Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Automatically Testing SMT Solvers

Bachelor Thesis

Olivier Becker

November 17, 2021

Advisors: Prof. Dr. Peter Müller, Alexandra Bugariu
Department of Computer Science, ETH Zürich

Abstract

Satisfiability modulo theories (SMT) solvers can determine the satisfiability of first-order formulas with respect to various theories, which enable them to reason about different data types and data structures. To be able to detect soundness issues related to the bit-vector theory and array theory in the implementation of such solvers we adopt a technique which generates formulas which are satisfiable or unsatisfiable by construction. We define two new transformations for the generation of unsatisfiable formulas which allows us to test the solvers more rigorously. Additionally, we extend the technique to be able to generate formulas which combine operations from multiple theories to test the interaction between different theories.

To adapt the technique to the bit-vector theory and array theory we implement an executable version of the SMT-LIB semantics. Using this executable semantics we can also generate formulas which combine multiple theories.

We use the extended technique to test widely used SMT solvers such as Z3 and CVC4. Our evaluation shows that the technique is able to find soundness issues in both of them.

Contents

Chapter 1

Introduction

Satisfiability modulo theories (SMT) solvers have many different applications including program verification, program synthesis, symbolic execution and test case generation. These solvers (such as Z3 [27] and CVC4 [23]) can determine the satisfiability of first-order formulas with respect to various theories, which enable them to reason about different data types and data structures. The SMT-LIB library [1] includes theories for Booleans, integers, real numbers, floating point numbers, strings, arrays and bit vectors.

For a given input formula, an SMT solver may return one of the following four results:

- **sat**: The SMT solver was able to find a model (i.e., a value for each free variable and an interpretation for each uninterpreted function), such that the input formula evaluates to true. The formula is said to be satisfiable.
- **unsat**: The SMT solver has determined that the input formula is unsatisfiable, no combination of values for the free variables or no interpretation for the uninterpreted functions for which the input formula evaluates to true exists. In addition, the SMT solver usually returns the set of clauses which lead to a contradiction, called the unsat core.
- **unknown**: SMT solvers support undecidable theories. As a result, they can sometimes not decide whether the input formula is sat or unsat. In such cases, they return unknown.
- **timeout**: Sometimes the SMT solver cannot determine the satisfiability of the input formula in the allocated timeframe and will time out.

An SMT solver can be affected by different problems including:

- **Unsoundness**: If an SMT solver returns sat for an unsat formula or unsat for a sat formula then it returns an incorrect result and is unsound.

In addition, an SMT solver is also unsound if it correctly returns sat for a sat formula but it returns an invalid model. Analogously, the solver is unsound if it returns an incorrect core for an unsat formula.

- **Incompleteness**: An SMT solver is incomplete if it returns unknown for a formula from a decidable theory which is known to be sat or unsat.
- **Timeouts**: If an SMT solver times out frequently, it could be an indication of underlying performance issues.

1.1 Motivation

If an SMT solver is affected by one of the aforementioned problems, its usefulness will be severely impacted. Furthermore, all the applications which rely on this SMT solver will also be impacted. Therefore, it is important to ensure that the solvers are sound, complete and efficient. Yet as SMT solvers are very complex programs, it is not only difficult to implement them but also to verify them. Thus it comes at no surprise that bugs are regularly found even in widely used SMT solvers.

For example, the following formula exposed a soundness bug in the Z3 solver from June 2020 [2]:

$$\text{store(const(Bool, true), false, false) = const(Bool, false)}$$

On the right-hand side a constant array is created. This array takes Boolean values as indices and has the constant value of false. Therefore it represents the following array:

$$[true \rightarrow false, false \rightarrow false] \ (1)$$

On the left-hand side the term const(Bool, true) analogously creates the following array:

$$[true \rightarrow true, false \rightarrow true] \ (2)$$

But then the array value at index false is replaced by false, returning the following array:

$$[true \rightarrow true, false \rightarrow false] \ (3)$$

Trivially we see that the array on the right-hand side (1) is not equal to the array on the left-hand side (3) and therefore the SMT solver should return unsat. Yet, the Z3 solver unsoundly returned sat.

1.2 This work

Program verification tries to formally prove the correctness of a system while testing seeks to find errors in the implementation of a system. This thesis will focus on *testing SMT solvers*. The goal is to automatically generate formulas which are satisfiable or unsatisfiable by construction. These formulas can then be used as test cases and the known ground truth can be used as the test oracle.

Bugariu and Müller introduced this idea in [26]. In their paper, they describe methods to generate satisfiable and unsatisfiable formulas with respect to the SMT-LIB string theory [3]. Furthermore, they also introduce satisfiability-preserving transformations to increase the complexity of the generated formulas.

This thesis adapts the approach proposed by Bugariu and Müller for the SMT-LIB bitvector and array theories ([4], [5]). Furthermore, we extend the approach to be able to generate unsatisfiable formulas with constant values such as the example given in Section 1.1. Finally, we adapt the proposed methods to be able to combine multiple theories. This is important as many reported issues with SMT solvers occur due to unexpected behaviour when combining theories. Our main focus in this thesis is to automatically identify soundness bugs in SMT solvers but completeness and efficiency issues might be still exposed as a ”by-product”.

1.3 Contributions

This thesis makes the following contributions:

- We adapted the methods and transformations described in [26] to be able to generate satisfiable and unsatisfiable formulas for bit-vectors and arrays.
- We adapted the transformations described in [26] to combine multiple theories and generate more complex formulas.
- We introduced two new transformations to be able to generate unsatisfiable formulas with constant values.
- We used the generated formulas to automatically test Z3 [27] and CVC4 [23]. We detected soundness issues in both solvers, which were confirmed and fixed by the developers.

1.4 Outline

The rest of this thesis is organized as follows: Chapter 2 gives an overview of the methods and transformations described by Bugariu and Müller in [26]

as well as the methods introduced by this thesis to generate unsatisfiable formulas with constant values. Chapter 3 gives in-depth explanations for the methods and transformations presented in Chapter 2. In Chapter 4 we discuss the implementation details. Chapter 5 presents and discusses our experimental results. In Chapter 6 we discuss related work. Chapter 7 draws a conclusion and discusses future work.

Chapter 2

Overview of the technique

On a high-level, the technique used in this thesis, as first described in [26], consists of two parts:

- first, generate simple formulas,
- then, use various satisfiability preserving transformation to increase the complexity of the generated formulas.

We present and illustrate this technique in this chapter using the operations from the fixed-sized bit-vector theory, which are summarized in Table 1. Additionally, to be able to carry out the different steps of the technique we use our implementation of the reference semantics of the individual operations as they are defined in the SMT-LIB bit-vector theory [4].

Section 2.1 presents the methods used to generated satisfiable and unsatisfiable formulas described by Bugariu and Müller in [26]. Section 2.2 presents the satisfiability preserving transformations described in [26]. Furthermore, we defined new transformations which are presented in Section 2.3. Chapter 3 discusses the technical details of the technique with respect to the individual theories that we use in this thesis.

2.1 Generating formulas

[26] describes two different methods to generate formulas, one for satisfiable and one for unsatisfiable formulas. They are presented in Section 2.1.1 and in Section 2.1.2 respectively.

2.1.1 Satisfiable formulas

The generated satisfiable formulas have the following form:

$$left - hand\ side = right - hand\ side$$

Table 1: Bit-vector operations, grouped by their return type

Return Type	Operations
Bit-vector	nat2bv(m, a), concat(s, t), extract(i, j, s), bvnot(s), bvand(s, t), bvor(s, t), bvneg(s), bvadd(s, t), bvmul(s, t), bvudiv(s, t), bvurem(s, t), bvshl(s, t), bvshr(s, t)
Integer	bv2nat(s)
Boolean	bvult(s, t)

s, t: Bit-vector of length m; *a, i, j*: Int; We use the nat2bv(m, a) notation, where the length of the bitvector m is passed as the first argument to the nat2bv function, instead of the SMT-LIB nat2bv[m](a) notation to avoid confusion with the array notation, which uses square brackets to denote indexing.

Therefore they are satisfiable if there exists a model for which both sides evaluate to the same result.

The initial satisfiable formulas involve a single operation from Table 1. By leaving the arguments of the given operation as well as its result unconstrained, we ensure that the formula is satisfiable. This is due to the fact that all operations from Table 1 are total functions. The generated formulas are as simple as possible yet Bugariu and Müller show in [26] that even such simple formulas can reveal bugs. Example 1 illustrates such a formula.

Example 1: A simple, satisfiable formula

bvadd(s, t) = res

where s, t and res are unconstrained bit-vectors of the same size.

2.1.2 Unsatisfiable formulas

To generate an unsatisfiable formula, we cannot use the same strategy as for satisfiable formulas. This is due to the fact that, to prove that a formula is satisfiable, one needs to find one combination of values for which the formula evaluates to true. For unsatisfiable formulas on the other hand, one needs to be sure that every combination evaluates to false. This is not practical as many SMT-LIB theories support a very large or even infinite number of values. To solve this issue, [26] proposes to generate formulas of

Table 2: Equivalent formulas for bit-vector operations.

A	s: (BitVec m), t: (BitVec n), res: (BitVec m+n) :: concat(s, t) = res
B	nat2bv(m+n, bv2nat(s)*2^m + bv2nat(t)) = res
A	s: (BitVec m), i: Int $\in \{0, ..., m\text{-}1\}$, j: Int $\in \{0, ..., i\}$, res: (BitVec i-j+1) :: extract(i, j, s) = res
B	nat2bv(i-j+1, bv2nat(bvlshr(bvshl(s, nat2bv(m, m-1-i)), nat2bv(m, m - (i-j+1))))) = res
A	s, res: (BitVec m) :: bvnot(s) = res
B	nat2bv(m, 2^m-1-bv2nat(s)) = res
A	s, res: (BitVec m) :: bvneg(s) = res
B	nat2bv(m, 2^m-bv2nat(s)) = res
A	s, t, res: (BitVec m) :: bvadd(s, t) = res
B	nat2bv(m, bv2nat(s) + bv2nat(t)) = res
A	s, t, res: (BitVec m) :: bvmul(s, t) = res
B	nat2bv(m, bv2nat(s) * bv2nat(t)) = res
A	s, t, res: (BitVec m) :: bvudiv(s, t) = res
B	(nat2bv(m, bv2nat(s) div bv2nat(t)) = res **if** bv2nat(t) $>$ 0) $\wedge$ (res = nat2bv(m, 2^m-1) **otherwise**)
A	s, t, res: (BitVec m) :: bvurem(s, t) = res
B	(nat2bv(m, bv2nat(s) mod bv2nat(t)) = res **if** bv2nat(t) $>$ 0) $\wedge$ (res = s **otherwise**)
A	s, t, res: (BitVec m) :: bvshl(s, t) = res
B	(extract(m-1, 0, s) = res **if** bv2nat(t) = 0) $\vee$ (concat(extract(m-2, 0, s), (BitVec 1) x#0) = res **if** (m $\geq$ 2 $\wedge$ bv2nat(t) = 1)) $\vee$ ... $\vee$ (concat(extract(0, 0, s), (BitVec m-1) x#0) = res **if** bv2nat(t) = m-1) $\vee$ ((BitVec m) x#0 = res **if** bv2nat(t) $\geq$ m)
A	s, t, res: bvlshr(s, t) = res
B	(extract(m-1, 0, s) = res **if** bv2nat(t) = 0) $\vee$ (concat((BitVec 1) x#0, extract(m-1, 1, s)) = res **if** (m $\geq$ 2 $\wedge$ bv2nat(t) = 1)) $\vee$ ... $\vee$ (concat((BitVec m-1) x#0, extract(m-1, m-1, s)) = res **if** bv2nat(t) = m-1) $\vee$ ((BitVec m) x#0 = res **if** bv2nat(t) $\geq$ m)
A	s, t, res: bvult(s, t) = res
B	bv2nat(s) $<$ bv2nat(t) = res
A	s, t, res: bvand(s,t) = res
B	$\exists$r: (BitVec 1) :: (r = (BitVec 1) x#1 **if** extract(m-1, m-1, s) = (BitVec 1) x#1 $\wedge$ extract(m-1, m-1, t) = (BitVec 1) x#1) $\wedge$ (r = (BitVec 1) x#0 **otherwise**) $\wedge$ (concat(r, bvand(extract(m-1, 0, s), extract(m-1, 0, t))) = res
A	s, t, res: bvor(s,t) = res
B	$\exists$r: (BitVec 1) :: (r = (BitVec 1) x#1 **if** extract(m-1, m-1, s) = (BitVec 1) x#1 $\vee$ extract(m-1, m-1, t) = (BitVec 1) x#1) $\wedge$ (r = (BitVec 1) x#0 **otherwise**) $\wedge$ (concat(r, bvand(extract(m-1, 0, s), extract(m-1, 0, t))) = res

A: bit-vector formula; **B**: equivalent formula; '(BitVec m)' denotes a bitvector of length m, where m is a strictly positive integer, and 'div' denotes the integer division. All equivalences A $\Leftrightarrow$ B are implicitly universally quantified over all free variables.

the following form:

$$\neg A \wedge B$$

where A and B are equivalent formulas, making ¬A ∧ B trivially unsatisfiable.

To generate our initial unsatisfiable formulas, we first choose a formula A from the generated formulas as described in Section 2.1.1. To obtain an equivalent formula, we use equivalences between the different operations, described in [33] and [4]. The equivalences are summarized in Table 2. Now that we have formula A and its equivalent formula B, we finally generate ¬A ∧ B. This method is shown in Example 2.

Example 2: Generating an unsatisfiable formula

First, we choose as an initial formula A:

bvadd(s, t) = res

Then we choose the equivalent formula B from Table 2:

nat2bv(bv2nat(s) + bv2nat(t)) = res

Where bv2nat returns an integer by interpreting a bit-vector as a binary representation of a number and nat2bv creates a bit-vector from an integer. Finally, we can create the following unsatisfiable formula:

¬(bvadd(s, t) = res) ∧ (nat2bv(bv2nat(s) + bv2nat(t)) = res)

2.2 Satisfiablity preserving transformations

[26] describes multiple satisfiability preserving transformations, two for satisfiable formulas, which are presented in Section 2.2.1, and four for unsatisfiable formulas, which are presented in Section 2.2.2.

2.2.1 Transformations for satisfiable formulas

Constant assignment transformation. To test the basic implementation of a given operation of the SMT solver, [26] proposes to assign concrete values to the arguments. These values are chosen from a list containing interesting values and corner cases. This list, adapted for the bit-vector theory, is shown

Table 3: Initial values for the bit-vector theory.

Type	Values
Bit-vector (of size m)	x#0, x#1, x#2, x#2^m-1
Integer	0, 1, 2, 2^m-1
Boolean	true, false

The bit-vectors in this table are represented using the hexadecimal representation. In this representation, the bit-vectors are interpreted as binary numbers which are then represented in their hexadecimal form.

in Table 3. Then using the reference semantics, we evaluate the operation with these fixed values. We use the result of the evaluation as the right-hand side in the formula that we generate. Thus, if the SMT solver implements the operation correctly it will return satisfiable for the given formula. Example 3 showcases this method.

Example 3: Creating a satisfiable formula with constant values

To transform bvadd(s, t) = res into a formula with constant values, we first select some values for s and t from Table 3, such as x#0 for both. x#0 denotes the hexadecimal notation of bit-vector of length 4 where all bits are set to 0. We obtain the following operation:

bvadd(x#0, x#0)

Using our executable semantics we obtain that bvadd(x#0, x#0) evaluates to x#0. Therefore, we generate the following formula, which is satisfiable by construction:

bvadd(x#0, x#0) = x#0

[26] proposes to create further satisfiable formulas by leaving certain variables unconstrained instead of assigning them the calculated value. In this case, the constants that were calculated for the variables, that are left unconstrained, form a model for the formula. Furthermore, [26] takes the power set of the constants to be able to generate each possible combination. This step is illustrated in Example 4.

Example 4: Creating all formulas from an operation with given constants

From Example 3, we know that (s = x#0, t = x#0, res = x#0) is a model for bvadd(s, t) = res. The power set of the variable set is: {{}, {s}, {t}, {res}, {s, t}, {s, res}, {t, res}, {s, t, res}}. For each set of the power set, we leave the variables contained in the set unconstrained. In this case, we generate the following formulas, which are satisfiable by construction:

bvadd(x#0, x#0) = x#0
bvadd(s, x#0) = x#0
bvadd(x#0, t) = x#0
bvadd(x#0, x#0) = res
bvadd(s, t) = x#0
bvadd(s, x#0) = res
bvadd(x#0, t) = res
bvadd(s, t) = res

Term synthesis transformation. To test the interactions between different operations and increase the the complexity of the created formulas, [26] proposes a transformation which they named term synthesis. To perform term synthesis, we first evaluate each operation from Table 1 with each possible combination of values from Table 3 to create a set of possible values with operations that generate them. Then for a specific operation with fixed values we perform the following steps:

1. We replace each constant variable of the operation with another term that evaluates to the same constant from our result set.
2. We evaluate the new operation with our executable semantics to obtain its result.
3. In the formula, we replace the result by an term from our result set that evaluates to the same value.
4. We now replace each constant with an unconstrained variable to obtain our final formula.

The resulting formula is satisfiable by construction and the constants that we replaced in the last step from a model. Example 5 demonstrates this transformation.

Example 5: Performing term synthesis

One of the possible operation obtained by combining the operations from Table 1 with the values from Table 3 is:

bvadd(x#0, x#0)

From Example 3, we know that x#0 can be created by bvadd(x#0, x#0), therefore bvadd(x#0, x#0) is part of the set of terms that can replace x#0. Now replacing the constants, we obtain:

bvadd(bvadd(x#0, x#0), bvadd(x#0, x#0))

Using our executable semantics we see that this operation evaluates to x#0. We replace the result and obtain the following formula:

bvadd(bvadd(x#0, x#0), bvadd(x#0, x#0)) = bvadd(x#0, x#0)

As a final step, we replace x#0 by a new unconstrained variable tmp_bv and obtain the following formula, which is satisfiable by construction:

bvadd(bvadd(tmp_bv, tmp_bv), bvadd(tmp_bv, tmp_bv)) = bvadd(tmp_bv, tmp_bv)

2.2.2 Transformations for unsatisfiable formulas

Increasing the unsat core. To increase the complexity, Bugariu and Müller propose in [26] to increase the unsat core of the formula as an SMT solver then has to reason about more conditions to determine that a formula is unsatisfiable. The current unsat core of the formula ¬A ∧ B contains A and B. Both A and B contain at least one shared unconstrained variable x as they are equivalent formulas. Therefore, one can rewrite the formula as: ¬A(x) ∧ B(x). To increase the unsat core, one introduces a new variable x_{fresh} and substitutes all occurrences of x in B by x_{fresh} so that one obtains the following formula:

$$\neg A(x) \wedge B(x_{fresh}/x)$$

Further, one introduces a new clause C(x, x_{fresh}) which implies that x = x_{fresh}. A list of equalities, which can be used to construct such a clause, are listed in Table 4. As a final step, one conjoins the formula in which one substituted x

by x_{fresh} and C(x, x_{fresh}) to obtain:

$$\neg A(x) \wedge B(x/x_{fresh}) \wedge C(x, x_{fresh})$$

which is an unsatisfiable formula whose unsat core now not only contains A and B but also C(x, x_{fresh}) [26]. This transformation is illustrated in Example 6.

Example 6: Increasing the complexity of an unsatisfiable formula

From Example 2, we know that:

$$\neg(\text{bvadd(s, t)} = \text{res}) \wedge (\text{nat2bv(bv2nat(s)} + \text{bv2nat(t))} = \text{res})$$

is an unsatisfiable formula of the form $\neg A \wedge B$. Furthermore, we see that a is an unconstrained variable and occurs in both A and B. Therefore, we introduce s_{fresh} and replace all occurrences of s in B by s_{fresh}, obtaining:

$$\neg(\text{bvadd(s, t)} = \text{res}) \wedge (\text{nat2bv(bv2nat}(s_{fresh}) + \text{bv2nat(t))} = \text{res})$$

In a final step, we conjoin a new clause C(s, s_{fresh}), which implies s = s_{fresh}, to generate a formula with a larger unsat core:

$$\neg(\text{bvadd(s, t)} = \text{res}) \wedge (\text{nat2bv(bv2nat}(s_{fresh}) + \text{bv2nat(t))} = \text{res}) \wedge \text{C(s, } t_{fresh})$$

Variable replacement transformation. To further increase the complexity as well as the number of conditions over which the solver needs to reason [26] proposes to use equalities, which are derived from the used theory, to transform C(x, x_{fresh}) into a more complex term. For a variable of a given type, one selects an appropriate equality from the corresponding list of equalities. Then, one obtains the new expression for C(x, x_{fresh}) by replacing the unconstrained variable that occurs on both sides of the equality on the left-hand side by x and on the right-hand side by x_{fresh}. As a final step, one replaces any other unconstrained variables that occur in the equality by fresh variables [26]. The equalities for variables of type bit-vector are provided in Table 4, NC1 - NC12. Example 7 shows this transformation.

Example 7: Variable replacement

Suppose we have an unsatisfiable formula of the following form:

$$\neg \text{ A(x)} \wedge \text{B(x/}x_{fresh}\text{)} \wedge \text{C(x, } x_{fresh}\text{)}$$

where x and x_{fresh} are bit-vectors of the same size. From Table 4 we choose equality NC2:

$$\text{bvnot(bvnot(s))} = \text{s}$$

where s is a bit-vector of the same size as x and x_{fresh}. The bvnot(s) operation flips the values of each bit of s. We now replace s on the left-hand side by x and on the right-hand side by x_{fresh}. We obtain the new expression for C(x, x_{fresh}):

$$\text{bvnot(bvnot(x))} = x_{fresh}$$

With which we can construct the following unsatisfiable formula:

$$\neg \text{ A(x)} \wedge \text{B(x/}x_{fresh}\text{)} \wedge \text{(bvnot(bvnot(x))} = x_{fresh}\text{)}$$

Constant replacement transformation. [26] also describes that one can increase the unsat core of the formula ¬ A ∧ B(c), where c is a constant that occurs in B, by introducing a new variable z_{fresh} and substituting c by z_{fresh} in B. Then one conjoins the term C(c, z_{fresh}) that is created from equalities, analogously to the variable replacement transformation, to the formula. The equalities for bit-vector constants are provided in Table 4, C1 - C15. This transformation is shown in Example 8.

Redundancy introduction transformation [26] proposes a transformation that does not increase the unsat core, but introduces redundancy. Bugariu and Müller describe redundancy as ”additional variables and terms that may obfuscate the proof of unsatisfiability”. This transformation applies the variable replacement transformation to a variable that is unconstrained in B but not in A.

Table 4: Equalities between bit-vector operations and non-constant bit-vectors (NC1 - NC12), constant bit-vectors (C1 - C14) and constant Booleans (C15).

ID	Equality
NC1	s: (BitVec m) :: extract(m-1, 0, s) = s
NC2	s: (BitVec m) :: bvnot(bvnot(s)) = s
NC3	s, t: (BitVec m) :: bvadd(s, t) = s **if** t = 0^m
NC4	s, t: (BitVec m) :: bvadd(s, t) = t **if** s = 0^m
NC5	s, t: (BitVec m) :: bvmul(s, t) = s **if** t = 1^m
NC6	s, t: (BitVec m) :: bvmul(s, t) = t **if** s = 1^m
NC7	s, t: (BitVec m) :: bvudiv(s, t) = s **if** t = 1^m
NC8	s, t: (BitVec m) :: bvurem(s, t) = s **if** t = 0^m
NC9	s, t: (BitVec m) :: bvshl(s, t) = s **if** t = 0^m
NC10	s, t: (BitVec m) :: bvlshr(s, t) = s **if** t = 0^m
NC11	s, t: (BitVec m) :: bvand(s, s) = s
NC12	s, t: (BitVec m) :: bvor(s, s) = s
C1	bvadd(0^m, 0^m) = 0^m
C2	bvadd(1^m, 0^m) = 1^m
C3	bvadd(0^m, 1^m) = 1^m
C4	bvmul(0^m, 0^m) = 0^m
C5	s: (BitVec m) :: bvmul(s, 0^m) = 0^m
C6	s: (BitVec m) :: bvmul(0^m, s) = 0^m
C7	bvshl(0^m, 0^m) = 0^m
C8	bvlshr(0^m, 0^m) = 0^m
C9	bvand(0^m, 0^m) = 0^m
C10	s: (BitVec m) :: bvand(s, 0^m) = 0^m
C11	s: (BitVec m) :: bvand(0^m, s) = 0^m
C12	bvor(0^m, 0^m) = 0^m
C13	bvor(1^m, 0^m) = 1^m
C14	bvor(0^m, 1^m) = 1^m
C15	s: (BitVec m) :: bvult(s, s) = false

We use 0^m as an abbreviation for (BitVec m) x#0 and 1^m for (BitVec m) x#1 analogously. m is a strictly positive integer.

Example 8: Constant replacement

Suppose we have an unsatisfiable formula of the following form:

$$\neg \text{ A} \wedge \text{B(x\#0)}$$

where x#0 is a constant. From Table 4 we choose equality C1:

$$\text{bvadd(x\#0, x\#0)} = \text{x\#0}$$

We now replace x#0 the right-hand side by z_{fresh} and obtain:

$$\text{bvadd(x\#0, x\#0)} = z_{fresh}$$

Which we use as new clause C(x, z_{fresh}):

$$\text{C(x\#0, } z_{fresh}\text{)} := \text{bvadd(x\#0, x\#0)} = z_{fresh}$$

Finally, we substitute x#0 by z_{fresh} in B and conjoin C(x#0, z_{fresh}) to obtain the new formula:

$$\neg \text{ A} \wedge \text{B}(z_{fresh}/\text{x\#0}) \wedge \text{C(x\#0, } z_{fresh}\text{)}$$

2.3 Additional transformations

Using the constant assignment transformation described in Section 2.2.1, we can test the basic implementation of an operation by comparing the operation with fixed values to the result returned by our executable semantics. However, we can also test the basic implementation by comparing an operation with fixed values to an unequal result. The resulting formulas will be unsatisfiable by construction, yet multiple bugs, such as the one from Section 1.1, have been reported where an SMT solver returned satisfiable for such formulas. Furthermore, these formulas do not have the form $\neg$A$\wedge$B, thus they cannot be generated by the unsatisfiability preserving transformations discussed so far. We defined two new transformations which are presented in Section 2.3.1 and Section 2.3.2.

2.3.1 Unsatisfiable formulas with constant values

Analogously to the constant assignment transformation, we start with an initially satisfiable formulas. We then assign concrete values from our list of

values, shown in Table 3, to each variable of the given operation. Using our executable semantics we can determine the result of the operation with the fixed values. Finally, we create an unsatisfiable formula by assigning a value ,from our list of values, to the result of the operation that is unequal to the actual result of operation. Example 9 presents this transformation.

Example 9: Creating an unsatisfiable formula with constant values

From Example 3, we know that bvadd(x#0, x#0) evaluates to x#0. We simply choose a value that is not equivalent to x#0, from our list of values from Table 3, such as x#1. x#1 denotes a bit-vector of length 4, where the first bit is set to 1 and all others to 0. Assigning x#1 to the right-hand side, we obtain the following unsatisfiable formula:

$$\text{bvadd}(x\#0, x\#0) = x\#1$$

2.3.2 Unsatisfiable formulas with constant values and more complex results

We can further extend our method to generate unsatisfiable formulas with constant values by borrowing parts of the term synthesis transformation, described in Section 2.2.1 and [26]. Similar to the term synthesis transformation, we first evaluate each operation from Table 1 with each possible combination of values from Table 3 and create a set of possible result values with operations that generate them. Now we generate our unsatisfiable formulas analogously to the transformation described in Section 2.3.1. The only difference is that instead of using the values from Table 3 as possible unequal results, we use the values from our set of possible results. Finally, we replace the value we use as result with the operation that generated it. Example 10 illustrates this transformation.

Example 10: Creating an unsatisfiable formula with constant values and a more complex result

Using our executable semantics, the bvadd() operation and the values x#0 and x#1 from Table 3, we can determine that:

bvadd(x#0, x#1) = x#1

We add x#1 to our set of possible results and keep track that it can be created by bvadd(x#0, x#1). One of the possible operation/value combinations which is used to generate our starting operations is:

bvadd(x#0, x#0)

Using our executable semantics we determine that bvadd(x#0, x#0) evaluates to x#0. Therefore, we want to assign a value from our set of values that is not equal to x#0 such as x#1 to the right-hand side of our formula. As a last step, we replace x#1 by the operation that generated it and obtain the following unsatisfiable formula:

bvadd(x#0, x#0) = bvadd(x#0, x#1)

Chapter 3

Technical details

In this chapter, we discuss the different theories for which we adapted the technique presented in Chapter 2. We explain the different values and equalities that are needed for the technique as well as the changes that are needed to use the technique with a given theory. The bit-vector theory is discussed in Section 3.1, the array theory in Section 3.2 and combining multiple theories in Section 3.3. Further, we also apply our new transformations, defined in Section 2.3, to the string theory, which is discussed in Section 3.7.

3.1 Bit-vector theory

The SMT-LIB theory for fixed-sized bit-vectors is defined in [4]. Bit-vectors can have any positive size.

3.1.1 Representing bit-vector values

In the SMT-LIB standard [1] bit-vector values are represented as follows: (_ bvX m). X represents the value of the bit-vector in decimal form and m designates the size. As an example, (_ bv0 4) represents the bit-vector 0000. Z3 and CVC4 use different notations to represent the values for bit-vectors in their produced models. Z3 either represents them using the hexadecimal representation, designated by x#(hex_value) or the binary representation, designated by b#(binary_value). The representation of (_ bv0 4) would be x#0 and b#0000 respectively. Additionally to the hexadecimal and binary representation, CVC4 also uses the SMT-LIB representation in its models. In this thesis, we chose to work with the hexadecimal representation as it is the most common representation used by Z3 and our executable semantics is implemented using the Z3 Java API [6].

3.1.2 Operations

The operations defined in the bit-vector theory are listed in Table 1. Most operations that take multiple bit-vector arguments only accept bit-vectors of the same size. The notable exception is the concat operation which accepts bit-vectors of varying sizes. The concat and extract operation are also the only two operations that can, and will in the case of concat, return bit-vectors of a different size than the bit-vectors that were passed as arguments.

The operations nat2bv and bv2nat. In [4], nat2bv and bv2nat are not defined as operations but as semantic abbreviations used in the definitions of other operations. We included them in Table 1 as both Z3 and CVC4 support interpreted versions of nat2bv and bv2nat [7]. Z3 calls these operations int2bv and bv2int while CVC4 names them int2bv and bv2nat. An important difference between the operation int2bv, implemented by Z3 and CVC4, and the definition of nat2bv in [4] is that int2bv is a total function defined on all integers, whereas nat2bv is only defined on non-negative integers. As a result, for the formula int2bv(tmp_int) = tmp_bv, where tmp_int is an unconstrained integer and tmp_bv an unconstrained bit-vector, Z3 and CVC4 might return a negative integer for tmp_int in their produced model. Adding the clause tmp_int ≥ 0 to the formula often resulted in Z3 (version 4.7.1) returning very large integers. To comply with the SMT-LIB definition of nat2bv and avoid very large integers, we chose to omit int2bv from generating terms for the term synthesis transformation.

3.1.3 Initial values

The set of initial values used in the technique with the bit-vector theory are provided in Table 3. For our evaluation, we fixed the bit-vector length to 4, thus the values for bit-vectors are x#0, x#1, x#2 and x#15. We chose these values as we want to represent all corner cases that are possible. x#0 is the smallest value possible for a bit-vector of size 4, as well as the neutral element for the bvor, bvadd, bvshl and bvlshr operations. x#1 is the neutral element for the bvmul, bvudiv and bvurem operations. x#2 represents a general value for a bit-vector of size 4. x#15 is the largest value possible for a bit-vector of size 4 and the neutral element for the bvand operation.

The integers are used as arguments for nat2bv and results for bv2nat. Therefore, we chose the integers that correspond to the bit-vector values that we chose. Additionally, we have a special set of integers that is used only for the arguments of the extract operation. In our case we used $\{0, 1, 3\}$. 0 is again the smallest possible value that can be passed to the extract operation and 3 the largest as we set the bit-vector size to 4. 1 is again used as a general value.

Table 5: Array operations, grouped by their return type.

Return Type	Operations
Type Y	select(array(X,Y), i)
Array(X, Y)	store(array(X,Y), i, e)

X: index type; *Y*: element type; *i*: type X; *e*: type Y;
array(X,Y): array with index type X and element type Y;

3.1.4 Adapting the technique

Each method for generating satisfiable and unsatisfiable formulas, described in Section 2 can be used with the values from Table 3 and the operations from Table 1 without modification. To generate the unsatisfiable formulas, we use the equivalent formulas from Table 2. To perfrom variable and constant replacement, as described in Section 2.2.2, we use the equalities defined in Table 4.

As the concat and extract operations can generate bit-vectors of different sizes, we need to slightly adapt the satisfiability preserving transformations. Each operation, that in a first step generates values by exhaustively combining the operations from Table 1 with the values from Table 3, needs to check the size of the bit-vector variables before replacing them. If a bit-vector variable gets replaced by a term with a different bit-vector size then the formula does not type check and is therefore invalid.

3.2 Array theory

The SMT-LIB theory for arrays is defined in [5]. Contrary to arrays in many programming languages, the SMT-LIB arrays can not only be indexed by integer but by any type. In this thesis, we use the following types: strings, boolean, integers and bit-vectors. We can create arrays using any type as index type and any type as element type. As a result, we have 16 different types of arrays.

3.2.1 Representing array values

As Z3 and CVC4 can return values for arrays in their model, we have to consider how an array is represented as a value, which is non-trivial. Both SMT solvers support the as-const constructor to create arrays with constant values. As an example, (as const (Array Int Int) 0) creates an array, which is indexed by integers and whose elements are also integers, with the value 0 initialized at each index. Z3 and CVC4 use this smt2 expression to represent array values in their models. To represent more complex arrays the SMT

Table 6: Initial values for the array theory.

Type	Values
Bit-vector (of size m)	x#0, x#1
Integer	-1, 0, 1
Boolean	true, false
String	””, ”a”, ”-1”

solvers start from a constant array as described above and use nested store operations, shown in Table 5, to store the desired values to their indices. Z3 also rarely uses lambda functions to represent the value of an array, which is further discussed in Section 3.2.4. In this thesis, we use the same representation to describe array values as it allows to even describe infinite arrays, such as arrays indexed by integers or strings.

3.2.2 Operations

The array operations are summarized in Table 5. Although there are only two operations defined in the SMT-LIB array theory [5], we consider each of the 16 different array types. As a result, more combinations to create formulas are possible than one might think at a first glance.

Example 11: Creating a satisfiable array formula with constant values

To transform select(array(Int, Int), i) = res into a formula with constant values, we first generate an initial value for an array of type array(Int, Int) by selecting a constant value from Table 6 such as 1. Thus, we generate the constant array (as const (Array Int Int) 1). Now we select a value for i, which is of type integer, from Table 6, such as -1. We obtain the following operation:

select((as const (Array Int Int) 1), -1)

Using our executable semantics we see that select((as const (Array Int Int) 1), -1) evaluates to 1. Therefore, we generate the following formula, which is satisfiable by construction:

select((as const (Array Int Int) 1), -1) = 1

Table 7: Common lambda functions and the corresponding smt2 expression for arrays.

lambda function	(lambda ((tmp_bool Bool)) tmp_bool)
smt2 expression	store(((as const (Array Bool Bool)) true), false, false)
lambda function	(lambda ((tmp_bool Bool)) (not tmp_bool))
smt2 expression	store(((as const (Array Bool Bool)) true), true, false)
lambda function	(lambda ((tmp_bv (_ BitVec m))) (= some_bv tmp_bv))
smt2 expression	store(((as const (Array (_ BitVec m) Bool)) false), some_bv, true)
lambda function	(lambda ((tmp_str String)) (= some_str tmp_str))
smt2 expression	store(((as const (Array String Bool)) false), some_str, true)

tmp_bool: Boolean; *tmp_bv, some_bv* : Bit-vector of length m; *tmp_str, some_str*: String; *m*: strictly positive Integer;

3.2.3 Initial values

The values for bit-vectors, integers and Boolean are taken from Table 3. We include -1 as a value for integers to have a negative index, which is allowed in the SMT-LIB array theory. The values for strings are taken from [26] and represent corner cases in the string theory. Unfortunately, due to the exhaustive nature of the technique and memory limitations (further explained in Section 5.4), we need to reduce the number of values and therefore consider only the values listed in Table 6. We use these values to generate our initial array values by creating each constant array that can be generated by using the values from Table 6 as constant elements.

3.2.4 Lambda functions

Additionally to the representation presented in Section 3.2.1, Z3 uses lambda functions to represent arrays. The returned lambda functions map an index value to the same value that is stored in the array at the given index. Our executable semantics does not support lambda functions as values for arrays. Therefore, we provide a mapping between most common lambda functions, returned in our test cases, and the corresponding array represented by an smt2 expression in Table 7.

3.2.5 Adapting the technique

As for the bit-vector theory, we can use the method for generating satisfiable formulas as well as the transformations for satisfiable formulas without major changes for the array theory. Generating a satisfiable array formula is illustrated in Example 11. Contrary to the bit-vector theory, the method for generating unsatisfiable formulas of the form $\neg A \wedge B$ cannot be applied to

the array theory. This is due to the fact that we cannot represent the select operation as a function of the store operation and vice versa. As a result, we do not have equivalent formulas for the array operations. Further, this means that no transformations from Section 2.2.2 is applicable to the array theory. The only transformations that can be used to generate unsatisfiable formulas using the array theory are the transformations described in Section 2.3 as they do not rely on formulas of the form $\neg A \wedge B$. Although we are limited only to two transformations, we are able to detect soundness issues as shown in Example 12.

Example 12: Creating an unsatisfiable arrray formula with constant values

Using our executable semantics, the store() operation, the constant arrays, generated with constant values from Table 6, (as const (Array Bool Bool) true) and (as const (Array Bool Bool) false) as well as the value false for Booleans, we determine that:

store((as const (Array Bool Bool) true), false, false) =

$$[true \rightarrow false, false \rightarrow false]$$

Using our executable semantics we determine that store((as const (Array Bool Bool) true), false, false) is not equivalent to (as const (Array Bool Bool) false). Therefore, we assign (as const (Array Bool Bool) false) as the result of the store operation and obtain the following unsatisfiable formula:

store((as const (Array Bool Bool) true), false, false) = (as const (Array Bool Bool) false)

This formula is the example formula that we provided in Section 1.1 and exposed a soundness issue in Z3 from June 2020 [2].

3.3 Combining theories

In Section 3.1 and Section 3.2 we have shown that the technique can test each theory in isolation. Yet, we can also generate formulas which combine

Table 8: Operations and their respective return type.

Return Type	Operations
String	at(s, off), concat(s, t), intToStr(n), replace(s, t, u), substr(s, off, len)
Boolean	contains(s, t), prefixOf(s, t), suffixOf(s, t), equals(s, t), bvult(v, w)
Integer	indexOf(s, t, off), length(s), strToInt(s), bv2nat(v)
Bit-vector	concat(v, w), extract(a, b, v), bvnot(v), bvand(v, w), bvor(v, w), bvadd(v, w), bvmul(v, w), bvudiv(v, w), bvurem(v, w), bvshl(v, w), bvlshr(v, w)
Array	store(array((any Type), (any Type)), i, e)
any Type	select(array((any Type), (any Type)), i)

s, t, u: String; a, b, n, off, len: Int; v, w: Bit-vector of length m, where m is a strictly positive integer; a: (Array (any Type) (any Type)); i, e: (any Type)

multiple theories using the technique. This allows us to identify soundness issues which are caused by the interaction between different theories. We use the string theory [3], bit-vector theory [4] and array theory [5] to generate formulas that contain multiple theories. One can also consider the integer theory and Boolean theory to be part of the theories that are combined, as multiple formulas from Table 2 use integer and Boolean operations.

3.3.1 Operations

The string operations are taken from [26], the bit-vector operations from Table 3 and the array operations from Table 5. All used operations are summarized in Table 8.

3.3.2 Initial values

For the combination of theories we use the same initial values from Table 6 as described in Section 3.2.3. The initial values for arrays are also generated as shown in Section 3.2.3. Due to the exhaustive nature of the technique, we have to limit the string values to ”” and ”-1” as well as the integer values to 0 and -1 for unsatisfiable formulas.

3.3.3 Adapting the technique

The applicability of the individual methods and transformations, from Section 2, when combining multiple theories highly depends on the theory to which the initial operation belongs. Operations from the string theory work with each method and transformation, shown in [26] and Section 3.7. All methods

Table 9: Equalities between array operations and non-constant variables.

ID	Equality
NC1	i: Int :: select(i, (as const (Array Int (any Type)) s)) = s
NC2	i: String :: select(i, (as const (Array String (any Type)) s)) = s
NC3	i: Bool :: select(i, (as const (Array Bool (any Type)) s)) = s
NC4	i: (BitVec m) :: select(i, (as const (Array (BitVec m) (any Type)) s)) = s

s: (any Type)

and transformations are also applicable to the operations from the bit-vector theory, shown in Section 3.1. If the initial operation belongs to the array theory, then not all methods and transformations to generate unsatisfiable formulas are available, as discussed in Section 3.2.5.

Although the method to generate satisfiable formulas (Section 2.1.1) and the constant assignment transformation (Section 2.2.1) are applicable to all operations from Table 8, they do not replace any variables by more complex terms. This means that they do not introduce a second operation apart from the initial operation from which the formula is built. They do not generate any new formula that has not already been generated by applying the technique to a given theory in isolation. Therefore, we omit generating these formulas. The term synthesis transformation, on the other hand, replaces values with more complex terms which are precomputed from the operations from Table 8. Therefore, the term synthesis transformation is the only transformation that generates satisfiable formulas involving multiple theories.

To generate unsatisfiable formulas of the form $\neg A \wedge B$, we need equivalent formulas, provided for string operations in [26], for bit-vector operations in Table 2 and not applicable for array operations. To increase the unsat core of these formulas, described in Section 2.2.2, we perform variable and constant replacement using equalities shown for string operations in [26] and for bit-vector operations in Table 4. In addition, we derived equalities from the array operations, shown in Table 9, which can be used with both the string and bit-vector theory. From the newly defined transformations in Section 2.3, only the transformation from Section 2.3.2, which replaces the result with a more complex term, can generate formulas using operations from different theories. Therefore, we do not use the transformation that does not replace the result as these formulas have already been created while testing the theories in isolation. Again, we only generate test cases for formulas that contain operations from at least two different theories. An example of generating an unsatisfiable formula with multiple theories is given in Example 13.

Example 13: Creating an unsatisfiable formula with constant values and a more complex result which combines the bit-vector and string theories

Using our executable semantics, the bv2nat and strToInt operations, the bit-vector value x#0 and the string ”-1”, we determine that:

bv2nat(x#0) = 0,
strToInt(”-1”) = -1

Using our executable semantics we also determine that 0 is not equivalent to -1. By replacing the result of the bv2nat() operation by -1, we obtain the following unsatisfiable formula:

bv2nat(x#0) = -1

To increase the complexity of the formula and combine multiple theories, we now replace -1 by an operation that evaluates to -1 from another theory, such as the strToInt operation. Finally, we obtain the following unsatisfiable formula which combines the bit-vector and the string theories:

bv2nat(x#0) = strToInt(”-1”)

3.4 Checking the correctness of a model

To check the correctness of a model for a given formula, we first replace the unconstrained variables in the formula with the values from the model. Then, we use our executable semantics to evaluate the left-hand side and the right-hand side of the formula, which now only contains constant values. If both sides are equivalent then the model is sound, otherwise it is unsound.

3.5 Checking the correctness of the unsat core

We cannot check the correctness of an unsat core using the executable semantics as for a model. However, the unsat cores of the initial unsatisfiable formulas, described in Section 2.1.2, are known by construction. Further,

we known which of the transformations, which are applied to unsatisfiable formulas, increase the unsat core as well as which clause they add to the unsat core. Therefore, we can simply compare the unsat core generated by the solver with the expected unsat core, which is known by construction.

3.6 Patterns for quantifiers in unsatisfiable formulas

Bugariu and Müller describe in [26] the use of patterns to to guide the instantiations of universal quantifiers. The same patterns are used in this work for string operations used in the generation of formulas which combine multiple theories.

3.7 Applying the new transformations to strings

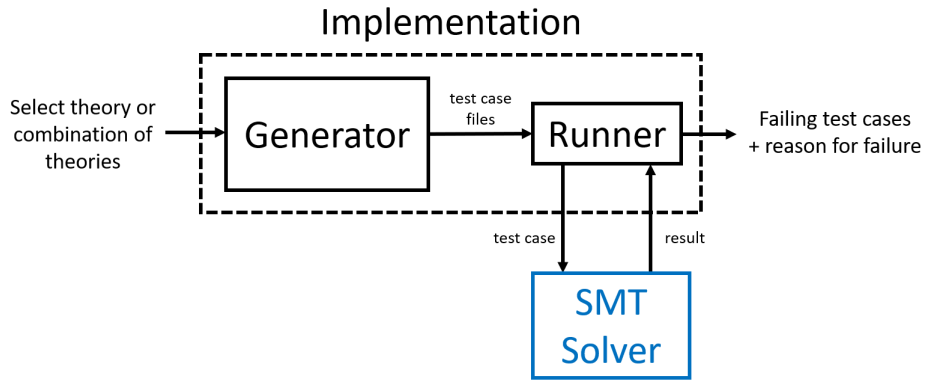
We apply the new transformations to create unsatisfiable formulas, proposed in Section 2.3 to the string theory. Thus, we extend [26] and we use the same string operations tested by Bugariu and Müller but, in our experiments, we restrict the initial values for strings to "" and "a". This is done because of the high number of possible combinations generated by exhaustively applying the technique and we reach the memory limits of our machine by using more values.

Chapter 4

Implementation

In this chapter, we discuss our implementation of the technique described in Chapter 2 with respect to the bit-vector and array theories as well as the combination of theories. We extended the original implementation from [25]. The implementation was built using Java JDK 10.0.2 [22] and uses the Z3 (4.7.1) Java API [8, 6].

Figure 1: Structure of the implementation



4.1 General structure

The implementation is divided in two main parts: the *generator* and the *runner*. The generator uses the methods and transformations described in Chapter 2 to generate satisfiable and unsatisfiable formulas. The generator then creates test case files containing the generated formulas and additional information on the satisfiability of the formula as well as a model for satisfiable formulas

and the unsat core for unsatisfiable formulas. The runner takes the generated test cases and runs them on the SMT solver under test. The runner then checks if the result returned by the solver is correct or incorrect. The general structure is shown in Figure 1, the generator is discussed in more detail in Section 4.3 and the runner in Section 4.4.

4.2 SMT-LIB reference semantics

Both the generator and the runner rely on an *executable version of the SMT-LIB semantics* to correctly evaluate the operations defined in the SMT-LIB standard [1]. [25] implements the executable semantics for string operations and provides the necessary Java classes to extend the executable semantics to the bit-vector and array operations.

New classes for SMT types are created by extending the abstract class *SMTConstructedObject*. A new SMT type class inherits multiple fields from the SMTConstructedObject class, shown in Listing 1. The name field represents the name of the modelled SMT variable. The value of the variable is represented by the value field. If the variable is the result of an operation then a reference to that operation is stored in the history field. This is important as certain transformations modify the operation from which the variable results. The isConstant field determines if a variable is currently unconstrained or if the concrete value is used. The witness field stores the last concrete value that was assigned to the variable. If the variable is unconstrained in the final formula then the witness is used as value in the generated model.

Listing 1: Fragment of SMTConstructedObject.java

```
public abstract class SMTConstructedObject extends SMTElement {

   protected String name;
   protected String value;
   protected String witness;
   protected Operation history;
   protected boolean isConstant;

}
```

New operations extend the abstract class *Operation*, shown in Listing 2. The arguments of the operation are stored in the arguments field. Subclasses of the Operation class implement the apply() function. For a specific operation the apply() function models the SMT-LIB semantics and returns the corresponding SMTConstructedObject.

Listing 2: Fragment of Operation.java

```
public abstract class Operation extends SMTElement {
```

```
   protected List<SMTConstructedObject> arguments;
   public abstract SMTConstructedObject apply();
}
```

4.2.1 Bit-vectors

Bit-vectors are modelled by our *SMTBitVector* class, shown in Listing 3. Additionally to the fields defined in its superclass, SMTConstructedObject, the SMTBitVector class has a length field. This field is used to determine the length of the given bit-vector as the SMT-LIB standard defines fixed-sized bit-vectors. To represent the value of bit-vectors we use the hexadecimal encoding as described in Section 3.1.1.

Listing 3: Fragment of SMTBitVector.java

```
public class SMTBitVector extends SMTConstructedObject {

   protected int length;

   public SMTBitVector(String value, Operation history, boolean
       isConstant, int length) {
        super(value, history, isConstant);
        this.length = length;
    }

    public int bvLength() {
      return length;
    }

}
```

4.2.2 Bit-vector operations

We use two different approaches to implement the apply() function for the different bit-vector operations:

- Directly compute the result using the value field.
- Using Java BitSets from the java.util.BitSet package [9].

For bit-vector operations which are defined in the SMT-LIB bit-vector standard [4] using mathematical operations, we take the values from the arguments, interpret them as decimal integers and apply the corresponding mathematical operation. In a final step, we re-encode the result of the mathematical operation as bit-vector and create a new SMTBitVector instance which has the encoded value and whose history is this operation. The bvadd operation is an example operation for which we use this method and is

shown in Listing 4. The other operations for which we use this method are: bv2nat, bvlshr, bvmul, bvneg, bvshl, bvudiv, bvurem, bvult and nat2bv.

Listing 4: Fragment of BVAdd.java

```
// This is a simplified version of the actual implementation

public class BVAdd extends BitVectorOperation {

   /* SMT-LIB definition: bvadd(s, t) := nat2bv(m, bv2nat(s) +
       bv2nat(t)) */
   @Override
    public SMTConstructedObject apply() {
        String firstArgValue = arguments[0].getValue();
        String secondArgValue = arguments[1].getValue();

        int firstValue = hexEncondigToInteger(firstArgValue);
        int secondValue = hexEncondigToInteger(secondArgValue);

        String result = integerToHexEncoding(firstValue + secondValue);

        return new SMTBitVector(result, this, true,
            arguments[0].bvLength());
    }
}
```

The BitSet class [9] implements non fixed-sized bit-vectors in Java. Furthermore, the BitSet class implements the and, or and flip operations. Therefore, we use BitSets to implement the apply() function of the bit-wise operations. We first construct BitSets that are equivalent to the arguments of the operation. Then we apply the BitSet version of the operation to the constructed BitSets. As a last step, we construct a new SMTBitVector from the resulting BitSet, ensuring the correct size for the bit-vector as BitSets have a non-fixed size. We implemented a wrapper function for the bvnot() operation as the flip(int bitIndex) function for BitSets only flips the bit at a given index. The wrapper function flips the bits for all positions [0, bvLength()-1]. Furthermore, we also implemented the concat and extract operations using BitSets. BVOr shows how BitSets are used in Listing 5.

Listing 5: Fragment of BVOr.java

```
// This is a simplified version of the actual implementation

public class BVOr extends BitVectorOperation {

   /* SMT-LIB definition: bvor(s, t) := lambda x:[0, m). if s[x] = 1
       then 1 else t[x] */
   @Override
   public SMTConstructedObject apply() {
```

```
      int resultLength = arguments[0].bvLength();
      String firstArgValue = arguments[0].getValue();
      String secondArgValue = arguments[1].getValue();

      BitSet firstBitSet = valueToBitSet(firstArgValue);
      BitSet seconfBitSet = valueToBitSet(secondArgValue);

      String result = bitSetToValue(firstBitSet.or(secondBitSet),
          resultLength);

      return new SMTBitVector(result, this, true, resultLength);
    }
}
```

4.2.3 Arrays

To model arrays, we defined the *SMTArray* class, shown in Listing 6. As SMT-LIB arrays can have any type as index type and value type, the SMTArray class has two fields, keyClass and valueClass, to be able to determine the type of the array. The constant arrays in the smt2 language are build using the *as const* constructor. To be able to use this constructor, the SMTArray class has the constantValue field as well as the hasConstantValue field which determines if no value, other from the constant value, has been stored in the array. The valueMap is used to record any elements that are stored in the array that differ from the constant element. This field is implemented as a HashMap<SMTConstructedObject, SMTConstructedObject>, which allows one to represent keys and values of arbitrary types.

Listing 6: Fragment of SMTArray.java

```
public class SMTArray extends SMTConstructedObject {
    protected HashMap<SMTConstructedObject, SMTConstructedObject>
        valueMap;
    protected String keyClass;
    protected String valueClass;
    protected SMTConstructedObject constantValue;
    protected boolean hasConstantValue;
}
```

4.2.4 Array operations

To implement the apply() functions for the select and store operation we use the get() and put() functions for HashMaps. To select from an array, we first check if the index is in the valueMap.keySet() of the array. If so we return valueMap.get(index), otherwise we return the constant value of the array. To store in an array, we check if the element that we want to store is

equal to the constant element of the array. If that is the case then we return a new SMTArray instance that is equivalent to the original SMTArray that we wanted to store to. Otherwise, we copy the valueMap of the original SMTArray, use valueMap.put(index, value) to perform the store operation. Finally, we return an SMTArray with the same constant value as the original SMTArray and the new valueMap.

4.3 Generator

The generator uses the executable semantics, described in Section 4.2, to generate the formulas that will be used in the test cases. The generator can be subdivided depending on the satisfiability of the formulas it generates as well as the theory that is used in the formulas.

4.3.1 Sat generators

We implemented 3 different generators that can generate satisfiable formulas. *BVSatExpressionsGenerator* implements a generator for satisfiable formulas with the bit-vector theory, *ArraySatExpressionsGenerator* uses the array theory and *CombinedSatExpressionsGenerator* combines multiple theories. These generators work similar as the *SatExpressionsGenerator* implemented in [25]. The only difference to the SatExpressionsGenerator is that the BVSatExpressionsGenerator uses SMTBitVectors and bit-vector operations instead of SMTStrings and string operations. Analogously, ArraySatExpressionsGenerator uses SMTArrays and array operations. CombinedSatExpressionsGenerator uses all types and operations from the three other sat generators.

4.3.2 Unsat generators

Analogously to the sat generators, we implemented three generators that generate unsatisfiable formulas: *BVUnsatExpressionsGenerator*, *ArrayUnsatExpressionsGenerator* and *CombinedUnsatExpressionsGenerator*. They use the respective types and operations of their theory as well as the equivalent formulas, described in Table 2, and constant/variable equalities, shown in Table 4, that are needed for the construction of unsatisfiable formulas. Furthermore, we implemented the unsatisfiability-preserving transformations presented in Section 2.3. The transformation that generates unsatisfiable formulas with constant values is shown in Listing 7. For a specific operation, we first exhaustively generate all possible combinations with the initial values. Then, we check which initial values are unequal to the result and generate the unsatisfiable formula by setting the result of the operation to these values. To generate formulas with constant values and more complex results, we perform the same steps. The only difference is that we generate more complex values using the different operations and the initial values.

We then use these more complex values to replace the result of the original operation. Our implementation for the transformation with more complex results is shown in Listing 8. ArrayUnsatExpressionsGenerator only contains the new transformations as the original transformations are not applicable.

Listing 7: Our implementation of the transformation generating unsatisfiable formulas with constant values

```
// This is a simplified version of the actual implementation
testWithConstants() {
   initialValues = initializeValues();
   for(Operation op : operationList) {
      satOperationss = createOperations(initialValues, op);
      for(Operation satOp : satOperations) {
         result = satOp.apply();
         for (SMTConstructedObject value : initialValues) {
            if(!value.equals(result)) {
               satOp.setResult(value);
               createTestCaseFile(satOp);
            }
         }
      }
   }
}
```

4.4 Runner

The runner was implemented in *TestRunner* in [25]. It uses the test cases created by the generator and runs them on the SMT solver under test. As the test cases also encode the satisfiability of the formulas, the runner knows what the expected output is. If the solver returns the correct satisfiability then the runner checks if a correct model, using the executable semantics, or a correct unsat core was returned. If a test case passes both checks then it is marked as passing. If a test case fails any of the checks then it is recorded as failing and the runner records the cause of the failure as well as any possible error messages.

Listing 8: Our implementation of the transformation generating unsatisfiable formulas with constant values and more complex results

```
// This is a simplified version of the actual implementation
testWithConstantsAndComplexResult() {
   initialValues = initializeValues();
   complexValues = createComplexValues(initialValues, operationList);
   for(Operation op : operationList) {
      satOperationss = createOperations(initialValues, op);
      for(Operation satOp : satOperations) {
```

```
            result = satOp.apply();
            for (SMTConstructedObject value : complexValues) {
               if(!value.equals(result)) {
                  satOp.setResult(value);
                  createTestCaseFile(satOp);
               }
            }
         }
      }
   }
```

Chapter 5

Evaluation

In this chapter, we discuss the evaluation of the technique on known issues of older versions of Z3 and CVC4 (Section 5.2). We also present the results of our tests on the latest versions of the SMT solvers (Section 5.3) and discuss the limitations of the technique and of our implementation (Section 5.4).

5.1 Experimental setup

Our tests are performed in a virtual machine running Ubuntu 20.04.3 with 10 GB memory. Otherwise, we use the same solver settings as in [26]. The random seed is fixed for all test cases and the timeout limit is set to 15 seconds. The options produce-models and produce-unsat-cores are enabled for all solvers. CVC4 is run with the strings-exp and full-saturate-quant options. Additionally, Z3 is run with smt.core.minimize option enabled. Due to memory limitations, the different transformations were only applied once in the test cases that we generated.

5.2 Evaluating the technique

To evaluate the effectiveness of the technique, we looked at known *soundness* issues for Z3 4.7.1 and CVC4 1.6. Afterwards, we ran the technique on both solvers and manually matched failing test cases with the known bugs.

5.2.1 Identifying known issues

To determine the known issues we looked at the issue trackers of Z3 [10] and CVC4 [11]. We looked at the issues reported from 23^{rd} May 2018, the release date of Z3 4.7.1, until 1^{st} January 2021 for Z3 4.7.1. For CVC4 1.6, we looked at issues reported from 26^{th} June 2018, the release date of CVC4 1.6, until 1^{st} January 2021. We only counted soundness issues which we

Table 10: Overview of known soundness issues from Z3 4.7.1

Theory	Known Issues	In Scope	Found
Bit-vector	5	0	0
Array	5	3	2
Combined	2	1	0

We consider issues caused by other theories, different configurations, user-defined sorts, user-defined functions and operations not defined in SMT-LIB out of scope.

could reproduce with Z3 4.7.1 and CVC4 1.6 respectively. We consider issues caused by other theories, different configurations, user-defined sorts, user-defined functions and operations not defined in SMT-LIB out of the scope of this work. The issues, summarized in Table 10 and Table 11, are grouped by theory. For the combination of theories, we considered issues that use any combination of the string, Boolean, bit-vector, integer or array theory. The list of known soundness issues that we identified on the issue tracker is provided in Appendix A.1 for Z3 4.7.1 and in Appendix A.2 for CVC4 1.6.

Table 11: Overview of known soundness issues from CVC4 1.6

Theory	Known Issues	In Scope	Found
Bit-vector	1	0	0
Array	2	2	0
Combined	3	2	0

We consider issues caused by other theories, different configurations, user-defined sorts, user-defined functions and operations not defined in SMT-LIB out of scope.

5.2.2 Performance of the technique

We manually investigated the failing test cases from our technique and matched them to known soundness issues. The results of our test cases for Z3 4.7.1 are shown in Table 12 and for CVC4 1.6 in Table 13. The first column shows the current theory under test. The second column denotes the satisfiability of the generated formulas. [p] in the second column indicates the use of patterns to instantiate the quantifiers used in the equivalent formulas. The third column indicates the total number of test cases generated. The remaining columns represent the individual reasons due to which a test case could fail. To match failing test cases to known soundness bugs, we looked at the highlighted columns of these tables as they represent soundness issues. The comparison for Z3 4.7.1 is presented in Table 10 and for CVC4 1.6 in Table 11. The number of soundness bugs we found is shown in the last

Table 12: Overview of the results for Z3 4.7.1

Theory	Expected	#Tests	Passed	IS	IM	IC	U	T	E
Bit-vector	SAT	9211	9211	0	0	0	0	0	0
Bit-vector	UNSAT	6109	6088	0	0	21	0	0	0
Bit-vector	UNSAT[p]	595	575	0	0	20	0	0	0
Array	SAT	8342	8310	0	0	0	32	0	0
Array	UNSAT	6920	5584	1336	0	0	0	0	0
Combined	SAT	8222	6824	0	82	0	1209	107	0
Combined	UNSAT	5013	4773	3	0	0	29	218	0
Combined	UNSAT[p]	696	454	4	0	0	29	209	0

IS = incorrect satisfiability; IM = incorrect model; IC = incorrect unsat core; U = unknown; T = timeout; E = error; [p] = patterns for quantified variables

column. The technique was able to find 40% of the array issues and none of the issues resulting from the combination of theories that we consider to be in scope. There were no bit-vector issues that we considered to be in scope. Generally, there are two main reasons why our test cases did not reveal more known issues:

- The formula exposing the issue can be generated by applying some transformations from the technique multiple times.
- The formula contains operations from the integer or Boolean theory which are not used to generate formulas in our current implementation.

Note that the results of our test cases on Z3 4.7.1 show incorrect unsat cores for bit-vector formulas (Table 12, column IC). These unsat cores are not minimal but correct. As such, we consider them to be imprecise rather than unsound. Furthermore, the soundness issues for the combination of theories are caused by string operations and are not rooted in the combination of theories. The incorrect models in Table 13 (column IM) are rooted in an soundness issue in CVC4 that was not reported before CVC4 version 1.8.

5.3 Testing latest versions of SMT solvers

We used the technique to detect soundness issues on the latest versions of Z3 and CVC4.

5.3.1 Z3 4.8.12

The results of our tests for the latest version of Z3, version 4.8.12, are provided in Table 14. The most interesting result is highlighted in red. We generated 32 test cases which exposed a previously-unknown soundness issue in Z3. Z3 returned SAT for an unsatisfiable formula which compares two arrays which are indexed by bit-vectors. An example of such a formula is provided

Table 13: Overview of the results for CVC4 1.6

Theory	Expected	#Tests	Passed	IS	IM	IC	U	T	E
Bit-vector	SAT	9211	9203	0	0	0	0	0	8
Bit-vector	UNSAT	6109	6109	0	0	0	0	0	0
Bit-vector	UNSAT[p]	595	595	0	0	0	0	0	0
Array	SAT	8342	6798	0	0	0	0	0	1544
Array	UNSAT	6920	5252	0	0	0	0	0	1668
Combined	SAT	8222	7607	0	18	0	0	0	597
Combined	UNSAT	5013	4230	0	0	0	0	189	594
Combined	UNSAT[p]	696	131	0	0	0	0	189	376

IS = incorrect satisfiability; IM = incorrect model; IC = incorrect unsat core; U = unknown; T = timeout; E = error; [p] = patterns for quantified variables

in Example 14. We reported this soundness issue to the Z3 developers who confirmed and fixed the issue.

Note that again the 15 test cases that result in an incorrect unsat core are in fact correct unsat cores which are just not minimal.

Example 14: An unsatisfiable formula exposing a soundness issue in Z3 4.8.12

The following formula is an example of the generated formulas which exposed an soundness issue in Z3 4.8.12:

```
store((as const (Array (BitVec 4) Int) 0), x#0, 1) =
    store((as const (Array (BitVec 4) Int) 1), x#0, 0)
```

Z3 returned SAT for this unsatisfiable formula.

5.3.2 CVC4 1.8

We tested CVC4 version 1.8, which is the last version of CVC4, afterwards the developers switched to CVC5. The results are shown in Table 15. As before, the most interesting results are highlighted in red. Our technique was able to generate test cases for which CVC4 returned an incorrect model. These test cases combine the array, string and bit-vector theory. An example of such a formula is given in Example 15.

Furthermore, the issue seems to be already present in CVC4 1.6. On the other hand, the issue is no longer present in the current version of CVC5. As such we consider this issue known and fixed.

Table 14: Overview of the results for Z3 4.8.12

Theory	Expected	#Tests	Passed	IS	IM	IC	U	T	E
Bit-vector	SAT	9211	9211	0	0	0	0	0	0
Bit-vector	UNSAT	6109	6097	0	0	0	0	12	0
Bit-vector	UNSAT[p]	595	582	0	0	0	0	13	0
Array	SAT	8342	8342	0	0	0	0	0	0
Array	UNSAT	6920	6888	32	0	0	0	0	0
Combined	SAT	8222	8200	0	0	0	5	17	0
Combined	UNSAT	5013	4747	0	0	8	0	258	0
Combined	UNSAT[p]	696	438	0	0	7	0	251	0

IS = incorrect satisfiability; IM = incorrect model; IC = incorrect unsat core; U = unknown; T = timeout; E = error; [p] = patterns for quantified variables

Note that the high number of test cases that report an error for CVC4 (Table 15, column E) are due to the fact that CVC4 does not support certain features that Z3 supports. For instance, CVC4 does not support write chains with constant arrays nor a non-constant argument for the as const array constructor. Furthermore, CVC4 1.8 renamed str.to.int and int.to.str and does not support the old names any more. We did not have the time to implement a solver based rewrite of these two methods for CVC4 1.8.

5.3.3 Evaluation for the string theory

As we implemented the two new transformations for unsatisfiable formulas (Section 2.3) for the string theory, we tested both Z3 4.8.12 and CVC4 1.8 on the generated test cases. Both solvers returned the correct result for all 11.036 test cases.

Table 15: Overview of the results for CVC4 1.8

Theory	Expected	#Tests	Passed	IS	IM	IC	U	T	E
Bit-vector	SAT	9211	9211	0	0	0	0	0	0
Bit-vector	UNSAT	6109	6109	0	0	0	0	0	0
Bit-vector	UNSAT[p]	595	595	0	0	0	0	0	0
Array	SAT	8342	7116	0	0	0	0	0	1126
Array	UNSAT	6920	5252	0	0	0	0	0	1668
Combined	SAT	8222	6993	0	16	0	0	0	1213
Combined	UNSAT	5013	4114	0	0	0	0	25	874
Combined	UNSAT[p]	696	136	0	0	0	0	25	535

IS = incorrect satisfiability; IM = incorrect model; IC = incorrect unsat core; U = unknown; T = timeout; E = error; [p] = patterns for quantified variables

Example 15: A satisfiable formula exposing a soundness issue in CVC4 1.8

The following formula is an example of the generated formulas which exposed an soundness issue in CVC4 1.8:

select(store(a, bv, b), bvnot(bv)) = contains(s, s)

with a: (Array (BitVec 4) Bool), bv: (BitVec 4), b: Bool, s: String. CVC4 returned the following model for this formula:

a = store((as const (Array (BitVec 4) Bool) false), x#1, true),
bv = x#0,
s = "",
b = false

which is unsound because, with these values, the left-hand side evaluates to false, while the right-hand side evaluates to true.

5.3.4 Other issues

Out tests also revealed other issues such as completeness issues in Z3 4.8.12 (Table 14, column U) and performance issues in both Z3 4.8.12 and CVC4 1.8 (Table 14 and Table 15, column T). Furthermore, CVC4 1.8 threw a "non-constant used as constant element" exception for the as const array constructor using the constant value (- 1) (Table 15, column E). This issue was already reported to the CVC4 issue tracker [12].

5.4 Limitations

We discovered limitations of both the technique and our implementation throughout the process of evaluating our technique on known issues as well as testing current verions of Z3 and CVC4.

5.4.1 Limitations of the technique

The transformations to generate satisfiable formulas can be applied to all theories, as only the set of operations and initial values need to be defined. However, the transformations to generate unsatisfiable formulas cannot be completely applied to every theory. The original transformations for unsatisfiable formulas from [26] require equivalent formulas which are not always

possible to construct. In this case, the technique can only generate unsatisfiable formulas with constant values, while there might exist unsatisfiable formulas with unconstrained variables. The array theory is an example of such a theory.

5.4.2 Limitations of the implementation

The main limitation of the implementation is its memory management. Due to the exhaustive nature of the technique, the implementation often runs out of memory. As a result, for certain theories, the set of initial values has to be limited and one might have to exclude interesting corner cases. Additionally, the memory limitation allows the implementation to only generate formulas where the transformations are applied once. Yet many formulas that are reported on the issue trackers have multiple nested operations. Furthermore, the current implementation only uses a limited set of operations from the integer and Boolean theory. As a result some soundness issues cannot be detected, as shown in Section 5.2.2.

Chapter 6

Related work

In this chapter we discuss work related to this thesis.

Existing benchmarks. The SMT-LIB Initiative maintains a list of existing SMT benchmarks [13] which can be used for testing. Furthermore, the developers of SMT solver created their own benchmarks, including tests which are derived from reported issues [14, 15]. Often, these tests have to be written by hand which is very time-consuming. This work facilitates the benchmark generation by automatically generating test cases with known ground truth, i.e., which are satisfiable or unsatisfiable by construction.

Differential testing. Differential testing [31] is a widely used method to test software. To test SMT solvers with this approach, one runs the same benchmark on two different SMT solvers and compares the result. If the solvers return different results, then an issue was found. Yet to determine which solver is affected needs further investigation. Furthermore, an issue might stay undetected when both solvers return the same incorrect result. The technique used in this work does not need a second solver to compare the result to, as the result of the benchmark is known by construction. Additionally, the technique can find soundness issues caused by incorrect models due to our executable semantics as well as incorrect unsat cores. Differential testing cannot expose such issues.

Fuzzing. Brummayer et al. apply blackbox fuzzing for the bit-vector theory [24]. Scott et al. apply fuzzing to all SMT-LIB theories as well as their combinations [35]. Yao et al. fuzz SMT solvers via a 2-dimensional input space exploration [38]. Mansur et al. apply blackbox mutational fuzzing to all Z3-supported logics [30]. Generally, these works are the closest related to this work as they can generate test cases for the bit-vector and array theory. Benchmarks generated by fuzzing cannot reliably detect soundness issues, as they do not posses a test oracle, and rely on differential testing to do so.

The test cases, that fuzzing generates, are often not minimal and require additional techniques such as delta debugging [39] to reduce their size. The technique used in this work, on the other hand, does not rely on differential testing. Furthermore, the technique gradually increases the complexity of the test cases and therefore does not need delta debugging.

Semantic fusion. Winterer et al. [37, 36] proposed a novel approach to create new test cases called semantic fusion. This approach fuses two test cases into a new test case with known result. Semantic fusion requires a set of seed formulas with known results from which it starts to generate new test cases. Our work does not need such a set of initial formulas as it generates them itself. Further, Winterer et al. focused on testing the core and string theories, while our work focuses on the bit-vector and array theory as well as the combinations of string, bit-vector, array and integer therories.

Generative type-aware mutation. Park et al. [34] proposed a hybrid technique which combines type-aware operator mutation, first proposed by Winterer et al. in [37, 36] with grammar-based fuzzing. This approach has an infinite mutation space, overcoming the main limitation of semantic fusion which is its finite mutation space. In their work Park et al. focused on the integer, real, reals_ints and string theories. Our work is able to generate formulas for the bit-vector and array theory as well as the combinations of string, bit-vector, array and integer therories.

Chapter 7

Conclusion

The main focus of this thesis was to automatically detect soundness issues in SMT solvers with respect to the bit-vector and array theory. To do so, we adopted the technique proposed by Bugariu and Müller which generates formulas that are satisfiable or unsatisfiable by construction. The technique then uses satisfiability preserving transformations to increase the complexity of the generated formulas. We defined two new transformations for the creation of unsatisfiable formulas which allows us to test SMT solvers more rigorously. Further, we also adapted the technique to be able to generate formulas which combine operations from the string, bit-vector, array and integer theory. To achieve this, we implemented an executable version of the SMT-LIB semantics for the bit-vector and array theory. We then used the technique to test the latest versions of two widely used SMT solver, Z3 4.8.12 and CVC4 1.8. Our experimental evaluation shows that the technique is able to find soundness issues in both, which have been confirmed (for Z3) and fixed (for Z3 and CVC4) by the developers.

7.1 Future work

Extend the technique to other theories. Currently, the technique allows the generation of formulas with operations from the string, bit-vector and array theory. The integer and core theory, which defines the Boolean operations, are only used in a limited manner. They are mostly used in the equivalent formulas of the aforementioned theories. As some of the operations for the integer [16] and core theory [17] are already implemented, one can extend the technique to fully support the generation of formulas with operations from these two theories. Furthermore, SMT-LIB defines three additional theories: the reals theory [18], the floating point theory [19] and the reals_ints theory [20]. The technique can be extended to also include the operations from these theories. Although, for some theories, one might not be able create equivalent

formulas, limiting the technique to the new transformations for unsatisfiable formulas. However, even in that case, one is able to generate satisfiable and unsatisfiable formulas. Generally, to extend the technique to a new theory, one needs to extend the executable SMT-LIB semantics with the operations from the theory. One also needs to define the equivalent formulas, if possible, to generate unsatisfiable formulas. Further, one needs to define initial values for the variables as well as variable equalities to increase the unsat core. The variable equalities, even if the theory does not support the original transformations for unsatisfiable formulas, are used for the combination of theories.

Test other provers. There exist more provers, other than Z3 and CVC4, which can reason about SMT-LIB formulas. One could use the technique to test these provers and detect soundness issues in their implementations. The list of possible provers includes: CVC5 [21], the successor of CVC4, the Vampire theorem prover [29], Boolector [32] and STP [28].

Improve the memory efficiency of the implementation. Our current implementation tends to run out of memory due to the exhaustive nature of the technique, as described in Section 5.4.2. As a result, we are only able to generate formulas for which the transformations were applied at most once. Thus, if the memory efficiency of the implementation is improved, one could generate more complex formulas.

Appendix A

Known soundness issues in Z3 4.7.1 and CVC4 1.6

A.1 Known soundness issues in Z3 4.7.1

List of bit-vector soundness issues considered out of scope:

- https://github.com/Z3Prover/z3/issues/2136
- https://github.com/Z3Prover/z3/issues/2173
- https://github.com/Z3Prover/z3/issues/2933
- https://github.com/Z3Prover/z3/issues/2965
- https://github.com/Z3Prover/z3/issues/4048

List of array soundness issues considered out of scope:

- https://github.com/Z3Prover/z3/issues/1847
- https://github.com/Z3Prover/z3/issues/4181

List of array soundness issues considered in scope:

- https://github.com/Z3Prover/z3/issues/2549
- https://github.com/Z3Prover/z3/issues/4515
- https://github.com/Z3Prover/z3/issues/4778

List of combined soundness issues considered in scope:

- https://github.com/Z3Prover/z3/issues/4808

List of combined soundness issues considered in scope:

- https://github.com/Z3Prover/z3/issues/4923

A.2 Known soundness issues in CVC4 1.6

List of bit-vector soundness issues considered out of scope:

- https://github.com/cvc5/cvc5/issues/4437

List of array soundness issues considered in scope:

- https://github.com/cvc5/cvc5/issues/4414
- https://github.com/cvc5/cvc5/issues/4758

List of combined soundness issues considered in scope:

- https://github.com/cvc5/cvc5/issues/4780

List of combined soundness issues considered in scope:

- https://github.com/cvc5/cvc5/issues/4546
- https://github.com/cvc5/cvc5/issues/4771

Bibliography

[1] SMT-LIB Library. (Visited: 14 Nov. 2021)
http://smtlib.cs.uiowa.edu/index.shtml.

[2] Z3 Solver issue 4515. (Visited: 14 Nov. 2021)
https://github.com/Z3Prover/z3/issues/4515.

[3] SMT-LIB Theory Declaration for Strings. (Visited: 14 Nov. 2021)
http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml.

[4] SMT-LIB Theory Declaration for Bitvectors. (Visited: 14 Nov. 2021)
http://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml.

[5] SMT-LIB Theory Declaration for Arrays. (Visited: 14 Nov. 2021)
http://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml.

[6] Z3 Java API Documentation. (Visited: 14 Nov. 2021)
https://z3prover.github.io/api/html/namespacecom_1_1microsoft_1_1z3.html.

[7] Z3 int2bv/bv2int Comment. (Visited: 14 Nov. 2021)
https://github.com/Z3Prover/z3/issues/1481#issuecomment-365030093.

[8] Z3 on GitHub. (Visited: 14 Nov. 2021)
https://github.com/Z3Prover/z3.

[9] Java BitSet class. (Visited: 14 Nov. 2021)
https://docs.oracle.com/javase/7/docs/api/java/util/BitSet.html.

[10] Z3 Issue Tracker. (Visited: 14 Nov. 2021)
https://github.com/Z3Prover/z3/issues.

[11] CVC4 Issue Tracker. (Visited: 14 Nov. 2021)
https://github.com/cvc5/cvc5/issues.

[12] CVC4 as const Constructor issue. (Visited: 14 Nov. 2021) https://github.com/cvc5/cvc5/issues/7596.

[13] SMT-LIB Benchmarks. (Visited: 14 Nov. 2021) http://smtlib.cs.uiowa.edu/benchmarks.shtml.

[14] Z3 Test Suite. (Visited: 14 Nov. 2021) https://github.com/Z3Prover/z3/tree/master/src/test.

[15] CVC5 Test Suite. (Visited: 14 Nov. 2021) https://github.com/cvc5/cvc5/tree/master/test.

[16] SMT-LIB Theory Declaration for Integers. (Visited: 14 Nov. 2021) http://smtlib.cs.uiowa.edu/theories-Ints.shtml.

[17] SMT-LIB Core Theory Declaration. (Visited: 14 Nov. 2021) http://smtlib.cs.uiowa.edu/theories-Core.shtml.

[18] SMT-LIB Theory Declaration for Reals. (Visited: 14 Nov. 2021) http://smtlib.cs.uiowa.edu/theories-Reals.shtml.

[19] SMT-LIB Theory Declaration for Floating Point Numbers. (Visited: 14 Nov. 2021) http://smtlib.cs.uiowa.edu/theories-FloatingPoint.shtml.

[20] SMT-LIB Theory Declaration for Reals and Integers. (Visited: 14 Nov. 2021) http://smtlib.cs.uiowa.edu/theories-Reals_Ints.shtml.

[21] CVC5. (Visited: 14 Nov. 2021) https://github.com/cvc5/cvc5.

[22] Java SE Development Kit 10.0.2. (Visited: 14 Nov. 2021). https://www.oracle.com/java/technologies/java-archive-javase10-downloads.html.

[23] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[24] Robert Brummayer and Armin Biere. Fuzzing and Delta-Debugging SMT Solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, page 1–5, New York, NY, USA, 2009. Association for Computing Machinery.

[25] Alexandra Bugariu and Peter Müller. StringSolversTests. (Visited: 14 Nov. 2021). https://github.com/alebugariu/StringSolversTests.

[26] Alexandra Bugariu and Peter Müller. Automatically Testing String Solvers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1459–1470, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[28] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[29] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[30] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 701–712, New York, NY, USA, 2020. Association for Computing Machinery.

[31] William M. McKeeman. Differential Testing for Software. *DIGITAL TECHNICAL JOURNAL*, 10(1):100–107, 1998.

[32] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014.

[33] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Yoni Zohar, Clark Barrett, and Cesare Tinelli. Towards Bit-Width-Independent Proofs in SMT Solvers. In Pascal Fontaine, editor, *Automated Deduction – CADE 27*, pages 366–384, Cham, 2019. Springer International Publishing.

[34] Jiwon Park, Dominik Winterer, Zhendong Su, and Chengyu Zhang. Generative type-aware mutation for testing SMT solvers. 2021-10.

[35] Joseph Scott, Trishal Sudula, Hammad Rehman, Federico Mora, and Vijay Ganesh. BanditFuzz: Fuzzing SMT Solvers with Multi-agent Reinforcement Learning. In Marieke Huisman, Corina Păsăreanu, and Naijun Zhan, editors, *Formal Methods*, pages 103–121, Cham, 2021. Springer International Publishing.

[36] Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[37] Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT Solvers via Semantic Fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 718–730, New York, NY, USA, 2020. Association for Computing Machinery.

[38] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 322–335, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

AUTOMATICALLY TESTING SMT SOLVERS

Authored by (in block letters):

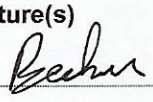
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
BECKER	OLIVIER

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 16th November 2021	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.