

Implementation and application of a parser for Boolector's witness format

Christoph Siller
Advisor: Christoph Kirsch

Paris Lodron University Salzburg, 5020 Salzburg , Austria
Department of Computer Science

May 31, 2020

Abstract. In this bachelor thesis, a parser for Boolector's witness format is introduced, which enables automated input validation. The toolchain takes a C* file containing read calls as input and generates a BTOR2 model of it, using Selfie's model generator. The resulting model is then fed to the Boolector-based bounded model checker BtorMC. By using SMT-solving, BtorMC is able to find error states in the model within a maximum number of executed instructions. In case an error state is found, BtorMC generates a witness. This witness is parsed in order to extract the input string that causes the C* program to run into the error. To validate that the extracted string indeed triggers an error, it is fed to the C* program while running it on Mipster. If the input string was generated correctly, Mipster runs into the predicted error. In addition to the parser, this thesis provides background information on all technologies used and the concepts they are based on.

Keywords: Selfie · Boolector · SAT · SMT · witness format · BTOR2 · symbolic execution · program verification · parsing

Table of Contents

| | |
|---------------------------------------------------------------------------|----|
| Implementation and application of a parser for Boolector's witness format | 1 |
| <i>Christoph Siller Advisor: Christoph Kirsch</i> | |
| 1 Introduction | 4 |
| 2 SAT-solving | 5 |
| 2.1 The SAT problem | 5 |
| 2.2 Solving the SAT problem | 6 |
| 3 SMT-solving | 8 |
| 3.1 The lazy SMT paradigm | 9 |
| 4 Symbolic execution | 10 |
| 5 Boolector | 12 |
| 5.1 Architecture | 12 |
| 5.2 Lambda calculus for Boolector | 13 |
| 5.3 Lemmas on demand for lambda | 15 |
| 5.4 BTOR2 | 17 |
| 5.5 BtorMC | 21 |
| 5.6 The witness format | 22 |
| 6 Selfie | 25 |
| 6.1 C* | 26 |
| 6.2 RISC-U | 28 |
| 6.3 Mipster | 28 |
| 6.4 Selfie model generator | 29 |
| 7 A parser for Boolector's witness format | 31 |
| 7.1 Objective | 31 |
| 7.2 From C* to witness | 31 |
| 7.3 Parsing the witness format | 32 |
| 7.4 Validating the error | 34 |
| 7.5 Program documentation | 34 |
| 8 Experiment/Examples | 37 |
| 8.1 Multiple read calls with division by zero | 37 |
| 8.2 Reading multiple bytes with memory constraint | 38 |
| 8.3 Invalid memory access | 39 |
| 9 Conclusion | 40 |

List of Figures

| | | |
|----|-------------------------------------------|----|
| 1 | Architecture of Boolector | 13 |
| 2 | Currying example | 14 |
| 3 | Array axioms of McCarthy | 14 |
| 4 | Lemmas on demmand | 15 |
| 5 | Function congruence axiom | 16 |
| 6 | BTOR2 instruction structure | 17 |
| 7 | BTOR2 grammar | 18 |
| 8 | BTOR2 operators | 19 |
| 9 | BTOR2 example | 20 |
| 10 | Witness grammar | 22 |
| 11 | Witness example part 1 | 23 |
| 12 | Witness example part 2 | 24 |
| 13 | C* grammar | 27 |
| 14 | RISC-U instructions | 28 |
| 15 | Generate executable model generator | 29 |
| 16 | Invoke Selfie model generator | 29 |
| 17 | List of bad-states | 30 |
| 18 | Execution command | 34 |
| 19 | <i>Validator</i> usage pattern | 35 |
| 20 | Experiment 1 | 37 |
| 21 | Experiment 2 | 38 |
| 22 | Experiment 3 | 39 |

1 Introduction

Testing source code, especially for large software projects, plays an important role in today’s software industry. However, it can be time consuming and therefore very expensive to find every last bug in a particular piece of code. This is one of the reasons why the development of efficient and automated software-testing-tools is of great interest for today’s computer science community as well as for big software companies.

A modern approach to software testing is to use a combination of *symbolic execution engines* (4) and *satisfiability modulo theories (SMT)* solvers (3) to find every error state a program’s control flow can run into (in theory). Since SMT-solvers decide satisfiability for boolean combinations of expressions in some first order theory, most modern implementations depend on an underlying SAT-solver (i.e. lazy approach) [27].

An example for a tool that combines both, a symbolic execution engine and an SMT-solver, is the Boolector (5) based bound model checker BtorMC (5.5). Boolector is an award winning SMT-solver for the theories of bit-vectors, arrays and uninterpreted functions developed at the Johannes Kepler University (JKU) Linz [24]. The input format for Boolector, as well as for BtorMC, is called BTOR2 (5.4). It is capable of modeling source code in bit-precise manner, which in turn can be executed on BtorMC. Symbolic execution is used to explore every possible execution path of the program, using symbolic rather than concrete input values. Since execution paths are represented as first-order formulas, Boolector is able to identify input values that actually lead to a bad-state [25,3]. This enables BtorMC to find error states in the given source code within a specified bound, the maximum number of instructions executed.

In order to actually check source code this way, the code must first be converted to a BTOR2 model. This is where the Selfie system (6) comes in handy. Selfie is a research and teaching project at the University of Salzburg and, among other things, contains a parser for its own programming language C* (6.1) as well as a tool to generate a BTOR2 model of a compiled C* program (6.4).

The workflow to find errors using these tools looks as follows. An arbitrary C* program which is reading values from *stdin* is parsed by Selfie and a BTOR2 model is generated from the resulting binary. This BTOR2 model is then fed to BtorMC and, if an error state is found, a witness is generated. The witness represents a concrete counter example trace, holding information about how the error occurs. The parser introduced in this thesis (7.3) allows to extract the input string that leads to an error in the C* code from the witness. It is embedded in a program (7.5) that runs the whole workflow, only requiring a C* file as input. The software also checks whether the string found actually triggers the error. For this, the C* program is run on Selfies emulator Mipster, reading the String extracted by parsing the witness from *stdin*. If the error was predicted correctly, Mipster terminates with a specific error message indicating the same error that was predicted by BtorMC.

2 SAT-solving

2.1 The SAT problem

The *satisfiability problem*, or in short *SAT problem*, is the problem of deciding whether a boolean formula can be satisfied or not, i.e. if there exists an assignment μ of the contained boolean variables so that the formula becomes true.

A boolean formula ϕ contains a set of boolean variables β where every variable $b \in \beta$ can take the values *true* or *false*. A boolean variable, negated or not negated (b or $\neg b$), occurring inside a boolean formula is called a literal l . Those literals are connected via boolean operators like \wedge (*and*) and \vee (*or*). Other boolean operators like $\{\Rightarrow, \oplus, \Leftrightarrow\}$ are allowed, however not necessary since every boolean formula can be expressed in CNF (Conjunctive Normal Form) using only \wedge, \vee, \neg . The CNF format represents a boolean formula as conjunction of clauses where each clause C contains a disjunction of literals. It is proven that it is possible to convert any boolean formula to CNF in linear time [11].

Example: Consider the following boolean formula:

$$(b_1 \Rightarrow b_2) \wedge (\neg b_1 \Rightarrow b_3) \quad (1)$$

It represents a binary *if-then-else* construct. Converted to CNF, the formula looks like this:

$$(\neg b_1 \vee b_2) \wedge (b_1 \vee b_3) \quad (2)$$

To satisfy the formula, the following variable configuration is sufficient:

$$b_1 = \text{true}, b_2 = \text{true} \quad (3)$$

Note that in this case, to satisfy ϕ , it does not matter if b_3 is assigned to *true* or *false* since b_1 is already *true*.

The most obvious example for an unsatisfiable formula is the following:

$$b_1 \wedge \neg b_1 \quad (4)$$

It does not matter whether b_1 is *true* or *false*, ϕ will always be *false*. Identifying such *contradictions* is the main purpose of a SAT-solver.

An assignment μ for β can be total or partial. Total assignments assigns *true* or *false* to every $b \in \beta$ whereas partial assignments just use a subset of β to satisfy ϕ . As for the formula in (2), in many cases a partial assignment can be enough to satisfy ϕ .

Deciding whether a boolean formula of arbitrary size is satisfiable, is one of the best known NP-complete problems. In fact, the SAT problem was one of the first NP-complete problems to be discovered [14] [11].

2.2 Solving the SAT problem

The most straightforward approach to solving the SAT problem is to brute force the formula ϕ by sequentially assigning every possible variable configuration to μ . However, this gets computationally infeasible very fast. To prove a formula ϕ with just 20 variables unsatisfiable, one would have to try $2^{20} = 1048576$ different variable configurations. Fortunately, there exist more efficient ways to solve the SAT problem.

To begin with, most modern SAT-solvers internally use the CNF format to represent boolean formulas. This is because using the CNF format transforms the problem to finding a configuration of μ that satisfies every clause C in ϕ . This problem is easier to solve since a clause is satisfied if at least one literal in C is *true*. Since every boolean formula can be transformed into CNF in linear time, using the CNF format is no problem in terms of computational effort [11].

Most state-of-the-art SAT-solvers implement the DPLL (Davis, Putnam, Logemann, Loveland) procedure to determine satisfiability. For a given formula ϕ in CNF format and an initially empty variable assignment μ , as introduced in [11] the DPLL procedure basically works as follows:

Algorithm 1 DPLL (ϕ, μ): as introduced in [11]

```

1: if  $\phi = \text{true}$  then
2:   return true                                ▷ Base Case: Satisfiable
3: else if  $\phi = \text{false}$  then
4:   return false                                ▷ Base Case: Not Satisfiable
5: else if  $\exists l \in \phi$  as unit-clause then
6:   return DPLL( $\phi(l = \text{true}), \mu \cup \{l\}$ )    ▷ Unit propagation
7: else if  $\exists l \in \phi$  occurring only positively then
8:   return DPLL( $\phi(l = \text{true}), \mu \cup \{l\}$ )    ▷ Pure literal
9: end if
10:  $l \leftarrow \text{chooseLiteral}(\phi)$ 
11: if DPLL( $\phi(l = \text{true}), \mu \cup \{l\}$ ) = true then
12:   return true                                ▷ Split
13: else
14:   return DPLL( $\phi(l = \text{false}), \mu \cup \{\neg l\}$ )  ▷ Backtrack
15: end if

```

The algorithm terminates in line 2 or 4, depending on whether the formula ϕ is satisfiable or not. If DPLL(ϕ) terminates as satisfiable, μ holds a valid total assignment for β . Otherwise μ is empty.

DPLL uses multiple approaches to determine satisfiability for ϕ :

- **Unit propagation** is used to satisfy *unit-clauses*. C is called a unit-clause if in the current partial assignment μ all but one literal in C is set to *false*. The remaining literal l in C needs to be set to *true* in order for ϕ to be satisfiable. DPLL is then called recursively with a new extended partial assignment $\mu \cup \{l = \text{true}\}$. This in turn may cause other clauses to become unit. Approximately 80% of the solving time is spent on unit propagation.
- **Pure literals** are literals for which all instances of the corresponding variable occur only negated ($l = \neg b$) or not negated ($l = b$) in ϕ . These expressions must always evaluate to *true*. Therefore, b is set to *true* for positive and *false* for negative variables.
- **Split and backtrack** is used if no unit-clause or pure literal is available. An unassigned literal $l \in \phi$ is chosen heuristically and set to *true*. DPLL is then called recursively with μ containing the newly assigned literal. If *false* is returned, l is set to *false* and DPLL is called again. This process is also known as *branching*.

Actual implementations of the DPLL algorithm may vary in the use of data structures and search algorithms.

In many cases, an *implication-graph* is built internally to make backtracking more efficient. If a conflict occurs in the solving process, the reason for unsatisfiability can be determined by analyzing the implication-graph. This in turn enables the generation of a *conflict-clause* that represents the incompatible assignments. These clauses are stored and used to avoid the same conflict in future branches.

Due to optimization of the used algorithms and data structures, memory requirements of state-of-the-art implementations are polynomial.

[11]

3 SMT-solving

Deciding satisfiability for boolean formulas is very useful for a variety of problems. However, there are many problems that cannot be interpreted as plain boolean formulas and require richer languages to be described properly.

So-called *satisfiability modulo theories (SMT) solvers* are capable of deciding satisfiability for first-order formulas with respect to some decidable first-order theory τ [11]. In order to get a better idea of what this means, some terms are discussed in the following.

First order logic, also called *predicate logic*, is defined as standard propositional logic (boolean variables with logical connectives), which is expanded through the use of quantifiers. If equality relations are allowed too, this is referred to as *first order logic with equality*. A formula expressed in first order logic is called a first order formula.

There exist not only first, but also second-, third- and higher-order logic [18].

First order theories are mathematical systems that are defined by a set of axioms in first order logic [5]. These axioms can also be defined recursively. Examples of first-order theories used in SMT are *equality and uninterpreted functions*, *linear arithmetic* or the theory of *bit-vectors and arrays* [27]. Boolector (5), for example, is an SMT-solver for the quantifier-free theories of fixed-size bit-vectors, arrays and uninterpreted functions [8].

Given these definitions, formulas used in SMT can be interpreted as boolean combinations of atomic propositions and atomic expressions in τ [27]. Therefore, compared to SAT-solving, in SMT the variables of a boolean formula in CNF are replaced with expressions of the used first-order theory.

Most SMT-solvers not only decide whether such a formula is satisfiable, but also provide a concrete model for satisfiability.

There are many different usage scenarios for SMT-solvers. They are used, for example, for interactive theorem provers and enable automated verification of source code through symbolic execution (4) [16]. In addition, SMT-solvers are often used in computer hardware design [11].

In recent years, there have been major advances in the development of SMT-solvers, mainly driven by the annual SMT competition (<http://www.smtcomp.org>). Boolector [8], Z3 [15] and Yices [17] are some examples of state-of-the-art SMT-solvers. Boolector, which plays an important role in this thesis, will be discussed in more detail in chapter 5.

3.1 The lazy SMT paradigm

Since SMT-solving is still a topic of research, there are currently many different approaches to build an efficient SMT-solver. These approaches are roughly divided into *lazy* and *eager* approaches, while most modern implementations use the lazy paradigm, as this appears to be the more efficient variant [27].

In this context, lazy means that the solver does not handle every part of the formula itself, but uses an underlying DPLL based SAT-solver to perform case analysis efficiently. Therefore, the DPLL algorithm is combined with one or more decision procedures (τ -solver) for the theory-specific part of reasoning. While the SAT-solver finds truth assignments that satisfy the boolean abstraction of the formula, consistency of these assignments with the first-order theory is checked by the τ -solver [27].

In contrast to the lazy approach, the eager approach encodes the entire SMT formula into an equivalent boolean formula, which is then fed to a SAT-solver [27]. The reduction of SMT to a boolean formula is called bit-blasting [11].

4 Symbolic execution

In many applications, it is crucial to check whether certain properties of a program hold for any possible usage scenario. A conventional approach to test such requirements, is to run a piece of software with different random input values to identify errors. Symbolic execution offers a different, more elegant approach.

In symbolic execution, several different execution paths are systematically explored at the same time without using concrete input values. Inputs are represented symbolically and actual values that may trigger an error are calculated by traversing the execution tree. Symbolic execution engines are able to, for example, check whether a piece of software can ever perform a division by zero or access an invalid memory location.

A symbolic execution engine maintains two sets of data for each explored control flow path:

- A mapping from variables to symbolic expressions or values
- A first-order boolean formula representing conditions satisfied by the branches along the path

To check whether an execution path is actually feasible or violates any of the considered properties, an SMT-solver (3) based model checking tool like BtorMC (5.5) is usually used. This combination of symbolic execution and SMT-solving is capable of determining concrete input values that will trigger an error.

Exhaustive symbolic execution denotes symbolic execution engines that indeed traverses every possible control flow path. Even though soundness and completeness is provided by this approach, there are significant drawbacks considering execution time. When exploring every path, a simple loop might lead to an exponential rise in the number of possible execution states. Such *path explosions* are one reason why exhaustively exploring every path will most likely not scale up for real-world applications.

Furthermore, commercial software is typically not self-contained. An evaluation of the entire software stack would hardly be possible and therefore makes it even more difficult to achieve soundness.

There are various design principles to cope with these issues:

- **Concolic execution** - A mixture of concrete and symbolic execution
Examples: dynamic symbolic execution [12], selective symbolic execution [13]
- **Path selection** - Heuristics to look at the most promising paths first
Examples: depth-first search, breadth-first search
- **Symbolic backwards execution**[19] - The analysis is performed in reverse direction, starting at a target point (e.g. a specific line of code) to an entry point.

Another very important design decision is the memory model that is used in the executor. Memory models can be roughly divided into four approaches:

- **Fully symbolic memory** - Memory addresses are treated as fully symbolic. This holds the most accurate description of memory behavior, but also the worst scalability.
- **Address concretization** - Binds each pointer to a single specific address instead of a big range of addresses. This may cause the engine to miss paths, but improves running time.
- **Partial memory modeling** - A middle point between the two above. Written addresses are concretized but read addresses are modeled symbolically.
- **Lazy initialization** - Intended for dynamically allocated objects. When an object is first accessed, its initial state is forked as `null`, a reference to a new object and a previously introduced object.

Symbolic execution has been a topic of research since the mid '70s, but just recently unfolds its full potential in combination with SMT-solving. Since 2008, Microsoft has been testing many of its software applications using symbolic execution, which had a major impact on the development of Windows 7.

For more detailed information on symbolic execution, see [\[3\]](#).

5 Boolector

Since we discussed the principles of SAT-solving, SMT-solving and symbolic execution in the last chapters, we will now have a look at a concrete example for an SMT-solver. In recent years, a particular SMT-solver has gained a lot of interest. It has won the SMT competition for QF_BV (quantifier-free theory of bit-vectors) and QF_AUFBV (bit-vectors with arrays and uninterpreted functions) several times [1] and is called *Boolector*. The current version of *Boolector*, which was developed at the JKU Linz, is available on GitHub [6].

A formula used as Boolector’s input can be represented either in BTOR/BTOR2 format [10,25] (5.4) or the SMT-LIB format [4], which is used in the SMT competitions. As output, Boolector returns SAT (satisfiable) or UNSAT (unsatisfiable). In terms of data types, Boolector focuses on the usage of bit-vectors and arrays. Bit-vectors can be used, for example, to express specifications directly on the word-level. Arrays, on the other hand, enable Boolector to reason about memory models in software and hardware. Both, bit-vectors and arrays, are particularly useful for software verification [8].

5.1 Architecture

A schematic representation of Boolector’s architecture can be seen in Fig.1. When a formula (BTOR(2) or SMT-LIB format) is fed to Boolector, it is first transformed into a DAG (Directed Acyclic Graph). This graph is then simplified, rewritten and passed on to the actual SMT-solver. The SMT-solver is based on one of several freely choosable SAT-solvers (Lingeling, CaDiCaL, PicoSAT, MiniSAT [6]). It uses *under-approximation* for formula refinement and an *array consistency checker* to check consistency with the first-order theory [8].

Boolector’s model generator is an additional feature that allows to generate concrete models of a given formula. Such a model represents an example that satisfies the formula and can therefore be used directly for debugging [9].

Because Boolector is written in C, it natively comes with a C interface. Since version 2.0, however, also a rich Python interface is provided. It is written as a Python module and basically enables Python programs to access the C interface.

Boolector depends on the concepts of *term rewriting*, *bit-blasting* and *lemmas on demand* to handle bit-vectors and arrays internally.

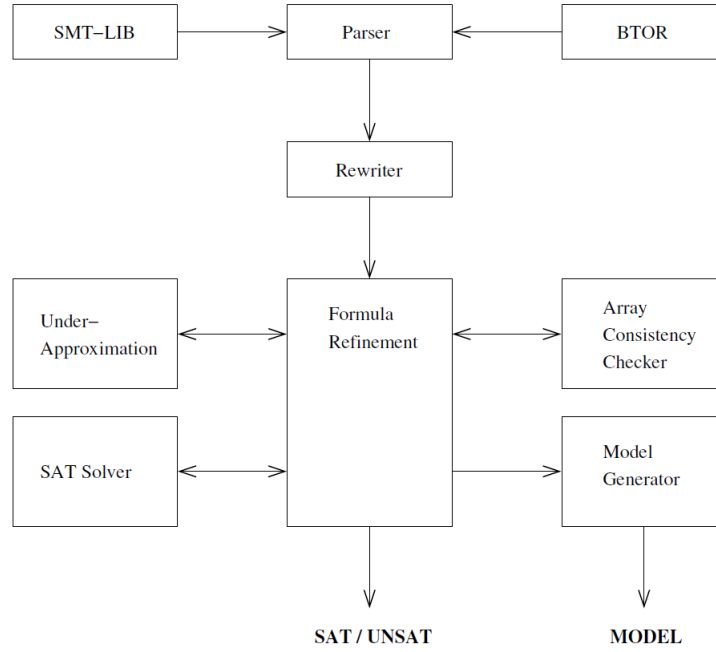


Fig. 1. Architecture of Boolector [8]

5.2 Lambda calculus for Boolector

Lemmas on demand, which is an extreme variant to lazy SMT, is one of the most important concepts of Boolector. Since version 2.0, Boolector applies its lemmas on demand approach to so-called *lambda terms* [24]. These terms are part of the *lambda calculus* (λ -calculus) formal system, which in its typical form represents a Turing-complete model of computation. In Boolector however, there are two fundamental restrictions to avoid Turing-completeness for the sake of decidability.

1. **Functions may not be recursive (non-recursive).**
This helps to keep λ -term handling as simple as possible.
2. **Arguments to functions may not be functions (non-extensional).**
Limits the λ -term handling to non-higher order functions.

If those restrictions are relaxed, the considered λ -calculus will turn Turing-complete which makes the decision problem undecidable in general [26].

λ -calculus is based on the usage of uninterpreted functions and variables, which can either be free or bound. A λ -term λ_x such as $\lambda_x.t(x)$ consists of the bound variable x and the term $t(x)$, which represents the scope of x . In λ -calculus, a λ -term can only have one bound variable. Therefore, only functions with one parameter are permitted in this system. Functions with multiple parameters, however, can be transferred into a chain of single-parameter terms. This process is called *currying* (Fig.2).

Formula: $f(x, y) := x + y$
 Chained λ -term: $\lambda_x.\lambda_y.x + y$

Fig. 2. Currying example

Another very important concept in λ -calculus is β -reduction. It describes the process of replacing bound variables with their actual parameter terms. Boolec-tor supports *full* and *partial* β -reduction. The difference between those two is the depth up to which λ -terms are expanded. Whereas full β -reduction reduces every λ -term in the scope of λ_x , partial β -reduction only reduces the top most λ -term. The decision to implement two versions of β -reduction was made to avoid an exponential blow-up through full β -reduction and still have the advantage of significant simplification through rewriting.

Boolec-tor's λ -calculus focuses on *many-sorted languages*. These languages use several different data types, called *sorts* [28], whereby each sort is represented as a bit-vector of different bit-width. Arrays are modeled as a mapping ($\tau_i \rightarrow \tau_j$) from one sort τ_i (the array index) to another sort τ_j (the corresponding array element) and are defined by the three axioms of McCarthy [23]:

$$i = j \rightarrow \text{read}(a, i) = \text{read}(a, j) \quad (A1)$$

$$i = j \rightarrow \text{read}(\text{write}(a, i, e), j) = e \quad (A2)$$

$$i \neq j \rightarrow \text{read}(\text{write}(a, i, e), j) = \text{read}(a, j) \quad (A3)$$

Fig. 3. Array axioms of McCarthy

Therefore, an array of sort ($\tau_i \rightarrow \tau_e$) is represented as a λ -term of the form $\lambda j.t(j)$ where j is of sort τ_i and $t(j)$ is of sort τ_e . To model array **read** and **write** operations, uninterpreted functions and if-then-else (ite) constructs are used [26].

$$f_a(i) \equiv \text{read}(a, i) \equiv \text{read}(\lambda j.t(j), i) \equiv (\lambda j.t(j))(i)$$

$$\text{write}(a, i, e) \equiv \lambda j. \text{ite}(i = j, e, f_a(j))$$

One of the key advantages of this kind of λ -calculus is the ability to reason about memory-models using arrays. For example, the C standard library functions `memset` and `memcpy` can be modeled using arrays and if-then-else constructs.

`memset(a, i, n, e)` sets each element of array a in the range $[i, i + n)$ to the value e . This behavior is represented by the λ -term:

$$\lambda j. \text{ite}(i \leq j \wedge j < i + n, e, f_a(j))$$

For `memcpy(a, b, i, k, n)`, which copies the elements in range $[i, i + n)$ from array a to array b beginning at index k , the λ -term looks like this:

$$\lambda j. \text{ite}(k \leq j \wedge j < k + n, f_a(i + j - k), f_b(j))$$

5.3 Lemmas on demand for lambda

Now that we have discussed the basics of λ -calculus, we will have a look at Boolector's *lemmas on demand* approach which makes use of non-recursive λ -terms. Fig.4 represents a top-level view of the used lemmas on demand approach.

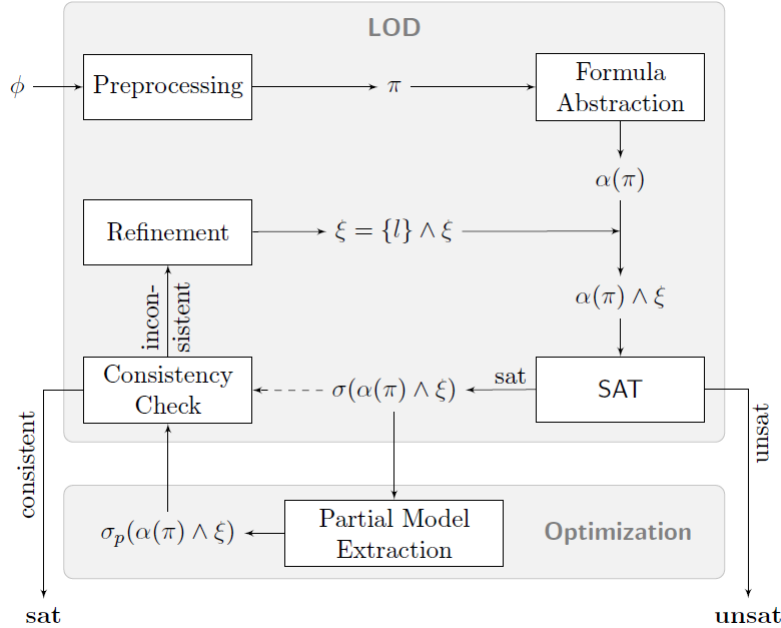


Fig. 4. Lemmas on demand for lambdas [24]

As an initial step, the formula ϕ is fed as input to the lemmas on demand decision procedure. ϕ represents a directed acyclic graph in which all array variables and array operations have been replaced by corresponding λ -terms, and inequality $a \neq b$ is represented as $\neg(a = b)$ [26] [9].

In the *preprocessing* step, a new λ variable and two new read terms $read(a, \lambda), read(c, \lambda)$ are introduced for each equality between array terms $a = c \in \phi$. Also a new top-level constraint is added:

$$a \neq c \rightarrow \exists \lambda. read(a, \lambda) \neq read(c, \lambda)$$

Furthermore, new top level constraints are added for each write term $write(a, i, e) \in \phi$:

$$read(write(a, i, e), i) = e$$

These additional constraints simplify reasoning about array inequality and consistency in write terms. Adding additional top-level constraints c_i, \dots, c_n to ϕ results in the formula π , which is defined as follows:

$$\pi := \phi \wedge \bigwedge_{i=1}^n c_i$$

After initial preprocessing, a *bit-vector skeleton* $\alpha(\pi)$ is generated as an abstraction of the preprocessed formula π . The abstraction function α maps each read and equality term to a new abstraction variable and each non-array variable or symbolic constant to itself [9].

In each iteration of the lemmas on demand procedure, a formula refinement ξ is added to the abstraction ($\xi = true$ for the first iteration) and satisfiability is determined by the underlying SAT-solver. If the SAT-solver concludes unsatisfiable, the lemmas on demand procedure also returns unsatisfiable since $\alpha(\pi) \wedge \xi$ is an over-approximation of ϕ . However, if the SAT-solver concludes satisfiable, a valid assignment μ is returned by the SAT-solver. Since the assignment only refers to $\alpha(\pi) \wedge \xi$, consistency needs to be checked with respect to the preprocessed formula π . [26]

To determine whether a satisfactory assignment μ can be expanded to a valid assignment for π , the assignment of the abstraction variable needs to be consistent with the corresponding value obtained by partial β -reduction. Also the function congruence axiom (Fig.5) must not be violated.

$$\forall \bar{x}, \bar{y}. \bigwedge_{i=1}^n x_i = y_i \rightarrow f(\bar{x}) = f(\bar{y})$$

Fig. 5. The function congruence axiom states, that a given function always returns the same value for the same set of input values.

In order to check the function congruence axiom, a hash table is used to map λ -term symbols to function applications. If the consistency checker does not find

any violations, the lemmas on demand procedure concludes that π is satisfiable. Otherwise, a symbolic lemma is generated and added to the formula refinement [26]. The type of lemma to be generated depends on the type of inconsistency. For example, if the first array axiom (A1) (Fig.3) is violated:

$$\mu(\alpha(i)) = \mu(\alpha(j)) \wedge \mu(\alpha(\text{read}(a, i))) \neq \mu(\alpha(a, j))$$

the following lemma is generated and added to the refinement:

$$i = j \Rightarrow \text{read}(a, i) = \text{read}(a, j)$$

For more details on lambda calculus and the lemmas on demand decision procedure, see [26,9].

5.4 BTOR2

The BTOR2 format [25] is one possible input format for Boolector and is the successor of the bit-precise word-level format BTOR [10]. BTOR can in turn be viewed as a word-level generalization of AIGER [7], which is used in hardware model checking. BTOR is capable of handling bit-vectors of arbitrary size in the form of variables, constants and one-dimensional arrays as well as modeling registers and memory. It represents an alternative for the SMT-LIB format [4] and can also capture model checking problems.

The structure of BTOR and BTOR2 is basically the same: a sequence of instructions without forward references or control flow options. An instruction in general has the following form:

| | | | |
|-------------------|----------|-------------|--------|
| unique identifier | operator | result size | {args} |
|-------------------|----------|-------------|--------|

Fig. 6. BTOR2 instruction structure

Intermediate variables, referenced by the *unique identifier*, are used to store the result of an operation. The *result size* specifies the number of bits the result is represented by. The number of arguments (*args*) depends on the *operator* used. Because of their simple structure, BTOR and BTOR2 can be easily parsed by a single pass compiler.

BTOR2 generalizes BTOR and extends it by the usage of *sorts*. Furthermore, in BTOR registers and memory were implicitly zero-initialized and uninitialized respectively. BTOR2's sequential extension in contrast, allows the explicit initialization of memory and registers.

The grammar of BTOR2 in EBNF can be seen in Fig.7.

```

symbols: ";", "0", "1", "-", "const", "constd", "consth", "input", "one", "ones", "zero",
        "state", "bitvec", "array", "sort", "init", "next", "bad", "constraint",
        "fair", "output", "justice", "\n"

num      = positive unsigned integer (>0)
uint     = unsigned integer
string   = sequence of whitespace and printable characters without '\n'
symbol   = sequence of printable characters without '\n'
alphanum = [0-9a-fA-F]

comment  = ";" string .
nid      = num .
sid      = num .
const    = "const" sid ( "0" | "1" ) { "0" | "1" } .
constd   = "constd" sid [ "-" ] uint .
consth   = "consth" sid alphanum { alphanum } .
input    = ( "input" | "one" | "ones" | "zero" ) sid | const | constd | consth .
state    = "state" sid .
bitvec   = "bitvec" num .
array    = "array" sid sid .
node     = sid "sort" ( array | bitvec )
          | nid ( input | state )
          | nid opidx sid nid unit [ unit ]
          | nid op sid nid [ nid [ nid ] ]
          | nid ( "init" | "next" ) sid nid nid
          | nid ( "bad" | "constraint" | "fair" | "output" ) nid
          | nid "justice" num { nid } .
line     = comment | node [ symbol ] [ comment ] .
btor     = { line "\n" } .

```

Fig. 7. Grammar of the BTOR2 format in EBNF [25]

First things first: Line comments start with ';'.

The **sort** keyword allows the definition of arbitrary bit-vector and array sorts like floating-point or multidimensional arrays. This renders the keywords **var**, **array** and **acond** of BTOR obsolete. The declaration of bit-vector and array variables of a given sort is done using the **input** keyword.

Identifiers are divided into *node identifiers* (nid) and *sort identifiers* (sid). In order to be able to distinguish between nodes and sorts, no identifier can appear in both sets. Identifiers usually appear in ascending order.

The operations `const`, `constd`, `consth` create binary, decimal or hexadecimal constants. The constant values 1, -1 and 0 are represented by `one`, `ones` and `zero`.

BTOR2's sequential extension allows to specify memory and registers using the `state` keyword. An explicit definition of their initialization is enabled by the `init` keyword. By using bit-masks, even partial initialization can be achieved. Using the `next` keyword with the current and next states as argument, defines a transaction function for memory and registers.

As in AIGER [7], BTOR2 supports `bad` and `justice` state properties. Global invariant constraints and fairness constraints can be introduced by using the `constraint` and `fair` keyword respectively.

The non-terminals `opidx` and `op` reference indexed and not indexed operators in Fig.(8), where the semantics of most operators are identical to that of the corresponding operator in SMT-LIB.

| Operator | Class | Mapping |
|----------------------------------------|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| indexed | | |
| [su]ext w | (un)signed extension | $\mathcal{B}^n \rightarrow \mathcal{B}^{n+w}$ |
| slice $u\ l$ | extraction, $n > u \geq l$ | $\mathcal{B}^n \rightarrow \mathcal{B}^{u-l+1}$ |
| unary | | |
| not | bit-wise | $\mathcal{B}^n \rightarrow \mathcal{B}^n$ |
| inc, dec, neg | arithmetic | $\mathcal{B}^n \rightarrow \mathcal{B}^n$ |
| redand, redor, redxor | reduction | $\mathcal{B}^n \rightarrow \mathcal{B}^1$ |
| binary | | |
| iff, implies | boolean | $\mathcal{B}^1 \times \mathcal{B}^1 \rightarrow \mathcal{B}^1$ |
| eq, neq | (dis)equality | $\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{B}^1$ |
| [su]gt, [su]gte, [su]lt, [su]lte | (un)signed inequality | $\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^1$ |
| and, nand, nor, or, xnor, xor | bit-wise | $\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^n$ |
| rol, ror, sll, sra, srl | rotate, shift | $\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^n$ |
| add, mul, [su]div, smod, [su]rem, sub | arithmetic | $\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^n$ |
| [su]addo, [su]divo, [su]mulo, [su]subo | overflow | $\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^1$ |
| concat | concatenation | $\mathcal{B}^n \times \mathcal{B}^m \rightarrow \mathcal{B}^{n+m}$ |
| read | array read | $\mathcal{A}^{\mathcal{I} \rightarrow \mathcal{E}} \times \mathcal{I} \rightarrow \mathcal{E}$ |
| ternary | | |
| ite | conditional | $\mathcal{B}^1 \times \mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^n$ |
| write | array write | $\mathcal{A}^{\mathcal{I} \rightarrow \mathcal{E}} \times \mathcal{I} \times \mathcal{E} \rightarrow \mathcal{A}^{\mathcal{I} \rightarrow \mathcal{E}}$ |

Fig. 8. Operators supported by BTOR2 as listed in [25]. \mathcal{B}^n represents bit-vectors of size n , $\mathcal{A}^{\mathcal{I} \rightarrow \mathcal{E}}$ represents an array with index \mathcal{I} and element \mathcal{E} .

In order to get a better understanding of the use of BTOR2, the following example shows the pathway of an input-variable through the program until a certain point is reached. All lines are part of an actual BTOR2 file used in 8, but in reality there are hundreds of lines between some of them.

```

1 1 sort bitvec 1 ; boolean
2 2 sort bitvec 64 ; 64-bit machine word
3 23 constd 2 3
4 28 constd 2 8
5 73 sort bitvec 24 ; 3 bytes
6 83 input 73 ; 3 bytes
7 93 uext 2 83 40 ; 3 bytes
8 40001160 sub 2 112 110 ; $a2 - $a0
9 40001161 ugte 1 40001160 28 ; $a2 - $a0 >= 8 bytes
10 ; read 8 bytes if $a2 - $a0 >= 8 bytes, or else $a2 - $a0 bytes
11 40001162 ite 2 40001161 28 40001160
12 ; unsigned-extended 3-byte input if increment == 3
13 40001172 eq 1 40001162 23
14 40001173 ite 2 40001172 93 40001171 ; unsigned-extended 3-byte

```

Fig. 9. BTOR2 example

In lines 1 and 2, which are actually also the first lines of the BTOR2 file, sorts for boolean variables and machine-words are defined. Machine-words with the constant values 3 and 8 are initialized in lines 3 and 4 and are accessible via node-IDs 23 and 28. Sort-ID 73 is defined as a 3-byte bit-vector, and an input-variable of that sort is declared for node-ID 83. In line 7, the input variable is unsigned-extended by 40 bits and is cast to machine-word. Node-ID 93 now holds the 3-byte input value.

In line 8 the number of input bytes, in this case for a read call, is calculated. It is checked whether the result is greater than or equal to 8. If this comparison returns true, the number of bytes to be read is set to 8, otherwise it is set to the previously calculated value with ID 40001160.

If the number of bytes read equals 3, in line 13 node 93 is selected as input-variable for the read call. Otherwise, the value of node 40001171 is used, which holds the result of the same comparison for a 2-byte input. Lines 13 and 14 are part of an if-else construct that determines which input-variable corresponds to the number of bytes read by the read function. Since the number of bytes read is fixed, only one input-variable can be selected. This is also the reason why only one input-variable should be non-zero, which will be of interest later on in 5.6.

5.5 BtorMC

With Selfie's model generator (6.4), it is possible to transfer source code into a BTOR2 model that contains state transactions to simulate execution. To find reachable bad-states in the BTOR2 model and thus errors in the source code, an SMT-based *model checker* is required.

BtorMC is a *reference bounded model checker*, which is based on Boolector 3.0. It is designed for checking bad-state properties for BTOR2 models with registers and memory and produces a *witness* in case the model is satisfiable. Such a witness represents an initialized finite path that reaches a bad-state in the BTOR2 file and satisfies all invariant constraints [25].

The term *bounded* refers to the maximum number of state transitions the generated path consists of. This limitation in the number of execution steps enables bounded model checkers (BMC) to find counter examples of minimal length much faster than "unbound" model checkers. The drawback, however, is that completeness is sacrificed for this step limit. Bad-states and counter examples that require more state transitions cannot be found. In BtorMC the maximum number of state transitions is adjusted using the `-kmax` option [22].

As part of the workflow presented in this thesis, a BTOR2 model of some C* code (6.1) is fed into BtorMC. The witness generated in this way contains the configuration of each input variable for each execution step. In this context, the number of state transitions corresponds to the number machine instructions required to reach the bad-state.

BtorMC unrolls the BTOR2 model using symbolic substitution of current state expressions with next state functions. In this way, BtorMC actually acts as a symbolic execution engine for BTOR2 models. Bad-states are found by incremental SMT-solving for each execution step. If a formula is satisfiable, the satisfying assignment returned by the SMT-solver is translated into a concrete counterexample trace. Otherwise, the bound is increased and the process is repeated [22].

To check a BTOR2 model using BtorMC, the following command is used:

```
btormc -kmax <step_limit> <Btor2_file>
```

With the enormous progress in SMT-solving in recent years, BMC became one of the most important application of SMT. BMC is also widely used in the EDA (Electronic Design Automation) industry [22].

BtorMC can be found in the Boolector GitHub repository here: [6]. For more information on BMC in general, see [22].

5.6 The witness format

This section describes the BTOR2 witness format with regard to the parser presented in 7. In this way it is possible to give concrete examples of how a witness should be interpreted in order to obtain input values that lead to a bad-state.

If a BTOR2 model (5.4), representing some program's source code, is checked by BtorMC and a bad-state is actually reached, BtorMC generates a *witness*. As stated in 5.5, this witness represents a finite number of execution steps that lead to a bad-state in the BTOR2 model. Each execution step is represented as *time-frame*, containing valid input assignments and memory states at specific points in execution time. Regarding the witness format, time is measured in assembly instructions. For example, the timeframe with $t = 42$ contains information about input and memory state for the 42nd executed assembly instruction.

The grammar of the witness format looks as follows:

```
symbols: "\n", ".", ";", "sat\n", "b", "j", "#", "@", "0", "1", "[", "]"

string = sequence of whitespace and printable characters without "\n" .
symbol = sequence of printable characters without "\n" .
uint   = digit { digit } .

witness      = { comment "\n" } | header { frame } "." .
comment      = ";" string .
header       = "sat\n" { prop } "\n" .
prop         = ( "b" | "j" ) uint .
frame        = [ state_part ] input_part .
state_part   = "#" uint "\n" model .
input_part   = "@" uint "\n" model .
model        = { comment "\n" | assignment "\n" } .
assignment   = uint ( bv_assignment | array_assignment ) [ symbol ] .
bv_assignment = binary_string .
binary_string = ( "0" | "1" ) { "0" | "1" } .
array_assignment = "[" binary_string "]" binary_string .
```

Fig. 10. Grammar of the witness format in EBNF [25]

Each witness starts with the string `sat`, followed by a list of properties. In this list, all bad-states reached by the witness are described using the letter `b` followed by an integer. The integer represents the position of the bad-state in the BTOR2 model. Property `b2`, for example, refers to the third bad-state defined in the BTOR2 model (counting starts at 0). In general, the witness format can also describe *justice states* (e.g. `j2`), but those states are not used in the BTOR2 models generated by Selfie.

Timeframes start right after the list of properties and are divided into a *state part* and an *input part*. The input part, starting with `@t`, consists of values assigned to the input-variables at the current point in execution time. For the BTOR2 models generated by Selfie, input-variables represent values read by a `read` call. The state part, starting with `#t`, contains memory assignments required in order to reach the bad-state. In contrast to the input part, the state part can be omitted if not needed [25].

For a better understanding, consider the following example:

The witness shown in Fig. 11 and Fig. 12 was generated by feeding the example code given in 8.2 to BtorMC. In line 1, the witness starts with the string `sat`. The following line holds the *list of properties*, which in this case only contains the bad-state number 1 (non-zero exit code).

The first timeframe starts at line 3. Lines 4 and 5 (actually one line, line break for better readability) hold a state assignment. This means, the memory location represented by the first bitstring (66096) needs to hold the value represented by the second bitstring (2). Starting at line 7, the input values for this frame are given. Those are zero here, since the first assembly instruction is not a **read** call.

[illegible]

Fig. 11. Start of a witness containing the list of properties and the first timeframe

Zero frames continue for several instructions until finally at frame 90 the `read` call is executed. Since a `read` function reads an adjustable number of characters, the 8 different inputs represent varying numbers of bytes. In this case, two bytes are read, whereby `input1` holds the bytes 00110010 (char '2') and 00110100 (char '4'). With respect to Selfie's (6) internal memory layout, this means that the string "42" triggers an error when read.

```

1  #90
2  @90
3  0 00000000 input0@90
4  1 0011001000110100 input1@90
5  2 000000000000000000000000 input2@90
6  3 0000000000000000000000000000 input3@90
7  4 00000000000000000000000000000000 input4@90
8  5 000000000000000000000000000000000000 input5@90
9  6 00000000000000000000000000000000000000 input6@90
10 7 0000000000000000000000000000000000000000 input7@90
11 ...
12 .

```

Fig. 12. Timeframe representing the read instruction

After the **read** call, there are several null frames again. Finally, the witness ends with the **'.'** symbol.

For the BTOR2 models generated by Selfie, there is an additional constraint regarding a valid witness file. Since the input variables represent the number of bytes read by the **read** function, only one input can be non-zero. This behavior is also implemented in the parser (7.3) and can be seen in a BTOR2 model in Fig. 9.

6 Selfie

An essential part of the presented error checking workflow, is to convert a piece of code to a BTOR2 model using the Selfie system. But what exactly is the Selfie system? This chapter aims to answer that question.

Primarily, Selfie is a tool to teach basic ideas of computer science to undergraduate and graduate students. Mainly topics of compiler construction and operating systems are covered, but also SAT-solving, symbolic execution and other concepts are implemented. Since Selfie is an open source project, its source code is available on GitHub [20].

The core functionality of Selfie is implemented in a self-contained file currently holding approximately 9k lines of C* code (6.1). It was originally developed for a 32bit MIPS architecture with 64MB of memory and signed integers, but has been ported to a 64bit RISC-V architecture with 4GB of memory and unsigned integers in recent years. Selfie mainly consists of a self-compiling C* compiler targeting RISC-U assembly code (6.2), a self-executing RISC-U emulator (Mipster 6.3) and a self-hosting hypervisor (Hypster) virtualizing the emulated machine [2]. In addition, Selfie contains many small tools and projects, such as a RISC-U disassembler, a SAT-solver and a BTOR2 model generator (6.4), some of which have recently been transferred to separate files. Today, besides its educational purpose, Selfie is also used as a research platform and is often part of students' final thesis, which is why it is continuously evolving.

Selfie can be run on the command line of any established operating system. The only thing that is needed in addition to Selfie itself is a C compiler like `cc` or `gcc`. To compile Selfie for the first time, the `make` file, which is also stored in the GitHub repository, can be used. From then on, Selfie can be compiled by itself. As of June 2020, the Selfie usage pattern looks as follows:

```
usage: ./selfie { -c { source } | -o binary |
[ -s | -S ] assembly | -l binary }
[ ( -m | -d | -r | -y ) 0-4096 ... ]
```

The `-c` option calls the C* compiler, taking a C* program as input, while the `-o` option specifies the file in which the binary is to be stored. If the compiler is called without the `-o` option, the binary is only stored in memory. The `-l` option, as opposed to the `-o` option, loads a binary into memory. Any binary stored in memory can be executed on Mipster using the `-m` option, where an integer indicates the number of MB allocated as Mipster's virtual memory. Alternatively, the hypervisor *Hypster* can be invoked using the `-y` option. The options `-S` and `-s` call the disassembler with or without additional information in the assembly file respectively [21].

To compile a C* file and store the resulting binary, the following command can be used:

```
./selfie -c file.c -o file
```

Due to Selfie’s self-executing nature, the same goal can be achieved through more complex commands, such as:

```
./selfie -c selfie.c -m 3 -c selfie.c -m 2 -c file.c -o file
```

While this is very complicated compared to the first command, it’s still a good example of what Selfie can do. For this command, Selfie compiles itself and executes itself on Mipster with 3MB of memory. This Selfie instance running on Mipster compiles itself again and executes Selfie on Mipster with 2MB of memory. The Selfie instance running on the second Mipster instance, now compiles `file.c` and stores the resulting binary. The result is the same as in the first example, but the execution is very slow.

If the second Mipster instance is replaced by a Hypster, execution time is reduced to just a few seconds. This is due to the difference between interpretation and virtualization, which is one of the first things students learn with Selfie. For more information on Selfie see [21,2,20].

6.1 C*

The programming language compiled by Selfie, and in which Selfie itself is written in, is called *C**. C* is a tiny but powerful subset of C that focuses on the core functionality of C and is designed for self-referential systems. C* contains the six keywords `uint64_t`, `void`, `if`, `else`, `while`, `return` and the five statements *assignment*, *while loop*, *if-then-else*, *procedure call* and *return*. Standard arithmetic and comparison operators are also included (`+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `<=`, `>`, `>=`). To bootstrap Selfie, the five system calls `exit`, `malloc`, `open`, `read` and `write` are used. The only data types in C* are 64bit integer (`uint64_t`) and pointer to 64bit integer (`uint64_t*`). Memory access is controlled only by dereferencing (`*` operator) and pointer arithmetic.

The grammar of C* in EBNF is provided in the GitHub repository [20] and can be seen in Fig.13.

```

C* Keywords: 'uint64_t', 'void', 'if', 'else', 'while', 'return'
C* Symbols: 'integer', 'character', 'string', 'identifier', '(', '(',
';', '(', ')', '{', '}', '+', '-', '*', '/', '%', '=', '==',
'!=', '<', '<=', '>', '>='

with:

integer    = digit { digit } .
character  = "'" printable_character "'" .
string     = "\"" { printable_character } "\"" .
identifier = letter { letter | digit | "_" } .
digit      = "0" | ... | "9" .
letter     = "a" | ... | "z" | "A" | ... | "Z" .

C* Grammar:

cstar      = { type identifier [ "=" [ cast ] [ "-" ] literal ] ";"
              | ( "void" | type ) identifier procedure } .
type        = "uint64_t" [ "*" ] .
cast        = "(" type ")" .
literal     = integer | character .
procedure   = "(" [ variable { "," variable } ] ")"
              ( ";" | "{" { variable ";" } { statement } "}" ) .
variable    = type identifier .
statement   = call ";" | while | if | return ";" |
              ( [ "*" ] identifier | "*" "(" expression ")" )
              "=" expression ";" .
call        = identifier "(" [ expression { "," expression } ] ")" .
expression  = simpleExpression [ ( "==" | "!=" | "<" | ">"
              | "<=" | ">=" ) simpleExpression ] .
simpleExpression = term { ( "+" | "-" ) term } .
term        = factor { ( "*" | "/" | "%" ) factor } .
factor      = [ cast ] [ "-" ] [ "*" ]
              ( identifier | call | literal | string | "(" expression ")" ) .
while       = "while" "(" expression ")"
              ( statement | "{" { statement } "}" ) .
if          = "if" "(" expression ")"
              ( statement | "{" { statement } "}" )
              [ "else" ( statement | "{" { statement } "}" ) ] .
return      = "return" [ expression ] .

```

Fig. 13. C* grammar in EBNF ([20])

Additional functionality such as arrays, structs and bitwise operators are usually implemented as assignments of the compiler class.

6.2 RISC-U

As mentioned earlier, Selfie’s C* compiler targets RISC-U assembly code. RISC-U is a tiny 64bit subset of RISC-V featuring only 14 instructions, mainly for unsigned arithmetic. The instructions and their functionality as pseudocode are shown in Fig.14.

| Instruction | Name | Pseudocode |
|------------------|------------------------|-------------------------------------------------|
| ADD rd,rs1,rs2 | Addition | $rd \leftarrow rs1 + rs2$ |
| SUB rd,rs1,rs2 | Subtraction | $rd \leftarrow rs1 - rs2$ |
| MUL rd,rs1,rs2 | Multiplication | $rd \leftarrow rs1 * rs2$ |
| DIVU rd,rs1,rs2 | Unsigned Division | $rd \leftarrow rs1 / rs2$ |
| REMU rd,rs1,rs2 | Remainder Unsigned | $rd \leftarrow rs1 \bmod rs2$ |
| SLTU rd,rs1,rs2 | Set Less Than Unsigned | $rd \leftarrow rs1 < rs2$ |
| ADDI rd,rs1,imm | Add Immediate | $rd \leftarrow rs1 + imm$ |
| JALR rd,imm(rs1) | Jump and Link Register | $rd \leftarrow pc + 4, pc \leftarrow rs1 + imm$ |
| LD rd,imm(rs1) | Load Double | $rd \leftarrow memory[rs1 + imm]$ |
| BEQ rs1,rs2,imm | Branch Equal | $if(rs1 == rs2) pc \leftarrow pc + imm$ |
| JAL rd,imm | Jump and Link | $rd \leftarrow pc + 4, pc \leftarrow pc + imm$ |
| SD rs2,imm(rs1) | Store Double | $memory[rs1 + imm] \leftarrow rs2$ |
| LUI rd,imm | Load Upper Immediate | $rd \leftarrow imm * 2^{12}$ |
| ecall | System Call | syscall number in a7, parameters in a0 – a2 |

Fig. 14. RISC-U instructions [20] (based on RISC-V [29])

All RISC-U instructions except system calls affect at most one 64bit machine word in either registers or memory. Of course this excludes the program counter, which is increased by 4 after every instruction. RISC-U binaries generated by Selfie are proper ELF binaries and even compatible with the official RISC-V toolchain [2].

6.3 Mipster

The name *Mipster* originates from the MIPS architecture Mipster was originally built for. However, since Selfie switched to RISC architecture, Mipster is an emulator for RISC-U code. Because it is a part of Selfie, Mipster can emulate itself, allowing multiple Mipsters to be stacked like we did in example two in 6. The memory size of a Mipster instance is defined by the integer number after Selfie’s -m option. Mipster’s debug mode, which prints each executed machine instruction and its effect on the machine state, can be invoked using the -d option.

6.4 Selfie model generator

Selfie's model generator is an essential part of the validation workflow as it enables Selfie to generate a BTOR2 model based on a RISC-U binary. Unfortunately, there is no literature on this part of Selfie because the model generator is a rather new feature. Therefore, the information in this chapter is based solely on the source file `modeler.c`, which can be found in the `tools` directory of the Selfie repository [20].

The goal of the Selfie model generator is to translate a given RISC-U binary into an equivalent BTOR2 model. This model can then be fed into BtorMC (5.5) in order to be checked for bad-states and thus errors in the source code. A C* program, whose RISC-U binary is used for this, usually contains *read-calls*. This is because the generated models are designed to find out by reading a certain input value whether it is possible to put the C* program into an error state. Error states taken into account are, for example, division by zero or a given *bad-exit-code*.

In order to get an executable model generator, Selfie's `Makefile` can be used to link Selfie as library to `modeler.c`:

```
make tools/modeler.selfie
```

Fig. 15. Generate executable model generator

To invoke the model generator and set 42 as bad exit code, the following command is used:

```
./modeler.selfie -c file.c - 42
```

Fig. 16. Invoke Selfie model generator

As a first step in creating a BTOR2 model, a new file is prepared. Its name and directory location are the same as for the original C* file, but with the extension *.btor2*. Next, the *bad-ext-code* is set and some general information is written as a comment in the BTOR2 file.

Then some lines of code are emitted that are the same for each BTOR2 model and contain the following:

1. Definition of general data types such as boolean variables or 64bit machine words.
2. Memory bounds and initial values for code and data segment.
3. Definition and initialization of 32 general purpose registers.
4. Definition of 8 input-variables.

Since the number of bytes to be read is variable for the syscall `read(fd, buf, count)`, the length of the generated byte array is also variable. For this reason, the eight input-variables represent byte arrays of different lengths (1-8). During the SMT-solving process, Boolector ultimately checks whether there exists a value for those variables that trigger a bad-state.

In the next step, the program counter is initialized and its path throughout the program is arranged. Data segment and stack are modeled and loaded into memory. The data flow is then modeled by fetching one RISC-U instruction after another and encoding it in corresponding BTOR2 statements.

Selfie's system calls are the next to be modeled. They are extended by additional validity checks to recognize bad-states for invalid syscall ids, invalid memory access and the bad exit code. For the `read` syscall the number of bytes to read is determined and the corresponding input-variable is assigned.

Last but not least, the control flow part is modeled. Here the encoded RISC-U instructions are fed with actual data from registers and the program counter is manipulated to execute instructions in the expected order. Afterwards, some checks for division and remainder by zero, as well as checks for valid memory access, are implemented.

The following bad-states with their corresponding model internal number are recognized:

| Identifier | Bad-state |
|------------|---------------------------------------|
| b0 | ecall invalid syscall id |
| b1 | non-zero exit code |
| b2 | division by zero |
| b3 | remainder by zero |
| b4 | memory access below lower bound |
| b5 | memory access at or above upper bound |
| b6 | word-unaligned memory access |
| b7 | memory access below lower bound |
| b8 | memory access at or above upper bound |
| b9 | word-unaligned memory access |

Fig. 17. List of bad-states

Note that the bad-states *b4*, *b5*, *b6* are the same as *b7*, *b8*, *b9*. This is because an invalid memory access can occur both in the control-flow part and in a syscall. Since bad-states are numbered according to their occurrence in the model, bad-states that occur several times are assigned several numbers.

7 A parser for Boolector's witness format

7.1 Objective

The purpose of the software presented in this chapter is to automatically check whether a given C* program can be put into an error state (respectively BTOR2 bad-state) by reading a certain input string via read calls. Everything we saw in the previous chapters comes together in this program.

In order to find error states in the C* (6.1) file, Selfie's model generator (6.4) is used to generate a BTOR2 (5.4) file. The Boolector (5) based model checker BtorMC (5.5) then searches for reachable bad-states in that BTOR2 model. If a bad-state is reached, a witness (5.6) file is generated. This witness is then parsed to obtain the input string that triggers the bad-state.

To validate that this string actually causes the program to run into the predicted error, Selfie (6) is used to execute the C* file on Mipster (6.3) with the string as input. The generated output text is then checked for the expected error message.

All this functionality is implemented in a single python script called `validator.py` (7.5). It uses Selfie and BtorMC to generate files and check outputs, and is capable of parsing the witness format.

7.2 From C* to witness

The starting point of the process is a single specified C* file, which is the only mandatory argument of *Validator*. In order for Selfie to be able to generate a BTOR2 model, the C* program must first be compiled. This is because the model generator is actually working with a RISC-U binary rather than C* code. After compiling the file with the `-c` option, the resulting binary is stored in Selfie's binary buffer. The model generator then works with this buffer.

The command to compile a C* file and generate a BTOR2 model of the resulting RISC-U binary can be seen in Fig.16. The model generator takes a single integer value as mandatory argument. This value represents the exit code that should trigger the *non-zero exit code* bad-state in the BTOR2 model.

Once the model is generated, BtorMC is used to look for reachable bad-states. If a bad-state is reached, a witness is generated. It contains the binary value that triggers the error at the corresponding point in execution time. This information must be extracted in order to get an actual input file for the C* program.

7.3 Parsing the witness format

The parser for Boolector’s witness format is the heart of *Validator*. It is based on the EBNF in Fig.10, whereby each line of the EBNF corresponds to exactly one function in the parser. The only difference to the EBNF, is that comments are handled globally in the parser and not as part of EBNF clauses.

The parser consists of the following functions:

```

– parse_witness()
– parse_header()
– parse_prop()
– parse_frame()
– parse_input_part()
– parse_state_part()
– parse_model()
– parse_assignment()
– parse_bv_assignment()
– parse_array_assignment()
– parser_error(expected: str)

```

In addition, the functions `get_symbol()` and `generate_output(frame)` are used to fetch the next symbol of the witness (separated by blank or line break) and to write the extracted input symbols to an external file.

`get_symbol()` is used to write the next symbol of the witness in the global variable `symbol`. The current line of the witness is split by blanks and stored in the global list `symbols`. The next `symbol` is obtained by calling `symbols.pop(0)`. If the first character in the list is a semicolon, the line buffer is deleted and the next line is read.

`generate_output(frame)` is called for each parsed frame. It is used to write non-zero input values to the output file. All zero input values of the frame are filtered out first. If more than one input-variable remains, an error is printed since only one input can be non-zero (see: 5.6, 9). The input value is split into byte-sized pieces, converted into characters and written to the output file in reverse byte order.

`parse_witness()` initializes the parsing process. The first symbol is fetched and the `parse_header()` function is called. Afterwards, frames are parsed until the `’.’` symbol finalizes the witness.

`parse_header()` is responsible for parsing the first few lines of the witness. The initial `sat` symbol is read and `parse_prop()` is called for each element in the property list.

parse_prop() uses regular expressions to parse a single element in the property list. The element is then stored in the global list **props** for later use. It should be noted that not only bad-states (b) but also justice states (j) are taken into account here.

parse_frame() is called for every frame. The global list **frame_content** is cleared, the functions for parsing state- and input part are called and the frame content is written to the output file with **generate_output(frame)**.

parse_input_part() recognizes the frame-number using regular expressions and stores it in the **frame_number** global variable. The frame's content is parsed subsequently by calling the **parse_model()** function.

parse_state_part() parses the optional memory information at a certain execution time. The frame-number is recognized again by regular expressions, but not stored. The frame's content is parsed subsequently by calling the **parse_model()** function.

parse_model() loops over the function **parse_assignment()** until all memory or input assignments of the frame have been parsed.

parse_assignment() decides for each assignment of the frame whether it is an array assignment or a bit-vector assignment. Array assignments contain memory information while bit-vectors hold input values. To distinguish between those two, regular expressions are used that focus on the brackets of an array. **parse_array_assignment()** and **parse_bv_assignment()** are called respectively.

parse_bv_assignment() stores the bit-vector representing an input value in the global list **frame_content**.

parse_array_assignment() stores a tuple of a memory address with its corresponding value in the global list **memory_constraints**. The first bit-vector of the assignment (in brackets) holds the memory address, the second one holds the value.

Both values are converted to decimal representation before being stored.

parse_error(expected: str) is called every time a parser function reads an invalid symbol. It takes the expected symbol as an argument and uses it together with the actual content of the **symbol** variable as part of the following error message:

Parser Error: "<expected>" expected but "<symbol>" found!

The result of parsing a witness, is a single string of input characters. When this string is piped to the initial C* file while executing it on Mipster, the program should encounter the predicted error.

7.4 Validating the error

In order to check whether the string generated by the parser actually causes the C* program to run into an error, the C* file needs to be executed on Mipster. Running on Mipster, the program reads the string using `read` calls to `stdin`. In the command line, the `stdin` stream is redirected to the file generated by the parser. For this reason it is important to set the file descriptor of the `read` calls to 0 (`=stdin`) in the C* files used for this. When the file descriptor is not set to 0, Selfie does not terminate since it cannot get any input data. To avoid this case, the execution command uses a timeout that terminates Selfie with an error after a certain time. The UNIX command for executing the C* file with the input string can be seen in Fig.18.

```
timeout <max execution-time> <selfie-executable path> -c <C*-file>
-m <number of MB for mipster-memory> < <file holding the input-string>
> <file storing selfie's output> ; if [ $? = 124 ];
then echo "selfie timed out" ; fi >> <file storing selfie's output>
```

Fig.18. Execution command

Selfie compiles the C* file and runs the binary on Mipster with the specified memory size. The file generated by the parser is used as `stdin`, while `stdout` is redirected to an empty file. If the timeout is triggered, the `timeout` command returns the exit code 124. In this case, the string *selfie timed out* is written to the output file in addition to Selfie's output so far.

After execution, the generated file is examined. If the string *selfie timed out* is found, an error is displayed and the program is terminated. Otherwise, the program looks for the characteristic string that represents the expected error. Selfie's error messages indicate the following bad-states (see Fig.17).

- **exit code n**: b1 (with value n)
- **division by zero**: b2, b3
- **uncaught invalid address**: b4, b5, b6, b7, b8, b9

If the message corresponding to the calculated bad-state is found, the predicted error is verified.

7.5 Program documentation

All functionality previously discussed in this chapter is implemented in a single Python script called `validator.py`. *Validator* can be run from a Linux shell without explicitly using the `python3` command. It is therefore sufficient to only enter `./validator.py`. However, Python 3 needs to be installed on the system used. *Validator*'s usage pattern is shown in Fig.19.

```
usage: validator.py [-h] [-d] [-e BAD_EXIT_CODE] [-s SELFIE_PATH]
                  [-m MODELER_PATH] [-b BTORMC_PATH] [-ts SELFIE_TIMEOUT]
                  [-tb BTORMC_TIMEOUT] [-kmax KMAX] [-mem MEMORY]
                  in_file

positional arguments:
  in_file              input C* file

optional arguments:
  -h, --help          show this help message and exit
  -d, --debug          debug mode (generated files are kept)
  -e BAD_EXIT_CODE, --exitcode BAD_EXIT_CODE
                      value for non-zero exit code bad-state
  -s SELFIE_PATH, --selfie SELFIE_PATH
                      path to selfie executable
  -m MODELER_PATH, --modeler MODELER_PATH
                      path to modeler.selfie
  -b BTORMC_PATH, --btormc BTORMC_PATH
                      path to btormc executable
  -ts SELFIE_TIMEOUT, --timeout_selfie SELFIE_TIMEOUT
                      timeout for execution of in_file on mipster
                      (example: 10s, 5m, 1h)
  -tb BTORMC_TIMEOUT, --timeout_btormc BTORMC_TIMEOUT
                      timeout for execution of btormc with the generated
                      btor2 file (example: 10s, 5m, 1h)
  -kmax KMAX          -kmax parameter for btormc
  -mem MEMORY, --memory MEMORY
                      memory [MB] for mipster
```

Fig. 19. *Validator* usage pattern

The only mandatory argument is the C* file, which should be checked for errors. However, there are several optional arguments.

The bad exit code used in Selfie's model generator is determined by an integer value given to the `-e` option. If the C* program is terminated with this exit code, the *non-zero exit code* bad-state is triggered.

The path to Selfie, Modeler and BtorMC can be set with the options `-s`, `-m` and `-b` respectively. By default, the paths `./selfie`, `./tools/modeler.selfie` and `btormc` are used. It should be noted that the path specified must contain the actual executable.

Timeout values can be set for Selfie and BtorMC using the `-ts` and `-tb` options. For Selfie, the timeout is used in error validation (see 7.4) with a default value of 10 seconds.

The BtorMC timeout is set to 9 minutes by default. With complex BTOR2 models, it can take a long time to find a bad-state and generate a witness. It may therefore be necessary to increase this time limit.

Since the timeout values are used directly as an argument for the Linux command `timeout`, the time unit used can be specified (`s`, `m`, `h`) (see `timeout` manual page).

The *kmax option* (`-kmax`) is equivalent to the `-kmax` option for BtorMC (5.5). The default value here is 10000, which should be enough for most models. However, to limit the number of instructions executed, this option can be used.

Memory size for Mipster can be set with the option `-mem`. The value is interpreted as the number of megabytes available for Mipster. The default is 2MB.

A *debug mode* can be invoked with the `-d` option. The debug mode extends *Validator*'s runtime output with the following information:

- Selfie's output while generating the BTOR2 model.
- Information about files produced in the execution process.
- List of properties contained in the witness.
- Which values were found at what timeframe?
- How are the found values divided into bytes?
- Number of timeframes parsed.
- The string representing the predicted error in Selfie's output.

More importantly, all temporary files generated by *Validator* are retained after execution when debug mode is active. The folder *temp* is located in the same directory as *Validator* and holds the following files after execution:

- `model.btor2`: The BTOR2 model generated by Selfie
- `witness.wit`: The witness generated by BtorMC
- `error_input.txt`: The input causing an error predicted in the witness
- `selfie_out.txt`: The output generated by Selfie while executing the C* file

***Validator* terminates with one of six different exit codes**

- **0: (Error successfully verified)** The program terminated without error and the predicted error state was successfully verified
- **1: (Could not verify predicted error)** The program terminated without error, but the predicted error state could not be verified
- **2: (File Not Found)** One of the external files was not found
- **3: (Parser error)** Is triggered when the parser reads an invalid symbol
- **4: (Timeout)** Execution of BtorMC or Selfie timed out
- **5: (Internal error)** This error should never occur. If it still does, it is due to a bug in the program.

8 Experiment/Examples

In the following experiments, C* (6.1) source code is given on the left hand side with the corresponding output of *Validator* (7.5) on the right. Note that this is not the total output of *Validator* but only the part regarding the parser and error validation (see 7.3, 7.4). The blue sections are generated by the debug mode, the yellow ones are the usual output.

8.1 Multiple read calls with division by zero

This first experiment demonstrates several important points when checking for source code errors.

| | |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> 1 uint64_t main() { 2 uint64_t* x; 3 uint64_t* y; 4 5 x = malloc(1); 6 read(0, x, 1); 7 8 y = malloc(1); 9 read(0, y, 1); 10 11 if (*x == '4') 12 if (*y == '2') 13 *y = *x/0; 14 15 return 0; 16 }</pre> | <pre> parsing witness... Properties: ['b2'] Value: 52 at frame 67 Byte#1 = 00110100 Value: 50 at frame 113 Byte#1 = 00110010 Parsing Witness finished Number of Frames parsed: 136 division by zero error state found! Error causing input written to ./temp/error_input.txt Input and Output Stream closed. Executing ../Arbeit/multReadDivZero.c on Mipster with calculated input Error text: "division by zero" division by zero error verified! All generated files in temp directory!</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Fig. 20. Experiment 1

In this source code two pointers are declared, both of which are reading a single byte from *stdin*. If the first pointer reads the char '4' and the second pointer reads '2', the program executes a division by zero on the value the pointer *x* points to.

First off, in the output it can be seen that the parser finds the *division by zero* error state by reading the list of properties containing the value 'b2'. Secondly, the value 52 is found in frame 67, which corresponds to the first *read* call reading the character '4'. The value of *Byte#1*, which is printed next, is not of great interest here and will be discussed in more detail in 8.2. For the second read call at frame 113, the value 50 is found which equals the ASCII character '2'.

When parsing is complete, the number of frames in the witness (136 in this case) is printed as additional debugging information. The characters '4' and '2' are written to the output file *error_input.txt* and the program is executed on Mipster. Since a *division by zero* error state was found, the string that is searched for in Selfie's output is *division by zero*.

Finally, the string is found in this example and the error is therefore verified.

8.2 Reading multiple bytes with memory constraint

This example shows how memory constraints are found and how they lead to errors that are predicted correctly, but cannot be verified by Selfie. Furthermore, reading multiple bytes is shown.

| | |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> 1 uint64_t main() { 2 uint64_t* x; 3 uint64_t* y; 4 5 x = malloc(2); 6 y = malloc(1); 7 8 read(0, x, 2); 9 10 // 12852 = "42" 11 if (*x == 12852) 12 if (*y == 2) 13 return 1; 14 15 return 0; 16 }</pre> | <pre> parsing witness... Properties: ['b1'] Value: 12852 at frame 90 Byte#2 = 00110100 Byte#1 = 00110010 Parsing Witness finished Number of Frames parsed: 124 non-zero exit code error state found! Memory constraints: Value: 2 at address 66096 Error causing input written to ./temp/error_input.txt Input and Output Stream closed. Executing ../Arbeit/readMultByteMemConst.c on Mipster with calculated input. Error text: "exit code 1" non-zero exit code error could not be verified. consider memory constraints and make sure the C* file is reading from stdin! All generated files in temp directory!</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Fig. 21. Experiment 2

In this file, again two pointers are allocated. The `read` call stores two bytes at the memory location pointed to by `x`.

Since it is not possible to compare strings in C* without implementing additional functions, the required input is represented as an integer value. The integer 12852 is equivalent to the bytes 00110010 and 00110100, which correspond to the ASCII characters '4' and '2'. This dependency between integer value, binary values and input string is displayed in the parser's output. The value 12852 is found in frame 90, which corresponds to the `read` call. Since two bytes are read, the binary string must be split in order to obtain the correct ASCII characters for the input string. How the binary string is split and which byte is written to which position is given by the values of `Byte#1` and `Byte#2`.

In addition to the input values, a memory constraint is found here. Namely the value 2 at memory address 66096. This occurs due to line 12 in the source code. The address `y` points to, must contain the value 2 in order to trigger the error `non-zero exit code`. Since no `read` call writes to this address, there is no way an input value can lead to the value 2 at this memory location. However, BtorMC still finds the error and prints information about this additional memory constraint into the witness. The error is predicted correctly, but verification via Selfie becomes a problem now. Even reading the correct input string does not trigger the error on Mipster because address 66096 does not hold the value 2. This is why the verification error message indicates that memory constraints could be the reason for failure.

8.3 Invalid memory access

The source code of this example differs from the code of the previous example (8.2) only by one additional line. This additional line, though, leads to a totally different verification outcome.

```

1  uint64_t main() {
2      uint64_t* x;
3      uint64_t* y;
4
5      x = malloc(2);
6      y = malloc(1);
7
8      // invalid memory access
9      *(x + 4294967296) = 0;
10
11     read(0, x, 2);
12
13     // 12852 = "42"
14     if (*x == 12852)
15         if (*y == 2)
16             return 1;
17
18     return 0;
19 }
```

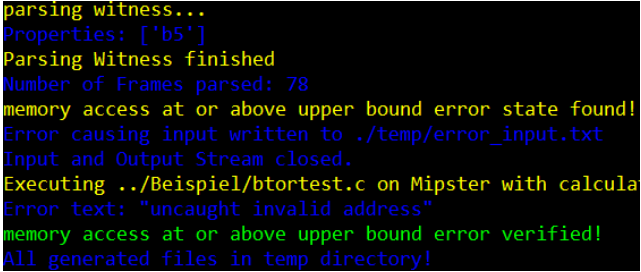


Fig. 22. Experiment 3

Line 9 holds an access to an invalid memory location. Since Selfie's address space is 4GB big, the highest possible memory address is 4294967295. Therefore, the address $x + 4294967296$ is out of bound for every non-negative x . BtorMC finds this error and displays it in the witness. The error state `b5` (`memory access at or above upper bound`) is found after executing 78 instructions. For this example, no input value is found in the witness since the value 0 is sufficient to trigger the error. And in fact, Mipster verifies the error by reading an empty input file.

Interestingly, even though the error state discussed in 8.2 is still contained in this code, it is not found by BtorMC. This is due to the nature of SMT-solving, where it is asked if a formula is satisfiable, but not in which ways it is satisfiable. Therefore BtorMC generates the witness only for the first error found by Boolector.

9 Conclusion

In this bachelor thesis, a program was introduced which is capable of finding particular errors in a given piece of C* (6.1) source code. In order to provide the necessary background information, basic ideas behind SAT-solving (2), SMT-solving (3) and symbolic execution (4) were discussed. The DPLL algorithm has been explained in more detail to enable an intuitive understanding of state-of-the-art SAT-solving procedures. Based on this information, a theoretical introduction to SMT-solving was given. To deepen the understanding of SMT-solving the specific SMT-solver Boolector (5) was presented later in the thesis. In order to get an insight into SMT based code verification, an overview of symbolic execution was given.

After discussing background information, some technologies required for the presented workflow were introduced. Boolector's BTOR2 (5.4) and witness (5.6) format were presented and examples related to experiments in 8 were given. Furthermore, the Boolector based model checking tool BtorMC (5.5) was introduced, acting as a symbolic execution engine for BTOR2 files. In addition to these technologies, which were developed at the JKU Linz, the Selfie system (6) developed at the University of Salzburg was presented. A general example of Selfie's capabilities was given before introducing the C* language, the RISC-U instruction set (6.2) and the Mipster emulator (6.3). Selfie's model generator (6.4), which is an essential part of the validation workflow, was discussed based on the source code of `modeler.c`.

In chapter 7 a software was introduced that combines all concepts and technologies discussed before. The software's purpose was explained and implementation of the required functionality was divided into three steps. First, the generation of a witness from a C* file using Selfie's model generator and BtorMC was shown (7.2). Secondly, the process of parsing a witness was presented (7.3) and finally validation of the resulting string via Mipster was demonstrated (7.4). To enable further use of the software, a short program documentation was given (7.5). At the end of the thesis, the capabilities of the program were shown in three different examples (8). These examples demonstrated both standard and exceptional cases using the program's debug mode.

In conclusion, a parser for Boolectors witness format as well as the workflow for automated input validation was successfully implemented in the software presented in this thesis. Combining Selfie and Boolector, this project can be seen as a proof of concept that demonstrates how the advances in SMT-solving can be used to achieve automated software testing. The enormous progress in SMT-solving in recent years makes this testing approach more and more interesting for all types of software developers. Many research facilities and also big companies like Microsoft put a lot of effort in the development of efficient SMT-solvers and symbolic execution engines [3]. Based on these concepts, future implementations could contribute to massive increase of code quality and decrease in software testing time, leading towards cheaper and more stable software products.

References

1. Smt competition winners. <https://smt-comp.github.io/2019/results>
2. Abyaneh, A., Bauer, S., Kirsch, C., Mayer, P., Mösl, C., Poncelet, C., Seidl, S., Sokolova, A., Widmoser, M.: Selfie: Towards minimal symbolic execution. In: Online Proc. Workshop on Modern Language Runtimes, Ecosystems, and VMs (MoreVMs) (2018)
3. Baldoni, R., Coppa, E., D’elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* **51**(3), 1–39 (2018)
4. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
5. Barwise, J.: An introduction to first-order logic. In: *Studies in Logic and the Foundations of Mathematics*, vol. 90, pp. 5–46. Elsevier (1977)
6. Biere, A.: Boolector at github. <https://github.com/boolector/boolector>
7. Biere, A.: The aiger and-inverter graph (aig) format version 20071012. FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr **69**, 4040 (2007)
8. Brummayer, R., Biere, A.: Boolector: An efficient smt solver for bit-vectors and arrays. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. pp. 174–177. Springer (2009)
9. Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. *Journal on Satisfiability, Boolean Modeling and Computation* **6**(1-3), 165–201 (2010)
10. Brummayer, R., Biere, A., Lonsing, F.: Btor: bit-precise modelling of word-level problems for model checking. In: *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*. pp. 33–38 (2008)
11. Bruttomesso, R.: RTL Verification: From SAT to SMT (BV). Ph.D. thesis, Ph. D. thesis, University of Trento (2008)
12. Chen, T., Zhang, X.s., Guo, S.z., Li, H.y., Wu, Y.: State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems* **29**(7), 1758–1773 (2013)
13. Chipounov, V., Georgescu, V., Zamfir, C., Candea, G.: Selective symbolic execution. In: *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*. No. CONF (2009)
14. Cook, S.A.: The complexity of theorem-proving procedures. In: *Proceedings of the third annual ACM symposium on Theory of computing*. pp. 151–158 (1971)
15. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. pp. 337–340. Springer (2008)
16. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. *Communications of the ACM* **54**(9), 69–77 (2011)
17. Dutertre, B., De Moura, L.: A fast linear-arithmetic solver for dpll (t). In: *International Conference on Computer Aided Verification*. pp. 81–94. Springer (2006)
18. Girle, R.A.: First-order logic and automated theorem proving (1998)
19. Husák, R., Zavoral, F.: Source code assertion verification using backward symbolic execution. In: *AIP Conference Proceedings*. vol. 2116, p. 350004. AIP Publishing LLC (2019)
20. Kirsch, C.: Selfie at github. <https://github.com/cksystemsteaching/selfie>

21. Kirsch, C.: Selfie and the basics. In: Proc. ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!). ACM (2017)
22. van Maaren, H., Biere, A., Walsh, T.: Handbook of satisfiability (2009)
23. McCarthy, J.: Towards a mathematical science of computation. In: Program Verification, pp. 35–56. Springer (1993)
24. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. Journal on Satisfiability, Boolean Modeling and Computation **9**(1), 53–58 (2014)
25. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, btormc and boolector 3.0. In: International Conference on Computer Aided Verification. pp. 587–595. Springer (2018)
26. Preiner, M., Niemetz, A., Biere, A.: Lemmas on demand for lambdas. Program Proceedings p. 28 (2013)
27. Sebastiani, R.: Lazy satisfiability modulo theories. Journal on Satisfiability, Boolean Modeling and Computation **3**(3-4), 141–224 (2007)
28. Walther, C.: Many-sorted inferences in automated theorem proving. In: Sorts and Types in Artificial Intelligence, pp. 18–48. Springer (1990)
29. Waterman, A., Lee, Y., Patterson, D., Asanovic, K., level Isa, V.I.U.: The risc-v instruction set manual. Volume I: User-Level ISA’, version **2** (2014)