Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Better SMT Language: Design & Tooling

Bachelor Thesis

Nico Darryl Hänggi

October 6, 2020

Advisor: Dr. Malte Schwerhoff
Supervisor: Prof. Dr. Peter Müller

Department of Computer Science, ETH Zürich

Abstract

In the last decade, SMT solvers have become an essential part of computer-aided verification and analysis of software. To interact with an SMT solver, many solvers adhere to the SMT-LIB specification, an international standard for expressing logic terms and formulas, featuring a fully parenthesized prefix notation. While this syntax is easy to parse, it is not very human-friendly and debugging SMT encodings is rendered extremely challenging.

In this thesis, we developed `yuk-that-smt`, a command-line interface tool that helps developers to understand and debug SMT-LIB encodings more easily. For this, we propose a novel SMT source language, called B-SMT, that features both a more human-friendly and extendable syntax at the same time. Our command-line-interface makes automatic conversion between SMT-LIB and B-SMT possible, whilst providing a user-friendly and performant experience. To illustrate the extension capabilities of our language, we propose three additional extensions that help to reduce further the encoding gap between SMT-LIB and our novel language.

As `yuk-that-smt` offers an extendable plugin architecture, it can be employed in a broad range of application areas and be adapted towards supporting a wide range of state-of-the-art program verifiers.

Acknowledgements

I want to express my sincere gratitude to Dr. Malte Schwerhoff, not only for guiding me through this work during the past six months and sharing his expertise on the matter but also for offering valuable feedback on my ideas and bringing up many on his own. Moreover, I would like to thank Dr. Peter Müller and the Programming Methodology Group for providing me with this opportunity in the first place; contributing to a state-of-the-art formal verifier was both demanding and fulfilling at the same time.

Contents

Chapter 1

Introduction

Ever since Floyd [15] and Hoare [22] set the basis for program verification in the late 1960s, the importance of formally verifying software correctness has been rising due to the sheer growth in application complexity. Since then, many different program verifiers with diverse application areas have emerged and been used widely in industry. For many of those tools, a Verification Condition Generator (VCG) [8] sits at the core of the verification process by producing formal verification conditions that are subsequently validated using an automated theorem prover.

SMT solvers, one particular type of such theorem provers, have been playing a leading role in automated reasoning in the last decade. While there exist numerous different SMT solvers, nearly all share a standard input and output language, SMT-LIB [5], to allow for checking the satisfiability of first-order logic formulas. Developers of program verifiers often need to understand and modify the generated SMT-LIB encodings to be able to reason about the correctness of the translated verification conditions in order to identify possible bugs in the resulting SMT encoding. Unfortunately, the fully parenthesized prefix notation of SMT-LIB coupled with the lack of a simple encoding for basic imperative language concepts leads to a significant encoding gap between the original program code and the resulting SMT-LIB encodings. As a result, it can be tedious and time-consuming to identify bugs and to reason about the resulting encoding for tool implementers.

The main focus of this project is to reduce the encoding gap between what the high-level program code describes and how said code is represented in SMT-LIB. To accomplish this, we propose an alternative syntax for SMT encodings that helps implementers of program verifiers to understand and reason about the underlying SMT code more efficiently. Moreover, we will allow users to customize parts of the syntax mentioned above. Based on this, we implement a CLI tool, named `yuk-that-smt`, that allows efficient conversion from SMT-LIB to the B-SMT language (and vice versa).

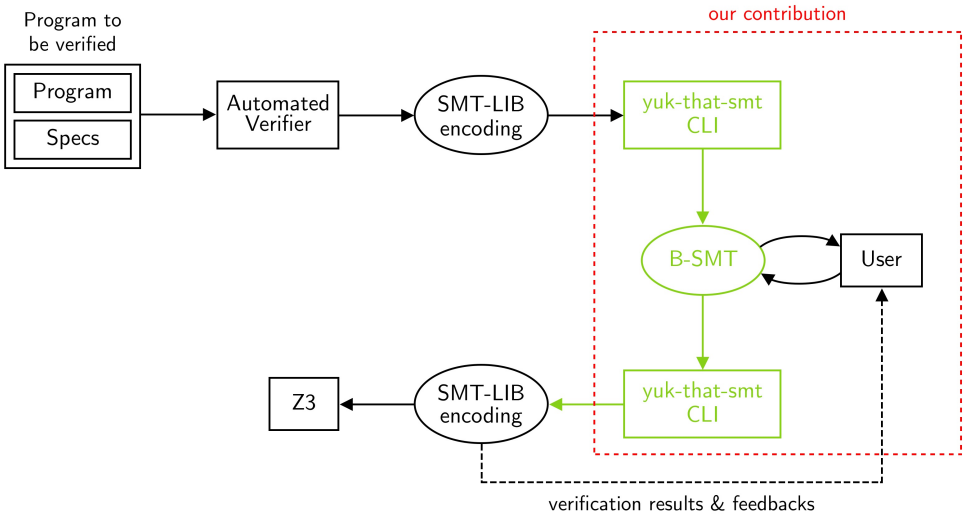

Figure 1.1: Debugging workflow with our CLI tool

We will now illustrate how our contribution facilitates the debugging workflow on a simple example. In Figure 1.1, the novel debugging workflow leveraging our CLI tool is shown. In the following, assume that we have to reason about the SMT-LIB code presented in Listing 1.1. The encoding introduces an algebraic `List` datatype with the two constructors `nil` and `cons` (see Section 2.1.4).

```
1 (declare-datatypes (T) ((List nil (cons (hd T) (tl List
    )))))
```

Listing 1.1: Algebraic datatype `List` in SMT-LIB

However, the parenthesized notation makes the encoding difficult to understand, and thus, we leverage the power of `yuk-that-smt` to produce a more human-friendly encoding, presented in Listing 1.2. We can then modify the encoding on the B-SMT level and convert it back to SMT-LIB, invoking an SMT solver to check the verification result, as depicted in Figure 1.1.

```
1 data List[T] = nil :| cons(hd: T, tl: List[T])
```

Listing 1.2: Algebraic datatype `List` in B-SMT

This thesis is structured as follows: In Chapter 2, we begin with a thorough introduction to the SMT-LIB specification, along with a brief overview of the Z3 Theorem Prover. In Chapter 3, we propose B-SMT, our novel SMT language. With the prerequisites covered in the first two chapters, in Chapter 4, we will discuss the extendability of B-SMT along with potential language extensions

in more detail. Chapter 5 motivates the language of choice for this thesis: TypeScript. In Chapter 6, we finally present `yuk-that-smt`, our command-line interface tool. The performance and correctness of `yuk-that-smt` are evaluated in Chapter 7, and ultimately, in Chapter 8, we provide a list of issues that could be addressed in future work.

Chapter 2

The SMT-LIB Language

SMT solvers form the basis of many program verification architectures. Usually, program verifiers translate their source programs into logical formulas which are then discharged using an SMT solver. For instance, the Viper verification infrastructure [37] developed at ETH Zürich introduces an intermediate language called Viper that provides simple imperative constructs, as well as statements for managing permission-based reasoning. Silicon [32], one of Viper's automated verifiers that leverages symbolic execution, then translates these programs into logical verification conditions in the form of SMT-LIB scripts. In the following chapter, we will introduce the SMT-LIB language standard along with the Z3 Theorem Prover, one of the most popular SMT solvers.

2.1 SMT-LIB

The SMT-LIB standard defines both a language to express logic terms and formulas in, and, at the same time, provides a scripting language that acts as a textual interface to communicate with SMT solvers in a read-eval-print loop (REPL) fashion. There have been many iterations of the SMT-LIB standard since its inception in 2003 [5]. Our work will mainly focus on Version 2.6 of the said standard; the latest official release at the time being. In the following, we will briefly explore the SMT-LIB syntax along with the three main components of the language: sorts, terms and commands. Furthermore, SMT-LIB currently does not support higher-order functions and SMT-LIB functions always need to be total. For a more thorough introduction to SMT-LIB, the reader is referred to the official standard [5].

2.1.1 Syntax

SMT-LIB features a syntax that is almost identical to the parenthesized notation introduced by Common Lisp. All commands, sorts, and terms are indeed

valid S-expressions, also known as symbolic expressions. S-expressions represent a list of nested tree-like structures, with the operator or function name coming first, followed by the respective operands or function arguments. Due to this prefix notation, parsing is facilitated considerably at the expense of a more human-friendly source syntax. However, because developers usually do not work on the generated SMT-LIB encoding itself, this compromise is tolerated in favour of tool performance.

```
1 (declare-const a Int)
2 (declare-const b Int)
3 (assert (< a b))
4 (assert (> a b))
5 (check-sat)
```

Listing 2.1: Checking satisfiability of given assertions

Before exploring the different aspects of the SMT-LIB language in more detail, we present a simple example. In Listing 2.1, we show how to check the satisfiability of a simple encoding in SMT-LIB. The first two commands introduce the uninterpreted constants `a` and `b`. The assertion on line 3 states that `a` must be smaller than `b`. Likewise, the next command asserts the opposite; requiring constant `a` to be greater than `b`. The **`check-sat`** command then enquires the underlying SMT solver about the satisfiability of the specified encoding. In this case, it is trivial to infer that the encoding is unsatisfiable, which is confirmed by the solver.

2.1.2 Sorts

In SMT-LIB, all terms and formulas are strongly-typed; that is, each one is associated with precisely one type. Consequently, SMT-LIB introduces the notion of sorts, which are the equivalent to types in logical terminology. In addition to simple non-parametric sorts, SMT-LIB also offers parametric types that share many similarities with the identically named feature in Haskell [23].

Numerous built-in sorts are available by default, including `Int`, `Real` and `Bool`. Moreover, many of the SMT-LIB background theories define additional sorts that can be used when the respective theory is enabled. For instance, the `ArraysEx` [25] background theory introduces the binary parametric sort `(Array o1 o2)`, where `o1` is an arbitrary sort denoting the index position type. Similarly, `o2` stands for the type of array elements. As a result, this theory allows us to represent both flat and arbitrarily nested arrays. For example, the sort `(Array Int Bool)` denotes an array of booleans, indexed by integers. In contrast, the sort `(Array Int (Array Int Int))` describes a (nested) two-dimensional integer array.

Additionally, SMT-LIB allows the declaration of new sorts, which are always uninterpreted; that is, no prior interpretation is attached to them.

2.1.3 Terms

Terms are arguably the most crucial aspect of the SMT-LIB language. They allow us to add specific restrictions on our logical model. In SMT-LIB, terms are well-sorted, and all formulas used in assertions are required to be of the Boolean sort. Similar to expressions in other programming languages, terms consist of a nested combination of operator and function applications, literals, and constant variables. In Listing 2.1, we have already seen two basic function application terms, namely `(< a b)` and `(> a b)`.

Quantifiers

In addition to the simple terms mentioned above, SMT-LIB also supports both the existential and universal quantifier known from first-order logic. For this, the language introduces the `forall` and `exists` keywords. Let us now consider Listing 2.2. Suppose we have an uninterpreted function `ident` that needs to behave like the identity function, i.e. always return the provided argument. To achieve this, we add an assertion that quantifies over all variables `x` of type `Int`. The quantifier now ensures that for every integer `x`, all calls to the identity function having `x` as an argument return `x` again.

```
1 (assert (forall ((x Int)) (= (ident x) x)))
```

Listing 2.2: Making assertions about all members of a set

Quantifying over multiple variables at the same time is also possible: one can list them next to each other. Likewise, the syntax for the existential quantifier `exists` is (except for the keyword) identical to what we have shown above.

Variable Binding

With the `let` binding operator, we can introduce new local variables in parallel. Generally, the `let` operator is of the form `(let ((v1 t1) ... (vn tn)) t)`. It introduces `n` new variables named `v1` to `vn` and each of those corresponds to exactly one term named `t1` to `tn`. Now, let us explore how we can refactor Listing 2.2 using the newly proposed variable binding operator. For this, consider Listing 2.3, where we introduce a new variable `res` that corresponds to the term `(ident x)`. Hence, in the body of the binding operator, we can then use `res` to refer to said function call. Semantically speaking, variable bindings are equivalent to replacing all occurrences of the newly defined variables with their corresponding assignment.

```
1 (assert (forall ((x Int)) (let
2         ( (res (ident x)) )
3         (= res x))
4 ))
```

Listing 2.3: Refactored Listing 2.2 with `let` binding operator

Pattern Matching

With the addition of datatypes in SMT-LIB Version 2.6 (see Section Datatypes), pattern matching on algebraic types was introduced together with it. Similar to how pattern matching works in Haskell or other functional programming languages, pattern matching in SMT-LIB allows you to match an instance of a data structure against constructors of that specific type. This information is then used as a basis for deciding what term should be returned. However, in contrast to Haskell, the matching cases must be exhaustive, i.e. every constructor of that datatype must be present in at least one of the patterns.

In Listing 2.4, we will investigate how to pattern match on a tree-like datatype. Let us assume the symbol `tree` is an instance of a tree-like structure with two constructors; `leaf` having no parameters and `node` having three: a value attached to the node itself together with left and right subtrees. For the declaration of such a data structure, please refer to Listing 2.6. The first matching case ensures that whenever `tree` is a `leaf`, the value `0` is returned as a result. Conversely, if `leaf` is a `node`, we return the integer that is currently attached to the given node; `val` in our case. Note that when matching a non-nullary constructor, its parameters are introduced in the form of variable bindings. Here, matching the `node` constructor also introduces three new variables, `val`, `left` and `right`.

```
1 (match tree (
2   (leaf 0)
3   ((node val left right) val)
4 ))
```

Listing 2.4: Pattern matching on tree-like data structure

Annotations

Every term in SMT-LIB can be annotated with several attributes to provide the underlying SMT solver with additional meta-information. For instance, we can name the first term in Listing 2.1 `foo` by wrapping it with the annotation expression along with the respective attribute. The annotated term then is `(! (< a b) :named foo)`. More generally, annotations are of the form `(! t a1 ... an)`, where `t` is a term and `a1` to `an` denote the `n` attributes attached.

Semantically speaking, annotating the term is equivalent to just the term `t` itself.

In practice, SMT solvers primarily use annotations to handle formulas involving quantifiers. In this approach, users define an instantiation pattern which is then consumed by the solver with the help of E-matching [35]. For a more thorough description, please refer to the work of Bjørner and de Moura [36].

2.1.4 Commands

After having presented both sorts and terms, we will now focus on the notion of commands. In SMT-LIB, a script always consists of a sequence of commands that are used to interact with the underlying SMT solver sequentially: we can declare new constants, functions and datatypes, introduce macros, restrict our model by using assertions, and also check the satisfiability of our C-like macros. In the following, we will briefly illustrate the most important features.

Declarations

The `(declare-sort s n)` command introduces a new uninterpreted sort `s` along with the respective arity `n`. For instance, `(declare-sort Tree 1)` introduces a generic tree type along with a single parameter denoting the value sort attached to a node. Thus, when dealing with a well-sorted integer Tree, its type must be `(Tree Int)`.

Similarly, `(declare-fun f (o1 ... on) o)` declares a function `f` with `n` arguments of respective sort `o1` to `on`; its return type is `o`. For example, `(declare-fun f ((Tree Int) Int) Int)` might declare a function that takes an integer tree and a position as its arguments and then returns the value at the specified node.

In SMT-LIB, constants are just functions without parameters. Thus, the command `(declare-const f o)` is just an abbreviation for `(declare-fun f () o)`.

With the addition of two new commands, SMT-LIB Version 2.6 now officially supports algebraic datatypes. `declare-datatype` declares a single, potentially recursive datatype. Equivalently, `declare-datatypes` allows the declaration of multiple, potentially mutually recursive datatypes. Because its behaviour is very similar to the simpler `declare-datatype`, we will omit it from the discussion. With algebraic datatypes, we not only have a convenient way of specifying data structures, but we can also denote enumeration types with ease.

```
1 (declare-datatype Day (Monday Tuesday Wednesday
    Thursday Friday Saturday Sunday))
```

Listing 2.5: Days-of-the-week enumeration type in SMT-LIB

Listing 2.5 shows how to define a simple days-of-the-week enumeration sort in SMT-LIB. The code listing specifies a new data structure called `Day` along with the seven data constructors `Monday`, `Tuesday`, `Wednesday`, and so on. The domain of enumeration sorts is always finite, i.e. the elements in the domain are the respective constructors.

In Listing 2.6, we show how a generic recursive binary tree is defined in SMT-LIB. In this case, the new parametric sort `Tree` has two data constructors `leaf` and `node`. However, in comparison to the enumeration type presented above, the `node` constructor now has three parameters itself; each argument having a dedicated selector, e.g. `value`, `left` and `right`. In contrast to Haskell, where pattern matching on data structures is implicit, SMT-LIB requires datatype selectors to be explicit. On the remaining lines, we declare a new integer tree `myTree` and then assert that its associated value must be `42` utilizing the `value` selector.

```
1 (declare-datatype Tree (par (T) ((leaf) (node (value T)
    (left (Tree T)) (right (Tree T))))))
2 (declare-const myTree (Tree Int))
3 (assert (= (value myTree) 42))
```

Listing 2.6: Recursive binary tree data structure

Assertions and Checking for Satisfiability

In the previous listings, we have already seen a few examples on how to add new assertions and then check the satisfiability of the specified model.

The `(assert t)` command adds a new assertion to our model, i.e. the formula `t` is required to hold in the current model. Please note that the term `t` must be well-sorted and of type `Bool`.

The `(check-sat)` command takes no arguments and directs the solver to check whether the conjunction of all assertions is satisfiable or not. It does this by trying to find a model that satisfies all asserted formulas. A return value of `sat` indicates that the solver found a model, `unsat` implies that the solver is sure that no model can exist and in all other cases, the solver will return `unknown`.

Sometimes, one does not only want to know whether a model exists but also retrieve the specific interpretation that the SMT solver found for that model. For this, we make use of the `(get-model)` command, which returns precisely one interpretation for all user-specified symbols in the current model.

Definitions

In addition to declarations, SMT-LIB also supports shorthand commands that make defining functions and interpreted sorts more accessible.

The `(define-sort s (p1 ... pn) o)` command introduces a new sort `s` with associated arity `n`. Semantically speaking, the type `(s o1 ... on)` is equivalent to the type obtained by substituting every occurrence of `p1` to `pn` with `o1` to `on` in `o`. Let us consider Listing 2.7, where we define a new generic array sort whose index type is always an integer. If we now, for instance, introduce a new symbol of the sort `(MyArray Bool)`, its concrete type will be `(Array Int Bool)`, which ensures that subsequent type checks will succeed.

```
1 (define-sort MyArray (T) (Array Int T))
```

Listing 2.7: Generic array definition in SMT-LIB

SMT-LIB also offers two additional commands for defining functions directly. The first one, **define-fun**, can be used to declare and define functions that are not (mutually) recursive. Semantically, **define-fun** is equivalent to a function declaration followed by an assertion quantifying over all function parameters. For example, the function definition `(define-fun plus ((a Int) (b Int) ) Int (+ a b))` is semantically equivalent to the two commands shown in Listing 2.8. Similarly, **define-funs-rec** can be used to define multiple, mutually recursive functions in parallel.

```
1 (declare-fun plus (Int Int) Int)
2 (assert (forall ((a Int) (b Int)) (= (plus a b) (+ a b)
     )))
```

Listing 2.8: Equivalent function definition in SMT-LIB

Scope

SMT-LIB proposes a stack-based execution model, meaning that all assertions and declarations only exist on the stack. Multiple assertions and declarations on the stack are combined into a stack level, which forms an atomic unit. Thus, it is not possible to remove individual elements from the stack without removing the entire level.

When a new assertion or declaration is introduced, it automatically belongs to the current level. The `(push n)` command pushes `n` empty levels onto the stack. If `n` is not specified, only a single new level is created. Likewise, `(pop n)` removes the `n` most recent levels from the stack. Removing a single level invalidates all assertions and symbol declarations that were introduced

on that specific level. By default, the stack starts with an initial level. All assertions and declarations are added to this level as long as no **`push`** command has occurred yet.

Executing the (**`reset-assertions`**) command removes all levels, including its assertions, from the stack.

Miscellaneous Commands

In addition to the ones introduced above, SMT-LIB features a range of other commands that can be used to interact with the underlying SMT solvers. With these commands, one can (re-)start and terminate the solver, enable specific logics, adjust the solver options and inspect the current model more closely. However, we will not explain these commands further as they offer no additional benefit for the understanding of this thesis.

2.2 The Z3 Theorem Prover

There exist numerous SMT solvers that support the SMT-LIB standard described previously, including CVC4 [6], MathSAT [9], OpenSMT [10], Yices [13] and Z3 [36]. Among one of the most popular SMT solvers is the Z3 Theorem Prover developed by Microsoft. Z3 is widely used in research and industry and forms the basis of many program verifiers, including Boogie [4], F* [39] and the Viper verification infrastructure mentioned earlier.

While Z3 officially adopts the SMT-LIB Version 2.0 standard, its implementation is a superset of SMT-LIB Version 2.6, i.e. Z3's accepted language syntax is less restrictive than the official specification for the most part. Our work is only concerned with supporting the SMT-LIB specification that Z3 accepts; other SMT-LIB dialects will be ignored. In the following section, we will show a brief overview of the most important differences between the two.

2.2.1 Differences to SMT-LIB

There are only a few differences between SMT-LIB and the language Z3 accepts. For instance, Z3 introduces the **`define-const`** command, which is a shorthand for a constant declaration along with an assertion. Furthermore, the **`declare-datatype`** command is extended such that it accepts both the SMT-LIB2.0 and SMT-LIB2.6 standard, which differ slightly.

2.2.2 The Language Grammar

In Figure 2.1, we present the grammar of the SMT-LIB dialect that is supported by Z3. Because there is no official documentation on the SMT-LIB dialect supported by Z3 and its grammar is not described publicly, we had

to formalize the grammar shown in Figure 2.1 by examining the lexer and parser of the Z3 project [36]. At the time of formalizing, version 4.8.8 was the latest stable release.

The grammar shown in 2.1 uses | to separate alternatives. The suffix ? indicates that the preceding component is optional, while the suffixes $\star$ and + specify a zero-or-more and one-or-more repetition of the previous component, respectively.

⟨*spec_constant*⟩ ::= ⟨*numeral*⟩ | ⟨*decimal*⟩ | ⟨*hexadecimal*⟩ | ⟨*binary*⟩ | ⟨*string*⟩

⟨*s_expr*⟩ ::= ⟨*spec_constant*⟩ | ⟨*all_symbols*⟩ | ⟨*keyword*⟩
| '(' ⟨*s_expr*⟩⋆ ')'

⟨*identifer*⟩ ::= ⟨*symbol*⟩ | '(_' ⟨*symbol*⟩ ⟨*numeral*⟩+ ')'

⟨*sort*⟩ ::= ⟨*identifer*⟩ | '(' ⟨*identifier*⟩ ⟨*sort*⟩+ ')'

⟨*attribute_value*⟩ ::= ⟨*spec_constant*⟩ | ⟨*symbol*⟩ | '(' ⟨*s_expr*⟩⋆ ')'

⟨*attribute*⟩ ::= ⟨*keyword*⟩ ⟨*attribute_value*⟩?

⟨*casted_identifer*⟩ ::= ⟨*identifier*⟩ | '(as' ⟨*identifier*⟩ ⟨*sort*⟩ ')'

⟨*var_binding*⟩ ::= '(' ⟨*symbol*⟩ ⟨*term*⟩ ')'

⟨*pattern*⟩ ::= ⟨*symbol*⟩ | '(' ⟨*symbol*⟩ ⟨*symbol*⟩+ ')'

⟨*match_case*⟩ ::= '(' ⟨*pattern*⟩ ⟨*term*⟩ ')'

⟨*term*⟩ ::= ⟨*spec_constant*⟩
| ⟨*casted_identifer*⟩
| '(' ⟨*casted_identifer*⟩ ⟨*term*⟩+ ')'
| '(let (' ⟨*var_binding*⟩+ ')' ⟨*term*⟩ ')'
| '(forall (' ⟨*sorted_var*⟩+ ')' ⟨*term*⟩ ')'
| '(exists (' ⟨*sorted_var*⟩+ ')' ⟨*term*⟩ ')'
| '(lambda (' ⟨*sorted_var*⟩+ ')' ⟨*term*⟩ ')'
| '(match' ⟨*term*⟩ '(' ⟨*match_case*⟩+ '))'
| '(!' ⟨*term*⟩ ⟨*attribute*⟩⋆ ')'

⟨*constructor_dec*⟩ ::= ⟨*symbol*⟩ | '(' ⟨*symbol*⟩ ⟨*sorted_var*⟩⋆ ')'

⟨*datatype_dec*⟩ ::= '(' ⟨*constructor_dec*⟩+ ')'
| '(par (' ⟨*symbol*⟩+ ') (' ⟨*constructor_dec*⟩+ '))'

⟨*datatypes_dec*⟩ ::= '(' ⟨*symbol*⟩ ⟨*constructor_dec*⟩+ ')'

⟨*smt_26_or_old*⟩ ::= ⟨*smt_26*⟩ | ⟨*smt_old*⟩

⟨*smt_old*⟩ ::= '(' ⟨*symbol*⟩⋆ ') (' ⟨*datatypes_dec*⟩⋆ ')'

⟨*smt_26*⟩ ::= '(' ⟨*idx_identifier*⟩+ ') (' ⟨*datatype_dec*⟩+ ')'

⟨*idx_identifier*⟩ ::= '(' ⟨*symbol*⟩ ⟨*numeral*⟩ ')'

⟨*function_dec*⟩ ::= '(' ⟨*symbol*⟩ '(' ⟨*sorted_var*⟩⋆ ')' ⟨*sort*⟩ ')'

⟨*function_declar*⟩ ::= ⟨*symbol*⟩ '(' ⟨*sorted_var*⟩⋆ ')' ⟨*sort*⟩ ⟨*term*⟩

⟨*command*⟩ ::= '`assert`' ⟨*term*⟩
| '`check-sat`' ⟨*term*⟩⋆
| '`check-sat-assuming (`' ⟨*term*⟩⋆ '`)`'
| '`declare-const`' ⟨*symbol*⟩ ⟨*sort*⟩
| '`declare-datatype`' ⟨*symbol*⟩ ⟨*datatype_dec*⟩
| '`declare-datatypes`' ⟨*smt_26_or_old*⟩
| '`declare-fun`' ⟨*symbol*⟩ '`(`' ⟨*sort*⟩⋆ '`)`' ⟨*sort*⟩
| '`declare-sort`' ⟨*symbol*⟩ ⟨*numeral*⟩?
| '`define-fun`' ⟨*function_declar*⟩
| '`define-fun-rec`' ⟨*function_declar*⟩
| '`define-funs-rec (`' ⟨*function_dec*⟩+ '`) (`' ⟨*term*⟩+ '`)`'
| '`define-sort`' ⟨*symbol*⟩ '`(`' ⟨*symbol*⟩⋆ '`)`' ⟨*sort*⟩
| '`define-const`' ⟨*symbol*⟩ ⟨*sort*⟩ ⟨*term*⟩
| '`echo`' ⟨*string*⟩
| '`exit`'
| '`get-assertions`'
| '`get-assignment`'
| '`get-info`' ⟨*keyword*⟩
| '`get-model`'
| '`get-option`' ⟨*keyword*⟩
| '`get-proof`'
| '`get-unsat-assumptions`'
| '`get-unsat-core`'
| '`get-value (`' ⟨*term*⟩+ '`)`'
| '`pop`' ⟨*numeral*⟩?
| '`push`' ⟨*numeral*⟩?
| '`reset`'
| '`reset-assertions`'
| '`set-info`' ⟨*attribute*⟩
| '`set-logic`' ⟨*symbol*⟩
| '`set-option`' ⟨*attribute*⟩

⟨*cmd*⟩ ::= '`(`' ⟨*command*⟩ '`)`'

⟨*script*⟩ ::= ⟨*cmd*⟩⋆

Figure 2.1: The SMT-LIB dialect supported by Z3

Chapter 3

The B-SMT Language

In this chapter we propose a novel SMT language, which we call B-SMT, with a more human-friendly syntax, drawing inspiration from existing languages such as Haskell, OCaml [27], Rust, Boogie, and Viper. In the next chapter, we will show how to extend the language, both syntactically and semantically, to make it more expressive. From now on, we will denote Z3's SMT-LIB dialect as SMT-LIB for better readability.

3.1 Restrictions in Language Design

Since our new language must support efficient conversion both from and to SMT-LIB, certain restrictions in language design are imposed on us. In the following, we will briefly explore them.

3.1.1 SMT-LIB Symbol Definitions

In contrast to many traditional programming languages, SMT-LIB's symbol definition is mostly unrestricted. Valid symbols can include many special characters, including any of the following characters `, ~ + = < > . ? / - |`. Such an extensive symbol definition is problematic since many of these characters have assigned a special meaning to them in other languages. For instance, consider the following function declaration `fun isValid(param1, param2);` a syntax that most modern programming languages adhere to in some form. However, in SMT-LIB, `param1,param2` will be a valid symbol token, and thus this declaration will be identified as a function with only a single argument.

To circumvent this issue, and allow for more freedom in the design of B-SMT, we have decided to restrict the symbol definition not to contain any commas. By doing so, we can achieve a more modern and better human-readable source syntax.

3.1.2 Reserved Keywords

As we have seen in the previous chapter, everything in SMT-LIB is an S-expression. As a consequence, in contrast to modern programming languages, SMT-LIB does not define any reserved keywords. For example, the command `(declare-const forall Int)`, while seemingly obscure, is perfectly valid in SMT-LIB because Z3 can infer its meaning based on the context. However, when trying to use this newly introduced constant within a term, Z3 throws an error.

Such a mechanism, where the context decides whether a symbol is valid or not, is overly confusing for developers. In order to prevent the introduction of an erratic symbol handling for our new B-SMT language, we decided to make all symbols leading to inconsistent behaviour reserved keywords, i.e. they will not be allowed to appear as symbols in B-SMT. Besides, we introduce two additional reserved symbols, namely `with` and `->`. This decision grants us extra leeway during the language design process. In Listing 3.1, all reserved symbols are shown.

```
as
let
forall
exists
lambda
match
!
_
par
with
->
```

Listing 3.1: Reserved keywords in B-SMT

3.1.3 Two-Way Transpilation

Moreover, different from most other projects involving code transpilation, we need to be able to transform back and forth between SMT-LIB and B-SMT. As a result B-SMT closely resembles SMT-LIB as introduced in the previous chapter.

3.2 Design Goals

The design of B-SMT is based on the assumption that the intermediate language is imperative. For the development of the B-SMT language, we focus on achieving two primary goals. First of all, our new source syntax should be easier to read by tool developers that have to reason about resulting SMT-LIB

encodings produced by their verifiers. We accomplish this by reducing the encoding gap between B-SMT and the intermediate verification language; in other words, we try to mimic syntax and semantics of higher-level programming languages.

The second major goal is concerned with performance. The language should be reasonably easy to lex and parse and thus allow us to process SMT encodings with tens of thousands of lines quickly. For this reason, it should be possible to parse the language grammar in linear time without applying any backtracking techniques.

3.3 An Overview of the B-SMT Language

In this section, we will briefly present the most critical aspects of the B-SMT language. Please note that all grammar rules presented in this section follow the same notation as introduced in the previous chapter. In addition to the aforementioned notation, ,⋆ and ,+ indicate a zero-or-more and one-or-more repetition of the previous component, respectively. The |⋆ and |+ annotations are defined in the same manner.

The B-SMT language features an LL(3) grammar. Therefore, the language itself is context-free and can be parsed by a recursive descent parser without the need for any backtracking. Parsing an LL(3) grammar requires a lookahead of at most three tokens, which implies that we can parse B-SMT within linear time.

Our new SMT encoding abandons the Lisp-style S-expressions and instead replaces them with a mixture of infix notation for operators, function application by encompassing its arguments in parentheses and ultimately on the global scope, statements. Unless stated otherwise, the semantics of B-SMT are equivalent to the respective SMT-LIB language concepts explained in the previous chapter.

Furthermore, while the B-SMT language is strongly typed, we will not carry out any type checks during the parsing phase. Thus, if the source program (e.g. SMT-LIB or B-SMT) is not well-sorted, it will still be transpiled into the respective language, carrying over all erroneous typings.

3.3.1 Tokens

In Figure 3.1, the token definitions for our novel B-SMT language are shown. Apart from the restrictions introduced in Section 3.1, the token definitions for B-SMT are equivalent to those of SMT-LIB.

⟨*symbol_char*⟩ ::= 'a' | 'b' | 'c' | ... | 'x' | 'y' | 'z' |
| 'A' | 'B' | 'C' | ... | 'X' | 'Y' | 'Z' |
| '~' | '!' | '@' | '$' | '%' | '^' | '&' | '*' | '_'
| '+' | '=' | '<' | '>' | '.' | '?' | '/' | '-'

⟨*digit*⟩ ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

⟨*zero_digit*⟩ ::= ⟨*digit*⟩ | '0'

⟨*numeral*⟩ ::= '0' | ⟨*digit*⟩⟨*zero_digit*⟩★

⟨*decimal*⟩ ::= ⟨*numeral*⟩'.'(⟨*zero_digit*⟩+)?

⟨*binary*⟩ ::= '#b' ('0' | '1')+

⟨*hex_chars*⟩ ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'

⟨*hexadecimal*⟩ ::= '#x' (⟨*zero_digit*⟩ | ⟨*hex_chars*⟩)+

⟨*string*⟩ ::= '"'⟨*any_non_linebreak_char*⟩★'"'

⟨*quoted_symbol*⟩ ::= '|'⟨*any_non_linebreak_char*⟩★'|'

⟨*simple_symbol*⟩ ::= ⟨*symbol_char*⟩(⟨*zero_digit*⟩ | ⟨*symbol_char*⟩)★

⟨*symbol*⟩ ::= ⟨*simple_symbol*⟩ | ⟨*quoted_symbol*⟩

⟨*keyword*⟩ ::= ':'(⟨*zero_digit*⟩ | ⟨*symbol_char*⟩)+

Figure 3.1: B-SMT token definitions language grammar

3.3.2 S-expressions

⟨*spec_constant*⟩ ::= ⟨*numeral*⟩ | ⟨*decimal*⟩ | ⟨*hexadecimal*⟩ | ⟨*binary*⟩ | ⟨*string*⟩

⟨*s_expr*⟩ ::= ⟨*spec_constant*⟩ | ⟨*symbol*⟩ | ⟨*keyword*⟩ | '(' ⟨*s_expr*⟩★ ')'

Figure 3.2: B-SMT s-expressions language grammar

3.3.3 Identifiers and Sorts

⟨*indexed_iden*⟩ ::= '{' ⟨*numeral*⟩,+ '}'
⟨*identifier*⟩ ::= ⟨*symbol*⟩⟨*indexed_iden*⟩?
⟨*sort_generics*⟩ ::= '[' ⟨*sort*⟩,+ ']'
⟨*sort*⟩ ::= ⟨*identifier*⟩⟨*sort_generics*⟩?
⟨*sorted_var*⟩ ::= ⟨*symbol*⟩':' ⟨*sort*⟩

Figure 3.3: B-SMT identifiers and sorts language grammar

3.3.4 Attributes

⟨*attribute_value*⟩ ::= ⟨*spec_constant*⟩ | ⟨*symbol*⟩ | '(' ⟨*s_expr*⟩⋆ ')'
⟨*attribute*⟩ ::= ⟨*keyword*⟩ ⟨*attribute_value*⟩?

Figure 3.4: B-SMT attributes language grammar

3.3.5 Declarations

Sort Constructors

The sort constructor is semantically equivalent to the **`declare-sort`** command from SMT-LIB. It declares a new uninterpreted sort with a default arity of 0, if unspecified. Transforming between B-SMT and SMT-LIB is trivial in this case and hence omitted. In Figure 3.6, the sort constructor grammar is shown.

⟨*arity_attr*⟩ ::= '{:arity' ⟨*numeral*⟩ '}'
⟨*sort_decl*⟩ ::= 'sort' ⟨*symbol*⟩ ⟨*arity_attr*⟩?

Figure 3.5: B-SMT datatype constructors language grammar

Datatypes

SMT-LIB offers various commands to introduce new datatypes. In B-SMT, we combine them into a single statement, offering a more concise representation while at the same time conserving the semantics.

⟨*generics*⟩ ::= '[' ⟨*symbol*⟩,+ ']'
⟨*selectors*⟩ ::= '(' ⟨*sorted_var*⟩,+ ')'
⟨*constr_decl*⟩ ::= ⟨*symbol*⟩⟨*selectors*⟩?
⟨*datatype_decl*⟩ ::= 'data' ⟨*symbol*⟩⟨*generics*⟩? ':=' ⟨*constr_decl*⟩:|+

Figure 3.6: B-SMT datatype constructors language grammar

Constants

⟨*const_decl*⟩ ::= const ⟨*sorted_var*⟩

Figure 3.7: B-SMT constant declarations language grammar

Functions

SMT-LIB offers various commands to introduce both recursive and non-recursive functions. In B-SMT, we perform an analysis on the AST to retrieve the original commands.

⟨*func_decl*⟩ ::= 'fun' ⟨*symbol*⟩ '(' ⟨*sort*⟩,⋆ '):' ⟨*sort*⟩

Figure 3.8: B-SMT function declarations language grammar

3.3.6 Terms

The semantics of the variable binding operator `let` differs slightly between SMT-LIB and B-SMT. The variable bindings introduced in B-SMT are allowed to be mutually recursive, i.e. the newly defined variables can reference each other. To achieve the same outcome in SMT-LIB, one has to introduce multiple nested variable bindings instead.

⟨*casted_identifier*⟩ ::= ⟨*identifier*⟩ | '(' ⟨*identifier*⟩ 'as' ⟨*sort*⟩ ')'
⟨*var_binding*⟩ ::= ⟨*symbol*⟩ ':=' ⟨*term*⟩
⟨*quant_binding*⟩ ::= ⟨*sorted_var*⟩,+ '::' ⟨*term*⟩
⟨*pattern_param*⟩ ::= '(' ⟨*symbol*⟩,+ ')'
⟨*pattern*⟩ ::= ⟨*symbol*⟩⟨*pattern_param*⟩?
⟨*match_case*⟩ ::= ':|' ⟨*pattern*⟩ '->' ⟨*term*⟩
⟨*annotation*⟩ ::= '{' ⟨*attribute*⟩,+ '}'
⟨*term_opt*⟩ ::= ⟨*spec_constant*⟩
| ⟨*casted_identifier*⟩ '(' ⟨*term*⟩,+ ')'
| ⟨*casted_identifier*⟩
| 'let' ⟨*var_binding*⟩,+ 'with' ⟨*term*⟩
| 'forall' ⟨*quant_binding*⟩
| 'exists' ⟨*quant_binding*⟩
| 'lambda' ⟨*sorted_var*⟩,+ '->' ⟨*term*⟩
| 'match' ⟨*term*⟩ 'with' ⟨*match_case*⟩+
⟨*term*⟩ ::= ⟨*annotation*⟩?⟨*term_opt*⟩

Figure 3.9: B-SMT terms language grammar

3.3.7 Macros

Sort Synonyms

⟨*sort_macro*⟩ ::= 'let' ⟨*symbol*⟩⟨*generics*⟩? ':=' ⟨*sort*⟩

Figure 3.10: B-SMT sort synonyms language grammar

Constants

⟨*const_macro*⟩ ::= 'let' ⟨*sorted_var*⟩ ':=' ⟨*term*⟩

Figure 3.11: B-SMT constants language grammar

Functions

⟨*func_macro*⟩ ::= 'let' ⟨*symbol*⟩ '(' ⟨*sorted_var*⟩,⋆ '):' ⟨*sort*⟩ ':=' ⟨*term*⟩

Figure 3.12: B-SMT functions language grammar

3.3.8 Statements

Assumptions

⟨*assume_cmd*⟩ ::= 'assume' ⟨*term*⟩

Figure 3.13: B-SMT assumptions language grammar

Inspections

⟨*inspections*⟩ ::= 'get-assertions'
| 'get-assignment'
| 'get-model'
| 'get-proof'
| 'get-unsat-assumptions'
| 'get-unsat-core'
| 'get-info {' ⟨*keyword*⟩ '}'
| 'get-option {' ⟨*keyword*⟩ '}'
| 'get-value' ⟨*term*⟩,+

Figure 3.14: B-SMT inspections language grammar

Satisfiability Checks

⟨*sat*⟩ ::= 'check-sat'
⟨*sat_assuming*⟩ ::= 'check-sat-assuming' ⟨*term*⟩,+

Figure 3.15: B-SMT satisfiability checks language grammar

Modifying Assertion Stack

⟨*push_cmd*⟩ ::= 'push' ⟨*numeral*⟩?
⟨*pop_cmd*⟩ ::= 'pop' ⟨*numeral*⟩?
⟨*reset_assertions_cmd*⟩ ::= 'reset-assertions'

Figure 3.16: B-SMT modifying assertion stack language grammar

(Re)starting and Terminating

⟨*reset_cmd*⟩ ::= '`reset`'
⟨*exit_cmd*⟩ ::= '`exit`'

Figure 3.17: B-SMT restarting & terminating language grammar

Script information

⟨*set_option_cmd*⟩ ::= '`set-option {`' ⟨*attribute*⟩ '`}`'
⟨*set_logic_cmd*⟩ ::= '`set-logic`' ⟨*symbol*⟩
⟨*set_info_cmd*⟩ ::= '`set-info {`' ⟨*attribute*⟩ '`}`'
⟨*echo_cmd*⟩ ::= '`echo`' ⟨*string*⟩

Figure 3.18: B-SMT script information language grammar

3.3.9 Program

⟨*command*⟩ ::= ⟨*sort_decl*⟩
| ⟨*sort_macro*⟩
| ⟨*datatype_decl*⟩
| ⟨*const_decl*⟩
| ⟨*const_macro*⟩
| ⟨*func_decl*⟩
| ⟨*func_macro*⟩
| ⟨*assume_cmd*⟩
| ⟨*inspections*⟩
| ⟨*sat*⟩
| ⟨*sat_assuming*⟩
| ⟨*set_info_cmd*⟩
| ⟨*echo_cmd*⟩
| ⟨*set_logic_cmd*⟩
| ⟨*set_option_cmd*⟩
| ⟨*exit_cmd*⟩
| ⟨*reset_cmd*⟩
| ⟨*reset_assertions_cmd*⟩
| ⟨*pop_cmd*⟩
| ⟨*push_cmd*⟩

⟨*statement*⟩ ::= ⟨*command*⟩ ';'?

⟨*program*⟩ ::= ⟨*statement*⟩⋆

Figure 3.19: B-SMT program language grammar

Chapter 4

Language Extensions

After having seen the core features of our new B-SMT language, we will now explore prospective language customizations that allow us to introduce both custom syntax and behaviour within B-SMT. Allowing users to define their custom language extensions is particularly useful for tool developers to support verifier-specific concepts with a human-friendly syntax in B-SMT. Defining a custom syntax will not only improve the readability of the resulting code but might also significantly decrease the encoding size.

Every program verifier encodes its verification conditions slightly differently. Hence, making assumptions on the specific form an SMT-LIB encoding adheres to might lead to erroneous behaviour if the premises are not satisfied. As a result, language extensions in B-SMT are not part of the core language specification and thus, are not required to behave correctly in all use cases. To mitigate this issue, users are allowed to enable or disable specific extensions.

In this section, we present a list of potential language extensions that were identified by analyzing real-world SMT encodings. In this chapter, we will refer to the core B-SMT language, excluding all extensions, with B-SMT. The extended B-SMT language will be denoted as E-BSMT in this section.

Over the course of this thesis, we have looked at various extendable programming languages, including Racket [38] and Grace [7]. However, their extension model could not accommodate the complex transformations that have to be performed on the B-SMT abstract syntax tree. Hence, we have decided to use a more general approach, by directly defining the transformations on the AST with a scripting language.

4.1 Assertions

In B-SMT, we adapt a different terminology for assumptions and assertions that matches the meaning of the same concepts in many intermediate ver-

ification languages, including Viper and Boogie. In these languages, the assumption statement adds the specified Boolean formula to the set of model constraints. In contrast, the assertion statement is coupled with a proof obligation, i.e. the solver has to prove that the given property always holds. This proof is usually achieved by assuming the negation of the given property along with a subsequent SMT solver check for satisfiability. The assertion holds whenever the underlying solver returns `unsat`. Thus, an SMT-LIB assertion is denoted as an assumption on the B-SMT level.

However, the core B-SMT language does not offer a corresponding assertion statement, which is the subject of our first language extension. For this, we introduce a new **`assert`** command. In general, its syntax is of the form **`assert`** `t`, where `t` is a Boolean formula. On the B-SMT level, this statement translates to the command sequence shown in Listing 4.1.

```
1 assume not(t)
2 check-sat
```

Listing 4.1: Transpiling E-BSMT assertion into B-SMT

Transpiling from B-SMT to E-BSMT is a bit more challenging and involves pattern matching on the B-SMT source code. Every B-SMT assumption that wraps a term `t` within the negation operator `not` and is followed by zero or more comments and ultimately a corresponding **`check-sat`** command will be transpiled into **`assert`** `t` on the E-BSMT level. All comments that occurred between the two statements will be attached after **`assert`** `t`. For example, let us consider Listing 4.1, where a B-SMT snippet is shown on the left-hand side. After applying the transformation explained above, we end up with the resulting E-BSMT code presented on the right-hand side.

```
1 assume not(=(y1, g(#x01)))
2 ' this is an example comment
3 ' this is another comment
4 check-sat
```

```
1 assert =(y1, g(#x01))
2 ' this is an example comment
3 ' this is another comment
```

Figure 4.1: Example transformation for the assertion extension

4.2 Infix & Mixfix Operators

In the B-SMT standard, there is no notion of an operator similar to what we know from traditional programming languages. Instead, operators are provided in the form of a conventional function in B-SMT. However, for specific operators such as +, =, ==>, the function application syntax is unusual and not very convenient. To circumvent this issue, we define an infix and

mixfix operator language extension that proposes a more convenient syntax for all operators defined in the various SMT-LIB Theories [24]. Now, let us consider Listing 4.2, which shows several terms in the core B-SMT language. By leveraging the infix and mixfix notation for our operators as defined above, the encoding shown in Listing 4.3 is generated; offering a much more concise and modern way of expressing the respective terms.

```
1 <=(s1, s2)
2 ite(<(c, 0), false, ite(=(c, 0), true, check(2)))
3 ite(>=(x, y), x, y)
4 ite(y, ping(+(x, 1), pong(x, x)), x)
5 =(a, z)
6 (<=(0, a)) and (<(a ,1032))
```

Listing 4.2: Standard B-SMT terms

```
1 s1 <= s2
2 c < 0 ? false : (c = 0 ? true : check(2))
3 x >= y ? x : y
4 y ? ping(x + 1, pong(x, x)) : x
5 a = z
6 0 <= a && a < 1032
```

Listing 4.3: Resulting E-BSMT terms

By introducing a modern operator notation into E-BSMT, we usually discard redundant parentheses, making the syntax more human-friendly. By specifying precedence for every operator, we can then use this knowledge to infer an unambiguous parse tree; making many parentheses obsolete.

4.3 Piecewise Constants

In this section, we will extend B-SMT with a language construct that allows users to define piecewise constants conveniently. For this, let us consider Listing 4.4, which shows our proposed E-BSMT syntax for defining a constant piecewise. In Listing 4.5, we suggest how a constant definition looks on the B-SMT level. Note that all B-SMT code shown in the remaining sections of this chapter assumes an infix notation for operators as introduced in Section 4.2.

```
1 const x: Int
2 x :=
3   :| cond = 12 -> 35
4   :| cond = 55 -> 11
5   :| otherwise -> unknown
```

Listing 4.4: E-BSMT: piecewise constant definition

```
1 const x: Int
2 assume cond = 12 ==> x = 35
3 assume cond = 55 ==> x = 11
```

Listing 4.5: B-SMT: piecewise constant definition

In comparison to Listing 4.5, the notation introduced in Listing 4.4 is much more intuitive. Note that in our new syntax, we utilize an additional piecewise definition case, which introduces two new keywords **otherwise** and unknown. With these, we state that for all cases not covered, we do not hold any information about the constant. This new syntax makes it explicit to the user that all functions and variables in SMT-LIB (and B-SMT respectively) are total.

The transformation step that goes from E-BSMT to B-SMT is relatively simple. However, the transformation into the opposite direction is much more involved and exposes a few issues to consider. We will now examine both transformation steps and the respective problems in more detail.

4.3.1 E-BSMT to B-SMT

In the following, for reasons of simplicity, we will only show the transformation with a single piecewise definition case. It is trivial to generalize the approach outlined below to multiple cases by applying the transformation step on each definition case, leading to multiple assumption statements instead of one. In its general form, the piecewise constant definition has the structure as shown in Listing 4.6. The notation <N: t> matches an element of type t and assigns the unique identifier N to it. In Listing 4.7, the resulting B-SMT code is shown.

```
<NAME: Symbol> :=
  :| <CONDITION: Term> -> <ASSIGNMENT: Term>
  :| otherwise -> unknown
```

Listing 4.6: E-BSMT: constant piecewise definition matching

```
assume <CONDITION> ==> <NAME> = <ASSIGNMENT>
```

Listing 4.7: B-SMT: constant piecewise definition transformation

4.3.2 B-SMT to E-BSMT

As before, we will only explain the transformation step for a single piecewise definition case. However, in contrast to the simple transformation described above, transforming from B-SMT to E-BSMT is more complicated, as B-SMT

offers different ways to express a piecewise constant definition. In Listing 4.8, we show a non-exhaustive list of all B-SMT statements that are identified as a piecewise definition. The first matching shown in Figure 4.8 does not produce an explicit `ELSE` term; resulting in an implicit assignment to the `unknown` keyword. In Listing 4.9, the corresponding E-BSMT code is illustrated.

```
assume <CONDITION: Term> ==>
  <NAME: Symbol> = <THEN: Term>
```

```
assume <CONDITION: Term>
  ? <NAME: Symbol> = <THEN: Term> : <NAME> = <ELSE: Term>
```

Listing 4.8: B-SMT: constant piecewise definition matchings

```
<NAME> :=
  :| <CONDITION> -> <THEN>
  :| otherwise -> <ELSE>
```

Listing 4.9: E-BSMT: constant piecewise definition transformation

4.3.3 Transformation Issues

During the transformation step from B-SMT to E-BSMT, we are faced with further challenges, which are briefly explored in the following.

Scope

Due to the stack-based execution model of SMT-LIB (and B-SMT respectively), we need to be cautious about which piecewise definition cases are allowed to be grouped together and where the resulting definition should be located. For this, consider the modified B-SMT example from Listing 4.5 shown in Listing 4.10. In this case, the **push** and **pop** statements alter the assertion stack, and as a result, the statements (1) and (2) cannot be grouped together into a piecewise definition, as shown in Listing 4.4. Doing so would result in a non-equivalent transformation, as either (1) or (2) will be held on the assertion stack when it should not. We can circumvent this issue by only grouping together assumptions that are defined in the same scope, i.e. on the same assertion stack level.

```
1 const x: Int
2 push
3 assume cond = 12 ==> x = 35  '(1)
4 pop
5 assume cond = 55 ==> x = 11  '(2)
```

Listing 4.10: Grouping piecewise definition cases with scope

In addition to **push** and **pop**, there are other statements that also trigger a similar behaviour, including **check-sat**, **check-sat-assuming**, **reset** and **reset-assertions**. The issues introduced by these statements, and how to resolve them, are very similar to the one we have presented above. Thus, we will not cover them in more detail in this thesis.

Exhaustive Matchings

In Section 4.3.2, we have seen a non-exhaustive list of all B-SMT statements that are identified as a piecewise definition. However, it is not clear how many other matchings possibly exist. For instance, consider the first matching from Listing 4.8. By swapping the two operands of =, we end up with a permutation of the original matching i.e. our current transformation step would not be able to correctly identify this matching case.

Although this might lead to the situation that certain piecewise constant definitions are being missed, it is typically not a problem, as language extensions are not part of the core language specification and thus, are not required to behave correctly in all use cases. Furthermore, by allowing implementers to specify their custom matchings with a configuration file, erroneous behaviour is reduced to a minimum.

4.4 Piecewise Functions

In the next step, we will explore how to adapt and extend the above definition for piecewise immutable functions. Because function definitions are usually expressed with the help of quantifiers, they are often annotated with triggers and unique identifiers to help the underlying SMT solver. In this section, annotations are ignored in favour of a more readable source.

Now, let us consider Listing 4.11, which shows our proposed E-BSMT syntax for defining a piecewise function. In Listing 4.12, we present how such a piecewise function definition looks on the B-SMT level.

```
1 fun map(Int): Int
2
3 map(r) :=
4   :| r % 2 -> 42
5   :| r = 13 -> aFun(r)
6   :| otherwise -> unknown
```

Listing 4.11: E-BSMT: piecewise function definition

```
1 fun map(Int): Int
2
3 assume forall r: Int :: r % 2 ===> map(r) = 42
4 assume forall r: Int :: r = 13 ===> map(r) = aFun(r)
```

Listing 4.12: B-SMT: piecewise function definition

Similarly, as before, we will now show the two transformation steps. All the issues and the respective solutions that were covered in the previous section still apply. In the following, for reasons of simplicity, we will only show the transformation step for functions with a single argument; lifting our findings to support multiple arguments is trivial.

4.4.1 E-BSMT to B-SMT

This transformation step is analogous to the one presented in Section 4.3.1. In its general form, the piecewise function definition has the structure shown in Listing 4.13. To arrive at a valid matching, the symbols `NAME1` and `NAME2` must be equal. If this is not the case, the matching is ignored. In Listing 4.14, the resulting B-SMT code is illustrated.

```
fun <NAME1: Symbol>(<SORT: Sort>): <RETURN_SORT: Sort>

<NAME2: Symbol>(<ARG: Symbol>) :=
  :| <CONDITION: Term> -> <ASSIGNMENT: Term>
  :| otherwise -> unknown
```

Listing 4.13: E-BSMT: piecewise function definition matching

```
assume forall <ARG>: <SORT> ::
  <CONDITION> ==> <NAME1>(<ARG>) = <ASSIGNMENT>
```

Listing 4.14: B-SMT: piecewise function definition transformation

4.4.2 B-SMT to E-BSMT

In order to support the transformation from B-SMT to E-BSMT, we can lift the cases presented in Listing 4.8 to support piecewise function definitions. For reasons of simplicity, we will only adapt the first matching case; the second one can be lifted accordingly.

In Listing 4.15, we show the adapted matching for immutable functions. In this case, the symbols `ARG1` and `ARG2` have to be identical. Otherwise, the matching will be ignored as it does not represent a valid piecewise function definition. The resulting E-BSMT code is illustrated in Listing 4.16.

```
assume forall <ARG1: Symbol>: <SORT: Sort> ::
  <CONDITION: Term> ==>
    <NAME: Symbol>(<ARG2: Symbol>) = <ASSIGNMENT: Term>
```

Listing 4.15: B-SMT: piecewise function definition matching

```
<NAME>(<ARG1>) :=
  :| <CONDITION> -> <ASSIGNMENT>
  :| otherwise -> unknown
```

Listing 4.16: E-BSMT: piecewise function definition transformation

4.5 Mutable Variables

After having presented immutable piecewise functions and constants, we now extend on this concept by proposing the idea of mutability. To achieve this, we introduce a new reserved keyword **`mut`** in E-BSMT along with the term `old(x)`, which allows referring to `x` from the previous state. The syntax of the `old` expression is inspired by Viper.

Now, let us consider Listing 4.17, which shows our proposed E-BSMT syntax for defining a mutable variable. As shown, we can declare a mutable variable with the **`mut`** modifier. Although the notion of a mutable constant is flawed, we will stick to the naming convention to prevent the introduction of another reserved keyword. In Listing 4.18, we suggest how such a definition might look on the B-SMT level.

```
1 mut const x: Int
2
3 x :=
4   :| cond = 12 -> 35
5   :| cond = 55 -> 11
6
7 x :=
8   :| otherCond = 15 -> 44
9   :| otherwise -> old(x)
```

Listing 4.17: E-BSMT: mutable variable definition

```
1 const x@1: Int
2 assume cond = 12 ==> x@1 = 35
3 assume cond = 55 ==> x@1 = 11
4
5 const x@2: Int
6 assume otherCond = 15 ? x@2 = 44 : x@2 = x@1
```

Listing 4.18: B-SMT: mutable variable definition

All transformations that have been introduced in the previous section still hold and do not require any change. However, two additional issues arise with mutability, which we will briefly explore below.

4.5.1 Naming Scheme

Because SMT-LIB (and B-SMT respectively) does not support variables, we have to convert a sequence of constant definitions into variable updates when transforming from B-SMT to E-BSMT. However, without additional information about the update naming scheme, we cannot perform the required transformations. Unfortunately, there does not exist a standard way of denoting variable updates, and hence, different automated verifiers follow contrasting strategies. For instance, consider the naming scheme used in Listing 4.18. In this case, the suffix `@` is used to version the variable update.

In order to solve this problem, we can allow tool implementers to define the update naming scheme as a pattern matching script as part of the tool configuration. By doing so, we can leverage tool-specific knowledge and thus support a wide range of automated verifiers with this language extension.

4.5.2 Old Expression

When transforming from E-BSMT to B-SMT, we have to deal with the newly introduced `old` term. Referring to `old(x)` is only allowed within the update of the variable `x`. By design, it is not permitted to refer to the previous state of a variable other than the one we are currently updating.

To properly support the `old` expression, we need a way to resolve it to the previous constant update. We can achieve this by traversing the AST and collecting all updates to the variable. These updates can then be transformed into a sequence of constant definitions by leveraging the specified naming scheme. Statements that modify the SMT-LIB stack, e.g. **`push`** or **`pop`** force us to do more bookkeeping during the AST traversal, but the complexity of the problem itself does not increase.

4.6 Mutable Functions

After having covered mutable variables in more detail, we will now extend our language to support mutable functions. Going from a piecewise function definition to mutable functions is analogous to going from a piecewise constant definition to mutable variables. Thus, all previously discussed issues and solutions also apply in this context. Hence, lifting the aforementioned transformations is trivial. In Listing 4.19, we show our proposed E-BSMT syntax for defining mutable functions. In Listing 4.20, we suggest how such

a definition might look on the B-SMT level, assuming that the piecewise function notation is being used.

```
1 mut fun map(Int): Int
2
3 map(r) :=
4   :| r % 2 -> 42
5   :| r = 13 -> aFun(r)
6
7 map(r) :=
8   :| r = 19 -> 20
9   :| otherwise -> old(map(r))
```

Listing 4.19: E-BSMT: mutable function definition

```
1 fun map@1(Int): Int
2 map@1(r) :=
3   :| r % 2 -> 42
4   :| r = 13 -> aFun(r)
5
6 fun map@2(Int): Int
7 map@2(r) :=
8   :| r = 19 -> 20
9   :| otherwise -> map@1(r)
```

Listing 4.20: B-SMT: mutable function definition

4.7 Defunctionalization

Unfortunately, in contrast to most modern programming languages, the current SMT-LIB standard (and thus, B-SMT) is only first-order, which makes it much more involved to express properties of higher-order languages in SMT-LIB. Because of that, many program verifiers, including Silicon, make use of defunctionalization, which is a transformation technique that eliminates higher-order functions. When defunctionalization is applied, a special `apply` function is introduced, which takes the respective identifier as its first argument along with the original function parameter as the second argument. This process is quite complex, as we have to introduce both a new type along with the respective `apply` function for all function signatures occurring in the program.

In order to achieve a simpler encoding, we introduce the **`defunc`** keyword in E-BSMT, which allows us to retrieve the original, defunctionalized representation when transforming back to B-SMT. Now, let us consider Listing 4.21, which shows our proposed E-BSMT syntax for defining a defunctionalized function. In Listing 4.22, we present how the respective defunctionalized function looks like on the B-SMT level.

```
1 defunc fun map(Int): Int
2
3 map(r) :=
4   :| r % 2 -> 1
5   :| otherwise -> 10
```

Listing 4.21: E-BSMT: defunctionalized function definition

```
1 fun Int<Int>.lookup(Int<Int>, Int): Int
2
3 const map: Int<Int>
4 assume forall r: Int ::
5   r % 2
6     ? Int<Int>.lookup(map, r) = 1
7     : Int<Int>.lookup(map, r) = 10
```

Listing 4.22: B-SMT: defunctionalized function definition

4.8 Partial Functions & Domains

As mentioned earlier, all functions are total in both SMT-LIB and B-SMT, which means that there is no built-in solution to define a partial function in B-SMT. In the following section, we will provide a rough outline on how partial functions and their domains could be added to our language in the form of an extension. This topic provides a promising starting point on potential future work based on this thesis.

For this, consider Listing 4.23, which presents a manual method to allow reasoning about partial functions in B-SMT. First, a "partial" function `f` is introduced, returning -1 when the argument is smaller than zero and 1 when the argument is greater than zero. When the argument is zero, the result is undefined. Now, in order to reason about the domain of the function `f`, we have to introduce a new function `dom_f` corresponding to its domain. This function returns `true` whenever its argument is part of the domain of `f`. After this preamble, we can finally make an assertion about the partial function `f`, e.g. by asserting that zero is not part of the function's range of values.

```
1 fun f(Int): Int
2
3 assume forall x: Int :: x < 0 ==> f(x) = -1
4 assume forall x: Int :: 0 < x ==> f(x) = 1
5
6 fun dom_f(Int): Bool
7 assume forall x: Int :: (x < 0 || 0 < x) <==> dom_f(x)
8
9 assert forall x: Int :: dom_f(x) ==> f(x) != 0
```

Listing 4.23: B-SMT: partial function and domain definition

Now, consider Listing 4.24, which shows our proposed E-BSMT syntax for defining and reasoning about partial functions. To achieve a more human-friendly syntax, we introduce a new keyword along with a method, **`part`** and `dom(f, x)`, respectively. The **`part`** keyword marks a function as partial, allowing us to generate the preamble when transforming from E-BSMT back to B-SMT. In addition, the `dom(f, x)` method takes a partial function `f` as an argument along with a second parameter `x`, returning `true` if `x` is part of the domain of `f`.

```
1 part fun f(Int): Int
2
3 f(x) :=
4   :| x < 0 -> -1
5   :| 0 < x -> 1
6
7 assert forall x: Int :: dom(f, x) ==> f(x) != 0
```

Listing 4.24: E-BSMT: partial function and domain definition

Chapter 5

Technology Stack

After having seen our novel B-SMT language and its potential extensions, we will now choose a suitable technology stack that allows us to implement a convenient toolkit for both B-SMT and SMT-LIB. Furthermore, we will briefly motivate our language of choice for this thesis: TypeScript [34].

All evaluations shown in this chapter were performed on a late 2016 MacBook Pro with a 2.9 GHz Quad-Core Intel i7, running on macOS Catalina.

5.1 Extendability

The extendability of B-SMT is one of the core pillars of the novel SMT language. Allowing users to extend the syntax and semantics of B-SMT easily is of critical importance. For this, users should be able to implement their language extensions quickly and without a steep learning curve. For such a task, a dynamic and interpreted programming language is especially suitable; with JavaScript being one the most prominent examples. In addition to JavaScript, the Rust [40] programming language is another promising candidate. Its focus on performance coupled with the fact that the language can be compiled into WebAssembly [44] makes Rust a compelling contender for the development of our tool.

Moreover, the architecture of our command-line interface should be designed in a way such that extending the tool with a future Visual Studio Code (VSCode) [12] plugin is facilitated. Because VSCode itself is built using Electron [18] and Node.js [19], it supports both JavaScript and TypeScript extensions natively. Even though WebAssembly extensions for VSCode are not supported out of the box yet [16], a workaround that involves compiling Rust into WebAssembly and then generating a Node.js package makes it possible to use Rust within a VSCode extension.

5.2 Performance

A large part of the computation performed by our CLI tool will involve the creation, traversal, and modification of abstract syntax trees. Thus, our language of choice must perform well for these types of computations. Based on the findings from Section 5.1, we have narrowed down our choice to two programming languages, Rust and JavaScript. By benchmarking the performance of different parsing libraries for both languages, we can estimate how fast our implementation for SMT-LIB might perform.

5.2.1 Parsing JSON

For our first experiment, we profile the performance of different parsing libraries in Rust and JavaScript. We will perform these evaluations based on a JSON [29] parser implementation for two reasons. First of all, JSON is a well-known standard, and hence, there exist various JSON parsers for both Rust and JavaScript, which allows for easier comparison. Second, the JSON standard is very similar to S-expressions in terms of parsing complexity, so we should get an approximate estimation of how well the respective SMT-LIB parser should perform.

In this evaluation, the overhead for initializing the parser and loading the source file into memory is not counted towards the execution time in the benchmark. Hence, in this case, only the raw parsing performance is profiled, and all other computational costs are ignored. The results of the evaluation are shown in Figure 5.1.

Library	Language	Throughput	Relative Speed
nom [20]	Rust	201.3 MB/s	100%
pest [26]	Rust	82.1 MB/s	40.7%
chevrotain [43]	JavaScript	81.3 MB/s	40.4%
PEG.js [41]	JavaScript	19.3MB/s	9.6%
parsimmon [28]	JavaScript	18.3MB/s	9.1%

Figure 5.1: Performance evaluation of JSON parsers

5.2.2 Parsing S-Expressions

Finally, based on the performance evaluation in Figure 5.1, we have decided to investigate both `nom` and `chevrotain` more closely as potential parsing library candidates. For this experiment, we have implemented an S-expression parser for both libraries. To evaluate their performance, we have implemented a language generator that builds arbitrary S-expressions based on the grammar specification. Equipped with a set of well-defined examples

containing 500 to 100,000 lines of code, we then measure the parsing throughput of our two implementations. As in the previous evaluation, only the raw parsing performance is measured, and all other computational costs are ignored. In Figure 5.2, we show the results of the evaluation for the different sample files. As expected, Rust performs two to three times faster than the equivalent JavaScript implementation.

Library	**Small** 500 LOC	**Medium** 1K LOC	**Large** 10K LOC	**Huge** 100K LOC
nom [RS]	910.00 ops/s	280.89 ops/s	26.31 ops/s	2.35 ops/s
chevrotain [JS]	537.20 ops/s	145.86 ops/s	12.03 ops/s	0.88 ops/s

Figure 5.2: Performance evaluation of S-expression parsers

5.3 Conclusion

In this section, we have investigated the parsing performance of different libraries to choose a suitable candidate for our implementation. Based on the findings made in this chapter, we have decided that our tool will be realized in TypeScript, which builds on JavaScript and offers both static and dynamic typing at the same time. Even though Rust performs better in our performance evaluation, the `chevrotain` implementation is still reasonably fast, parsing 100,000 lines of code in roughly one second. Besides, VSCode's native support of TypeScript extensions facilitates the development of a future plugin and does not require an involved implementation workaround as it is the case with Rust.

Chapter 6

Implementation

This chapter briefly covers the design and implementation of `yuk-that-smt`, our command-line interface. In Figure 6.1, its overall architecture is shown. The application can be divided into two key components; the client and its corresponding backend. The `yuk-that-smt` client is responsible for handling all user interactions, i.e. requesting a new code transpilation, listing all enabled plugins or installing a new one. On the other hand, the backend service is responsible for lexing, parsing, and transpiling of the specified input for both B-SMT and SMT-LIB.

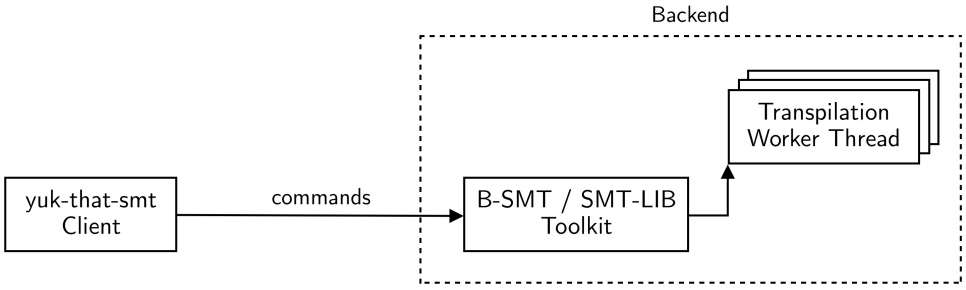

Figure 6.1: Broad architecture of `yuk-that-smt`

6.1 Architecture

6.1.1 Client

Users interact with `yuk-that-smt` directly over the command-line interface, which offers a variety of different commands that can be executed. In the following, we will explore them in more detail.

Transpilation

Our tool offers two separate ways to transpile from B-SMT to SMT-LIB or vice versa; deciding which command to use in what situation depends on the specific use case.

The first command can be used to transpile a single SMT-LIB source file into a B-SMT source file, or vice versa. To achieve this, the `transpile` command is used. In its general form, `transpile` takes one required argument, the path to the input file, along with two optional parameters that allow changing the input format and specifying where the output is written. By default, the input format is determined by the file extension, and the result will be reported to the standard output. In Listing 6.1, a few example commands are shown.

```
1  $ yuk-that-smt transpile ./quicksort.smt2 -o ./quicksort.bsmt
2  $ yuk-that-smt transpile ./bubble.txt -o ./quicksort.smt -f bsmt
3  $ yuk-that-smt transpile ./selection.txt -o ./selection.bsmt --
     input-format smt2
```

Listing 6.1: yuk-that-smt one-time transpilation

The second command is more beneficial when used in a recurrent debugging approach, as `yuk-that-smt` listens to all file changes within a specified directory and then automatically re-transpiles modified files. The introduction of a watch-mode significantly improves user-experience, as the manual tool invocation step is removed. Our tool can be launched in watch-mode by executing `$ yuk-that-smt watch SMT2_DIR BSMT_DIR`, where `SMT2_DIR` stands for the directory containing the SMT-LIB files and `BSMT_DIR` for the directory containing the B-SMT files respectively. By default, it watches for changes in the SMT-LIB directory and writes them into the B-SMT directory. By specifying the boolean flag `d`, `yuk-that-smt` watches for changes in the B-SMT directory instead. In Listing 6.2, we present a few example commands.

```
1  $ yuk-that-smt watch ./smt2-src/ ./bsmt-result/
2  $ yuk-that-smt watch ./smt2-result/ ./bsmt-src/ -d
3  $ yuk-that-smt watch ./smt2-result/ ./bsmt-src/ --direction
```

Listing 6.2: yuk-that-smt watch mode

Plugins

As we have seen in Chapter 4, the extendability of B-SMT is one of its main features. Thus, we allow users to define custom syntax and behaviour for B-SMT and make it available in `yuk-that-smt`. To accomplish this, users can implement an extension in the form of a `yuk-that-smt` plugin, which provides the respective functionality.

Our CLI tool offers a convenient interface for managing plugins: We can install a new extension, uninstall an existing one, enable or disable installed plugins. The `plugin-list` command, as shown in Figure 6.2, displays all currently installed plugins. For every extension, additional information, including the unique name of the plugin, the currently installed version along with a short description, is given. Besides, our tool provides so-called internal plugins, which come pre-packaged with `yuk-that-smt`; without the user needing to install them manually. However, in contrast to external plugins, internal extensions cannot be uninstalled; users can only choose to disable them. Also, installing an external plugin with the same name as an already existing internal plugin will overwrite the internal extension, making it possible to customize all aspects of the B-SMT language.

The commands `plugin-uninstall` and `plugin-disable` behave as expected: They take a single argument, specifying the name of the affected plugin and then perform the respective action on it; either uninstalling or disabling the extension. Similarly, the `plugin-enable` takes a single argument, specifying the plugin to be enabled, along with an optional priority parameter that determines the plugin application order during the transpilation phase. `0` indicates the highest plugin priority, while larger numbers indicate a decreasing priority. We will discuss the importance of this priority in Section 6.1.3.

Besides, `yuk-that-smt` also offers the possibility to install a custom, external plugin with the `plugin-install` command. In its general form, `plugin-install` takes two required parameters, a unique name for the extension and a directory path to where the plugin implementation is located.

name	version	author	description	isInternal	enabled
assertion	0.0.1	Nico Darryl Haenggi	Adds support for assertions.	true	true
dummy	0.0.1	Nico Darryl Haenggi	This dummy plugin is externally overwritten.	false	false
infix	0.0.1	Nico Darryl Haenggi	This plugin adds support for infix operators.	true	false
postfix	0.0.1	Nico Darryl Haenggi	This plugin adds support for postfix operators.	false	false

Figure 6.2: `plugin-list` command, showing an overview of installed plugins

6.1.2 Backend

The backend service of `yuk-that-smt` consists of two principal components, as shown in Figure 6.1. First of all, the B-SMT and SMT-LIB toolkit combine the functionalities of all backend modules into a consistent and straightforward interface; only exposing the respective transpilation functions to the outside. Second, the backend service follows a threaded pool architecture, dispatching an incoming transpilation task to a pool of dedicated workers. For this, our backend service launches a certain number of threads, depending on how many CPU cores are available. These threads are implemented in the form of a

worker, i.e. they execute one task at a time, and when idle, wait for a new job to arrive. This implementation allows us to leverage the multi-core architecture of modern CPUs, executing many transpilation tasks concurrently.

In the following, we will briefly introduce the two transpilation stages, i.e. transpiling from SMT-LIB to B-SMT and vice versa.

6.1.3 SMT-LIB to B-SMT

When a worker thread receives a new transpilation job to convert SMT-LIB into B-SMT, the thread executes the individual stages depicted in Figure 6.3. First of all, the lexer is invoked, passing the resulting SMT-LIB tokens onto the parser, which then produces an SMT-LIB abstract syntax tree. Secondly, the actual transpilation step is performed, converting the SMT-LIB AST into a B-SMT abstract syntax tree. In the next step, the transformations defined by all currently enabled plugins are performed on the B-SMT AST, resulting in an E-BSMT abstract syntax tree. The order in which the plugin transformations are applied is determined by the plugin priority parameter, as explained previously. Plugins with high priority are executed first. The previous step produces an E-BSMT AST, which will then be passed to the pretty printer. In this step, plugins can also customize the pretty-printing, with the highest priority plugin taking precedence when multiple printing definitions are provided for the same AST node.

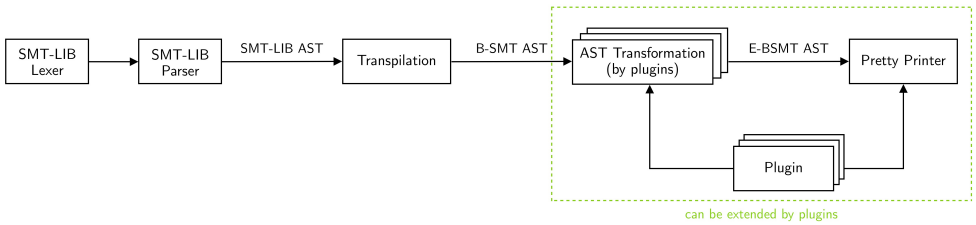

Figure 6.3: Individual stages when transpiling SMT-LIB to B-SMT

6.1.4 B-SMT to SMT-LIB

Similarly to before, when a worker thread receives a new transpilation job to convert B-SMT into SMT-LIB, the thread executes the individual stages presented in Figure 6.4. First of all the extendable lexer is invoked, passing the resulting E-BSMT tokens to the parser. The extendable parser then produces an E-BSMT abstract syntax tree, which is then transformed by the enabled plugins in the same manner as before. This step creates a B-SMT AST, on which the actual transpilation step is performed subsequently. The resulting SMT-LIB AST is then passed to the pretty printer, which will return our transpiled source program.

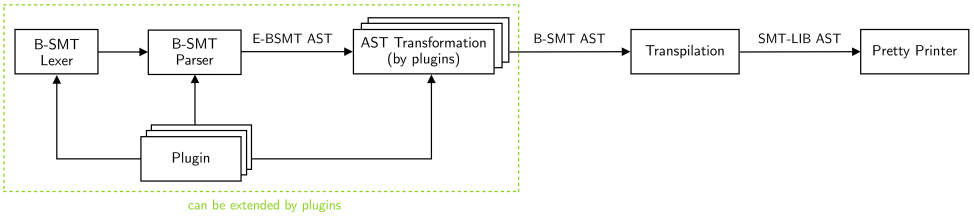

Figure 6.4: Individual stages when transpiling B-SMT to SMT-LIB

6.2 Plugins

We will now briefly explain the structure of an individual `yuk-that-smt` extension. First of all, all plugins need to be located in a separate directory, which must contain a configuration file named `.yuk-that-smt.json`. In Listing 6.3, we show an example plugin configuration. Every plugin must provide a version identifier, an author, and a description along with the primary entry point to the actual plugin implementation (in the form of a JavaScript file).

```
1 {
2   "version": "0.0.1",
3   "author": "Nico Darryl Haenggi",
4   "description": "Adds support for assertions.",
5   "main": "./index.js"
6 }
```

Listing 6.3: `yuk-that-smt` plugin configuration

The plugin implementation itself exports a JavaScript object that contains all the required functionality to customize the B-SMT Lexer and Parser, perform the AST transformations and provide pretty printing on the AST nodes. In Listing 6.4, we present the format of the exported interface. Please note that `yuk-that-smt` passes the parameters shown in Listing 6.4 to the respective plugin methods, i.e. users can utilise these parameters to implement their extensions more easily.

```
1 {
2   lexer(newToken, symbolToken, reservedToken)
3   parser(parser, tokens, nodes, helpers)
4   fromBSMT(ast, nodes, traversor)
5   toBSMT(ast, nodes, traversor)
6   beautify(printer)
7 }
```

Listing 6.4: Exported plugin implementation

For instance, in the `lexer` method, users will be able to customize and extend the B-SMT lexer by creating new token definitions. A new token definition can be generated by applying one of the three helper functions `newToken`, `symbolToken` or `reservedToken`, which introduces a new token, an SMT symbol token or a reserved keyword respectively. Similar to the `lexer` method, users will be able to customise and extend the B-SMT parser with the corresponding `parser` method.

Moreover, the `fromBSMT` and `toBSMT` define the AST transformations performed when translating from B-SMT to E-BSMT, and from E-BSMT to B-SMT, respectively. The abstract syntax tree is provided in the form of the `ast` argument. To traverse the AST, implementers can specify an AST visitor, that provides many helper functions to modify the abstract syntax tree. For this, `yuk-that-smt` provides an API, which is inspired by Babel [3]. Finally, the `beautify` method allows users to specify pretty-printing methods for specific AST nodes. A helper class instance is provided in the form of the `printer` argument, facilitating the beautifying process further.

6.3 Syntax Highlighting

A common strategy to improve the readability of programs is syntax highlighting. Nowadays, most text editors provide some form of syntax highlighting for almost all major programming languages. VSCode is no exception to this, supporting a broad range of different languages by default and hundreds more in the form of extensions.

Traditionally, syntax highlighting support is added by specifying a possibly overapproximated programming language grammar that associates categories to document elements, e.g. language keywords, function and variable definitions and many more. Then, a scope selector is assigned to every category. Ultimately, these scope selectors are used by code editor themes to style the resulting output. Scope selectors are very similar to Cascading Style Sheets (CSS) [45] selectors, i.e. they work hierarchically, and a specific styling is assigned to them. In an attempt to ensure proper syntax highlighting across different editor themes, a naming convention for scope selectors is introduced. For instance, a standard selector to denote a double-slash comment is `comment.line.double-slash`.

In recent years, one particular way of specifying syntax highlighting, known as a TextMate [33] grammar, has been adopted by many text editors and IDEs, including Atom [2], IntelliJ [42], Eclipse [17], Sublime Text [31], and VSCode. A TextMate grammar consists of a set of rules in the form of regular expressions. The program source code is then sequentially matched against the rules contained in the TextMate grammar; with the first successful match taking priority over the others. After processing the complete document,

we end up with a collection of matched document elements, which are then assigned to their respective scope selectors.

For better overall user experience, we provide syntax highlighting grammars for both SMT-LIB and B-SMT in the form of a TextMate grammar, thus enabling us to experience syntax highlighting for a wide range of different text editors. Along with the TextMate grammars themselves, we provide a pre-packaged VSCode language extension that already contains the required language bindings.

Chapter 7

Evaluation

The evaluation of our CLI-tool `yuk-that-smt` is split into three main sections. First, the command line interface is evaluated based on correctness on a well-defined test suite. Second, we evaluate the performance of our CLI when processing real-life SMT-LIB encodings. Finally, to conclude the evaluation, we will investigate the intrinsic limitations of our implementation approach.

7.1 Test Suite

Our test suite consists of two major components. The first one is a set of well-defined SMT-LIB and B-SMT encodings, covering all existing production rules in the form of unit tests. For the second component, we collected a series of real-life SMT-LIB encodings generated by Silicon during the verification phase of the Viper programs.

For both the core B-SMT language and also the SMT-LIB language, a total of 164 unit tests ensure the correctness of the lexing, parsing and pretty-printing phase. Likewise, additional unit tests guarantee the correctness of the language extensions explained in Chapter 4 for B-SMT.

Our integration tests unify the test suites of multiple projects based on the Viper verification infrastructure; specifically from Nagini [14], Prusti [1], Voila [46] and finally, the test suites of Silicon and Silver [30] themselves. All these Viper encodings are then verified with the help of Silicon, using a single parallel verifier, a timeout of 200 seconds along with an increased Java stack size of 64MB. For every Viper program, Silicon outputs two SMT-LIB encodings that will be used in our correctness and performance evaluation. All individual test suites combined, we end up with 3,914 SMT-LIB test files, making up a total of 34.9 million lines of code.

7.2 Correctness

7.2.1 Unit Tests

For our unit tests, we follow a naive testing approach: We start with an SMT-LIB input file (or respectively, a B-SMT input file), lex and parse the contents, transform into our SMT-LIB AST (respectively B-SMT AST) and then ultimately pretty-print the result into SMT-LIB again (respectively B-SMT). With this testing mechanism, no transpilation steps are required, and hence we can directly compare input and output with each other. The test case succeeds if input and output are syntactically equivalent and fails otherwise. Since these unit tests are extremely small and simple, it is feasible to directly compare the two encodings syntactically. However, for more complex SMT-LIB files, this approach does not work. In the next section, we introduce a process to mitigate this issue.

7.2.2 Integration Tests

The main goal of our integration tests is to ensure the correctness of the parsing and transpilation steps for real-world SMT-LIB encodings. In Figure 7.1, the two evaluation strategies are depicted. Analogous to the naive approach, we start with an SMT-LIB encoding, lex and parse the contents and then transform them into an SMT-LIB abstract syntax tree (AST). In the more straightforward case (see Figure 7.1a)), we then pretty-print the result into SMT-LIB again, just as we did in the naive approach. In this case, except for syntactical differences due to pretty-printing, input and output should be the same. In the second case (see Figure 7.1b)), we first transform the SMT-LIB AST into a B-SMT AST before transpiling it back into an SMT-LIB AST. Next, we also pretty-print the result into SMT-LIB again.

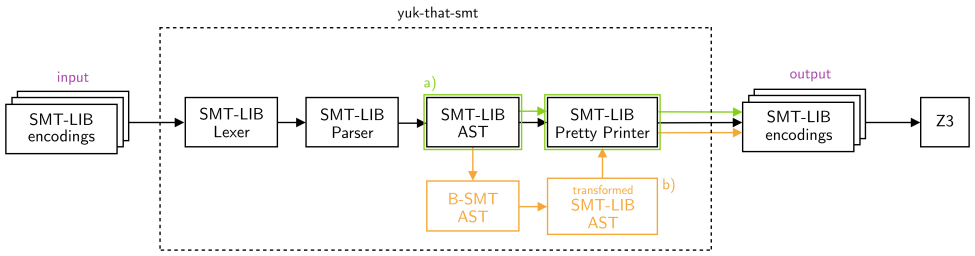

Figure 7.1: Correctness evaluation strategies

Transforming the SMT-LIB encodings back and forth leads to semantically equivalent SMT encodings, but their syntax might differ slightly. Because of that, we need to find a different way to compare the input and output SMT-LIB encodings. We circumvent this issue by invoking the Z3 solver on

both encodings and comparing the resulting output.

Our integration test results are shown in Figure 7.2. As expected, most tests finish successfully and no deviating behaviour is discovered. However, for some SMT encodings, the satisfiability checks within Z3 change from `sat` or `unsat` to `unknown` (or vice versa). This imprecision surfaces because commands occasionally get reordered during our transformation steps. While this does not semantically change the encoding, it still affects the solver, since Z3 assigns unique identifiers to expressions based on their position, and these identifiers are then sometimes used to break ties in certain heuristics used by Z3. Such imprecision cannot be avoided, but is typically not a problem, as the automated verifier itself usually shows imprecise behaviour.

	Successful	Imprecise	Unsound
Strategy 1.a	3,905	9	0
Strategy 1.b	3,884	30	0
Total	7,789	39	0

Figure 7.2: Integration testing results

7.3 Performance

We will now evaluate the performance of our tool on the real-life SMT-LIB encodings generated by Silicon. First, we transpile from our original SMT-LIB file into B-SMT, measuring the performance in the individual transpilation phases, as shown in Figure 6.3. Then, after having generated the B-SMT source code, we will transpile the given encoding back into SMT-LIB again, also measuring the performance in the individual transpilation phases, as shown in Figure 6.4. In this performance evaluation, we will ignore the AST transformations performed by plugins.

Group	Approx. File Size	Approx. LOC
1	10 KB	300
2	250 KB	8,000
3	500 KB	16,000
4	1 MB	30,000
5	10 MB	250,000
6	50 MB	1,000,000

Figure 7.3: SMT-LIB file sizes corresponding to groups

To get a more meaningful performance report, we have grouped the SMT-LIB encodings into six categories, corresponding to the approximate file size of

the respective SMT-LIB encodings. In Figure 7.3, we show which file size corresponds to which group, and the average lines of code for a file within this group.

In Figure 7.4 and Figure 7.5, we now show the performance evaluation for the different categories and transpilation phases for both B-SMT and SMT-LIB. As can be seen from the two tables, our tool `yuk-that-smt` is able to handle both SMT-LIB and B-SMT encodings with approximately 30,000 lines of code within half a second, which is fast enough for our use case. Moreover, we can see that with increasing file sizes, the total time needed to process the source code increases linearly.

Phase	300	8K	16K	30K	250K	1M
Lexing	9.7ms	41.2ms	101.7ms	110.5ms	0.875ms	1.478s
Parsing	12.3ms	80.6ms	135.1ms	153.6ms	1.113s	2.462s
Transpiling	12.0ms	63.3ms	164.3ms	267.9ms	2.830s	8.338s
Pretty-Printing	7.4ms	34.5ms	78.6ms	92.0ms	0.744s	2.830s
Total	41.4ms	219.6ms	479.7ms	624.0ms	5.562s	15.108s

Figure 7.4: Performance evaluation of SMT-LIB transpilation

Phase	300	8K	16K	30K	250K	1M
Lexing	8.5ms	45.5ms	58.9ms	85.0ms	0.683s	1.234s
Parsing	11.9ms	160.7ms	224.7ms	193.3ms	2.316s	4.686s
Transpiling	19.2ms	105.7ms	261.1ms	144.4ms	2.302s	5.668s
Pretty-Printing	2.0ms	11.1ms	23.3ms	18.4ms	0.433s	0.993s
Total	41.6ms	323.0ms	568.0ms	441.1ms	5.734s	12.581s

Figure 7.5: Performance evaluation of B-SMT transpilation

7.4 Limitations

During the development and implementation of this thesis, several limitations have been discovered. In this section, we will briefly examine the most significant ones.

- Chevrotain, the library used in this project to facilitate the lexing and parsing step, does not support stream-parsing. In other words, the parser expects the entire source program as an input string and only returns after a complete parse of the content. This limitation leads to an increased latency until we can start the parsing phase. Moreover, we cannot process exceptionally big files due to V8's [21] hard-coded maximum allowed string length; which lies at 500MB at the time of writing this thesis.
- Currently, our command-line interface restricts comments to occur only between SMT-LIB commands or B-SMT statements, respectively. Consequently, we will not be able to annotate lengthy terms with comments for better readability. However, many real-world SMT-LIB encodings contain comments within certain commands, which would render our CLI tool unusable for those programs. Because of that, `yuk-that-smt` aggregates all comments that occur within a SMT-LIB command and inserts them into the top-level hierarchy before the command itself.
- Similarly, our implementation discards whitespace after lexing, leading to a more accessible parser implementation and grammar. However, this implies that all custom formatting done by the user will be lost when transforming back and forth between B-SMT and SMT-LIB.

Chapter 8

Conclusion & Future Work

8.1 Conclusion

The main focus of this project was to facilitate the debugging process for developers of automated verifiers by providing a more convenient approach to understanding and reasoning about the underlying SMT-LIB code more efficiently. With the introduction of B-SMT, an extendable plugin architecture, and our CLI tool `yuk-that-smt`, we have made an essential contribution to streamline the debugging process and make SMT encodings more accessible.

The extendable plugin architecture has allowed us to explore various feasible language extensions, revealing a broad range of possible applications for our tool.

To choose the technology for our command-line interface, we have performed an extensive evaluation, considering different programming languages and parsing libraries. The implementation of our command-line interface has proven to perform well, processing source files exceeding tens of thousands of lines of code within under a second; for both SMT-LIB and B-SMT. Additional features were added to `yuk-that-smt`, resulting in improved user experience. These features include a watch-mode, where our tool will automatically re-transpile on file modifications, and a TextMate syntax highlighting grammar along with a prepackaged VSCode extension.

8.2 Future Work

During the development of our CLI tool `yuk-that-smt`, most of the planned features could be considered. However, throughout this thesis, we discovered several potentially useful additions that could be made to our current project but were out of scope at the time. Due to the architecture and modular design of our tool, there are endless possibilities for doing future work based on this thesis. In the following, we will briefly discuss the most promising paths to explore:

- In the previous chapter, we have examined the most notable limitations of our work. From a user perspective, mitigating these limitations would significantly improve both the user experience and workflow productivity.
  - For instance, the parsing library used in this project does not support stream-parsing, leading to an increase in memory usage and response latency. One can think of several solutions to circumvent this problem: (1) The `chevrotain` library could be replaced by providing a hand-written recursive-descent parser that supports streaming. (2) The input file could be split into a group of smaller chunks and then passed to the parser separately. With this approach, one could even parallelize the parsing step by leveraging the worker threads offered in `yuk-that-smt`.
  - Our current implementation does not allow any commands within a B-SMT statement in favour of a purer AST representation. However, this decision is inconvenient for developers that want to annotate their formulas and terms with meaningful comments. To circumvent this limitation, during the parsing phase, one could attach the comment to the nearest corresponding AST node. While this will not always accurately maintain the comment position within the source code, it is precise enough for most use cases.
  - Similarly, our current implementation discards all whitespace after lexing, which implies that all custom formatting done by the users will be lost after transpilation. This problem can be mitigated by adding whitespace information to the lexer tokens and leveraging that additional knowledge during the subsequent parsing phase.
- Currently, we only provide a command-line interface for interacting with our tool. While the CLI tool is sufficient for basic usage, providing a Visual Studio Code extension to further streamline the development process would be highly beneficial for increased user experience. By doing so, one could directly integrate useful analysis features (e.g. data-flow analysis) along with possible visualizations. In addition, the introduction of a Language Server [11] will provide the possibility to implement

incremental parsing methods, i.e. avoiding a complete reparse every time the document is changed and hence improving the performance of the tool.

- At the moment, `yuk-that-smt` supports type checking for neither B-SMT nor SMT-LIB. As a result, users won't notice typing errors unless invoking Z3 on the resulting SMT-LIB file. Offering type checking support results in a more robust tool experience. With the modular approach of our software, type checking support could be added in two ways: (1) By providing a `yuk-that-smt` plugin that traverses the abstract syntax tree of the parsed source code and collects the required typing information for the subsequent type check. (2) By extending the implementation of `yuk-that-smt` directly.
- Currently, the transformation stage might occasionally reorder specific commands. As we have seen in Section 7.2.2, this leads to a potential imprecision between the source code and the transformed encoding. To circumvent this imprecision, we need to find a way to preserve the order of the commands within a program. A possible solution approach is to store additional positional information about the parsed program in the respective AST, which can then be leveraged during the transformation state to conserve the existing command order.

Bibliography

[1] V. Astrauskas et al. "Leveraging Rust Types for Modular Specification and Verification". In: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. Vol. 3. OOPSLA. ACM, 2019, 147:1–147:30. DOI: 10.1145/3360573.

[2] Atom. *Atom: A desktop application built with HTML, JavaScript, CSS, and Node.js integration*. URL: https://atom.io/ (visited on 10/06/2020).

[3] Babel. *Babel.js: A JavaScript compiler*. URL: https://babeljs.io/ (visited on 10/06/2020).

[4] Mike Barnett et al. "Boogie: A modular reusable verifier for object-oriented programs". In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2005, pp. 364–387.

[5] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The SMT-LIB standard: Version 2.0". In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*. Vol. 13. 2010, p. 14.

[6] Clark Barrett et al. "CVC4". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177. ISBN: 978-3-642-22110-1.

[7] Andrew P Black et al. "Seeking Grace: a new object-oriented language for novices". In: *Proceeding of the 44th ACM technical symposium on Computer science education*. 2013, pp. 129–134.

[8] Robert S Boyer and J Strother Moore. *A Verification Condition Generator for FORTRAN*. Tech. rep. SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB, 1980.

[9] Roberto Bruttomesso et al. "The mathsat 4 smt solver". In: *International Conference on Computer Aided Verification*. Springer. 2008, pp. 299–303.

[10] Roberto Bruttomesso et al. "The opensmt solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2010, pp. 150–153.

[11] Microsoft Corporation. *Language Server Extension Guide*. URL: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide (visited on 10/06/2020).

[12] Microsoft Corporation. *Visual Studio Code*. URL: https://code.visualstudio.com/ (visited on 10/06/2020).

[13] Bruno Dutertre. "Yices 2.2". In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 737–744.

[14] Marco Eilers and Peter Müller. "Nagini: a static verifier for Python". In: *International Conference on Computer Aided Verification*. Springer. 2018, pp. 596–603.

[15] Robert W Floyd. "Assigning meanings to programs". In: *Program Verification*. Springer, 1993, pp. 65–81.

[16] Evgeny Fomin. *Allow to reference WebAssembly modules in extension*. URL: https://github.com/microsoft/vscode/issues/65559 (visited on 10/06/2020).

[17] Eclipse Foundation. *Eclipse: Enabling Open Innovation Collaboration*. URL: https://www.eclipse.org/ (visited on 10/06/2020).

[18] OpenJS Foundation. *Electron: Build cross-platform desktop apps with JavaScript, HTML, and CSS*. URL: https://www.electronjs.org/ (visited on 10/06/2020).

[19] OpenJS Foundation. *Node.js*. URL: https://nodejs.org/en/ (visited on 10/06/2020).

[20] Geal. *nom, eating data byte by byte*. URL: https://github.com/Geal/nom (visited on 10/06/2020).

[21] Google. *V8 JavaScript engine*. URL: https://v8.dev/ (visited on 10/06/2020).

[22] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[23] Huskell. *Haskell: An advanced, purely functional programming language*. URL: https://www.haskell.org/ (visited on 10/06/2020).

[24] SMT-LIB Initiative. *SMT-LIB Theories*. URL: http://smtlib.cs.uiowa.edu/theories.shtml (visited on 10/06/2020).

[25] The SMT-LIB Initiative. *ArraysEx SMT-LIB Theory*. URL: http://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml (visited on 10/06/2020).

[26] The pest initiative. *pest. The Elegant Parser*. URL: https://github.com/pest-parser/pest (visited on 10/06/2020).

[27] INRIA. *OCaml*. URL: https://ocaml.org/ (visited on 10/06/2020).

[28] jneen laughinghan jneen. *Parsimmon*. URL: https://github.com/jneen/parsimmon (visited on 10/06/2020).

[29] JSON. *JSON (JavaScript Object Notation)*. URL: https://www.json.org/json-en.html (visited on 10/06/2020).

[30] Uri Juhasz et al. *Viper: A verification infrastructure for permission-based reasoning*. Tech. rep. ETH Zurich, 2014.

[31] Sublime HQ Pty Ltd. *Sublime Text: A sophisticated text editor for code, markup and prose*. URL: https://www.sublimetext.com/ (visited on 10/06/2020).

[32] Schwerhoff Malte. "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution". Ph.D. Thesis. ETH Zürich, 2016.

[33] MarcoMates. *TextMate for macOS*. URL: https://macromates.com/ (visited on 10/06/2020).

[34] Microsoft. *TypeScript: Typed JavaScript at Any Scale*. URL: https://www.typescriptlang.org/ (visited on 10/06/2020).

[35] Leonardo de Moura and Nikolaj Bjørner. "Efficient E-Matching for SMT Solvers". In: *Automated Deduction – CADE-21*. Ed. by Frank Pfenning. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 183–198. ISBN: 978-3-540-73595-3.

[36] Leonardo de Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver". In: vol. 4963. Apr. 2008, pp. 337–340.

[37] Peter Müller, Malte Schwerhoff, and Alexander J Summers. "Viper: A verification infrastructure for permission-based reasoning". In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer. 2016, pp. 41–62.

[38] Racket. *Racket, the Programming Language*. URL: https://racket-lang.org/ (visited on 10/06/2020).

[39] Microsoft Research. *F*: A Higher-Order Effectful Language Designed for Program Verification*. URL: https://www.fstar-lang.org/ (visited on 10/06/2020).

[40] Rust. *Rust: A language empowering everyone to build reliable and efficient software.* URL: https://www.rust-lang.org/ (visited on 10/06/2020).

[41] Futago-za Ryuu. *Peg.js: Parser Generator for JavaScript*. URL: https://pegjs.org/ (visited on 10/06/2020).

[42] JetBrains s.r.o. *IntelliJ IDEA: Capable and erconomic IDE for JVM*. URL: https://www.jetbrains.com/idea/ (visited on 10/06/2020).

[43] SAP. *Chevrotain: Parser Building Toolkit for JavaScript*. URL: https://sap.github.io/chevrotain/docs/ (visited on 10/06/2020).

[44] W3C. *WebAssembly*. URL: https://webassembly.org/ (visited on 10/06/2020).

[45] CSS WG. *All CSS Specifications*. URL: https://www.w3.org/Style/CSS/specs.en.html (visited on 10/06/2020).

[46] Felix Wolf. "Verifying Fine-Grained Concurrent Data Structures". PhD thesis. Master thesis, ETH Zurich, 2018.

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):

A Better SMT Language : Design & Tooling

Authored by (in block letters):

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

Name(s):	**First name(s):**
Hänggi	Nico Darryl

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)**
Nunningen, 06.10.2020	N. Hänggi

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