To illustrate the conversion process of transforming a clang AST node into an *irept* node and then into the symbol, let us use the variable declaration node as an example. The structure of a variable declaration node with its child nodes is shown in Figure 9. Each child node contains just one piece of semantic information of the variable declarations. For each clang AST node in Figure 9, there exists an equivalent *irept* node in ESBMC, which preserves the semantic information. Table 5 shows the mapping between clang node classes and ESBMC *IRep* node classes.

Node Type	clang class	irept class
Name, ID	*clang::IdentifierInfo*	*std::string (C++ data type)*
Location	*clang::SourceLocation*	*locationt*
Type	*clang::QualType*	*typet*
Storage Class	*clang::StorageClass*	*Decomposed and represented by 'bool'*
Init Value (or Function body)	*clang::Expr*	*exprt*

Table 5: Mapping clang classes to ESBMC irept nodes.

For variable name and ID, *clang_c_converter* uses C++ build-in class *std::string*. As for the storage class, it is decomposed and represented by three Boolean variables: *static_lifetime*, *is_extern*, and *file_local*.

The *typet* class is a base class that also implements the *irept* interface. Different C data types are represented by different derived classes of this base class as shown in Figure 10. A typet node may contain multiple child nodes to hold the semantic information of a more complex data structures. For example, the *subtye* node of an *array_typet* node represents the type of the elements stored in an array. The *size* node represents the size expression. In C language, it is legal to use an arithmetic expression as the argument of the array subscripting operator. For this reason, the *size* node is represented by *exprt* class. The structure of *array_typet* node is shown in Figure 11.

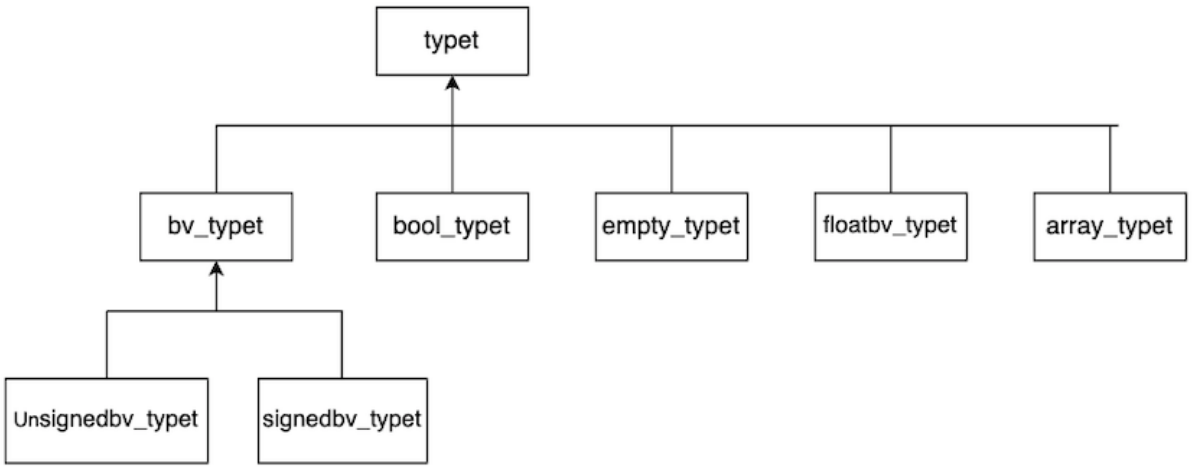


Figure 10: Class inheritance hierarchy of typet

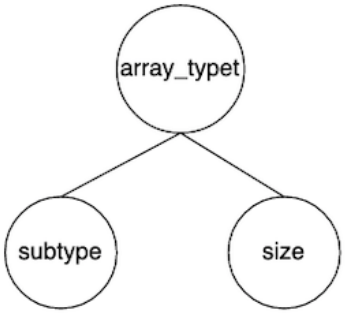


Figure 11: Structure of the array_typet tree

During the conversion process, the *clang_c_converter* first creates an equivalent *irept* node based on the mapping shown in Table 5. Next, the *clang_c_converter* calls the corresponding function to complete the conversion. These conversion functions are shown in Table 6.

Node Type	Child Node Conversion Function
Name, ID	*void* ***get_decl_name****(const clang::NamedDecl &, std::string &, std::string &)*
Location	*void* ***get_location_from_decl****(const clang::Decl &, locationt &)*
Type	*bool* ***get_type****(const clang::QualType &, typet &)*
Storage Class	Decomposed and processed by the ***get_var*** function in Table 4
Init Value (or Function body)	*bool* ***get_expr****(const clang::Stmt &, exprt &)*

Table 6: Conversion functions for child nodes.

When converting a variable declaration node, function *get_var* (cf. Table 4) will be called to process each child node extracted by the getters as listed in Table 3. The *irept* nodes listed in Table 5 will be created when calling each corresponding conversion function listed in Table 6. Each conversion function will annotate the equivalent *irept* node to preserve the semantic information held in each child node. Figure 12 shows the transformation of a clang *VarDecl* node into an *irept* node of ESBMC.

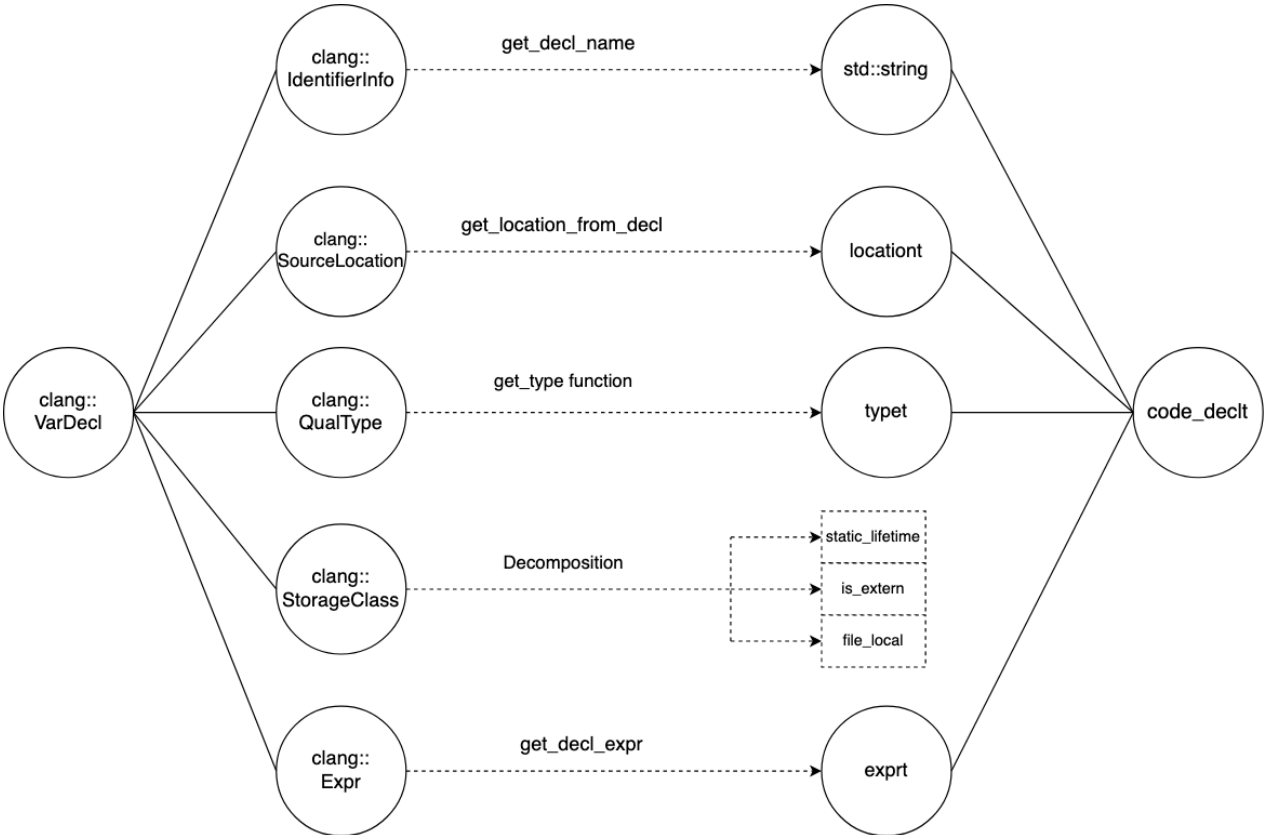


Figure 12: Transform clang::VarDecl into code_declt

In Figure 12 each clang class is a generalised data structure to hold an AST node's semantic information and metadata. They are not exclusive to the *clang::VarDecl* node but can be used to represent the semantics of other AST nodes. For example, *clang::QualType* can be used to represent the type of another language construct, including:

- The type of a variable
- The type of the operands in a *BinaryOperator* expression
- The type of a referenced variable in *DeclRef* expression
- The return type of a function declaration
- The return type of a function call in *CallRef* expression
- The type of the elements of an array (the subtype as shown in Figure 11)

Similar to *clang::QualType*, clang::SourceLocation can also be used to represent the location of any language construct, whether it is a variable declaration, a block of statements, a single statement, a control statement (e.g. an *if* statement), or other types. Therefore, the conversion functions in Table 6 are not only used for the conversion of a *VarDecl* node conversion, but also used for other types of nodes, e.g. a function declaration node, a block, or an expression node, .etc. As shown in Figure 13, the conversion functions are generalised to process the semantic information and meta data held in each type of child nodes. The parent of these child nodes can be of any type listed in Table 2.

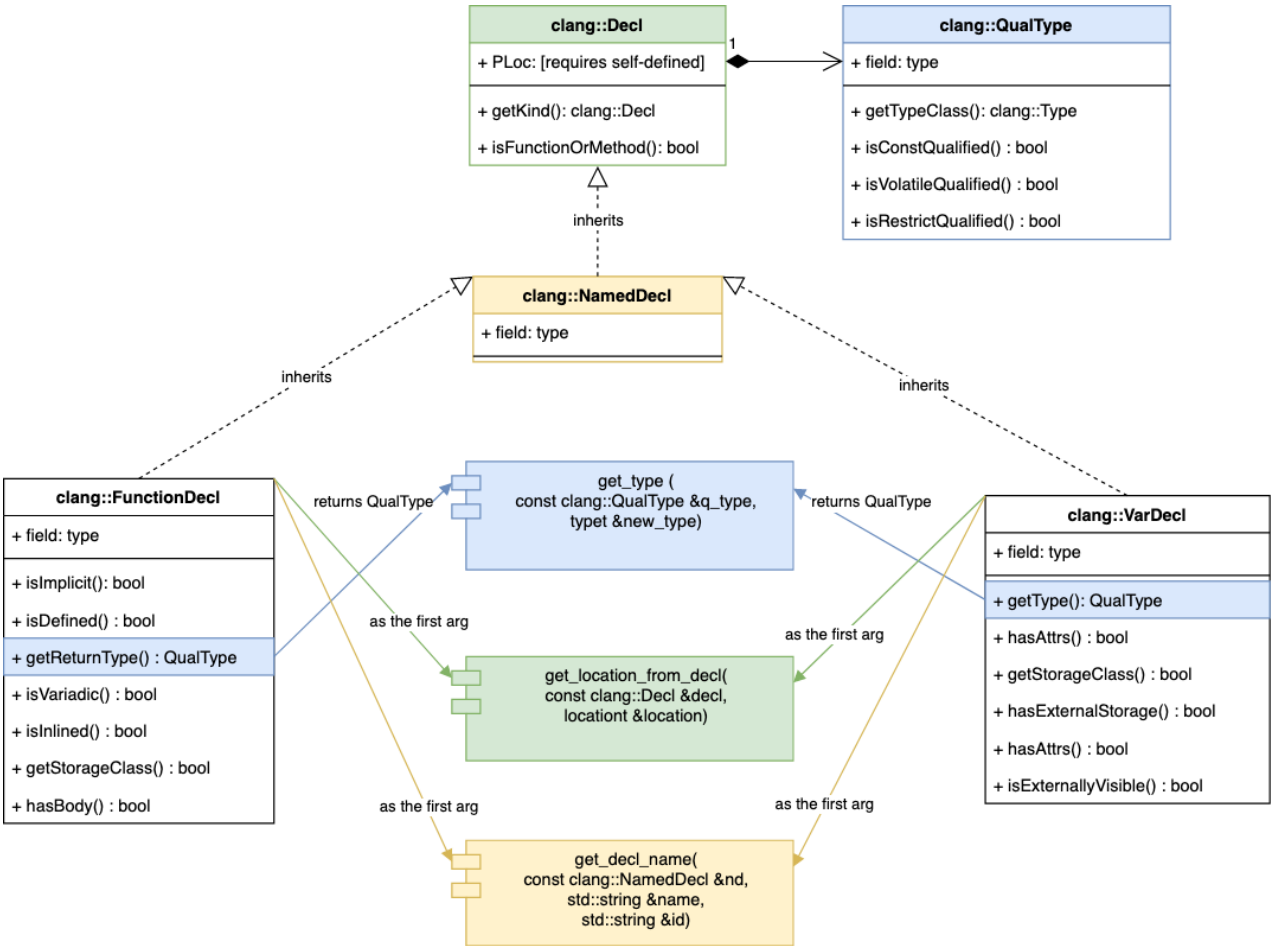


Figure 13: UML of clang-based frontend for VarDecl and FunctionDecl conversion.

3 Methodology and Implementation

This chapter aims to outline the methodology that guides the implementation of the new Solidity frontend. Section 3.1 uses an illustrative example to present an overview of the final methodology implemented, which hopefully helps the reader to grasp the big picture. At the beginning of the project, two methodologies were proposed: one relies on the Solidity compiler libraries, and the other one uses Solidity JSON AST. The latter was chosen to implement. Section 3.2 discusses the design rationale and explains why the second methodology is chosen. As a result, there two versions of implementation for the second methodology. Section 3.3 describes the limitations of the first version based on *Tracker-Based Hybrid Conversion*. Section 3.4 outlines the improved version based on *Grammar-Based Hybrid Conversion*.

3.1 Illustrative Example

This section gives an overview of the final methodology and implementation, which is referred to as *Grammar-Based Hybrid Conversion.* The verification pipeline that uses the new Solidity frontend is shown in Figure 14.

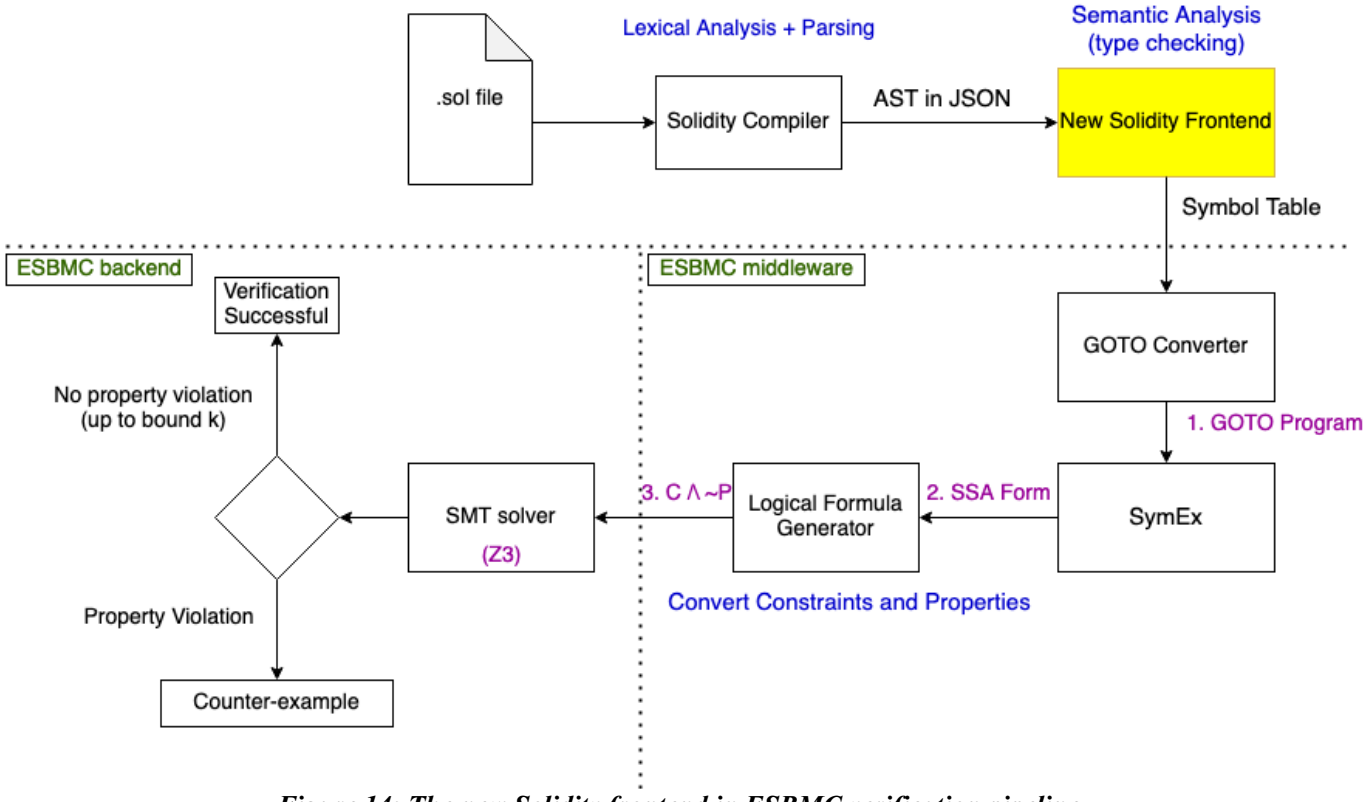


Figure 14: The new Solidity frontend in ESBMC verification pipeline.

As shown in Figure 14, the new Solidity frontend takes Solidity JSON AST as input. The ultimate goal is to convert Solidity JSON AST into the quantifier-free formulae *C* and *P*. The conversion steps are a follow:

1. Generate the GOTO program
2. Generate the SSA form
3. Generate the C /\ ~P

We will use the example code in Figure 15 to show the intermediate output and illustrate each conversion step.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.4.26;
3
4  contract MyContract {
5      uint8 x;
6
7      function nondet() public pure returns(uint8)
8      {
9        uint8 i;
10       return i;
11     }
12
13     function get_x() public returns(uint8)
14     {
15       x = 253;
16       return x;
17     }
18
19     function func_case_study() external
20     {
21       uint8 y = nondet();
22       uint8 sum;
23
24       x = get_x();
25
26       if (y > x)
27       {
28         sum = get_x() + 10;
29       }
30       else
31       {
32         sum = x + 1;
33       }
34
35       assert(sum > 100);
36     }
37 }
```

Figure 15: Example code to illustrate conversion steps.

The smart contact shown in Figure 15 contains one state variable *x*, and three functions:

- *nondet*: a pure function that does not change the state variable *x*. This function is used to assign a non-deterministic value to a variable shown in line 21.
- *get_x*: a public function that changes value of the state variable *x* to 253 and returns this value.
- *func_case_study*: This is the function to be verified using ESBMC. This function calls the other two functions.

The Solidity smart contract shown in Figure 15 is syntactically correct. However, the *func_case_study* function contains an error of arithmetic overflow. All variables were declared as *uint8*, which represents a value within the range from 0 to 255. The final value of *sum* depends on the predicate "*y* > *x*". The value of *x* is 253, but the value of *y* is non-deterministic: it can take any value from 0 to 255 returned from the function *nondet* as shown in line 21. If *y* is 254, then the expression "*y* > *x*" evaluates to *true* and the addition expression in line 28 will become "*sum = 253 + 10*". The final value of *sum* would become 263 but this value is not within the valid range 0-255 represented by *uint8*, which leads to the arithmetic overflow error. The following subsections will walk the reader through the intermediate output of each conversion step, and finally shows the detection of this error by ESBMC.

3.1.1 GOTO Program

To verify the Solidity smart contract shown in Figure 15, the type checker converts each AST node into a symbol and generate the symbol table. Then the GOTO converter will use this symbol table to produce the GOTO program. The GOTO program is the language-independent IR in ESBMC.

Figure 16, Figure 17, and Figure 18 compare each original Solidity function with the equivalent GOTO program. The state variable *x* is shown as the global variable in *__ESBMC__main* shown in Figure 19. Each statement is colour coded to show the correspondence.

```
function nondet() public pure returns(uint8)
{
  uint8 i;
  return i;
}
```

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

nondet (sol:@F@nondet):
        // 17 file MyContract_case_study.sol line 1 function nondet
        unsigned char i;
        // 18
        RETURN: i
        // 19 file MyContract_case_study.sol line 1 function nondet
        END_FUNCTION // nondet
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Figure 16: GOTO program of nondet function.

```
function get_x() public returns(uint8)
{
  x = 253;
  return x;
}
```

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

get_x (sol:@F@get_x):
        // 20 file MyContract_case_study.sol line 1 function get_x
        x=253;
        // 21
        RETURN: x
        // 22 file MyContract_case_study.sol line 1 function get_x
        END_FUNCTION // get_x
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Figure 17: GOTO program of get_x function.

```
    function func_case_study() external
    {
      uint8 y = nondet();
      uint8 sum;

      x = get_x();

      if (y > x)
      {
        sum = get_x() + 10;
      }
      else
      {
        sum = x + 1;
      }

      assert(sum > 100);
    }
}
```

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

func_case_study (sol:@F@func_case_study):
        // 23 file MyContract_case_study.sol line 1 function func_case_study
        unsigned char y;
        // 24 file MyContract_case_study.sol line 1 function func_case_study
        unsigned char return_value$_nondet$1;
        // 25 file MyContract_case_study.sol line 1 function func_case_study
        FUNCTION_CALL:  return_value$_nondet$1=nondet()
        // 26 file MyContract_case_study.sol line 1 function func_case_study
        y=return_value$_nondet$1;
        // 27 file MyContract_case_study.sol line 1 function func_case_study
        unsigned char sum;
        // 28 file MyContract_case_study.sol line 1 function func_case_study
        FUNCTION_CALL:  x=get_x()
        // 29 no location
        IF !((signed int)y > (signed int)x) THEN GOTO 1
        // 30 file MyContract_case_study.sol line 1 function func_case_study
        unsigned char return_value$_get_x$2;
        // 31 file MyContract_case_study.sol line 1 function func_case_study
        FUNCTION_CALL:  return_value$_get_x$2=get_x()
        // 32 file MyContract_case_study.sol line 1 function func_case_study
        sum=(unsigned char)((signed int)return_value$_get_x$2 + 10);
        // 33 file MyContract_case_study.sol line 1 function func_case_study
        dead return_value$_get_x$2;
        // 34 file MyContract_case_study.sol line 1 function func_case_study
        GOTO 2
        // 35 file MyContract_case_study.sol line 1 function func_case_study
     1: sum=(unsigned char)((signed int)x + 1);
        // 36 file MyContract_case_study.sol line 1 function func_case_study
     2: ASSERT (signed int)sum > 100
        // 37 file MyContract_case_study.sol line 1 function func_case_study
        dead sum;
        // 38 file MyContract_case_study.sol line 1 function func_case_study
        dead y;
        // 39 file MyContract_case_study.sol line 1 function func_case_study
        dead return_value$_nondet$1;
        // 40 file MyContract_case_study.sol line 1 function func_case_study
        END_FUNCTION // func_case_study
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Figure 18: GOTO program of func_case_study.

```
__ESBMC_main (__ESBMC_main):
        // 41 file esbmc_intrinsics.h line 16
        __ESBMC_alloc=ARRAY_OF(0);
        // 42 file esbmc_intrinsics.h line 19
        __ESBMC_deallocated=ARRAY_OF(0);
        // 43 file esbmc_intrinsics.h line 22
        __ESBMC_is_dynamic=ARRAY_OF(0);
        // 44 file esbmc_intrinsics.h line 25
        __ESBMC_alloc_size=ARRAY_OF(0);
        // 45 file esbmc_intrinsics.h line 30
        __ESBMC_rounding_mode=0;
        // 46 file MyContract_case_study.sol line 1
        x=0;
        // 47 file pthread_lib.c line 54
        __ESBMC_num_threads_running=0;
        // 48 file pthread_lib.c line 53
        __ESBMC_num_total_threads=0;
        // 49 no location
        FUNCTION_CALL:  pthread_start_main_hook()
        // 50 no location
        FUNCTION_CALL:  func_case_study()
        // 51 no location
        FUNCTION_CALL:  pthread_end_main_hook()
        // 52 file MyContract_case_study.sol line 1 function func_case_study
        END_FUNCTION // func_case_study

```

Figure 19: Statement variable and function call.

As shown in Figure 18 and Figure 19, the GOTO program of the Solidity smart contract has three important features:

- **New intermediate variables.**
  It introduces new intermediate variables to facilitate the creation of SSA form by the symbolic execution engine *SymEx* in the verification pipeline. For example, the return value of *get_x* function is represented by the intermediate variable *return_value$_nondet$1*, as shown in line 73 of Figure 18.

- **Change of Control Flow.**
  The GOTO program represents the control flow of the original program using guarded GOTO statements. For example, the *if-then-else* statement is represented by the IF-THEN-GOTO X statement, where X represents the label number. The expression "*sum = x + 1"* is represented by its equivalent labelled statement in line 95 of the GOTO program.

- **Statement variable as Global variable.**
  In the GOTO program the statement variable in a smart contract is represented by the global variable in *__ESBMC__main* function.

3.1.2 SSA Form

```
1  // all variables were declared as uint8
2  y1 = nd_uchar1;
3  x1 = 253;
4  g1 = y1 > x1;
5  sum1 = x1 + 10; // 7 due to overflow!
6  sum2 = x1 + 1;  // 254
7  sum3 = ite(g1, sum1, sum2);
8
9  assert(sum3 > 100);
```

```
46  Thread 0 file MyContract_case_study.sol line 1 function func_case_study
47  ASSIGNMENT ()
48  y?1!0&0#1 == func_case_study::$tmp::return_value$_nondet$1?1!0&0#1
49
50  Thread 0 file MyContract_case_study.sol line 1 function get_x
51  ASSIGNMENT ()
52  x&0#2 == 253
53
54  Thread 0
55  ASSIGNMENT (HIDDEN)
56  x&0#3 == 253
57
58  Thread 0 file MyContract_case_study.sol line 1 function func_case_study
59  ASSIGNMENT (HIDDEN)
60  goto_symex::guard?0!0&0#1 == (signed int)y?1!0&0#1 > 253
61
62  Thread 0 file MyContract_case_study.sol line 1 function get_x
63  ASSIGNMENT ()
64  x&0#4 == 253
65
66  Thread 0
67  ASSIGNMENT (HIDDEN)
68  func_case_study::$tmp::return_value$_get_x$2?1!0&0#1 == 253
69
70  Thread 0 file MyContract_case_study.sol line 1 function func_case_study
71  ASSIGNMENT ()
72  sum?1!0&0#1 == 7
73
74  Thread 0 file MyContract_case_study.sol line 1 function func_case_study
75  ASSIGNMENT (HIDDEN)
76  x&0#5 == 253
77
78  Thread 0 file MyContract_case_study.sol line 1 function func_case_study
79  ASSIGNMENT ()
80  sum?1!0&0#2 == 254
81
82  Thread 0 file MyContract_case_study.sol line 1 function func_case_study
83  ASSIGNMENT (HIDDEN)
84  x&0#6 == 253
85
86  Thread 0 file MyContract_case_study.sol line 1 function func_case_study
87  ASSIGNMENT (HIDDEN)
88  sum?1!0&0#3 == (goto_symex::guard?0!0&0#1 ? 7 : 254)
89
90  Thread 0 file MyContract_case_study.sol line 1 function func_case_study
91  ASSERT
92  execution_statet::\guard_exec?0!0 => (signed int)sum?1!0&0#3 > 100
93  assertion
```

Figure 20: SSA form of function_case_study shown in Figure 15.

The SSA form is shown in Figure 20. The SSA trace generated by the symbolic execution engine of ESBMC is shown on the right. The manually simplified SSA is shown on the left. Each statement is underlined and colour coded to show the correspondence. Compared to the simplified SSA, the *SymEx*-generated SSA trace has the following features:

- **Naming Convention.**
  The name of each indexed variable is shown in the front, and the index is shown at the end. E.g., "*x&0#4*" where "*x*" denotes the variable name, and "#4" denotes the index in SSA form.

- **More intermediate assignments.**
  The *SymEx-generated* SSA trace contains more intermediate assignments because of function call. It seems that these intermediate steps could be simplified.

- **Guarded GOTO predicate.**
  This is represented by "*goto_symex::guard*". (Line 60 in Figure 20)

- **Assertion.**
  The assertion is shown with a prefix "*execution_statet::".* (Line 92 in Figure 20)

A simplified SSA trace can be printed using the option *"--ssa-smt-trace*". As shown in Figure 21, when generating the logic formulae, ESBMC uses a simplified *SymEx-generated* SSA trace.

```
1  Generated 1 VCC(s), 1 remaining after simplification (5 assignments)
2  Encoding remaining VCC(s) using bit-vector/floating-point arithmetic
3  Thread 0
4  ASSIGNMENT (HIDDEN)
5  func_case_study::$tmp::return_value$_nondet$1?1!0&0#1 == i?1!0&0#0
6
7  Thread 0 file MyContract_case_study.sol line 1 function func_case_study
8  ASSIGNMENT ()
9  y?1!0&0#1 == func_case_study::$tmp::return_value$_nondet$1?1!0&0#1
10
11 Thread 0 file MyContract_case_study.sol line 1 function func_case_study
12 ASSIGNMENT (HIDDEN)
13 goto_symex::guard?0!0&0#1 == (signed int)y?1!0&0#1 > 253
14
15 Thread 0 file MyContract_case_study.sol line 1 function func_case_study
16 ASSIGNMENT (HIDDEN)
17 sum?1!0&0#3 == (goto_symex::guard?0!0&0#1 ? 7 : 254)
18
19 Thread 0 file MyContract_case_study.sol line 1 function func_case_study
20 ASSERT
21 execution_statet::\guard_exec?0!0 => (signed int)sum?1!0&0#3 > 100
22 assertion
23
24 Encoding to solver time: 0.001s
25 Solving with solver Z3 v4.8.10
26 Encoding to solver time: 0.001s
27 Runtime decision procedure: 0.002s
28
```

Figure 21: Simplified SSA trace during SMT encoding.

3.1.3 Logic Formulae and Z3 Representation

```
1   C = [
2     y1 = nd_uchar   /\
3     x1 = 253        /\
4     g1 = y1 > x1    /\
5     sum1 = 253 + 10 /\
6     sum2 = 253 + 1  /\
7     sum3 = ite(g1, sum1, sum2)
8   ]
9
10  p = [ sum3 > 100 ]
11
12
```

Figure 22: C and P formulae.

To use the Z3 solver, ESBMC must extract the logic formulae from the SSA form using Z3 syntax. The C and P formulae that are manually derived are shown in Figure 22. ESBMC-generated formulae are shown in Figure 23.

```
58  (assert (= |sol:MyContract_case_study.solast@445@F@nondet@i?1!0&0#0|
59     |sol:@F@func_case_study::$tmp::return_value$_nondet$1?1!0&0#1|))
60  (assert (= |sol:@F@func_case_study::$tmp::return_value$_nondet$1?1!0&0#1|
61     |sol:MyContract_case_study.solast@445@F@func_case_study@y?1!0&0#1|))
62  (assert (= (bvsgt ((_ zero_extend 24)
63              |sol:MyContract_case_study.solast@445@F@func_case_study@y?1!0&0#1|)
64            #x000000fd)
65     |goto_symex::guard?0!0&0#1|))
66  (assert (= (ite |goto_symex::guard?0!0&0#1| #x07 #xfe)
67     |sol:MyContract_case_study.solast@445@F@func_case_study@sum?1!0&0#3|))
68  (assert (let ((a!1 (=> true
69                 (=> |execution_statet::\\guard_exec?0!0|
70                     (bvsgt ((_ zero_extend 24)
71                             |sol:MyContract_case_study.solast@445@F@func_case_study@sum?1!0&0#3|)
72                           #x00000064)))))
73    (not a!1)))
```

Figure 23: ESBMC-generated C ∧ ~P formulae

Any string bounded by "**|…|**" denotes a variable. E.g. in line 65 of Figure 23, "|*goto_symex::guard?0!0&0#1*|" represents the guard "*g1*" as shown in line 4 of Figure 22.

Figure 23 shows the Z3 representation of the formulae in Figure 22. To understand the Z3 representation, we are going to walk through a list of Z3 syntax:

- ***assert* command.**
  (assert (EXPR)) means "assert *EXPR* is true".

- **"=" operator.**
  (= (EXPR_A) (EXPR_B)) means assigning *EXPR_B* to *EXPR_A*, where *EXPR* denotes a variable, a literal or a more complex expression that uses another Z3 command.

- ***bvsgt* command.**
  (bvsgt (EXPR_A) (EXPR_B)), the bit-vector signed greater-than, which returns true if "*EXPR_A > EXPR_B*". The default operand is 32 bits width.

- **(_ zero_extend 24) command.**
  This command represents zero extension with bit width 24. Since our example uses *uint8* (Figure 15), ESBMC needs to use *(_ BitVec 8)* to encode this data type. In order to match the default operand bit width of 32, this value has to be zero extended with an additional bit width of 24, i.e. pad 24 zeros to the front.

- ***not* command**.
  (not EXPR_A) means the negation of *EXPR_A.*

- ***ite* (if-then-else) command**.
  (ite (EXPR_A) (VAL_1) (VAL_2)) means "if *EXPR_A* is true, then returns *VAL_1*, else return *VAL_2*".

- **"=>" (implication) command**.
  "=>" denotes implication. *(=> (EXPR_A) (EXPR_B))* means "*EXPR_A* → *EXPR_B*".

- ***let* command**.
  (let (IDENTIFIER_A (EXPR_A)) (IDENTIFIER_B (EXPR_B)) ...) means
  "let IDENTIFIER A denotes EXPR_A, and let IDENTIFIER B denotes EXPR_B".
  The user may declare more identifiers if needed.

The list above explains all the Z3 commands that appear in Figure 23, which enables us to map the C and P formulae to the corresponding Z3 representations. This mapping is shown in Figure 24.

Note that Z3 representation is a simplified version of the C/\~P formulae. The clause "*x1 = 253*" is simplified to a hex constant shown in line 64 in Figure 24. "*#x00000064"* represents *253* in decimal. The same simplification approach also applies to the clauses "*sum1 = 253 + 10*" and "*sum2 = 253 + 1*" as shown in line 66 in Figure 24: the former is simplified to a hex constant *"#x07*" representing 7 in decimal and the latter is simplified to a hex constant "*#xfe*" representing 254. The value of *sum1* should be *263*, but it wraps around and finally becomes *7* due to the arithmetic overflow error. The verification result in Figure 25 shows that Z3 found satisfiability with respect to the formulae in Figure 22. ESBMC reported a counterexample that satisfies the negation of the property "*p = [sum3 > 100]*". The counterexample indicates the presence of the arithmetic overflow error.

```
1   C = [
2     y1 = nd_uchar   /\
3     x1 = 253        /\
4     g1 = y1 > x1    /\
5     sum1 = 253 + 10 /\
6     sum2 = 253 + 1  /\
7     sum3 = ite(g1, sum1, sum2)
8   ]
9
10  p = [ sum3 > 100 ]
11
12
```

```
58  (assert (= |sol:MyContract_case_study.solast@445@F@nondet@i?1!0&0#0|
59     |sol:@F@func_case_study::$tmp::return_value$_nondet$1?1!0&0#1|))
60  (assert (= |sol:@F@func_case_study::$tmp::return_value$_nondet$1?1!0&0#1|
61     |sol:MyContract_case_study.solast@445@F@func_case_study@y?1!0&0#1|))
62  (assert (= (bvsgt ((_ zero_extend 24)
63              |sol:MyContract_case_study.solast@445@F@func_case_study@y?1!0&0#1|)
64           #x000000fd)
65     |goto_symex::guard?0!0&0#1|))
66  (assert (= (ite |goto_symex::guard?0!0&0#1| #x07 #xfe)
67     |sol:MyContract_case_study.solast@445@F@func_case_study@sum?1!0&0#3|))
68  (assert (let ((a!1 (=> true
69                 (=> |execution_statet::\\guard_exec?0!0|
70                     (bvsgt ((_ zero_extend 24)
71                             |sol:MyContract_case_study.solast@445@F@func_case_study@sum?1!0&0#3|)
72                            #x00000064)))))
73    (not a!1)))
```

Figure 24: Z3-representations of the formulae

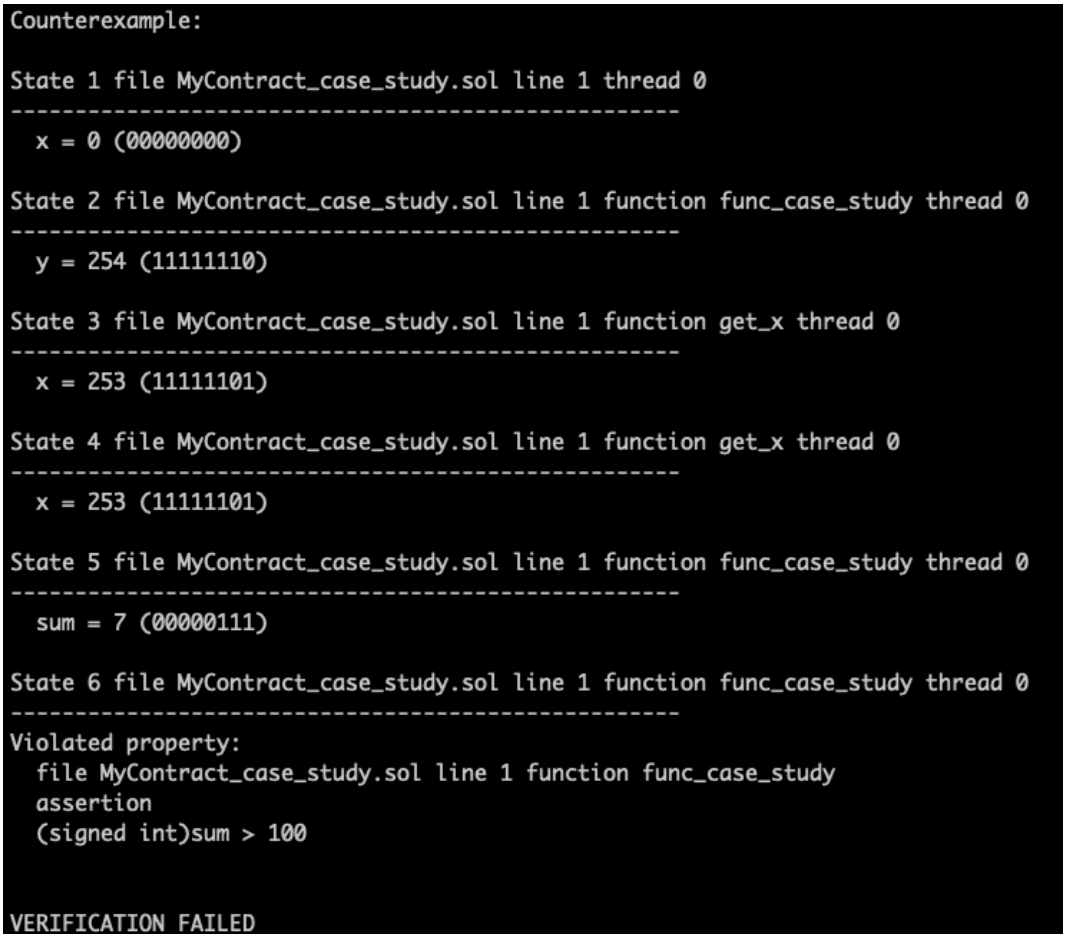


Figure 25: Verification result of the illustrative example.

3.2 Tracker-Based Hybrid Conversion

This section aims to explain the reason why the methodology that relies on Solidity JSON AST was chosen, as well as the implementation of such methodology in ESBMC. This section starts with a description of generalised frontend actions in ESBMC to verify general programming language. Given an input Solidity program, our goal is to generate the symbol table using ESBMC's internal data structure *symbolt*, which is used in a later stage to generate the GOTO program. This section describes the design challenges and outlines a new methodology to resolve these challenges.

3.2.1 Generalised Frontend Actions

As described in Section 2.2, when verifying programs written in a general programming language, ESBMC frontend actions can be generalised as follow:

- **Pre-processing (pre-processor)**
  For C and C++, the purpose of this step is to perform speicfic manipulations based on the preprocessor directives, e.g. substitute or expand macros or removing a code block if it is bounded by the directives '#if 0' and '#endif' [49].

- **Lexical analysis (scanner)**
  The step aims to understand the "word" of the C or C++ source code. The scanner groups the characters into lexemes and generate a sequence of tokens [21].

- **Syntax analysis (parser)**
  The purpose of this step is to understand the structure of the input C or C++ source code. The parser usually generates the AST to diagram the source code.

- **Type checking (type checker)**
  The aim of this step is to convert each AST node into ESBMC's intermediate representation *irept* and generate a symbol table in which each symbol is represented by ESBMC's *symbolt* data structure.

Action #1, #2 and **#3** are typical compiler phases. In ESBMC, these steps are handled by the clang APIs. ESBMC has its type checker for step #4.

These actions lead to the creation of a symbol table. In ESBMC, the symbol table enables the middle end to perform further actions, including:

- Convert the original program into the equivalent GOTO program
- Symbolically execute the program and generate the SSA form

3.2.2 Design Challenges and Decisions

Unlike general programming languages, Solidity is a domain-specific language (DSL). A DSL is a programming language for a specific field, and it is designed so that the users can be particularly productive in that field [22]. Solidity is a DSL with OOP features that are tailored to smart contracts of the Ethereum blockchain.

Compared with C and C++ languages heavily used in the industry for many years, Solidity is a relatively new language. To verify a programming language, there are two items to be considers:

- The language standard
- The toolchain (e.g. compiler, linker and debugger)

The language standard provides a thorough and detailed description of the lexical convention, formal grammar, and the production rules, e.g. C++11 standard [50] and C99 standard [51]. The toolchain provides libraries and APIs that allow the developers to design a language verification tool. For C and C++, such libraries and APIS are provided by the clang compiler suite. Additionally, there exists plenty of publications and online resources to assist the developers in designing a new language verification tool based on the existing tools, e.g. the tools in [45, 47, 48]. However, this is not the case for Solidity. Due to the lack of, designing a new ESBMC frontend to verify smart contracts is a challenging task. The design challenges are outlined as follows:

Design Challenge #1. Unlike C or C++, there is no officially published document of the Solidity language standard. The only language documentation available is [16]. There is not enough information to implement a scanner and a parser from scratch.

Design Challenge #2. Apart from the Github repository of the Solidity compiler[1] (*solc*), there are no officially published books or papers to help the developers to use the libraries and APIs provided by Solidity compilers[2]. To use these libraries and APIs, one has to become a *solc* expert.

Design challenge #3. Solidity is a relatively new DSL that keeps evolving. Based on Solidity's history, there were many breaking changes between two major versions [16]. These changes are not backwards compatible. For example, it is impossible to use *solc* version 0.4.20 to compile a Solidity program containing features of *solc* version 0.X.Y, where X > 4. The breaking changes are of different types:

- Syntax-only changes
- Semantic-only changes
- Semantic and Syntax changes
- Deprecated elements
- New features
- Explicitness requirements, e.g. mark a function ‘virtual’ explicitly if it is defined outside an interface without implementation (it is legal to do so in Solidity)
- Interface changes (including the changes in JSON AST)

[1] Solidity compiler (solc): https://github.com/ethereum/solidity
[2] Having looked at the source code repository and the build directory of *solc,* there seem to exist some kind of libraries and the corresponding *include* files, e.g. *liblangutil, libsolc, libsolidity* .etc.

The design goals are as following:

Design Goal #1. The new Solidity frontend of ESBMC shall complete lexical analysis and syntax analysis of the input Solidity source code and generate the AST.

Design Goal #2. The new Solidity frontend of ESBMC shall complete the type checking of the AST in **Design Goal #1**. The frontend shall transform the Solidity AST nodes into the equivalent ESBMC *irept* nodes whilst preserving the semantic information, and generate the corresponding ESBMC symbols modelled by *symbolt* class.

There are two methods to achieve these goals:

Methodology #1. Use Solidity source code as input. This frontend uses the libraries and *include* files provided by *solc*. This methodology requires the integration of ESBMC with *solc* libraries. The conversion functions of the new type checker rely on the *solc* libraries.

Methodology #2. Use JSON representation of Solidity AST as input. The JSON representation of Solidity AST can be generated using the Solidity compiler option *"--ast-compact-json*". Then the new frontend needs to handle a JSON file. The conversion functions of the new type checker is based on the high-level language constructs of Solidity.

Both methodologies can achieve **Design Goal #1** and **#2**. Both methodologies obviate the need to implement a scanner and a parser, and hence both can resolve **Design Challenge #1**. However, **Design Challenge #2** and **#3** cannot be resolved using **Methodology #1**. For this reason, we decided to go for **Methodology #2** to implement a new Solidity frontend that processes the JSON representation of Solidity AST.

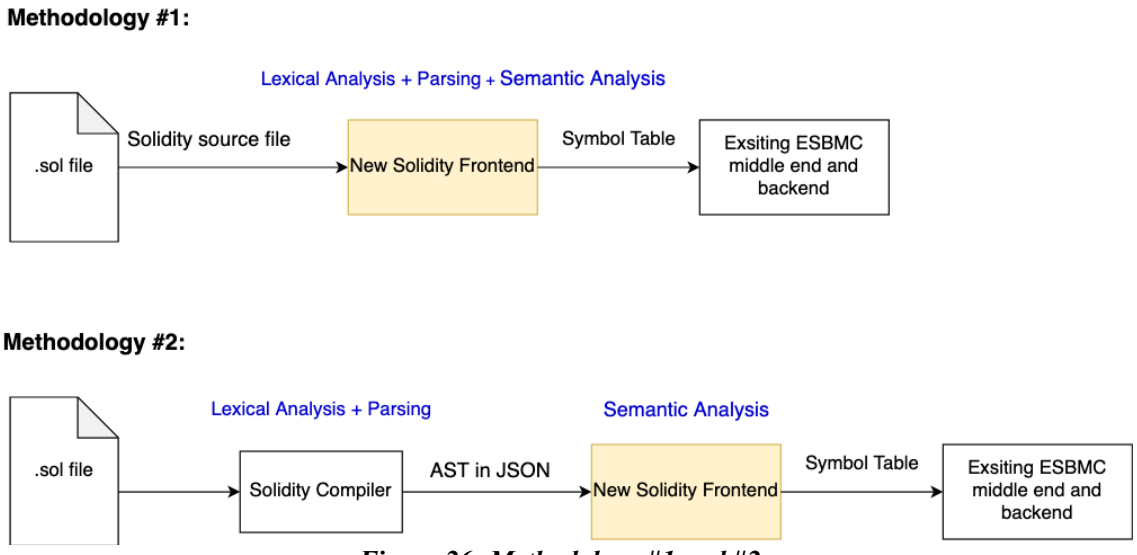


Figure 26: Methodology #1 and #2.

As shown in Figure 26, the lexical analysis and parsing phases are "outsourced" to the *solc* (Solidity compiler), which just leaves the semantic analysis to be implemented. But the new frontend of Methodology #1 has to handle all three phases. Due to the lack of documentation of Solidity compiler, the programmable interfaces provided by *solc* are challenging to use.

The disadvantages of **Methodology #1** are as follows:

- As described by **Design Challenge #2**, due to the lack of publications and learning resources of *solc* toolchain, one must become a *solc* expert to use the libraries and *include* files provided by *solc*. Considering the time limit of this MSc project, becoming a *solc* expert is a mission impossible. Having investigated the source code and *include* files, using these libraries is as difficult as developing the new frontend itself.

- Since Solidity keeps evolving (cf. **Design Challenge #3**), the dependency of the new type checker on *solc* libraries is likely to break if there are corresponding changes in those libraries. Maintaining such dependency will become a Herculean task because we may need to worry about almost all types of breaking changes as listed in **Design Challenge #3,** which may change how a programmable interface is used.

As for **Design Challenge #2** and **#3**, the advantages of **Methodology #2** are as follow:

- **Methodology #2** does not use *solc* libraries. Therefore, **Design Challenge #2** is resolved.

- **Methodology #2** uses Solidity JSON AST as input. It is less sensitive to the breaking changes because AST is a tree-structured IR representing the syntactic structure of a source program [21]. Because we only need to worry about the structural changes with respect to the JSON AST. Based on the historical records of Solidity breaking changes, the frequency of such changes is very low. It only occurs once between version 0.5.0 (released in November 2018) and version 0.8.7 (released in July 2021). Therefore, **Methodology #2** is more robust to the breaking changes in as described in **Design Challenge #3.**

Table 7 compares **Methodology #2** to **Methodology #1** with respect to the coverage of **Design Goals** and **Design Challenges**. Table 8 compares these methodologies with respect to the coverage of the frontend actions as described in Section 3.2.1.

Methodologies	Design Goal #1	Design Goal #2	Design Challenge #1	Design Challenge #2	Design Challenge #3
Methodology #1	✓	✓	✓	✗	✗
Methodology #2	✓	✓	✓	✓	✓

Table 7: Coverage of Design Goals and Design Challenges

Methodologies	Preprocessing	Lexical Analysis	Syntax Analysis	Type Checking
Methodology #1	N/A to Solidity	✗	✗	Expected a new type checker to be implemented in both methodologies
Methodology #2		✓ (bypassed)	✓ (bypassed)	

Table 8: Coverage of frontend actions.

The lexical and syntax analysis phases can be bypassed in **Methodology #2** because these phases are handled by the *solc* compiler whilst generating the Solidity JSON AST.

As a short summary of this subsection, **Methodology #2** gives better coverage of the **Design challenges** shown in Table 7. It also obviates the need to become a *solc* expert to use the APIs and libraries.

Methodology #2 leads us to a roadmap that contains the implementation milestones as follow:

- **Milestone #1**. **[Implementation of the New Language Mode]**
  Before adding the new Solidity frontend, ESBMC used to supported C and C++ only. A new language mode must be added to support Solidity.

- **Milestone #2**. **[Implementation of the New Type Checker]**
  The new frontend takes the JSON-represent of Solidity AST. A new type checker is required to work with this format.

- **Milestone #3**. **[Add support for ESBMC and SV-COMP Variables and Function]**
  The new frontend needs to support all ESBMC and SV-COMP variables and functions.

Each subsection describes a detailed solution to achieve each milestone as listed above.

3.2.3 Solidity as A New Language Mode in ESBMC

To achieve **Milestone #1,** ESBMC must be extended to support Solidity as a new language mode.

The list below summarised all the modifications in ESBMC to support a new language. This list can also be referenced by other developers to facilitate future extensions, which hopefully save a developer's time during the project ramp-up phase.

- Add a new *enum* entry of the language to be supported in the *mode_table* initialised in *src/esbmc/globals.cpp.*

- Define a new extension in *src/langapi/mode.cpp* to let ESBMC know about the extension of the source file name.

- Define a new macro *LANGAPI_HAVE_MODE_X* in *src/langapi/mode.h,* where *X* denotes the name of the language to be supported.

- Add the new frontend placeholder (usually a new directory) under the source directory, e.g. "*src/solidity-frontend/<source files of the new frontend>*" was added to support the new Solidity mode that was added in the above steps.

- Add the corresponding directives in the *CMakeList.txt* at various level of the source code repository to build the new frontend.

Since ESBMC is well-structured, achieving **Milesonte #1** is relatively easy compared to the other milestones. The above list covers most, if not all, major modifications to support a new language mode in ESBMC.

3.2.4 Tracker-Based Conversion

The new Solidity frontend takes JSON-representation of the AST as input. To generate the symbol table, a new type checker must transform each AST JSON node into an *irept* node and convert the *irept* node into the corresponding *symbolt*, a data structure representing a symbol in ESBMC.

There are various third-party libraries available to work with JSON files in C++. The most popular one is *nlohmann/json* library[3] developed by Niels Lohmann. This library has been used by many tools[4], including *American fuzzy lop*, *CMake*, *Doxygen*, *Valgrind*, and *Clang* that is used by ESBMC's clang-based C frontend.

There are three challenges with respect to the implementation of the conversion process to transform JSON-representation of an AST node to ESBMC *irept* node:

- **Conversion Challenge #1**.
  To convert each AST node, the new frontend needs to traverse the Solidity AST. AST is a tree structure. In clang-based frontend, this tree structure is well preserved by clang and can be traversed using the APIs provided by clang. However, the input JSON file is flat. Each AST node is a JSON object that just contains key-value pairs.

- **Conversion Challenge #2**.
  The new type checker must have common conversion functions to process the child nodes that hold the semantic information of a parent declaration node. (cf. Table 3)

- **Conversion Challenge #3**.
  When processing different types in Solidity, the conversion functions must be able to switch between different types of Solidity language constructs.

An illustrative example of **Conversion Challenge #1** is shown in Figure 27 and Figure 28. Take the expression "*sum = get_x() + y + z*" as an example, this expression contains nested binary operation expressions with a function call expression shown in Figure 27. The json representation of such recursion is shown in Figure 28. The left-hand-side expression is parsed as "*((get_x() + y) + z)*". Table 9 summarizes the solutions to resolve all **Conversion Challenges.**

[3] *nlohmann/json* library: https://github.com/nlohmann/json
[4] A list of tools that use *nlohmann/json* library can be found at https://github.com/nlohmann/json#used-third-party-tools

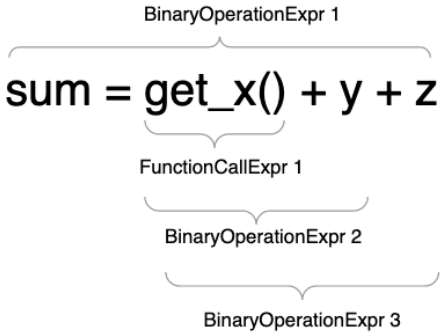


Figure 27: Recursion - nested BinaryOperation Expressions.

Conversion Challenges	**Solutions**	**Remarks**
#1	Introduce a common data structure *tracker* to re-construct the tree	Implemented in *solidty_decl_tracker.cpp*[5]
#2	Introduce a common data structure for each type of child nodes , *SourceLocationTracker*, *NamedDeclTracker*, and *QualTypeTracker*	
#3	Introduce the type conversion functions	Implemented in *solidity_type.cpp*[6]

Table 9: Solutions to design challenges

[5] The source code of *solidity_decl_tracker*.cpp is available at https://github.com/kunjsong01/esbmc/blob/a863663bc9c3ba4c7d219cb014483170f75fcd8d/src/solidity-ast-frontend/solidity_decl_tracker.cpp

[6] The source code of *solidity_type.cpp* is available at https://github.com/kunjsong01/esbmc/blob/a863663bc9c3ba4c7d219cb014483170f75fcd8d/src/solidity-ast-frontend/solidity_type.cpp

Figure 28: JSON AST of a nested BinOpExpr.

To resolve all the **Conversion Challenges**, a *tracker-based conversion method* was proposed to transform the AST JSON nodes into the equivalent *irept* nodes.

When traversing the AST node in the input JSON file, the **Tracker-based Conversion Method** uses a common data structure called *trackers* to represent each AST node. This method uses different types of trackers to model different types of child nodes shown in Figure 29, for example:

- *NamedDeclTracker* to hold the name information
- *SourceLocationTracker* to hold the location information
- *QualTypeTracker* to hold the type information

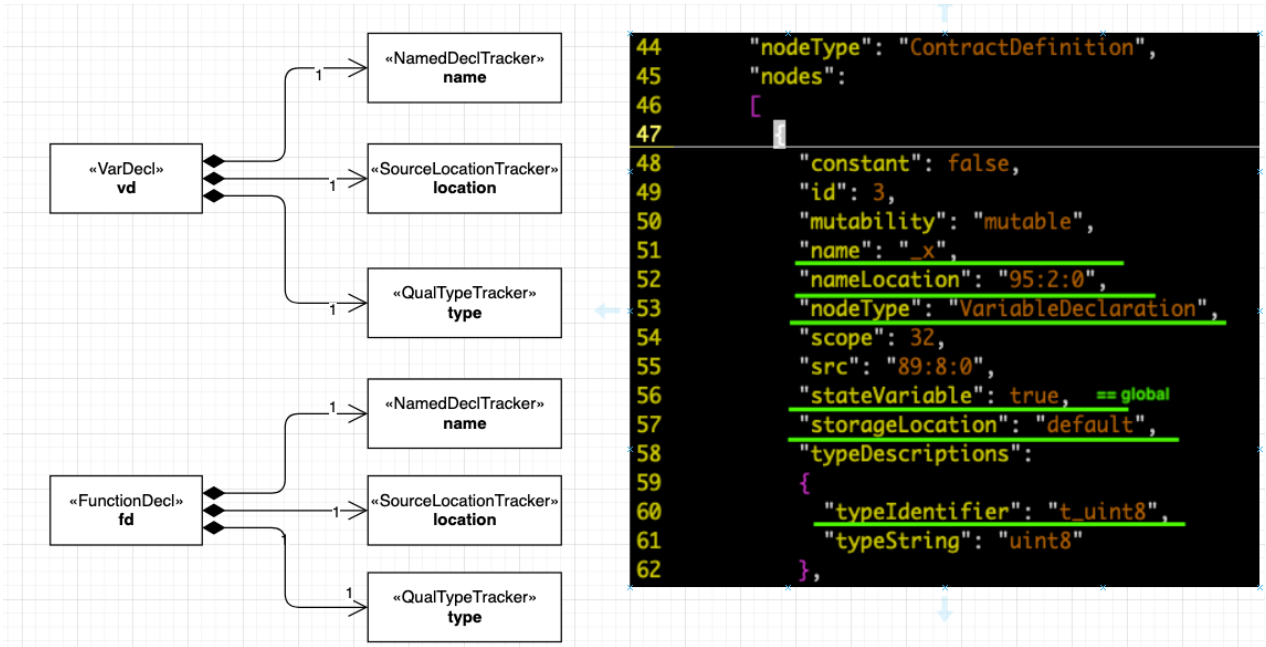


Figure 29: Trackers.

Figure 29 shows different types of semantic trackers to track the information when traversing the AST node in the input JSON file. Since the new frontend uses a *for* loop to iterate over the *nodes* array in the input JSON file, the corresponding trackers are instantiated on the fly. When the loop reaches the end of the array, the tree structure will be re-constructed. In a sense, it "tracks" the progress as the loop moves from one node to another.

As shown in Figure 29, the *Tracker-based Conversion Method* also uses different types of declaration trackers to represent different types of the declaration nodes in Solidity AST, for example:

- A base *DeclTracker* class to track declaration node with the following derived classes:
  - *VarDeclTracker* represents a variable declaration node.
  - *FunctionDeclTracker* represents a function declaration node.

- A base *StmtTracker* class to track the statement node with the following derived classes:
  - *CompoundStmtTracker* represents a block of statements.
  - *DeclRefExprTracker* represents a statement node being an expression of declared variable.
  - *BinaryOperatorTracker* represents a statement node being an expression of binary operation
  - *CallExprTracker* represents a statement node being an expression of function call.

Let us use another code example shown in Figure 30. The composition and inheritance relations are shown in Figure 31.

```
 9     function func_example() external
10     {
11       _x = 100;
12       _y = 240;
13       assert(_x == 100);
14     }
```

Figure 30: Example of a Solidity function.

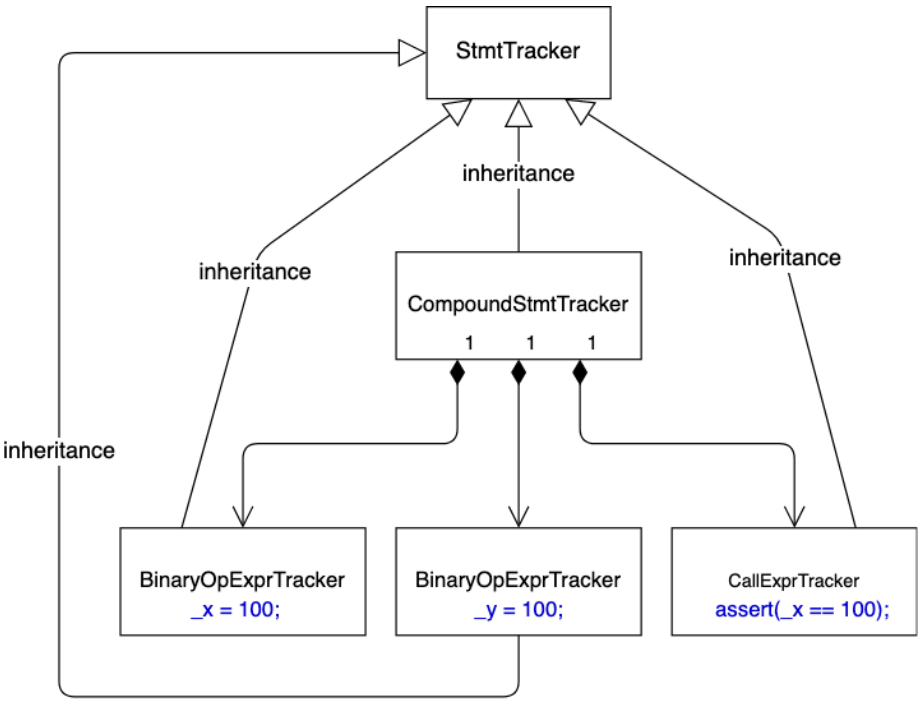


Figure 31: AST of the function body.

In Figure 31 Each node in this tree structure is represented by a tracker as the new frontend iterates over the "nodes" array in the input JSON file. All expression trackers are derived classes of the base class StmtTracker. Since a statement node can be a block statement or an expression statement, and the expression statement can also be a block containing multiple statements, it can become challenging to deal with such recursions. This challenge can be resolved by the class inheritance hierarchy shown in Figure 31. The type checker can traverse each statement node by recursively calling the function *get_expr* shown in line 229 of Figure 32.

```
208  bool solidity_convertert::get_expr(const StmtTracker* stmt, exprt &new_expr)
209  {
210    static int call_expr_times = 0; // TODO: remove debug
211    locationt location;
212    get_start_location_from_stmt(stmt, location);
213    assert(stmt);
214
215    switch(stmt->get_stmt_class())
216    {
217      case SolidityTypes::stmtClass::CompoundStmtClass:
218      {
219        printf("  @@@ got Expr: SolidityTypes::stmtClass::CompoundStmtClass, ");
220        printf("  call_expr_times=%d\n", call_expr_times++);
221        const CompoundStmtTracker* compound_stmt =
222          static_cast<const CompoundStmtTracker*>(stmt); // pointer to const CompoundStmtTracker
223
224        code_blockt block;
225        unsigned ctr = 0;
226        for (const auto &stmt : compound_stmt->get_statements())
227        {
228          exprt statement;
229          if(get_expr(stmt, statement))
230            return true;
231
232          convert_expression_to_code(statement);
233          block.operands().push_back(statement);
234          ++ctr;
235        }
```

Figure 32: get_expr function.

First, *get_expr* function checks the type of the statement tracker. Then, depending on the type, the function converts the statement tracker into an *exprt* node (the second argument of *get_expr* function), where *exprt* implements the *irept* interface.

Suppose the tracker represents a compound statement tracker. In that case, it will be statically casted to a "*CompoundStmtTracker*" and recursively calls the *get_expr* function to convert each individual statement into the equivalent *exprt* node. Figure 33 shows two more examples to convert binary operation tracker and declaration reference tracker. The conversion of binary operation trackers will be handled by *get_binary_operator_expr* function (Figure 34). The *get_binary_operator_expr* function calls back into *get_expr* function when converting the LHS and RHS expressions. When converting a binary operator expression, the call stack is shown in Figure 35.

```
250     // Binary expression such as a+1, a-1 and assignments
251     case SolidityTypes::stmtClass::BinaryOperatorClass:
252     {
253       printf("  @@@ got Expr: SolidityTypes::stmtClass::BinaryOperatorClass, ");
254       printf("  call_expr_times=%d\n", call_expr_times++);
255       const BinaryOperatorTracker* binop =
256         static_cast<const BinaryOperatorTracker*>(stmt); // pointer to const CompoundStmtTracker
257
258       if(get_binary_operator_expr(binop, new_expr))
259         return true;
260
261       break;
262     }
263
264     // Reference to a declared object, such as functions or variables
265     case SolidityTypes::stmtClass::DeclRefExprClass:
266     {
267       printf("  @@@ got Expr: SolidityTypes::stmtClass::DeclRefExprClass, ");
268       printf("  call_expr_times=%d\n", call_expr_times++);
269
270       const DeclRefExprTracker* decl =
271         static_cast<const DeclRefExprTracker*>(stmt);
272
273       // associate previous VarDecl AST node with this DeclRefExpr
274       // In order to get the referenced declaration, we want two key information: name and type.
275       // We need to do the followings to achieve this goal:
276       //  1. find the associated AST node json object
277       //  2. use that json object to populate NamedDeclTracker and QualTypeTracker
278       //     of this DeclRefExprTracker
279       assert(decl->get_decl_ref_id() != DeclRefExprTracker::declRefIdInvalid);
280       assert(decl->get_decl_ref_kind() != SolidityTypes::declRefError);
281
282       if(get_decl_ref(decl, new_expr))
283         return true;
284
285       break;
286     }
```

Figure 33: Conversion of BinOpStmt and DeclRefExor.

```
461   bool solidity_convertert::get_binary_operator_expr(
462     const BinaryOperatorTracker* binop,
463     exprt &new_expr)
464   {
465     exprt lhs;
466     if(get_expr(binop->get_LHS(), lhs))
467       return true;
468
469     exprt rhs;
470     if(get_expr(binop->get_RHS(), rhs))
471       return true;
472
473     typet t;
474     if(get_type(binop->get_qualtype_tracker(), t))
475       return true;
```

Figure 34: get_binary_operator_expr calls back into get_expr.

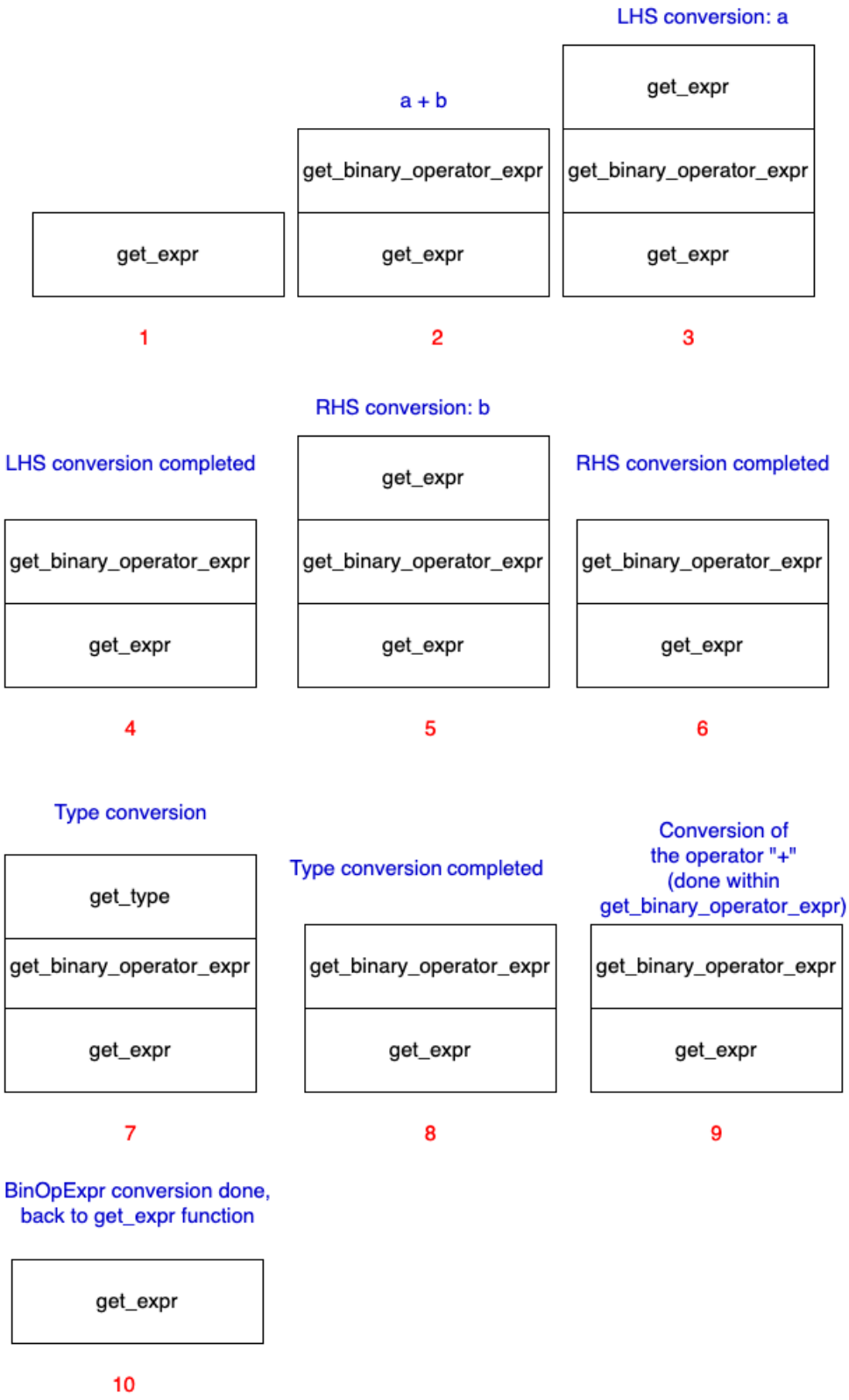


Figure 35: Call stack usage when converting "a+b"

As a concluding example of this subsection, let us look at a re-constructed tree of the Solidity function shown in Figure 36, and its conversion to ESBMC *irept*.

```
 9     function func_example2() external
10     {
11       sum = get_x() + y + z;
12     }
```

Figure 36: concluding example.

The re-constructed tree of the *function_example2* is shown in Figure 37. This tree is converted into ESBMC tree-structured intermediate representation shown in Figure 38.

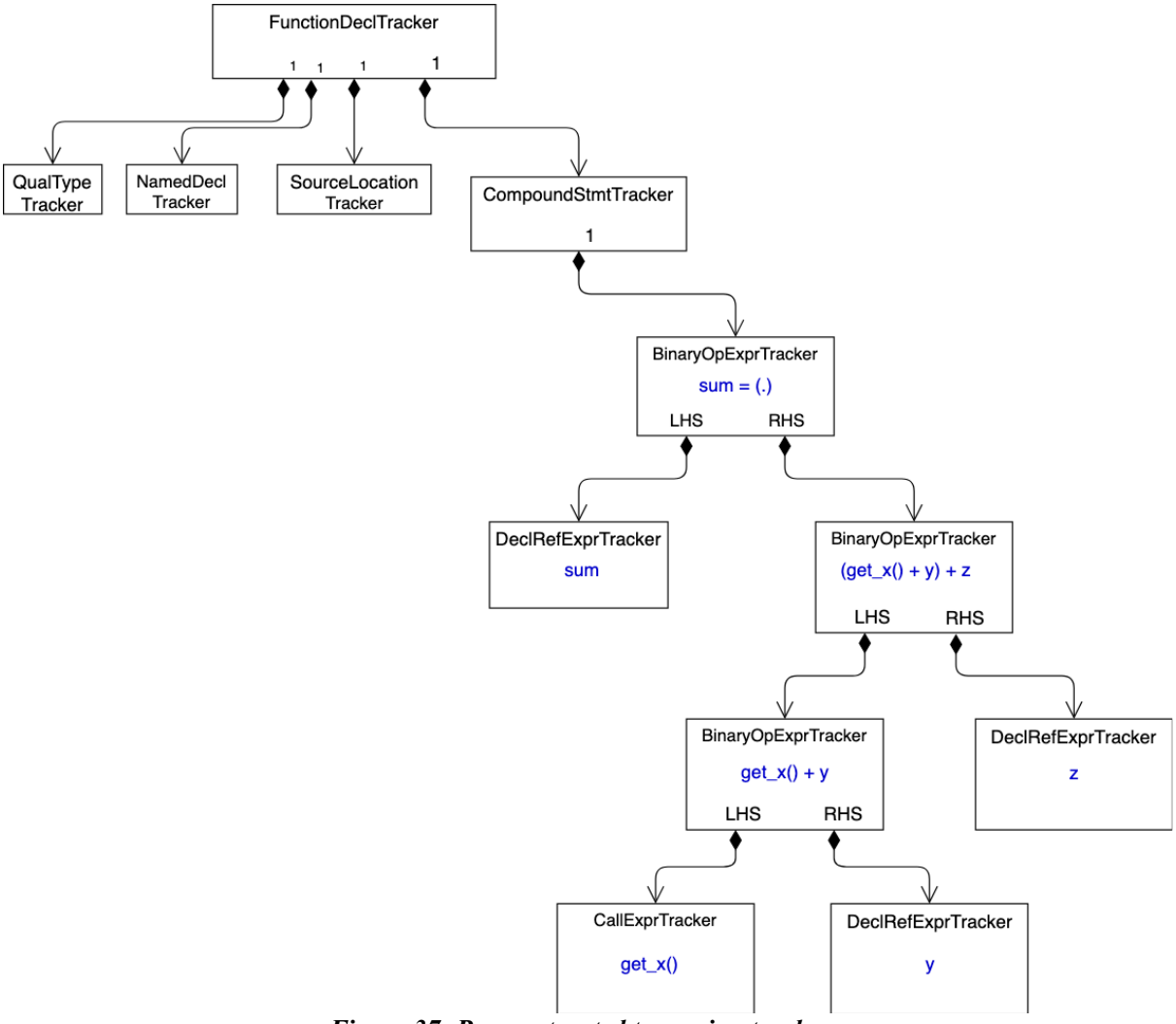


Figure 37: Re-constructed tree using trackers.

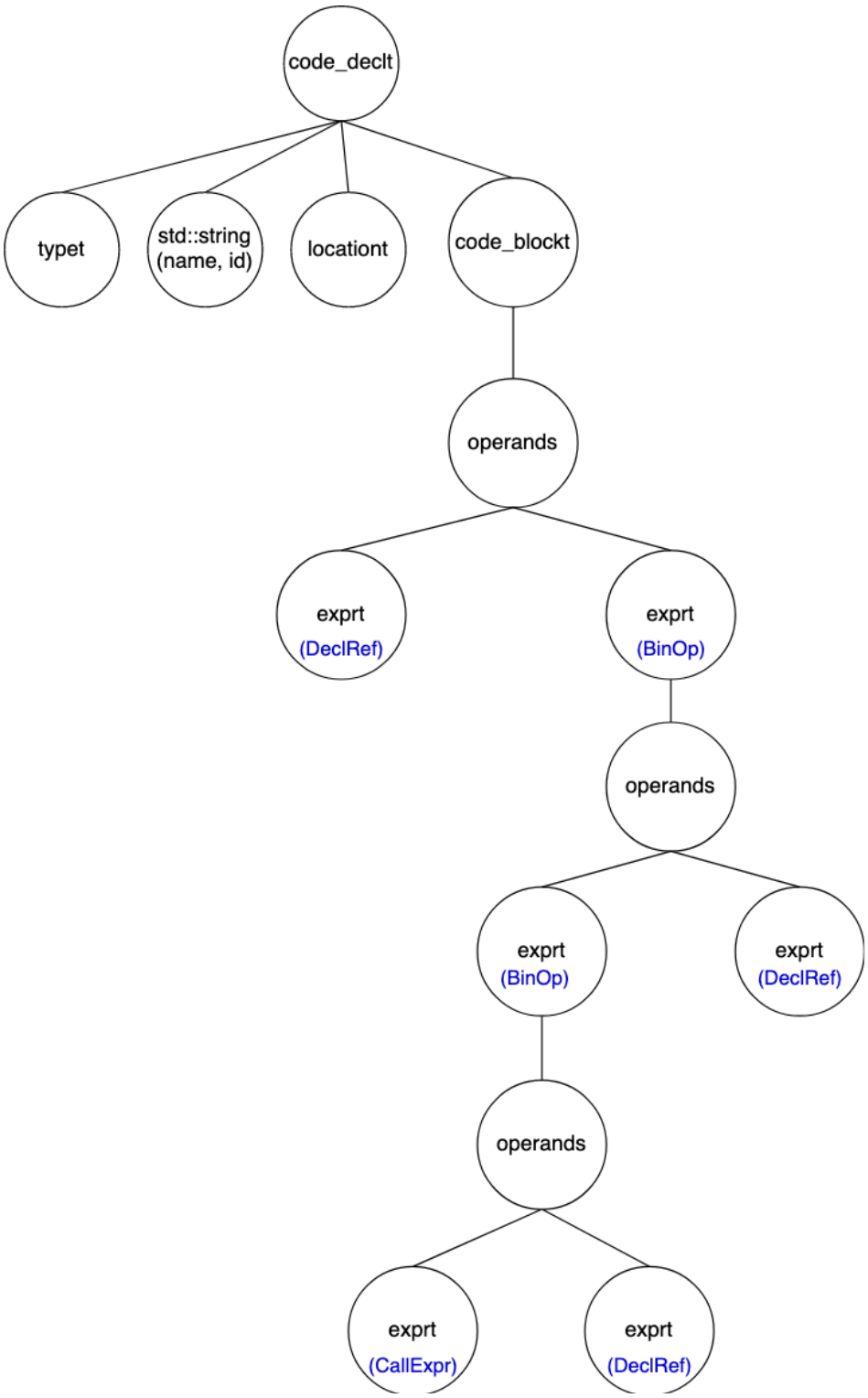


Figure 38: ESBMC irept parse tree..

After the conversion to *irept*, the corresponding *symbolt* can be generated using the standard function *get_default_symbol*.

3.2.5 Hybrid Symbol Conversion for Intrinsic Declarations

There are three important functionalities a BMC needs to support:

- ***assert()***. This function enables the user to define properties.
- ***assume()***. This function enables the user to define constraints.
- ***nondet()***. This function enables the user to assign a non-deterministic value to a variable.

To support these functionalities, the new frontend needs to support ESBMC/SV-COMP variables and functions shown in Figure 39. As described in Section 3.2.4, the tracker-based conversion mechanism only works with the JSON representation of the Solidity AST nodes. It does not work with C-style declarations.

```
281   std::string intrinsics =
282     R"(
283 # 1 "esbmc_intrinsics.h" 1
284 void __ESBMC_assume(_Bool);
285 void __ESBMC_assert(_Bool, const char *);
286 _Bool __ESBMC_same_object(const void *, const void *);
287 void __ESBMC_atomic_begin();
288 void __ESBMC_atomic_end();
289
290 int __ESBMC_abs(int);
291 long int __ESBMC_labs(long int);
292 long long int __ESBMC_llabs(long long int);
293
294 // pointers
295 unsigned __ESBMC_POINTER_OBJECT(const void *);
296 signed __ESBMC_POINTER_OFFSET(const void *);
297
298 // malloc
299 __attribute__((annotate("__ESBMC_inf_size")))
300 _Bool __ESBMC_alloc[1];
301
302 __attribute__((annotate("__ESBMC_inf_size")))
303 _Bool __ESBMC_deallocated[1];
304
305 __attribute__((annotate("__ESBMC_inf_size")))
306 _Bool __ESBMC_is_dynamic[1];
```

Figure 39: ESBMC intrinsic variable and function declarations.

To support these intrinsic declarations, the new Solidity frontend needs to convert them into symbols and add them to the symbol table. The final symbol table should contain not only the symbols of Solidity declarations but also the symbols of ESBMC intrinsic declarations:

$$symbol\ table = \{I_0 \dots I_n, S_0 \dots S_n\} \qquad (3.1)$$

where $I_0 \dots I_n$ represents the symbols of the intrinsic declarations and $S_0 \dots S_n$ represents the symbols of Solidity declarations.
There are two methods to generate the symbol table defined in (3.1):

- **Method #1**. Provide the JSON representations of these variables and functions. For example, the function declaration, "*void __ESBMC_assert(_Bool, const char *)*", can be converted into the equivalent JSON representation shown in Figure 40.

- **Method #2**. Use a hybrid conversion mechanism shown in Figure 41:
  - Use clang-c-frontend to convert ESBMC/SV-COMP declarations
  - Use solidity-frontend to convert Solidity declarations

```
{
  "name" : "__ESBMC_assert",
  "returns" : "void",
  "parameters" :
  [
    {
      "type" : "_Bool"
    },
    {
      "type" : "const char *"
    }
  ]
}
```

***Figure 40: JSON-representation of* __ESBMC_assert**

Since there are more than 70 intrinsic declarations, manually converting them into the equivalent JSON representation is a time-consuming task. If anything changes in the intrinsic declarations, the developers must change the corresponding JSON representations, which leads to more maintenance duties. Adding the JSON representations of the intrinsic declarations in the new frontend appears to be reinventing the wheel. Therefore, **Method #2** was chosen to guide the implementation of the new Solidity frontend.

Hybrid conversion mechanism. The new Solidity frontend contains an instantiation of clang-c-frontend that add the symbols of intrinsic declarations to the symbol table generated by the Solidity type checker. Figure 41 illustrates this mechanism. The new Solidity frontend contains an instantiation of clang-c-frontend to handle the symbol conversion of ESBMC intrinsic declarations.

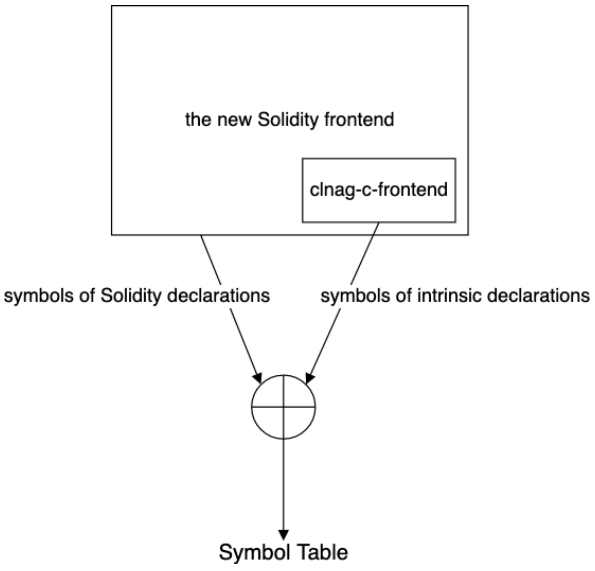


Figure 41: Hybrid conversion mechanism.

3.3 Limitations of Trackers

As described in Section 3.2.4, tracker is a data structure used to reconstruct the tree from the Solidity JSON AST nodes:

- Each node is represented by a tracker object.

- Each edge is represented by the composition relation between two tracker objects, e.g. A *CompoundStmtTracker* is not a *BinaryOpExprTracker*; it may have one. A *BinaryOpExprTracker* is not a *CallExprTracker* (i.e. function call); it may have a *CallExprTracker* as the LHS or RHS operand.

Trackers preserve the syntactic structure whilst holding the semantic information of each node in the original AST. However, trackers are not free to use. This section explains the limitations of the tracker-based symbol conversion.

3.3.1 Scalability

The new Solidity frontend takes the JSON-representation of the original AST as input. In order to re-construct the tree, the trackers essentially replicate the semantic information stored in each JSON object, i.e. the data is duplicated and stored in two places in the memory: the JSON objects and the tracker objects. The data replication is wasting the memory during the tree restoration phase.

For the verification of small Solidity programs, the impact of data replication is negligible. However, the tracker-based conversion mechanism may suffer from scalability and performance problems when it comes to verify large and complex Solidity programs. Because there are lots of recursions when converting a tracker node into *irept* node. Figure 42 shows an illustrative example of a nested binary operation expression "*sum = a + (b * (c * (...)))*".

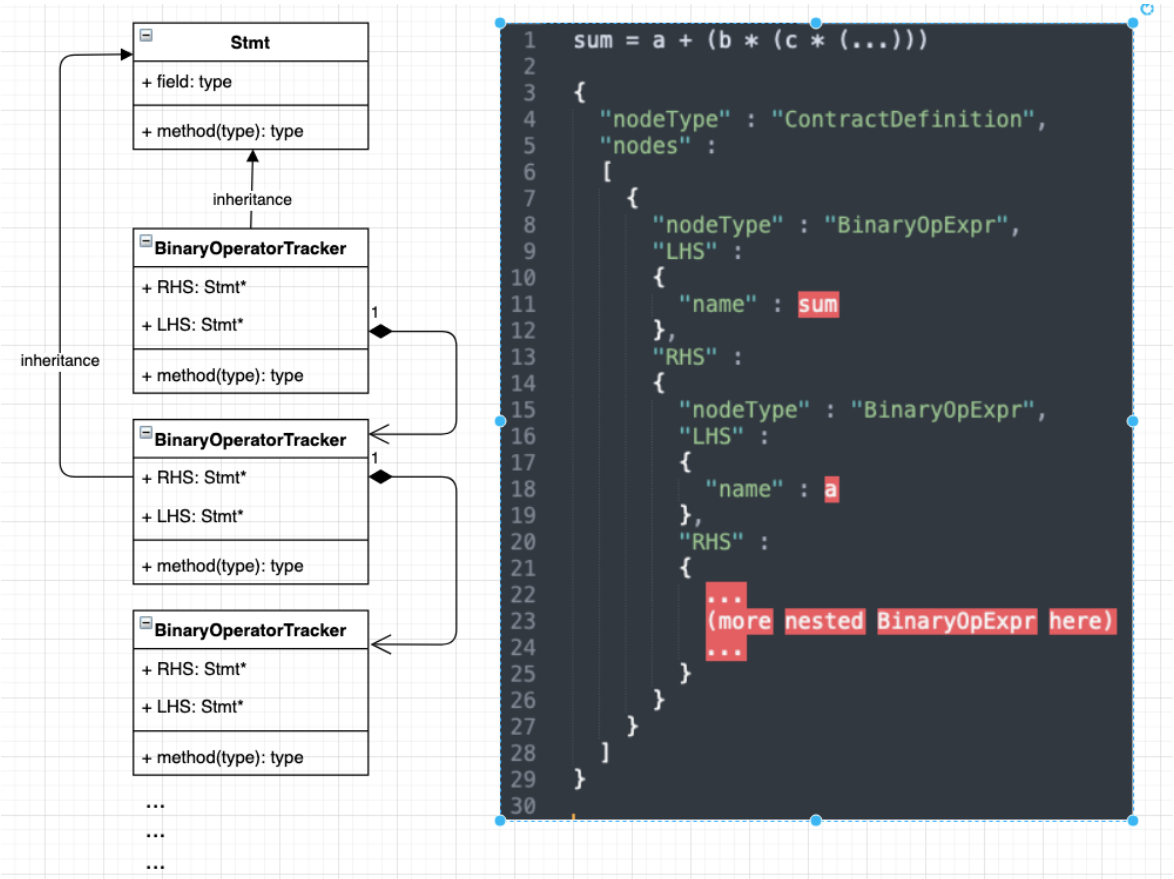


Figure 42: Trackers of nested BinOpExpr.

As the binary operator express grows, more tracker objects will be created to replicate the data of the corresponding JSON objects, which will waste more memory. It can become even worse when verifying multiple complex and large programs.

3.3.2 Maintainability, Extendibility and Readability

Since the first goal of the new Solidity frontend is to reconstruct the tree, the tracker class is a "helper" data structure. The developers would need to maintain and extend such data structure in the codebase. Since Solidity keeps evolving, here are the potential issues with respect to extendibility and maintainability:

- **Maintainability.**
  The structure of the tracker class may change according to changes of the corresponding JSON objects. However, frequency of such changes is quite low based on the analysis of the historical records of breaking changes as discussed in Section 3.2.2. To cope with such changes, the developers may need to update the base tracker class and the corresponding declaration class that derives the tracker class.

- **Extendibility.**
  If a new type of tracker is required, the developers may also need to update three places in the code:

  i. Add a new tracker class in *solidity_decl_tracker.h and solidity_decl_tracker.h*

  ii. Update the type files (*solidity_types.h* and *solidity_types.cpp*) to include the new type.

  iii. Add a new conversion function that converts this type of tracker into the *irept* node.

- **Readability.**
  The trackers hold the semantic information of the AST nodes, which facilitate the conversion of AST nodes into *irept* nodes. The conversion functions are developed based on the production rules of the formal grammar of the Solidity programming language [16]. Using the trackers makes the program more complex to a developer who does not know about the design rationale behind using the tracker data structure.

The tracker-based conversion mechanism gives rise to performance and scalability problems and requires more efforts when it comes to maintaining and extending the code base.

3.4 Grammar-Based Hybrid Conversion

A new conversion methodology was proposed to resolve the limitations of the tracker-based hybrid conversion methodology as previously mentioned. The new methodology is referred to as the *Grammar-Based Hybrid Conversion* that was implemented with improvement as follow:

- **The scalability problem is resolved**.
  Completely removed data replication due to the usage of the tracker data structure

- **Code readability has been improved**.
  The implementation of conversion functions reflects the production rules in formal grammar of the Solidity programming language, i.e. the Solidity grammar documentation serves as the design specification of the new type checker of the Solidity frontend in ESBMC. If a developer knows about Solidity grammar, the developer should be able to observe the mapping of a production rule to the corresponding conversion function.

- **Maintainability and extendibility have also been improved**.
  Reduced the number of files to be updated in case of a major update of the JSON structure.

This section starts by investigating the feasibility of tracker removal. The following subsections outline the design of grammar-based hybrid conversion.

3.4.1 Feasibility of Tracker Removal

As shown in Figure 43, recall that the new Solidity frontend reconstructs the tree whilst iterating over the elements of JSON array "*nodes*". The restored tree is formed of:

- Nodes represented by tracker objects that hold the semantic information

- Edges represented by the composition relations among different types of the tracker classes

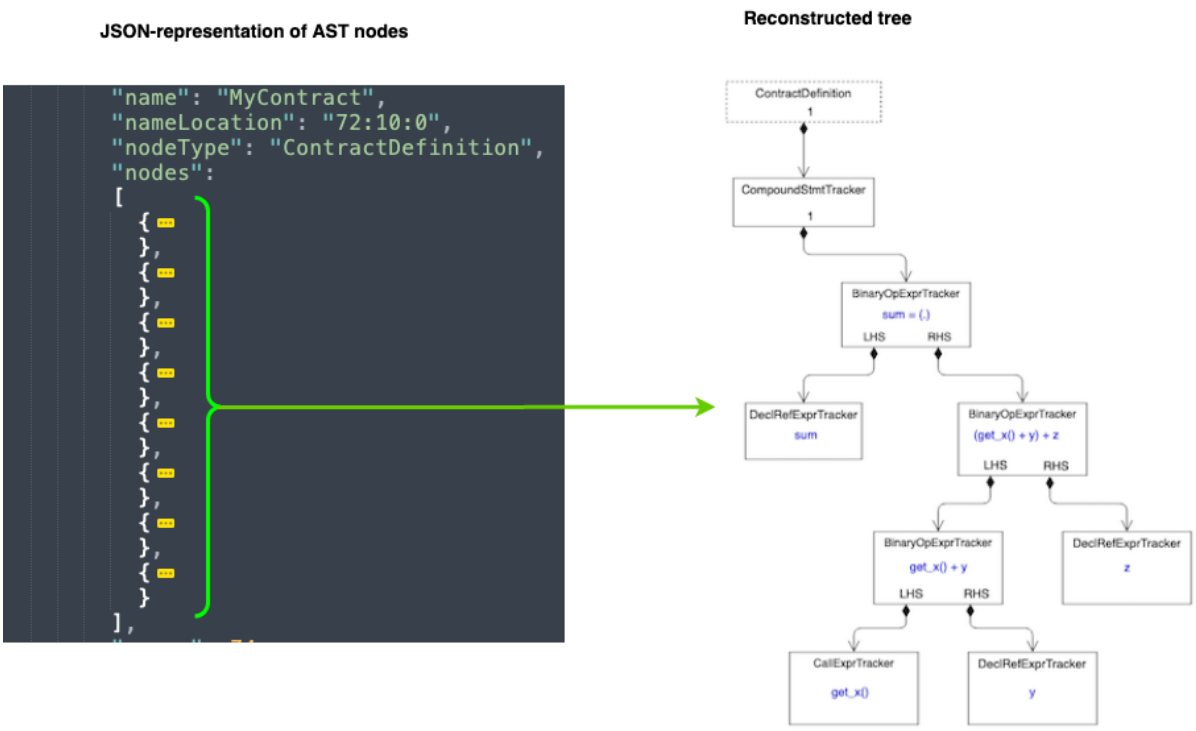


Figure 43: Tree re-constructed from the JSON AST.

In tracker-based conversion methodology, the conversion functions can be implemented so that each conversion function handles one type of tracker as shown in Table 10.

Original Node Types	**Conversion Functions**	***irept* Nodes**
VarDeclTracker	*get_var*	*code_declt*
FunctionDeclTracker	*get_function*	*code_declt*
QualTypeTracker	*get_type*	*typet*
NamedDeclTracker	*get_decl_name*	*irep_idt*
BinaryOperatorExprTracker	*get_binary_operator_expr*	*exprt*
SourceLocationTracker	*get_location_from_decl*	*location*
DeclRefExprTracker	*get_decl_ref*	*exprt*
StmtTracker (the base class of all other tracker classes)	*get_expr*	N/A

Table 10：Conversion functions for irept nodes.

Note that get_expr by itself does not convert any node, because it is the traversal function that walks through each node. Depending on the type of the tracker, get_expr calls other functions to perform the actual conversion.

To remove the trackers, the following conditions must be met:

- **Condition #1**.
  The semantic information of the AST JSON node must be preserved, which will be used to annotate *irept* nodes.

- **Condition #2**.
  The composition relation between a declaration node and a child node must be preserved.

- **Condition #3.**
  The traversal function *get_expr* can be reused in a recursive manner provided that the correct order of function calls can be preserved.

As shown in Figure 44, a new conversion methodology, called *Grammar-based conversion*, that meets all conditions was proposed to tackle the limitations of the *tracker-based hybrid conversion method*. *Grammar-based conversion* method does not use trackers because all the conversion functions are re-designed to work with the *nlohmann::json* objects.

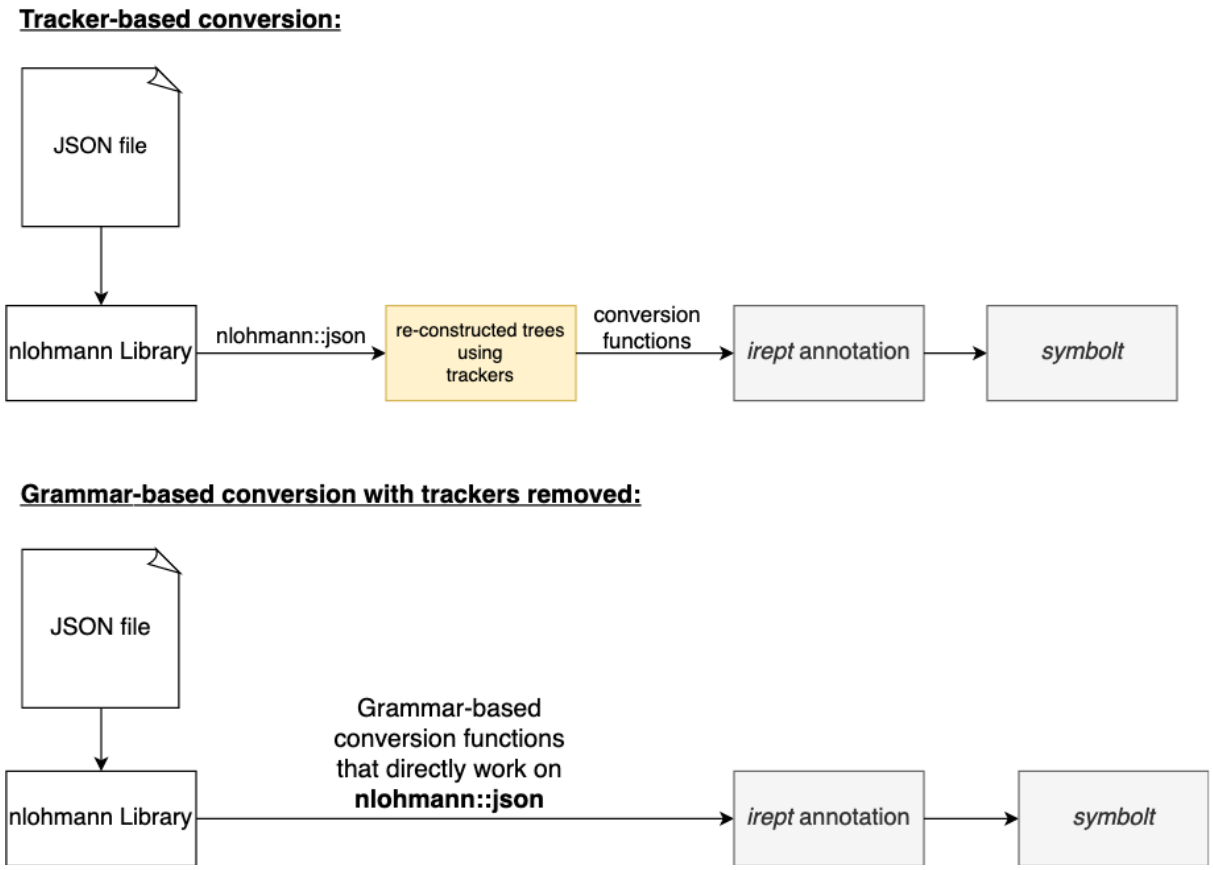


Figure 44: Tracker-based vs. Grammar-based conversion.

3.4.2 Grammar-Based Conversion

A detailed study of the *nlohmann* JSON library for C++ shows that there exists a base type to represent all types of JSON objects: the *nlohmann::json* data type [12]. All JSON value types can be implicitly converted to *nlohmann::json* type.

For example, the *nodes* array, the first element in that array, and *typeDescription* object can be implicitly converted into the *nlohmann::json* data type as shown in Figure 45. The nlohmann::json data type can be used to represent different JSON objects regardless of the actual data structure. The JSON objects #2 and #3 can be represented by a constant reference of the type *nlohmann::json* shown in Figure 46.

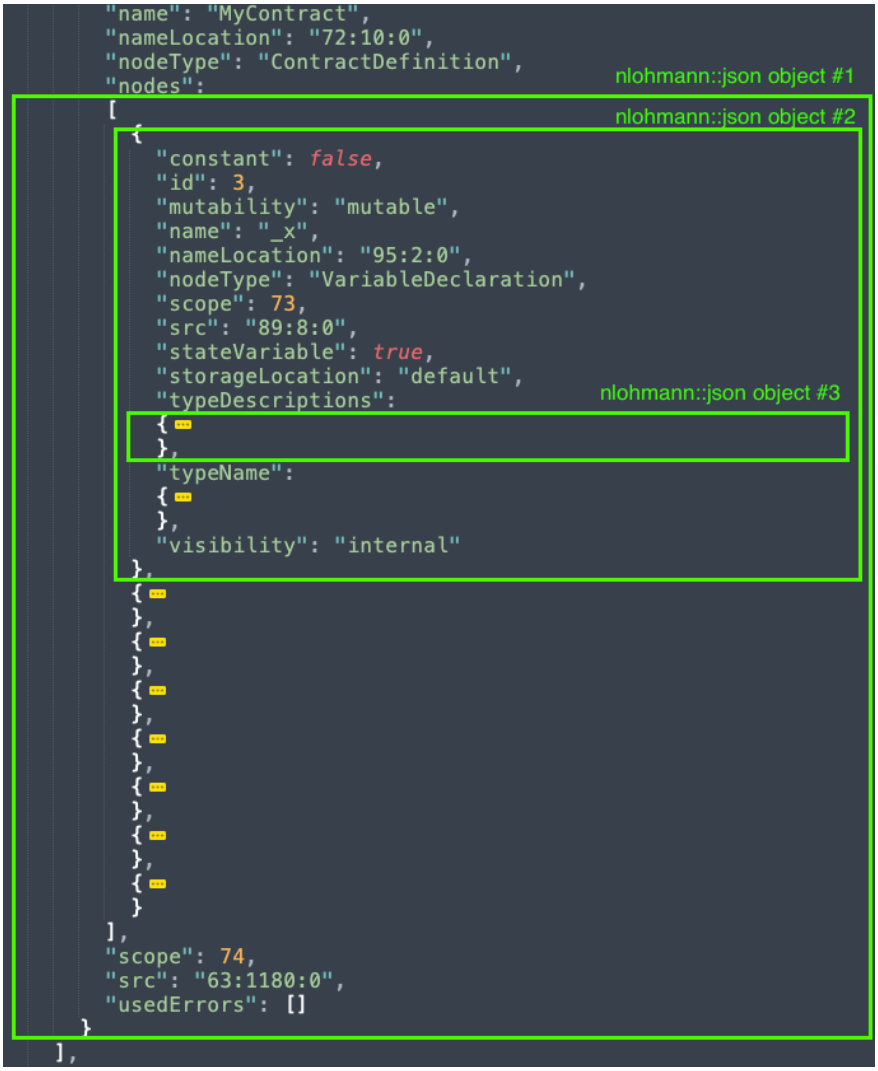


Figure 45: The nlohmann::json data type.

```
const nlohmann::json &object2 = nodes[0];
const nlohmann::json &object3 = object2["typeDescriptions"];
```

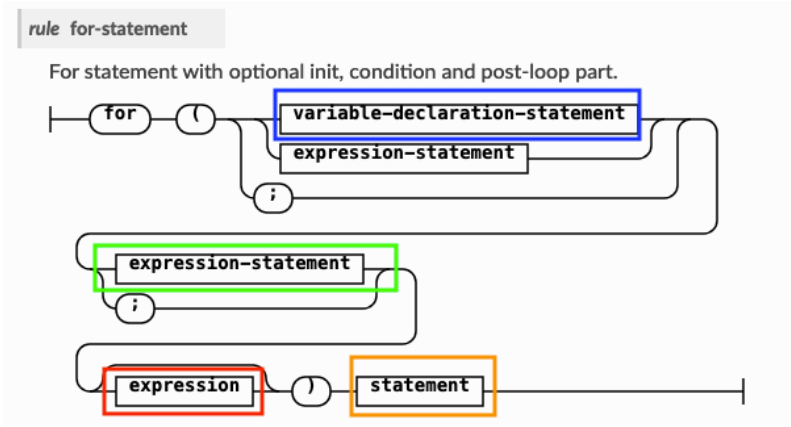
Figure 46: Type casting of AST nodes.

This feature of the *nlohmann*::json library fulfils the requirements as per **Condition #1 and #2** in Section 3.4.1.

To fulfil the requirement as stated in **Condition #3**, the conversion functions have to be re-designed to extract the structure of Solidity language construct from *nlohmann::json* data type whilst preserving the correct order of functions calls when it comes to annotate the *irept* nodes. This is to ensure that all child nodes are visited in the correct order. For example, a function parameter node must be converted before the function body node. Because the function parameters may appear in the function body. If a function body node is converted before its parameter node, the type checker will fail to convert any reference to the parameter.

The grammar-based conversion methodology uses the production rules of the Solidity grammar to make sure each node is visited in the correct order. The production rule is specified in Solidity documentation [16]. For example, the *init* node of the *for* loop needs to be converted before the body. Figure 47 (a) shows the production rule of the *for* loop in Solidity grammar. Figure 47 (b) shows corresponding conversion steps in *get_statement* function. The steps shown in Figure 47 (b) falls within the case of *ForStatement* as shown in Figure 48. "*SolidityGrammar::StatementT::ForStatement*" represents the *rule for-statement* as part of the *rule statement* shown in Figure 47. The colour coding shows the composition relation between these two rules.

(a)



(b)

```
589     // 1. annotate init
590     codet init = code_skipt();
591     if (get_statement(stmt["initializationExpression"], init))
592       return true;
593
594     convert_expression_to_code(init);
595
596     // 2. annotate condition
597     exprt cond = true_exprt();
598     if (get_expr(stmt["condition"], cond))
599       return true;
600
601     // 3. annotate increment
602     codet inc = code_skipt();
603     if (get_statement(stmt["loopExpression"], inc))
604       return true;
605
606     convert_expression_to_code(inc);
607
608     // 4. annotate body
609     codet body = code_skipt();
610     if (get_statement(stmt["body"], body))
611       return true;
612
613     convert_expression_to_code(body);
614
615     code_fort code_for;
616     code_for.init() = init;
617     code_for.cond() = cond;
618     code_for.iter() = inc;
619     code_for.body() = body;
620
621     new_expr = code_for;
622     break;
623   }
```

Figure 47: The conversion steps of a for loop.

```
487
488  bool solidity_convertert::get_statement(const nlohmann::json &stmt, exprt &new_expr)
489  {
490    // For rule statement
491    // Since this is an additional layer of grammar rules compared to clang C, we do NO
492    // Just pass the new_expr reference to the next layer.
493    static int call_stmt_times = 0; // TODO: remove debug
494
495    SolidityGrammar::StatementT type = SolidityGrammar::get_statement_t(stmt);
496    printf("  @@@ got Stmt: SolidityGrammar::StatementT::%s, ", SolidityGrammar::statem
497    printf("  call_stmt_times=%d\n", call_stmt_times++);
498
499    switch(type)
500    {
501      case SolidityGrammar::StatementT::Block:
502      {...
505      }
506      case SolidityGrammar::StatementT::ExpressionStatement:
507      {...
510      }
511      case SolidityGrammar::StatementT::VariableDeclStatement:
522      {...
541      }
542      case SolidityGrammar::StatementT::ReturnStatement:
543      {...
576      }
577      case SolidityGrammar::StatementT::ForStatement:
580      {...
623      }
624      case SolidityGrammar::StatementT::IfStatement:
639      {...
655      }
656      default:
657      {...
660      }
661    }
662
663    return false;
664  }
```

Figure 48: the conversion function for "rule statement".

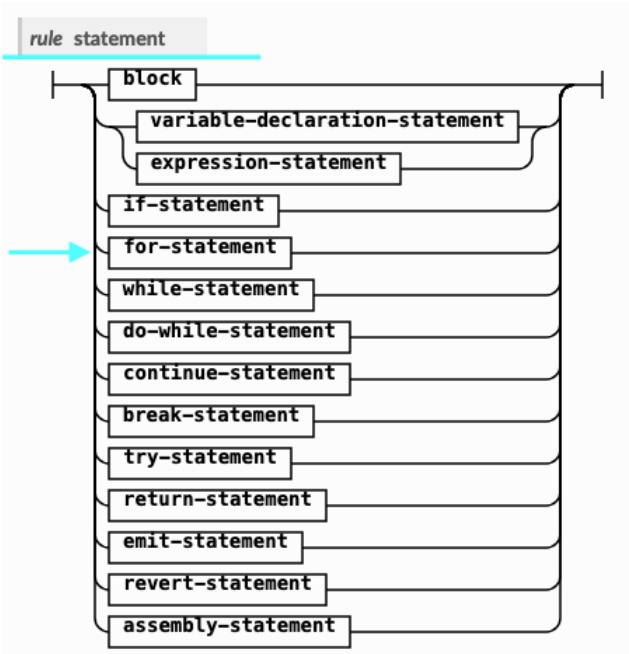


Figure 49: Production rules of Solidity statement.

Figure 47 shows the grammar-based conversion steps that convers a Solidity *for* loop into ESBMC IR *code_fort* shown in Figure 50. Note that the condition part is currently modelled as exprt as shown in Line 597 of Figure 47. It can be changed to code_t if the for loop contains empty or multiple conditional expressions.

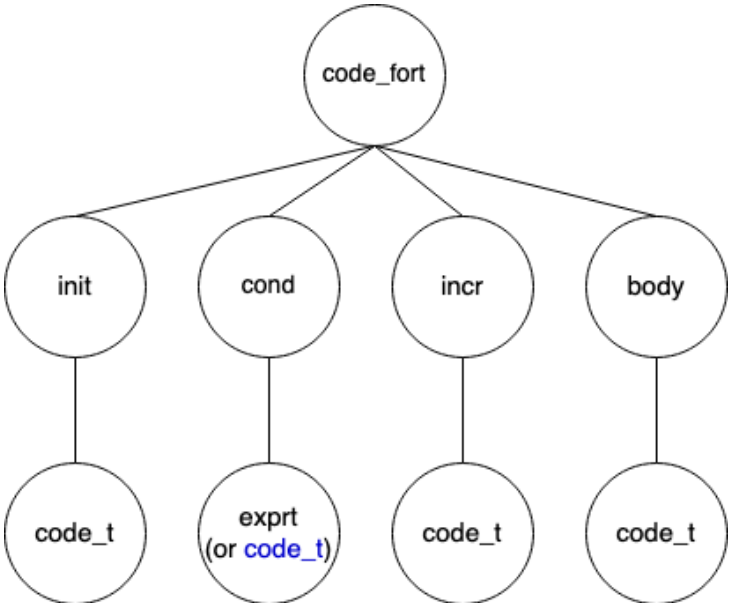


Figure 50: equivalent ipret node of the Solidity "for" loop.

According to the production rule shown in Figure 47 (a), it is legal to write a *for* statement that has empty *init*, *condition* and *increment*, such as "*for (; ;)*". The code can be easily extended to handle such cases. For example, Figure 51 shows a patch with just a few code lines to support empty *init* expression.

```
diff --git a/src/solidity-frontend/solidity_convert.cpp b/src/solidity-frontend/solidity_convert.cpp
index 921b3d87a..0a67d4f0b 100644
--- a/src/solidity-frontend/solidity_convert.cpp
+++ b/src/solidity-frontend/solidity_convert.cpp
@@ -587,9 +587,12 @@ bool solidity_convertert::get_statement(const nlohmann::json &stmt, exprt &new_e
       current_forStmt = &stmt;

       // 1. annotate init
-      codet init = code_skipt(); // code_skipt() means no init in for-stmt, e.g. for (; i< 10; ++i)
-      if (get_statement(stmt["initializationExpression"], init))
-        return true;
+      codet init = code_skipt();
+      if (stmt.contains("initializationExpression"))
+      {
+        if (get_statement(stmt["initializationExpression"], init))
+          return true;
+      }

       convert_expression_to_code(init);
```

Figure 51: Patch to support empty init expression.

During the implementation of the grammar-based conversion method, it was found that the Solidity grammar contains some cyclic but non-ambiguous production rules as shown in Figure 52.

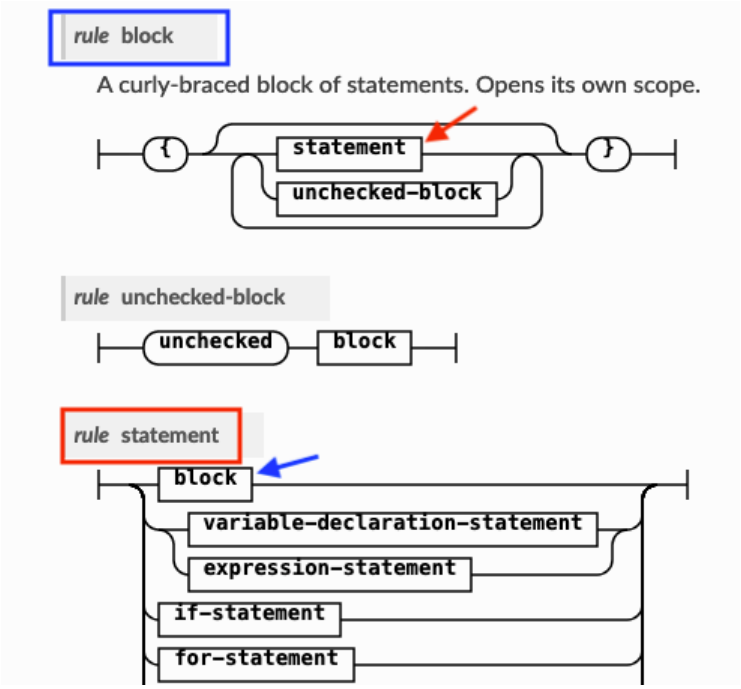


Figure 52: cycle references in Solidity grammar.

As shown in Figure 52, one of the production rules of *rule statement* generates *block*; one of the production rules of *block* leads back to *statement*. These rules are unambiguous because *statement* is bounded by curly braces in rule *block*. This relation of mutual inclusion is reflected in the implementation shown in Figure 53.

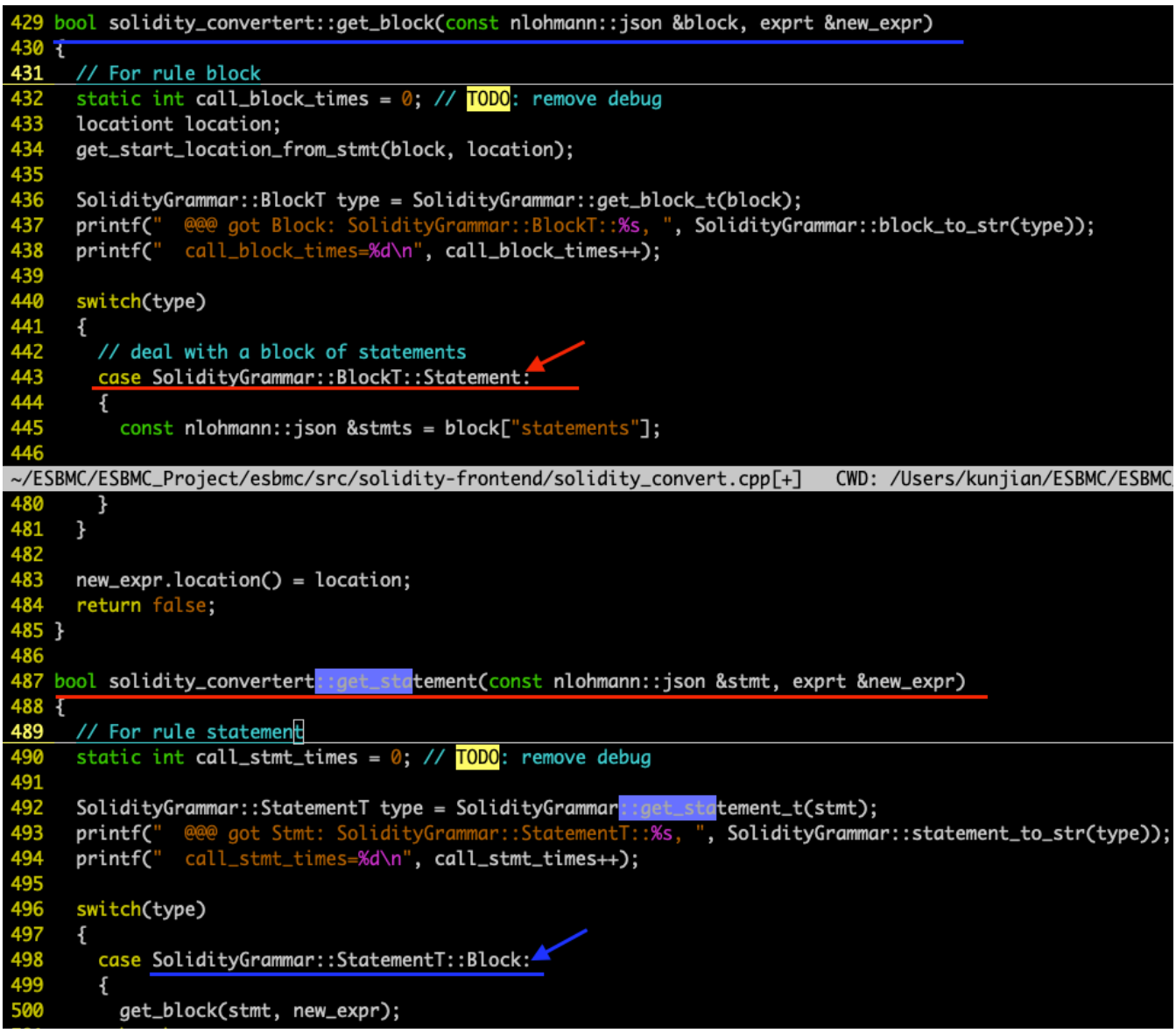


Figure 53: cyclic references in Solidity grammar.

3.4.3 Improved Readability, Maintainability and Extendibility

As shown in Figure 47, Figure 48 and Figure 49, the grammar-based conversion method is easy to follow because the implementation reflects the production rules in the Solidity grammar specification. The names of different types strictly follow the naming conventions in Solidity grammar. Compared to the tracker-based conversion, it significantly improves the readability of the code compared to the tracker method as the trackers may seem confusing to the new developers who do not necessarily know the specifics and design intentions of the tracker data structure. The implementation of Solidity grammar production rules and conversion steps were kept in two separate source files in *tracker-based hybrid conversion method*:

- The production rules were implemented in the *config* function of a tracker.
- The conversion steps were implemented in the conversion functions that deals the tracker.

In contrast, *the grammar-based hybrid conversion method* has everything contained in one place. The conversion steps can be directly mapped to the production rules defined in the Solidity grammar specification. For example, the conversion functions are named after the rule names; each *case* statement of a conversion function is named after the corresponding component names in that production rule. It would be relatively straightforward to locate the conversion steps for a specific language construct. Therefore, it improves maintainability and extendibility. To demonstrate the extendibility, Table 11 shows a group of patches to add additional features.

Commit	Patche Description	Changes	Remarks	Link
ad8680a	Add conversion steps of the binary operators "<" and *"-"*	**28 additions, 0 deletions**		All patches are available in kunjson01/esbmc Github[7].
c80ba64	Add conversion steps of *assume* function, as well as binary operator "!="	**74 additions, 22 deletions**		
8ff8734	Add support for __VERIFIER__assume	**6 additions, 3 deletions**	__VERIFIER__assume is an important function defition for SV-COMP	
85654eb	Add conversion steps of *for* loop	**113 additions, 3 deletions**		

Table 11: Extendibility of the new Solidity frontend.

[7] Commit history: https://github.com/kunjsong01/esbmc/commits/dev-solidity-support

3.5 Summary of Methodology

This section describes two methodologies to implement the new Solidity frontend:

- Tracker-Based Hybrid Conversion
- Grammar-Based Hybrid Conversion

The implementation of these methodologies is shown in Figure 54 and Figure 55. Compared to the *Tracker-Based Hybrid Conversion* method, the *Grammar-Based Hybrid Conversion* method is more compact, and has improved readability, maintainability, and extendibility.

```
  83 ./solidity_convert.h
  15 ./CMakeLists.txt
  78 ./solidity_ast_language.h
  34 ./solidity_convert_literals.cpp
  93 ./solidity_type.h
   9 ./typecast.h
 224 ./solidity_type.cpp
 931 ./solidity_convert.cpp
  12 ./typecast.cpp
 129 ./solidity_ast_language.cpp
1057 ./solidity_decl_tracker.cpp
 604 ./solidity_decl_tracker.h
3269 total
```

Figure 54: Workload of Tracker-Based Hybrid Conversion method

```
 116 ./solidity_convert.h
  14 ./CMakeLists.txt
 509 ./solidity_grammar.cpp
  44 ./solidity_convert_literals.cpp
  76 ./solidity_language.h
 190 ./solidity_grammar.h
  36 ./pattern_check.h
   9 ./typecast.h
1831 ./solidity_convert.cpp
 131 ./solidity_language.cpp
  12 ./typecast.cpp
 119 ./pattern_check.cpp
3087 total
```

Figure 55 Workload of Grammar-Based Hybrid Conversion method.

4 Evaluation

4.1 Test Suite Design

Since Solidity does not have a standard benchmark, a test suite was created. This test suite contains t test cases, each of which is for a specific type of vulnerability. The test cases are shown in Table 12.

Category	Test Case ID	Description
Pattern-Based Vulnerability	#1	Authorization Through Tx.origin in a payable function in the smart contract
Reasoning-Based Vulnerability	#2	Arithmetic overflow with nested binary operator expression
	#3	Arithmetic underflow with unary expression
	#4	Loops. Use incremental-bmc to detect arithmetic underflow in a loop.
	#5	Array Out-of-Bound exception in a loop.
	#6	Satisfiability test with *nondet*, *assume* and *assert*
	#7	Test __VERIFIER__assume

Table 12: Test suite

The test cases are classified into two groups: pattern-based vulnerability and reasoning-based vulnerability. For pattern-based vulnerabilities, the new Solidity frontend loops over the AST nodes and tries to detect a pattern of unsafe code. As for reasoning-based vulnerabilities, the new Solidity frontend transforms the AST nodes into the *irept* nodes and generates the symbols table. Then the rest is handed over to the middle end and backend of ESBMC verification pipeline.

These test cases are microbenchmarks, which serve three purposes:

- To guide the development of a Solidity type checker. This project employs test-driven development.
- To test the Solidity verification pipeline in ESBMC
- To test other verification tools and compare the results to ESBMC's results.

Each subsection describes one test case in Table 12.

4.1.1 TC1: Authorization Through Tx.orgin

This test case is shown in Figure 56. The function *transferTo* in Line 10 is used to transfer Ethers to another smart contract. Since this function is used for making payment, it is protected by the statement "*require(tx.origin == owner);*". It means that the payment is authorized if and only if the caller of this function is the owner of this smart contract. It seems reasonably safe to authorize a payment by checking the precondition "*(tx.origin == owner)*". However, it can be easily attacked by the malicious contract shown in Figure 57.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.7.0 <0.9.0;
3  contract TxOriginVictim {
4      address owner;
5
6      constructor() {
7          owner = msg.sender;
8      }
9
10     function transferTo(address payable dest, uint amount) public {
11         require(tx.origin == owner);
12         dest.transfer(amount);
13     }
14 }
```

Figure 56: TC1 - Authorization using Tx.Origin.[8]

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.7.0 <0.9.0;
3  interface TxOriginVictim {
4    function transferTo(address to, uint amount);
5  }
6
7  contract TxOriginAttacker {
8
9    address owner;
10   function TxOriginAttacker() public {
11     owner = msg.sender;
12   }
13
14   function getOwner() public returns (address) {
15     return owner;
16   }
17
18   function() payable public {
19     TxOriginVictim(msg.sender).transferTo(owner, msg.sender.balance);
20   }
21 }
```

Figure 57: Attacker smart contract.[8]

[8] This example is taken from: https://medium.com/coinmonks/solidity-tx-origin-attacks-58211ad95514

With the new Solidity frontend, ESBMC can detect this vulnerability by checking an AST node that contains the pattern as follow:

- A call to the authorization function "*require*"
- The argument of this function is a "*BinaryOperation*" expression that uses "==" operator
- The "*leftExpression*" is a "*MemberAccess*" expression referring to the special identifier "*tx*" and accessing to the member "*origin*".

This pattern is shown in Figure 58.

Figure 59 shows that ESBMC successfully detected the vulnerability "authorization through Tx.Origin" and identified it as SWC-115 listed in the *SWC registry* for Smart Contract Weakness Classification and Test Cases [54].

Figure 58: Pattern of "Autorization through Tx.origin"

```
Done conversion of intrinsics...
Checking function transferTo ...
  - Pattern-based checking: SWC-115 Authorization through tx.origin
statements in function body array ...
@@ checking body stmt 0
        - Found vulnerability SWC-115 Authorization through tx.origin
Assertion failed: (0), function check_authorization_through_tx_origin
Abort trap: 6
```

Figure 59: ESBMC detects authorization through Tx.origin.

4.1.2 TC2: Arithmetic Overflow

This test demonstrates that ESBMC can detect arithmetic overflow error in a nested binary operation expression shown in line 15 in Figure 60.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.4.26;
3
4  contract MyContract {
5      uint8 x;
6      uint8 y;
7      uint8 z;
8      uint8 sum;
9
10     function func_overflow() external {
11       x = 100;
12       y = 240;
13       z = 3;
14
15       sum = x + y + z;
16       assert(sum > 100);
17     }
18 }
```

Figure 60: TC2 - arithmetic overflow in a nested binary operation expression.

As shown in Figure 61, the arithmetic overflow error is successfully detected by ESBMC. The counterexample shows that there exists a state, *State 9,* that violates the safety property "*sum* > *100*" as specified by the *assert* statement in line 16 in Figure 60. Because *sum* was declared as *uint8*, which can only represent values from 0 to 255. The expression "$x + y + z$" evaluates to $100 + 240 + 3 = 343$, a value that cannot be represented by *uint8*.

```
Counterexample:

State 1 file MyContract_overflow_nested.sol line 1 thread 0
------------------------------------------------------
  x = 0 (00000000)

State 2 file MyContract_overflow_nested.sol line 1 thread 0
------------------------------------------------------
  y = 0 (00000000)

State 3 file MyContract_overflow_nested.sol line 1 thread 0
------------------------------------------------------
  z = 0 (00000000)

State 4 file MyContract_overflow_nested.sol line 1 thread 0
------------------------------------------------------
  sum = 0 (00000000)

State 5 file MyContract_overflow_nested.sol line 1 function func_overflow thread 0
------------------------------------------------------
  x = 100 (01100100)

State 6 file MyContract_overflow_nested.sol line 1 function func_overflow thread 0
------------------------------------------------------
  y = 240 (11110000)

State 7 file MyContract_overflow_nested.sol line 1 function func_overflow thread 0
------------------------------------------------------
  z = 3 (00000011)

State 8 file MyContract_overflow_nested.sol line 1 function func_overflow thread 0
------------------------------------------------------
  sum = 87 (01010111)

State 9 file MyContract_overflow_nested.sol line 1 function func_overflow thread 0
------------------------------------------------------
Violated property:
  file MyContract_overflow_nested.sol line 1 function func_overflow
  assertion
  (signed int)sum > 100


VERIFICATION FAILED
```

Figure 61: TC2 result.

4.1.3 TC3: Arithmetic Underflow

This test demonstrates that ESBMC can detect arithmetic underflow in a program that contains unary operators shown in line 9 and 10 in Figure 62.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.4.26;
3
4  contract MyContract {
5      uint8 x;
6
7      function func_underflow() external {
8        x = 1;
9        --x;
10       --x;
11       assert(x < 5);
12     }
13 }
```

Figure 62: TC3 –arithmetic underflow with unary operators

As shown in Figure 63, the arithmetic underflow is successfully detected by ESBMC. The counterexample shows that there exists a state, *State 5,* that violates the safety property "$x < 5$" as specified by the *assert* statement in line 11 in Figure 62. Because *x* was declared as *uint8*, which can only represent values from 0 to 255. The expression "*--x*" in line 9 and 10 decrements *x* twice. The result is *-1*, a value that cannot be represented by *uint8*. Due to arithmetic underflow, *-1* wraps back to 255.

```
332 Counterexample:
333
334 State 1 file MyContract_underflow_UnaryOp.sol line 1 thread 0
335 ----------------------------------------------------
336   x = 0 (00000000)
337
338 State 2 file MyContract_underflow_UnaryOp.sol line 1 function func_underflow thread 0
339 ----------------------------------------------------
340   x = 1 (00000001)
341
342 State 3 file MyContract_underflow_UnaryOp.sol line 1 function func_underflow thread 0
343 ----------------------------------------------------
344   x = 0 (00000000)
345
346 State 4 file MyContract_underflow_UnaryOp.sol line 1 function func_underflow thread 0
347 ----------------------------------------------------
348   x = 255 (11111111)
349
350 State 5 file MyContract_underflow_UnaryOp.sol line 1 function func_underflow thread 0
351 ----------------------------------------------------
352 Violated property:
353   file MyContract_underflow_UnaryOp.sol line 1 function func_underflow
354   assertion
355   (signed int)x < 5
356
357
358 VERIFICATION FAILED
```

Figure 63: TC3 result

4.1.4 TC4: Loops

This test case aims to test the verification strategy "*--incremental-bmc*" in ESMBC. The code is shown in Figure 64. This test case contains a bug of arithmetic overflow in the 3rd iteration of the loop.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.4.26;
3
4  contract MyContract {
5      uint8 _x;
6
7      function func_loop() external {
8        _x = 2;
9        for (uint8 i = 0; i < 3; ++i)
10       {
11         _x = _x - 1;
12         assert(_x < 5);
13       }
14     }
15 }
```

Figure 64: TC4 – loop

As shown in Figure 65, the bug is successfully detected by ESBM. To show which loop iteration triggers this bug, TC#4 was tested using "*--incremental-bmc*" option so that ESBMC can unwind the loop incrementally with the index *k*. ESBMC reported "*Bug found (k = 3)*", which means that the bug was found in the 3rd iteration. After the 2nd iteration, *x* becomes 0. In the 3rd iteration, *x* is decremented by 1, which leads to the arithmetic underflow error.

```
Counterexample:

State 1 file MyContract_underflow_loop.sol line 1 thread 0
----------------------------------------------------
  _x = 0 (00000000)

State 2 file MyContract_underflow_loop.sol line 1 function func_loop thread 0
----------------------------------------------------
  _x = 2 (00000010)

State 3 file MyContract_underflow_loop.sol line 1 function func_loop thread 0
----------------------------------------------------
  i = 0 (00000000)

State 4 file MyContract_underflow_loop.sol line 1 function func_loop thread 0
----------------------------------------------------
  _x = 1 (00000001)

State 5 file MyContract_underflow_loop.sol line 1 function func_loop thread 0
----------------------------------------------------
  i = 1 (00000001)

State 6 file MyContract_underflow_loop.sol line 1 function func_loop thread 0
----------------------------------------------------
  _x = 0 (00000000)

State 7 file MyContract_underflow_loop.sol line 1 function func_loop thread 0
----------------------------------------------------
  i = 2 (00000010)

State 8 file MyContract_underflow_loop.sol line 1 function func_loop thread 0
----------------------------------------------------
  _x = 255 (11111111)

State 9 file MyContract_underflow_loop.sol line 1 function func_loop thread 0
----------------------------------------------------
Violated property:
  file MyContract_underflow_loop.sol line 1 function func_loop
  assertion
  (signed int)_x < 5


VERIFICATION FAILED

Bug found (k = 3)
```

declared before the loop

assignment before the loop

k = 1

k = 2

k = 3

Violated the above property when k = 3

Figure 65: TC4 result

4.1.5 TC5: Array Out-of-Bound Exception in a loop

This test demonstrates that ESBMC can detect out-of-bound exceptions in an array subscript expression shown in line 10 in Figure 66. Similar to TC4, TC5 was also tested using the option "*--incremental-bmc*" to verify array out-of-bound exception in a loop.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.4.26;
3
4  contract MyContract {
5
6      function func_array_loop() external pure {
7        uint8[2] memory a;
8
9        a[0] = 100;
10       for (uint8 i = 1; i < 3; ++i)
11       {
12         a[i] = 100;
13         assert(a[i-1] == 100);
14       }
15     }
16 }
```

Figure 66: TC5 – Array out-of-bound access a loop.

As shown in Figure 67, ESBMC successfully detected the bug. ESBMC reported "*Bug found (k = 2)*", which means that the bug was found in the 2nd iteration of the loop. In this iteration, the array subscript expression "*a[i] = 100*" contains an invalid index *i = 2*, which exceeds the bound of this array.

```
388  Counterexample:
389
390  State 1 file MyContract_array_loop.sol line 1 function func_array_loop thread 0
391  ------------------------------------------------
392    a[0] = 100 (01100100)                                   assignment before loop
393  ================================================
394  State 2 file MyContract_array_loop.sol line 1 function func_array_loop thread 0
395  ------------------------------------------------
396    i = 1 (00000001)                                                    k = 1
397
398  State 3 file MyContract_array_loop.sol line 1 function func_array_loop thread 0
399  ------------------------------------------------
400    a[1] = 100 (01100100)
401  ================================================
402  State 5 file MyContract_array_loop.sol line 1 function func_array_loop thread 0
403  ------------------------------------------------
404    i = 2 (00000010)                                                    k = 2
405
406  State 6 file MyContract_array_loop.sol line 1 function func_array_loop thread 0
407  ------------------------------------------------
408  Violated property:
409    file MyContract_array_loop.sol line 1 function func_array_loop
410    array bounds violated: array `a' upper bound
411    (signed long int)i < 2
412
413
414  VERIFICATION FAILED
415
416  Bug found (k = 2)
```

Figure 67: TC5 result.

4.1.6 TC6: Satisfiability Test using *nondet*, *assume* and *assert*

ESBMC can find a counterexample to satisfy the negation of the property we would like to check. This test case aims to show the effect of additional constraints. The test case is shown in Figure 68. The data type used in this test case is *uint8*.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.4.26;
3
4  contract MyContract {
5      uint8 x;
6      uint8 sum;
7
8      function nondet() public pure returns(uint8)
9      {
10       uint8 i;
11       return i;
12     }
13
14     function __ESBMC_assume(bool) internal pure { }
15
16     function func_sat() external {
17       x = 0;
18       uint8 y = nondet();
19       sum = x + y;
20
21       // C : Add additional constraints here
22
23       // P : Properties we want to check
24       assert(sum % 16 != 0);
25     }
26 }
```

Figure 68: TC6 – effect of "assume" on finding satisfiability.

The satisfiability problem is described as follow:

Satisfiability_#1. *Given the binary operation expression "sum = x + y" where x = 0, find a value of y that satisfies the NEGATION of the property "sum % 16 != 0".*

The negation of the property in line 24 is "*sum % 16 == 0*". ESBMC finds the answer to ***Satisfiability_#1:*** *y = 240* shown in line 357 in Figure 69.

```
355  State 1 file MyContract_satisfiability.sol line 1 function func_sat thread 0
356  ------------------------------------------------
357    y = 240 (11110000)
358
359  State 2 file MyContract_satisfiability.sol line 1 function func_sat thread 0
360  ------------------------------------------------
361    sum = 240 (11110000)
362
363  State 3 file MyContract_satisfiability.sol line 1 function func_sat thread 0
364  ------------------------------------------------
365  Violated property:
366    file MyContract_satisfiability.sol line 1 function func_sat
367    assertion
368    (signed int)sum % 16 != 0
369
370
371  VERIFICATION FAILED
```

Figure 69: Answer to Satisfiability_#1.

Satisfiability_#2*. Given the binary operation expression "sum = x + y" where x = 0 and* ***220 < y < 255****, find a value of y that satisfies the NEGATION of the property "sum % 16 != 0".*

To specify the range "220 < y < 255", additional constraints are added using the specifical function *"__ESBMC__assume"* in line 22 and 23 in Figure 70. In this range, there are two numbers satisfying the negation of the property in line 26 of Figure 70: {224, 240}. ESBMC successfully found the answer *y = 224* shown in line 389 in Figure 71.

```
1   // SPDX-License-Identifier: GPL-3.0
2   pragma solidity >=0.4.26;
3
4   contract MyContract {
5       uint8 x;
6       uint8 sum;
7
8       function nondet() public pure returns(uint8)
9       {
10        uint8 i;
11        return i;
12      }
13
14      function __ESBMC_assume(bool) internal pure { }
15
16      function func_sat() external {
17        x = 0;
18        uint8 y = nondet();
19        sum = x + y;
20
21        // C : Add additional constraints here
22        __ESBMC_assume(y < 255);
23        __ESBMC_assume(y > 220);
24
25        // P : Properties we want to check
26        assert(sum % 16 != 0);
27      }
28  }
```

Figure 70: updated TC6 for Satisfiability_#2.

```
373  Counterexample:
374
375  State 1 file MyContract_satisfiability_2.sol line 1 thread 0
376  ------------------------------------------------------
377    x = 0 (00000000)
378
379  State 2 file MyContract_satisfiability_2.sol line 1 thread 0
380  ------------------------------------------------------
381    sum = 0 (00000000)
382
383  State 3 file MyContract_satisfiability_2.sol line 1 function func_sat thread 0
384  ------------------------------------------------------
385    x = 0 (00000000)
386
387  State 4 file MyContract_satisfiability_2.sol line 1 function func_sat thread 0
388  ------------------------------------------------------
389    y = 224 (11100000)
390
391  State 5 file MyContract_satisfiability_2.sol line 1 function func_sat thread 0
392  ------------------------------------------------------
393    sum = 224 (11100000)
394
395  State 8 file MyContract_satisfiability_2.sol line 1 function func_sat thread 0
396  ------------------------------------------------------
397  Violated property:
398    file MyContract_satisfiability_2.sol line 1 function func_sat
399    assertion
400    (signed int)sum % 16 != 0
401
402
403  VERIFICATION FAILED
```

Figure 71: Answer to Satisfiability_#2.

Satisfiability_#3*. Given the binary operation expression "sum = x + y" where x = 0 and 220 < y < 255,* ***and y is not 224****, find a value of y that satisfies the NEGATION of the property "sum % 16 != 0".*

To specify the the additional condition "***y is not 244"****,* additional constraint is added to exclude the number *224* in line 24 in Figure 72. In this range, there are two numbers satisfying the negation of the property in line 27: {224, 240}. Since the number *224* is excluded, this only leaves us with the number 240. ESBMC successfully found the answer y = 240 shown in line 399 in Figure 73.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.4.26;
3
4  contract MyContract {
5      uint8 x;
6      uint8 sum;
7
8      function nondet() public pure returns(uint8)
9      {
10       uint8 i;
11       return i;
12     }
13
14     function __ESBMC_assume(bool) internal pure { }
15
16     function func_sat() external {
17       x = 0;
18       uint8 y = nondet();
19       sum = x + y;
20
21       // C : Add additional constraints here
22       __ESBMC_assume(y < 255);
23       __ESBMC_assume(y > 220);
24       __ESBMC_assume(y != 224); // 224 = 16 * 14;
25
26       // P : Properties we want to check
27       assert(sum % 16 != 0);
28     }
29 }
```

Figure 72: updated TC6 for Satisfiability_#3.

```
383  Counterexample:
384
385  State 1 file MyContract_satisfiability_3.sol line 1 thread 0
386  ------------------------------------------------
387    x = 0 (00000000)
388
389  State 2 file MyContract_satisfiability_3.sol line 1 thread 0
390  ------------------------------------------------
391    sum = 0 (00000000)
392
393  State 3 file MyContract_satisfiability_3.sol line 1 function func_sat thread 0
394  ------------------------------------------------
395    x = 0 (00000000)
396
397  State 4 file MyContract_satisfiability_3.sol line 1 function func_sat thread 0
398  ------------------------------------------------
399    y = 240 (11110000)
400
401  State 5 file MyContract_satisfiability_3.sol line 1 function func_sat thread 0
402  ------------------------------------------------
403    sum = 240 (11110000)
404
405  State 9 file MyContract_satisfiability_3.sol line 1 function func_sat thread 0
406  ------------------------------------------------
407  Violated property:
408    file MyContract_satisfiability_3.sol line 1 function func_sat
409    assertion
410    (signed int)sum % 16 != 0
411
412
413  VERIFICATION FAILED
```

Figure 73: Answer to Satisfiability_#3.

Satisfiability_#4*. Given the binary operation expression "sum = x + y" where x = 0 and 220 < y < 255, and* ***y is not 224 or 240****, find a value of y that satisfies the NEGATION of the property "sum % 16 != 0".*

To exclude the number *240,* additional constraint is added to exclude the number *240* in line 25 in Figure 74. In this range, there are two numbers that satisfies the negation of the property in line 28: {224, 240}. Since both *224* and 240 are excluded, this only leaves us with an empty set ∅. As shown in Figure 75, ESBMC reports "VERIFICATION SUCCESSFULL" because it cannot find a counterexample to satisfy the negation of the property in line 28 of Figure 74.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.4.26;
3
4  contract MyContract {
5      uint8 x;
6      uint8 sum;
7
8      function nondet() public pure returns(uint8)
9      {
10       uint8 i;
11       return i;
12     }
13
14     function __ESBMC_assume(bool) internal pure { }
15
16     function func_sat() external {
17       x = 0;
18       uint8 y = nondet();
19       sum = x + y;
20
21       // C : Add additional constraints here
22       __ESBMC_assume(y < 255);
23       __ESBMC_assume(y > 220);
24       __ESBMC_assume(y != 224); // 224 = 16 * 14;
25       __ESBMC_assume(y != 240); // 240 = 16 * 15;
26
27       // P : Properties we want to check
28       assert(sum % 16 != 0);
29     }
30 }
```

Figure 74: updated TC6 for Satisfiability_#4.

```
378  Generating GOTO Program
379  GOTO program creation time: 0.294s
380  GOTO program processing time: 0.000s
381  Starting Bounded Model Checking
382  Symex completed in: 0.002s (21 assignments)
383  Slicing time: 0.000s (removed 1 assignments)
384  Generated 1 VCC(s), 1 remaining after simplification (20 assignments)
385  No solver specified; defaulting to Boolector
386  Encoding remaining VCC(s) using bit-vector/floating-point arithmetic
387  Encoding to solver time: 0.000s
388  Solving with solver Boolector 3.2.1
389  Encoding to solver time: 0.000s
390  Runtime decision procedure: 0.001s
391  BMC program time: 0.005s
392
393  VERIFICATION SUCCESSFUL
```

Figure 75: Answer to Satisfiability_#4.

4.1.7 TC7: Satisfiability Test using SV-COMP Function

This test case aims to show the effect of additional constraints using *__VERIFIER__assume* function. The test case repeats the test of ***Satisfiability_#3*** defined in the previous section. As shown in Figure 76, TC7 is an updated version of TC6 with the modifications as follow:

- *__ESBMC__assume* function is replaced by *__VERIFIER__assume function* in line 15.
- The additional constrains are specified in using *__VERIFIER__assume* in lines 23, 24 and 25.

As shown in Figure 77, ESBMC can find the answer *y = 240,* which proves that the new Solidity frontend also supports *__VERIFIER_assume* function.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.4.26;
3
4  contract MyContract {
5      uint8 x;
6      uint8 sum;
7
8      function nondet() public pure returns(uint8)
9      {
10       uint8 i;
11       return i;
12     }
13
14     //function __ESBMC_assume(bool) internal pure { }
15     function __VERIFIER_assume(bool) internal pure { }
16
17     function func_sat() external {
18       x = 0;
19       uint8 y = nondet();
20       sum = x + y;
21
22       // C : Add additional constraints here
23       __VERIFIER_assume(y < 255);
24       __VERIFIER_assume(y > 220);
25       __VERIFIER_assume(y != 224); // 224 = 16 * 14;
26
27       // P : Properties we want to check
28       assert(sum % 16 != 0);
29     }
30 }
```

Figure 76: TC7 – effect of "assume" on finding satisfiability.

```
State 1 file MyContract_satisfiability_VERIFIER.sol line 1 thread 0
------------------------------------------------
  x = 0 (00000000)

State 2 file MyContract_satisfiability_VERIFIER.sol line 1 thread 0
------------------------------------------------
  sum = 0 (00000000)

State 3 file MyContract_satisfiability_VERIFIER.sol line 1 function func_sat thread 0
------------------------------------------------
  x = 0 (00000000)

State 4 file MyContract_satisfiability_VERIFIER.sol line 1 function func_sat thread 0
------------------------------------------------
  y = 240 (11110000)

State 5 file MyContract_satisfiability_VERIFIER.sol line 1 function func_sat thread 0
------------------------------------------------
  sum = 240 (11110000)

State 9 file MyContract_satisfiability_VERIFIER.sol line 1 function func_sat thread 0
------------------------------------------------
Violated property:
  file MyContract_satisfiability_VERIFIER.sol line 1 function func_sat
  assertion
  (signed int)sum % 16 != 0


VERIFICATION FAILED
```

Figure 77: TC7 result. Answer to Satisfiability_#3.

4.2 Threats to Validity

Internal Validity.

- Each test case was designed to just contain one vulnerability. The vulnerability does not have dependencies on a second vulnerability. The vulnerability in each test case is of the type specified by the SWC registry [54]. For example, the Tx.Origin test case is the vulnerable example from the official Solidity document. Using Tx.Origin for authorization is considered a pitfall in Solidity document [55]. Remix IDE, Slither, Mythril and ESBMC were able to detect such vulnerability. However, SmartCheck and Oyente are not able to detect it.

Generalizability.
The test cases are not tailored SMT-Based Bounded Model Checking. They were designed to be used as a general case.

- Each test case was a well-formed Solidity program (c.f. Section 2.1.2) because it is syntactically correct, and can be compiled by Solidity compiler without any errors or warnings. All language constructs were used according to Solidity grammar rules.

4.3 Findings and Comparison to Other Verification Frameworks

The test cases were also run with other state-of-the-art Solidity verification frameworks. This section compares ESBMC to other verification frameworks. As shown in Table 13 and Table 14, only ESBMC can verify all tests cases and provide counterexamples for each type of vulnerability.

Vulnerability Detection	**Remix IDE**	**Smartcheck**	**Slither**	**Oyente**	**Mythril**	***ESBMC***	**SolAnalyser**
Overflow	Not found	Not found	Not found	Not found	Found	Found	This framework does not work with Solidity compiler version 0.8.26
Underflow	Not found	Not found	Not found	Not found	Not found	Found	
TxOrigin	Found	Not found	Found	Not found	Found	Found	
Array out of bound access	Not found	Not found	Not found	Not found	Found	Found	

Table 13: Compare ESBMC to other tools.[9]

Counterexamples	**Remix IDE**	**Smartcheck**	**Slither**	**Oyente**	**Mythril**	***ESBMC***	**SolAnalyser**
Overflow	N/A	N/A	N/A	N/A	No counter-example provided	Counter-example provided	This framework does not work with Solidity compiler version 0.8.26
Underflow	N/A	N/A	N/A	N/A	N/A	Counter-example provided	
TxOrigin	TxOrigin Identified	N/A	TxOrigin Identified	N/A	TxOrigin Identified	TxOrigin Identified	
Array out of bound access	N/A	N/A	N/A	N/A	Counter-example provided	Counter-example provided	

Table 14: Availability of counterexamples.

The evaluation shows that ESBMC (Solidity frontend) outperforms all other tools.

[9] https://github.com/kunjsong01/data_set/tree/main/vulnerability_examples/results_Nedas

5 Conclusion and Further Work

This chapter reviews the deliverables of this project to determine to what extent the objectives have been met, and reflect on the project to assess what went well and what could be improved. This chapter ends by discussing limitations and recommending future work.

5.1 Deliverables and Key Achievements

Deliverables.
A new Solidity frontend was developed to enable ESBMC to verify smart contracts written in Solidity programming language. In this new frontend, the most critical component is the type checker. Two methods were proposed and implemented to implement the new type checker to convert Solidity AST nodes into ESBMC *irept* nodes: *Tracker-Based Hybrid Conversion* and *Grammar-Based Hybrid Conversion*. As a result, two versions of the new Solidity frontend were developed:

Versions	Methodology	Workload
f061108[10]	*Tracker-Based Hybrid Conversion*	3629 lines of C++ code
66f36ff[10]	*Grammar-Based Hybrid Conversion*	3087 lines of C++ code

Table 15: Two versions of the new Solidity frontend.

As shown in Table 15, the new Solidity frontend that was implemented using *Grammar-Based Hybrid Conversion* is more compact. To integrate the new frontend with existing ESBMC language infrastructure, the following patches were merged to the *dev-solidity-support* branch:

Patches	Description	Workload
Commit d7ac874[10]	Added Solidity placeholders	144 additions and 1 deletion
Commit 8011413[10]	Fixed linking error in CMake	5 additions and 5 deletions

Table 16: Integrate the new Solidity frontend with ESBMC.

Table 16 shows that ESBMC is well-structured and can be easily extended to add a new frontend to support a new language.

Since Solidity does not have a standard benchmark, a test suite was also developed to test the new Solidity frontend. The *Grammar-based Hybrid Conversion method* facilitates the extension and maintainability of code to support the verification of more complex Solidity programs that contain advanced language constructs and special functions like *assume* and *nondeterminism.*

Key Achievements.
In this project a new Solidity frontend is implemented, which enables ESBMC to verify Solidity smart contracts using SMT-based Bounded Model Checking. The key component in this new frontend is the type checker, which was implemented based on the new symbol conversion methodology *Grammar-based Hybrid Conversion*. It also supports the main

[10] Available at: https://github.com/kunjsong01/esbmc/commits/dev-solidity-support/src

features of a bounded model checker: *nondet*, *assert* and *assume*. Apart from the new Solidity frontend, three patches were submitted and merged to ESBMC main line:

Patch	Description	Patch size
Commit 34cfd4a6	Improved building instructions for macOS	42 additions and 4 deletions
Commit 3f9d3f8b	PR #485 - Fixed symbol table printing	3 additions and 1 deletions
Commit 39bf25d4	Added test case for PR #485	12 additions and 0 deletions

Table 17: Contributions to ESBMC main line.

5.2 Reflection

This subsection summarizes what went well and what could be improved, which is similar to Agile Retrospective [56].

What went well?
Due to the lack of a standard benchmark for Solidity smart contracts, this project employs the test-driven development method [57]. Before extending the code to support a new Solidity language construct that usually needs some prerequisites to support multiple related production rules in Solidity grammar, a test case that contains such construct is developed before writing the code.

What could be improved?
Similar to other software development projects, this project also contains development tasks that consume more time than the original estimate. Making a precise estimate for each task is as difficult as the project itself. In this project, a 25% of the buffer time was used, e.g. if the original estimate for a development task is 2d (i.e. two days in Jira time unit [58]), a buffer time of 4h (i.e. four hours in Jira time unit) is taken into the overall estimate. However, there still exists a few tasks that exceed 100% of the original estimate.

For example, the original estimate of adding support for function return was 6 hours, but the actual time logged was two days. This overflow happened because of the unexpected blocking task to implement *FunctionToPointer decay*. A similar case also happened when implementing the feature to type check array due to *ArrayToPointer decay*.

5.3 Limitations and Future Work

To support all Solidity features, the new Solidity frontend must be extended to support all production rules as specified in Solidity grammar.

Solidity is an imperative programming language supporting the objected oriented programming paradigm. Similar to a class in other OOP programming languages like C++ and Java, a Solidity contract is a container that includes the data and corresponding methods. Solidity supports:

i. Multiple inheritance as well as polymorphism
ii. Interface that contains function declarations without implementation
iii. Special functions like constructor and destructor (called *selfdestruct*).
iv. Visibility specifiers, such as public, private, external, and internal.

Apart from these standard OOP features, Solidity also supports some advanced features including:

i. Cryptographic hash functions, e.g. keccak256, sha256 and ripmd160.
ii. Callable objects,
iii. An unnamed fallback function to be called when no other functions match the callee's reference id provided by the caller. Each contract is only allowed to have one unnamed fallback function
iv. Types with unconventional bit width, such as bytes3, int24, uint56 and int256.
v. Multiple return values
vi. Ethereum Virtual Machine has three types of memory: "storage" to hold the contract state variables, "memory" to hold temporary values and stack to hold small local variables. The users can manipulate data in "storage" and "memory" areas using the keywords *storage* and *memory* respectively.

To support the OOP features, advanced data structures and the crypto functions, we might need to extend the *irept* class, add new encoding schemes to combine various background theories in SMT-LIB [59], and add new operational modes [60].

6 Reference

[1]. Bashir, Imran. *Mastering Blockchain: A Deep Dive into Distributed Ledgers, Consensus Protocols, Smart Contracts, DApps, Cryptocurrencies, Ethereum, and More*. Third edition, PACKT, 2020.

[2]. Lantz, Lorne, and Daniel Cawrey. *Mastering Blockchain: Unlocking the Power of Cryptocurrencies, Smart Contracts, and Decentralized Applications*. First edition, O'REILLY, 2020.

[3]. Lexi Brent and Anton Jurisevic and Michael Kong and Eric Liu and Franccois Gauthier and Vincent Gramoli and Ralph Holz and Bernhard Scholz (2018). Vandal: A Scalable Security Analysis Framework for Smart Contracts. *CoRR, abs/1809.03981.*

[4]. Solorio, Kevin, et al. *Hands-on Smart Contract Development with Solidity and Ethereum: From Fundamentals to Deployment*. First edition, O'Reilly Media, Inc, 2019.

[5]. Antonopoulos, Andreas M., and Gavin Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. First edition, O'Reilly, 2019.

[6]. Bin Hu and Zongyang Zhang and Jianwei Liu and Yizhong Liu and Jiayuan Yin and Rongxing Lu and Xiaodong Lin (2021). A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. *Patterns, 2(2), 100179.*

[7]. Muhammad Izhar Mehar and Charles Louis Shier and Alana Giambattista and Elgar Gong and Gabrielle Fletcher and Ryan Sanayhie and Henry M. Kim and Marek Laskowski (2019). Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack. *J. Cases Inf. Technol., 21(1), 19–32.*

[8]. 'The DAO Attacked: Code Issue Leads to $60 Million Ether Theft - CoinDesk'. *CoinDesk: Bitcoin, Ethereum, Crypto News and Price Data*, https://www.coindesk.com/markets/2016/06/17/the-dao-attacked-code-issue-leads-to-60-million-ether-theft/. Accessed 30 Aug. 2021.

[9]. Daniel Perez and Benjamin Livshits (2019). Smart Contract Vulnerabilities: Does Anyone Care?. *CoRR, abs/1902.06710.*

[10]. *Bounties — 0x Protocol 4.0 Documentation*. https://protocol.0x.org/en/latest/additional/bounties.html. Accessed 30 Aug. 2021.

[11]. *Welcome to Mythril's Documentation! — Mythril v0.22.24 Documentation*. https://mythril-classic.readthedocs.io/en/master/.

[12]. Dowek, Gilles, and Jean-Jacques Lévy. *Introduction to the Theory of Programming Languages*. Springer, 2011.

[13]. John W. Backus and Friedrich L. Bauer and Julien Green and C. Katz and John McCarthy and Alan J. Perlis and Heinz Rutishauser and Klaus Samelson and Bernard Vauquois and Joseph Henry Wegstein and Adriaan van Wijngaarden and Michael Woodger (1960). Report on the algorithmic language ALGOL 60. *Commun. ACM, 3(5), 299–314.*

[14]. Sebesta, Robert W. *Concepts of Programming Languages*. Eleventh edition, Pearson, 2016.

[15]. Harper, Robert. *Practical Foundations for Programming Languages*. Cambridge University Press, 2013.

[16]. *Language Grammar — Solidity 0.8.7 Documentation*. https://docs.soliditylang.org/en/v0.8.7/grammar.html.

[17]. Huth, Michael, and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. 2nd ed, Cambridge University Press, 2004.

[18]. The CProver user manual, https://www.cprover.org/cbmc/doc/manual.pdf

[19]. Gabbrielli, Maurizio, et al. *Programming Languages: Principles and Paradigms*. Springer, 2010.

[20]. Cooper, Keith D., and Linda Torczon. *Engineering a Compiler*. 2nd ed, Elsevier/Morgan Kaufmann, 2012.

[21]. Aho, Alfred V., and Alfred V. Aho, editors. *Compilers: Principles, Techniques, & Tools*. 2nd ed, Pearson/Addison Wesley, 2007.

[22]. Parr, Terence. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2010.

[23]. Dirk Beyer. Second competition on software testing: Test-comp 2020. *Fundamental Approaches to Software Engineering*, LNCS, vol. 12076, 2020; 505–519.

[24]. *ESBMC*. http://www.esbmc.org/.

[25]. Lucas Cordeiro, Bernd Fischer, Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering* 2012; 38(4):957–974.

[26]. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu (2003). Bounded model checking. *Adv. Comput., 58, 117–148.*

[27]. C. Barrett and C. Tinelli, "CVC3," *Proc. Int'l Conf. Computer Aided Verification*, pp. 298-302, 2007.

[28]. R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays," *Proc. Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems*, pp. 174-177, 2009.

[29]. Feist, J., Greico, G., Groce, A.: Slither: A static analysis framework for smart contracts. In: *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. pp. 8–15. IEEE (2019)

[30]. J. Mccarthy, "Towards a Mathematical Science of Computation," *Proc. Int'l Federation of Information Processing Congress*, pp. 21-28, 1962.

[31]. E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSIC Programs," Proc. *Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168-176, 2004.

[32]. D. Gries and G. Levin, "Assignment and Procedure Call Proof Rules," ACM Trans. *Programming Languages and Systems*, vol. 2, no. 4, pp. 564-579, 1980.

[33]. A. Armando, J. Mantovani, and L. Platania, "Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers," *Int'l J. Software Tools Technology Transfer*, vol. 11, no. 1, pp. 69-83, 2009.

[34]. 'Ethereum Glossary'. *Ethereum.Org*, https://ethereum.org. Accessed 2 Sept. 2021.

[35]. C. Peng and S. Akca and A. Rajan (2019). SIF: A Framework for Solidity Contract Instrumentation and Analysis. In *26th Asia-Pacific Software Engineering Conference, APSEC 2019, Putrajaya, Malaysia, December 2-5, 2019* (pp. 466–473). IEEE.

[36]. Jorgensen, Paul. *Software Testing: A Craftsman's Approach*. Fourth edition, CRC Press, Taylor & Francis Group, 2014.

[37]. Loi Luu and Duc-Hiep Chu and Hrishi Olickel and Prateek Saxena and Aquinas Hobor (2016). Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016* (pp. 254–269). ACM.

[38]. Thomas Durieux and Joao F. Ferreira and Rui Abreu and Pedro Cruz (2020). Empirical review of automated analysis tools on 47, 587 Ethereum smart contracts. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020* (pp. 530–541). ACM.

[39]. TheDAO smart contract. http://etherscan:io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code.

[40]. S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, ''SmartCheck: Static analysis of ethereum smart contracts,'' in Proc. IEEE/ACM 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB), Jun. 2018, pp. 9–16.

[41]. *XML Path Language (XPath) 2.0 (Second Edition)*. https://www.w3.org/TR/xpath20/.

[42]. Mythril, https://github.com/ConsenSys/mythril

[43]. ‘Blockchain Technology Solutions | Ethereum Solutions’. *ConsenSys*, https://consensys.net/.

[44]. *MythX: Smart Contract Security Service for Ethereum*. https://mythx.io/.

[45]. Lopes, Bruno Cardoso, and Rafael Auler. *Getting Started with LLVM Core Libraries: Get to Grips with LLVM Essentials and Use the Core Libraries to Build Advanced Tools*. Packt Publ, 2014.

[46]. Mikhail Y. R. Gadelha and Felipe R. Monteiro and Jeremy Morse and Lucas C. Cordeiro and Bernd Fischer and Denis A. Nicole (2018). ESBMC 5.0: an industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018* (pp. 888–891). ACM.

[47]. *Clang: Clang::Tooling::ClangTool Class Reference*. https://clang.llvm.org/doxygen/classclang_1_1tooling_1_1ClangTool.html. Aug. 2021.

[48]. *Clang: Clang::ASTUnit Class Reference*. https://clang.llvm.org/doxygen/classclang_1_1ASTUnit.html.

[49]. Deitel, Paul J., and Harvey M. Deitel. *C: How to Program; with an Introduction to C++*. 8., ed.Global ed, Pearson, 2016.

[50]. ISO (2012). *ISO/IEC 14882:2011 Information technology –- Programming languages –- C++*. International Organization for Standardization.

[51]. ISO/IEC 9899:1999’. *ISO*, https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/02/92/29237.html.

[52]. *JSON for Modern C++: JSON for Modern C++*. https://nlohmann.github.io/json/doxygen/index.html.

[53]. Sefa Akca and Ajitha Rajan and Chao Peng (2019). SolAnalyser: A Framework for Analysing and Testing Smart Contracts. In *26th Asia-Pacific Software Engineering Conference, APSEC 2019, Putrajaya, Malaysia, December 2-5, 2019* (pp. 482–489). IEEE.

[54]. *Overview · Smart Contract Weakness Classification and Test Cases*. http://swcregistry.io/. Accessed 29 Aug. 2021.

[55]. *Security Considerations — Solidity 0.6.2 Documentation*. https://docs.soliditylang.org/en/v0.6.2/security-considerations.html. Accessed 3 Sept. 2021.

[56]. Derby, Esther, and Diana Larsen. *Agile Retrospectives: Making Good Teams Great*. Pragmatic Bookshelf, 2006.

[57]. Langr, Jeff, and Michael Swaine. *Modern C++ Programming with Test-Driven Development: Code Better, Sleep Better*. The Pragmatic Bookshelf, 2013.

[58]. Macneil, Dean. Cader, Aslam.*: A Practical Guide to Strategically Scaling Agile across Teams, Programs, and Portfolios in Enterprises.* PACKT PUBLISHING LIMITED, 2020.

[59]. *SMT-LIB The Satisfiability Modulo Theories Library*. http://smtlib.cs.uiowa.edu/. Accessed 30 Aug. 2021.

[60]. Felipe R. Monteiro and Mikhail R. Gadelha and Lucas C. Cordeiro (2021). Model Checking C++ Programs*. CoRR, abs/2107.01093.*