LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

LEHR- UND FORSCHUNGSEINHEIT FÜR THEORETISCHE INFORMATIK UND THEOREMBEWEISEN

Minimierung von Problemen im SMT-LIB v2 Format

Maximilian Öttl

Bachelorarbeit
im Studiengang Informatik

Betreuer: Prof. Jasmin Blanchette

Mentor: Lydia Kondylidou

Ablieferungstermin: 20. April 2025

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 20. April 2025 Autor

Kurzfassung

Diese Arbeit beschäftigt sich mit der Minimierung von Problemen im SMT-LIB v2 Format. SMT-Solver spielen eine zentrale Rolle in der formalen Verifikation, stoßen jedoch oft an ihre Grenzen, insbesondere bei komplexen Formeln aus der TPTP-Bibliothek. Ziel dieser Arbeit ist es, ein Verfahren zur Reduktion von SMT-Problemen zu entwickeln, das überflüssige Funktionsargumente entfernt, boolesche Operationen vereinfacht und ein strukturiertes Refactoring vornimmt. Dabei kommen verschiedene Strategien zum Einsatz, darunter ein optimaler und ein Greedy-Ansatz. Erste Ergebnisse zeigen eine deutliche Reduktion der Komplexität der Probleme bei gleichbleibendem Lösungsstatus. Dieses Verfahren bietet somit eine effiziente Methode zur Vorverarbeitung von SMT-Problemen und erleichtert das Debugging, die Analyse komplexer Probleme und die Verbesserung von SMT-Solvern.

Keywords: SMT-LIB v2, SMT-Solver, Minimierung, formale Verifikation, TPTP-Bibliothek, Greedy-Ansatz, optimaler Ansatz, Refactoring

Inhaltsverzeichnis

1 Einleitung

Die automatische Überprüfung logischer Formeln spielt nicht zuletzt im Bereich der formalen Verifikation eine zentrale Rolle. Satisfiability Modulo Theories (SMT)-Solver prüfen die Erfüllbarkeit logischer Formeln unter Einbeziehung spezieller Theorien und stellen ein wichtiges Werkzeug dar, um diese logischen Probleme zu lösen. Während klassische Satisfiability (SAT)-Solver ausschließlich aussagenlogische Formeln analysieren, erweitern SMT-Solver dieses Konzept, indem sie zusätzliche mathematische Theorien integrieren, um komplexere Probleme zu behandeln. Betrachten wir zunächst das unerfüllbare Axiom $(\forall x.\, p\, x) \wedge \neg p\, b$, um die Arbeitsweise eines SMT-Solvers zu verdeutlichen. Ein SAT-Solver ermittelt zunächst ein Modell, in dem $\forall x.\, p\, x$ als wahr und $p\, b$ als falsch bewertet wird. Im Anschluss daran werden in einem heuristischen Verfahren die Quantoren instanziiert, indem zunächst x durch einen geeigneten Term, z. B. a, ersetzt wird, sodass sich die Implikation $(\forall x.\, p\, x) \implies p\, a$ ergibt. Nachdem der SAT-Solver ein Modell gefunden hat, in dem auch $p\, a$ als wahr gilt, erfolgt eine weitere Instanziierung mit b, was zu $(\forall x.\, p\, x) \implies p\, b$ führt. Da es unmöglich ist, dass $p\, b$ zugleich wahr und falsch ist, führt dies zu einem Widerspruch. Das Axiom ist folglich unerfüllbar Kondylidou et al.. SMT-Solver stoßen allerdings bei großen und komplexen Problemen häufig an ihre Grenzen und können diese nicht auf zufriedenstellende Weise verarbeiten. Zudem stellen diese Probleme eine große Herausforderung für Forscher bei der Suche nach einem Beweisansatz dar. Um diesen Umstand zu verbessern und um Forschern dabei zu helfen, SMT-Solver zu verbessern, lohnt es sich, die gegebenen Probleme zu minimieren. Diese Arbeit befasst sich mit einem neuartigen Verfahren, das verschiedene Minimierungen vornimmt. Es wurde ein Algorithmus entwickelt, der überflüssige Funktionsargumente und unnötige Operationen aus den Problemen entfernt. Darüber hinaus werden weitere Techniken angewandt, welche die Probleme reduzieren und deren Lesbarkeit erhöhen, jedoch keine Auswirkungen auf deren Semantik haben. Zudem wird sichergestellt, dass jeder Minimierungsschritt kontinuierlich validiert wird. Der neuartige Ansatz achtet dabei darauf, eine Balance zwischen dem Minimierungsgrad und dem Erhalt der Grundstruktur der Eingabeprobleme zu gewährleisten. Es wird stets darauf geachtet, dass man kleinere und einfacher verständliche Probleme erhält, um das Debugging und die Verbesserung von SMT-Solvern zu erleichtern. Zudem wird darauf geachtet, dabei weder das Kernproblem stark zu verändern, noch die korrekte Lösung der Probleme zu ändern. Dies ist in dieser Form bei keinen der gängigen Minimierungstools festzustellen, was die Relevanz dieses neuartigen Verfahrens unterstreicht. Im folgenden Abschnitt wird aufgezeigt, aus welchen praktischen Gründen das Minimieren von SMT-Problemen für viele Anwendungsfälle entscheidend ist.

2 Motivation

Higher-Order-Logic-Probleme Van Benthem & Doets (1983) aus der TPTP-Bibliothek Sutcliffe (n.d.) können von SMT-Solvern wie `CVC5` Barbosa et al. (2022) nur teilweise und von `z3` De Moura & Bjørner (2008) in der Regel gar nicht gelöst werden. Darüber hinaus befinden sich in der Bibliothek auch viele komplexe First-Order-Logic-Probleme Barwise (1977), welche SMT-Solver ebenfalls vor Herausforderungen stellen. Viele Ausdrücke in den Problemen tragen nicht zur Änderung des Solver-Outputs bei, erhöhen aber die Komplexität der Eingabe. Daher bietet es sich an, überflüssige Ausdrücke zu entfernen, um die Probleme kleiner und verständlicher zu machen, was das Debugging und die gezielte Verbesserung von SMT-Solvern erleichtert, ohne den korrekten Lösungsstatus zu

verändern. Bisher musste man die Minimierung oft per Hand vornehmen oder auf existierende Tools wie `ddsmt` Kremer et al. (2021) zurückgreifen. `Ddsmt` bietet eine Vielzahl an Minimierungsstrategien, verändert jedoch teils die grundlegende Struktur der ursprünglichen Probleme so stark, dass diese nicht mehr erkennbar ist. Darüber hinaus werden boolsche Operationen und bestimmte Funktionsargumente nicht entfernt, und ein umfassendes Refactoring zur Verbesserung der Lesbarkeit fehlt. All diese Mängel wurden mit unserem neuen Ansatz behoben. Mit Hilfe eines Fixpunkt-Algorithmus wird zudem iterativ nach einer optimalen Vereinfachung gesucht. Ziel dieser Arbeit ist es, Probleme aus der TPTP-Bibliothek so zu vereinfachen, dass eine Lösung für das vereinfachte Problem gefunden werden kann, wobei das vereinfachte Problem dem ursprünglichen Problem sehr ähnlich bleiben soll. Dabei bleibt das zu beweisende Ziel, also die zentrale Aussage und die unterstützenden Axiome, vollständig erhalten. Es werden also ausschließlich unnötig komplexe Ausdrücke entfernt, die weder zur zentralen Aussage beitragen, noch den Beweis unterstützen. Dadurch wird klar ersichtlich, was bewiesen werden soll. Dies hilft Forschern, den Beweisansatz besser zu erkennen und SMT-Solver zu verbessern. Nachdem nun deutlich wurde, warum eine Minimierung dieser Probleme relevant ist, gibt der folgende Überblick einen kompakten Einblick in den Aufbau und die zentralen Inhalte dieser Arbeit.

2.1 Überblick

In dieser Arbeit wird ein neuer Ansatz zur Minimierung von SMT-Problemen aus der TPTP-Bibliothek vorgestellt. Ziel ist es, die Komplexität der Probleme zu reduzieren, indem unter anderem überflüssige Funktionsargumente und unnötige boolsche Operationen entfernt werden, ohne dabei die Grundstruktur der Probleme zu verändern. Durch die Reduktion soll klar ersichtlich werden, was bewiesen werden soll, was Forschern dabei hilft, schneller und einfacher einen Beweisansatz zu finden sowie SMT-Solver zu verbessern. Der Minimierungsprozess erfolgt iterativ mittels eines Fixpunkt-Algorithmus. Zudem wird jeder Änderungsschritt durch Solver-Checks validiert. Anschließend wird ein Refactoring durchgeführt, um die Lesbarkeit der Probleme zu verbessern, indem beispielsweise bestimmte Präfixe entfernt und ungenutzte quantifizierte Variablen angepasst werden. Zur effizienten Verarbeitung mehrerer Probleme wird ein Python-Skript eingesetzt, das viele Minimierungsvorgänge gleichzeitig startet. Die Analyse der erfassten Metriken zeigt unter anderem, dass sich die Effektivität unseres Minimierungsansatzes zwischen Problemen unterschiedlicher Logiken unterscheidet. Im Folgenden werden zunächst die theoretischen Grundlagen vorgestellt, anschließend die Ergebnisse der Analyse präsentiert und schließlich die Implementierung des Verfahrens beschrieben.

3 Hintergrund

3.1 First-Order-Logic vs. Higher-Order-Logic

In der First-Order-Logic (FOL) werden Aussagen über Individuen[1] getroffen. Dabei werden zwar Funktions- und Prädikatssymbole verwendet, Quantoren können jedoch ausschließlich auf Indi-

[1] Mit „Individuen“ sind die konkreten Elemente der betrachteten Domäne gemeint, die in logischen Formeln durch Variablen repräsentiert werden.

viduen angewandt werden. Im Gegensatz dazu erlaubt die Second-Order-Logic Boolos (1975) zusätzlich die Quantifizierung über Prädikate 1. Ordnung und Relationen. Dies kennzeichnet sie als Zwischenstufe zwischen FOL und Higher-Order-Logic (HOL). In HOL ist darüber hinaus auch die Quantifizierung über Prädikate höherer Ordnung möglich, das heißt, es können Prädikate quantifiziert werden, die selbst andere Prädikate als Argument akzeptieren. Diese Erweiterung führt zu einer erheblich höheren Ausdrucksstärke, hat aber auch zur Folge, dass HOL im Allgemeinen als unentscheidbar gilt. Es gibt hier also keinen Algorithmus, der für alle Formeln aus dieser Logik in endlicher Zeit entscheiden kann, ob diese erfüllbar ist oder nicht. FOL-Formeln sind semi-entscheidbar, das heißt, für jede erfüllbare Formel wird in endlicher Zeit ein Beweis gefunden, während die Suche nach einem Beweis für unerfüllbare Formeln im schlimmsten Fall niemals terminiert. SMT-Solver sind größtenteils auf FOL optimiert, weshalb HOL-Formeln oftmals zuerst in FOL-Formeln überführt werden müssen, um überhaupt ein Entscheidungsverfahren zu ermöglichen. Um zu veranschaulichen, wie HOL- sowie FOL-Probleme aufgebaut sind, betrachten wir folgende Beispiele: Ein Beispiel für eine Formel in FOL ist

$$\forall x\, , \exists y.\, R(x,y) \implies R(x, f(x)).$$

Diese Formel enthält die Variablen x und y, wobei es sich um die Individuen innerhalb der Domäne handelt. Zudem kommt ein Funktionssymbol f vor, das eine unäre Funktion darstellt. Das bedeutet, dass die Elemente aus der Domäne wiederum auf die Domäne abgebildet werden. Das Prädikatensymbol R steht für eine binäre Relation zwischen Objekten der Domäne. Die Quantoren $\forall x$ und $\exists y$ beziehen sich ausschließlich auf einzelne Elemente der Domäne Hetzl (2019). Betrachten wir nun ein vereinfachtes Beispiel für ein HOL-Problem:

$$\forall Q, P, x.\, P(x) \implies Q(x).$$

Hier sind P und Q Prädikate höherer Ordnung[2]. Da in der HOL auch über Prädikate quantifiziert werden kann, handelt es sich folglich um ein HOL-Problem. Dieses einfache Beispiel wurde gewählt, da HOL-Probleme sehr komplex sein können und eine detaillierte Erklärung dieser Probleme den Rahmen sprengen würde. Das Beispiel soll lediglich das grundsätzliche Verständnis der HOL erleichtern Hetzl (2019).

3.2 Satisfiability Modulo Theories (SMT)

Das Erfüllbarkeitsproblem der Aussagenlogik (SAT) ist ein fundamentales Problem der Informatik. Ziel ist es, für eine aussagenlogische Formel eine Variablenbelegung zu finden, die die Formel erfüllbar macht. SMT erweitert dieses klassische SAT-Problem mit Theorien über z. B. Zahlen (Integer, reelle Zahlen) oder Datenstrukturen wie Arrays, Listen, Bitvektoren und vieles mehr (siehe Abbildung 1). Ein SMT-Solver kombiniert typischerweise einen SAT-Solver mit speziellen theoriespezifischen Modulen, um die Erfüllbarkeit der Constraints prüfen zu können. Der SMT-Solver erhält dabei eine logische Formel im SMT-LIB-Format (in unserem Fall SMT-LIB v2 Barrett et al. (2010)) als Eingabedatei. Zuerst wird diese an den SAT-Solver weitergegeben, welcher beispielsweise mit einem DPLL-Algorithmus Ernst (2023) oder einem CDCL-Algorithmus Shirokikh et al. (2023)

[2]Prädikate höherer Ordnung sind jene, die entweder quantifiziert oder von einem anderen Prädikat als Argument verwendet werden, oder beides.

nach einer erfüllbaren Belegung der atomaren Propositionen sucht und eine erste Teillösung liefert. Erst wenn eine solche Teillösung auch Theorieteile wie z. B. Arithmetik, Bitvektoren oder Arrays betrifft, werden die zusätzlichen theorie-spezifischen Module aufgerufen, um die jeweilige partielle Belegung auf Widersprüche in der entsprechenden Theorie zu überprüfen. Anschließend wird das Ergebnis aus den theorie-spezifischen Modulen wieder an den SAT-Solver übergeben und erneut überprüft. Dieser Prozess geht so lange vonstatten, bis der Solver-Output, `sat` (erfüllbar), `unsat` (unerfüllbar) oder `unknown` (unbekannt: Tritt bei Timeout auf, wenn das Problem zu schwer ist) generiert wurde Ernst (2023).

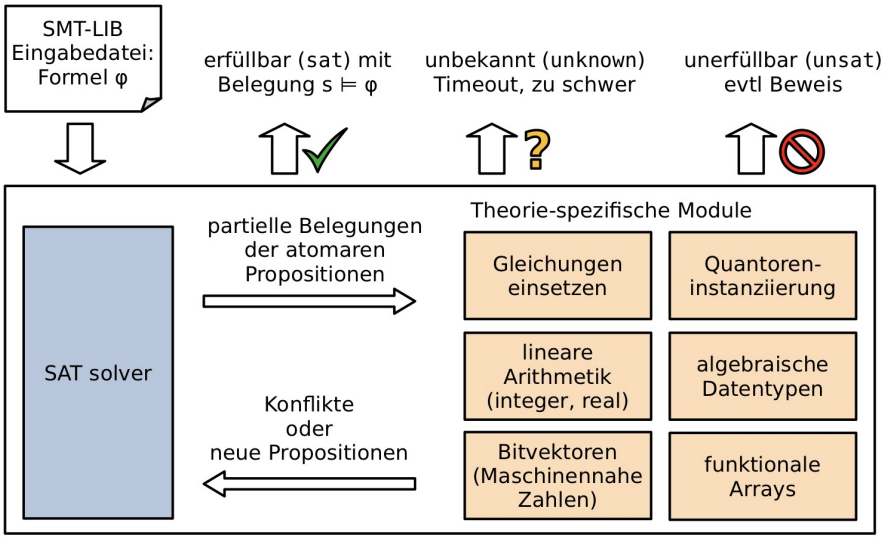


Abbildung 1: Funktionsweise eines SMT-Solvers (Quelle: Ernst (2023))

3.3 Minimierung von SMT-Problemen

Die Minimierung von SMT-Problemen spielt insofern eine wichtige Rolle, da SMT-Probleme in der Praxis sehr groß und komplex sein können, was die Lösungszeit von SMT-Solvern erhöht und die Analyse der gegebenen Probleme sowohl für den Menschen als auch für SMT-Solver erschwert. Bei der Minimierung geht es im Wesentlichen darum, eine gegebene Formel so zu reduzieren und zu vereinfachen, dass alle irrelevanten und überflüssigen Ausdrücke entfernt werden. Der Lösungsstatus (z. B. `unsat`) des ursprünglichen Problems soll dabei jedoch (mit wenigen Ausnahmen) dem des neuen, minimierten Problems entsprechen. Erfüllbare Formeln sollen beispielsweise erfüllbar bleiben und unerfüllbare weiterhin unerfüllbar. Bei Formeln, bei denen der Solver-Output `unknown` ist, wird in unserem Fall nach der Minimierung bei der Solver-Validierung entweder `unknown`, `unsat` oder `sat` akzeptiert, abhängig vom jeweiligen Problem. Des Weiteren wird bei Problemen mit Solver-Output `None` auch der wahre Lösungsstatus (beispielsweise `unsat`) anerkannt (vgl. Abschnitt 4.4). Dies hat mit speziellen Eigenschaften von Problemen aus der TPTP-Bibliothek zu tun, auf welche der neu entwickelte Algorithmus zugeschnitten ist. Im Zentrum der Minimierung steht in unserem Projekt jedoch nicht speziell das Lösen von SMT-Problemen, sondern vor allem das Hervorheben des zu beweisenden Kerns, der Kernaussage der gegebenen SMT-Probleme. Durch die Reduktion soll klar ersichtlich werden, was bewiesen werden soll. Dies kann Forschern dabei helfen,

einen Beweisansatz für ein gegebenes SMT-Problem besser und schneller identifizieren zu können. Auf diese Weise ist es Forschern auch einfacher möglich, Bereiche zu identifizieren, in denen die Solver nicht weiterkommen und diese dahingehend zu optimieren. Um all dies zu erreichen, muss stets darauf geachtet werden, dass, trotz der Reduktion, die relevanten Eigenschaften der Probleme erhalten bleiben.

3.4 Minimierungsstrategien

Eine bekannte Minimierungstechnik ist das sogenannte Delta-Debugging Kremer et al. (2021). Dabei wird die fehlerverursachende Eingabe iterativ mithilfe einer Divide-and-Conquer-Strategie in kleinere Teile zerlegt und nach jedem Schritt überprüft, ob das gleiche Fehlverhalten (z. B. ein Solver-Absturz oder ein falscher Output) noch auftritt. Ziel ist es, ein möglichst kompaktes Minimalbeispiel zu erhalten, das den ursprünglichen Fehler weiterhin reproduziert. Obwohl Delta-Debugging nicht garantiert, den absolut minimalen Input zu liefern, ist es in der Praxis oft ausreichend, um irrelevante Teile der Formel zu entfernen und so die wesentlichen, problemrelevanten Komponenten zu isolieren. So könnten beispielsweise in einem gegebenen Problem die Teile `(assert (> x 12))` und `(assert (< x 4))` isoliert werden. Dies ist möglich, da hier widersprüchliche Bedingungen formuliert werden. Denn es wird zum einen gefordert, dass `x` größer `12` und gleichzeitig kleiner `4` sein soll, was zu einem Fehler führt. Tools wie `ddsmt` setzen hingegen unterschiedliche Strategien zur Minimierung ein, die sich hinsichtlich ihrer Vorgehensweise unterscheiden:

3.4.1 ddmin-Strategie

Diese Strategie basiert auf einer Variante des ursprünglichen Delta-Debugging-Ansatzes. Hierbei werden mehrere S-Expressions[3] zeitgleich in verschiedene Teilmengen partitioniert und parallel vereinfacht. Der Prozess wird iterativ so lange fortgeführt, bis kein weiterer Vereinfachungsschritt mehr möglich ist. Die ddmin-Strategie weist dabei Parallelen zur Tiefensuche auf. Auch hier werden die Minimierungsschritte auf Korrektheit geprüft. Die ddmin-Strategie bringt starke Vorteile hinsichtlich der Verarbeitungsgeschwindigkeit mit sich, jedoch hat dies zur Folge, dass die Probleme eigentlich noch weiter minimiert werden könnten Kremer et al. (2021).

3.4.2 hierarchical-Strategie

Um ein höheres Maß an Minimierung zu erreichen, wird bei dieser Methode auf einen Breitensuchansatz zurückgegriffen. Hier wird die Eingabe systematisch, Ebene für Ebene (also in einer "Breitensuche") durchlaufen. Jede S-Expression wird einzeln betrachtet und mit allen aktiven Mutatoren[4] geprüft. Auf diese Weise können auch tiefere hierarchische Strukturen der Eingabe erfasst und gezielt vereinfacht werden. Dabei erzielt dieser Ansatz in den meisten Fällen zwar stärkere Ergebnisse hinsichtlich der Minimierung, was jedoch eine höhere Laufzeit zur Folge hat Kremer et al. (2021).

[3] Bei S-Expressions handelt es sich um rekursiv geschachtelte Listen, welche unter anderem für die Darstellung der hierarchischen Struktur von Formeln im SMT-LIB-Format verwendet werden.

[4] Bei Mutatoren handelt es sich um Module zur Umsetzung der Minimierungsregeln.

3.4.3 hybrid-Strategie

Um das Beste beider Strategien zu vereinen, wird bei dieser Strategie zunächst die ddmin-Strategie angewandt, um eine erste grobe Reduktion zu erzielen. Anschließend wird die hierarchical-Strategie eingesetzt, um den bereits vereinfachten Input weiter zu reduzieren und in tieferen Ebenen gezielte Minimierungen vorzunehmen Kremer et al. (2021).

Aufbauend auf diesen Grundlagen wird im folgenden Kapitel das eigentliche Vorgehen des Minimierungsverfahrens vorgestellt. So werden nicht nur die Minimierung an sich, sondern auch die dazugehörigen Teilprozesse sowie das Refactoring dargestellt.

4 Methodik

4.1 Parsing

Die im SMT-LIB v2 Format vorliegenden Formeln werden mithilfe eines dafür entwickelten Parsing-Moduls in eine interne Repräsentation in Form von S-Expressions überführt. Die hierarchische Klammerstruktur bildet den Aufbau der logischen Ausdrücke explizit ab und eröffnet so die Möglichkeit einer strukturierten Analyse und Manipulation der Formeln. Als Beispiel nehmen wir einen Teil des Problems `SYO001^1` im SMT2-Format:

```
(set-info :tptpstatus Theorem)
(declare-sort $$unsorted 0)
...
```

Nachdem der Parsing-Prozess durchgeführt wurde, wird das Problem wie folgt intern repräsentiert:

```
[[Symbol('set-info'), Symbol(':tptpstatus'), Symbol('Theorem')],
   [Symbol('declare-sort'), Symbol('$$unsorted'), 0]]
```

Auf dieser internen Repräsentation des Problems werden dann alle im Folgenden beschriebenen Schritte, wie beispielsweise die Minimierung oder das Refactoring, angewandt.

4.2 Minimierung

Hat das vorliegende Problem einmal den Parsing-Prozess durchlaufen, kommt der Minimierungsprozess ins Spiel. Dieser besteht einerseits aus der Entfernung von gewissen Funktionsargumenten und andererseits aus der Modifikation boolscher Operationen innerhalb der Probleme. Ziel der Minimierung ist es letztendlich, das gegebene Problem so zu vereinfachen, dass eine Lösung für ein vereinfachtes Problem gefunden werden kann. Dieses vereinfachte Problem soll dabei dem ursprünglichen Problem sehr ähnlich sein.

4.2.1 Entfernung von Funktionsargumenten

Bei dieser Methode kann ein Argument dann entfernt werden, wenn bereits ein anderes mit demselben Typ vorhanden ist und die Solver-Validierung dies erlaubt (vgl. Abschnitt 4.4). So könnte beispielsweise

```
(declare-fun tptp.diffprop (tptp.nat tptp.nat tptp.nat) Bool)
```

auf diesen Ausdruck reduziert werden:

```
(declare-fun tptp.diffprop (tptp.nat) Bool)
```

Es bleibt hier also beispielsweise nur noch ein `tptp.nat` als Argument erhalten. Die anderen beiden Argumente wurden entfernt, da sie gleichen Typs waren. Eine solche Entfernung lassen wir bewusst zu, da so die grundlegende Struktur des ursprünglichen Problems trotzdem erhalten bleibt. Wird eine solche Änderung in der Funktionsdeklaration durchgeführt, so wird diese auch auf alle weiteren Vorkommen der Funktion innerhalb des Problems angewandt. Es wird zudem immer sichergestellt, dass niemals alle Argumente entfernt werden. Es bleibt also immer mindestens ein Argument übrig, es sei denn, die Funktion hatte schon im Ursprungsproblem keinerlei Argumente. Hat eine Funktion mehrere Argumente verschiedenen Typs, so wird sichergestellt, dass von jedem Typ auch immer mindestens ein Argument erhalten bleibt, wie in diesem Beispiel zu sehen ist:

```
(declare-fun tptp.diffprop (tptp.nat tptp.nat Bool Bool) Bool)
```

kann auf diesen Ausdruck reduziert werden:

```
(declare-fun tptp.diffprop (tptp.nat Bool) Bool)
```

Auch hier werden wieder alle weiteren Vorkommen der Funktion innerhalb des Problems entsprechend angepasst. Es wurde zudem sichergestellt, dass von jedem Argumenttyp mindestens ein Argument erhalten bleibt. Wenn eine Funktion wiederum eine Funktion als Argument besitzt, wie beispielsweise hier zu sehen ist: `(declare-fun tptp.some ((-> tptp.nat Bool))Bool)`, so werden keinerlei Minimierungen vorgenommen, da dies die übergeordnete Struktur des Problems stark verändern und verfälschen würde. Das Entfernen bestimmter Funktionsargumente ist entscheidend dafür, das Problem einerseits zu verkleinern und andererseits den Aufbau des Ursprungsproblems weiterhin möglichst gut abzubilden. Der Minimierungsansatz weist daher eine hohe Balance zwischen der Minimierung an sich und dem Erhalt der Problemstruktur auf. Zudem gibt es zwei verschiedene Vorgehensweisen, um diese Minimierung durchzuführen, welche abhängig von der Größe der gegebenen Probleme zum Einsatz kommen:

Optimaler Ansatz Hier werden alle möglichen Kombinationen an möglichen Änderungen generiert und getestet, beziehungsweise durch einen Solveraufruf überprüft. Dadurch, dass zuerst alle Möglichkeiten an verschiedenen Funktionsdeklarationen erzeugt werden und erst anschließend die minimale, vom Solver validierte Lösung verwendet wird, ist dieser Ansatz optimal, wenn auch sehr rechenintensiv. Daher wird diese Vorgehensweise nur bei sehr kleinen Problemen mit wenigen Parametern angewandt. Es gäbe ansonsten zu viele Kombinationen, was die Rechenzeit in erheblichem Maße negativ beeinflussen würde. Dies könnte im schlimmsten Fall dazu führen, dass das Programm niemals terminiert.

Greedy-Ansatz Aufgrund des oben beschriebenen Problems bezüglich der hohen Rechenintensität kommt der Greedy-Ansatz bei Problemen, die eine gewisse Schwelle an Parametern überschreiten, zum Einsatz. Hier wird nicht nach einer global optimalen Lösung gesucht, sondern es wird schrittweise geprüft, ob einzelne Funktionsargumente entfernt werden können. Anstatt alle möglichen Kombinationen auszuprobieren, wird jeweils die erste Änderung übernommen, die sich als verträglich erweist. So wird in jedem Schritt ein Teil der Formel vereinfacht, was zu einer schnelleren

und weniger rechenintensiven Minimierung führt, wodurch das Ergebnis aber auch möglicherweise nicht absolut minimal sein kann.

Zusammengefasst erfolgt die Entfernung eines Funktionsarguments nur dann, wenn mindestens ein Argument desselben Typs erhalten bleibt und die Änderung durch wiederholte Solver-Checks bestätigt wird, sodass der wesentliche Kern des Problems vollständig erhalten bleibt.

4.2.2 Modifikation boolscher Operationen

Um die Komplexität von SMT-Formeln weiter zu reduzieren, werden Disjunktionen, Konjunktionen, Implikationen sowie Gleichheitsoperationen modifiziert. Dabei können sowohl die Operatoren `and`, `or`, `=>` und `=` als auch die dazugehörigen Operanden geändert werden. Die Reduktion boolscher Operationen ist essenziell, da diese häufig zu unnötig komplexen logischen Strukturen führen, die den Suchraum des Solvers erheblich vergrößern und somit die Effizienz der Solver beeinträchtigen. Daher hilft die Reduktion dieser Operationen Forschern dabei, einen Beweisansatz zu finden und die Solver zu optimieren. Auch hier unterscheidet man zwischen zwei Ansätzen:

Optimaler Ansatz Bei diesem Ansatz werden für boolsche Operationen mit mehr als zwei Operanden alle möglichen Kombinationen der zu entfernenden Operanden systematisch erzeugt und geprüft. Jede Variante wird durch den Solver validiert, sodass nur diejenigen Modifikationen übernommen werden, die den ursprünglichen Lösungsstatus beibehalten (es sei denn, der Lösungsstatus war `unknown` oder `None`). Obwohl diese Methode das theoretisch optimale Ergebnis liefert, ist sie aufgrund der exponentiell wachsenden Anzahl möglicher Kombinationen nur bei kleineren Problemen praktikabel. Des Weiteren wird auch sichergestellt, dass niemals alle Operanden entfernt werden. Eine Modifikation bei `and`, `or` oder `=` läuft dabei immer wie folgt ab: Entweder der "linke" oder der "rechte" Operand wird verworfen. Falls dies jedoch nicht möglich ist, weil die Änderung vom Solver nicht validiert wird, wird keine Änderung durchgeführt. Der Operator wird ebenfalls entfernt, sofern es die Solver-Validierung (siehe Abschnitt 4.4) zulässt. Dieser Fall tritt jedoch nur dann auf, wenn nur zwei Operanden vorhanden sind und einer von beiden entfernt werden könnte. Es sei zudem erwähnt, dass bei `=` im Unterschied zu den anderen Operatoren noch geprüft wird, ob der Rückgabewert der Operanden `Bool` ist, da es andernfalls keinen Sinn ergibt, eine solche Modifikation durchzuführen. Bei der Implikation verhält es sich etwas anders. Hier wird entweder dem "linken" Operand ein `not` davorgesetzt und der "rechte" Operand, mitsamt Operator, entfernt. Andernfalls wird der "linke" Operand, einschließlich des Operators, entfernt und der "rechte" Operand beibehalten, oder es wird gar keine Änderung durchgeführt. Die spezielle Behandlung von Implikationen ist dabei dem Umstand geschuldet, dass sich beispielsweise der Ausdruck $A \implies B$ in $\neg A \vee B$ umformen lässt.

Greedy-Ansatz Im Gegensatz zum optimalen Ansatz, der alle möglichen Kombinationen systematisch durchprobiert, wählt der Greedy-Ansatz iterativ und schrittweise die erste verträgliche Änderung aus. Auch hier wird bei jeder möglichen Änderung eine Solver-Validierung durchgeführt. Jede boolsche Operation wird einzeln betrachtet und schrittweise modifiziert, um eine sofortige Komplexitätsreduktion zu erzielen. Die Modifizierung der Operationen an sich unterscheidet sich dabei nicht von dem oben beschriebenen optimalen Ansatz. Bei Operationen mit zwei Operanden, bei denen man theoretisch entweder die eine oder die andere "Seite" entfernen könnte, wird die

“Seite“ entfernt, die eine größere Komplexität aufweist beziehungsweise deren Entfernung zu einer stärkeren Minimierung des Problems führen würde. Die Operatoren werden, falls möglich, ebenfalls wie oben beschrieben mitentfernt. Bei Operationen mit mehr als zwei Operanden (tritt in der Regel nur bei `and` sowie bei `or` auf) wird hingegen iterativ versucht, einzelne Operanden zu entfernen, um so den Ausdruck schrittweise zu optimieren. Der Ansatz wird iterativ so lange fortgeführt, bis keine weitere lokale Verbesserung erzielt werden kann. Dabei arbeitet diese Strategie zwar deutlich schneller als der optimale Ansatz, liefert jedoch nicht immer das absolut minimale Ergebnis. Für große und komplexe Probleme bietet sie jedoch einen guten Kompromiss zwischen Effizienz und Minimierungsgrad.

4.2.3 Beispiele

Um die Modifikation boolscher Operationen zu veranschaulichen, betrachten wir folgende Beispiele:

Modifikation von Disjunktionen In diesem fiktiven Beispiel könnte die assert-Bedingung des Ausdrucks:

```
...
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(assert (or A B C))
...
```

in `(assert (or A B))`, `(assert (or B C))`, `(assert (or A C))`, `(assert B)`, `(assert A)` oder `(assert C)` umgewandelt werden. Falls dies jedoch aufgrund der Solver-Validierung nicht möglich ist, bleibt der Ausdruck unverändert. Bei `and` sowie = verhält es sich analog.

Modifikation von Implikationen Die in diesem Beispiel dargestellte Implikation:

```
...
(declare-const A Bool)
(declare-const B Bool)
(assert (=> A B))
...
```

könnte man beispielsweise in `(assert (not A))` oder `(assert B)` umwandeln. Falls dies jedoch wegen der Solver-Validierung nicht möglich ist, bleibt der Ausdruck auch hier unverändert.

4.3 Fixpunkt-Algorithmus

Um den Grad der Minimierung zu erhöhen, wird ein iterativer Fixpunkt-Algorithmus eingesetzt, der in jedem Iterationsschritt den aktuellen, vereinfachten Ausdruck mit dem Ergebnis der vorherigen Runde vergleicht. In jedem Schritt wird versucht, erneut eine Minimierung durchzuführen, bis keine Änderungen mehr möglich sind beziehungsweise bis der Fixpunkt erreicht wurde. Um letzteres festzustellen, werden die fertigen Ausdrücke rekursiv miteinander verglichen und auf Gleichheit überprüft. In diesem Projekt wurde die maximale Anzahl an Iterationen auf 10 gesetzt. Es wäre daher möglich, dass noch weitere Minimierungen durchführbar wären, die aufgrund

dieser Limitierung (aus Performancegründen) nicht erfasst und durchgeführt werden können. Der Algorithmus terminiert dann, wenn entweder ein Fixpunkt gefunden oder die maximale Anzahl an Iterationen erreicht wurde.

4.4 Solver-Validierung

Um sicherzustellen, dass die vorgenommenen Minimierungsschritte korrekt sind, wird nach jeder Modifikation eine automatische Validierung durch einen SMT-Solver (in unserem Fall `CVC5` oder `z3`) durchgeführt. In der TPTP-Bibliothek wird für jedes Problem der Lösungsstatus, beispielsweise `Satisfiable` oder `Unsatisfiable`, festgehalten. Dieser Lösungsstatus dient dabei unter anderem der korrekten Durchführung der Minimierung. Beim Parsing des Problems wird der Lösungsstatus erstmalig erfasst. Außerdem wird vor der Minimierung ein Solveraufruf mit dem Ursprungsproblem durchgeführt. Bei Problemen, die sowohl in der TPTP-Bibliothek als auch gemäß dem Solver-Output als erfüllbar oder unerfüllbar klassifiziert werden, wird eine Minimierung nur dann akzeptiert, wenn der Solver-Output des minimierten Problems mit dem des ursprünglichen Problems sowie mit dessen TPTP-Lösungsstatus übereinstimmt. Bei Problemen, bei denen der initiale Solver-Aufruf `unknown` ausgibt, was in unserem Fall nur bei Problemen mit TPTP-Lösungsstatus `Theorem` eintritt, verhält es sich etwas anders. Bei Problemen mit TPTP-Lösungsstatus `Theorem` handelt es sich eigentlich um unerfüllbare Probleme, jedoch sind `CVC5` und `z3` in vielen Fällen nicht in der Lage, diese zu lösen, weshalb `unknown` ausgegeben wird. Aus diesem Grund wird hier sowohl `unknown` als auch `unsat` für die Validierung des Minimierungsschritts akzeptiert. Analog verhält es sich bei erfüllbaren Problemen, bei denen der Solver `unknown` ausgibt. Hier wird sowohl `unknown` als auch `sat` akzeptiert. In jedem der oben beschriebenen Minimierungsverfahren kommt diese Solver-Validierung zum Einsatz. Um sicherzustellen, dass die Minimierung das gewünschte Verhalten zeigt, wird daher bei jeder möglichen Änderung sofort das soeben beschriebene Verfahren angewandt und geprüft, ob die Änderung möglich ist. Darüber hinaus können auch Probleme, bei denen der Solver aufgrund eines Timeouts `None` ausgibt, minimiert werden. So kann beispielsweise ein `unsat`-Problem, bei dem der Solver `None` ausgibt, so reduziert werden, dass der Solver nun für das Problem den wahren Lösungsstatus, also `unsat`, ausgibt.

4.5 Timeouts

Für die Minimierung sowie für die Solver-Validierung kommen mehrere Timeouts zum Einsatz, da die Probleme aus der TPTP-Bibliothek zum Teil sehr groß und komplex sein können und dadurch die Berechnung des Lösungsstatus unter Umständen sehr lange dauern kann. Zum einen gibt es einen initialen Timeout, der für die erstmalige Ermittlung des Lösungsstatus des Ursprungsproblems festgelegt wird. Schafft es der Solver nicht, den initialen Lösungsstatus innerhalb dieser Zeit zu ermitteln, wird das Problem nicht weiter beachtet und es wird zum nächsten Problem übergegangen. Eine Minimierung oder gar ein Refactoring wird bei so einem Problem dann nicht durchgeführt. Dies tritt jedoch nur in sehr seltenen Fällen bei sehr großen und komplexen Problemen auf. Es wird so sichergestellt, dass solche Probleme frühzeitig aussortiert werden. Zusätzlich wird ein separater Timeout für jeden Minimierungsschritt und die anschließende Solver-Validierung verwendet. Überschreitet der Solver einen vorgegebenen Zeitraum (z. B. weil die Lösungsfindung sehr lange dauert), wird der jeweilige Minimierungsschritt als gescheitert markiert und nicht übernommen. Darüber hinaus ist auch eine maximale Anzahl an Timeout-Ereignissen definiert. Wird diese Grenze

erreicht, bricht der gesamte Minimierungsprozess bei dem jeweiligen Problem ab, um die Wartezeit in Grenzen zu halten. Die Timeouts stellen sicher, dass die Ressourcen effizient genutzt werden und dass der Minimierungsprozess nur auf Probleme angewendet wird, die in einem praktikablen Zeitrahmen bearbeitet werden können. In diesem Projekt wurde der initiale Timeout auf 60 Sekunden, der Timeout für die Minimierungsschritte auf 10 Sekunden sowie die maximale Anzahl an Timeouts auf 10 gesetzt. Diese Werte lassen sich aber nach Belieben ändern.

4.6 Refactoring

Nachdem die Minimierung abgeschlossen wurde, wird darauffolgend ein Refactoring durchgeführt, wobei hier das Ziel vorwiegend in der Verbesserung der Lesbarkeit der Probleme liegt. Dabei werden Präfixe wie `tptp.` oder `$$` aus den Problemen entfernt. So wird beispielsweise der Ausdruck `(declare-fun tptp.diffprop (tptp.nat tptp.nat tptp.nat)Bool)` in `(declare-fun diffprop (nat nat nat)Bool)` umgewandelt oder der Ausdruck `(declare-sort $$unsorted 0)` in `(declare-sort unsorted 0)` geändert. Häufig kommt es vor, dass eine Variable durch den Minimierungsprozess vollständig entfernt werden müsste, jedoch weiterhin von einem Quantor (wie `forall` oder `exists`) gebunden bleibt, wie man z. B. hier sieht: `(forall ((Xz nat)(Xx nat)(Xy nat))(=> (@ (@ lessis Xx)Xy)(@ (@ moreis Xy)Xx)))`. Dabei wird `Xz` quantifiziert, aber nicht genutzt, weshalb `Xz` beziehungsweise `(Xz nat)` entfernt werden kann. Das Problem wird durch diese beiden Prozesse zusätzlich verkleinert, sodass man es in gewisser Hinsicht auch als Teil der Minimierung betrachten kann. Zudem werden beim Refactoring alle Vorkommen der genannten Präfixe sowie der Quantoren im Problem erfasst und gegebenenfalls angepasst. Es wird außerdem sichergestellt, dass, wenn nach dem Entfernen eines Präfixes ein reserviertes SMT-LIB v2-Schlüsselwort übrigbleibt, dieses so modifiziert wird, dass sich die Semantik des Problems nicht ändert. Sollte beispielsweise `and` nach dem Entfernen eines Präfixes (z. B. `tptp.`) übrigbleiben, wird diesem ein Unterstrich angehängt (also `and_`). Auf diese Weise bleibt die Semantik des bereits minimierten Problems erhalten, sodass auch keine zusätzliche Solver-Validierung erfolgen muss.

4.7 Zentraler Algorithmus zur Minimierung von SMT2-Dateien

Im Folgenden wird der zentrale Algorithmus zur Minimierung von SMT2-Dateien vorgestellt. Das Verfahren beginnt damit, dass die Eingabedatei im SMT-LIB v2 Format geparst wird, um sie in eine interne Repräsentation in Form von S-Expressions zu überführen. Anschließend wird ein erster Solver-Check durchgeführt, um den Ausgangsstatus des Problems (z. B. `unsat` oder `sat`) sowie den in der TPTP-Bibliothek festgehaltenen Lösungsstatus zu ermitteln. Dieser Ausgangsstatus dient als Referenz für alle nachfolgenden Minimierungsschritte. Auf diesem Ausgangspunkt wird dann iterativ der Minimierungsprozess angewendet. Zunächst wird E' mit dem ursprünglichen Ausdruck E initialisiert. In wiederholten Iterationen werden verschiedene Minimierungsoperationen durchgeführt. Dabei werden beispielsweise überflüssige Funktionsargumente entfernt und boolsche Operationen reduziert. Dies erfolgt so lange, bis festgestellt wird, dass keine weiteren Veränderungen eintreten oder eine festgelegte maximale Anzahl an Iterationen erreicht ist. In jedem Iterationsschritt erfolgt ein weiterer Solver-Check, um zu überprüfen, ob der Lösungsstatus des minimierten Ausdrucks dem ursprünglichen Status entspricht (mit Ausnahmen vgl. Abschnitt 4.4). Erst wenn dies der Fall ist, wird abschließend ein Refactoring vorgenommen, wobei beispielsweise

bestimmte Präfixe sowie ungenutzte quantifizierte Variablen entfernt werden. Anschließend wird der finale, minimierte Ausdruck ausgegeben.

Algorithm 1 Minimierung von SMT2-Dateien

$E \leftarrow$ parse(input_file)
$s_{\text{valid}} \leftarrow \{\text{call\_solver}(E), \text{get\_tptp\_status}(E)\}$
$E' \leftarrow E$
$count \leftarrow 0$
repeat
  $E_{\text{alt}} \leftarrow E'$
  $E' \leftarrow \text{minimize\_func\_args}(E', s_{\text{valid}})$
  $E' \leftarrow \text{minimize\_bool\_ops}(E', s_{\text{valid}})$
  $count \leftarrow count + 1$
until $\text{deep\_compare}(E', E_{\text{alt}})$ is True $\vee\ count \geq 10$
if $\text{call\_solver}(E') = s_{\text{valid}}$ **then**
  $E'' \leftarrow \text{refactor}(E')$
  output E''
else
  report error
end if

4.8 Anwendungsbeispiel

Betrachtet wird das Problem `NUN019+1` in seiner ursprünglichen Form vor der Minimierung und dem Refactoring. Zur besseren Verständlichkeit und zur Veranschaulichung wurden lediglich die Funktionsnamen leicht angepasst. Alles weitere blieb unverändert.

```
(set-info :tptpstatus Satisfiable)
(declare-sort $$unsorted 0)
(declare-fun tptp.s ($$unsorted) $$unsorted)
(declare-fun tptp.g ($$unsorted $$unsorted) Bool)
(assert (forall ((X $$unsorted)) (tptp.g (tptp.s X) X)))
(assert (forall ((X $$unsorted) (Y $$unsorted)) (=> (tptp.g X Y)
   (tptp.g (tptp.s X) Y))))
(assert (forall ((X $$unsorted) (Y $$unsorted)) (=> (tptp.g X Y)
   (not (= X Y)))))
(set-info :filename NUN019+1)
(check-sat)
```

Nachdem der Minimierungs- sowie der Refactoring-Prozess durchgeführt wurde, hat sich das Problem auf diesen Ausdruck reduziert:

```
(set-info :tptpstatus Satisfiable)
(declare-sort unsorted 0)
(declare-fun s (unsorted) unsorted)
(declare-fun g (unsorted) Bool)
(assert (forall ((X unsorted)) (g (s X))))
```

```
(assert (forall ((X unsorted)) (not (g X))))
(assert (forall ((X unsorted)) (not (g X))))
(set-info :filename NUN019+1)
(check-sat)
```

Die Entfernung bestimmter Funktionsargumente, überflüssiger Präfixe und unnötiger boolscher Operationen macht das ursprüngliche Problem kompakter und dadurch leichter zu verstehen. In unserem Anwendungsbeispiel wurde ein Funktionsargument (`$$unsorted`) der Funktion `tptp.g` entfernt. Darüber hinaus wurden die Implikationen aufgelöst, sodass nur noch der "linke", negierte Operand erhalten blieb. Die Anzahl quantifizierter Variablen wurde im Rahmen des Refactorings ebenfalls angepasst. Zudem wurden die Präfixe `tptp.` sowie `$$` entfernt. Das nun vereinfachte Problem weist dabei dennoch eine hohe Ähnlichkeit zu dem ursprünglichen Problem auf. Unser Ziel, ein einfacheres, jedoch der ursprünglichen Eingabe sehr ähnliches Problem zu finden, wurde hiermit erreicht.

Nach der Darstellung der Methodik folgt nun die Analyse der Minimierungsergebnisse

5 Auswertung und Analyse

Im Folgenden wird gezeigt, wie sich die im Rahmen des Minimierungsprozesses aggregierten Daten hinsichtlich der Veränderung der Funktionsargumente, booleschen Operationen und Timeouts interpretieren und in Zusammenhang mit den unterschiedlichen Problemkategorien (HOL und FOF) einordnen lassen.

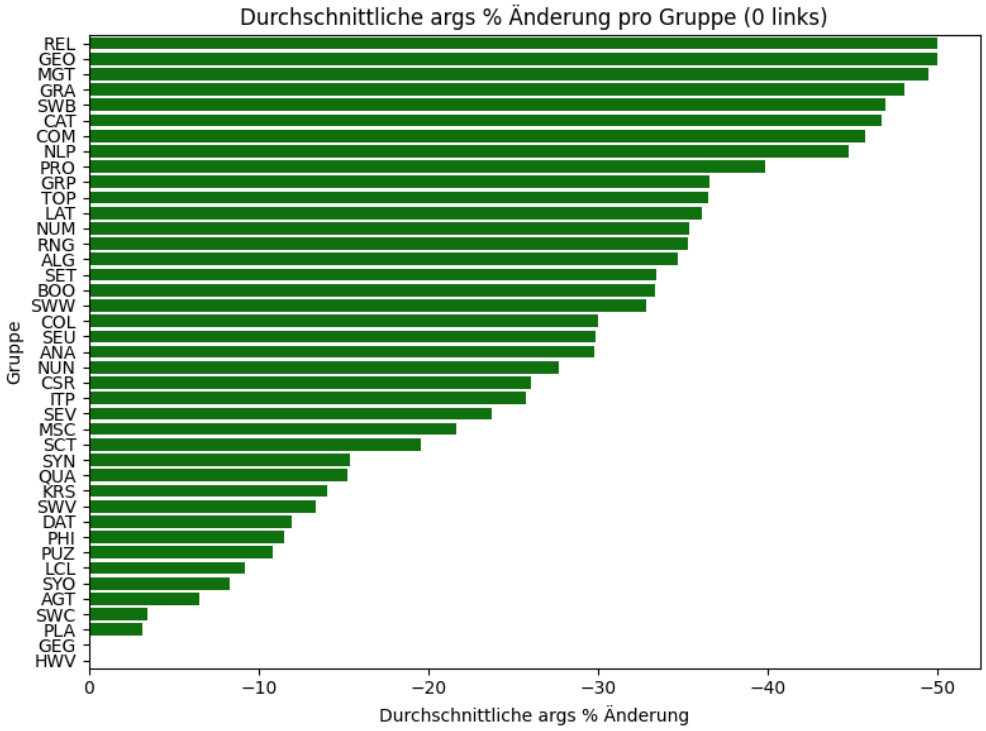


Abbildung 2: Durchschnittliche prozentuale Änderung der Funktionsargumente

Da es sich um tausende relevante Probleme handelt, wurde pro Gruppe ausgewertet. Auffällig sind dabei die Gruppen `REL` sowie `GEO`. Hier konnten durchschnittlich 50% der Funktionsargumente entfernt werden. Des Weiteren sind die Gruppen `GEG` und `HWV` insofern auffällig, als dass dort keinerlei Funktionsargumente entfernt werden konnten (siehe Abbildung 2). Insgesamt wurden 6673 Probleme reduziert. Davon waren 3780 FOF- und 2893 HOL-Probleme. Der Minimierungsgrad hinsichtlich der Anzahl an boolschen Operatoren lässt sich in Abbildung 3 gut ablesen. Auffällig ist hier zusätzlich, dass es bei einer Vielzahl von Gruppen, wie beispielsweise `AGT` oder `SWC`, möglich war, die relevanten boolschen Operatoren[5] komplett zu entfernen. Zudem ist auch eine Vielzahl von Gruppen festzustellen, bei denen annähernd alle relevanten boolschen Operatoren entfernt werden konnten. Im Gegensatz zur Entfernung von Funktionsargumenten lässt sich bei diesem Teilprozess der Minimierung festhalten, dass er bei jeder der Problemgruppen definitiv zum Einsatz kam. Selbst in der "schlechtesten" Gruppe `COL` konnten durchschnittlich 60% der boolschen Operationen aufgelöst werden. Betrachten wir nun die Anzahl der Timeouts je Problemgruppe, die während der Ausführung des Minimierungsprozesses aufgetreten sind (siehe Abbildung 4).

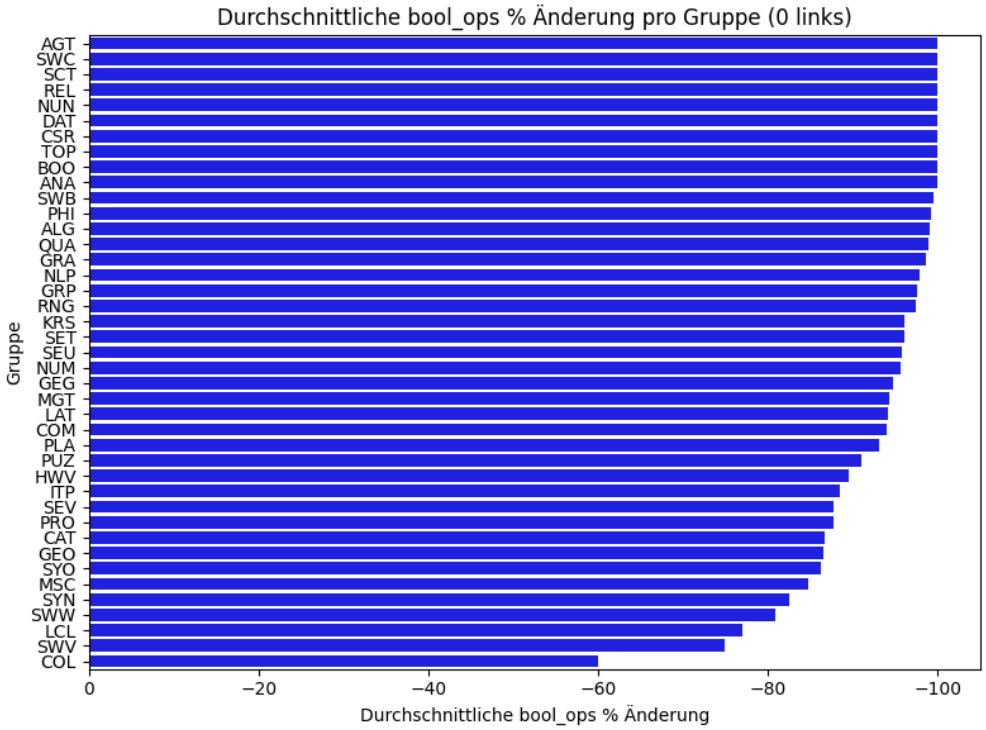


Abbildung 3: Durchschnittliche prozentuale Änderung der boolschen Operatoren

Dabei springen einem insbesondere die Problemgruppen `SEU`, `SYO` sowie `LCL` ins Gesicht, bei welchen es im Durchschnitt zu außergewöhnlich vielen Timeouts kam. Dies lässt sich mit den teilweise sehr großen und komplexen Problemen, welche sich in diesen Gruppen befinden, begründen. Die Gruppen `ANA`, `BOO` und `NUN`, sowie einige weitere Gruppen, zeigten keinerlei Timeouts, da sich in diesen Gruppen viele kleine und wenig komplexe Probleme befinden.

[5] Die relevanten boolschen Operatoren sind in unserem Fall `and`, `or`, `=>` und `=`.

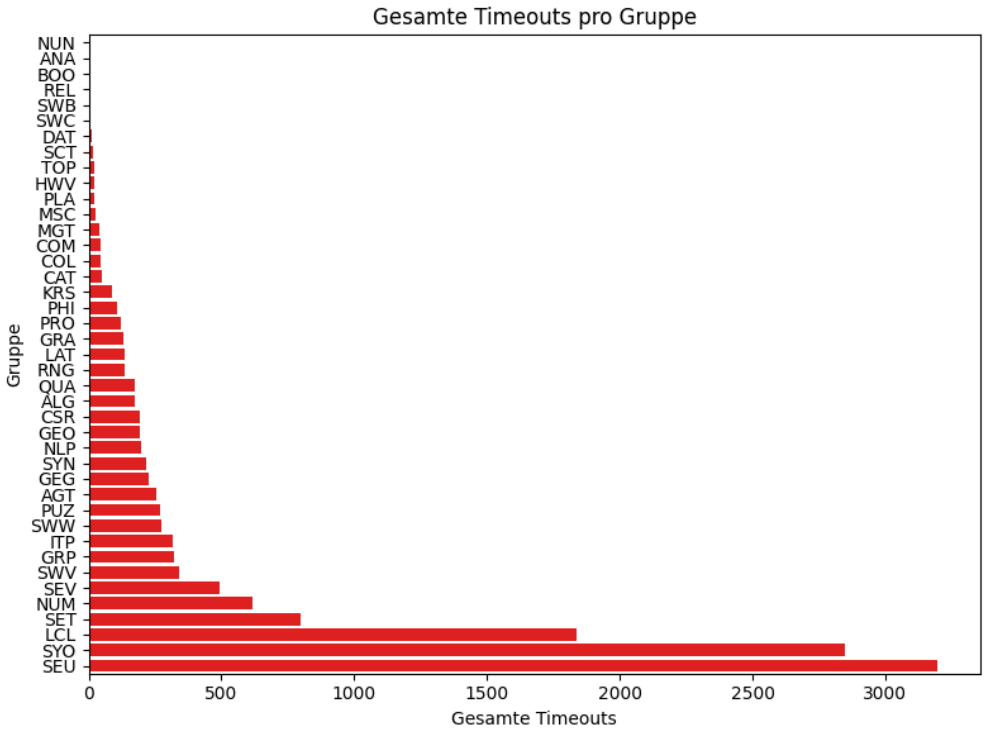
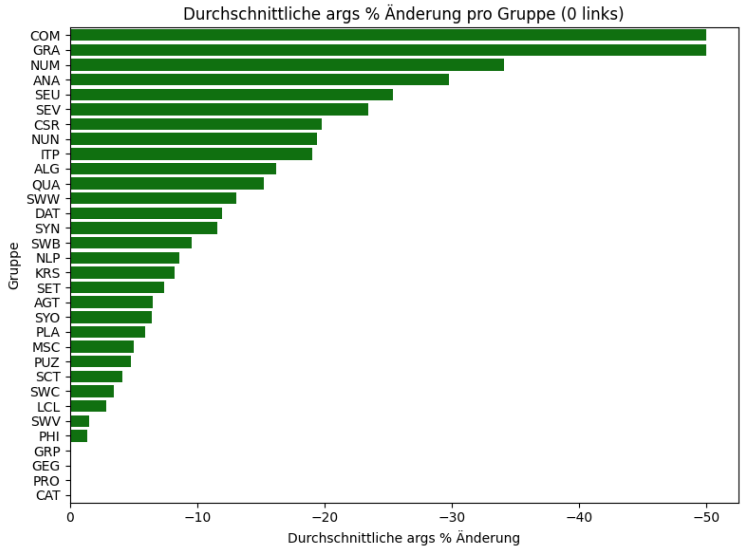
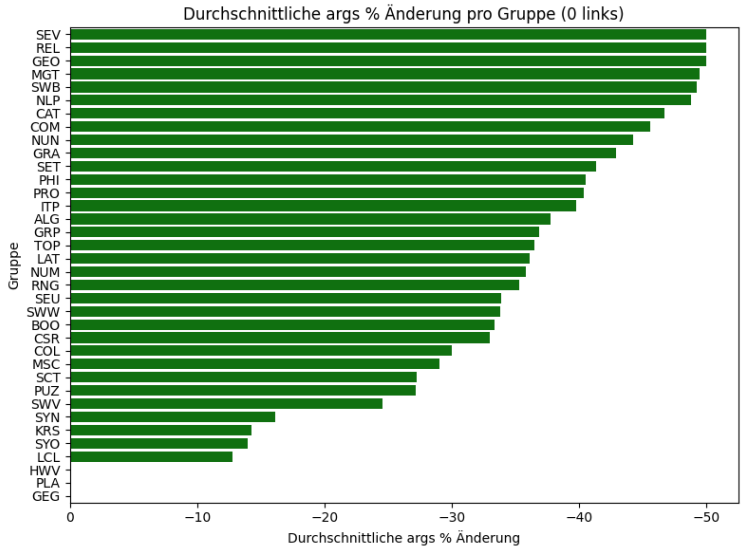


Abbildung 4: Anzahl an Timeouts je Problemgruppe

Die oben gezeigten Statistiken beziehen sich auf alle untersuchten Probleme. Daher bietet es sich an, nun die Ergebnisse nach Logik zu unterscheiden. Während sich bei einigen Metriken nur geringe Unterschiede zwischen FOF- und TH0-Problemen zeigen, fällt besonders auf, dass der durchschnittliche Reduktionsgrad der Funktionsargumente bei FOF-Problemen deutlich höher ist als bei TH0-Problemen (siehe Abbildung 5).



(a) Durchschnittliche prozentuale Änderung der Funktionsargumente bei TH0-Problemen



(b) Durchschnittliche prozentuale Änderung der Funktionsargumente bei FOF-Problemen

Abbildung 5: Vergleich der Funktionsargument-Reduktion bei TH0- und FOF-Problemen

Nach der Auswertung und Analyse der Ergebnisse werden im nächsten Abschnitt konkrete Details zur Implementierung des Projekts erläutert.

6 Implementierung

6.1 SMT-LIB v2 Standardformat

Bei SMT-LIB v2 handelt es sich um ein standardisiertes Eingabeformat für SMT-Solver, welches dafür entwickelt wurde, um eine einheitliche Darstellung von SMT-Problemen zu ermöglichen. Die Syntax basiert dabei auf Lisp-ähnlichen S-Expressions, die eine hierarchische Klammerstruktur aufweisen. Des Weiteren deckt das SMT-LIB v2 eine Vielzahl an Theorien und Logiken wie z. B. `QF_LIA` für quantorfreie lineare Ganzzahlarithmetik u. v. m. ab und ist zudem erweiterbar. Darüber hinaus wird das Format auch beispielsweise im jährlichen SMT-COMP-Wettbewerb Barrett et al. (2013) verwendet, wobei die Leistungsfähigkeit verschiedener SMT-Solver verglichen wird. In dem hier beschriebenen Verfahren werden ausschließlich in diesem Format vorliegende Probleme minimiert. Daher ist es essenziell, sich über die Syntax sowie die Funktionsweisen dieses Standards im Klaren zu sein. Um die Syntax des SMT-LIB v2 Standards zu veranschaulichen, betrachten wir folgendes Beispiel:

```
(set-logic QF_LIA)
(declare-const x Int)
(assert (> x 10))
(check-sat)
```

Zu Beginn wird festgelegt, welche Logik verwendet wird. In diesem Fall wird `QF_LIA` verwendet. Im Anschluss daran wird eine ganzzahlige Konstante `x` deklariert. Darauffolgend wird die Bedingung aufgestellt, dass `x` größer als `10` sein muss. Zum Schluss wird diese Bedingung dann auf Erfüllbarkeit geprüft. Beachtet werden sollte dabei, dass hier alle Operatoren in Präfixnotation geschrieben werden. Der Operator steht also immer vor seinen Operanden.

6.2 TPTP-Bibliothek

Da sich diese Arbeit ausschließlich auf die Minimierung und das Refactoring von Problemen aus der TPTP-Bibliothek (Thousands of Problems for Theorem Provers) fokussiert, ist es sinnvoll, diese Bibliothek genauer zu beleuchten. Bei der TPTP-Bibliothek handelt es sich um eine umfangreiche Sammlung standardisierter Testprobleme für automatisierte Theorem-Beweis-Systeme (ATPs, *automated theorem proving systems*). Sie wird von der ATP-Forschungscommunity seit vielen Jahren gepflegt und wurde, unter anderem, um vergleichbare Benchmarking- und Evaluationsmöglichkeiten zu schaffen, ins Leben gerufen. Die Bibliothek umfasst tausende Probleme, welche alle in einer einheitlichen Sprache (der TPTP-Syntax) verfasst und nach Schwierigkeitsgrad sowie Logik-Kategorie organisiert (z. B. FOF für Formeln erster Ordnung, CNF für Klauselnormalform, THF für höhere Ordnung usw.) sind. Die Bereitstellung einer solchen standardisierten Bibliothek ermöglicht dabei nicht nur den objektiven Vergleich bestehender ATP-Systeme, sondern liefert zudem auch eine solide Grundlage für die Evaluation und Weiterentwicklung neuer Beweistechniken. Denn so können Forscher bestehende ATP-Systeme vergleichen und deren Schwächen identifizieren, was wiederum

als Grundlage für die Entwicklung und Verbesserung neuer Techniken dient. Das hier vorgestellte Minimierungsverfahren ist dabei ausschließlich auf die TPTP-Bibliothek zugeschnitten.

6.3 Konvertierung von TPTP-Problemen

Da die in diesem Projekt verwendeten SMT-Solver (`CVC5` und `z3`) Probleme nur dann erfolgreich verarbeiten können, wenn diese im SMT2-Format vorliegen, müssen zu Beginn die Probleme aus der TPTP-Bibliothek in dieses Format überführt werden. Dies ist der Tatsache geschuldet, dass die Probleme aus dem TPTP-Ordner in einem anderen Format vorliegen. Mithilfe von `CVC5` lassen sich die TPTP-Probleme jedoch in das gewünschte Format umwandeln. Um nicht tausende Probleme einzeln umwandeln zu müssen, wurde ein Shell-Skript entwickelt, das den TPTP-Ordner mit den darin befindlichen Problemen systematisch durchläuft und die Probleme in das SMT2-Format umwandelt. Die umgewandelten Probleme werden in einem neuen Ordner abgelegt. Die ursprüngliche Ordnerstruktur bleibt dabei erhalten. Probleme aus dem Ordner `NUM` werden nach der Umwandlung beispielsweise wieder in einem neuen Ordner `NUM` abgelegt. Darüber hinaus filtert das Skript nur die für dieses Projekt relevanten Probleme heraus. So werden ausschließlich FOF- und HOL-Probleme konvertiert. Der Nutzer kann zudem auswählen, ob nur HOL- oder nur FOF-Probleme oder beides gleichzeitig umgewandelt und in einen Ordner abgelegt werden sollen. Das Skript beachtet dabei immer die Zeile `SPC_mode`, welche in jedem TPTP-Problem zu finden ist. An dieser Stelle steht, welcher Logik das Problem zugehörig ist. So kann man, wenn dort beispielsweise `TH0` (für typed higher order) stehen würde, daraus schließen, dass es sich um ein HOL-Problem handelt.

6.4 Erfassung und Speicherung der Metriken

Zur praktischen Umsetzung des Minimierungsverfahrens wurde ein Python-Skript entwickelt (das gesamte Projekt wurde in Python, Version 3.10.10, umgesetzt), das für jedes SMT2-Problem aus der TPTP-Bibliothek folgende Metriken automatisiert erfasst:

- die Anzahl der Funktionsargumente vor und nach der Minimierung,
- die Anzahl der boolschen Operationen vor und nach dem Minimierungsprozess,
- den Lösungsstatus (z. B. `sat`, `unsat` oder `unknown`) vor und nach der Minimierung, wobei `None` protokolliert wird, wenn der Solver nicht innerhalb des festgelegten Zeitrahmens reagiert oder zu viele Timeouts auftreten,
- und die Anzahl der während des Minimierungsprozesses aufgetretenen Timeouts.

Diese Daten werden in einer CSV-Datei abgelegt, wobei zusätzliche Parameter (z. B. eine maximale Anzahl an Timeouts von 11 sowie das Ausschließen von Problemen mit mehr als 200 Zeilen) zur Filterung der Probleme verwendet werden. Standardmäßig wird `CVC5` eingesetzt, da dieser insbesondere bei HOL-Problemen zuverlässigere Ergebnisse liefert. Alternativ kann auch `z3` verwendet werden, der erfahrungsgemäß jedoch eher für FOF-Probleme geeignet ist.

Ausgehend von den beschriebenen Implementierungsdetails betrachten wir im Folgenden verwandte Arbeiten und diskutieren Gemeinsamkeiten sowie Unterschiede zu unserem Ansatz.

7 Verwandte Arbeiten

Im Kontext der Minimierung von SMT-Problemen gibt es verschiedene Ansätze, die darauf abzielen, die Größe und Komplexität der Eingaben zu reduzieren und damit SMT-Solver zu verbessern. Ein bereits diskutierter Ansatz stellt dabei das Delta-Debugging dar. Hierbei zielt man darauf ab, aus fehlerverursachenden Eingaben ein möglichst kleines, minimal fehlerinduzierendes Beispiel zu extrahieren. Um einen speziellen Anwendungsfall dieser Technik handelt es sich bei dem Tool `ddsmt`. Dieses Tool setzt unterschiedliche Delta-Debugging-Strategien ein (vgl. Abschnitt 3.4), wobei ausschließlich im SMT-Format vorliegende Probleme behandelt werden. Während `ddsmt` in vielen Fällen zwar sehr kompakte Minimalbeispiele erzeugt, kann dies teilweise dazu führen, dass die ursprüngliche Problemstruktur stark verändert wird. Dies bildet einen interessanten Vergleichspunkt zu unserem Ansatz, der insbesondere darauf abzielt, die Grundstruktur des ursprünglichen Problems zu erhalten. Ein vielversprechender Ansatz im Umgang mit quantifizierten SMT-Problemen wird in einer Arbeit von Niemetz et al. beschrieben. Dabei werden vordefinierte Grammatiken genutzt, um gezielt relevante Instanziierungen zu generieren, sodass unnötige und redundante Ableitungen vermieden und der Suchraum des SMT-Solvers verkleinert wird. Die Effizienz der Solver konnte dadurch erheblich gesteigert werden, was einen interessanten Vergleichspunkt zu Verfahren bietet, deren Fokus primär auf der Minimierung der Eingabeformeln liegt Niemetz et al. (2021). In einem Verfahren von Reynolds et al. wird gezeigt, dass durch systematisches Erzeugen und Überprüfen möglicher Instanziierungen quantifizierter Variablen der Suchraum verkleinert werden kann, was dazu führt, dass überflüssige Bestandteile der Eingabe entfernt werden und die Rechenintensität reduziert wird Reynolds et al. (2018). Ein Ansatz von Bjørner und Fazekas beschreibt die Vorverarbeitung von SMT-Formeln, bei der vereinfachte Formeln wiederverwendet werden, wenn neue Bedingungen hinzukommen, sodass der gesamte Vorverarbeitungsprozess nicht wiederholt werden muss. Hierbei werden Teile des Problems durch einfachere Ausdrücke ersetzt und bei Bedarf wieder eingefügt, was die Effizienz der Solver erhöht Bjørner & Fazekas (2023). Schließlich wird in einem Beitrag von Mikek und Zhang ein Verfahren vorgestellt, das SMT-Formeln in ein von Compilern verwendetes Zwischenformat überführt, in dem Optimierungstechniken angewandt werden, bevor das Ergebnis wieder in das ursprüngliche Format zurückgeführt wird. Dadurch können komplexe Formeln deutlich schneller bearbeitet und mehr Formeln in vorgegebener Zeit gelöst werden. Diese Methode ist solverunabhängig und erfordert keine tiefgreifenden Kenntnisse der internen Arbeitsweise der Solver Mikek & Zhang (2023). Die beschriebenen Ansätze fokussieren größtenteils direkt auf die Verbesserung der Solver-Effizienz. Unser Ansatz hingegen zielt primär darauf ab, die Problemstruktur so zu vereinfachen, dass Forschern das Debugging und die Verbesserung von SMT-Solvern erleichtert wird, ohne dabei den korrekten Lösungsstatus der Probleme zu verändern. Dabei erfolgen alle Minimierungsschritte stets unter Berücksichtigung des Erhalts der Grundstruktur des ursprünglichen Problems. Das Resultat ist ein vereinfachtes Problem, das dem Original sehr ähnlich bleibt. Dies unterstreicht die Einzigartigkeit unseres Ansatzes.

8 Zusammenfassung

Die vorliegende Arbeit präsentiert einen neuartigen Ansatz zur Minimierung von Problemen im SMT-LIB v2 Format. Durch den gezielten Einsatz von Methoden zur Modifikation boolscher Operationen, zur Entfernung bestimmter Funktionsargumente sowie durch die Verwendung eines

iterativen Fixpunkt-Algorithmus können SMT-Probleme aus der TPTP-Bibliothek signifikant vereinfacht werden. Darüber hinaus wird die Lesbarkeit der Probleme mithilfe des neuen Ansatzes verbessert, was eine weiterführende Analyse erleichtert. Besonders wichtig ist, dass der zu beweisende Kern, also die zentrale Aussage und die unterstützenden Axiome, vollständig erhalten bleibt, wobei überflüssige Details entfernt werden, die weder zur Kernaussage beitragen, noch den Beweis unterstützen. Dies unterstützt sowohl den Menschen als auch den Solver dabei, den eigentlichen Beweisansatz zu erkennen und Schwächen im Beweisprozess zu identifizieren. Insgesamt liefert der vorgestellte Ansatz wichtige Erkenntnisse zur Balance zwischen der Reduktion der Komplexität und dem Erhalt der ursprünglichen Problemstruktur und legt damit die Grundlage für eine gezielte Weiterentwicklung von SMT-Solvern.

9 Zukünftige Arbeit

Für zukünftige Arbeiten ergeben sich mehrere Ansatzpunkte zur Weiterentwicklung des Minimierungsverfahrens. So könnte durch mehr Rechenleistung und Zeit die maximale Anzahl an Iterationen beliebig erhöht werden, um noch stärker minimierte Probleme zu erhalten. Auch könnte man den Threshold, ab dem statt des optimalen Verfahrens auf den Greedy-Ansatz zurückgegriffen wird, erhöhen. So könnte man auch für etwas größere Probleme eine optimale Lösung garantieren, auch wenn dies aufgrund der hohen Komplexität signifikant mehr Rechenleistung und Zeit erfordern würde. Zudem ließen sich auch die gesetzten Timeouts noch deutlich nach oben setzen, was ebenfalls zu noch stärkeren Minimierungen führen könnte. Ebenso würde es sich anbieten, das Projekt so zu erweitern, dass weitere mathematische Operationen, die beispielsweise bei Problemen anderer Bibliotheken auftreten, ebenfalls modifiziert werden können. Des Weiteren bietet sich die Möglichkeit, noch stärkere Minimierungen vorzunehmen, wie beispielsweise komplette `assert`-Anweisungen zu entfernen. Dabei stellt sich jedoch immer die Frage, wie viel der Struktur des ursprünglichen Problems erhalten bleiben soll. Zudem könnte man den Refactoring-Prozess deutlich erweitern, sodass mehr Präfixe entfernt werden können (beispielsweise auch für Probleme außerhalb der TPTP-Bibliothek). Darüber hinaus würde es sich anbieten, die einzelnen Minimierungs- und Refactoring-Schritte in eine Klassenstruktur zu überführen, wie es in `ddsmt` bei den Mutatoren der Fall ist. So könnte man das Projekt einerseits besser erweitern und warten, und andererseits könnten einzelne Minimierungs- und Refactoring-Schritte gezielter eingesetzt werden. Zusammenfassend bietet der dargestellte Ausblick einen klaren Fahrplan für zukünftige Forschungen, der das Potenzial birgt, nicht nur die SMT-Solver, sondern auch die Flexibilität des Minimierungsverfahrens signifikant zu verbessern.

Literatur

BARBOSA, HANIEL; CLARK BARRETT; MARTIN BRAIN; GEREON KREMER; HANNA LACHNITT; MAKAI MANN; ABDALRHMAN MOHAMED; MUDATHIR MOHAMED; AINA NIEMETZ; ANDRES NÖTZLI; ET AL. 2022. cvc5: A versatile and industrial-strength smt solver. *International conference on tools and algorithms for the construction and analysis of systems*, Springer, 415–442.

BARRETT, CLARK; MORGAN DETERS; LEONARDO DE MOURA; ALBERT OLIVERAS; und AARON STUMP. 2013. 6 years of smt-comp. *Journal of Automated Reasoning* 50.243–277.

BARRETT, CLARK; AARON STUMP; CESARE TINELLI; ET AL. 2010. The smt-lib standard: Version 2.0. *Proceedings of the 8th international workshop on satisfiability modulo theories (edinburgh, uk)*, Ausg. 13, 14.

BARWISE, JON. 1977. An introduction to first-order logic. *Studies in logic and the foundations of mathematics*, Ausg. 90, 5–46. Elsevier.

BJØRNER, NIKOLAJ, und KATALIN FAZEKAS. 2023. On incremental pre-processing for smt. *International conference on automated deduction*, Springer, 41–60.

BOOLOS, GEORGE S. 1975. On second-order logic. *The Journal of Philosophy* 72.509–527. URL `http://www.jstor.org/stable/2025179`.

DE MOURA, LEONARDO, und NIKOLAJ BJØRNER. 2008. Z3: An efficient smt solver. *International conference on tools and algorithms for the construction and analysis of systems*, Springer, 337–340.

ERNST, PROF. DR. GIDON. 2023. Formale spezifikation und verifikation — folien. Vorlesungsfolien, LMU München. Verfügbar auf Anfrage oder interne Quelle.

HETZL, STEFAN. 2019. Higher-order logic. Technical report, Vienna University of Technology. Abgerufen am 28.03.2025. URL `https://www.dmg.tuwien.ac.at/hetzl/teaching/hol.20190125.pdf`.

KONDYLIDOU, LYDIA; ANDREW REYNOLDS; und JASMIN BLANCHETTE. Augmenting Model-Based Instantiation with Fast Enumeration. URL `https://kondylidou.github.io/assets/pdf/conf.pdf`.

KREMER, GEREON; AINA NIEMETZ; und MATHIAS PREINER. 2021. ddsmt 2.0: Better delta debugging for the smt-libv2 language and friends. *International conference on computer aided verification*, Springer, 231–242.

MIKEK, BENJAMIN, und QIRUN ZHANG. 2023. Speeding up smt solving via compiler optimization. *Proceedings of the 31st acm joint european software engineering conference and symposium on the foundations of software engineering*, 1177–1189.

NIEMETZ, ANDREAS; MARTIN PREINER; ANDREW REYNOLDS; CLARK BARRETT; und CESARE TINELLI. 2021. Syntax-guided quantifier instantiation for smt. *Proceedings of tacas*, 145–163.

REYNOLDS, ANDREW; HANIEL BARBOSA; und PASCAL FONTAINE. 2018. Revisiting enumerative instantiation. *Tools and algorithms for the construction and analysis of systems: 24th international conference, tacas 2018, held as part of the european joint conferences on theory and practice of software, etaps 2018, thessaloniki, greece, april 14-20, 2018, proceedings, part ii 24*, Springer, 112–131.

SHIROKIKH, MIKHAIL; ILYA SHENBIN; ANTON ALEKSEEV; und SERGEY NIKOLENKO. 2023. Machine learning for sat: Restricted heuristics and new graph representations. *arXiv preprint arXiv:2307.09141*.

SUTCLIFFE, GEOFF. n.d. The tptp problem library, tptp v9.0.0. Technical report, Department of Computer Science, University of Miami. URL `http://www.tptp.org`.

VAN BENTHEM, JOHAN, und KEES DOETS. 1983. Higher-order logic. *Handbook of philosophical logic*, 189–243. Springer.