LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

SOFTWARE AND COMPUTATIONAL SYSTEMS LAB

Bachelor's Thesis in Computer Science

Adding the SMT solver OpenSMT2 to the JavaSMT Framework and Evaluation using CPAchecker

Daniel Raffler

Advisor:	Daniel Baier
Supervisor:	Prof. Dr. Dirk Beyer
Submission Date:	September 13, 2023

I hereby declare that I wrote this thesis myself, marked all citations as such and indicated all sources and resources used.

Munich, September 13, 2023 Daniel Raffler

Acknowledgments

First of all I'd like to thank Prof. Dr. Dirk Beyer for enabling me to write this thesis at the chair for Software and Computational Systems at Ludwig-Maximilians-Universität München. Further thanks goes to my advisor, Daniel Baier, who was always available when there were questions and provided me with constant guidance throughout the writing of this thesis.

Special thanks also goes to Gina Franke, for her emotional support during these last few months.

Abstract

SMT solvers are software tools that can automatically determine if a set of input formulas, expressed in first-order logic, has a solution **modulo** a certain background theory. While the problem is computationally expensive, there has been incremental progress over the years and today there are many different solvers options to choose from.

The JavaSMT project has developed a library that provides a unified interface to SMT solvers and allows developers to easily switch between different solver options without major rewrites to their code. This thesis will introduce a new backend for JavaSMT that extends the framework with support for the OpenSMT2 solver. OpenSMT2 supports linear integer/real arithmetics, uninterpreted functions and array logic. More recently interpolation was added, however, the implementation is still incomplete and only some of the *logics* are supported. Nevertheless this feature sets OpenSMT2 apart from many other solver, and was one of the main reasons we wanted to include the solver in JavaSMT.

In spite of some challenges that were encountered during the developed the new backend, we were eventually able to support all of the features of OpenSMT2 in JavaSMT, and the new solver backend has performed above average in most of our benchmarks. Results were more mixed when it comes to interpolation. However, we were able to show that these results are due to missing features in OpenSMT2, and that the performance is among the fastests solvers in the test otherwise.

Contents

Acknowledgments iii

Abstract iv

1 Introduction 1

2 Related Work 5

3 Background 7
3.1 SMT 7
3.2 Interpolation 8
3.3 JavaSMT 8
3.4 OpenSMT 9

4 Implementation 10
4.1 JNI Bindings 10
4.1.1 Patched Issues 12
4.2 Example 13
4.3 Implementing the JavaSMT Interface 15
4.3.1 Implementation Classes 15
4.4 Limitations 23
4.4.1 Logic Selection 24
4.4.2 Interpolation 24
4.4.3 Models for Array Logic 25
4.4.4 Formula Simplification 25
4.5 Example 26

5 Evaluation 29
5.1 BMC 30
5.2 IMC 35

6 Conclusion 41

Abbreviations 42

List of Figures **43**

List of Tables **44**

Listings **45**

Bibliography **46**

1 Introduction

SMT solvers are software tools that can automatically determine if a set of constraints, expressed as logical formulas, is satisfiable. This makes them similar to the much more familiar SAT solvers. However, unlike SAT solvers, that are limited to propositional logic, SMT solvers are capable of handling full first-order logic, and often allow integers, reals or even bitvectors in their formulas. Examples of SMT formulas include linear inequalities over integers, such as $a + 3 > c \wedge c \geq 0$, but also formulas that that can involve functions, such as $x = y \Rightarrow f(x) = f(y)$ where f is an uninterpreted function symbol.

Most solvers only support a subset of theories, and features can vary from solver to solver. The same is true for performance, where some solvers may take much longer for certain tasks than others. Because no solver is best suited for all tasks, it would be ideal to simply switch between different solvers as needed. However, in practice this is complicated by the lack of a standard interface for SMT solvers. Every solver implements its own native API and switching from one to the other usually requires some significant rewrites in the code.

There is an international standard, called SMT-LIB2, that aims to solve this issue by defining a universal input format for SMT solvers. The format is text-based and uses a LISP-like language that can interact with the solver through special commands. Users can express SMT problems as short scripts in this language and then sent them to any compatible solver. We will cover an example of such a script at the end of this chapter, and more information can always be found directly in the standard document [1].

However, while the standard provides a good exchange format for SMT problems, it does not fully specify all solver interactions and some solver specific code is usually still necessary. For instance, models are returned as simply strings, requiring users to write their own parser code. More importantly, SMT-LIB2 does not support some important features, such as interpolation, optimization or Unsat core. Note that a proposal for interpolation support has been available[2] since 2012, but is yet to be adopted. Here the JavaSMT library tries to offer a solution by defining a common API for SMT solvers that provides support for the features missing in the SMT-LIB2 standard. The project currently supports the solvers Boolector, CVC5, MathSAT5,

[1] http://smtlib.cs.uiowa.edu/language.shtml, September 13, 2023
[2] https://ultimate.informatik.uni-freiburg.de/smtinterpol/proposal.pdf, September 13, 2023

Princess, SMTInterpol, Yices2 and Z3 through their own native backends. With this thesis we aim to add OpenSMT to the list of supported solvers. OpenSMT is a well-established solver that has featured in SMT-COMP for many years. It has shown promising performance results and supports many different *logics*, including linear real/integer arithmetics, uninterpreted functions and array logic. Combinations of these *logics* are also available, and more recently interpolation support has been implemented for some of the *logics*. This last point makes OpenSMT especially interesting to JavaSMT as there are few solvers that support the feature. Princess and SMTInterpol can both produce interpolants, but they fall short on performance, and for Z3 interpolation support was removed in version 4.8 [3]. This leaves MathSAT5 as the best option for many applications that need interpolation, and it would be great to have another alternative in OpenSMT.

It would be useful to have another alternative if MathSAT5 were to follow suit. There is a possibility that MathSAT5 will follow suit, as interpolation is not considered a very popular feature, and OpenSMT could provide another alternative In that case OpenSMT could prove to be a good alternative.

In the remainder of this thesis we will now provide some more background on SMT and the JavaSMT framework, as well as related projects, before then focusing entirely on the implementation of our new solver backend. Chapter 2 gives an overview of projects similar to JavaSMT that also aim to provide a common interface for SMT solvers. In Chapter 3 we then introduce SMT in a more formal way and provides a quick overview of the JavaSMT project and the OpenSMT solver. Chapter 4 will be focused on the implementation, and includes a detailed description of its features, along with a discussion of any issues that were encountered during development. In Chapter 5 we evaluate the performance of the new OpenSMT with CPAchecker, a the software verification framework that uses JavaSMT, and Chapter 6 provides the conclusion to this thesis with a summary of our results.

Example

An example for a SMT problem in SMT-LIB2 format can be seen in Listing 1.1. In the beginning we use *set-option* to enable interpolation, and then set the logic to QF_LIA with *set-logic*. Here QF_LIA is the name of one of the logic fragments defined by the SMT-LIB2 standard. The QF stands for "Quantifier Free" and LIA is for "Linear Integer Arithmetics". Within this fragment we are allowed to define linear inequations over integers, as well as use the regular boolean operations to build our terms. Note that is not possible to change the logic later in the program. The same is true for some of

[3]https://github.com/Z3Prover/z3/releases/tag/z3-4.8.1, September 13, 2023

```
(set-option :produce-interpolants true)
(set-logic QF_LIA)

(declare-const a Int)
(declare-const b Int)

(assert (! (< a b)
  :named f1))

(push 1)

(assert (! (< b a)
  :named f2))

(check-sat)
(get-interpolants f1 f2)

(pop 1)

(check-sat)
(get-model)
```

Listing 1.1: Example SMT problem in SMT-LIB2 format.

the solver options as these too have to be set at the beginning of the file. In lines 4 and 5 we then continue by declaring two integer variables and push the first assertion, which we will name *f1*, to the stack. This name will later be used as a reference to the formula when calculating the interpolants. We then use *push* in line 10 to add a new level to the assertion stack. After that we put the second assertion onto the stack and name it *f2*. We then run *check-sat*, which is expected to fail as the two assertions form a contradiction. With *get-interpolants* we can now print the interpolant. The formula printed will be equivalent to $b - a >= 1$ and it's an interpolant as it follows from **f1** while negating **f2**. For a formal definition of interpolant refer to Section 3.2. In line 18 we then pop the last level from stack, remove *f2* from our list of assertions. Only *f1* remains, and because of this *check-sat* in line 20 will return **true** for "satisfiable". The final line then prints a model for the remaining assertion. In our case OpenSMT chose -1 for **a** and 0 for **b**.

2 Related Work

There are a number of ways to write SMT code that is independent of the solver being used. Maybe the most obvious one is to fall back to the SMT-LIB2 format, however, as was pointed out in Chapter 1 this solution has its own shortcomings. Several projects have been developed over the year to address the issue and provide a better alternative. Some of them like PySMT, metaSMT and smt-switch take an approach similar to JavaSMT and implement a library that allow access to many solvers through a single, standardized interface. Another such library, SMT KIT, was available for C++ but now seems to have been discontinued as of 2021.

For a comparison of the features of these libraries refer to Table 2.1. We divided the table into sections, and libraries that use native bindings at the top of the table. For the table it is worth pointing out that smt-switch can also be used from Python, where it can be used from PySMT as one of its solver backend. Note that PySMT, unlike the other two options, also supports a generic backend that allows access to any SMT-LIB2 compatible solver. However, native solver backends are still needed by PySMT for some features that are not covered by the SMT-LIB2 standard. Notably this includes Interpolation, Optimization and Unsat Core, and PySMT comes with specialized backends for several solvers. For applications that do not require these features, there are a wide variety of libraries available that allow generic access to SMT solvers through the SMT-LIB2 interface.

The remainder of the table lists some of the options that are available for different programming languages. Here the feature 'Introspection' means that formulas that have been built can be traversed again to gain access to the subterms. Note that most of these libraries still only support a small subset of SMT-LIB2 solvers. For instance rsmt2 only supports Z3, CVC4 and Yices 2 out of the box, whereas for SBV the list is much longer and include ABC, Boolector, Bitwuzla, CVC4 and CVC5, DReal, MathSAT5, Yices2 and Z3. Other solvers can often be added quite easily, but some additional code to handle solver specific parsing may be required. Note that Scala SMT-LIB is somewhat special in that it provides its own domain-specific language to express SMT problems directly in Scala code. This language is an extension of the SMT-LIB2 standard and includes support for interpolation, as well as operator overloading.

Project	Language	Features				Backends								
		Interpolation	Introspection	Optimization	Unsat Core	*SMT-LIB2 format*	Bitwuzla	Boolector	CVC4+5	MathSAT5	Princess	SMTInterpol	Yices2	Z3
JavaSMT	Java													
PySMT	Python													
smt-switch	C/C++													
jSMTLIB	Java													
metaSMT	C++													
rsmt2	Rust													
SBV	Haskell													
Scala SMT-LIB	Scala													

Table 2.1: Table with features of related projects

3 Background

We will begin this chapter by giving a more formal introduction to the SMT problem and interpolation, before then moving on to JavaSMT and OpenSMT. We proving some background information on both projects and introduce their most important features. In Chapter 4 we will then cover the implementation of our new OpenSMT solver backend.

3.1 SMT

Satisfiability Modulo Theories (SMT) is the problem of determining if a given formula in first order logic (with equality) is satisfiable under a certain background theory. Here the restriction to a certain background theory makes sure that for symbols of the theory the standard interpretation of that theory must be used. For instance, with integers as our background theory we are not interested in any non-standard interpretations of the plus symbol that are not consistent with the usual laws of integer addition. Possible background theories include integers, reals or fixed sized bitvectors and combinations of these theories are also possible. The full specifications of these theories can be found directly on the SMT-LIB2 website [1]. Note, however, that most SMT solvers only support a subset of these theories. Often they also have restrictions for the language of the input formulas. For instance only linear terms may be allowed, or the formulas are expected to be quantifier free. When such restrictions on the input formulas are combined with a certain background theory, the SMT-LIB2 standard simply refers to this as a *logic*. Some of these *logics* are very common and have been given their own names in the standard. An example of this is QF_LIA, where the background theory is the theory of integers, and input formulas must not contain quantifiers or non-linear terms. Many more *logics* exist and a partial list can be found SMT-LIB2 website [2].

Not all of these *logics* are widely supported by solvers, however, over the years SMT solvers have made steady progress. This has lead to variety of real world applications, ranging from artificial intelligence to software testing. One area where SMT has been especially useful, however, is software analysis and software verification as many

[1]http://smtlib.cs.uiowa.edu/theories.shtml, September 13, 2023
[2]http://smtlib.cs.uiowa.edu/logics.shtml, September 13, 2023

problems in the field can easily be expressed as constraints. An example of this is the software verification tool CPAchecker that we will use to evaluate our own SMT solver backend in Chapter 5. In its analysis phase it constructs constraints from the input program, that are then passed on to a SMT solver that proves that the desired property has been preserved. Other verification tools follow a similar path and benefit from the higher expressiveness that SMT solvers offers when compares to SAT solver, that are limited to propositional logic.

3.2 Interpolation

Interpolants were first defined by William Craig in [Cra57] and have only been used in the context of Model Checking [McM03] and SMT solvers [CHN12] much more recently.

Given a pair of formulas (A,B), such that $A \wedge B$ is satisfiable, we call the formula I an (Craig) Interpolant if the following conditions hold:

1. $A \wedge \neg I$ is unsatisfiable
2. $I \wedge B$ is unsatisfiable
3. all symbols from I occur in both A and B

Intuitively, an interpolant provides a *local* explanation for why A is inconsistent with B. Since the interpolant "forgets" everything about A that is not needed to show this inconsistency, interpolation can also be seen as a form of abstraction.

Interpolants have found many uses, and have proven an invaluable tool in Model Checking where they can sometimes be used to avoid costly predicate abstraction. For example, in Chapter 5 we shall look at such an approach that uses interpolants to over-approximate the set of reachable states to derive loop invariants.

3.3 JavaSMT

JavaSMT is a library that aims to provides a unified interface for many different solvers. Compared to the SMT-LIB2 format it supports more features, and solvers are accessed directly through their native APIs, avoiding the need for additional parsing. JavaSMT has been developed as an open-source project an currently supports the solvers Boolector, CVC5, MathSAT5, Princess, SMTInterpol, Yices2 and Z3 through its backends. Two more backends for Bitwuzla and dReal are development, and so is support for reading and writing SMT-LIB2 files. Finally we aim to add support for the solver OpenSMT as part of this thesis. The design of JavaSMT is extendable and new

support for new solvers can be easily integrated through their own backends. Features such as interpolation and optimization that SMT-LIB2 lacks are supported by JavaSMT and can be easily accessed through the JavaSMT API. Multi-threading is supported and even multiple threads per context are possible if the solver is capable of this. The API avoids costly abstractions, and uses native solver structures wherever possible. Some features, such as automatic memory management by the Java garbage collector, require additional solver support and are available only for some of the backends.

3.4 OpenSMT

OpenSMT is an SMT solver that has been in development since 2008 at Università della Svizzera italiana (USI) in Lugano, Switzerland[3]. Its latest iteration is OpenSMT2, and this is the only version still in development. Throughout this paper when we refer simply to OpenSMT we always mean the second version of the project. OpenSMT is written entirely in C++ and supports the SMT-LIB2 input format, along with its own native interface. Many of the logic types defined by the SMT-LIB2 standard are supported, and OpenSMT can handle integers, real, uninterpreted functions and array in its logic. Support for bitvectors is incomplete and only fragments of the theory are supported. Interpolation is available, however, currently limited to QF_LIA, QF_LRA and QF_UF. In these logics OpenSMT offers several different interpolation algorithms and can change between these algorithms at runtime. This support for interpolation alone makes OpenSMT an interesting candidate for inclusion in JavaSMT.

The solver has been a participant in the SV-COMP competition for many [4] [5] years where it has recently competed on the **QF_Equality**, **QF_Equality+LinearArith**, **QF_LinearIntArith** and **QF_LinearRealArith** tracks.

[3]https://verify.inf.usi.ch/opensmt, September 13, 2023
[4]https://smt-comp.github.io/2020/participants/opensmt, September 13, 2023
[5]https://smt-comp.github.io/2023/participants/opensmt, September 13, 2023

4 Implementation

This chapter will cover the main part of this work, which is the implementation of a new solver backend for JavaSMT that adds support for OpenSMT to the library. We will start off by providing an overview of the architecture and design of OpenSMT, before then diving into the implementation of our solver backend in the following sections. Towards the end of the chapter we will then discuss some of the limitations and issues that were encountered during development.

4.1 JNI Bindings

While JavaSMT runs on the JVM, OpenSMT is written entirely in C++ with no Java interface. The first step towards integrating the two projects is therefore to develop Java bindings for OpenSMT. While this could be done by hand OpenSMT has a rather extensive interface and the task of writing Java Native Interface (JNI) wrappers for all these classes would be rather error-prone and time consuming. Instead we chose to rely on Simplified Wrapper and Interface Generator (SWIG), a tool that can semi-automatically generate the necessary bindings for us. The tool can read C/C++ headers and then automatically generates the necessary wrapper code as well as the JNI interface needed to directly access C API calls from the JVM. If the headers are written in an object oriented language like C++, as is the case with OpenSMT, SWIG will add proxy classes in Java that mirror the behavior of their C++ counterparts.

While most of the code generation is automated, some aspects require fine tuning by the developer. For this SWIG supports interface files, and we include the necessary file with all required specifications for OpenSMT as part of this project. With this the native API bindings that are needed to access the OpenSMT interface from JVM code can automatically be generated as part of the build process of the new OpenSMT backend.

We won't go into much details about the specific classes and methods of the generated bindings as they closely resemble their native counterparts. However, table 4.1 does provides an overview of the main classes in OpenSMT. Note that the table has been divided into sections, and that some of them have been given a caption. In the **Values** section we list all classes that represent terms in OpenSMT. In its implementation the solver uses several hash-tables in the *Logic* object to store these value objects. There are functions to access values in the tables, for instance to gain access to any subterms, but

Group	Class	Description
	SMTConfig	Configuration options for OpenSMT
	SMTOption	Holds the values for configuration options
	LogicFactory	Factory for *Logic*
	Logic_t	Logic types (see SMT-LIB2 standard)
	Logic	Boolean logic, arrays and uninterpreted functions.
	ArithLogic	Extends *Logic* with integers and reals.
Values	Symbol	Definition of a function symbol
	Pterm	Definition of a formula term
	SortSymbol	Definition of a sort symbol
	Sort	Definition of a sort
Refs	SymRef	Reference to a *Symbol*
	PTRef	Reference to a *Pterm*
	SSymRef	Reference to a *SortSymbol*
	SRef	Reference to a *Sort*
Vectors	VectorInt	Vector of *Ints*
	VectorPTRef	Vector of *PTRefs*
	VectorSRef	Vector of *SRefs*
	VectorSymRef	Vector of *SymRefs*
	VectorVectorInt	Vector of *VectorInts*
	MainSolver	Manages assumptions and checks satisfiability
	sstat	Status of the solver.
	Model	Model with values for variables and UFs.
	TemplateFunction	Definition of a (non-recursive) function.
	InterpolationContext	Provides interpolants.

Table 4.1: The main classes of the OpenSMT API

most functions in OpenSMT only return reference objects. All of these reference types have been grouped together under the **Reference** caption, and there is a reference type for each of the values types from the **Values** section. Internally OpenSMT represents references as simple integers that have been wrapped inside a struct. The integers are keys to the hash-tables in the *Logic* object and can be used to look up values. For the Java proxy object we made sure to implement the equals/hashcode methods to allow comparisons between references. Here we simply compare the integers from the references, and if they are the same, then the term values must also be equal. In the section **Vectors** we list vector classes for some of the reference types, as well as integers and vectors of integers. All of the classes here were autogenerated by SWIG and are, strictly speaking, not part of OpenSMT. They are however needed to handle *std::vectors* in the arguments – and sometimes return types – of some of the OpenSMT methods. Internally all the proxy classes map to native *std::vectors*, however on the Java side they implement the *AbstractList* interface and can be used just like any other list. Finally, there is one more class worth pointing out: *TemplateFunction* has been grouped along with *MainSolver* and *Model* as this is where it is really used. In OpenSMT the class can be used to define any non-recursive function, and the template can then be instantiated for different calls by expanding the definition, similar to a macro. JavaSMT does not support this, however, and the only place where the class is still used is in the OpenSMT *Model* where it wraps UF values. In OpenSMT UF values are represented as nested if-then-else expression and these values need to be wrapped inside a *TemplateFunction* object. An example for this can be found in figure 4.1, and we will cover the details later in section 4.3.1 when we will discuss the implementation of *OpenSmtModel*. For more information on the classes of OpenSMT you can refer to the JavaDoc files that have been included in the distribution of the **javasmt-solver-opensmt** backend.

4.1.1 Patched Issues

While OpenSMT supports most of the functionality required to implement the JavaSMT interface some aspects were missing or incomplete and had to be patched in the source. We will use this section to quickly run through these issues and talk about how we solved them.

4.1.1.1 Array Sorts

Support for arrays is still new to OpenSMT and the solver initially lacked a way of creating array variables through the native interface. The solver binary uses the SMT-LIB2 interface and a function for the native interface had been overlooked. After pointing this out to the developers a new API call was created and the issues has now

been fixed.

4.1.1.2 Parsing

OpenSMT is capable of reading SMT-LIB2 files and includes a parser for the format as part of its interface. However this parser can only be accessed to read and process entire SMT-LIB2 files. There is no way to read a single formula and add it to an already existing context. Since this feature is however required to fully support the JavaSMT we wrote a patch that adds the necessary calls to OpenSMT. For now this is included as part of the JavaSMT distribution, but hopefully it can be merged into OpenSMT in the future.

4.1.1.3 Sort Introspection

The OpenSMT lacks a way of inspecting sorts. Most notably there is no way to get the index or element types from an array sort. This is a minor oversight and we added our own patch to fix the issue.

4.2 Example

In order to get a better feeling for how to use the OpenSMT API, let us now return to our running example from section and translate the code to our new JNI bindings. The full code can be found in Listing 4.1: We start by getting an instance of Logic and then declare the two variables a and b of type integer. Then we build the two assertions **f1** and **f2**, before creating a new *SMTConfig* object. This new config object holds all settings for the solver, and we manually have to enable interpolation by setting the **:produce-interpolants** option. We now proceed to create a solver instance and then push the two assertions. Note that *push(PTRef)* pushes a new frame to the stack and then adds the formula. We could also have used *insertTerm(PTRef)* and *push()* (without argument) to handle those two steps separately. Once the assertions are on the stack we run *mainSolver.check()* to see if the formulas are satisfiable. Since **f1** and **f2** form a contradiction this call will return **false**, and we can now get the interpolant by first using *mainSolver.getInterpolationContext()* to get an *InterpolationContext*, and then calling *getSingleInterpolant* on this context to get the interpolant itself. The argument to this last call is a bitmask that specifies which formulas are in the A set and everything else is considered to be part of B. We simply create a new *VectorInt* and add the index 0 as we only want **f1** to be included in A. Once the interpolant has been calculated we can pop the last stack frame to remove **f2** from our assertions. Now when we run *check()* again,

```java
NativeLibraries.loadLibrary("opensmt");
NativeLibraries.loadLibrary("opensmtjava");

ArithLogic logic = LogicFactory.getLIAInstance();

PTRef varA = logic.mkIntVar("a");
PTRef varB = logic.mkIntVar("b");

PTRef f1 = logic.mkLt(varA, varB);
PTRef f2 = logic.mkLt(varB, varA);

SMTConfig config = new SMTConfig();
config.setOption(":produce-interpolants",
  new SMTOption(1));

MainSolver mainSolver =
  new MainSolver(logic, config, "");

mainSolver.push(f1);
mainSolver.push(f2);

sstat _1 = mainSolver.check();

InterpolationContext context =
  mainSolver.getInterpolationContext();

VectorInt mask = new VectorInt();
mask.add(0);

PTRef _2 = context.getSingleInterpolant(mask);

mainSolver.pop();
sstat _3 = mainSolver.check();

Model model = mainSolver.getModel();

PTRef _4 = model.evaluate(varA);
PTRef _5 = model.evaluate(varB);
```

Listing 4.1: Example SMT problem solved with the JNI bindings

the result is **true** as only **f1** remains on the stack. To get the model we use *getModel()* and then *evaluate()* to access the values of the variables.

4.3 Implementing the JavaSMT Interface

With the JNI bindings covered let us now move on to the implementation of the JavaSMT interface. JavaSMT does provide a number of abstract classes that help with the task and allow the interface to be quickly implemented for new solvers. We will make heavy use of the classes for our own solver as we implement the solver interface on top of our existing Java bindings.

OpenSMT has a fairly extensive user-interface that is provided through its C++ headers. However most of the functions that are relevant to application developers are focused on two classes: *Logic*/*ArithLogic* and *MainSolver*. *Logic* and its subclass *ArithLogic* are used to build terms that can then be passed to the solver. These two classes also provide ways of handling sorts and defining interpreted functions that can then be used as part of the terms. Their counterpart in JavaSMT is the *FormulaManager*. *Logic* itself only handles array formulas, uninterpreted functions and formulas over booleans. In JavaSMT this relates to *ArrayFormulaManager*, *UFManager* and *BooleanFormulaManger* almost directly. *ArithLogic* is an extension of the logic and adds integers and reals. Its counterpart are *IntegerFormulaManager* and *RationalFormulaMangager*. The other big class is *MainSolver*. It provides the solver stack and allows the user to check satisfiability, but there are also methods to query the model or calculate interpolants. In JavaSMT this functionality is covered by the *ProverEnvironment*.

4.3.1 Implementation Classes

In this section we will go through all the classes in the **solvers.opensmt** package, introduce their most important methods and the JavaSMT interfaces they implement. Wherever necessary we will look at the details of the implementation of individual methods. This will help to illustrate some of the challenges we encountered during the development of our OpenSMT backend and how we were eventually able to solve them.

4.3.1.1 The Solver Context Factory

The first step to add any new solver to JavaSMT is to extend the class *SolverContextFactory* in **org.sosy_lab.java_smt**, which serves as an entrypoint to the entire library. For this we first add the name of our new solver, “OPENSMT”, to the enum *SolverContextFactory.Solvers* and then extend the method *generateContext(Solvers)* with a new case

for our solver. In it we simply call the method *OpenSmtSolverContext.create(...)* which is itself a factory method and returns a new solver context for our own solver. We will get into the details of this method in the next section, but for now it's enough to know that the call simply passes on any arguments and configuration options that were given to *generateContext(..)* to our factory method.

While this is usually enough for most solver, OpenSMT required a little more work as we had to add a new method for logic selection to the library interface. For this we added the enum *Logics*, which enumerates all possible logics according to the SMT-LIB2 standard. We then extended the method *generateContext(..)* with a second parameter so that users can pass the required logic to the solver. If no logic is provided the implementation simple picks the most general logic supported.

We will return to this topic in Section 4.4.1 and explain the reasoning behind the new interface.

4.3.1.2 The OpenSMT Solver Context

OpenSmtSolverContext derives from *AbstractSolverContext*, which is part of the **basicimpl** package and extends the abstract class for our solver. *SolverContexts* are central objects in JavaSMT and represent an instance of the solver. All other important objects, such as the formula managers or the prover prover environment can be accessed from it.

In our implementation we made the constructor of the class private and added a new method *create()* that is then called from *generateContext(..)* in *SolverContextFactory* to create a new context. This is necessary to allow for some initialization of the solver backend before the actual objects are created. When create is called from *generateContext(..)* it receives all configuration options from the user as those are simply passed on. It then proceeds to make sure that the native OpenSMT libraries have been loaded by the JVM before first creating a new *OpenSmtFormulaCreator* object. The *FormulaCreator object* is in fact central for the **basicimpl** package as it provides the glue necessary to translate from native OpenSMT objects to the JavaSMT interface - and back again. We will get into the details of *FormulaCreator*, as well as our own *OpenSmtFormulaCreator*, in Subsubsection 4.3.1.3. For now it's enough to know that the formula creator is passed to the formula managers when they're created. There are in fact five of them: *OpenSmtUFManager*, *OpenSmtBooleanFormulaManger*, *OpenSmtIntegerFormulaManager*, *OpenSmtRationalFormulaManger* and *OpenSmtArrayFormulaManager*. Each of these classes implements one of the JavaSMT formula manager interfaces, and the implementation relies quite heavily on the creator object as we shall see in Subsubsection 4.3.1.5 where we will have a closer look at the classes. Once all the formula managers objects are created they're used to instantiate *OpenSMTFormulaManager*. The (general) formula manager object serves as a way to access individual formula mangers,

but also has some important methods of its own. We will have a closer look at this object in Subsubsection 4.3.1.5. With the creator and the formula manager instantiated the *create(..)* method now finally creates a new *OpenSMTSolverContext* instance that it then returns to *generateContext(..)* and consequently the user.

Most other methods in *OpenSmtSolverContext* are quite straight forward: *getSolverName()* returns *Solver.OPENSMT*, and *getVersion()* will return the version of the **javasmt-solver-opensmt** backend. We write some extra code in the SWIG headers for that as OpenSMT itself does not have a way of getting a version string. The method *newOptimizedProverEnvironment0(..)* is not implemented as OpenSMT does not support optimization, and new *ProverEnvironment0(..)* and *newProverEnvironmentWithInterpolation0(..)* return new prover environments with or without support for interpolation. Notice that interpolation is only supported for some logics and the solver will simply crash otherwise. Refer to sections Subsubsection 4.3.1.7 and Subsubsection 4.3.1.9 for a closer look at the OpenSMT prover environments. The final method *supportsAssumptionSolving()* always returns false as assumptions are not supported natively by OpenSMT. However, since the base class *AbstractSolverContext* automatically wraps the prover environment, it is still possible to use assumptions in the JavaSMT interface.

4.3.1.3 The OpenSMT Formula Creator

We will now move on to the class *OpenSmtFormulaCreator* which extends the abstract class *FormulaCreator*. As mentioned in the last section, *FormulaCreator* is central to the **basicimpl** package. Most other classes from the package use it in their implementation to translate between native solver structures and the corresponding objects from the JavaSMT interface. The abstract class takes 4 type parameters: **TFormulaInfo**, which is the type of native formulas and we set it to *PTRef*. **TType**, the type of sorts in the implementation, which is *SRef* for OpenSMT. **TEnv**, the type of the solver context, which we will set to *Logic*. And **TFuncDecl**, the type of function declarations – or *Logic* for OpenSMT. The methods of the class then operates on theses native types, and provide ways of translating objections from the solver to those of JavaSMT and back again. In our solver backend the factory method *newCreator(..)* is used to instantiate a new creator object. It takes an SMT-LIB2 logic as its argument and then creates a new native OpenSMT solver context for the selected logic. The methods *hasArrays()*, *hasUFs()*, *hasIntegers()*, *hasReals()* and *hasInterpolation()* are all used to query features of the selected logic. These helper functions are later used to safeguard certain features that may not be available under the selected logic. Without them OpenSMT simply crashes without any useful failure messages. Another helper function, this time part of the formula creator interface, is *extractInfo()*, which returns the underlying OpenSMT term of a JavaSMT formula. It's inverse encapsulate can be used to wrap a native OpenSMT

term as a JavaSMT formula. The methods *declareFunctionImpl()* and *callFunctionImpl()* are used to handle UF formulas and simply forward to the respective calls for the OpenSMT solver. Note that *callFunctionImpl()* can also be used for interpreted functions, such the operators **and** or **or** in boolean logic. The method *getBooleanVarDeclaration()* returns the *SymRef* of a OpenSMT term representing a boolean variable. Here the variable is treated as a nullary function symbol and *callFunctionImpl(..)* can be used on the returned *SymRef* to once again get the OpenSMT term. Another way of creating a new variable is to use *makeVariable()*. Here a sort can be passed as one of the arguments, so variables of any type, including arrays, can be created. In order to get construct the array type the method *getArrayType()* can be used before calling *makeVariable()*. All other sorts have constant constructors and don't need to be instantiated. Instead the interface functions *getIntegerType()* and *getRationalType()* can be used to get the OpenSMT *SRef* for integer and rational sorts respectively. The next method is *getFormulaType()*, which comes in two versions: one that returns the formula type of a native OpenSMT term and another that does the same for JavaSMT formulas. Since OpenSMT doesn't provide a way to unpack the element or index types of arrays a small patch to the code was required (see Subsubsection 4.1.1.1). The last two methods remaining are *visit()* and *convertValue()*. The first is used in the implementation of the *FormulaVisitor* pattern for OpenSMT. It checks the formula type and then gets the children of the term (if any any) before calling the matching *visit** method of the visitor. The latter converts OpenSMT terms that represent values, such as integers or booleans, to their Java representation. This is mostly needed in *OpenSmtModel* where the method is used to extract the values of variables from the native OpenSMT model.

4.3.1.4 OpenSMT Formulas

OpenSmtFormula implements the JavaSMT *Formula* interface and provides a thin wrapper for the native solver terms. The nested classes *OpenSmtArrayFormula*, *OpenSmt-BooleanFormula*, *OpenSmtIntegerFormula* and *OpenSmtRationalFormula* each extend OpenSmt-Formula and can be used for formulas with a matching type. There is a method a method *getOsmtTerm()* that can be used to get the underlying OpenSMT term, however it is only accessibly from the package. Note that internally *OpenSmtFormula* actually stores two pointer: one for the *PTRef* of the term, and another for its associated *Logic*. The is necessary for the implementation of *Formula.toString()* which uses *logic.pp(f)* to print a term f that was created with *Logic* logic. In OpenSMT all formulas are handled through references and the actual term structure is stored in logic. Because of this it's not possible to access (or print) a term without its logic.

4.3.1.5 The OpenSMT Formula Manager

The next class is *OpenSmtFormulaManager*, and it implements the *FormulaManager* interface by extending *AbstractFormulaManager* from the **basicimpl** package. The purpose of the class is mainly to serve as a way to access the other formula managers. All five of them are passed to it as arguments in the constructor and the methods *getArrayFormulaManager()*, *getUfManager()*, *getBooleanManager()*, *getIntegerManger()* and *getRationalManager()* can then be used to access them from outside the class. It's these formula managers that are then used to build the actual formulas and we will have a closer look at each of them in the next section. Other than that formula manager comes with some functionality of its own. The method *parse()* can be used to read in a formula by parsing a string, and *dumpFormula()* does the inverse by printing a JavaSMT formula as a SMT-LIB2 string. The implementation of *parse()* proved somewhat difficult as the OpenSMT API is lacking the needed features. We were able to fix the issue with a small patch to OpenSMT and the details can be found in Subsubsection 4.1.1.2.

4.3.1.6 Supported Formula Managers for OpenSMT

Our OpenSMT solver backend support five formula manager: *OpenSmtUFManager* for building uninterpreted functions, *OpenSmtBooleanFormulaManager* for boolean operations, *OpenSmtNumericalFormula* manager with its subclasses *OpenSmtIntegerFormulaManager* and *OpenSmtRationalFormulaManger* for integers and rationals respectively, and *OpenSmtArrayManager* for array logic. Each of the formula managers provides methods to build and combine formulas of its own type. For instance *BooleanFormualManager* has methods *and(..)*, *or(..)*, *not(..)* or *isFalse(..)*. The implementation is straight-forward as we can rely on the base classes from **basicimpl** to handle most of the heavy lifting. All that is left to do is implement the operations while the wrapping and unwrapping of values is handled by the base class. In most cases this simply means calling the matching OpenSMT method. For instance the implementation of OpenSmtBooleanFormulaManager.and quite simply calls Logic.mkAnd to pass the call to OpenSMT. As this is how most of the interface is implemented we will skip the discussion and simply refer the reader to the JavaSMT documentation. There a detailed list of all the supported methods and their use can be found. One exception worth noting is the method floor from *RationalFormulaManager*. As OpenSMT does not support mixed integer-real arithmetic floor isn't available and our backend will simple throw an *UnsupportedOperation* exception.

4.3.1.7 The two Prover Environments for OpenSMT

There are two prover environments in OpenSMT: *OpenSmtTheoremProver* and *OpenSmtInterpolatingProver*. They can be accessed from the *SolverContext* through *newProverEnvironment(..)* and *newProverEnvironmentWithInterpolation(..)* respectively. *OpenSmtInterpolatingProver* comes with additional operations to handle interpolation, but apart from that most of the functionality is shared. Both of these class derive from *OpenSmtAbstractProver*, which is itself a subclass of *AbstractProver* from the **basicimpl** package and implements the *ProverEnvironment* interface. In JavaSMT prover environments are used to represent the assertion stack: Users can push (and pop) assertion, check for satisfiability or read-out the model if the assertions are satisfiable. Interpolating prover environments extend this interface by adding methods to get an interpolant. In this section we will focus on the basic prover environment, which is implemented by *OpenSmtTheoremProver*, and its base class *OpenSmtAbstractProver*. Interpolation will then be the topic of our next section as we have a look at *OpenSmtInterpolatingProver*. The class *OpenSmtTheoremProver* is very simple and only consists of a constructor and a single method, *addConstraintImpl(..)*. As already mentioned, new instances of *OpenSmtTheoremProver* are created by *newProverEnvironemnt(..)* in *OpenSmtTheoremProver*, and most of the arguments to the constructor are simply passed on to the base class. This includes the formula manager and the formula creator, along with the shutdown notifier and configuration options. An exception here is the random number seed, which is passed to the static method *getConfigInstance(..)* from the base class. Here we also set interpolation to **false** and the method then returns a new *SMTConfig* object that can be passed to the constructor of the superclass. Other than that *OpenSmtTheoremProver* only has one more method, *addConstraintImpl(..)*. It overrides an abstract method from *OpenSmtAbstractProver* and takes a *PTRef* as an argument. The method then adds the formula to the assertion stack and returns an index – or in case of *OpenSmtTheoremProver* simply **null**. As the class doesn't use interpolation an index to address the formulas on the stack is not needed. The method *addConstraintImpl(..)* is called by *addConstraint(..)* from the *BasicProverEnvironment* interface, which is implemented by *OpenSmtAbstractProver*. Here the implementation of *addConstraint(..)* in the base class updates the internal assertion stack and converts the argument to a native *PTRef* before then calling the abstract method *addConstraintImpl(..)*. Both *OpenSmtTheoremProver* and *OpenSmtInterpolatingProver* have to override this abstract method to add their own implementation. The base class *OpenSmtAbstractProver* is also where the rest of the *BasicProverEnvironment* interface is implemented. For this the class uses a member variable **assertionStack**, of type *Deque<List<PTRef»*, that keeps track of all levels of the assertion stack. In the constructor the stack is initialized and then a new *MainSolver* instance is created. MainSolver is an OpenSMT classes and handles operations on

the assertion stack much like *OpenSmtAbstractProver* does for JavaSMT. Most of the *BasicProverEnvironment* interface can then be handles by simply relaying the calls to the *MainSolver* instance. Whenever formulas are added or removed from the assertion stack the internal variable **assertionStack** also needs to be updated. There are two more methods worth pointing out: *getModel()* and *isUnsat()*. In *getModel()* we create a new *OpenSmtModel* and then register it as a new evaluator. The method *registerEvaluator(..)* that is used here is part of an interface in *AbstractProver* and simply makes sure that the model is closed when it is no longer valid. The class *OpenSmtModel* implements *Model* from JavaSMT and will be covered in the next section. Its constructor, however, requires us to have a list of all formulas on the assertion stack. This will then be used by *OpenSmtModel* to get a list of variables, which is needed to build the model. It is for this reason that we had to use **assertionStack** to keep track of the constraints on the stack. We then simply use the *getAssertedFormulas()* to concatenate the levels and get a full list of assertions. This would not be possible without the internal variable as OpenSMT does not provide any way to read out the formulas on the assertion stack. The other method that required a little more work is **isUnsat()**. Here the issue is that – during creation – OpenSMT does not always check if formulas are possible under the selected logic. It is, for instance, always possible to create array formulas, even when the logic doesn't allow for it. Only when the users calls check to see if the formulas are satisfiable will OpenSMT throw an exception. However, the error message can still be rather cryptic. It is for this reason that we added another method, *checkCompatability(..)*, to our implementation. This method is called from *isUnsat* – just before the constraints are passed on to the solver – and makes sure that all the formulas are compatible with the selected logic. If not an exception is throw with the error message containing the required features that the logic doesn't support.

4.3.1.8 The OpenSMT Model

We will now have a quick look at *OpenSmtModel*, before then moving on to the interpolating prover environment in the next section. *OpenSmtModel* derives from *AbstractModel* in the **basicimpl** package and implements the *Model* interface from JavaSMT: The class itself is fairly simple with most of the functionality being contained in the constructor. New instances are created by *getModel()* in *OpenSmtAbstractProver*, and the arguments to the constructor include the prover environment itself, the formula creator, and a full list of all the formulas on the assertion stack. Upon initialization the class will then create a list of assignments that maps all variables of the model to their respective values. This list is created by first getting a list of all variables and (uninterpreted) function symbols in the assertions and then extracting their values from the native OpenSMT model. This is straight forward for variables, but uninterpreted functions

```
(define-fun g ((x1 Int) (x2 Int)) Int
  (ite (= 5 x1)
    (ite (= 3 x2)
      7
      (ite (= 1 x2)
        2
        0))
    (ite (= 3 x1)
      (ite (= 3 x2)
        2
        0)
      0)))
```

Figure 4.1: Example of an UF value in OpenSMT

require a little more work as OpenSMT represents their values as nested if-then-else expressions.

For an example of this encoding refer to figure 4.1. Here you can find the value of an uninterpreted function *g* as it might be returned by OpenSMT. In JavaSMT the same function is represented quite differently, and instead of the nested if-then-else statements a simple list of value assignments is used to define the function. For the function from the example we would get the assignments *g(5,3)=7*, *g(5,1)=2* and *g(3,3)=2*. It is possible to extract these values from the original if-then-else form by (recursively) descending the chain and collecting the values. We handle this task in a helper function "unroll" and with this we can create value assignments for all the variables of the model. The function *asList()* from the *Model* interface can then be used to get this list.

4.3.1.9 The Interpolating Prover Environment for OpenSMT

Let us now move on the the last implementation class remaining, *OpenSmtInterpolatingProver*. Just like *OpenSmtTheoremProver* from Subsubsection 4.3.1.7 it derives from *AbstractProver*, and much of the functionality is shared. Since *OpenSmtInterpolationProver*, however, implements the more general *InterpolatingProverEnvironment* it supports additional calls for interpolation. These come along with already familiar calls from *BasicProverEnvironment* that are mostly concerned with handling the assertion stack, satisfiability and model generation. Just like for its sibling, *OpenSmtTheoremProver*, the constructor of *OpenSmtInterpolatingProver* simply passes on its arguments to the

base class. There are now three more arguments to the constructor, that are used to select the interpolation algorithm for propositional logic, QF_UF, and QF_LRA (and QF_LIA) respectively. For a list of options refer to the documentation on the OpenSMT website [1]). Within the constructor we use the helper method *getConfigInstance()*, where we set interpolation to **true** and then pass the random number seed, along with these 3 parameters to the function to create a new *SMTConfig* instance. This object, together with the other arguments for the constructor, is then passed to the base class constructor. With the constructor covered, we still need to implement the abstract method *getConstraintImpl(..)* from the *AbstractProver* interface. Just as for *OpenSmtTheoremProver* we simply call *MainSolver.insertFormula(..)* from the native OpenSMT classes to add the formula to the assertion stack. However, here we can't just return null. Instead we calculate the (total) number of assertions on the stack and then return it as the index. The index identifies the assertion on the stack and will later be used to define **A** and **B** sets for interpolation. To generate an interpolant the method *getInterpolant()* is used. The argument is set of indices (as returned by addConstraint) and defines the **A** set. Everything else will be considered part of **B**. In the implementation we simply call *MainSolver.getInterpolationContext()* to get an OpenSMT interpolation context, and the use the method *getInterpolant(..)* on it. Internally OpenSMT encodes the **A** and **B** sets a little differently, but this conversion is already handled in the SWIG header and won't be of concern here. There is one more way to generate interpolants and that is through the method *getSeqInterpolants(..)*. The method takes a list of index sets and then calculates a sequence of interpolants. In our implementation we have to first convert the arguments OpenSMT expects a sequence of index sets, with every set containing its immediate predecessors. We accomplish this by simply creating prefixes of the list of sets in the argument for *getSeqInterpolants(..)* and then merging those prefixes before passing them as arguments to *getPathInterpolants(..)* from OpenSMT to calculate the actual interpolation sequence.

4.4 Limitations

While most of the JavaSMT interface maps directly to its OpenSMT counterpart, there were some aspects that caused issues. Most of these problems are down to differences in design, but some expose actual limitations of the solver. We will use this section to talk about the problems that were encountered and how we tried to fix them.

[1]https://verify.inf.usi.ch/interpolation, September 13, 2023

4.4.1 Logic Selection

Unlike most solvers OpenSMT requires the user to set a specific logic before any formulas can be checked or even built. This was an early design choice during the development of OpenSMT and can't be changed now without significant work. While the design is in line with the SMT-LIB2 standard, most solvers today don't have such a requirement, and JavaSMT even lacks a mechanism for the user to set a specific logic.

This leads to obvious difficulties with the implementation of the JavaSMT interface. One solution is to delay the logic selection by first caching all the formulas internally and then copying them over to the solver as they're actually used and the required logic can be decided. However this not only complicated the implementation but also leads to some significant performance overhead. We also considered running several solver instances, each with a different logic, in parallel while building the formulas. As more formulas are encountered some of the instances would drop out as their logic is not general enough and does not support the features used in the formulas. Eventually only a few of these logic instances would be left and we could then pick any one of them to calculate the result. However, similar to the first approach, this design would have caused a significant performance overhead: Even though most of the instances should quickly drop out, there simply are to many logics that have to be considered for this approach to be efficient.

We consulted with the developers of OpenSMT and they were able to offer us another option by adding support for QF_AUFLIRA to the solver. The implementation of this logic is incomplete as the it only supports integers OR reals, but not mixed terms with both integers and reals. However for our application this is still sufficient as the new logic subsumes all other logics supported by OpenSMT. This means we can use it as a default logic, and for most problems our backend will now simply use QF_AUFLIRA without any user selection. Only in some instances user intervention is still needed, and we will have a look at some of those examples in the next section.

4.4.2 Interpolation

While the new logic QF_AUFLIRA should work for all problems there are still some instances where one might want to set a specific logic. OpenSMT has specialized solvers for several of its sublogics, so pricking one of them were the problem allows it could greatly improve performance.

More importantly there is one case where setting the logic manually is actually still required in OpenSMT: and that is when interpolation is used. Unfortunately OpenSMT currently does not support interpolation for the new logic QF_AUFLIRA – or indeed

any mixed logic. Only QF_LIA, QF_LRA and QF_UF are supported [2]. This limitation is something the developers of OpenSMT would like to work on, but it would require significant work and a quick fix seems unlikely. At least for now it is therefore necessary to manually select the logic whenever interpolation is to be used.

Since JavaSMT so far has been lacking a mechanism for logic selection we had to extend the interface with some new functions. Note that this new feature also comes in handy for some of the other solvers that struggle with certain problems that require a combination of different features. Not all of these combinations are generally supported in logic ALL and selecting a more restrictive logic manually can often help to solve the issue.

4.4.3 Models for Array Logic

Another limitation worth pointing out is that OpenSMT does not support model generation for array logic. It is still possible to get values for the non-array variables in the model, even when arrays are used in the formula. However, if we try to get the value of a variable with array type the solver simply returns an abstract value with no definition. Other solvers return a nested “store expression” as a concrete representation of the array value. After talking to the developers it seems that the abstract value returned is mostly due to an oversight an should not be used [3]. Because of this we disabled model generation for arrays logic in our backend. The method *getModel()* in *OpenSmtAbstractProver* will simple throw an exception whenever array variables are found in the assertions.

4.4.4 Formula Simplification

One of the design choices for OpenSMT is that formulas are simplified very early, and while they are still being built. For instance OpenSMT automatically cancels out double not terms or simplifies linear equations when they are first created. This however also means that there is no way of getting the original formula. Instead of the term for “a - b > b” we would simply get “0 > 2b - a” and “(not (not a))” will reduce to just “a” when the second not is applied. While this is fine for most applications as the formulas returned are still equivalent, there are some instances where it does matter. Most notably these simplification will affect any function that uses visitors to traverse through the formula. The result may then often be counter-intuitive. For instance on would expect that the free variables in “a + b = a” are “a” and “b”. However, as OpenSMT automatically applies simplification the formula become “b = 0” and “b”

[2]Note that the logic QF_UF gets further limited by JavaSMT as sort declarations are not supported.
[3]https://github.com/usi-verification-and-security/opensmt/issues/630, September 13, 2023

would be returned as the only free variable in the expression. Unfortunately it is not possible to turn these simplifications off in OpenSMT. We could write an abstraction layer and use our own format to store terms before they are handed off to OpenSMT. However, this would require a signification amount of work and inevitably lead to some runtime overhead as formulas have to be stored twice. In the end we decided against such an option, and it is now up to the user to keep in mind that the formulas returned might not be entirely identical to the formulas that were created.

4.5 Example

Let us now return to our running example from previous chapters and show how the same problem can be solved in JavaSMT. Listing 4.2 shows a code snippet, and the full project files are publicly available online [Raf23]. In our example we start by creating three objects: *Configuration*, *LogManager* and *ShutdownManager*. *Configuration* is a class from the **sosy-lab.commons** package and allows us to pass configuration options to the solver classes, either directly from the command-line or by reading a separate configuration file. *LogManager* comes from the same package and can be used to log all calls to JavaSMT, which is useful for collecting statistics and debugging, among other things. *ShutdownManager* is the last of the common classes and it's main use is to send shutdown request to the solver when it's taking to long and the run should be aborted. In our example we don't need any of this functionality so we simply create default instances and then use them to create our context factory. With this factory we can now create a new solver context. This is where the solver backend and the logic have to be selected and we choose OpenSMT and QF_LIA for our example problem. Now with the context available we can first get the formula manager, and is use that to get the integer formula manager that we will use to build the assertions. We then create our variables **a** and **b** by using *makeVarriable()* from the integer formula manager. Then we build our two assertions by using the methods *and()* and *lessThan()* from the two formula managers. Once complete we can open a new prover environment by calling *context.newProverEnvironmentWithInterpolation(..)*. In JavaSMT the class *ProverEnvironment* is used to manage the assertion stack, and *InterpolatingProverEnvironment* adds additional calls to handle interpolation. We then proceed to use *prover.push()* to add our assertions to the assertion stack. The return values of the calls are labels for our formulas and we need to save them for later. If we now check for satisfiability the result is 'unsat' as **f1** and **f2** form a contradiction, and we can then get the interpolant by calling *getInterpolant(..)* on our prover environment. The argument **mask** defines the A set and we have to use the labels returned by the push function earlier here. Since we only want **f1** to be in A we simply pick **labelA**

```
1  Configuration config = Configuration.defaultConfiguration();
2
3  SolverContextFactory factory = new SolverContextFactory(config,
4    BasicLogManager.create(config),
5    ShutdownManager.shutdown.getNotifier());
6
7  SolverContext context = factory.generateContext(
8    Solvers.OPENSMT, Logics.QF_LIA);
9
10 FormulaManager mgr = context.getFormulaManager();
11 IntegerFormulaManager imgr = mgr.getIntegerFormulaManager();
12
13 IntegerFormula varA = imgr.makeVariable("a");
14 IntegerFormula varB = imgr.makeVariable("b");
15
16 BooleanFormula f1 = imgr.lessThan(varA, varB);
17 BooleanFormula f2 = imgr.lessThan(varB, varA);
18
19 InterpolatingProverEnvironment<Integer> prover =
20   (InterpolatingProverEnvironment<Integer>)
21   context.newProverEnvironmentWithInterpolation(
22     ProverOptions.GENERATE_MODELS);
23
24 Integer labelA = prover.push(f1);
25 Integer labelB = prover.push(f2);
26
27 boolean _1 = !prover.isUnsat();
28
29 ImmutableList<Integer> mask = ImmutableList.of(labelA);
30 BooleanFormula _2 = prover.getInterpolant(mask);
31
32 prover.pop();
33 boolean _3 = !prover.isUnsat();
34
35 Model model = prover.getModel();
36
37 BigInteger _4 = model.evaluate(varA);
38 BigInteger _5 = model.evaluate(varB);
```

Listing 4.2: Example SMT problem solved with JavaSMT.

as parameter. Once the interpolant has been printed we can now pop the last stack frame again. This will remove **f2** from the assertions. With only **f1** remaining the call to *prover.isUnsat()* now returns “false” and we can get a model with *prover.getModel()*.

5 Evaluation

In this final chapter we will examine how well the new OpenSMT performs in a real-world application. For this we will rely on CPAchecker, a software verification tool that uses JavaSMT as one of its supported backends. CPAchecker is a mature project and a participant in the yearly SV-COMP competition that aims to compare the performance of different verification tool with each other. Together with the benchmarking tool benchexec[1], it is well suited to the performance of our new backend to that of other existing JavaSMT backends under realistic conditions. For this comparison we used the same benchmark setup that is used in the SV-COMP competition. That is, all benchmarks were executed on Intel Xeon E3-1230 processors, and the resource limits were set to 8 cores with 15GB of memory and a 900s timeout. With this setup we will now run two different analysis algorithms in CPAchecker. Benchexec will be used to measure the performance of the solvers, and we will use the test set from the SV-COMP competition as input for both runs. More precisely, we used the following tasks with the **unreach-call** specification:

- ReachSafety-Arrays
- ReachSafety-BitVectors
- ReachSafety-ControlFlow
- ReachSafety-ECA
- ReachSafety-Floats
- ReachSafety-Heap
- ReachSafety-Loops
- ReachSafety-ProductLines
- ReachSafety-Recursive
- ReachSafety-Sequentialized

[1]https://github.com/sosy-lab/benchexec, September 13, 2023

- SoftwareSystems-AWS-C-Common-ReachSafety
- SoftwareSystems-DeviceDriversLinux64-ReachSafety

The options *cpa.predicate.encodeFloatAs=INTEGER* and *cpa.predicate.encodeBitvectorAs=INTEGER* were set globally for all the solvers. These settings will lead to more incorrect results in the analysis, however, they are needed as some of the solvers don't support floats or bitvectors. For the **BMC** benchmark we also use the option *cpa.predicate.useArraysForHeap=false* since OpenSMT does not support model generation for array logic (see Subsection 4.4.3). Also note that we decided not to include the Boolector and Yices2 in our benchmarks as they lack needed features and would have required different configuration options, which complicates the comparison with other solvers.

All of the benchmarks were executed with version 2.2.1-svn-43817 of CPAchecker, and the versions for the solvers can be found in Table 5.1. Finally, the version of JavaSMT was 3.14.3-137-gbe28b75f. It's based on the 3.14 branch of JavaSMT and contains the final prerelease version of our new OpenSMT backend.

Solver	Version
CVC4	1.8-prerelease
MathSAT 5	5.6.8
OpenSMT	2.5.2-7-gbb0e019d
Princess	2022-11-03
SMTInterpol	2.5-1242-g5c50fb6d
Z3	4.12.1.0

Table 5.1: Table of solver versions for the benchmark.

5.1 BMC

Bounded Model Checking (BMC) is a technique in software verification that unrolls loop up to **k** time to look for a violation of the guaranteed properties for the program. While this is by definition incomplete, as the properties may still be violated later, in practice a lot of bugs in software development can be found this way.

5.1.0.1 Testrun bmc-k1

The results for the **BMC** benchmark with **k**=1 can be found in Table 5.2. All data for the benchmarks in this chapter is also publicly available online [Raf23]. For this test run we see that MathSAT5, OpenSMT and SMTInterpol all return roughly the same number of

Solver	Correct Results			Incorrect Results		
	∑	true	false	∑	true	false
CVC4	1019	650	369	492	12	480
MathSAT5	1121	652	469	542	12	530
OpenSMT	1121	652	469	541	12	529
Princess	1040	650	390	374	12	362
SMTInterpol	1121	651	470	547	12	535
Z3	1022	599	423	386	9	377

Table 5.2: Result table for the **BMC** benchmark with k=1.

correct results. Only CVC4, Princess and Z3 lack behind as they score less than 1000 correct results. Also note that there were no large differences between solvers when it comes to the number of incorrect results. Ideally such results should be as low as possible, especially for the "incorrect true" category. For such runs the analysis claims that the program is correct – even though it has a bug. The other category, "incorrect false", is less problematic as it contains runs where the analysis found a bug in a program that is really correct. Such results should still be rare as the lead to unnecessary work, but they don't affect the reliability of the analysis in the same way. Note that most of the incorrect results, both true and false, were caused our use of the options *cpa.predicate.encodeFloatAs=INTEGER* and *cpa.predicate.encodeBitvectorAs=INTEGER*. With these setting CPAchecker will use integers to emulate bitvectors and floating point variables. However, this technique can provide only an approximation as integers can grow to arbitrary sizes, whereas real bitvectors and floats are of fixed-width and have to deal with overflow conditions. Because of this the analysis is no longer sound and we start to see a lot more incorrect results when these options are used. Once we change these options and allow CVC4, MathSAT5 and Z3 to use their bitvector and floating point suppport we don't just see much fewer incorrect results for these solvers, but there are also some changes when it comes to the number of correct results, and Z3 scores much higher while CVC4 loses ground. However, since we are trying to create an equal playing field for all solvers, and some don't support these features, we simply left the option disable for all solvers.

Let us now have a look at the runtime performance of the solvers. In the "quantile chart" in Figure 5.1 the individual runs are ordered along the x axis according to their runtime. On the y axis you can see the exact run time of the tasks in seconds. Note that this axis is logarithmic, and that is starts at 1 second for better readability. Ideally the lines in the plot should be as long as possible, while also staying as low as possible. Longer lines mean that the solver was able to correctly handle more of the tasks from

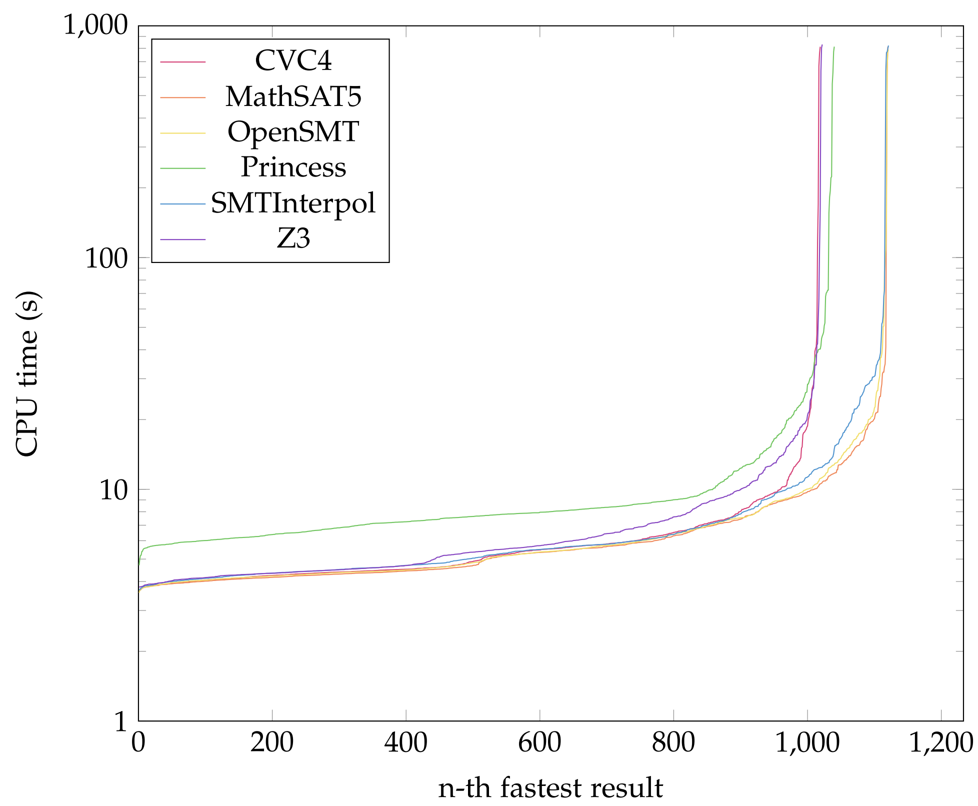

Figure 5.1: Quantile plot for the **BMC** benchmark with k=1.

Solver	**Correct Results** ∑	true	false	**Incorrect Results** ∑	true	false
CVC4	1165	855	310	444	12	432
MathSAT5	1955	860	1095	668	12	656
OpenSMT	1890	857	1033	594	12	582
Princess	1406	830	576	378	12	366
SMTInterpol	1896	858	1038	652	12	640
Z3	1906	805	1101	678	9	669

Table 5.3: Result table for the **BMC** benchmark with k=10.

the benchmark set, while also staying below the timeout limit of 15 minutes. Lower lines correspond to lower runtimes, and we're especially interested in the steepness of the curve towards the end as this gives us some hints about scalability to larger problems.

Looking at the graph we can again see the same division from the table. MathSAT5, OpenSMT and SMTInterpol are in the lead, where as CVC4 and Princess and Z3 lack further behind. Note that Princess is slower than other solvers across the field, but scales better than CVC4 and Z3 until it eventually evertakes those solvers.

5.1.0.2 Testrun bmc-k10

Let us now increase the number of loop iterations that the analysis will consider. For this we set **k** to 10 and then rerun the test set with the **BMC** algorithm. Table 5.3 shows the result, and they show a similar pattern as in the last benchmark. CVC4 and Princess still struggle, and the gap between them and the other solver seems to have grown even larger. Interestingly enough Z3 performed much better this time, and is now on par with MathSAT5, OpenSMT and SMTIntpol, which all scored around 1900 results.

We can confirm these results by having a look at the quantile plot in Figure 5.2 Here we observe that – with the exception of Z3 – the general ranking between solvers has not changed, however, the gaps between them seem to have grown larger. This may be explained by the larger formulas that are generated for **k**=10 as these could easily exesserbate any performance issues that were already visible for smaller **k**s. Looking at the performance of CVC4 we can see how the solver seem to run into performance issues with harder problems, even though it was able to keep up well with other solvers initially. Conversely Princess showed higher runtimes throughout the entire test set, but then scaled much better than CVC4 and was once more able to overtake it towards the end.

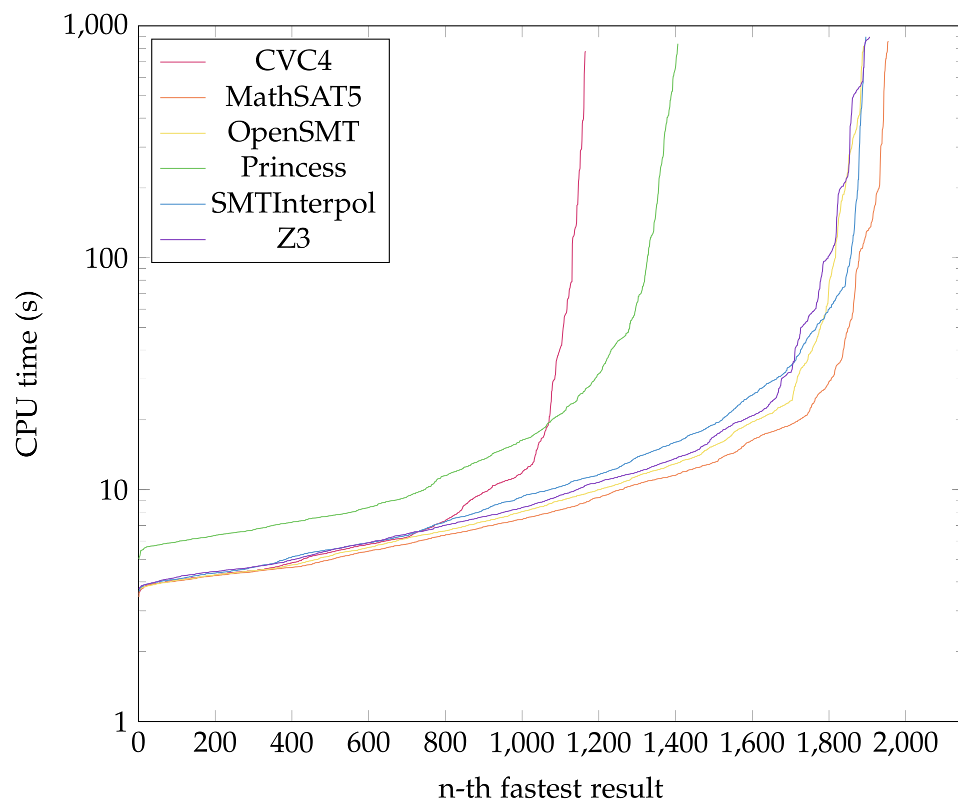

Figure 5.2: Quantile plot for the **BMC** benchmark with k=10.

Solver	Correct Results			Incorrect Results		
	$\sum$	true	false	$\sum$	true	false
MathSAT5	2788	1517	1271	963	17	946
OpenSMT	1617	1015	602	54	14	40
Princess	1849	1131	718	548	16	532
SMTInterpol	1979	1080	899	877	11	866

Table 5.4: Result table for the **IMC** benchmark.

5.2 IMC

While the last section was concerned with BMC, we will now move on to Interpolation-based Model Checking (IMC). The algorithm we will use in this benchmark is relatively new in the context of software verification and has been described in [BLW22]. It intertwines BMC with interpolation by first generation a formula expressing the set of states reachable in k transitions. If this formula is satisfiable an "error state" has been reached and the program is incorrect. Otherwise the analysis moves on to calculating a sequence of interpolants for the individual steps of the transition relation. If this sequence converges to a fixed point, all possible runs are covered and the program is correct. Otherwise the analysis returns to the **BMC** step and unrolls the program further. The algorithm offers an alternative to more traditional approaches such as k-Induction and has been fully implemented in CPAchecker. For us it is of interest mainly as a benchmark as the algorithm makes heavy use of interpolation. Interpolation is also one of the key features of OpenSMT, and it would be interesting to compare its performance in this benchmark to that of other solvers. Unfortunately many solvers lack support for interpolation, and out of all the solvers with JavaSMT backend only MathSAT5, OpenSMT, SMTInterpol and Princess support the feature. In this benchmark ran the **IMC** from CPAchecker on the same SV-COMP test sets that were listed at the beginning of this chapter. Note that we chose to remove the option *cpa.predicate.useArraysForHeap=false* from out configuration, as OpenSMT does not support interpolation for Array logic anyway, and it no longer was needed.

The results from Table 5.4 clearly show that MathSAT5 performed by far the best, scoring almost 2800 correct results, as all other solvers failed to pass the 2000 result mark. OpenSMT seems to have fared particularly poorly as it only returned 1617 correct results. Notice however, that it also only returned 54 incorrect results, which is a fraction of the number of incorrect results returned by the other solvers.

Looking at the quantile plot in Figure 5.3 can see this first impression table confirmed. Princess scaled a little better than SMTInterpol, and was quickly able to overtake OpenSMT before all three solvers were clearly beat by MathSAT5. The performance of

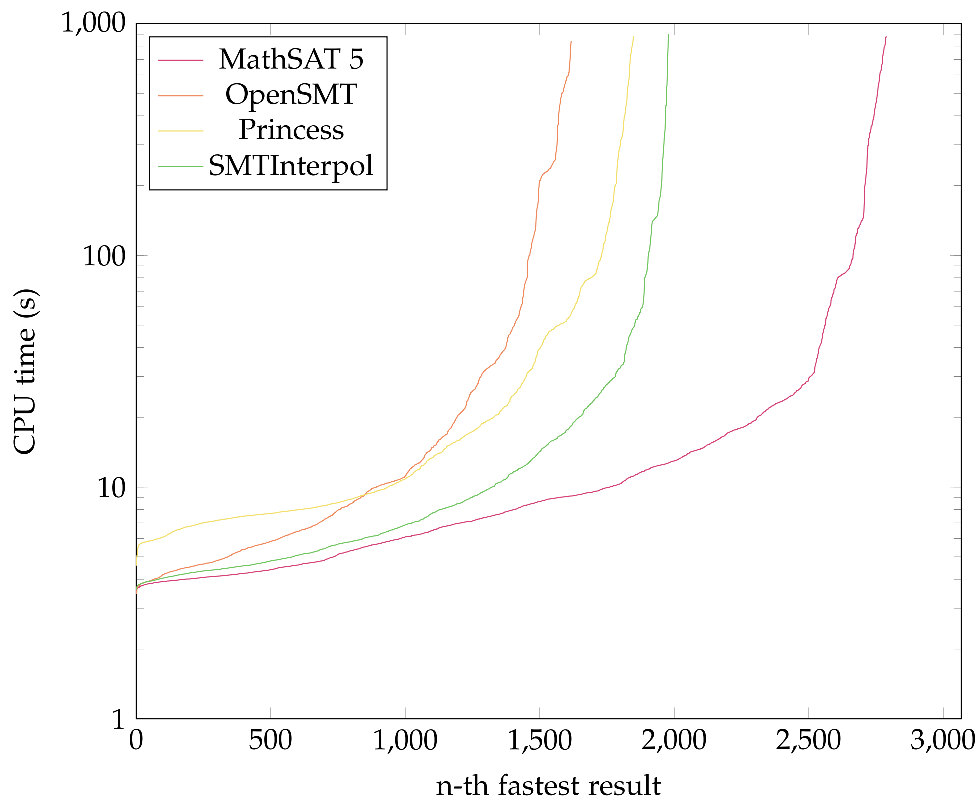

Figure 5.3: Quantile plot for the **IMC** benchmark.

Solver	Correct Results ∑	true	false	Incorrect Results ∑	true	false
MathSAT5	1639	1009	630	51	11	40
OpenSMT	1617	1015	602	54	14	40
Princess	1112	773	339	43	11	32
SMTInterpol	990	677	313	52	5	47

Table 5.5: Result table for the **IMC** benchmark without *Error* results.

OpenSMT looks particularly bad in the plot, however, on closer inspection this may only be partially true: Out of the 7238 tasks of the **IMC** test set for all solvers the majority of the results is neither **correct** nor **incorrect**, but Instead these runs usually ended in a *TIMOUT*, and or even an *ERROR* result. For OpenSMT the proportion of *ERROR* results is much higher, and if we look closer at the log files of those runs it turns out that almost all of these *ERROR* results were due to missing features, such as arrays and uninterpreted functions. Here OpenSMT simply aborts the calculation as the formulas require features that are unsupported if interpolation is also being used. Since *ERROR* results, however, don't show up in the plot, this can be a bit misleading. We tried to counterbalance this by creating another version of the benchmark that removes all tests where OpenSMT returned an *ERROR* result. Table 5.5 shows that the image has now shifted, and we can confirm this by looking at the quantile plot in Figure 5.4 OpenSMT is now competitive with MathSAT5 as the other two solvers are less affected, even though SMTInterpol seems to have lost some ground to Princess. What this clearly shows is that the poor results for OpenSMT earlier are caused by missing features, and not due to any underlying performance issues. In fact, we were able to show that OpenSMT performs very well, and is second only to MathSAT5, on the tasks that it does support. It could of course be argued that limiting the benchmark to tasks that worked for OpenSMT is not entirely fair either. Notice, however, that we don't get the same effect if we try to apply the same trick for other solvers and filter out their *ERROR* results. These of these solvers barely return any *ERROR* results, and because of this filtering those tasks out does not affect the ranking of the solvers. One argument that remains is that OpenSMT may have an unfair advantage as it is the only solver running in QF_LIA, and doesn't have to handle the overhead that comes with supporting a more general logic. In the end the only way to get a definitive result is to redo the test with all solvers set to QF_LIA – which required extended the new logic selection interface to the other solvers – or to wait until more general support for interpolation is added to OpenSMT.

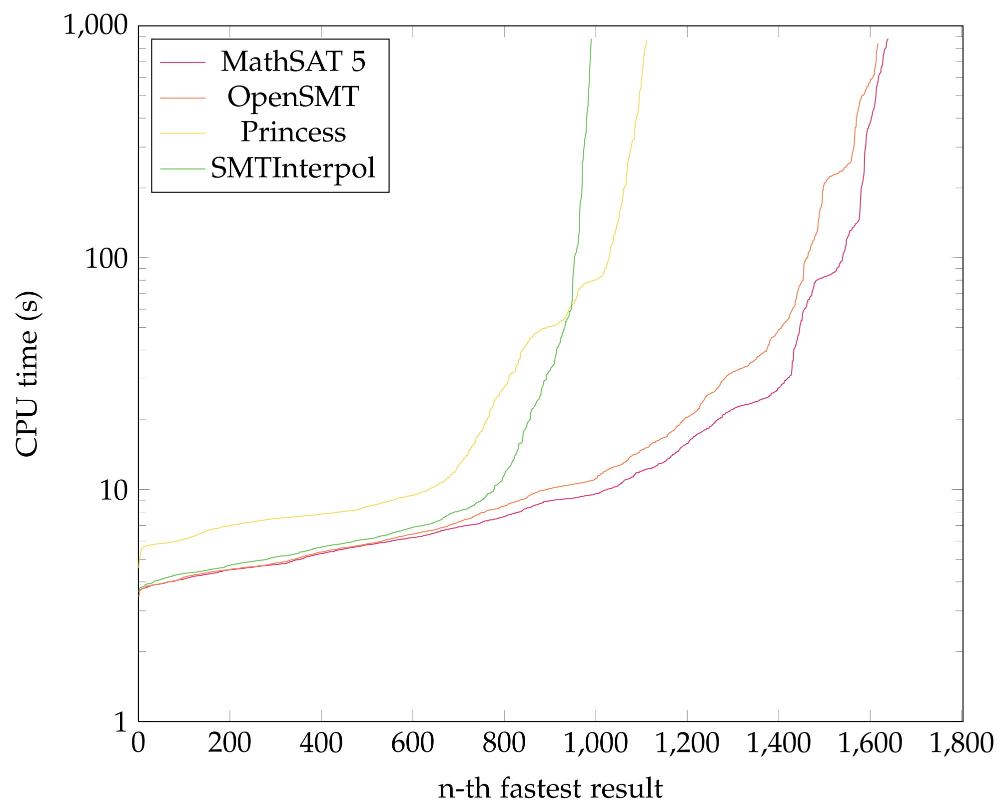

Figure 5.4: Quantile plot for the **IMC** benchmark without *ERROR* results.

Solver	Correct Results			Incorrect Results		
	∑	true	false	∑	true	false
Decomposed	1608	1006	602	56	16	40
Dual Decomposed	1595	994	601	55	15	40
Dual Farkas	1601	1000	601	55	15	40
Farkas	1617	1015	602	54	14	40
Flexible	1616	1015	601	55	15	40

Table 5.6: Result table for the **IMC** benchmark with different interpolation algorithms.

5.2.0.1 Results for Alternativev Interpolation Strategies

Before we end this chapter we'd like to introduce one more benchmark that demonstrates the performance of some of the interpolation strategies available in OpenSMT. For this we ran multiple instances of OpenSMT, and selected different interpolation strategies through the **solver.opensmt.algLIA** option. With this we then repeated the same **IMC** benchmark from the last section. The results can be found in Table 5.6 and we included the quantile plot in Figure 5.5. Here it can be seen that the performance difference between strategies are generally quite small. This is not entirely surprising as most of the calculation time is not spent on the actual interpolation step. In fact, with OpenSMT the overhead of building a proof is entirely separate from the actual interpolation strategy, which is only concerned with extracting the interpolant. Do note, however, that the y axis in the plot is logarithmic, and that the differences in performance seem to get bigger for the more difficult task. In fact some of these tasks do timeout for the slower strategies, showing that the difference can sometimes be significant. Luckily the fastest strategy, **Farkas**, is also the default option for OpenSMT. It showed the best performance across the board with only **Flexible** coming close. The **Dual*** strategies tend to take significantly longer on large tasks, and **Decomposition** lies somewhere in the middle.

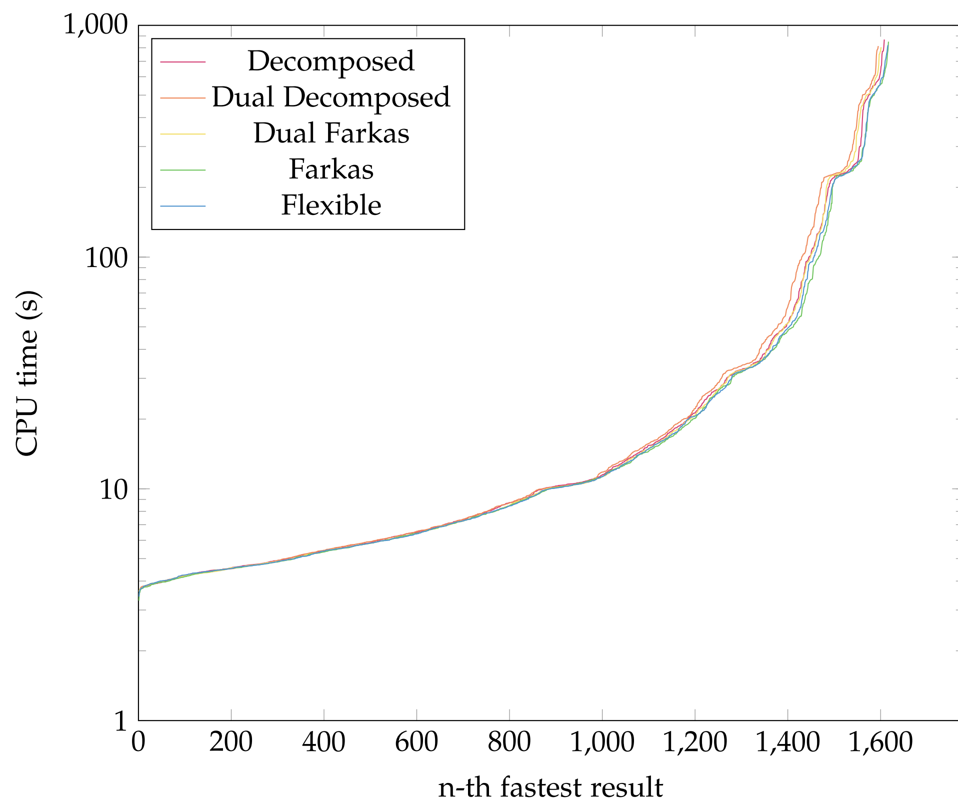

Figure 5.5: Quantile plot for the **IMC** benchmark with different interpolation algorithms.

6 Conclusion

In this thesis we set out to extend the JavaSMT framework with support for the solver OpenSMT as an additional backend. OpenSMT was mainly chosen because of its support of interpolation, which make it an interesting option for many research projects. For our task we first had to develop JNI binding for the native OpenSMT API as the solver itself is written in C++ and does not come with its own Java support. The semi-automated tool SWIG was of great help with this and we were able generate the necessary JNI headers and Java proxy classes to make the solver available to us. The next step was then to implement the JavaSMT interface on top of those native bindings. With the help of the abstract classes from the **bascimpl** package this task was relatively straight forward. We did encounter some issues, mostly due to limitations and bugs in the solver, and all of those issues have been documented as part of this thesis. With a little help from the developers of OpenSMT most of the problems could be sorted out relatively quickly. Some, however, required a little more ingenuity. Most importantly the lack of support for interpolation in **all** logics required us to introduce a new interface for logic selection to the JavaSMT framework. We belief that this feature can be useful to other solver too, and are looking forward to seeing it implemented in their backends. With all these fixes in place we were able to finish the JavaSMT interface and ran several benchmarks with CPAchecker to test the new backend under real-world conditions. The results for OpenSMT looked quite promising as the solver performed above average in the **BMC** benchmark set that tests general performance. For interpolation the picture is more mixed as OpenSMT fails to return a result for many of the task of the benchmark. The issue here is however with the lack of support for certain features like arrays or uninterpreted functions while interpolation is used. If we limit the benchmark to tasks that only require integers the results start to look much more promising for OpenSMT as it is now on par with MathSAT5, the leader of the field. It would certainly be interesting to redo these tests if OpenSMT gets full support for interpolation. In the mean time OpenSMT still provides a welcome addition to the JavaSMT portfolio as one of the more mature solver options.

Abbreviations

BMC Bounded Model Checking

IMC Interpolation-based Model Checking

JNI Java Native Interface

SWIG Simplified Wrapper and Interface Generator

USI Università della Svizzera italiana

List of Figures

4.1 Example of an UF value in OpenSMT 22

5.1 Quantile plot for the **BMC** benchmark with k=1. 32
5.2 Quantile plot for the **BMC** benchmark with k=10. 34
5.3 Quantile plot for the **IMC** benchmark. 36
5.4 Quantile plot for the **IMC** benchmark without *ERROR* results. 38
5.5 Quantile plot for the **IMC** benchmark with different interpolation algorithms. 40

List of Tables

2.1 Table with features of related projects 6

4.1 The main classes of the OpenSMT API 11

5.1 Table of solver versions for the benchmark. 30
5.2 Result table for the **BMC** benchmark with k=1. 31
5.3 Result table for the **BMC** benchmark with k=10. 33
5.4 Result table for the **IMC** benchmark. 35
5.5 Result table for the **IMC** benchmark without *Error* results. 37
5.6 Result table for the **IMC** benchmark with different interpolation algorithms. 39

Listings

1.1 Example SMT problem in SMT-LIB2 format. 3

4.1 Example SMT problem solved with the JNI bindings 14
4.2 Example SMT problem solved with JavaSMT. 27

Bibliography

[BLW22] D. Beyer, N.-Z. Lee, and P. Wendler. *Interpolation and SAT-Based Model Checking Revisited: Adoption to Software Verification*. 2022.

[CHN12] J. Christ, J. Hoenicke, and A. Nutz. "SMTInterpol: An interpolating SMT solver." In: *International SPIN Workshop on Model Checking of Software*. Springer. 2012, pp. 248–254.

[Cra57] W. Craig. "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory." In: *The Journal of Symbolic Logic* 22.3 (1957), pp. 269–285.

[McM03] K. L. McMillan. "Interpolation and SAT-based model checking." In: *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15*. Springer. 2003, pp. 1–13.

[Raf23] D. Raffler. *Dataset for "Adding the SMT solver OpenSMT2 to the JavaSMT Framework and Evaluation using CPAchecker"*. Zenodo, Sept. 2023. DOI: 10.5281/zenodo.8342375.