Department of Computer Science
Chair of Software and Computational Systems
LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

Implementing a Solver-Independent SMT-LIB2 Parser-Interpreter and Code-Generator for JavaSMT with Subsequent Evaluation

Janelle King

Computer Science

Supervisor: Prof. Dr. Dirk Beyer
Mentor: Daniel Edwin Baier
Hand-in Date: 20th of December 2023

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgements. This applies also to all graphics, drawings, maps and images included in the thesis

Munich, 20th of December 2023

. . . . . .

(Signature)

Abstract

SMT solvers have become an important cornerstone in the area of software verification and practical problem-solving. Some SMT solvers excel in specific domains, which makes the ability to switch between multiple solvers useful to achieve optimal results for a given problem. However, SMT solvers lack a standardized API, which makes it difficult to switch between them. Bridging this gap among users and developers is a major factor in driving the broader adoption and enhancement of SMT solvers [4]. One standard which aims to mitigate such issues is SMT-LIB2, which offers a common syntax for the description of SMT problems. The solver-independent framework JavaSMT[1] on the other hand addresses the problem of interoperability by bundling multiple SMT solvers, while offering a common API for all of them.
This bachelor's thesis suggests a solution to improve the accessibility and interoperability of JavaSMT. For that purpose, JavaSMT was enhanced by introducing a code-generator and a parser-interpreter for SMT-LIB2. The code-generator translates JavaSMT calls into the SMT-LIB2 format, enabling compatibility with solvers that lack an API but are SMT-LIB2-compliant. The parser-interpreter works in the opposite direction, translating SMT-LIB2 specifications into JavaSMT representations. This facilitates the application of functions to formulas before executing a solver run and also allows to flexibly switch between different solvers.
The implemented solution was evaluated by connecting the Princess solver over the new SMT-LIB2 interface. The traditional integration over the Princess API was compared with the new interface, revealing a similar performance overall. The new binding even outperformed the native API in some areas.

[1] https://github.com/sosy-lab/java-smt, accessed 11.2023

Contents

List of Figures

List of Tables

Acronyms

ANTLR ANother Tool for Language Recognition

API Application Programming Interface

AST Abstract Syntax Tree

BFS Breadth-First Search

BMC Bounded Model Checking

CNF Conjunctive Normal Form

DFS Depth-First Search

FN false negative

FP false positive

LIA Linear Integer Arithmetics

NP Non-deterministic Polynomial-time

pAPI Princess API

pBin Princess binary interface

QF_BV Closed quantifier-free formulas over the theory of fixed-size bitvectors.

QF_UF Unquantified formulas over uninterpreted sort and function symbols.

SAT Boolean satisfiability problem

SMT Satisfiability Modulo Theories

SMT-LIB2 Satisfiability Modulo Theories Library v2

TN true negative

TP true positive

UFs Uninterpreted Functions

Chapter 1

Introduction

1.1 Motivation

The Boolean satisfiability problem (SAT) resides at the very core of theoretical computer science, having important implications for areas of research such as proof complexity and practical satisfiability problems [9]. While SAT is limited to propositional logic, several problems require solutions for more complex formulas like first-order logic. Hence, SAT was generalized to Satisfiability Modulo Theories (SMT), supporting first-order theories like Linear Integer Arithmetics (LIA) and Quantifiers over LIA [13].
This generalization opens up a vast field of applications for SMT solvers, such as predicate abstraction, automated test generation, equivalence checking, extended static checking, the verification of processors and many more [5]. In addition, several SAT use cases could be lifted to SMT, allowing for efficient solutions to known problems. The wide area of applications, coupled with an increased expressiveness and efficiency [17] nowadays makes SMT solvers essential tools on which many applications rely as back-end components [12].
These developments have greatly contributed to further progress in the area of SMT solving, which has become a popular and active research field over the past years. While many solvers have powerful engines for constraint resolution and are conforming to the Satisfiability Modulo Theories Library v2 (SMT-LIB2) standard, they often differ considerably regarding specific capabilities and performance characteristics [3]. First of all, solvers are implemented in various programming languages and offer bindings in different languages too, which has an obvious effect on accessibility. Solvers are also different regarding core functionality, such as

- the general approach (eager vs. lazy [15]),
- supported theories (e.g. arrays, bitvectors etc. [17]),
- supported SMT-LIB2 logics (the solver Boolector [19] e.g. does not support any logics that contain the theory of Integers)[1],
- and support for proof engines.

The dependency on particular features can also lead to lock-in effects, requiring the use of certain solvers, which may entail a degradation of performance for specific application domains. In summary, a plethora of SMT solvers exist, differing heavily regarding

[1] Notably, some older SMT solvers did not even ship any built-in SMT-LIB2 support.

features and accessibility. One solution for this problem is the solver-independent framework **Java SMT**[2], which enables the use of multiple SMT solvers over a single, unified Application Programming Interface (API) [3]. Integrating a solver into a unifying framework may not only compensate for an otherwise lack of standard-compliance, but also allows to comfortably invoke integrated solvers over well-defined APIs, abstracting from the complex inner workings of the solvers and even allowing to switch between different solvers, preventing lock-in effects. Hence, contributing to Java SMT yields great benefits for the accessibility of already available and powerful SMT solvers, facilitating the integration of these solvers in a large range of applications.

1.2 Contribution

In the course of this thesis, JavaSMT was extended by a code-generator and a parser-interpreter for SMT-LIB2.

- The **code-generator** translates JavaSMT to the SMT-LIB2 format. For that purpose, JavaSMT calls are logged, and each object is mapped onto the corresponding SMT-LIB2 representation. The SMT-LIB2 file generated this way can then be used as input for any solver conforming to the SMT-LIB2 standard. This allows to connect solvers without an already existing API to JavaSMT, assuming they support the SMT-LIB2 standard.
- The **parser-interpreter** works the opposite direction, by translating objects defined in SMT-LIB2 to a JavaSMT representation. For that purpose, an SMT-LIB2 file is taken as input and parsed into a tree. The parse tree is subsequently recursively traversed, and a corresponding JavaSMT representation is created in the process. This approach allows to manipulate or apply functions to formulas before they are actually solved and to use JavaSMT API features. Furthermore, the transfer of formulas between solvers is facilitated. Also, models generated by any solver conforming to the SMT-LIB2 standard can be accessed by parsing them to JavaSMT using the parser-interpreter.

Subsequently, the implementation results were evaluated by connecting the solver Princess, which required the implementation of a suitable model interface.
This thesis refers to the following software versions:

- JavaSMT 4.1.0 (revision 21d18cd)
- SMT-LIB Standard Version 2.6
- ANother Tool for Language Recognition (ANTLR) v4 (revision 4.13.1)
- CPAchecker 2.2 (revision 45462)
- Princess - The Scala Theorem Prover (revision 2023-06-19)

1.3 Related Work

The goal of this work is to implement a code-generator and a parser-interpreter for JavaSMT, thus creating an interface to SMT-LIB2. There are already several projects

[2]https://github.com/sosy-lab/java-smt, accessed 11.2023

aiming to translate SMT-LIB2 specifications into internal representations for specific programming languages, such as:

- jSMTLIB
- pySMT
- Parsers in Haskell, SWI-Prolog and other programming languages

jSMTLIB jSMTLIB[3] is a Java library that provides SMT-LIB2 parsing and type-checking as a front-end application, client-server application, Java API and Eclipse plugin. The library takes SMT-LIB2 input either directly as a string or as a file, and then parses this input into an Abstract Syntax Tree (AST). The parsed constraints can then be processed further. For instance, jSMTLIB offers the option of coupling an SMT solver and having it solve the constraints. In addition, jSMTLIB also provides a function that can be used to translate from SMT-LIB2 to other formats that are accepted as input by (some) SMT solvers [11].
The question arose as to whether jSMTLIB would be a suitable basis for implementing a parser-interpreter for JavaSMT. Since it appears the project is no longer actively maintained, and with the capabilities of the ANTLR grammar (see 2.4), there is an alternative which also offers the desired functionality in Java. Hence the decision was made against jSMTLIB.

pySMT pySMT[4] implements a similar approach as JavaSMT: various solvers are connected to the library via their respective APIs and can be used to work with SMT constraints. In addition, pySMT has its own parser for SMT-LIB2 and the option of dumping it. This can be used as a wrapper for working with solvers that are not integrated via their APIs.

Parsers in other languages There are several other parsers that parse SMT-LIB2 for their respective language. For instance, there is one such parser for Haskell[5] and one for SWI-Prolog[6] as listed on the website[7] of the maintainer of SMT-LIB2.

1.4 Outlook

The structure of this thesis is as follows: The second chapter depicts an overview of SMT fundamentals and the current state of research by presenting related work. The third chapter describes the overall architecture and implementation details for the generator and parser-interpreter. This includes several code samples as well as illustrations of challenges that were encountered during the implementation phase. The fourth chapter outlines the evaluation results with respect to unit testing and the verification of the implementation via CPAchecker[8]. The last chapter concludes this thesis by discussing the results, as well as future work.

[3] https://github.com/smtlib/jSMTLIB, accessed 12.2023
[4] https://github.com/pysmt/pysmt/tree/master, accessed 12.2023
[5] https://hackage.haskell.org/package/smt-lib, accessed 12.2023
[6] https://eu.swi-prolog.org/pack/list?p=smtlib, accessed 12.2023
[7] https://smtlib.cs.uiowa.edu/utilities.shtml, accessed 12.2023
[8] https://github.com/sosy-lab/cpachecker, accessed 11.2023

Chapter 2

Background

This chapter introduces some core concepts and definitions in the area of SMT solving and language recognition on which the contribution in the main section of this thesis is built on.

2.1 SMT Solving

SAT solving is a typical decision problem, and is classified as NP-complete in terms of complexity [9]. Since SMT extends the concept of SAT to include theories beyond propositional logic, SMT is also NP-hard and often undecidable [2]. In order to clearly define the objective of SMT solving, it is important to highlight the difference between **satisfiability** and **validity** [12]:

- A formula is valid if it evaluates to `true` for *any* possible assignment.
- A formula is said to be satisfiable if there is *at least one* assignment of values so that the formula evaluates to `true`[1].

Hence, satisfiability is concerned with the existence of a solution to a set of constraints, while validity is concerned with proving statements. To that effect, the goal of SMT solving is to find a satisfiable instance for a set of constraints, with constraints being expressed as formulas. Some SMT solvers are capable to dually check the validity of formulas though [13].Notably, numerous applications do not require *general* first-order satisfiability but satisfiability with respect to *a particular* background theory [5], such as Integers `('Int')` or Bitvectors `('BV')` [3].
If a solver can find a valid assignment for a given formula, it returns the status SAT, else UNSAT is returned.

- The formula $(x \Leftrightarrow y \wedge true)$ will result in SAT, since it evaluates to `true` e.g. for $(x \Leftrightarrow true)$ and $(y \Leftrightarrow true)$.
- The formula $(y \wedge (\neg y))$ will result in UNSAT, since no solution satisfying these conditions exists.

In some cases, a solver may be unable to determine whether a problem is satisfiable or not, which results in the status UNKNOWN [12]. For satisfiable formulas, many solvers have the capability to not only inform about the existence of a solution by

[1] In both scenarios, assignments have to be valid with respect to the domain, i.e. no integer values must be assigned to boolean data types.

returning SAT, but are also capable to provide a variable assignment for which the formula evaluates to `true`, which is called a model [9, 13].
While SAT solvers accept boolean constraints in a Conjunctive Normal Form (CNF) and attempt to find an assignment for which the CNF is satisfiable, SMT solvers accept constraints described in first-order logic, such as arithmetic formulas [20], making it a powerful tool for expressing and solving complex logical problems in various domains.

2.2 SMT-LIB

An example use case for SMT is automatic software verification, in which a backend application solving constraints for the purpose of software verification. In this context, the solver often has to interact with other tools from which it receives input and to which output has to be delivered [12]. Without a common understanding of the exchange format, such interfaces are laborious. The SMT-LIB2[2] initiative is purposed specifically to address such scenarios. It was designed as standardized language and format, as well as for describing (benchmark) problems in the domain of SMT that are understood across different SAT solvers.
Listing 2.1 depicts an example of a simple Boolean logic problem expressed in SMT-LIB2 format. The solver Princess[3] [25] was used for the evaluations in 2.1 and 2.2.

```
1 (declare-const x Bool)
2 (declare-const y Bool)
3 (assert (= x (and y true)))
4 (check-sat)
5 (get-model)
6
7 sat
8 (model
9 (define-fun y () Bool true)
10 (define-fun x () Bool true)
11 )
```

Listing 2.1: *SMT-LIB: SAT example*

Boolean variablea x and y are declared (1-2) for which a constraint is defined (3) that requires the value of x to be equivalent to $y \wedge true$. Princess then checks if this result is satisfiable (4) and retrieves a model (i.e. a valid assignment) for the formula (5). As can be seen in line 7, Princess states the formula is satisfiable and returns the value *true* as possible assignments for both x and y, i.e. model (8-11).
Listing 2.2 is identical to the previous example, however a different constraint is used (2) which is not satisfiable.

```
1 (declare-const y Bool)
2 (assert (and y (not y)))
3 (check-sat)
4 (get-model)
5
6 unsat
7 (error "no model available")
```

Listing 2.2: *SMT-LIB: UNSAT example*

[2] https://smtlib.cs.uiowa.edu/language.shtml, accessed 11.2023
[3] 5philipp.ruemmer.org/princess.shtml, accessed 12.2023

Princess returns the result `unsat` (6) and consequently establishes that no model for this formula exists (7). The SMT-LIB2 specification serves as universal exchange format for the code developed in the course of this thesis.

2.3 Java SMT

With the rise of SMT solving, the SMT-LIB2 format has become increasingly popular. Nonetheless, some tool developers lean towards utilizing solver API's instead, as the use of SMT-LIB2 has several drawbacks, such as [17]:

- Numerous functionalities provided by SMT solvers are not directly supported by SMT-LIB2, such as interpolation and formula decomposition.
- Communication via SMT-LIB2 may introduce performance overhead due to the serialization of queries to the solver into strings.
- Direct use of the solver API may result in a solver lock-in, which makes it difficult to switch between solvers.

JavaSMT proposes a solution for these challenges. Figure 2.1[4] outlines the high-level architecture of JavaSMT. Multiple solvers are integrated over bindings and made accessible over a single, unified Java API. Between the bindings and the API sits the core of JavaSMT, the `SolverContext`. The `SolverContext`'s purpose is to load the background SMT solver and to define the lifetime as well as the scope of the objects that have been generated. Inside that solver context, `FormulaManager`s are provided in order to create formulas in the theories available for the given solver. Also, `ProverEnvironment`s are provided to solve SMT queries [3].

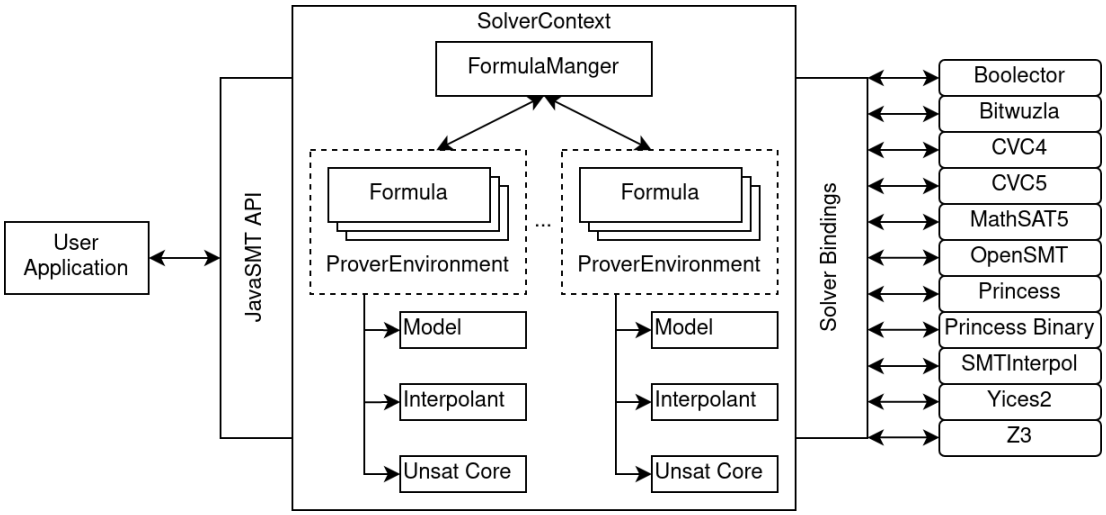

Figure 2.1: *High-level architecture of basic JavaSMT*

Due to the integration of multiple different solvers, JavaSMT offers numerous features, such as support for different theories, e.g. integer arithmetic, arrays or bitvectors [17]. JavaSMT is heavily tested, with its most notable application being the verification

[4]Figure 2.1 is an updated version of the architecture diagram presented by Baier, Beyer and Friedberger [3]

framework CPAchecker [3].
The JavaSMT extensions developed throughout this thesis were also evaluated with CPAchecker and tested using the Princess solver. For the purpose of the evaluation, two different binding types were compared:

- The pre-existing JavaSMT Princess bindings, based on the native Princess API (pAPI),
- and a newly developed binding, based on a compiled Princess binary addressed over SMT-LIB2-conforming inputs.

The new binding type is described in detail in chapter 3.4.

2.4 ANTLR

The parser-interpreter implemented in the course of this thesis maps formulas given in SMT-LIB2 format to JavaSMT objects. A parse tree of the given input file is generated to decide if the input adheres to the SMT-LIB2 standard, and to build the JavaSMT objects while traversing the parse tree.
For that purpose, the well-proven application "ANTLR" is utilized. ANTLR is best introduced as a parser generator capable of processing structured text and binary files. When given a grammar, ANTLR is able to generate a lexer and a parser which can be used to build and traverse parse trees [22]. ANTLR consists of two distinct components:

1. a Java application that translates grammars to parsers[5]. Hence, ANTLR is also a code generator, striving to produce parser code that look similar to what a real programmer would build manually [21].
2. a runtime library which is required by the generated parsers/lexers for actual code execution[6].

2.4.1 Parsing Terminology

In this section, essential terminology from the realm of parsing is introduced. First of all, it is important to note that parsers work on *languages*, in the case at hand on the SMT-LIB2 language. *Grammars*[7] are a formal way to describe a language by defining the syntax of the language through production rules. A *token* is a symbol that is part of a language [21]. It can either belong to a category of symbols (e.g. `UndefinedSymbol` in the SMT-LIB2 grammar) or represent a single keyword (such as `PS_Bool` in the SMT-LIB2 grammar).
The grammar which defines the SMT-LIB2[8] ships pre-defined categories of symbols, keywords and commands. Operators are not comprehensively specified, since the definition of custom operators is possible in the form of Uninterpreted Functions (UFs).
A *lexer* (or tokenizer) is used to segment an input stream of characters into tokens [21]. ANTLR generates the lexer as well as the parser from the grammar. Through the parsing process, it is possible for the parser to check if an expression belongs to a language by matching the structure against the grammar rules. Since the ANTLR-generated parser

[5] Including Java, 10 target languages are supported for the generation of parser code [23]
[6] https://github.com/antlr/antlr4/blob/master/doc/getting-started.md, accessed 11.2023
[7] ANTLR can process context-free grammars [24]
[8] https://github.com/antlr/grammars-v4/blob/master/smtlibv2/SMTLIBv2.g4, accessed 11.2023

can be used to map SMT-LIB2 input directly into Java objects, it also implements the functionality of an interpreter, hence the labeling as "parser-interpreter".

2.4.2 Building a Parse Tree

The parsing process is crucial for the implementation of the parser-interpreter, hence the details of parsing are presented in more detail. The parsing process unfolds in two stages [21]:

1. **Tokenizing**: an input stream of *characters* is grouped into tokens by the lexer. Token classes are built from tokens which belong together, such as integers/identifiers in programming languages, or `CMD_Assert`/`ParOpen` in SMT-LIB2. ANTLR tokens consist of at least two pieces of information: the token type and the matched text.

2. **Parsing**: an input stream of *tokens* is used by the parser to perceive the structure of the input. The produced syntax (or parse) tree represents the structure of the processed (SMT-LIB2) expression and allows retracing how the parser recognized the input.

This process shall be illustrated by an example of the basic SMT-LIB2 expression

```
( assert [...]  )
```

where `[...]` is a placeholder for another (sub)expression. As can be seen in Figure 2.2[9], first the lexer splits the input stream of characters into classified tokens. Next, the parser is recognizing the structure of the tokens, e.g. it recognizes the parentheses as enclosing operators, and builds the parse tree accordingly.

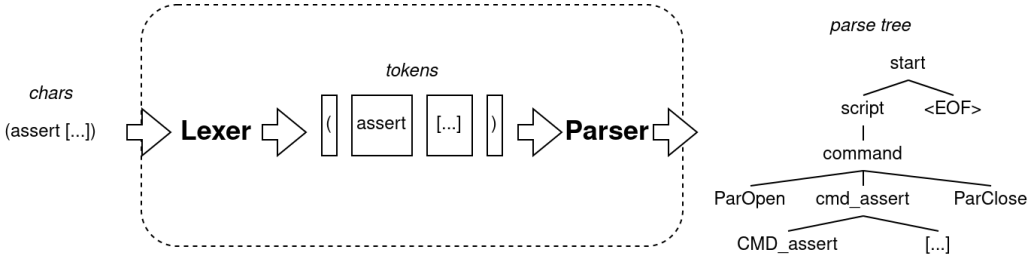

Figure 2.2: *Processing of basic SMT-LIB2 expression by a lexer and parser*

2.4.3 Tree Traversal

Alongside the creation of parse trees, ANTLR also produces code designed for further processing of these trees. Since trees are inherently recursive data structures, the corresponding code for tree traversal also follows a recursive approach. ANTLR produces recursive-descent parsers, which are a "specific kind of top-down parser[10] implemented with a function for each rule in the grammar" [21]. This means the tree walk starts from the root and proceeds to the leafs, and for each nonterminal symbol of the SMT-LIB2 grammar a separate function is generated.
The ANTLR runtime generally support two mechanisms for the traversal of trees [21]:

[9] Figure 2.2 is based on the ANTLR reference [21]

[10] ANTLR uses Depth-First Search (DFS) as default, however custom Breadth-First Search (BFS) implementations are also supported

1. **Listener**: a grammar-specific class is created with enter and exit methods for each rule of the grammar. When the built-in DFS tree walker encounters a node, the `enter()` method is invoked and the current context passed as parameter. After all children of the node have been visited, the `exit()` method is invoked. Consequently, this is a purely event-based pattern with methods being invoked during a "guided" tree traversal.

2. **Visitor**: In order to govern the traversal of the tree, ANTLR supports the visitor pattern. This generates one `visitXXX()` method with a default implementation for each rule of the grammar, hence not every method has to be overwritten manually. While the visitor tree walker also performs a DFS, the way the tree is traversed can be influenced, e.g. by returning values or even preventing the traversal of subtrees.

Full control over the traversal of the parse tree was required for the implementation of the parser-interpreter, since the leaves have to be returned as JavaSMT objects when reached, hence the visitor pattern was utilized. Figure 2.3 depicts a tree walk for a parse tree that was generated based on the already used SMT-LIB2 expression

```
( assert [...]  )
```

Figure 2.3 also illustrates the corresponding `visitXXX()` methods which were automatically generated by ANTLR. While these methods have a default implementation, many had to be overwritten in order to influence the tree walk and also to interface with JavaSMT.

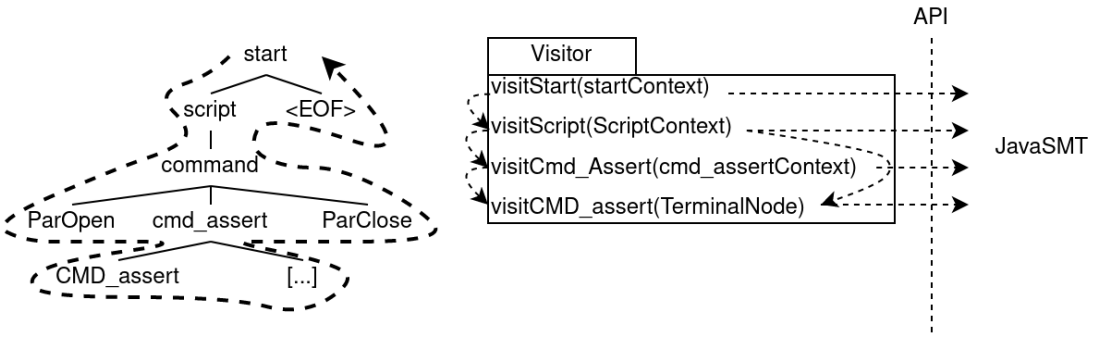

Figure 2.3: *Tree walk via visitor pattern in ANTLR. This figure is based on the ANTLR reference [21]*

Chapter 3

Implementation

This chapter covers details about the implementation of the code-generator and parser-interpreter. Furthermore, the novel Princess integration, as well as the changes to the existing JavaSMT model implementation, are explained in depth. The chapter starts off with the definition of some required terminology.

Formula type Terms in JavaSMT have a specific type, which will be referred to as *formula type* [17]. *Formula type* describes the domain the term belongs to, e.g. Integer or Bitvector theory.

Sort Terms in SMT-LIB2 also have a specific type, which will be referred to as *sort* [10]. *Sort* describes the domain the term belongs to, e.g. Integer or Bitvector theory. In addition to the domain, a sort can also contain additional information, such as its length in the case of a bitvector.

Abstract Syntax Tree An Abstract Syntax Tree (AST) is a tree representing the structure of some input without syntactical details, such as tree nodes for commas or semi-colons [16]. The code-generator will create an AST in order to generate SMT-LIB2 output from the given JavaSMT input.

3.1 Architecture

The primary goal of this work was to extend JavaSMT with an solver-independent code-generator and parser-interpreter for SMT-LIB2. This chapter explains the integration of these new features into the existing JavaSMT framework on an architectural level.
Figure 3.1 illustrates the interaction of existing and newly developed components: Traditionally, the JavaSMT binding uses the solver APIs for the integration. Now, the code-generator may be used to consume JavaSMT objects and produce SMT-LIB2 code which is understood by the majority of solvers. The parser-interpreter may be used to consume SMT-LIB2 and produce JavaSMT constraints, or to import models into JavaSMT. Hence, these new components act as connector, allowing the integration of any

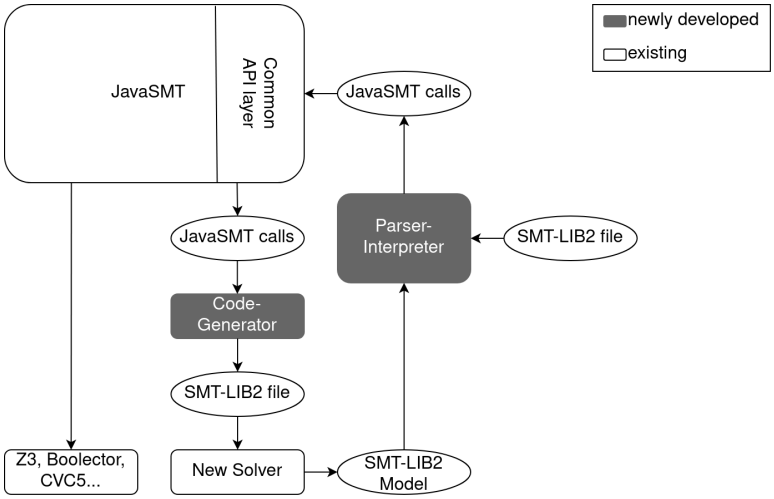

Figure 3.1: *Integration of newly developed components in existing architecture diagram*

solver that adheres to the SMT-LIB2 standard without the need to implement laborious bindings. As a proof of concept, the solver Princess was connected to JavaSMT.

The generator functionality may be enabled by configuring the flag `SMTLIB2` located in the class `SolverContextFactory`. If this flag is set to `true`, the code-generator tracks every formula-related operation performed by JavaSMT. At the end of this process, an SMT-LIB2-compliant file is created, which contains the formulas equivalent to the invoked JavaSMT calls. The final assembly of this file is carried out when executing `isUnsat()` from the `ProverEnvironment`, which triggers the `Generators` `dumpSMTLIB2()` method.

The parser-interpreter has two functions:

First, it can be used to read SMT-LIB2 formulas and interpret these as constraints in JavaSMT. To make use of this functionality, two new methods have been declared in the `FormulaManager` interface and implemented in the `AbstractFormulaManager` class.

- The method `universalParse()` takes a file name as input parameter. The content of the given file is loaded into a string object, lexed, parsed and interpreted as JavaSMT constraints.
- The method `universalParseFromString()` takes an SMT-LIB2 string instead of a file name as input. That string will also be lexed, parsed and interpreted as JavaSMT constraints.

Second, it can be used for importing models generated by a solver that is not part of JavaSMT or does not provide an API. As prerequisite, the model has to follow the SMT-LIB2 standard. In SMT-LIB2, models are distinguished from constraints by introducing the keyword `model`. However, in JavaSMT a model is a list of value assignments, with a value assignment consisting of a variable name (`key`) and the value of that variable (`value`). For the parser-interpreter to be able to distinguish whether a model or a constraint is consumed, the `Visitor` carries a flag. When a node `model` has been visited, that flag is set to `true` and the parser's input will be

interpreted as a JavaSMT model, not constraints.

3.2 Code-Generator

The code-generator's purpose is to map JavaSMT formulas to their counterparts in SMT-LIB2. The SMT-LIB2 formulas are then assembled into a valid constraint and dumped into a file which adheres to the SMT-LIB2 standard. To use the code generation feature, the flag `generateSMTLIB2` has to be set by starting the execution of JavaSMT with the commandline parameter `--solver.generateSMTLIB2=true`. Alternatively, when creating the `SolverContext`, the solver `PRINCESS_BINARY` can be selected, which will also set the flag `generateSMTLIB2` to `true`. This is useful if JavaSMT is not executed from the commandline.

3.2.1 Mapping

JavaSMT has the abstract class `AbstractBaseFormulaManager` that supplies a specific implementation for each supported theory. When an operation regarding the construction of formulas in JavaSMT is executed, the compatible method in the corresponding `AbstractFormulaManager` will be invoked, regardless of which specific solver is used. These methods were modified to call a corresponding method from one of the newly developed `Generator` class if the flag `generateSMTLIB2` is set to `true`, matching the specific theory. Below an example of weaving the `BooleanGenerator` in the `AbstractBooleanFormulaManager` class is given:

```
1 public final BooleanFormula equivalence(BooleanFormula pBits1,
      BooleanFormula pBits2) {
2   [...]
3   if (Generator.isLoggingEnabled()) {
4     BooleanGenerator.logEquivalence(result, pBits1, pBits2);
5   }
6   return result;
7 }
```

Listing 3.1: *Logging in AbstractBooleanFormulaManager*

There are four different kinds of methods in JavaSMT, which will be demonstrated using Boolean theory as depicted in Table 3.1

Type	**JavaSMT**	**SMT-LIB2**
Variable decl.	makeVariable(*name*)	(declare-const *name* Bool)
Constant decl.	makeTrue()	(true)
Function decl.	declareFunction *(name, returnType, inputTypes*)	(declare-fun *name* (*inputSorts*) *returnSort*)
Functions	function(*var_0, ..., var_n*)	(assert (function *var_0, ..., var_n*))

Table 3.1: *Mapping of JavaSMT to SMT-LIB2 for Boolean theory*

The following code depicted in Listing 3.2 is an exemplary generator method for logging the `equivalence()` method of Boolean theory, which belongs to the categorie "functions".

```
1 protected static void logEquivalence(Object result,
      BooleanFormula pBits1, BooleanFormula pBits2) {
2   List<Object> inputParams = new ArrayList<>();
3   inputParams.add(pBits1);
4   inputParams.add(pBits2);
5   Function<List<Object>, String> functionToString =
6   inPlaceInputParams -> "(= " + inPlaceInputParams.get(0) + " "
        + inPlaceInputParams.get(1) + ")";
7   Generator.executedAggregator.add(new FunctionEnvironment(
      result, inputParams, functionToString, Keyword.SKIP));
8 }
```

Listing 3.2: *Logging of Boolean equivalence*

In lines 3 to 5, the input parameters from the function that is being logged are saved into a list. In line 7, a lambda function which serves the purpose of assembling the SMT-LIB2 string corresponding to the JavaSMT method is saved as `functionToString()`. This function will be applied to the recursively evaluated input parameters during the execution of the code in 3.4.

```
1 protected FunctionEnvironment(Object pResult, List<Object>
      pInputParams, Function<List<Object>, String>
      pFunctionToString, Keyword pKeyword) {
2
3   result = pResult;
4   inputParams = pInputParams;
5   functionToString = pFunctionToString;
6   expressionType = pKeyword;
7 }
```

Listing 3.3: *class `FunctionEnvironment`*

The gathered information is then saved as an object of the newly introduced class `FunctionEnvironment` (its constructor is shown in listing 3.3) that consists of matching fields, together with a keyword (`Skip` in this particular case), which is implemented as an `enum` called `keyword`. The keyword is used to determine what kind of type that method resembles in SMT-LIB2. In line 9, this instance of `FunctionEnvironment` is then added to a global list that contains a `FunctionEnvironment` object for each JavaSMT method that has been used to assemble the final constraint.
As the methods in JavaSMT and operations in SMT-LIB2 are not congruent and not all available theories are supported by the generator, an overview of the covered theories and subsequently operations is given below.

Boolean Terms The class `AbstractBooleanFormulaManager` contains the methods resembling the available methods for Boolean terms. Boolean variables and constants may hold either the value `true` or `false`. The `BooleanGeneratorclass` subsequently implements the following methods: `makeVariable()`, `makeTrue()`, `makeFalse()`, `and()`, `or()`, `not()`, `xor()`, `equivalence()`, `implication()`, `ifThenElse()`

Real and Integer Terms Real theory and Integer theory in JavaSMT share the abstract `AbstractNumeralFormulaManager` class, which implements the `NumeralFormulaManager` interface, which itself is extended by

the `IntegerFormulaManager` interface and the `RationalFormulaManager` interface. The `Rational` formula type resembles the Real sort in SMT-LIB2. Each solver then implements an `IntegerFormulaManager` class of its own and a `RationalFormulaManager` class, that contain the methods from the `AbstractNumeralFormulaManager` class. Integer constants consist of any whole number, while Real constants may consist of any real number
The `NumeralGenerator` subsequently implements the following methods for both Real and Integer theory:
`makeNumber(), makeVariable(), add(), sum(), equal(), negate(), subtract(), divide(), multiply(), distinct(), greaterThan(), greaterOrEqual(), lessThan(), lessOrEqual(), floor()`
JavaSMT also provides a modulo operation which is only available for Integer theory.

Bitvector Terms The class `AbstractBitvectorFormulaManager` implements the operations for Bitvector theory. Bitvector variables are defined by their length only, whereas bitvector constants hold information to their length, as well as their assignments consisting of ones and zeros.
The `BitvectorGenerator` implements the following methods, for which it does not matter whether the bitvector is to be interpreted as signed or unsigned:
`makeBitvector(), makeVariable(), equal(), negate(), add(), subtract(), multiply(), not(), and(), or(), xor(), shiftLeft(), concat(), extract()`
For the following methods an unsigned as well as a signed interpretation of the bitvector(s) given as input has been implemented:
`divide(), modulo(), greaterThan(), greaterOrEqual(), lessThan(), lessOrEqual(), shiftRight(), extend()`

Array Terms Operations for array theory are implemented in the class `AbstractArrayFormulaManager`. An array holds information about its index and element type. The types presented above, as well as the array type itself, are available for index and element types. If the `store()` operation has been applied to an array, it also contains information about the stored value.
The following methods have been implemented: `makeArray(), select(), store(), equivalence()`

Uninterpreted Functions The `UFManager` class implements the available methods for the theory of UFs. An UFs consists of a name, the type of the returned formula and the formula types of an arbitrary number of input parameters. The following operations have been implemented in the `UFGenenerator` class:
`declareUF(), callUF, declareAndCallUF()`

3.2.2 Recursive Term Evaluation

JavaSMT constraints may consist of function calls that could be arbitrarily deeply nested, hence recursive evaluation is helpful to resolve constraints and to assemble the atomic terms into String representations of valid SMT-LIB2 constraints. This is realized in the code illustrated in Listing 3.4.

```
protected static String evaluateRecursive(Object constraint) {

  if (constraint instanceof String) {
    return (String) constraint;

  } else {
    Optional<FunctionEnvironment> methodToEvaluate =
    executedAggregator.stream().filter(x -> x.getResult().
    equals(constraint)).findFirst();

    if (methodToEvaluate.isPresent() && !methodToEvaluate.get()
    .expressionType.equals(Keyword.DIRECT)) {
      registeredVariables.add(methodToEvaluate.get());
    }

    List<Object> evaluatedInputs = new ArrayList<>();

    for (Object value : Objects.requireNonNull(methodToEvaluate
    ).get().getInputParams()) {
      String evaluatedInput = evaluateRecursive(value);
      evaluatedInputs.add(evaluatedInput);
    }

    return methodToEvaluate.get().getFunctionToString().apply(
    evaluatedInputs);
  }
}
```

Listing 3.4: *Recursive evaluation of constraint*

The processing of the `evaluateRecursive()` method results in an AST. An example for such an AST based on the constraint
`constraint = equivalence(makeVariable(x), and(makeTrue, makeVariable(y)))`
is given in Figure 3.2. This signifies that the variable 'constraint' is assigned the following equation: $(x = (true \wedge y))$
In this example, all atomic terms are of the type Boolean formula. The method takes a constraint of type `Object` as input (1), which has been assembled by the different formula managers during the mapping. At first, a check for the termination condition is carried out (2), which is met when the constraint is an instance of the class `String`. In that case, a tree leaf is reached and the string (terminal symbol) is returned (4).
Otherwise, the constraint is an instance of the class `Formula`, in which case the `executedAggregator` list (6), that was previously constructed during the mapping, is filtered for the `FunctionEnvironment` object with a `result` attribute that matches the given constraint (8). The result of that operation is saved to the variable `methodToEvaluate` for further processing (6). After that, if the keyword of the resulting `FunctionEnvironment` object is anything but `Direct` (12), it is added to the list `registeredVariables`

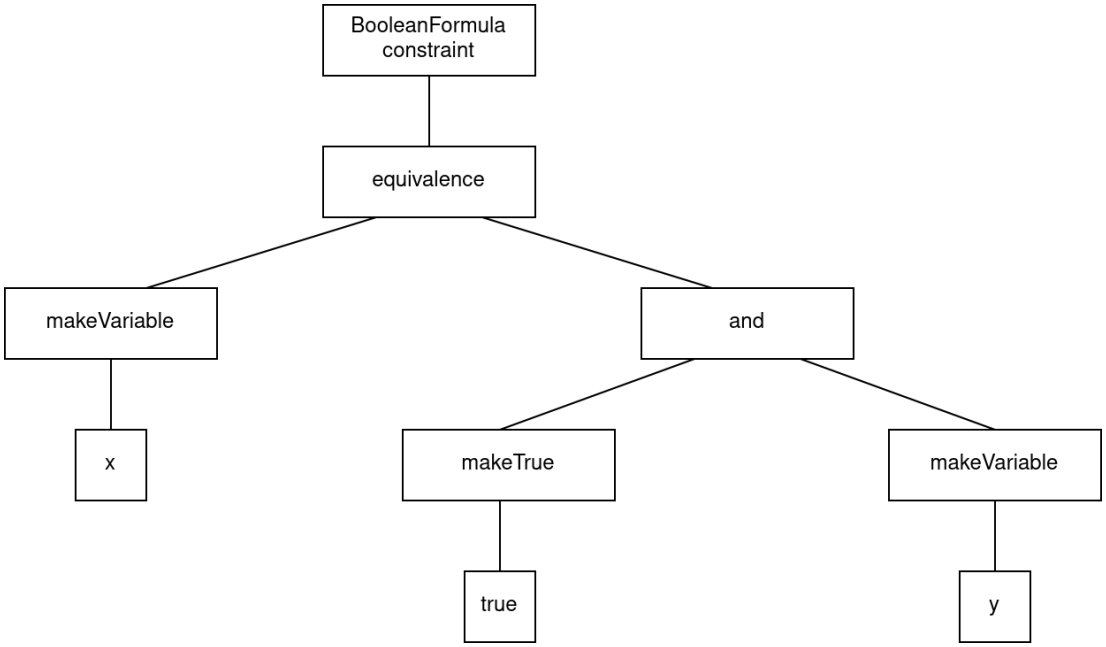

Figure 3.2: *Abstract Syntax Tree for the expression (constraint = equivalence(makeVariable(x), and(makeTrue, makeVariable(y)))*

(13), which holds all variables that need to be declared in SMT-LIB2. Since integer numbers, real numbers, `true` and `false` need to be declared in JavaSMT but not in SMT-LIB2, i.e. are *directly* used, their corresponding `FunctionEnvironments` hold the keyword `Direct`.
Afterwards, every entry of the list `inputParams` extracted from the given `FunctionEnvironment` is evaluated recursively and added to a list called `evaluatedInputs` (16-19). After this list is applied to the function saved in `getFunctionToString()` in the `FunctionEnvironment`, the resulting String is returned (22).

After assembling the formulas, a SMT-LIB2-compliant file is constructed. To achieve this, the list `registeredVariables`, which was put together during the recursive evaluation (Listing 3.4 line 13) needs to be mapped to constant and function declarations in SMT-LIB. The keyword attribute assigned to the entries determines which command and which sort is used. To avoid duplicates, the list is reduced to unique values. The final expressions are then appended to a `StringBuilder`. To extend the example given above, the expressions `(declare-const x Bool)\n` and `(declare-const y Bool)\n` would be added here.
The evaluated string representation of the formula is appended in the form `(assert (`*`formula`*`))`.
Finally, the content of the `StringBuilder` needs to be dumped to a file. This happens when the `isUnsat()` method is executed. This method calls the `dumpSMTLIB2()` method of the `Generator`, which will append the commands `(check-sat)`, `(get-model)` and `(exit)` in order to use the generated file for model generation. The command `(check-sat)` checks if the given constraints are satisfiable. This command has to be placed before `(get-model)`, since a model can only be generated for satisfiable formulas.

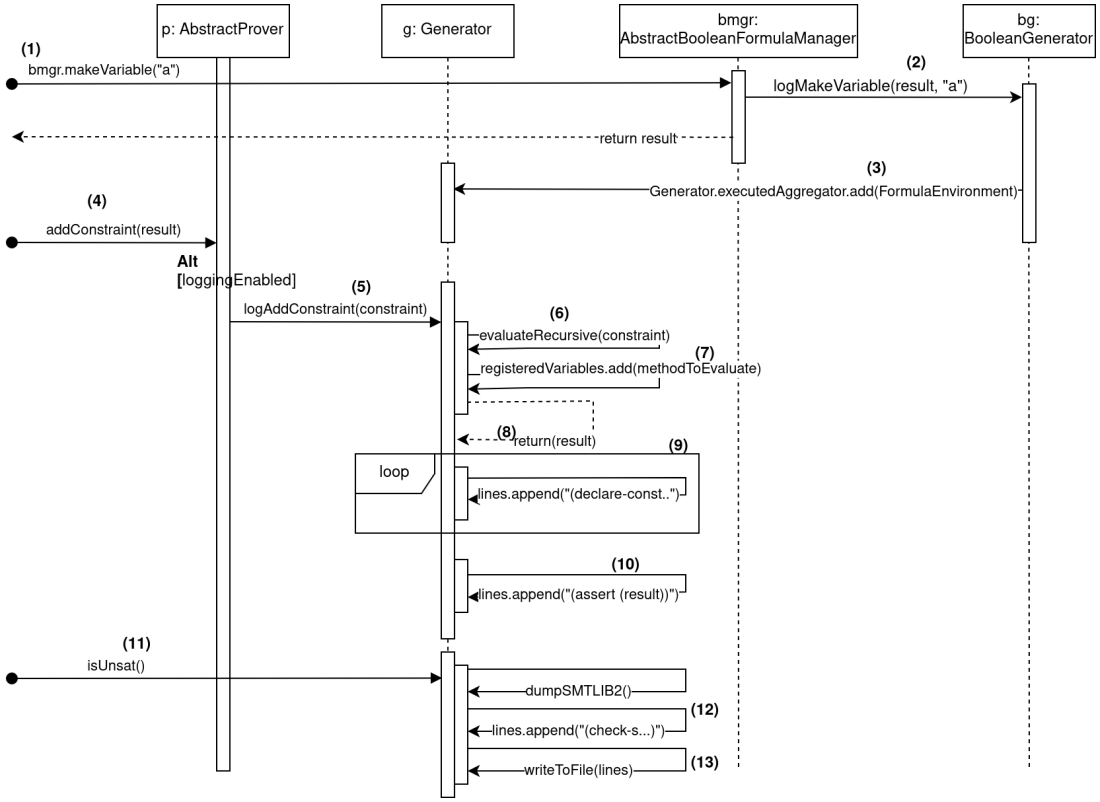

Figure 3.3: *Sequence diagram for generating SMT-LIB2*

The `(exit)` command instructs the solver to terminate. After these final commands have been added to the `Stringbuilder`, its content is written to a file called `Out.smt2`. The `Stringbuilder` is then cleared.
Alternatively, this process can also be triggered by the user without using `isUnsat()` by calling the `dumpSMTLIB2()` method provided by the class `FormulaManager`. This method also executes the `dumpSMTLIB2()` method of the `Generator`.
An example of the contents from a resulting file are shown below:

```
1 (declare-const x Bool)
2 (declare-const y Bool)
3 (assert (= (and true y)))
4 (check-sat)
5 (get-model)
6 (exit)
```

Listing 3.5: *SMT-LIB2 output in file*

The full cycle of obtaining an SMT-LIB2 file from JavaSMT objects is illustrated in the sequence diagram of Figure 3.3:

(1) The user creates a formula by invoking some implementation of the `AbstractFormulaManager`, matching a specific formula type.

(2) This prompts the instantiation of a new `FormulaEnvironment`, holding all relevant context information regarding the created formula.

(3) Said `FormulaEnvironment` is then added to list `executedAggregator`.

(4) The constraint is added to the used solver.

(5) The `logAddConstraint()` method from the Generator is called if the Boolean flag `loggingEnabled` is set to `true`.

(6) The method `evaluateRecursive()` (3.4) of the generator is executed.

(7) The variable and function declarations are added to the list named `registeredVariables` (Listing 3.4 line 13).

(8) The result based on the SMT-LIB2 `(assert ...)` string is returned.

(9) The unique values from the `registeredVariables` list will be looped through, and the necessary declarations for SMT-LIB2 will be appended to the `StringBuilder` object.

(10) The result from evaluating the constraint is also appended. If more constraints are supposed to be added, steps (1) to (8) are repeated.

(11) After all necessary constraints are added, executing the `isUnsat()` method from the solver will call `dumpSMTLIB2()`, which triggers the creation of an SMT-LIB2 file.

(12) The SMT-LIB2 commands `(check-sat)`, `(get-model)` and `(exit)` are appended to the `StringBuilder`.

(13) A file is created and the content of the `StringBuilder` object is written to that file.

3.3 Parser-Interpreter

The parser-interpreter component translates input in SMT-LIB2 format into JavaSMT objects. Similar to the code-generator, recursion is required in order to parse nested formulas. For that purpose, an ANTLR grammar is used, based on which a

- `Lexer`,
- `Parser`,
- and `Visitor`

are created. Inside the `Visitor` class, the mapping from SMT-LIB2 to JavaSMT is carried out.

3.3.1 Parsing

ANTLR grammar The ANTLR grammar used for the parser-interpreter is a (slightly modified) version of Julian Thome's grammar[1].
The grammar is non-deterministic, that is the grammar contains production rules with more than one possible transition. Even though the ANTLR parser can automatically handle this ambiguity [21], it introduces problems during tree traversal. Changing the grammar itself must be avoided, as even altering a single transition would necessitate far-reaching changes to all associated transitions. Hence the issue is addressed by incorporating custom labels, which allows differentiating between transitions without altering the grammar itself. A section of the labeled grammar is shown in 3.6. In line (1), the name of the state is given (`term`), followed by a colon. The succeeding lines describe all possible transitions from that state. The labels are the expressions following `#` at the end of each line containing a transition. Without labeling, all transitions from one state would be covered by one single `Visitor` method. With labeling, each transition has a unique `Visitor` method, which allows to distinguish precisely between paths in the parse tree.

```
1 term:
2 spec_constant
      #term_spec_const
3 | qual_identifer
      #term_qual_id
4 | ParOpen qual_identifer term+ ParClose
      #multiterm
5 | ParOpen GRW_Let ParOpen var_binding+ ParClose term ParClose
      #term_let
6 [...]
7 ;
```

Listing 3.6: *Grammar with labels*

[1] https://github.com/antlr/grammars-v4/blob/master/smtlibv2/SMTLIBv2.g4, accessed 11.2023

3.3.2 Mapping

The mapping in the parser-interpreter is the counterpart of the mapping in the code-generator. A string containing SMT-LIB2 code can either be parsed directly or loaded from a file. The SMT-LIB2 statements are mapped onto the corresponding JavaSMT formula. In addition to the available statements listed in table 3.1, the parser-interpreter also needs an implementation of the expressions `(define-fun)` in order to parse models and `(let)` in order to parse dumped SMT-LIB2 code from other solvers, as listed in 3.2.
For these statements, there is no equivalent in JavaSMT, hence they had to be implemented as illustrated in 3.3.2 and 3.3.2.

Type	**SMT-LIB2**	**JavaSMT**
Variable decl.	(declare-const *name* Bool)	makeVariable(*name*)
Constant decl.	(true)	makeTrue()
Function decl.	(declare-fun *name* (*inputSorts*) *returnSort*)	declareFunction (*name*, *returnType, inputTypes*)
Function def.	(define-fun *name* (*inputSorts*) *returnSort formula*)	not available
Functions	(assert (function *var_0, ..., var_n*))	function(*var_0, ..., var_n*)
Lambda func.	(let (*name formula*) *formula*)	not available

Table 3.2: *Mapping of SMT-LIB2 to JavaSMT for Boolean theory*

Unlike the code-generator component, the parser-interpreter performs this mapping during the recursive traversal of the parse tree and not beforehand. This is possible because the individual tokens are already labeled in the parse tree, which was created on the basis of the grammar, making further pre-processing unnecessary. The following section presents an overview of the mapping functionality.
The functionality of the parser-interpreter is implemented in the `Visitor` class. This class has four instance variables:

`variables:` a map that holds all `Formula` objects that are assigned to a variable name.

`letVariables:` a map that holds all `Formula` objects whose variable names are only valid within a certain scope.

`constraints:` a list that holds the `Formula` objects equivalence of all interpreted `(assert ...)` statements.

`assignments:` a list, which is used for translating the model and will be discussed in detail in chapter 3.4.

Furthermore, the class `ParserFormula` is used, which is depicted in listing 3.7. `ParserFormula` objects require the attribute `javaSmtInterpretation`, which consists of a JavaSMT Formula. Optionally, a `type` attribute may be set, which is used to distinguish between UFss and other formula types. The fields `returnType` and `inputParams` are also only initialized in case of an UFs formula type and are also optional.

```
public class ParserFormula {

  Object javaSmtInterpretation;
  @Nullable
  String type;
  @Nullable
  FormulaType<?> returnType;
  @Nullable
  List<FormulaType<?>> inputParams;

  public ParserFormula(Object pJavaSmtInterpretation) {
    javaSmtInterpretation = pJavaSmtInterpretation;
  }
  [...]
}
```

Listing 3.7: *ParserFormula.class*

In the following, the mapping of the five command types from table 3.2 is elaborated on in detail:

Variable Declarations If the SMT-LIB2 file contains a `(declare-const)` statement, the code in 3.8 is executed. The name of the variable that is to be declared is retrieved (2). The sort type of that variable is saved in `sorts` (3) and can be any of the types that JavaSMT implements. Examples include `BooleanType, IntegerType` or `ArrayType`. Depending on the type of `sorts`, a new `ParserFormula` object of a matching type, such as a `BooleanFormula` for the sort `Bool`, is stored in the `variables` map (5-8), using the symbolic name of the variable (2) as `key` (6).

```
public Object visitCmd_declareConst(Cmd_declareConstContext ctx
    ) {
  String variableSymbol = ctx.symbol().getText();
  FormulaType<?> sort = (FormulaType<?>) visit(ctx.sort());

  if (sort.isBooleanType()) {
    variables.put(variableSymbol, new ParserFormula(bmgr.
    makeVariable(variableSymbol)));
  } else if (sort.isIntegerType()) {
    [...]
  }
}
```

Listing 3.8: *Mapping of the (declare-const) keyword*

Constant Declarations The second type of statement that may occur are constants. Constans can only appear inside terms, which is why the mapped JavaSMT object is not only added to `variables`, but also returned to the calling function, that is the parent node of the parse tree. Those can either be

- numerals of type `Int`, `Real`, binary numbers (like `#b0101`) or hexadecimal numbers (e.g. `#x1010`),
- or `Bools`, i.e. `true` and `false`.

Whenever a constant occurs in a term, the `HashMap variables` structure is examined for a constant with the given value.

- In case of a match, the `ParserFormula`'s `javaSmtInterpretation` field is returned.
- If no such constant is already stored, a new `ParserFormula` object is created, containing the matching JavaSMT `Formula` as `javaSmtInterpretation` attribute. The object added to the `HashMap variables` and returned for further processing.

Function declarations The third type of statement that may be encountered is the declaration of functions which have the following form:
`(declare-fun` *name* `(`*inputSorts*`)` *returnSort*`)`
Each function can only return one value, i.e. one sort, but takes an arbitrary number of values (sorts) as input parameters. JavaSMT provides a corresponding method:
`declareUF(`*name*`,` *returnFormulaType*`,` *List<inputFormulaType>*`)`
The visitor's method for encountering a function declaration retrieves the necessary information (*name*, *returnFormulaType* and *inputFormulatypes*). This processes is shown in 3.9:

1. The declared name of the UFs is retrieved as textual representation from the end of the path below the node `symbol` (3).
2. The `FormulaType` to be returned by the UFs is determined (5, 6). Since the grammar does not distinguish between sorts as input parameter and as return type, the position is used to make this distinction. SMT-LIB2 defines the last sort appearing in the function declaration as return type. Hence, the `FormulaType` returned by the last sort node in the current function declaration statement is used for the `returnType`.
3. It is checked whether UFs takes any input (8).
4. If the UFs takes input parameters, these are added to the `inputParams` list (9, 10) by visiting all sort nodes except the last one, since this is the return sort.

Finally, this information is used to create a `FunctionDeclaration` object in JavaSMT, which is then saved in a `ParserFormula` and added to the `variables HashMap`. The `FunctionDeclaration` is obtained by executing the JavaSMT method `declareUF()` with the desired input and return types as parameters.

```
1  public Object visitCmd_declareFun(Cmd_declareFunContext ctx) {
2
3    String variable = replaceEscapeChars(ctx.symbol().getText());
4
5    FormulaType<?> returnType = (FormulaType<?>) visit(ctx.sort(
       ctx.sort().size()-1));
6
7    List<FormulaType<?>> inputParams = new ArrayList<>();
8    if (ctx.sort().size() > 1) {
9      for (int i = 0; i < ctx.sort().size() - 1; i++) {
10       inputParams.add((FormulaType<?>) visit(ctx.sort(i)));
11     }
```

```
12    }
13
14    [...]
15 }
```

Listing 3.9: *Mapping of (declare-fun)*

(Function Definitions) While function declarations in SMT-LIB2 have an equivalent in JavaSMT, function definitions do not. There are two major differences: `(define-fun)` not only requires the sort of any input to be defined, but takes a variable declaration of the form (*name sort*) as input. Furthermore, the function which will be applied to the input has to be defined. Therefore, a function definition has the form `(define-fun` *name* `(`*variableName inputSort*`)` *returnSort* `(`*function*`))`. To implement this functionality in JavaSMT, `(function-def)` is treated as a `Formula` object and processed like any other function or term, the only difference being that the input variable (when defined for the function) has to be interpreted as JavaSMT and added to the `variables` map for that input to be used in the defined function. Afterwards, said function is evaluated like an `(assert)` statement, resulting in a `Formula` object. This object is also put into the same map, using the *name* as `key`.

Functions Functions in SMT-LIB2 are always introduced with the keyword `assert`, followed by a term that may be nested. Figure 3.4 shows the parse tree for the statement `((assert (= x (and true y))))`. Since that formula contains two operators (`=` and `and`)) and is therefore nested, the node `multiterm` is visited twice during recursion. The visitor's method for the node `multiterm` contains a switch-case branching with a case for each predefined operator (such as `and` and `=`). There is also one additional case for UFs, where the given UFs serves as an operator. Each case returns the `Formula` object which results from applying the operator to the current operands. Since the visitor method linked to a node `term_qual_id` returns the atomic term as a JavaSMT `Formula`, functions are directly assembled when encountered.

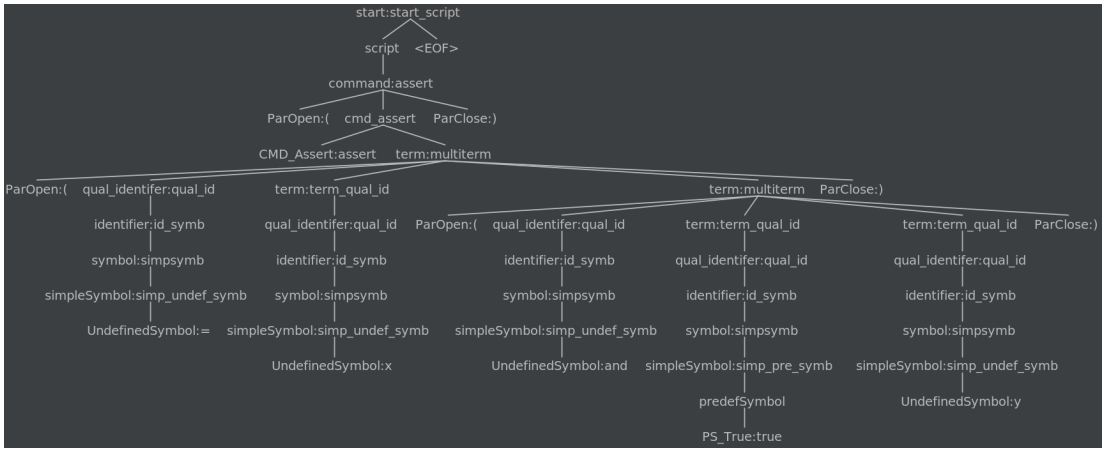

Figure 3.4: *Parse tree for nested terms*

```
[...]
switch (operator) {
  case "and":
    [...]
    return bmgr.and(booleanOperands);
    [...]
  case "or":
    [...]
    return bmgr.or(booleanOperands);
    [...]
```

Listing 3.10: *Node `multiterm`*

Lambda Functions Lambda functions in SMT-LIB2 are introduced with the keyword `let` [10]. Lambda functions have a temporary scope, i.e. are only valid inside the surrounding parentheses. The usage can be demonstrated by the following example:

```
(let ((a (+ x y)) (b (- x y))) (+ a b))
```

is equivalent to

```
(+ (+ x y) (- x y)).
```

A `let` expression consists of

- an arbitrary number of terms assigned to a variable name,
- further processing of these variables.

Unlike SMT-LIB2, JavaSMT does not support lambda functions. In order to interpret `let` expressions in JavaSMT, the following implementation was chosen:

```
public Object visitTerm_let(Term_letContext ctx) {
  for (int i = 0; i < ctx.var_binding().size(); i++) {
    visit(ctx.var_binding(i));
  }
  Formula formula = (Formula) visit(ctx.term());
  for (int j = 0; j < ctx.var_binding().size(); j++) {
    letVariables.remove(ctx.var_binding(j).symbol().getText());
  }
  return formula;
}
```

Listing 3.11: *Node `let`*

```
public Object visitVar_binding(Var_bindingContext ctx) {
  String name = ctx.symbol().getText();
  Formula formula = (Formula) visit(ctx.term());
  letVariables.put(name, new ParserFormula(formula));
  return visitChildren(ctx);
}
```

Listing 3.12: *Node `var_binding`*

The functionality of this code is explained following the example `a (+ x y) and b (- x y)`) from above: First, the variable bindings are visited (3.11, 2-4). The corresponding visitor method (3.12) adds a `ParserFormula` object containing the JavaSMT representation of the term (3). In this case `(+ x y)` and `(- x y)` are added to the map `letVariables` (4) using the variables name `a` and `b` (2) as keys. Afterwards, the term following the variable binding is evaluated with usage of the new entries in `letVariables`, (3.11, 5). The term which was evaluated and interpreted as JavaSMT is then returned (line 9).
It is important to notice that the hash map `letVariables` differs from the general hash map `variables`. While the content of `variables` is consistent during the whole parsing process, entries in `letVariables` are deleted immediately after leaving the corresponding node. This is necessary to prevent conflicts when reusing variable names in `let` expressions.

3.4 Model

The implemented features are incorporated by passing an SMT-LIB2 string, not by addressing the solver over the API. Thus, the output of the solver is initially only present in SMT-LIB2, which is why additional functionality was required to convert the SMT-LIB2 model into a JavaSMT model.
A model constitutes a variable assignment that satisfies the formula (see chapter 2.1). To introduce a model in SMT-LIB2, either the keyword `sat` or `unsat` is used. In cases where no solution is found, `unsat` is followed by a solver-specific error message. For example, the constraint $x \wedge \neg x$ results in the following Princess "model"[2] (3.13):

```
1  unsat
2  (error "no model available")
```

Listing 3.13: *Princess model for $x \wedge \neg x$*

Listing 3.14 exemplifies the case where a solution exists, based on the constraint $x = y$, where both x and y are of the sort `Bool`. The keyword `sat` (1) is followed by an expression in parentheses, which is introduced with the keyword `model` (2). Subsequently, function definitions (see chapter 3.3.2) are listed, with value assignments for each variable that result in an overall satisfiable formula (3-4).

```
1  sat
2  (model
3    (define-fun y () Bool true)
4    (define-fun x () Bool true)
5  )
6
```

Listing 3.14: *Princess model for $x = y \wedge true$*

If the requested formula does not contain any variables, the number of function definitions is zero. Listing 3.15 illustrates the Princess output of such a case, based on the example constraint *true*:

[2]Since the formula is not satisfiable, no actual model exists.

```
  sat
(model
)
```

Listing 3.15: *Princess model for* `true`

In JavaSMT, a model is constituted of a `List` that stores instances of the class `ValueAssignment`, with a `ValueAssignment` consisting of the fields in Listing 3.16.

```
private final Formula keyFormula;
private final Formula valueFormula;
private final BooleanFormula formula;
private final Object value;
private final ImmutableList<Object> argumentsInterpretation;
private final String name;
```

Listing 3.16: *JavaSMT model*

- The `keyFormula` (1) consists of the JavaSMT Formula of the variable to which a value is assigned.
- The `valueFormula` (2) is the JavaSMT Formula of the value that is assigned to the given keyFormula.
- The `formula` (3) represents the equivalence relationship between the `keyFormula` and `valueFormula`, i.e. `formula` is constructed by applying the equivalence operation from the corresponding `FormulaManager` to `keyFormula` and `valueFormula`.
- The variable `value` (4) holds the Java representation of the given `valueFormula`.
- The field `ArgumentsInterpretation` consists of an empty list for value assignments that have no input parameters, e.g. UFs. If a value assignment has input parameters, it holds an `ImmutableList` of `Objects`, which contains a Java representation of each input parameter.
- The `String` representation of the `keyFormula`, i.e. the variable name is assigned to the field `name` (6).

Since the model returned by Princess (and other SMT-LIB2-compliant solvers) adheres to the SMT-LIB2 format, the parser-interpreter described in 3.3 is able to process such models. The `isModel` flag is used so that the parser-interpreter can distinguish between retrieving a constraint and a model. This flag is set to `true` as soon as the node `Resp_get_model` (see listing 3.17) is encountered.

```
public Object visitResp_get_model(Resp_get_modelContext ctx) {
  isModel = true;
  return visitChildren(ctx);
}
```

Listing 3.17: *Resp_get_model*

For each occurrence of the expression `(define-fun ...)`, the presence of this flag is checked, as the individual value assignments for a model of Princess are always created this way. If `isModel` equals `true`, the

`assignments` list is updated. For that purpose, the information from the `(define-fun)` expression is mapped to the corresponding attributes of the class `ValueAssignment`. To visualize this process, it is illustrated using the example `(define-fun example ((x Bool)) Bool (= x true))` in table 3.3:

JavaSMT	SMT-LIB2	Mapping
`keyFormula`	`example`	`makeVariable("example");`
`valueFormula`	`(x Bool)` `(assert (= x true))`	`makeVariable("x");` `equivalence(x, makeTrue)`

`formula`	`(define-fun example`
	`((x Bool))`
	`Bool (= x true))`
		`equivalence`
		`(example, equivalence`
		`(x, makeTrue));`

`value`	`(x Bool)` `(assert (= x true))`	`"x = true"`
`argumentsInterpretation`	`[]`	
`name`	`example`	`"example"`

Table 3.3: *Mapping of SMT-LIB2 to JavaSMT model*

The information required for the mapping is already available through the implementation of the `Visitor` method for `(define-fun)` (see 3.3.2), and can therefore be easily assigned to the corresponding attributes in `ValueAssignment` when a model is parsed. Notably, the field `argumentsInterpretation` is assigned an empty list. This is due to the fact that it is not possible to retrieve the values from the input parameters only from the given string.

3.4.1 Arrays as Constants

For the model interpretation, it was necessary to implement a specific case that is not supported when parsing and interpreting constraints: the command `(as const (Array ...))`. In SMT-LIB2, this expression is employed to characterize an array that has its elements set to a fixed value at all indices. For example, `(as const (Array Int Int) 2)` represents an array that uses integers for the index and integers as data type for the elements, and in which all indices hold the value 2. However, JavaSMT does not implement such a feature. Since arrays, unlike bitvectors, do not have a fixed length, it is also not possible to replicate this function using the `store()` method. Due to this factor, the parser-interpreter does not support this function in general. Because models in the array theory of Princess occasionally contain expressions like `as const ...`, it was necessary to come up with a substitute for this particular purpose.
The model only considers non-default values, hence constant arrays are substituted with arrays that hold no default values when interpreted as JavaSMT. To still pass the information about the default assignment to the user, the `String` representation is adapted so that it contains the SMT-LIB2 expression: `(as const (Array` *sort sort*`)` *value*`)`. For the purpose of visualization, the assignment of the `ValueAssignment` in the case of an `as const` operation is shown here using an example:

- `keyFormula: ArrayFormula<IntegerFormula, IntegerFormula> example`
- `valueFormula: ArrayFormula<IntegerFormula, IntegerFormula>`
- `formual: example = ArrayFormula<IntegerFormula, IntegerFormula>`
- `value: "(as const (Array Int Int) 0)"`
- `argumentsInterpretation: []`
- `name: "example"`

3.5 JUnit Testing

Testing may seem like a good solution to verify the correctness of code, however, as Dijkstra put it [14]:

> "Software testing can be used to show the presence of bugs but never to show the absence of bugs."

Passing all tests without errors does not guarantee the correctness of code. But it may help to uncover bugs, which increases the reliability of code [1]. Keeping the limitations of testing in mind, the quality of the code implemented in the course of this thesis was evaluated by unit tests. The specifics and the results of these tests are outlined in this section.
In order to reduce the probability of errors, the implementation was tested extensively by the means of JUnit tests. Despite having an extensive test suite, The existing JavaSMT Unit tests were unsuitable for testing the recently implemented functionality. Accordingly, the existing test suite was extended by additional JUnit tests. For that purpose, a test class was written for each of the different generator classes listed in 3.5.1. Furthermore, one large test class covers the code of the `Visitor` class, which implements the parser-interpreter and the model translation. Each of these test classes aims to cover the possible input and edge cases as widely as possible.

3.5.1 Code-Generator

The classes that were tested to ensure the functionality of the code-generator are as follows:

- `BooleanGenerator`
- `NumeralGenerator`
- `BitvectorGenerator`
- `ArrayGenerator`
- `UFGenerator`

Each JUnit test invokes all methods of the corresponding generator class, using various parameters to cover all edge cases. The expected result was a priori specified as string representation, i.e. as SMT-LIB2 string, and subsequently compared with the actual output of the generator of the corresponding method. Separate test cases were implemented to cover exceptions based on the newly implemented class `GeneratorException`, which

Element	Class, %	Method, %	Line, %	Branch, %
all				
org.sosy_lab.java_smt.basicimpl.NumeralGenerator	100% (1/1)	100% (35/35)	100% (116/116)	91% (11/12)
org.sosy_lab.java_smt.basicimpl.BooleanGenerator	100% (1/1)	100% (26/26)	100% (79/79)	100% (0/0)
org.sosy_lab.java_smt.basicimpl.BitvectorGenerator	100% (1/1)	100% (63/63)	100% (202/202)	100% (0/0)
org.sosy_lab.java_smt.basicimpl.ArrayGenerator	100% (1/1)	100% (10/10)	100% (56/56)	100% (20/20)
org.sosy_lab.java_smt.basicimpl.UFGenerator	100% (1/1)	100% (9/9)	100% (74/74)	100% (26/26)

Figure 3.5: *Test coverage of the code-generator*

Element	Class, %	Method, %	Line, %	Branch, %
org.sosy_lab.java_smt.basicimpl.parserInterpreter.Visitor	100% (1/1)	100% (37/37)	93% (674/722)	78% (395/504)

Figure 3.6: *Test coverage of the parser-interpreter*

may be triggered by invalid input parameters.
The JUnit tests achieve a full test coverage of 100 % for the generator classes, as shown in Figure 3.5.

3.5.2 Parser-Interpreter

To ensure proper functioning of the parser-interpreter, it was necessary to thoroughly test the class `Visitor`, which encapsulates the entire implementation. Compared to the code-generator, the parser-interpreter works the opposite direction, which made it possible to re-use basically the same JUnit tests. The key difference is that the previously expected results now serve as input, while the previously used JavaSMT input code now represents the expected result. Hence, instead of comparing strings, JavaSMT constraints are compared.
In addition, new test cases had to be developed for operations that JavaSMT does not implement, such as `(define-fun [...])`.
Exceptions were also tested separately. Due to the parser-interpreter's upstream position in relation to JavaSMT, it encounters a significantly greater number of exceptions compared to the code-generator. The code-generator is positioned downstream of JavaSMT, which means that a variety of possible invalid inputs are already handled by the methods calling the generator.
Another part of the `Visitor` class is the conversion of the Princess model into a JavaSMT model. Test cases for the model conversion were also designed, with the exception of models that contain `Reals`. While the parser-interpreter is able to interprete the sort `Real`, Princess is not. Hence, no test cases using Reals were added to test the new Princess backend.
The newly implemented test code for the `Visitor` class exhibits a total code coverage of 100 % for methods and 93 % coverage on lines of code, as shown in Figure 3.6. The lower coverage is due to the fact that not all exceptions were tested due to their sheer number.

3.6 Challenges and Solutions

In the course of this thesis, some general challenges that typically arise from the work on a large, pre-existing code base were encountered. Besides, the

new functionality depended on, and was limited by, existing features. Resolving such dependencies without breaking existing functionality turned out to be a challenging task in many scenarios.
Besides these general aspects of collaborative software development, several technical challenges were encountered during the implementation phase. One major problem throughout development was the mapping between JavaSMT and SMT-LIB2 operations, since no one-to-one relationship exists. For example, JavaSMT does not support the definition of functions. However, the parser-interpreter requires this functionality for the model parsing. In other cases, JavaSMT supported operations like modular congruence for which no equivalent in the SMT-LIB2 is defined. Fully implementing an extensive specification such as SMT-LIB2 was no more feasible than initiating changes to the SMT-LIB2 specification. To resolve these tensions, thorough considerations what operations to support and how to implement the corresponding functionality had to be made for each individual case.
Another issue emerged from the ambiguity of the SMT-LIB2 grammar. Because changes to the grammar were not an option, labeling production rules for the sake of clarity proved an elegant solution to this problem.
Yet another challenge arose from the implementation of the model interface, because the proper implementation of this interface requires a solver context. Since the creation of a pseudo solver for that purpose would have entailed great efforts, the solver context of Princess was repurposed instead.

Chapter 4

Evaluation

In this chapter, the newly added code-generator and the parser-interpreter was evaluated against the existing Princess bindings. Before going into details about the results, our evaluation setup and methodology is explained in detail.

Data Availability Statement The parser-interpreter and code-generator implementations of this thesis can be found in the JavaSMT GitHub repository[1]. Furthermore, all data produced during the evaluation can be found on Zenodo [18].

4.1 Benchmarking

The benchmark tasks were carried out with the framework CPAchecker [6]. Just like JavaSMT, this framework is developed by SoSy-Lab[2]. CPAchecker is a software verification framework, mainly for C programs. Many algorithms within CPAchecker utilize SMT solvers provided by JavaSMT. To be able to compare the capabilities of different solver implementations in JavaSMT, we utilize CPAchecker by verifying the same tasks with the same algorithm using different SMT solvers. Since we want a broad set of benchmarking tasks, we use tasks from the SV-Benchmarks repository of software verification tasks, that is also used by the Competition of Software Verification[3].

4.1.1 Layout

To evaluate the new implementation of the Princess integration via SMT-LIB2, the existing Princess bindings over the API are used for reference. All benchmarking tasks were conducted between the former (API) and the new (Binary) Princess bindings exclusively. The following frame conditions apply to the benchmark:

- The evaluation was performed using the VerifierCloud cluster of the SoSy-Lab that is also used for the International Competition of Software Verification.
- The tasks were performed on identical Intel Core i9-9900 processors with a clock speed of 3.10 GHz.
- The running operating system was Ubuntu 22.04.

[1] github.com/sosy-lab/java-smt/pull/344, accessed 12.2023
[2] www.sosy-lab.org
[3] sv-comp.sosy-lab.org/2023/benchmarks.php

- It was specified that only 2 cores were to be used and that the tasks could only consume 900 seconds and 7 GB of memory.
- The benchmarking framework Benchexec[4] [7] was used with the Bounded Model Checking (BMC) verification approach. In this approach, the program to be verified is transformed into predicates, which are then checked by a SMT solver regarding some specification while unrolling only a predefined number loops (k-1 and k-10 in this case) [8].
- The ReachSafety category of tasks was chosen for the evaluation, as it provided the most benchmarking tasks. The specification defines that some error locations in the program should not be reachable. The following subset from the test sets of the SV-Benchmarks[5] was utilized here:

  `ReachSafety-Arrays`,
  `ReachSafety-BitVectors`,
  `ReachSafety-ControlFlow`,
  `ReachSafety-ECA`,
  `ReachSafety-Floats`,
  `ReachSafety-Heap`,
  `ReachSafety-Loops`,
  `ReachSafety-ProductLines`,
  `ReachSafety-Recursive`,
  `ReachSafety-Sequentialized`,
  `SoftwareSystems-AWS-C-Common-ReachSafety`,
  `SoftwareSystems-DeviceDriversLinux64-ReachSafety`

Each test set was run twice for the two respective solvers: Once with one loop unrolling and once with 10-k loop unrolling. In addition, the prover option `use_binary` was set for tasks executed with the Princess binary interface (pBin), hence not only SMT-LIB2 code was generated but also used for model generation. The configuration with which CPAchecker performs the evaluation is defined in `rundefinition`. The `rundefinition` for the Princess binary with 10-k loop unrollings is shown as an example in Listing 4.1. As the options for Princess API (pAPI) and pBin were selected identically, except for the model (6), deviations in the results are not caused by the configuration.

- In line (1) the run definition is given a name (`k10-princess_binary`).
- In line (2) the number of loop unrollings k is defined.
- Lines (3) and (4) contain a command to encode both Floating Point Numbers and Bitvectors as Integers.
- Line (5) defines the solver to use `princess_binary`.
- In line (6) the `ProverOption` to use the Princess binary is set.

```
1 <rundefinition name="k10-princess_binary">
2   <option name="-setprop">cpa.loopbound.maxLoopIterations=10</option>
3   <option name="-setprop">cpa.predicate.encodeFloatAs=INTEGER</option>
```

[4] github.com/sosy-lab/benchexec, release 3.19, accessed 11.2023
[5] gitlab.com/sosy-lab/benchmarking/sv-benchmarks, revision 509aa682, accessed 12.2023

```
4   <option name="-setprop">cpa.predicate.encodeBitvectorAs=INTEGER</
      option>
5   <option name="-setprop">solver.solver=princess_binary</option>
6   <option name="-setprop">proveroptions.option=use_binary</option>
7 </rundefinition>
```

Listing 4.1: *Rundefinition Princess_Binary k10*

4.1.2 Benchmark Results

The results of the tasks show that the performance of pAPI and pBin are very similar, as visualized in table 4.1, broken down by true positive (TP), true negative (TN), false positive (FP), false negative (FN), memory usage and CPU time.

	TP	TN	FP	FN	Memory usage	CPU time
k-1 pAPI	1	106	0	35	23805.24 MB	2055.33 s
k-1 pBin	1	105	0	35	25191.07 MB	1831.48 s
k-10 pAPI	111	34	0	18	36519.31 MB	1894.44 s
k-10 pBin	109	35	0	17	37170.27 MB	1860.45 s

Table 4.1: *Results of benchmarking Princess*

The only difference in the results for the execution with k-1 loop unrollings is pAPI recognizing one more true negative in a test case. For this test case, pBin produced a null pointer exception. The reason for this behavior could not be clearly derived from the log files.
For the execution with k10 loop unrollings, pAPI recognized two more true positives, while pBin timed out for these two cases. However, execution with pBin detected one more true negative, while execution via pAPI led to a segmentation fault for the same test. The cause of this could not be determined from the log files either. In addition, using pAPI resulted in one extra false negative, for which pBin led to a timeout. In summary, the following accuracy can be calculated for the individual runs:

- k-1 Princess API: 75,4 %
- k-1 Princess Binary: 75,2 %
- k-10 Princess API: 89,0 %
- k-10 Princess Binary: 89,4 %

When evaluating the correctness of the solutions provided by pAPI and pBin, it was expected that the results would be identical. This is almost the case, with a deviation of 0.2 % for runs with one loop unrolling and 0.4 % for runs with ten loop unrollings. Not only was the performance of the different implementations measured in terms of the correctness of the results, but also the consumption of the resources required to achieve the results, more specifically CPU time and memory consumption. The results of these measurements can be found in the figures 4.1, 4.2, 4.3 and 4.4.

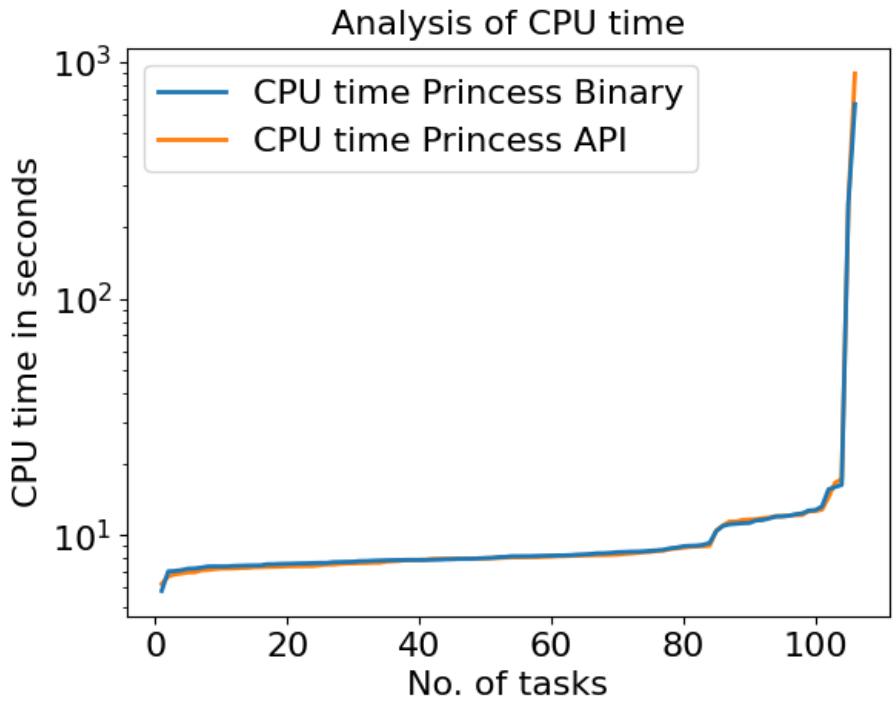

Figure 4.1: *CPU time for intersection of correctly solved tasks with k-1 loop unrollings, sorted by seconds*

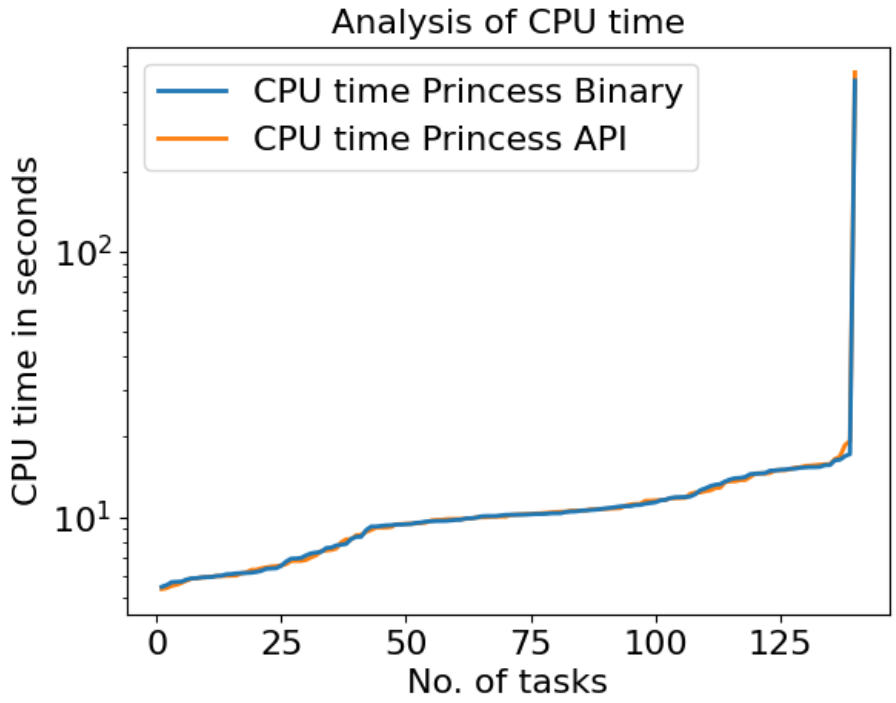

Figure 4.2: *CPU time for intersection of correctly solved tasks with k-10 loop unrollings, sorted by seconds*

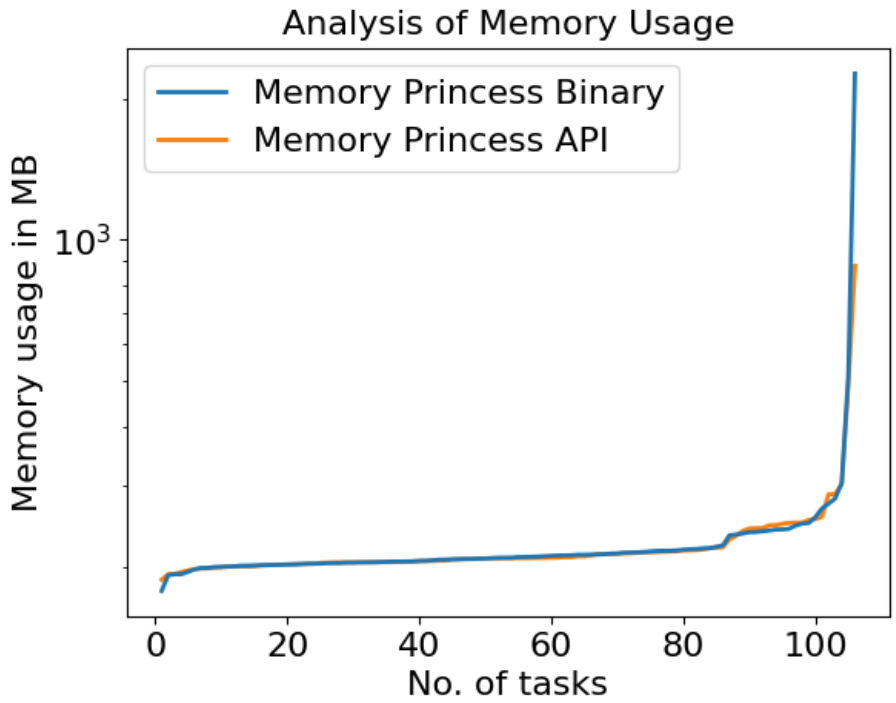

Figure 4.3: *Memory usage for intersection of correctly solved tasks with k-1 loop unrollings, sorted by seconds*

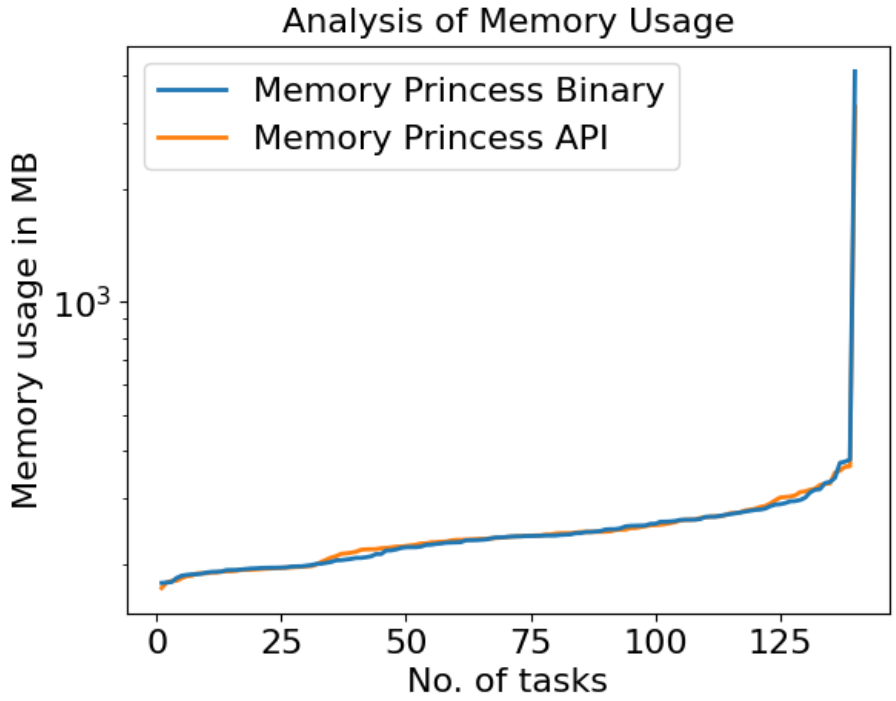

Figure 4.4: *Memory usage for intersection of correctly solved tasks with k-10 loop unrollings, sorted by seconds*

As one can see in Table 4.2, the differences between the two types of connecting the

solver are rather marginal. pBin requires slightly more memory than pAPI, whereby the percentage of additional memory required decreases as the loop unrollings increase.

Solver	**k**	**Memory usage**	**excess in %**
Princess API	k-1	23805.24 MB	
Princess binary	k-1	25191.07 MB	5,82 % excess
Princess API	k-10	36519.31 MB	
Princess binary	k-10	37170.27 MB	1,78 % excess

Table 4.2: *Comparison of memory footprint of pAPI and pBin*

However, pBin consumes less CPU time overall than pAPI as depicted in Table 4.3. Since the use of serialized strings is expected to be much slower than via a native API, this was quite unexpected [17].

Solver	**k**	**CPU time**	**excess in %**
Princess API	k-1	2055.33 s	12,22 % excess
Princess binary	k-1	1831.48 s	
Princess API	k-10	1894.44 s	1,83 % excess
Princess binary	k-10	1860.45 s	

Table 4.3: *Comparison of CPU time of pAPI and pBin*

The pAPI performs slower than expected in these test sets compared to pBin. In order to find the cause of this, further investigations need to be conducted with regard to pAPI. It might also be beneficial to connect other solvers using code-generator and parser-interpreter in order to make a comparison of the efficiency of the different APIs and to identify potential bottlenecks.

Chapter 5

Summary

5.1 Conclusion

Throughout this thesis, the capabilities of JavaSMT have been advanced by developing a solver-independent code-generator and parser-interpreter for SMT-LIB2. These features were tested by integrating the solver Princess over this new interface.
The code-generator enables the translation of JavaSMT objects into SMT-LIB2 format, providing a bridge between JavaSMT and SMT-LIB2-compliant solvers. This extension enhances the flexibility and interoperability of JavaSMT, allowing seamless integration of solvers without a public API, assuming the solvers support the SMT-LIB2 format. In addition, SMT-LIB2 formulas can now also be dumped from solvers that lack this functionality.
Conversely, the parser-interpreter facilitates the translation of SMT-LIB2 representations into JavaSMT objects. This allows JavaSMT users to manipulate formulas and apply functions before actually starting the solving process, leveraging the full spectrum of JavaSMT features. Moreover, formulas may be processed by different solvers, promoting collaboration and compatibility across different SMT solvers. In addition, the parser-interpreter grants access to any model generated by SMT-LIB2-compliant solvers by translating the models into JavaSMT objects, further enriching the utility of JavaSMT.
The evaluation of the new features has shown that they not only perform as anticipated, but even exceed expectations in terms of CPU time.
In conclusion, the contribution of this thesis enhances the JavaSMT project by adding relevant functionality, opening up new possibilities for collaboration and integration within the broader landscape of SMT solving.

5.2 Future Work

The results of the present work are rather positive, but could be improved further by taking additional measures: In order to connect further solvers using the code-generator and parser-interpreter more conveniently, the implementation of a pseudo-solver would offer great benefits in the future, as the reuse of an existing `SolverContexts` could be avoided.
Furthermore, while a native interface should be faster than communication over serialized strings in theory, this could not be observed during the evaluation, hinting on performance issues with the Princess API. Investigating these issues may boost the performance of Princess-JavaSMT significantly. Assuming a pseudo-solver would be

implemented, all solvers for which bindings exist may be additionally connected via the new interface to systematically identify bottlenecks in native APIs. Connecting additional solvers over the newly implemented interface could extend the already comprehensive feature set of JavaSMT.
Furthermore, both the parser-interpreter and the code-generator do not support all theories that JavaSMT provides. Extending the functionality of those new features to include more theories would lead to greater versatility.

Bibliography

[1] Mian Asbat Ahmad. New strategies for automated random testing. pages 2–3, 2014. URL: https://api.semanticscholar.org/CorpusID:17051666.

[2] Hafiz Munsub Ali and Daniel C. Lee. Solving the max-sat problem by binary enhanced fireworks algorithm. In *2016 Sixth International Conference on Innovative Computing Technology (INTECH)*, pages 204–209, 2016. doi:10.1109/INTECH.2016.7845071.

[3] Daniel Baier, Dirk Beyer, and Karlheinz Friedberger. *JavaSMT 3: Interacting with SMT Solvers in Java*, pages 195–208. Springer International Publishing, Cham, 2021. doi:10.1007/978-3-030-81688-9_9.

[4] Clark Barrett and Christopher Conway. Leveraging smt: Using smt solvers to improve verification; using verification to improve smt solvers. pages 2–3, 05 2012. URL: https://www.researchgate.net/publication/265674312_Leveraging_SMT_Using_SMT_Solvers_to_Improve_Verification_Using_Verification_to_Improve_SMT_Solvers.

[5] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-10575-8_11.

[6] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 184–190. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-22110-1_16.

[7] Dirk Beyer, Stefan Loewe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. 2019. doi:10.1007/s10009-017-0469-y.

[8] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/3-540-49059-0_14.

[9] Sam Buss and Jakob Nordstroem. *Proof complexity and SAT solving*, pages 233–350. IOS Press, Netherlands, 2021. doi:10.3233/FAIA200990.

[10] Cesare Tinelli Clark Barrett, Pascal Fontaine. The smt-lib standard version 2.6. pages 26–27, 2021. URL: https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf.

[11] David R. Cok. jsmtlib: Tutorial, validation and adapter tools for smt-libv2. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617, page 8 ff., 2011. doi:10.1007/978-3-642-20398-5_36.

[12] David R. Cok. The smt-libv2 language and tools: A tutorial. volume 1.2.1, pages 5–15, 2013. URL: smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf.

[13] Leonardo de Moura and Nikolaj Bjoerner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-78800-3_24.

[14] Edsger Wybe Dijkstra. In *Notes on structured programming*, page 7, 1970. URL: dl.acm.org/doi/10.5555/1243380.1243381.

[15] Liana Hadarean, Kshitij Bansal, Dejan Jovanovic, Clark Barrett, and Cesare Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. pages 680–695, 07 2014. doi:10.1007/978-3-319-08867-9_45.

[16] Joel Jones. Abstract syntax tree implementation idioms. *Pattern Languages of Program Design*, page 2, 2003. Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003). URL: hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf.

[17] Egor Karpenkov, Karlheinz Friedberger, and Dirk Beyer. Javasmt: A unified interface for smt solvers in java. volume 9971, pages 139–148, 2016. doi:10.1007/978-3-319-48869-1_11.

[18] Janelle King. Benchmark results for javasmt code-cenerator and parser-interpreter for smt-lib2 [data set], 2023. doi:10.5281/zenodo.10307339.

[19] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014. doi:10.3233/sat190101.

[20] Mizuhito Ogawa and To Van Khanh. Sat and smt: Their algorithm designs and applications. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 2, pages 83–84, 2013. doi:10.1109/APSEC.2013.118.

[21] Terence Parr. *The Definitive ANTLR 4 Reference*, pages 9–30. Pragmatic Bookshelf, 2013. URL: https://dl.icdst.org/pdfs/files3/a91ace57a8c4c8cdd9f1663e1051bf93.pdf.

[22] Terence Parr. ANTLR. https://www.antlr.org/, Online; accessed 11.2023.

[23] Terence Parr. ANTLR. https://github.com/antlr/antlr4, Online; accessed 11.2023.

[24] Terence Parr and Kathleen Fisher. Ll(*): the foundation of the antlr parser generator. pages 425–436, 2011. doi:10.1145/1993498.1993548.

[25] Philipp Ruemmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International*

Conference, volume 5330 of *Lecture Notes in Computer Science*, pages 274–289, 2008. doi:10.1007/978-3-540-89439-1_20.