Ludwig-Maximilians-Universität München
Institut für Informatik

Integration des SMT-Solvers Boolector in das Framework JavaSMT und Evaluation mit CPAchecker

Bachelorarbeit

Daniel Baier

Abgabedatum: 18.11.2019

1. Betreuer: Prof. Dr. Dirk Beyer
2. Betreuer: Karlheinz Friedberger

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

$Unterschrift$:

$Ort, Datum$:

Danksagung

Zunächst möchte ich mich bei Prof. Dirk Beyer für die Chance bedanken, meine Bachelorarbeit am Software and Computational Systems Lab der Ludwig-Maximilians-Universität München schreiben zu dürfen.
Einen besonderen Dank möchte ich Karlheinz Friedberger aussprechen. Auf seine Unterstützung und seinen guten Rat konnte ich mich immer verlassen.
Weiterhin bedanke ich mich bei Adrian Leimeister, Mathias Both und Anke Dickhoff für ihre Unterstützung und Ausdauer meinen Redestrom zu ertragen, sowie zusätzlich Luisa Mauerer für ihre seelische Unterstützung.
Zuletzt will ich noch Jutta und Roland Baier danken, ohne die ich nicht die Möglichkeit gehabt hätte meine Bachelorarbeit zu schreiben.

Zusammenfassung

SMT-Solver sind ein beliebtes Werkzeug in der computergestützten Verifikation von Programmen und künstlicher Intelligenz. Aufgrund der verschiedenen Theorien, die in diesen Solvern Verwendung finden, ist es von Vorteil eine breite Auswahl an verschiedenen Solvern zur Verfügung zu haben, möglichst ohne für jeden Solver eine neue API lernen zu müssen. Diese Eigenschaften werden von JavaSMT verkörpert, welches ein Java basierendes Framework mit gemeinsamer API für SMT-Solver bietet.
In dieser Arbeit wird, neben der Einführung in das Thema SMT Solving, das Projekt JavaSMT sowie der auf Bitvektoren spezialisierte SMT-Solver Boolector beleuchtet. Anschließend wird die Implementierung von Boolector in JavaSMT beschrieben, inklusive seiner Besonderheiten und Hürden. Zuletzt wird eine Evaluation mithilfe des formalen Software Verificationstools CPAchecker durchgeführt, um Boolector gegenüber der bereits in JavaSMT vorhandenen Solver zu bewerten. Boolector erzielte gute Resultate im Vergleich zu den anderen mit Bitvektoren arbeitenden SMT-Solvern.

Abstract

SMT solvers are a popular tool to use in computer-guided verification of computer programs and artificial intelligence. Because of the numerous used Theories in these solvers, it is useful to have a broad spectrum of different solvers and, if possible, avoid the downside of having to learn a new API for each of them. Exactly these attributes are embodied by JavaSMT, a Java based framework for SMT solver that shares a common API between them.
This thesis first describes the necessary knowledge to understand SMT solving,
JavaSMT and the SMT solver Boolector, which is specialized in bit-vector theory. The main objective lies on the implementation of Boolector into the JavaSMT framework, in company with the unique characteristics and problems that arose. At last, the formal software verification tool CPAchecker is used, to evaluate Boolector in regards to the already implemented SMT solvers in JavaSMT. Boolector achieved good results in comparison to the other SMT solvers capable of using bitvectors.

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface / Programmierschnittstelle
BMC	Bounded Model Checking
GMP	GNU Multiple Precision Arithmetic Library
GUI	Graphical User Interface
JNI	Java Native Interface
KNF	Konjunktive Normalform
QF_ABV	Closed Quantifier-Free Formulas over the Theory of Bitvectors and Bitvector Arrays
QF_AUFBV	Closed Quantifier-Free Formulas over the Theory of Bitvectors and Bitvector Arrays extended with free Sort and Function Symbols
QF_BV	Closed Quantifier-Free Formulas over the Theory of Fixed-Size Bitvectors
QF_UFBV	Unquantified Formulas over Bitvectors with Uninterpreted Sort Function and Symbols
SAT	Satisfiable / Erfüllbar
SIG-SEV	Segmentation fault
SMT	Satisfiability Modulo Theories
SMT-COMP	The International Satisfiability Modulo Theories Competition
SMT-LIB	The Satisfiability Modulo Theories Library
SV	Software Verification
SWIG	Simplified Wrapper and Interface Generator
UNSAT	Unsatisfiable / Unerfüllbar

1 Einleitung

In der heutigen Welt bildet die Mathematik mit ihren vielen Facetten einen integralen Teil in der Lösung von Problemen und Weiterentwicklung von zahlreichen Technologien. Leider sind wir Menschen in der Regel nicht in dazu in der Lage komplexere mathematische Probleme im Kopf zu lösen, weshalb wir zahlreiche Hilfsmittel, wie Taschenrechner oder Computer, nutzen. Ein Hilfsmittel, das erst durch Computer und geschickt angewandte Mathematik zur Verfügung steht, sind Satisfiability Modulo Theories Solver. Diese SMT-Solver können bestimmen, ob prädikatenlogische Entscheidungsprobleme lösbar sind und oft sogar Modelle für mögliche Lösungen bereitstellen. So kann man, mit ihnen als Basis, z.B. neu entwickelte Programme verifizieren oder auf ihr Verhalten bei unterschiedlichsten Eingaben testen und dadurch Fehler finden. In der Natur der Informatik liegt, dass unterschiedlichste Programmiersprachen verwendet werden, unter anderem für hardwarenahe Programmierung oder die Möglichkeit der direkten Speicherverwaltung. Aufgrund dessen ist es wenig verwunderlich, dass es SMT-Solver in einer Vielzahl von Programmiersprachen gibt. Weiterhin hat jeder Solver eine andere Benutzerschnittstelle, mit deren Hilfe man auf ihn zugreift. Ein Beispiel hierfür ist Boolector, welcher in der Sprache C implementiert ist, aber in C und Python Benutzerschnittstellen bietet. Da unterschiedliche Solver auf unterschiedliche Anwendungsgebiete spezialisiert sind, ist es manchmal sinnvoll mehrere von ihnen zu verwenden. Um nun das Problem mehrerer unterschiedlicher Programmiersprachen und Benutzerschnittstellen von Solvern zu beheben, gibt es Projekte wie JavaSMT, welche es sich zum Ziel gesetzt haben mehrere SMT-Solver in einem einzigen Framework, mit uniformer Benutzerschnittstelle, zusammenzufassen. Der Kern dieser Arbeit bildet die Integration von Boolector in JavaSMT, mithilfe der von Java zur Verfügung gestellten Funktionalität zur Integration anderer Programmiersprachen. Nachdem sich diese Arbeit zunächst mit verwandten Veröffentlichungen befasst, werden im Anschluss SMT-Solver im Allgemeinen und Boolector im Speziellen vorgestellt. Nach einer Einführung in das JavaSMT Projekt wird in Kapitel 4 die Implementierung von Boolector in selbiges beschrieben. Anschließend wird auf die zugehörigen Tests eingegangen, welche die Funktionalität des neuen Solvers bestätigen, bevor die neue Implementierung, gegenüber den anderen Solvern die bereits in JavaSMT verfügbar sind, in CPAchecker evaluiert wird. Zuletzt werden die Ergebnisse dieser Arbeit zusammengefasst und ein Ausblick auf weitere Arbeiten im Zusammenhang mit dieser gegeben.

2 Verwandte Arbeiten

Der Kern dieser Arbeit ist die Implementierung eines neuen SMT-Solvers in das JavaSMT[KFB16] Framework, dessen Ziel es ist eine uniforme Schnittstelle für alle integrierten Solver zu bieten. Dieses Ziel wird ebenfalls von ScalaSMT[CS17] verfolgt, welches bereits einige Solver implementiert. Das Projekt bietet eine domänenspezifische Sprache, um Nutzer von solverspezifischer API zu isolieren, die leicht von dem bekannten SMT-LIB2 Standard abweicht, mit dem Ziel diese zu verbessern. Es wird besonderen Wert auf die Benutzbarkeit der Bibliothek gelegt, was aber auch den Nachteil mit sich bringt, dass regelmäßige Änderungen der API die Benutzbarkeit von alten Programmen einschränkt.
Ein weiterer Vertreter der Frameworks ist MetaSMT[HFF+11], das sowohl eine C als auch eine Python-API zur Verfügung stellt und sowohl SMT als auch SAT-Solver benutzt. MetaSMT bildet die Eingabe, der auf SMT-LIB2 basierende API, mithilfe eines statischen, zur Compilezeit generierten, Syntaxbaumes ab. Dieser wird in eine Zwischenschicht weitergereicht, die entweder die Eingabe direkt an den Backend-Solver weiter gibt oder zuvor einige Anpassungsmöglichkeiten bietet. So steht z.B. BitBlasting zur Verfügung um Formeln mit Bitvektoren zu vereinfachen, bevor diese von einem beliebigen SAT oder SMT-Solver der Backend-Schicht letztendlich gelöst werden. Neben Multithreadunterstützung steht auch eine Server-Client Architektur zur Verfügung, die genutzt werden kann, um einen Cluster zur Verarbeitung der Eingaben zur Verfügung zu stellen. Auch der Performance Overhead des Projekts wird so niedrig gehalten wie nur möglich.
Wiederum stellt PySMT[GM15] in Python eine Bibliothek zur Verfügung die, neben integrierten nativen Solvern, auch die Möglichkeit bietet einen SMT-LIB2-konformen Solver zu wrappen. Das Projekt stellt eine Solver unabhängige API zur Verfügung, die durch einen Konverter an den gewählten Solver weitergegeben wird und bietet damit, ohne Mehraufwand, einen einfachen Weg mehrere Solver zu nutzen, um das gleiche Problem zu lösen. Jedoch entsteht durch den Transfer der Formeln zwischen verschiedenen Kontexten ein erhöhter Speicheraufwand.

	Boolector[1]	Z3[2]	MathSAT5[3]	CVC4[4]	Princess[5]	SMTInterpol[6]	Yices[7]
JavaSMT[8]	ja	ja	ja	ja	ja	ja	in Arbeit
ScalaSMT[9]	nein	ja	ja	ja	nein	ja	ja
MetaSMT[10]	ja	ja	nein	ja	nein	nein	nein
PySMT[11]	ja	ja	ja	ja	nein	nein	ja

Tabelle 1: Zusammenstellung der wichtigsten, verfügbaren SMT-Solver in den jeweiligen Frameworks

[1]https://github.com/Boolector/boolector, 24.10.2019
[2]https://github.com/Z3Prover/z3, 24.10.2019
[3]http://mathsat.fbk.eu/index.html, 24.10.2019
[4]https://github.com/CVC4/CVC4, 24.10.2019
[5]http://www.philipp.ruemmer.org/princess.shtml, 24.10.2019
[6]https://github.com/ultimate-pa/smtinterpol, 24.10.2019
[7]https://github.com/SRI-CSL/yices2, 24.10.2019
[8]https://github.com/sosy-lab/java-smt, 27.10.2019
[9]https://github.com/regb/scala-smtlib, 24.10.2019
[10]https://github.com/agra-uni-bremen/metaSMT, 24.10.2019
[11]https://github.com/pysmt/pysmt, 24.10.2019

3 Grundlagen und Motivation

Zunächst wird auf die Gründe für die Wahl des Solvers, sowie die nötigen Grundlagen eingegangen, um die spätere Implementierung verstehen zu können. Dies umfasst SMT-Solver im allgemeinen, Kommunikationsstandards, Boolector im speziellen, sowie die Idee hinter JavaSMT und dessen aktueller Stand.

3.1 SMT-Solver

SMT-Solver versuchen prädikatenlogische Entscheidungsprobleme zu lösen. Das heißt, sie versuchen eine Eingabeformel zu prüfen, bis sie entweder SAT (= Satisfiable bzw. Erfüllbar) oder UNSAT (= Unsatisfiable bzw. nicht Erfüllbar) ist. Sollte die Formel nicht entscheidbar für den Solver sein, wird stattdessen UNKNOWN zurückgegeben.

3.1.1 SAT-Solver

Die Basis von SMT-Solvern sind oft sogenannte SAT-Solver. Diese Solver sind ebenfalls dazu gedacht aussagenlogische Formeln zu lösen, in dem sie versuchen logische Formeln auf wahr oder falsch zu reduzieren. Im Vergleich zu SMT-Solvern müssen die Eingabewerte und Formeln für SAT-Solver jedoch komplett in Aussagenlogik formuliert werden, also boolesche Variablen mit logischen Operatoren. Viele SAT-Solver verlangen, dass ihre Eingaben in konjunktiver Normalform, kurz KNF, formuliert werden. Zum einen wird die KNF aus historischen Gründen verwendet, zum anderen weil viele Algorithmen zum Prüfen auf Erfüllbarkeit diese als Eingabe erwarten. Jede beliebige Eingabeformel kann in die KNF umgewandelt werden und sowohl Erfüllbarkeit als auch Allgemeingültigkeit werden beibehalten. Die Umwandlung einer aussagenlogischen Formel in die KNF geschieht durch das Aufspalten einer Formel mithilfe von Algorithmen wie der Zeitin-Transformation[Zei70]. Versucht man jetzt allerdings, z.B. mit einer Wahrheitstabelle, die Eingabeformeln auf Erfüllbarkeit zu prüfen, muss man mindestens eine Belegung der Variablen finden, für die die Formel erfüllbar ist. Der Aufwand hierfür ist abhängig von der Menge an zu prüfenden Variablen.

$(A \vee (B \wedge C))$
$(A \vee x) \wedge (\neg x \vee B) \wedge (\neg x \vee C) \wedge (\neg B \vee \neg C \vee x)$ //KNF mit x für $(B \wedge C)$

A	B	C	x	A ∨ x	¬x ∨ B	¬x ∨ C	¬B ∨ ¬C ∨ x	SAT?
w	w	w	w	w	w	w	w	SAT
f	w	w	w	w	w	w	w	SAT
w	f	w	f	w	w	w	w	SAT
f	f	w	f	f	w	w	w	-
w	w	f	f	w	w	w	w	SAT
f	w	f	f	f	w	w	w	-
w	f	f	f	w	w	w	w	SAT
f	f	f	f	f	w	w	w	-

Tabelle 2: Umwandlung der Formel $(A \vee (B \wedge C))$ in KNF und finden aller möglichen, erfüllbaren Belegungen mithilfe einer Wahrheitstabelle

Weiterhin ist erkenntlich, dass der Aufwand potentiell schnell ansteigt, sollte die Menge an Variablen steigen. Das Problem ist, dass Erfüllbarkeitsprobleme NP-Vollständig[Coo71] sind und somit im schlimmsten Falle *vermutlich* nicht polynomiale Laufzeiten aufweisen. Dementsprechend kann die Laufzeit im schlimmsten Falle exponentiell sein, da die Komplexitätsklasse NP eine Teilmenge von EXPTIME ist. Schon am Beispiel in Tabelle 2 ist zu erkennen, dass das Umwandeln in die KNF weitere Variablen einführt, die geprüft werden müssen; zusätzlich zur Ausführung der Wahrheitstabelle.
Es gibt jedoch einige clevere Implementierungen, die weit bessere Laufzeiten ermöglichen. So nutzt zum Beispiel der DPLL[IR06] Algorithmus Backtracking um die Erfüllbarkeit von Formeln in KNF zu entscheiden und bietet im Normalfall eine weit bessere Laufzeit. Weitere Beispiele für Optimierung von SAT-Solvern und ihrer Ergebnisse über die Jahre sind mit Hilfe von Wettkämpfen, wie SAT Live![Ber] und der SMT-Competition[LH19], findbar. So lässt sich die Entwicklung von SAT-Solvern wie MiniSat[12], PicoSAT[13] und Lingeling[14] verfolgen, die in Boolector Verwendung finden.

[12]https://github.com/niklasso/minisat, 25.10.2019
[13]http://fmv.jku.at/picosat/, 25.10.2019
[14]https://github.com/arminbiere/lingeling, 25.10.2019

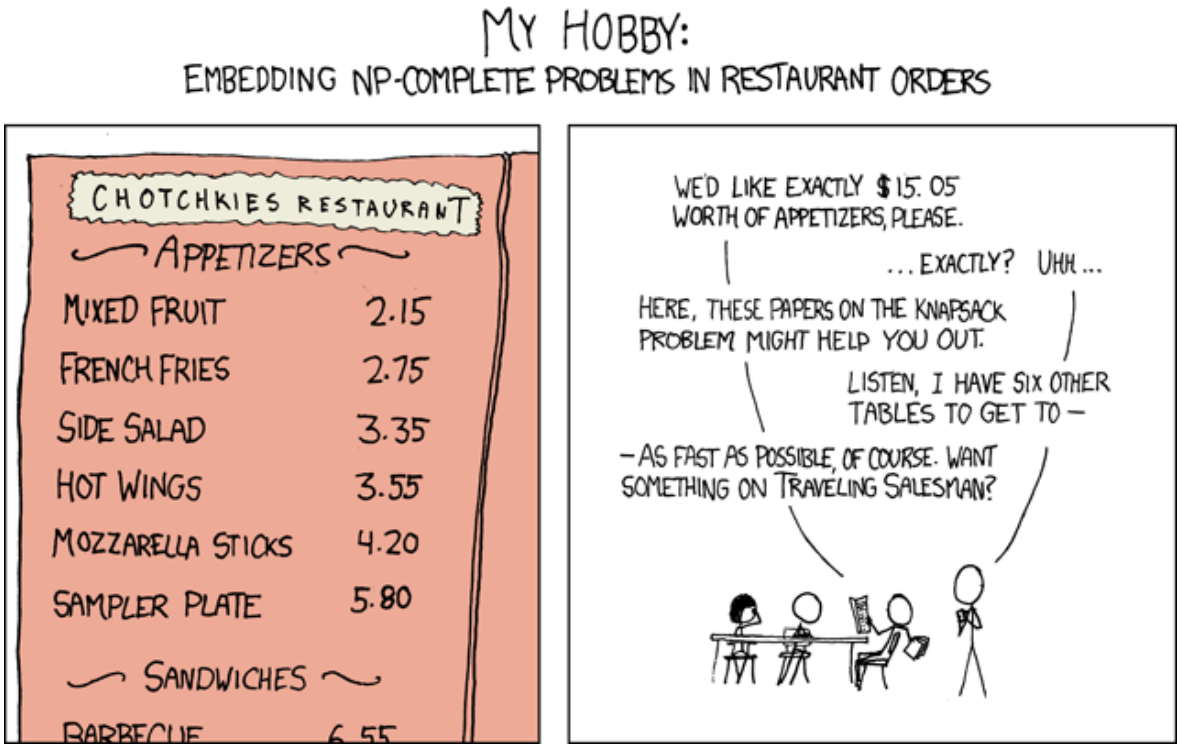

Abbildung 1: Beispiel eines klassischen, nicht trivialen SMT typischen Problems[15]

3.1.2 SMT-Solver

Das Problem von SAT-Solvern ist, dass die Eingabeformeln in Aussagenlogik formuliert werden müssen, was in der Realität oft problematisch ist. Versucht man beispielsweise ein Entscheidungsproblem, welches Zahlen enthält, etwa ein Sudoku, mithilfe von SAT-Solvern zu lösen, stößt man schnell an seine Grenzen, da auch Zahlen in Aussagenlogik kodiert werden müssten. Dieses Problem gehen Satisfiability Modulo Theories Solver, mithilfe ihres höheren Abstraktionsgrades, direkt an.
Wie der Name bereits vermittelt, implementieren die jeweiligen Solver Theorien, wie zum Beispiel Integers, Arrays und Uninterpreted Functions. Dies ermöglicht eine signifikant einfachere Eingabe der gegebenen Probleme, im Kontext der Realitätsnähe, im Vergleich zum Zwang der Übersetzung in Aussagenlogik bei SAT-Solvern. Grundlegend verarbeiten SMT-Solver die ihnen übergebene Formel so, dass sie entweder durch einen SAT-Solver lösbar werden, genannt Eager Approach, oder integrieren eine vom DPLL Algorithmus typische Suchlogik, um theoriespezifische Solver, sogenannte T-Solver, nutzen zu können, die in ihrer Domäne schneller und effizienter die Eingabeformel lösen. Letzteres wird auch der Lazy Approach genannt und zum Beispiel von dem SMT-Solver Z3 verwendet. Ein typisches Beispiel für den Eager Approach wäre Boolector, was zum Umwandeln von Eingabeformeln der Theorie der quantorenfreien Bitvektoren fixer Größe in Aussagenlogik Bit-blasting verwendet, die am weitesten verbreitete Prozedur zum Umwandeln von Bitvektoren in Aussagenlogik. Der Vorteil

[15] https://xkcd.com/287, 29.10.2019

der Eingabe bei SMT-Solvern steht also dem Problem der Übersetzung in für den SAT bzw. T-Solver verständliche Logik gegenüber. Die Eingabe bei SMT erfolgt mithilfe von prädikatenlogischen Formeln, teilweise inklusive Quantoren, aufgeteilt in Theorien, die wiederum verschiedene Logiken haben können. Der Solver muss dazu in der Lage sein diese zu unterscheiden, verschiedene Theoreme konfliktfrei zusammenzufassen und diese anschließend lösen. Der Unterschied von SAT zu SMT-Solvern ist vergleichbar mit Programmiersprachen auf höherem Level, wie Java, gegenüber hardwarenahen Assemblersprachen. Aufgrund der direkten Kontrolle über die Art und Weise in der die Formeln gelöst werden, können in SAT und T-Solvern Probleme weit effizienter gelöst werden, aber dafür ist es weit schwieriger diese zu programmieren, bedenkend der bereits vorgestellten Probleme dieser Solver. Neben der Unterscheidung von Erfüllbarkeit und Unerfüllbarkeit besitzen die meisten SMT-Solver noch die Fähigkeit ein Modell zu bilden, das mögliche Belegungen der Eingabeformeln abbildet, sofern diese erfüllbar sind.
Zwei wichtige und häufig vorkommende Theorien sind Arrays und Uninterpreted Functions. Letztere haben keine feste Interpretation bezüglich ihrer Operation, können also modular mithilfe von Konstanten und Variablen aufgebaut und dann auf Erfüllbarkeit geprüft werden. Sie gelten als besonders wichtig, da in SMT-Solvern viele Theorien auf sie reduziert werden können. Arrays funktionieren ähnlich der bekannten Datentypen in der Informatik. Man kann Werte ablegen und wieder zurückgeben und damit prüfen ob Arrays, z.B. in Algorithmen, erfüllbare Ergebnisse liefern. Außerdem gibt es bei manchen Solvern die Möglichkeit Assumptions, also Annahmen, einfließen zu lassen. Damit kann die Fähigkeit des Solvers, die Erfüllbarkeit der Eingabe festzustellen, verbessert werden, ohne die Erfüllbarkeit zu behindern, sollten die Assumptions nicht hilfreich sein.
Um Formeln zu prüfen, müssen die einzelnen Terme, die sie aufbauen, geprüft werden. Hierfür werden diese auf einen Stack mit den zu prüfenden Formeln gelegt, der bei einem Aufruf zum Prüfen abgearbeitet wird. Es gibt viele weitere Zusatzfunktionen, wie die Möglichkeit Eingabeformeln vereinfachen zu lassen oder sich die unerfüllbare Teilmenge der Eingabe geben zu lassen, genannt UnsatCore. Die hier aufgeführten Punkte sind wichtig, sowohl für das Verständnis von JavaSMT, Boolector und der Arbeit als Ganzes, zeigen jedoch deutlich, dass eine präzise, solverunabhängige Sprache gebraucht wird um über SMT reden zu können. Außerdem wird eine Metrik gebraucht, nach der man Solver unterscheiden kann sowohl in Bezug auf Fähigkeiten als auch Leistungsfähigkeit.

3.2 SMT-LIB Standard und SMT-Competition

Die grundlegende Idee hinter Mathematik ist es, die Welt so genau wie möglich zu beschreiben. Um diesen Prozess zu beschleunigen hat man sich auf ein gewisses Vokabular

geeinigt, unsere mathematische Schreibweise. Im Feld der Informatik stellt man jedoch sehr schnell fest, dass es eine nahezu unendliche Anzahl an Schreibweisen gibt, seien es Programmiersprachen oder die von Programmen benutzten Programmierschnittstellen. Es ist wenig verwunderlich, dass es, besonders in einem mathematischen Teilbereich, Versuche gibt die unterschiedlichen Eingaben und Ausgaben der SMT-Solver zusammenzuführen, um sie allgemein verständlich zu machen, ohne für jeden eigenen Solver eine neue Schreibweise zu lernen.
Genau hier setzt der SMT-LIB2[BST+10][BT17] Standard an. Er stellt eine uniforme Schreibweise bereit, die es erlaubt SMT-Solver, und ihre Ein- und Ausgaben zu verstehen, stellt aber auch ein Benchmarking Framework bereit um Solver miteinander zu vergleichen. Im Folgenden wird nicht der gesamte Standard aufgeführt, sondern nur ein Teil, ausreichend um den Rest der Arbeit und die aufgeführen Beispiele zu verstehen. Weiterführende Informationen sind auf der Website des Standards zu finden.
Es wurden bereits einige Theorien wie Arrays oder Uninterpreted Functions vorgestellt. Um jedoch eine noch genauere Abgrenzung in und zwischen diesen zu erhalten, werden die Theorien noch einmal in Logiken unterteilt. Dadurch wird eine Basis geschaffen, um später genaue Benchmarks von Solvern in gleichen Kategorien zu ermöglichen und eine allgemeine Unterscheidung der Fähigkeiten von Solvern zu haben, auch wenn dies nicht immer von den Entwicklern genutzt wird oder genutzt werden kann.

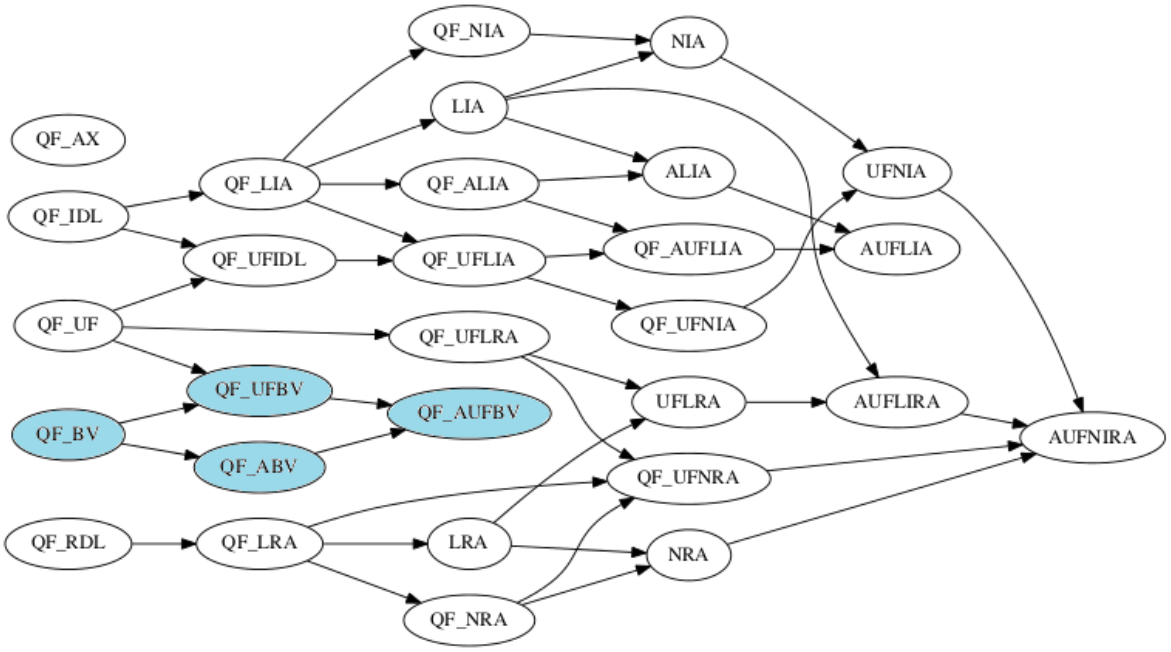

Abbildung 2: Verwandtschaft der wichtigsten Logiken im SMT-LIB2 Format[16]

In Abbildung 2 sieht man die wichtigsten Logiken und ihre Verwandtschaft. Die farbig Markierten sind für diese Arbeit besonders wichtig, und werden im Folgenden erklärt, sodass in Zukunft nur noch die Abkürzungen verwendet werden müssen.

[16]http://smtlib.cs.uiowa.edu/logics.shtml, 29.10.2019

```
(set-logic QF_BV)
(declare-const a (_ BitVec 32))
(declare-const b (_ BitVec 32))
(declare-const z (_ bv0 32))
(assert (=> (and (> a z) (> b z)) (> (+ a b) z)))
(check-sat)
(exit)
```

Abbildung 3: SAT Prüfung des Theorems a > 0 & b > 0 ⇒ a + b > 0 im SMT-LIB2 Format dargestellt[17]

In Abbildung 3 wird eine SMT-Instanz in SMT-LIB2 Format aufgeführt. So wird zunächst die Logik festgelegt, also QF_BV, was für quantorenfreie Bitvektoren fester Größe steht. Weitere Beispiele sind: QF_ABV, QF_UFBV und QF_AUFBV, welche die quantorenfreien Arrays mit Bitvektoren, Uninterpreted Functions mit Bitvektoren und zuletzt die Kombination aus beiden bilden. Der Grund, warum viele dieser Logiken quantorenfrei sind, ist, dass Quantoren eine Erhöhung der Komplexität in der Berechnung der Lösungen bedeuten und sofern möglich vermieden werden. Die Bitvektor-Konstante z = 0, sowie die Bitvektorvariablen a und b, werden mit Länge 32 initialisiert. Anschließend wird das Theorem aufgeführt und direkt auf den Assertionstack gelegt. Man beachte die prefix Notation der Rechen- und Logikoperatoren. Anschließend wird auf Erfüllbarkeit geprüft und das Programm beendet. Der SMT-LIB Standard bietet weiterhin ein Repository an Benchmarks an, die in einem Wettbewerb eingereicht werden können, in dem die Solver in den jeweiligen Logiken gegeneinander antreten, die SMT-Competition. Die Tests werden in verschiedenen Kategorien durchgeführt und sind ein wichtiger Indikator für die Leistungsfähigkeit von SMT-Solvern.

3.3 Boolector

Wie bereits erwähnt, wurde der SMT-Solver Boolector ausgewählt um in JavaSMT integriert zu werden. Die Gründe hierfür, sowie eine Einführung in den Solver, mitsamt Eigenheiten, werden im Folgenden erörtert.
Boolector ist mit einer C-API[18] verfügbar, sowie einem Python Wrapper, genannt Poolector, der die gesamte Funktionalität auch in Python zur Verfügung stellt. Der Solver ist spezialisiert auf die Logiken QF_BV, QF_UFBV, QF_ABV und QF_AUFBV und wird mithilfe der MIT-Lizenz zur Verfügung gestellt. Während Boolector an sich kein

[17] Das Theorem ist erfüllbar in der gegebenen Logik, auch wenn das Problem des Überlaufs von Bitvektoren, aus Gründen der Komplexität, hier außer Acht gelassen wird.
[18] https://boolector.github.io/docs/cboolector_index.html, 25.10.2019

konkretes Modell erzeugt, lassen sich in der Regel Belegungen für erfüllbare Formeln bilden. Es gibt jedoch Ausnahmen, abhängig von den gewählten Optionen. Neben dem Lösen von festen Constraints ist auch das Lösen mithilfe von Assumptions möglich, die zwar zur Ergebnisfindung hilfreich sein können, aber nach einer Erfüllbarkeitsprüfung wieder automatisch entfallen. Weiterhin besteht die Möglichkeit begrenzt Quantoren über Bitvektoren zu verwenden. Formeln lassen sich in diversen Formaten, inklusive SMT-LIB2 ausgeben. Es kann zwischen zwei lokalen Suchstrategien zum Finden von erfüllbaren Belegungen für die QF_BV Logik gewählt werden. Eine basierend auf SLS[NPFB15], die Andere basierend auf Propagation[NPB17]. Diese Einstellungen lassen sich mit Bit-blasting kombinieren. Weiterhin besteht die Möglichkeit den zugrundeliegenden SAT-Solver zu wechseln. Es kann zwischen CaDiCaL[19], Lingeling, PicoSAT und MiniSat gewählt werden. Dies, verbunden mit den im Solver selbst verfügbaren Einstellungen, sorgt für eine breite Basis, auf der man viele Möglichkeiten zur Verbesserung der Performance seiner individuellen Nutzung von Boolector hat. Der Solver wurde in C geschrieben, wird ständig weiterentwickelt und befindet sich zum Zeitpunkt dieser Arbeit in Version 3.0.0. Boolector nahm bereits mehrfach an den SMT-Competitions teil, so auch wieder 2019. Während bereits in der Vergangenheit gute Platzierungen erzielt wurden, die maßgeblich zur Entscheidung für Boolector als Solver zum Integrieren in JavaSMT beigetragen haben, erzielte er auch dieses Jahr wieder gute bis sehr gute Ergebnisse[20]. Auch die Weiterentwicklung[PAB19] des Programms im Vergleich zu 2018 ist erkenntlich. So wurde zum Beispiel für die SMT-COMP 2019 die GNU Multiple Precision Arithmetic Library, kurz GMP[GMP16], als Backend für Boolectors lokale Suchfunktion, nach erfüllbaren Belegungen, für quantorenfreie Bitvektorkonstanten implementiert. Diese verbessert die Suche nach erfüllbaren Belegungen bei größeren Bitvektoren, sodass diese nahezu breitenunabhängig ist.
Bei genauerer Betrachtung von Boolector stellt man einige Eigenschaften fest, die zwar sinnvoll in Hinblick auf Geschwindigkeit und Benutzbarkeit sind, aber im späteren Verlauf der Implementierung noch problematisch werden. So verfügt Boolector zwar über die Möglichkeit ganze Formeln via Strings einzulesen, aber keine einzelnen Teile, wie z.B. nur eine Variable. Eine geparste Formel muss immer sofort auf SAT geprüft werden und ist nicht speicherbar. Weiterhin sind Quantoren eingeschränkt, da eine eigene Variable für die Argumente benutzt werden muss, die nie außerhalb genau eines Terms benutzt werden darf. Ferner sind Boolesche Variablen als Bitvektoren der Länge 1 gespeichert und nicht durch Prüfmethoden unterscheidbar. Zu Beginn der Arbeit gab es weiterhin keinerlei Möglichkeit auf Formeln, Variablen oder Konstanten zuzugreifen, die nicht explizit deklariert und gespeichert werden. Auf besagte Probleme wird im Abschnitt 4: Implementierung noch an den jeweiligen Stellen eingegangen.

[19] https://github.com/arminbiere/cadical, 25.10.2019
[20] https://smt-comp.github.io/2019/results, 25.10.2019

Alles in allem gilt Boolector im Bereich der Bitvektoren als führend, wird kontinuierlich weiterentwickelt und bildet eine gute Ergänzung zu JavaSMT, das im nächsten Kapitel vorgestellt wird.

3.4 JavaSMT

JavaSMT ist ein Projekt des Software und Computational Systems Lab der Ludwig Maximilians Universität München, das es sich zum Ziel setzt ein in Java implementiertes Rahmenwerk für SMT-Solver zu bilden, so dass Nutzer nur eine Schnittstelle für multiple SMT-Solver nutzen können, statt mehrerer.

```
(1)  (assert (= (+ x y) (* x y )))

(2)  boolector_assert(btor,
       boolector_eq(btor, boolector_add(btor, x, y), boolector_mul(btor, x, y)))

(3)  prover.addConstraint(bvmgr.equals(bvmgr.add(x, y), bvmgr.multiply(x, y)))
```

Abbildung 4: Assertion des Theorems x + y = x * y in (1) SMT-LIB2 Format, (2) Boolector-API, (3) JavaSMT-API dargestellt

JavaSMT legt hohen Wert darauf, so wenig Overhead wie möglich zu generieren, damit die Geschwindigkeits- und Speicheranforderungen an das System so wenig wie möglich durch das Rahmenwerk reduziert werden. Weiterhin wird die Anpassbarkeit der nativen Solver bestmöglich beibehalten, so dass, unter anderem, Optionen direkt an native Solver weitergereicht werden können. Ein weiterer Punkt, auf den geachtet wird, ist Typensicherheit, die verhindern soll, dass zwei nicht kompatible Logiken von Solvern gemischt werden, notfalls auch auf Kosten der Ausführbarkeit. Die Plattformunabhängingkeit von Java wird nur durch die Eignung der jeweiligen Solver für das zu benutzende Betriebssystem eingeschränkt. Jedoch wird JavaSMT mit all seinen Solvern für modernes Linux bereitgestellt und die Installation kann von Buildtools wie Maven unterstützt werden. Multithreading wird unter gewissen Umständen unterstützt und eine Schnittstelle für die Speicherverwaltung in nativem Code ermöglicht es Speicherlecks vermeiden zu können. JavaSMT nutzt weiterhin einen erweiterbaren LogManager, um Java und Solvereingaben aufzuzeichnen, sofern gewünscht und besitzt die Möglichkeit Solver mithilfe eines ShutdownManagers zu beenden. Weiterhin verfügt JavaSMT, dank der schon integrierten Solver, über diverse Theorien, inklusive Integer, Arrays und Bitvektoren, die es ermöglichen Solver zu vergleichen und zwischen ihnen zu wechseln, um ein Problem optimal zu lösen. Die verfügbaren Theorien sind nur durch die implementierten Solver beschränkt.
Neben der Grundfähigkeit zum SAT-Check gibt es noch Erweiterungen wie z.B. Unsat-Core, Interpolation und AllSAT, sofern der genutzte Solver dies unterstützt. JavaSMT

besitzt weiterhin ein ausgiebiges Testpaket, mit dessen Hilfe schnell erkannt werden kann, ob Standardfunktionen den Anforderungen entsprechend korrekt implementiert wurden.

	Boolector	CVC4	MathSAT5	Princess	SMTInterpol	Z3
AllSAT	nein	nein	ja	ja	ja	ja
AssumptionSolving	ja	nein	ja	nein	nein	ja
Interpolation	nein	nein	ja	ja	ja	ja
Optimization	nein	nein	ja	nein	nein	ja
UnsatCore	nein	ja	ja	ja	ja	ja
UnsatCore with Assumptions	nein	nein	ja	nein	nein	ja

Tabelle 3: Zusammenstellung der zusätzlichen Features in den in JavaSMT verfügbaren SMT-Solvern

4 Implementierung

In diesem Kapitel wird die Implementierung von Boolector in das JavaSMT Framework beschrieben. Beginnend mit der Erstellung des Java Native Interfaces und anschließender Einbindung des selbigen in JavaSMT, inklusive einer Beschreibung der speziellen Herausforderungen und Probleme um die für JavaSMT nötige standardisierte Nutzung zu gewährleisten. Zusätzlich wird auf die Erweiterung des Testpaketes von JavaSMT eingegangen, so dass dieses auch Boolector umfasst.

4.1 Schnittstelle zwischen Boolector und JavaSMT

Wie bereits erwähnt, ist Boolector in C implementiert und muss mit Hilfe eines speziellen standardisierten Interfaces mit Java kommunizieren, da es meist nicht möglich ist, fremde Programmiersprachen direkt zu nutzen. Hierzu wird das von Java bereitgestellte Java Native Interface, kurz JNI, genutzt, welches es erlaubt eine Palette von Programmiersprachen, wie C oder C++, in Java einzusetzen. Um aus Java heraus native Methoden aufzurufen, benötigt man zum einen den nativen Code als Programmbibliothek und zum anderen müssen sowohl in nativem Code, als auch in Java ein Interface erstellt werden. Dieses Interface wird auf nativer Seite, nach der Erstellung, mit Boolector und JNI zusammen zu einer Shared Library kompiliert. Technisch gesehen muss für jede Methode, die in Java aus nativem Code aufgerufen werden soll, ein Wrapper erstellt werden. Dieser bildet mit Hilfe der nativen Programmiersprache und JNI ein Bindeglied, das die Daten aus Java verarbeitet, an nativen Code übergibt, und die Rückgabewerte wieder an Java zurückgibt. Die Wrapper Methoden müssen einen Namen nach JNI Standard tragen, um später aus Java heraus aufgerufen werden zu können. Hier gilt besondere Vorsicht, da sogar Paketnamen in Java genau mit

eingearbeitet werden müssen. Weiterhin gilt es zu beachten, dass alle von Java übergebenen Typen in für die native Sprache verständliche Werte übersetzt werden müssen und eventuelle Rückgabewerte der nativen Methode ebenfalls in für Java verständliche Typen. Jedoch ist die Palette dieser Eingabewerte und Rückgabewerte auf primitive Java Datentypen und Objekte wie Strings oder Arrays beschränkt. Auch gilt es zu beachten, Objekte die innerhalb des Wrappers erstellt werden, um Daten von Java für C verwendbar zu machen, wieder zu löschen, sobald sie nicht mehr benötigt werden.
Boolector nutzt eigene Datenstrukturen um z.B. SAT oder UNKNOWN nach einer Berechnung anzuzeigen. Hier muss eine Umwandlung in für Java verständliche Werte und eine Rückinterpretation stattfinden, sobald der Rückgabewert in Java verwendet wird. Beispielsweise wird SAT, UNSAT und UNKNOWN in Boolector als Struct abgespeichert, was eine dementsprechende Interpretation in Java leicht macht, da nur die Indexstelle innerhalb des Structs geprüft werden muss. Ähnlich muss mit dem Struct für Optionen in Boolector verfahren werden, nur eben muss dort der Wert von der Java Seite aus bereits bekannt sein; also welcher Struct Eintrag zu welcher Option gehört, um diese später in nativen Code setzen zu können. Ein weiterer wichtiger Punkt in der Benutzung dieser Methoden ist, dass die meisten Objekte aus Boolector nicht direkt, sondern nur der Zeiger zu ihnen gespeichert und benutzt wird. Hier wird einfach der Wert des Zeigers als Long zurückgegeben, in Java gespeichert sofern nötig, und in der JNI Methode auf C Seite wieder als Zeiger auf das passende Boolector Objekt gecasted. Manche Methoden in Boolector geben jedoch keine Werte zurück, sondern erwarten als Eingabewert einen Pointer auf ein leeres Objekt, welches von der passenden Methode gefüllt wird. Diese Methoden sind in ihrer Rohform ungeeignet für JNI und werden nicht direkt, sondern mittels einer Hilfsmethode ausgeführt, die das passende Objekt erstellt, von Boolector füllen lässt und dann z.B. in ein mehrdimensionales Array packt, welches in Java verwendet werden kann.

Da die Erstellung des JNI-Wrappers, aufgrund der Menge an benötigten Methoden und Einstellungsmöglichkeiten, eine lange Zeit benötigt hätte, wurde ein Hilfsprogramm benutzt, welches einen großen Teil des Codes erstellt hat. Der Simplified Wrapper and Interface Generator, kurz SWIG, ist ein Programmierwerkzeug welches darauf spezialisiert ist, in C oder C++ geschriebenen Code für eine Vielzahl von Programmiersprachen verfügbar zu machen. SWIG ist Open Source und plattformübergreifend nutzbar und kann sowohl den C-Wrapper-Teil für Boolector, als auch das Java Interface für diesen generieren. Nach dem Erstellen eines SWIG Headers, in den die gewünschten Einstellungen und Quellmethoden oder gar ganze Quell-Dateien, wie Header, eingefügt werden, wird der Quellcode im gewünschten Format, hier JNI, erstellt. Das Nachprüfen und Korrigieren des generierten Codes war an einigen Stellen nötig, da SWIG dazu tendiert, manche Rückgabetypen grob zu generalisieren oder gar Zeiger mit Objekten zu verwechseln. So wird beispielweise ein 32-bit Integer Rückgabetyp einfach zu einem

Long gecastet, oder der Zeiger auf einen 32-bit Integer als Long zurückgegeben anstatt des Integers selbst. Die bereits erwähnten Hilfsfunktionen mussten komplett per Hand erstellt werden. Außerdem wurde, neben Fehlerbehebungen, der von SWIG generierte Code noch zusammengefasst und vereinfacht, so dass am Ende statt 16 Java Dateien, nur noch 2 existieren, eine für Boolectors Optionen und eine für den allgemeinen Wrapper.
Sobald man versucht, die nativen Methoden in Java zu testen, ist es schwierig Fehler zu erkennen, da die Java Virtual Machine keine Fehlerinformationen wie etwa eine NullpointerException erkennen kann. Zwar existiert ein Flag, mit dessen Hilfe die Java Virtual Machine mehr Informationen über JNI ausgeben kann, jedoch hat sich der Nutzen dessen auf die Suche von Memoryleaks beschränkt. Es ist möglich, dass Boolector selbst den Fehler erkennt, was aber nicht immer in einer sinnvollen Fehlermeldung resultiert, aber wiederum immer das Programm beendet. Tritt der Fall ein, dass auf eine Speicherstelle zugegriffen wird, auf die nicht zugegriffen werden darf, wird vom Betriebssystem ein sogenannter SIG-SEV ausgelöst, eine Speicher Zugriffsverweigerung, die in Beendigung des nativen Programms, als auch der Java Virtual Machine enden kann. Die Möglichkeiten die Fehler hier zu finden sind begrenzt und wurden hauptsächlich durch die Analyse der automatisch generierten Fehlerlogs durchgeführt. Diese gaben aber, neben der Methode in der der Fehler aufgetreten ist, nur Maschinencode wieder, was bedeutet, Assemblercode muss aus dem Fehlerdump erstellt und analysiert werden, um den Programmierfehler zu finden und zu beheben. Es ist zu beachten, dass nach einer Änderung des Interfaces auf der Seite von Boolector, die Shared Library neu kompiliert werden muss.
Um die Wartbarkeit des Interfaces auch auf der Seite von Boolector in Zukunft möglich zu machen, wurde der Code zusammen mit der kompilierten Library bereitgestellt. Weiterhin wurde eine Dokumentation erstellt und hinzugefügt, die genaue Schritte angibt, mit deren Hilfe eine zukünftige Wartung und Erweiterung des Interfaces möglich ist. Verwoben mit der Erstellung der dynamischen Bibliothek, bildet dies den ersten Teil der Implementierung, deren Beschreibung nun in JavaSMT fortgesetzt wird.

4.2 Integration in JavaSMT

Möchte man nun einen neuen Solver in JavaSMT integrieren, muss man den Solver so bereitstellen, dass er in Java nutzbar ist, und anschließend die SolverContext Schnittstelle erweitern. Weiterhin muss eine Factory für den neuen Solver erstellt werden. Sofern das Backend in JavaSMT geschrieben wird, kann jedoch die SolverContextFactory von JavaSMT genutzt werden. Anschließend müssen die Schnittstellen der verschiedenen Funktionen von JavaSMT mit denen des neuen Solvers verbunden werden, um der standardisierten Funktionsweise von JavaSMTs API möglichst nahe zu kommen. Sollte dies nicht möglich sein, wird eine **UnsupportedOperationException** geworfen, um

zu signalisieren, dass eine Funktion nicht implementiert ist.

4.2.1 Context und Environment

Zunächst wird die **SolverContextFactory** erweitert, welche zum einen das Enum der verfügbaren Solver enthält, und zum anderen eine SolverContext Instanz für den zu benutzenden Solver konstruiert. Die Schnittstelle **BoolectorSolverContext** hat zum einen Aufgaben wie das Schließen des gesamten Kontextes für den Solver, aber auch das Initialisieren der Prover Umgebungen, was im Falle Boolectors nur ein Standard Theorem Prover ist. Ebenso werden die benutzerspezifischen Objekte weitergegeben an das **BoolectorEnvironment**; unter anderem die möglichen Konfigurationen. Als übergeordnetes Konstrukt wird hier die **Configuration** Klasse verwendet, die mithilfe der sosy_lab.common Library bereitgestellt wird. Es wurden zum einen einige Grundeinstellungen so gewählt, dass Boolector standardmäßig keine expliziten Optionen braucht um in JavaSMT zu funktionieren. Automatisches Löschen des benutzten Speichers bei Schließen der Instanz und die Möglichkeit, mehr als ein Mal, auf SAT zu prüfen, sowie die Auswahl zwischen den SAT Solvern Lingeling und PicoSAT, sind nennenswerte Beispiele. Cadical kann im Moment nicht als SAT-Solver gewählt werden, da der Incremental-Mode mit diesem nicht verwendbar, dieser aber zwingend notwendig für die Verwendung des Stacks ist. Außerdem wurde aufgrund der massiven Palette an Einstellungen eine Möglichkeit implementiert, einfach durch einen String die Option aus der Boolector API zu kopieren und zu übernehmen. Neben dem **LogManager** und dem Pfad zur Log-Datei, die Boolector benutzt um seine API Aufrufe aufzuzeichnen. Da im Moment nur der SAT-Solver Lingeling die Funktion zum Beenden der Kalkulation bietet, wurde der ShutdownManager zum Beenden der Ausführung nicht implementiert, um etwaige Probleme zu verhindern, sollte ein anderer SAT-Solver genutzt werden.

4.2.2 Manager

Die Funktionen der verwendeten SMT Theorien werden durch Schnittstellen, die intern Manager genannt werden, integriert. Der **BoolectorFormulaManager** iniziiert die anderen implementierten Manager, inklusive des **BoolectorFormulaCreator**. Außerdem wird in dieser Klasse das Parsing und Dumping der Formeln eingebaut. Boolector verfügt über die Möglichkeit Formeln, Variablen und Konstanten im SMT-LIB2 Format auszugeben. Realisiert wird dies mithilfe einer der Hilfsmethoden innerhalb des C-Wrappers, die eine Datei erstellt, diese von Boolector mit dem Dump füllen lässt, anschließend wieder gelesen und an Java als String weitergegeben wird. Parsing wurde aufgrund der in Kapitel 3.3 beschriebenen Probleme nicht implementiert.
Eine Besonderheit bildet der **BoolectorBooleanManager** und die enthaltene Boolean-Theorie. In Boolector werden boolesche Werte als Bitvektoren der Länge eins kodiert

und sind nicht durch Prüfmethoden unterscheidbar. Deshalb wurden alle Bitvektoren der Länge eins als Boolean implementiert, was sowohl Anpassungen in der Auswertung, als auch Erstellung der Booleschen- und Bitvektorvariablen nach sich zieht. Die Theorie der Bitvektor-Quantoren wurde so weit wie möglich implementiert, allerdings bildet hier die unterschiedliche API zwischen Boolector und JavaSMT ein Problem. Boolectors Quantoren brauchen den Variablentypen **boolector_param**, der nur in Quantoren verwendet wird, inkompatibel ist mit den sonst verwendeten Variablentypen und pro Deklaration nur einmal an eine Funktion gebunden werden darf. In JavaSMT gibt es keine einfache Möglichkeit, dies ohne Umstände in der API klar zu machen. Die Klasse wurde weitestgehend implementiert, konnte allerdings nicht fertiggestellt werden. Details dazu folgen im Kapitel Ergebnisse und zukünftige Arbeiten.

4.2.3 Creator

Der Creator regelt neben der Variablenerstellung noch die Erstellung von Arrays und uninterpreted Functions. All diese sind in JavaSMT von einem Wrapper umgeben, der es ermöglicht schnell den Typ des Objektes zurückzugeben, ohne Checks in nativem Code. Um diesen für Boolector Objekte richtig nutzbar zu machen, mussten die zugehörigen Klassen in **BoolectorFormula** erweitert werden, so dass die Kapsel die Formel enthält, wie auch die Kapsel-Daten: wie Typ oder bei z.B. Arrays der Typ des Indexes. Die Boolector Instanz muss ebenfalls immer abrufbar sein, da sie für jede native Methode gebraucht wird. Weder Boolector noch JavaSMT implementieren standardmäßig einen Variablencache, was aber von JavaSMT gefordert wird. Daher wurde dieser in der Klasse **BoolectorVariablesCache** implementiert und sorgt dafür, dass Variablen derselben Art und des selben Namens auch auf dasselbe Objekt umgeleitet und nicht doppelt erstellt werden oder es zu einem Crash des nativen Programms kommt.
Auch im Creator wird die Übersetzung von Rückgabewerten des Solvers im späteren Modell gehandhabt. Also eine Übersetzung des Assignmentstrings für Bitvektoren von nativem Code in BigInteger oder Wahrheitswerte für Booleans. Mit einer Besonderheit: Boolector setzt für Werte die beliebig sein können ein x in den String, welches durch eine Konstante in 1 umgewandelt wird. Die 1 wurde statt der 0 ausgewählt, da Boolector in der Standardeinstellung dazu tendiert Werte so groß wie möglich zu wählen.
Eine weitere Kernfunktion von JavaSMT ist die **FormulaVisitor** Klasse, die die Möglichkeit bieten soll, alle Formeln besuchen zu können, auch die, die innerhalb einer anderen Formel deklariert werden, ohne explizit gespeichert zu werden. Die Funktionalität des **FormulaVisitor** wird mithilfe der **visit()** Methode für Solver bereit gestellt. Das Problem hier ist, dass zum Zeitpunkt dieser Arbeit keine Möglichkeit in Boolector besteht, auf innere Formel oder Variablen in Formeln zuzugreifen. Deshalb wurde die **visit()** Methode aktuell nicht implementiert.

Dies führt dazu, dass einige Funktionen, die auf dem **FormulaVisitor** aufbauen,

```
(assert (= (declare-const a (_ BitVec 32)) b))
```

Abbildung 5: In Grün eine Variablendeklaration, auf die im Moment nicht zugegriffen werden kann, im SMT-LIB2 Format

nicht zur Verfügung stehen, wie die bereits erwähnten Quantoren. Es wäre zwar grundlegend möglich, die Eingabe der Formel zu zerlegen und jede Formel, Variable und Konstante, auch die Inneren, einzeln zu deklarieren und zu speichern, dies würde aber einen Speicheroverhead erzeugen, da jedes Objekt doppelt gespeichert werden müsste, in JavaSMT und in nativem Boolector. Nach Kontaktaufnahme mit dem Boolector Entwicklerteam wurde an einer Lösung gearbeitet, die in Kapitel 6 noch weiter beleuchtet wird.

4.2.4 Prover und Model

Nachdem man Formeln erstellt hat, will man diese natürlich auch lösen. Dementsprechend wurden in der Klasse **BoolectorAbstractProver** die Grundfunktionen von Provern implementiert. In deren Unterklasse **BoolectorTheoremProver** hingegen, wurde nichts außer dem Superkonstruktor implementiert, sodass in Zukunft, falls Boolector sich dazu entschließt weitere Prover zu implementieren, bereits der Grundstein zur Trennung dieser gelegt wurde. Im Prover werden neben der Funktion **assert()**, zum Hinzufügen von Formeln, auch das Prüfen auf SAT bzw. UNSAT mithilfe von **isUnsat()** übernommen. Ein wichtiger Bestandteil, der ebenfalls hier seine Funktionen findet, ist der Assertionstack; implementiert durch **push()** und **pop()**.
Eine Belegung für die erfüllbaren Formeln, kann mithilfe des Modells, implementiert in **BoolectorModel**, erzeugt werden. Boolector selbst gibt Assignmentstrings nur dann zurück, wenn man sie für einen spezifischen Term anfragt. Zum einen muss die Art des Terms geprüft werden, bevor die zugehörige native Methode für Assignments genutzt wird, zum anderen ist auch hier das Problem präsent, dass es keine Möglichkeit gibt auf nicht gespeicherte Terme zuzugreifen, ähnlich wie bei der bereits vorgestellten **visit()** Methode. Dieses Problem ist in doppelter Hinsicht hinderlich, da die **toList()** Methode, die nach Erstellung eines Modells unter anderem versucht, alle Assignments zu allen Variablen zu speichern, genau auf der Visitor-Klasse, und deren Fähigkeiten alle Terme abzugehen, aufbaut. Dementsprechend wurde die Methode nicht implementiert. Jedoch ist die Fähigkeit vorhanden, jede explizit deklarierte Funktion, Variable oder Konstante abzufragen.

4.3 Tests

Da Fehler beim Programmieren genauso wie Planungsfehler nie ganz vermieden werden können, muss Code getestet werden. JavaSMT bietet einige ausführliche Testklassen

genau zu diesem Zweck an. Diese JUnit Tests können ausgeführt werden, um die Integration eines neuen Solvers, oder die Erweiterung bestehenden Codes zu überprüfen und decken ein breites Spektrum an Testfällen ab.
Boolector besitzt einige Eigenheiten, die, zumindest in Bezug auf die existierenden Solver, einzigartig sind. Dementsprechend muss die Testumgebung angepasst werden, um sowohl den alten Solvern, als auch Boolector gerecht zu werden. Das einschneidendste Beispiel hierfür ist, das Boolector ausschließlich Bitvektortheorien benutzt. Da aber die meisten Tests mit Integer Theorie als Basisfälle implementiert wurden, mussten diese angepasst werden. Zunächst wurde die Funktion **requireIntegers()** eingeführt, die prüft, ob ein Solver überhaupt über die Möglichkeit verfügt, Integer Theorie zu benutzen. Für alle Testfälle, die trotzdem möglich sind, aber eben nur mithilfe von Bitvektoren, wurde je nach Ausführlichkeit des Tests, entweder ein neuer Test erstellt, der dann auch für andere Solver, die die Bitvektor Theorie unterstützen, nutzbar ist, oder der bestehende Test mit einer Teilung zwischen den Theorien erweitert. Weiterhin mussten alle Testfälle für Bitvektoren der Länge 1 angepasst werden, da diese in Boolektor als boolesche Variablen verwendung finden und damit nicht zur Verfügung stehen. Des Weiteren wurde Boolector von Tests, die grundsätzlich nicht möglich sind, ausgeschlossen, beispielsweise UnsatCore Tests.
Ein weiteres Problem war das Fehlen der Funktionen **toList()** und **visit()**, die beide oft in Tests Verwendung finden. Außerdem basieren einige Tests darauf, Strings mithilfe der Parsefunktion einzulesen und zu verwenden, was in Boolector ebenfalls nicht möglich ist. Da es eine radikale Abweichung von den Tests der anderen Solver bedeutet hätte, wurde davon abgesehen, diese Tests für Boolector zu implementieren.

5 Evaluation

5.1 Testbedingungen

Im folgenden Kapitel wird gezeigt, dass die Implementierung von Boolector nicht nur theoretisch, sondern auch praktisch verwendbar ist. Dennoch stellt sich die Frage, wie Boolector im Vergleich zu den anderen Solvern in JavaSMT abschneidet. Dementsprechend werden die Bedingungen beschrieben unter denen der neu integrierte SMT-Solver mithilfe des Softwareanalysetools CPAchecker[BK11], gegenüber den anderen Solvern in JavaSMT begutachtet wird.
CPAchecker bietet, neben der Möglichkeit zur Analyse von Software in verschiedenen Kategorien, noch die Möglichkeit an, Benchmarks für diese auszuführen. So lassen sich, unter anderem, die in JavaSMT verwendeten SMT-Solver testen, da es als Basis für CPAchecker verwendet werden. Da Boolector auf Bitvektoren spezialisiert ist, kann er nur mit Solvern verglichen werden, die ebenfalls diese SMT-Theorie nutzen. Dementsprechend wurde Boolector gegenüber Z3, CVC4 und MathSAT5 jeweils mit den gleichen Optionen begutachtet. Als grundlegendes Modell wurde die Software Verification Competition SV-COMP[21] gewählt, um bereits bewährte Standards bezüglich Benchmarktasks nutzen zu können. Alle Benchmarks wurde auf baugleichen Intel Xeon E3-1230 v5 @ 3.40GHz durchgeführt, die jeweils 33 GB Speicher besaßen und unter Ubuntu 18.04 liefen. In diesem Aufbau wurde der CPAchecker[22] mit Revision *32242* und das Benchmarking Framework Benchexec[23] verwendet und für den Vergleich der Verifikationsansatz Bounded Model Checking, kurz BMC. Weiterhin wurde eine Teilmenge der SV-Benchmarks[24] als Testsets gewählt. Konkret wurden die Sets **ReachSafety-ControlFlow**, **SoftwareSystems-DeviceDriversLinux64-ReachSafety**, **ReachSafety-ECA**, **ReachSafety-Heap**, **ReachSafety-Loops**, **ReachSafety-Sequentialized** und **ReachSafety-ProductLines** gewählt und unter der **unreach-call** Spezifikation ausgeführt. Die in den Tasks Verwendung findenden Schleifen werden limitiert auf einmal 1 Abrollung und einmal 10. Der verwendete Speicher wurde auf 15 GB, verwendbare CPU Kerne auf 2 und maximale Dauer eines Tests auf 15 Minuten (900s) limitiert. Der Heap für die Java Virtual Machine wurde auf 13000 MB beschränkt. Aufgrund der Beschränkungen Boolectors mussten einige Optionen speziell gewählt werden. Infolge dessen werden Floats als Integer mithilfe von **encodeFloatAs=INTEGER** und Integer als Bitvektoren mithilfe von **encodeIntegerAs=Bitvector** kodiert. Der **FormulaManager** wird durch **createFormulaEncodingEagerly=false** verzögert initialisiert. Die Ausgabe der Testergebnisse wurde aufgrund des Fehlens von notwendigen Methoden im Model, z.B.

[21] https://sv-comp.sosy-lab.org/2020/, 25.10.2019
[22] https://svn.sosy-lab.org/software/cpachecker/trunk/, 13.11.2019
[23] https://github.com/sosy-lab/benchexec, 25.10.2019
[24] https://github.com/sosy-lab/sv-benchmarks/, 26.10.2019

toList(), durch die Option **-noout** deaktiviert. Interne Optimierungen mussten, basierend auf dem Fehlen des FormulaVisitors, abgeschalten werden. Darauf basierend wurde die Option **handlePointerAliasing=false** gesetzt. Aufgrund dieser Optionen ist zu erwarten, dass die Genauigkeit der Ergebnisse etwas sinkt, was aber keinen Einfluss auf die Bewertung haben wird, da alle Solver mit den gleichen Bedingungen antreten.

5.2 Ergebnisse der Evaluation

Verglichen werden zunächst die Ergebnisse der Testsets. Wünschenswert ist eine hohe Anzahl an korrekten Ergebnissen aber eine möglichst niedrige an inkorrekten. In Tabelle 4 ist zu erkennen, dass Boolector bei nur 1er Abrollung der Schleifen vergleichbare Ergebnisse zu den anderen Solvern erzielt. Bei 10 Abrollungen jedoch, erzielt Boolector zum einen die höchste Zahl an korrekten, sowie die höchste Zahl an inkorrekten Ergebnissen. Letzteres ist, im Detail betrachtet, aber kein Problem, da die von den SMT-Solvern durchgeführten Rechnungen selbst nicht falsch sind, weshalb die inkorrekten Ergebnisse in den folgenden Speicherbedarfs- und CPU-Zeit-Analysen nicht außer Acht gelassen werden. Weiterhin stellt sich bei detailierter Betrachtung heraus, dass die anderen Solver bei den inkorrekten Tasks in den meisten Fällen aufgrund von Speicherproblemen gar keine Ergebnisse liefern, was einen Vergleich allerdings erschwert.

k	Solver	Korrekte Ergebnisse			Inkorrete Ergebnisse		
		total	true	false	total	true	false
1	Boolector	729	415	314	70	0	70
	CVC4	726	**419**	307	72	0	72
	MathSAT5	**738**	**419**	**319**	**73**	0	**73**
	Z3	728	**419**	309	64	0	64
10	Boolector	**1587**	**720**	867	**246**	**120**	**126**
	CVC4	1252	540	712	118	1	117
	MathSAT5	1446	553	**893**	127	1	**126**
	Z3	1300	543	757	119	1	118

Tabelle 4: Vergleich der korrekten und inkorrekten Ergebnisse in den durchgeführten Tests aufgeteilt in SMT-Solver und k-maligem Abrollen der Schleifen

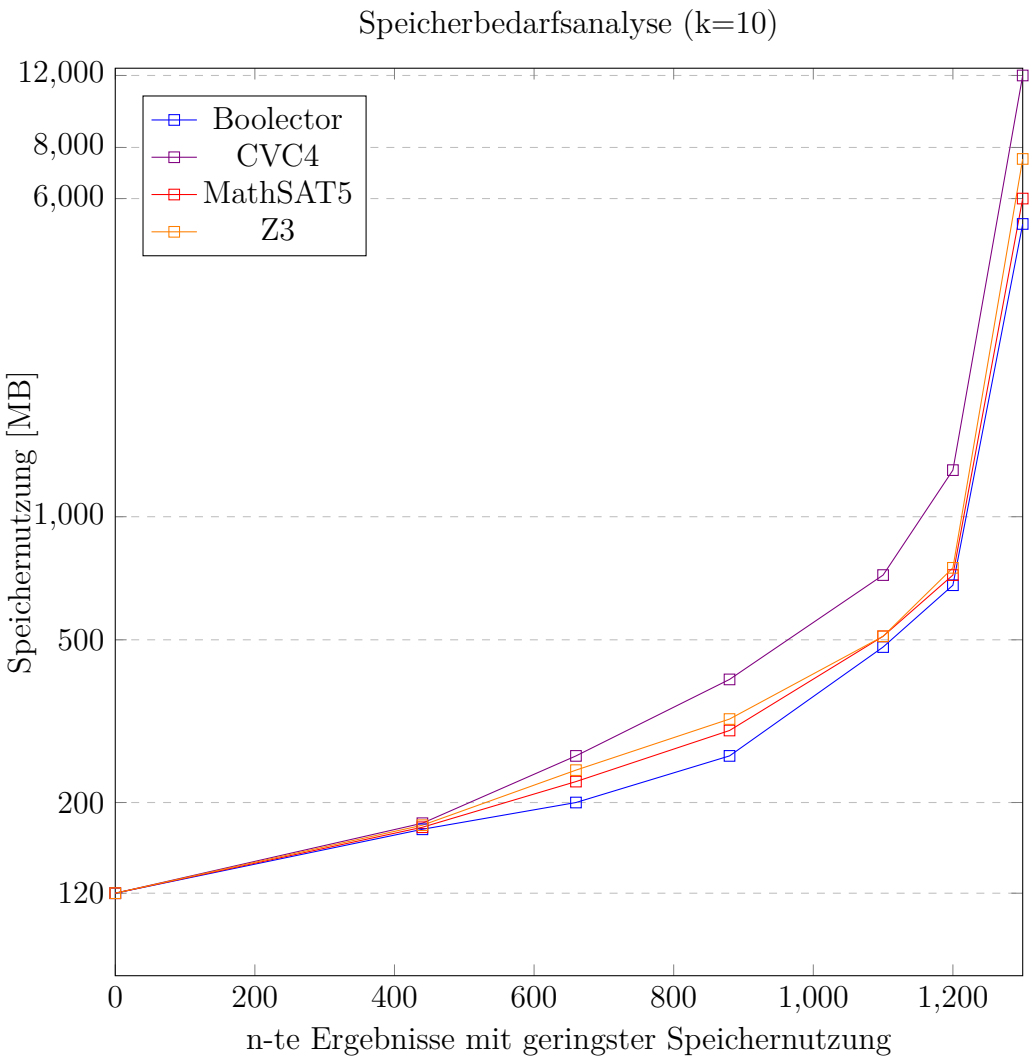

Abbildung 6: Quantil-Funktion bezüglich Speichernutzung [MB] (Y-Achse) und der Anzahl an Tasks, sortiert nach Speichernutzung, (X-Achse) über die Schnittmenge der von allen Solvern korrekt und inkorrekt gelösten Tasks, mit k=10 Abrollungen der Schleifen

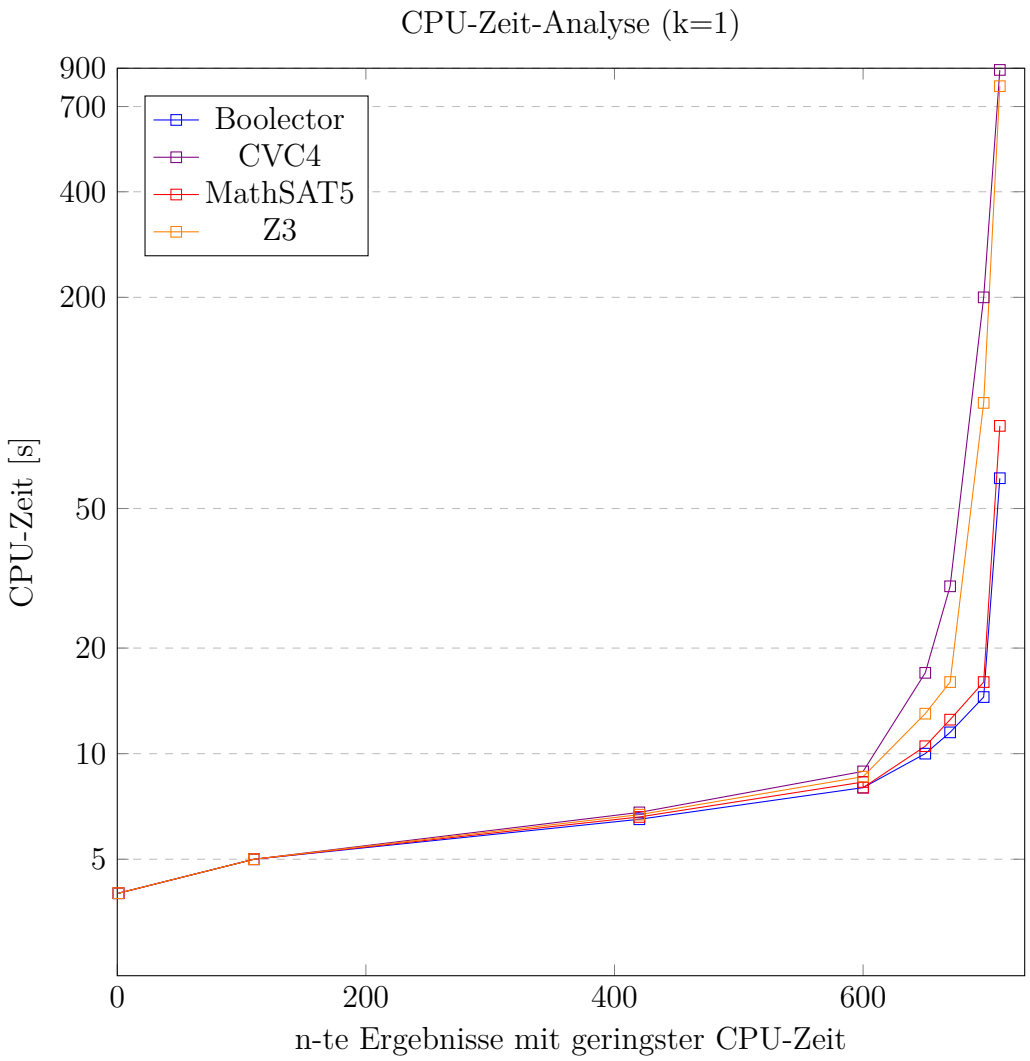

Abbildung 7: Quantil-Funktion bezüglich CPU-Zeit [s] (Y-Achse) und der Anzahl an Tasks, sortiert nach CPU-Zeit, (X-Achse) über die Schnittmenge der von allen Solvern korrekt und inkorrekt gelösten Tasks, ausgeführt mit k=1 Abrollungen der Schleifen

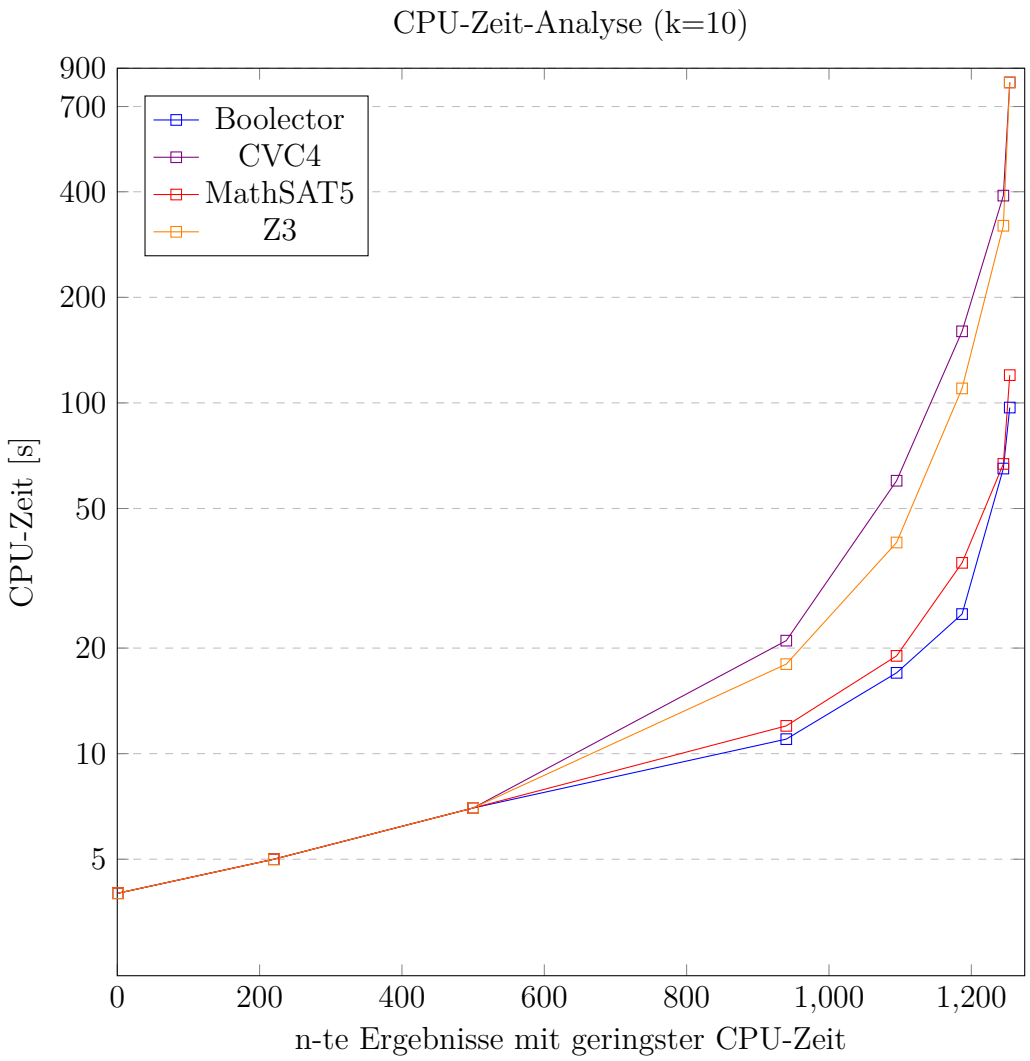

Abbildung 8: Quantil-Funktion bezüglich CPU-Zeit [s] (Y-Achse) und der Anzahl an Tasks, sortiert nach CPU-Zeit, (X-Achse) über die Schnittmenge der von allen Solvern korrekt und inkorrekt gelösten Tasks, ausgeführt mit k=10 Abrollungen der Schleifen

Für die Analyse wurde die Teilmenge der korrekt und inkorrekt gelösten Tasks von allen Solvern gebildet, um eine möglichst hohe Vergleichbarkeit zu gewährleisten. In Bezug auf die Speichernutzung der Solver in Abbildung 6 ist zu erkennen, dass Boolector bei 10 Schleifenabrollungen im Durchschnitt knapp am wenigsten Speicher benötigt. Bei nur 1er Abrollung ist die Speichernutzung unter den Solvern nahezu gleich, weshalb auf eine Darstellung verzichtet wurde.
Betrachtet man die CPU-Zeit in Abbildung 7 bei nur 1 Schleifenabrollung, zeigt sich eine vergleichbare Leistung von Boolector und MathSAT5. Beide verweisen jedoch Z3 und CVC4 auf die beiden letzten Plätze. In Abbildung 8 ist ein knapper Vorsprung vor MathSAT5 erkennbar, sowie ein deutlicher im Vergleich zu Z3 und CVC4.

Zusammenfassend kann man sagen, dass Boolector in Bezug auf CPU-Zeit einen wenn auch nur minimalen Vorsprung vor den anderen Solvern hat. Bezieht man die Ergebnisse der korrekten Lösung der Tasks und der niedrigen Speichernutzung mit ein, hat sich Boolector als etwas leistungsfähiger gegenüber den anderen Solvern in Bezug auf Bitvektoren in JavaSMT gezeigt.

6 Ergebnisse und zukünftige Arbeiten

Die in dieser Arbeit beschriebene Implementierung des auf Bitvektoren spezialisierten SMT-Solvers Boolector in das JavaSMT Framework bietet eine Erweiterung der Palette an verfügbaren Solvern in diesem. Es konnte gezeigt werden, dass die Funktionen zum Aufstellen und Lösen von Formeln, auf der Basis von Bitvektoren, sowie die Tests zum Überprüfen dieser, weitestgehend funktionieren.

Weiterhin wurde durch die Evaluation mithilfe des CPAcheckers die Funktionalität der Implementierung trotz einiger Probleme bestätigt. Boolector lieferte weitestgehend gute Ergebnisse, vergleichbar mit den bereits existierenden Solvern. Bei der Analyse des Speicherbedarfs wurde deutlich, dass Boolector knapp der Solver mit der geringsten Speichernutzung war. Während bei geringer Komplexität der Tests die benötigte CPU-Zeit vergleichbar gut mit bereits existierenden Solvern ist, zeigte sich bei erhöhter Komplexität eine minimal bessere Leistung von Boolector in Bezug auf Bitvektoren.

Das Fehlen einiger Funktionalitäten in Boolector sorgte jedoch dafür, dass zwei wichtige Methoden in JavaSMT nicht implementiert werden konnten. Diese Methoden haben Auswirkungen auf Teile der Nutzbarkeit von JavaSMT und die Tests der Implementierung. Quantoren über Bitvektoren, eine Teilmenge der Tests und die Methoden selbst sind im Moment nicht nutzbar. Nach Rücksprache mit den Entwicklern[25] von Boolector, wurde jedoch zugesichert, dass diese Funktionalitäten nachgereicht werden. Sobald diese zur Verfügung stehen, können die fehlenden Methoden an den bereits vorbereiteten Stellen implementiert und ein höherer Grad an Funktionalität erreicht werden. Eine erneute Evaluation mit diesen Funktionen würde genauere Ergebnisse liefern in Bezug auf die anderen Solver in JavaSMT. Weiterhin wurde von den Entwicklern eine weit leistungsstärkere Version von Boolector angekündigt. Diese sollte, auf Basis dieser Arbeit, ebenfalls in JavaSMT Verwendung finden können und die Leistungen, welche in der Evaluation gezeigt wurden, noch einmal verbessern.

[25] https://groups.google.com/forum/#!topic/boolector/3ygEjhYi05w, 29.10.2019

Literaturverzeichnis

[Ber] Berre, Daniel L.: SAT Live! keep up to date with research on the satisfiability problem, URL: http://www.satlive.org

[BK11] Beyer, Dirk ; Keremoglu, M. E.: CPAchecker: A Tool for Configurable Software Verification. In: Gopalakrishnan, G. (Hrsg.) ; Qadeer, S. (Hrsg.): *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011, Snowbird, UT, July 14-20)*, Springer-Verlag, Heidelberg, 2011 (LNCS 6806). – ISBN 978–3–642–22109–5, 184-190

[BST+10] Barrett, Clark ; Stump, Aaron ; Tinelli, Cesare u. a.: The smt-lib standard: Version 2.0. In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)* Bd. 13, 2010, S. 14

[BT17] Barrett, Fontaine ; Tinelli: The smt-lib standard: Version 2.6, 2017

[Coo71] Cook, Stephen A.: *The complexity of theorem-proving procedures.* 1971

[CS17] Cassez, Franck ; Sloane, Anthony M.: ScalaSMT: Satisfiability Modulo Theory in Scala (tool paper). In: *SCALA 2017*, Association for Computing Machinery, Inc, 10 2017, S. 51–55

[GM15] Gario, Marco ; Micheli, Andrea: PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In: *SMT Workshop 2015*, 2015

[GMP16] *The GNU Multiple Precision Arithmetic Library.* `https://gmplib.org/`. Version: 2016

[HFF+11] Haedicke, Finn ; Frehse, Stefan ; Fey, Görschwin ; Grosse, Daniel ; Drechsler, Rolf: metaSMT: Focus on Your Application not on Solver Integration. In: *DIFTS@ FMCAD*, 2011

[IR06] In Rossi, Peter; Walsh Toby (. Francesca; Van Beek B. Francesca; Van Beek: Backtracking search algorithms. In: *Handbook of constraint programming* (2006), S. 122

[KFB16] Karpenkov, Egor G. ; Friedberger, Karlheinz ; Beyer, Dirk: JavaSMT: A Unified Interface for SMT Solvers in Java. In: *VSTTE*, Springer, 2016 (LNCS 9971), 139–148

[LH19] Liana Hadarean, Antti Hyvarinen et. a.: *The International Satisfiability Modulo Theories Competition.* https://smt-comp.github.io/2019/index.html, 2019

[NPB17] Niemetz, Aina ; Preiner, Mathias ; Biere, Armin: Propagation based local search for bit-precise reasoning. In: *Formal Methods in System Design* 51 (2017), Nr. 3, 608–636. http://dx.doi.org/10.1007/s10703-017-0295-6. – DOI 10.1007/s10703–017–0295–6

[NPFB15] Niemetz, Aina ; Preiner, Mathias ; Fröhlich, Andreas ; Biere, Armin: Improving Local Search For Bit-Vector Logics in SMT with Path Propagation. In: *Proc. 4th Intl. Work. on Design and Implementation of Formal Tools and Systems (DIFTS 15)*, 2015, S. 10 pages

[PAB19] Prof. Armin Biere, Mathias P. Aina Niemetz N. Aina Niemetz: *Boolector at the SMT Competition 2019*. https://boolector.github.io/system-descriptions/NiemetzPreinerBiere-SMT-Competition-2019.pdf. Version: 2019

[Zei70] Zeitin, G. S.: *On the complexity of derivation in propositional calculus.* 1970