LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

LEHR- UND FORSCHUNGSEINHEIT FÜR THEORETISCHE INFORMATIK UND THEOREMBEWEISEN

Proof Visualization

Youlguk Choi

Thesis type (Bachelor's Thesis)
im Studiengang 'Studiengang (Informatik plus Statistik)'

Betreuer: Prof. Jasmin Blanchette

Mentorin: Lydia Kondylidou

Ablieferungstermin: 24. April 2025

Erklärung

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

Munich, 24. April 2025 Author

Danksagungen

Mein besonderer Dank gilt Frau Professorin Kondylidou für ihre kontinuierliche Unterstützung und die zahlreichen hilfreichen Rückmeldungen während der Erstellung dieser Arbeit. Durch ihre zeitnahen Rückmeldungen und fundierten Ratschläge konnte ich sowohl die Programmierumsetzung als auch die schriftliche Ausarbeitung zielgerichtet voranbringen und erfolgreich abschließen.

Kurzfassung

SMT-Solver geben ihre Beweise typischerweise in maschinenlesbaren Formaten wie *Alethe* aus, welche für Menschen jedoch schwer verständlich sind. Ziel dieser Arbeit ist es, ein System zu entwickeln, das den vom SMT-Solver *cvc5* erzeugten Beweisprozess in eine für Menschen verständliche Form überführt. Dazu wird die Alethe-Ausgabe zunächst geparst und anschließend in mathematische Notation transformiert. Daraufhin wird der gesamte Beweisfluss als DOT-Graph visualisiert, um die logischen Zusammenhänge auf einen Blick erfassbar zu machen. Für die Evaluation wurden insgesamt 11 SMT-Testdateien verwendet. In der aktuellen Version konnte das System 10 dieser Dateien erfolgreich transformieren und visualisieren. Die Ergebnisse zeigen, dass selbst komplexe logische Strukturen in eine intuitive und verständliche Sprache überführt werden konnten. Das System hat somit großes Potenzial für den Einsatz in der Fehlersuche und in der Lehre im Bereich SMT.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation 1
1.2 Zielsetzung der Arbeit 1

2 Hintergrund **2**
2.1 Satisfiability Modulo Theories (SMT) 2
2.2 cvc5 und die Alethe-Beweisstruktur 2
2.3 Beweisformate: Alethe, LFSC und DOT 3

3 Methodologie **3**
3.1 Architektur des Systems 3
3.2 Parsing 4
3.3 Umwandlung in mathematische Notation 7
3.4 Visualisierung mit Graphviz 10
3.5 Beispiele zur Methodologie 10
3.5.1 Beispielanalyse: Widerspruchsbeweis (Beispiel 1) 11
3.5.2 Beispielanalyse: Instantiierung und Quantorauflösung (Beispiel 2) 12
3.5.3 Beispielanalyse: Arithmetischer Gleichungswiderspruch (Beispiel 3) 14

4 Evaluation **17**

5 Ausblick **18**

6 Schlussfolgerung **18**

Literatur **22**

1 Einleitung

1.1 Motivation

Die Satisfiability Modulo Theories (SMT)-Solver stellen eine Kombination aus einem booleschen SAT-Solver und Entscheidungsverfahren für interpretierte Theorien dar. Verschiedene SMT-Solver wie *Bitwuzla* (11), *Boolector* (12), *cvc5* (1), *veriT* (6) und *Z3* (7) werden verwendet, um logische Formeln auf ihre Erfüllbarkeit hin zu prüfen(9).

In dieser Arbeit wird der vom SMT-Solver *cvc5* erzeugte Beweis analysiert. *cvc5* ist ein Werkzeug zur Bestimmung, ob eine gegebene Aussage erfüllbar ist oder nicht. Im Falle eines negativen Ergebnisses (*unsat*) gibt der Solver zusätzlich einen Beweis aus, um die Gültigkeit der Ableitung zu verifizieren (3). Dieser Beweis kann in verschiedenen Formaten wie *Alethe*, *LFSC* oder *DOT* ausgegeben werden (2). Jedes dieser Formate ist so strukturiert, dass es sich entweder für die maschinelle Verarbeitung oder für die menschliche Interpretation eignet (15).

Das *Alethe*-Format beispielsweise zielt auf eine menschenfreundliche Darstellung ab, während das *DOT*-Format eine graphische Visualisierung der Beweisstruktur ermöglicht (8). Obwohl das *Alethe*-Format keine Maschinensprache im engeren Sinne ist, handelt es sich um eine hochstrukturierte logische Sprache, die sowohl von Menschen als auch Maschinen interpretiert werden kann. Diese Struktur ermöglicht es theoretisch, die Beweisschritte nachzuvollziehen. In der Praxis ist dies jedoch aufgrund der Komplexität und Länge der Schlussfolgerungsketten nur schwer möglich.

Besonders für Personen ohne Vorkenntnisse im Bereich formaler Beweise oder SMT-Solver ist das Verständnis nahezu unmöglich, was einen hohen Lernaufwand mit sich bringt und entsprechend viel Zeit kostet. Eine Transformation solcher Beweise in eine für Menschen verständliche Form bietet daher einen konkreten Nutzen für Ausbildung, Fehlersuche und Sicherheitsüberprüfungen.

1.2 Zielsetzung der Arbeit

Ziel dieser Arbeit ist es, die durch SMT-Solver – insbesondere durch *cvc5* – erzeugten, maschinenfreundlichen Beweise in eine Form zu überführen, die für Menschen verständlich ist (3). Zu diesem Zweck wurde ein System entwickelt, das zwei Hauptfunktionen umfasst: Einerseits ein Modul, das die Beweisausgabe in eine mathematisch verständliche und lesbare Sprache transformiert, andererseits ein Modul, das den Beweisprozess mithilfe einer grafischen Visualisierung übersichtlich darstellt.

Im Zentrum dieser Arbeit steht das *Alethe*-Format, welches bei *cvc5* eine wichtige Rolle als standardisiertes Beweisformat spielt (2). Auf Grundlage dieses Formats wird ein Parser entwickelt, der die einzelnen Beweisschritte rekonstruiert, die verwendeten Regeln identifiziert und daraus eine Struktur erzeugt, die sowohl mathematisch als auch visuell interpretierbar ist. Für die grafische Darstellung wird das *DOT*-Format verwendet, wobei die Visualisierung mithilfe von Graphviz umgesetzt wird (8). Dadurch können die Beweisschritte als gerichteter Graph strukturiert dargestellt werden.

Durch die Verwendung dieses Tools können Nutzer:innen die Beweisschritte interaktiv durchgehen, bestimmte Ableitungen hervorheben und die Beweisstruktur insgesamt besser erfassen. Das entwickelte Tool richtet sich an Studierende, Forschende und Entwickler:innen, die SMT-Solver insbesondere im Bildungsbereich nutzen möchten. Die Arbeit soll somit dazu beitragen, die Lücke zwischen maschinell erzeugten Beweisen und menschlichem logischen Verständnis zu überbrücken.

Die folgenden Kapitel behandeln zunächst die technischen Grundlagen, bevor im Methodenteil der entwickelte Lösungsansatz im Detail erläutert wird.

2 Hintergrund

2.1 Satisfiability Modulo Theories (SMT)

In der Informatik und der mathematischen Logik bezeichnet *Satisfiability Modulo Theories (SMT)* das Problem, zu entscheiden, ob eine gegebene mathematische Formel erfüllbar ist (15). Es stellt eine Verallgemeinerung des klassischen SAT-Problems dar, da es komplexere Ausdrücke wie reelle Zahlen, Ganzzahlen sowie Datenstrukturen wie Listen, Arrays, Bitvektoren und Zeichenketten einbezieht. SMT-Solver sind spezialisierte Werkzeuge zur Lösung solcher Probleme; bekannte Vertreter sind *Z3*, *cvc5*, *Yices2*, *Bitwuzla* und *veriT*. Je nach Solver unterscheiden sich unterstützte Theorien, Schnittstellen und Einsatzbereiche, wobei insbesondere Z3 und cvc5 breite Anwendung in Theorembeweis, Programmanalyse, formaler Verifikation und Softwaretest finden (3).

Da bereits das SAT-Problem NP-vollständig ist (13), sind SMT-Probleme im Allgemeinen mindestens NP-schwer. Durch die Integration semantisch reicherer Theorien wie Arithmetik, Arrays oder Strings erhöht sich die Komplexität weiter, und einige Probleminstanzen sind sogar unentscheidbar. Die Forschung untersucht daher, welche Theorien zu entscheidbaren Teilproblemen führen und welche Komplexität dabei vorliegt. Ein klassisches Beispiel für eine entscheidbare Theorie ist die *Presburger-Arithmetik*, die sich ausschließlich auf Addition und Ungleichungen über ganze Zahlen stützt. Zudem lassen sich SMT-Probleme als Constraint Satisfaction Problems (CSP) verstehen und somit auch im Kontext von Constraint Programming interpretieren.

Aufgrund dieser theoretischen Tiefe und algorithmischen Komplexität sind die von SMT-Solvern erzeugten Beweise meist maschinennah und schwer verständlich. Ziel dieser Arbeit ist es daher, diese Beweisausgaben in eine für Menschen lesbare mathematische Darstellung zu transformieren und durch Visualisierungen strukturell nachvollziehbar zu machen.

2.2 cvc5 und die Alethe-Beweisstruktur

Der in dieser Arbeit betrachtete Solver *cvc5* ist die neueste Version der CVC-Solverfamilie (3), die auf der `DPLL(T)`-Architektur basiert (13) und zahlreiche Theorien wie lineare Arithmetik über rationale und ganze Zahlen, Bitvektoren, Gleitkommazahlen, Zeichenketten, Datentypen, Sequenzen, endliche Mengen, uninterpreted functions und Separationslogik unterstützt. Er erlaubt Eingaben in `SMT-LIB`, `TPTP` und `SyGuS-IF` und bietet Schnittstellen für `C++`, `Python` und `Java`.

Für diese Arbeit ist insbesondere das durch die Option `--dump-proofs` erzeugte *Alethe*-Beweisformat (2) relevant. Ein solcher Beweis besteht aus strukturierten Anweisungen wie `declare` (Deklaration von Symbolen und Typen), `define` (Definition von Symbolen durch Ausdrücke), `assume` (Einführung von Annahmen) und `step` (Anwendung logischer Regeln). Die `step`-Anweisung ist zentral, da sie unter Angabe einer Regel und ihrer Prämissen eine Schlussfolgerung erlaubt und die einzelnen Schritte zu einem kohärenten Beweis verbindet, der zur Herleitung von `unsat` führt.

2.3 Beweisformate: Alethe, LFSC und DOT

Zur Repräsentation der Beweise unterstützt `cvc5` die Formate **Alethe**, **LFSC** und **DOT**, die jeweils unterschiedliche Zwecke erfüllen. Das textbasierte **Alethe**-Format ist modular aufgebaut (2) und für maschinelle Verarbeitung optimiert, weniger jedoch für menschliche Lesbarkeit. Das **LFSC**-Format hingegen bietet eine stark typisierte, formal verifizierbare Struktur, ist aber aufgrund seiner Komplexität kaum lesbar. Es wird vor allem im Bereich der formalen Beweissysteme wie Twelf verwendet (14). Demgegenüber zielt das **DOT**-Format auf die grafische Darstellung ab: Beweisschritte werden als Knoten, deren Abhängigkeiten als gerichtete Kanten visualisiert, etwa mit `Graphviz` (8). Damit eignet sich DOT besonders zur intuitiven Analyse komplexer Beweisketten.

Ein tabellarischer Vergleich in Tabelle 1 gibt einen Überblick über Zweck, Lesbarkeit und Visualisierbarkeit der einzelnen Formate.

Tabelle 1: Vergleich von cvc5-Beweisformaten

Format	**Zweck**	**Lesbarkeit**	**Visualisierbar**
Alethe	Standardisiertes Logging	Niedrig	Indirekt (nach Umwandlung)
LFSC	Formale Verifikation	Sehr niedrig	Nein
DOT	Visualisierung	Hoch (grafisch)	Ja (z. B. mit Graphviz)

3 Methodologie

3.1 Architektur des Systems

Das vorliegende System besteht aus insgesamt drei Hauptkomponenten. Zunächst wird eine Parsing-Phase durchgeführt, in der die einzelnen Ausdrücke aus der Ausgabestruktur des SMT-Solvers entsprechend ihrer jeweiligen Typen analysiert werden. Anschließend erfolgt eine Umwandlung in mathematische Notation, gefolgt von einer Visualisierung, die den gesamten Beweisfluss in Form eines gerichteten Graphen darstellt.

Beim Ausführen des Systems wird eine `.smt2`-Datei als Eingabe verwendet, wobei der SMT-Solver `cvc5` mithilfe der Option `--dump-proofs` den Beweisprozess im Alethe-Format ausgibt. Für Benutzerinnen und Benutzer ohne Kenntnisse dieses Formats stellt diese Ausgabe jedoch lediglich einen

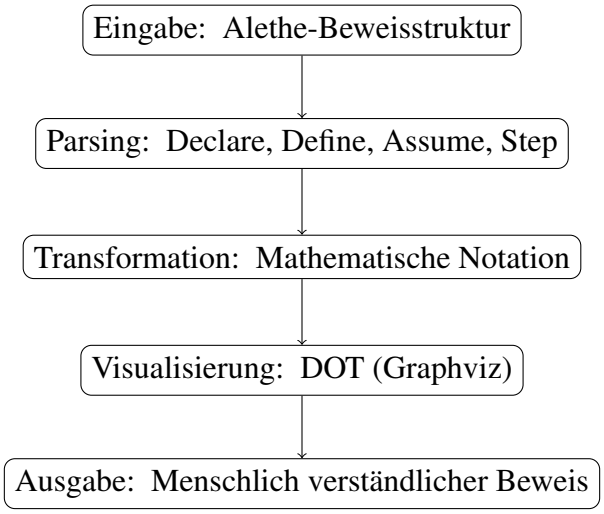

Abbildung 1: Ablaufschema der Beweistransformation

3.2 Parsing

Wie bereits in der Systemarchitektur beschrieben, stellt das Parsing die erste Phase des gesamten Verarbeitungsprozesses dar. In diesem Schritt gibt der SMT-Solver `cvc5` einen Beweis im Alethe-Format aus, welcher anschließend in eine strukturierte interne Darstellung überführt wird.

Das Alethe-Format ist ein textbasiertes Ausgabeformat. Aufgrund seiner komplexen Struktur ist es jedoch für Benutzerinnen und Benutzer schwer zu lesen und direkt zu visualisieren. Daher ist es notwendig, die verschiedenen Ausdrucksarten in jeder Zeile klar zu unterscheiden und zu analysieren.

Das vorliegende System unterteilt die Ausgaben im Alethe-Format in vier Haupttypen: `declare`, `define`, `assume` und `step`. Das Parsing erfolgt auf Basis dieser Typen. Jeder dieser Ausdrücke besitzt eine eigene Struktur und Semantik. Das Parsing legt somit die Grundlage für die spätere Transformation in mathematische Ausdrücke sowie für die grafische Visualisierung.

Deklaration von Symbolen

Der erste Parser, der betrachtet wird, ist der `declare`-Parser. Er dient dazu, Symbole (z. B. Variablen oder Funktionen), die im SMT-Problem verwendet werden, zusammen mit ihrem Typ zu deklarieren. Diese Deklaration bildet eine Voraussetzung dafür, dass im weiteren Beweisprozess klar ist, welche Bedeutung das jeweilige Symbol trägt.

Der `declare`-Parser extrahiert zunächst das Symbol und den zugehörigen Typ. Anschließend wird das Ergebnis in der Form „Symbol = Typ“ in der internen Datenstruktur gespeichert.

Algorithm 1 Parsing eines `declare-const`-Statements

```
1: function PARSEDECLARE(line)
2:     if line starts with "(declare-const" then
3:         symbol ← EXTRACTSYMBOL(line)
4:         type ← EXTRACTTYPE(line)
5:         symbolTable[symbol] ← type
6:     end if
7: end function
```

Ein Ausdruck wie `(declare-const a Int)` deklariert zum Beispiel eine Konstante `a`, wobei `a` eine Ganzzahl (`Int`) repräsentiert. Über ein `declare`-Statement eingeführte Symbole können in späteren Beweisschritten referenziert werden.

Darüber hinaus sind auch Funktionsdeklarationen möglich, wie z. B.:

```
(declare-fun g (Int Bool) Real)
```

Dieser Ausdruck bedeutet, dass die Funktion `g` zwei Eingaben vom Typ `Int` und `Bool` erhält und einen Wert vom Typ `Real` zurückgibt.

Definition von Ausdrücken

Das `define`-Statement ermöglicht es, in SMT-LIB neue Funktionen oder Konstanten zu definieren, die zuvor nicht existiert haben. Solche Definitionen können in späteren Ausdrücken oder Ableitungsschritten mehrfach wiederverwendet werden.

Algorithm 2 Parsing eines `define`-Statements

```
1: function PARSEDEFINE(line)
2:     name ← EXTRACTDEFINENAME(line)
3:     body ← EXTRACTDEFINEBODY(line)
4:     definitions[name] ← body
5: end function
```

Ein Ausdruck wie `(define @t1 () (+ x 1))` ist ein `define`-Statement, das ein neues Symbol `@t1` definiert. Dabei handelt es sich um eine nullstellige (0-äre) Funktion, deren Rumpf der Ausdruck `(+ x 1)` ist. Der `define`-Parser extrahiert aus diesem Ausdruck den Funktionsnamen `@t1` und ordnet ihm die Definition `x + 1` zu.

Solche Definitionen werden als interne Darstellung gespeichert, sodass sie auch in anderen komplexen Ausdrücken verwendet werden können – zum Beispiel in weiteren `define`-, `assume`- oder `step`-Statements.

Verarbeitung von Annahmen

Als Nächstes wird das `assume`-Statement betrachtet, das die Vorbedingungen eines SMT-Beweises verarbeitet. Der `Assume`-Parser extrahiert dabei sowohl den Namen des Statements als auch dessen Inhalt, formatiert den logischen Ausdruck gemäß der internen Repräsentation und speichert ihn ab.

Algorithm 3 Parsing eines `assume`-Statements

```
1: function PARSEASSUME(line)
2:     name ← EXTRACTASSUMENAME(line)
3:     formula ← EXTRACTLOGICALEXPRESSION(line)
4:     simplified ← SIMPLIFY(formula)
5:     assumptions[name] ← simplified
6: end function
```

Ein Beispiel für ein solches Statement ist:

```
(assume @p1 (and @t3 (and (< 0 a) (< a 2)) (not @t1)))
```

Diese Aussage enthält eine komplexe logische Vorbedingung. Zunächst wird angenommen, dass das zuvor definierte `@t3` wahr ist. Gleichzeitig wird vorausgesetzt, dass die Variable `a` größer als 0 und kleiner als 2 ist. Darüber hinaus wird angenommen, dass `@t1` falsch ist, was durch die Negation `(not @t1)` ausgedrückt wird.

Diese drei Bedingungen sind durch den logischen Operator `and` miteinander verknüpft und bilden somit eine zusammengesetzte Vorbedingung. Der Parser verarbeitet diesen Ausdruck und überführt ihn in eine vereinfachte interne Darstellung wie:

$$@t3 \land (0 < a \land a < 2) \land \neg @t1$$

`assume`-Statements können nicht nur einfache Gleichungen, sondern auch zuvor definierte Ausdrücke enthalten und als komplexe logische Formeln dargestellt werden. Sie sind von zentraler Bedeutung, da sie den Ausgangspunkt für den Beweis im SMT-Solver darstellen und als essentielle Vorannahmen dienen.

Verarbeitung von Beweisschritten

Der `step`-Parser ist der letzte und zugleich wichtigste der vier Parser. Er verarbeitet die Ausdrücke, die tatsächliche Schlussfolgerungsschritte im Beweis darstellen. Ein `step`-Statement enthält typischerweise mehrere Bestandteile wie den Namen des Schritts, die verwendete Regel, Prämissen sowie weitere Argumente.

Algorithm 4 Parsing eines `step`-Statements

```
1: function PARSESTEP(line)
2:     stepName ← EXTRACTSTEPNAME(line)
3:     rule ← EXTRACTRULENAME(line)
4:     args ← EXTRACTARGUMENTS(line)
5:     premises ← RESOLVEPREMISES(args)
6:     conclusion ← APPLYRULE(rule, premises)
7:     steps[stepName] ← conclusion
8: end function
```

Ein Beispiel:

```
(step t4 (and_elim @p1) :args (@t3))
```

bedeutet, dass auf die Prämisse `@p1` die Regel `and_elim` angewendet wurde, um die Aussage `@t3` herzuleiten.

Der Parser extrahiert den Namen des Schritts, die verwendete Regel, die übergebenen Argumente sowie die zugrunde liegenden Prämissen. Anschließend wird das Resultat der Regelanwendung berechnet und intern gespeichert. Charakteristisch für `step`-Ausdrücke ist, dass sich die benötigten Prämissen und Argumente je nach Regel unterscheiden können. Entsprechend ist eine präzise Regelverarbeitung notwendig, da jede Regel einem eigenen Ableitungsverfahren folgt. Dadurch kann die Bedeutung eines Schritts auch dann verständlich gemacht werden, wenn der Nutzer oder die Nutzerin die konkrete Regel im Detail nicht kennt – die Beweisschritte werden auf diese Weise in eine interpretierbare Form gebracht. Im Folgenden werden einige typische Beispiele von `step`-Anweisungen vorgestellt, um deren Funktionsweise zu erläutern.

Ein Beispiel ist:

```
(step @p2 :rule and_elim :premises ((p ∧ ¬p)) :args (1))
```

Dieser Befehl verwendet die Regel `and_elim`, wobei aus einer Konjunktion $(p \wedge \neg p)$ das zweite Element extrahiert wird. Das Argument `:args (1)` gibt dabei an, dass der zweite Teil der Konjunktion gewählt wird. Das Ergebnis dieser Regelanwendung ist somit:

$$@p2 := \neg p$$

Ein weiteres Beispiel ist: `(step @p4 false :rule contra :premises (@p3 @p2))`. Hier wird ein Widerspruch hergeleitet. Die Anweisung vergleicht die Schritte `@p3` und `@p2` und verwendet die Regel `contra`, um aus diesen ein logisches Paradoxon abzuleiten. Das resultierende Ergebnis ist:

$$(p \wedge \neg p) \Rightarrow \bot$$

Wie diese Beispiele zeigen, basiert eine `step`-Anweisung stets auf drei Komponenten: `rule`, `premises` und `args`. Bei Regeln wie `and_elim` ist es erforderlich, dass die `premises` eine bestimmte logische Struktur (z. B. eine Konjunktion) aufweisen. Nur wenn diese Struktur gegeben ist, kann die Regel korrekt interpretiert und angewendet werden.

Die konkrete Funktionsweise der einzelnen Parser sowie detaillierte Anwendungsbeispiele werden im Abschnitt *Beispiele* näher erläutert.

3.3 Umwandlung in mathematische Notation

Nachdem die Alethe-Ausgabe in eine strukturierte interne Darstellung geparst wurde, erfolgt im nächsten Schritt die Umwandlung dieser Darstellung in mathematische Notation. Diese Transformation geschieht nahezu parallel zum Parsing-Prozess. Ziel ist es, technische Ausdrücke, die für Maschinen lesbar sind, in standardisierte mathematische Schreibweisen zu überführen. Dies verbessert nicht nur die Lesbarkeit der einzelnen Beweisschritte, sondern unterstützt auch die spätere Analyse und Visualisierung.

Viele Ausdrücke in der Alethe-Syntax basieren auf Präfix-Notation. Das System ersetzt diese durch klassische mathematische Symbole. Im Folgenden werden häufige Transformationen exemplarisch dargestellt:

Logische Operatoren

- `(and A B)` $\rightarrow A \wedge B$
- `(or A B)` $\rightarrow A \vee B$
- `(¬ A)` $\rightarrow \neg A$

Vergleichsoperatoren

- `(= a b)` $\rightarrow a = b$
- `(< a b)` $\rightarrow a < b$
- `(> a b)` $\rightarrow a > b$
- `(<= a b)` $\rightarrow a \leq b$
- `(>= a b)` $\rightarrow a \geq b$

Mengenoperationen

- `(element_of_set x S)` $\rightarrow x \in S$
- `(subset_set A B)` $\rightarrow A \subseteq B$

Arithmetische Ausdrücke

- `(+ a b c)` $\rightarrow a + b + c$
- `(- a b)` $\rightarrow a - b$
- `(* a b)` $\rightarrow a \cdot b$

Array-Zugriffe und Speicherungen

- `(select A i)` $\rightarrow A(i)$
- `(store A i v)` $\rightarrow A(i) = v$

Listen- und Mengenstruktur

- `(@list x y z)` $\rightarrow \{x, y, z\}$

Wie zuvor beschrieben, erfolgt die Transformation von Präfix-Ausdrücken in mathematische Schreibweise in allen Parsern – also `declare`, `define`, `assume` und `step` – auf ähnliche Weise. Darüber hinaus führt jedoch insbesondere der `step`-Parser eigene, regelbasierte Umwandlungen durch, die sich an der Bedeutung der verwendeten Beweisregel orientieren.

Ein `step`-Ausdruck hat die Struktur, dass auf der Grundlage bestimmter Prämissen eine Schlussregel angewendet wird, um eine neue Aussage herzuleiten. Dabei hängt die mathematische Interpretation des Ausdrucks maßgeblich von der verwendeten Regel ab.

Ein einfaches Beispiel ist der folgende Ausdruck:

```
(step t4 (and_elim @p1) :args (@t3))
```

Dieser Ausdruck bedeutet, dass auf die Prämisse `@p1` die Regel `and_elim` angewendet wurde, um daraus `@t3` abzuleiten. Mathematisch lässt sich dies so interpretieren, dass `@p1` eine Konjunktion $A \wedge B$ ist und `@t3` einem der beiden Konjunktionsglieder (A oder B) entspricht.

Ein weiteres Beispiel ist die Regel `contra`, welche aus einem Widerspruch zweier Aussagen eine Kontradiktion ableitet. Dies entspricht folgender logischer Schlussweise:

$$A, \neg A \Rightarrow \bot$$

Der `step`-Parser kombiniert somit den Namen der Regel mit den gegebenen Prämissen und erzeugt daraus eine mathematische Repräsentation, die der semantischen Bedeutung der jeweiligen Regel entspricht. Die zentrale Funktion dieses Parsers besteht daher nicht nur in der syntaktischen Verarbeitung, sondern vor allem in der semantischen Interpretation der Beweisschritte auf mathemischer Ebene.

Im Folgenden werden exemplarisch einige der wichtigsten Regeln sowie deren Umwandlung in mathematische Notation vorgestellt.

Regelbasierte Transformationen im `step`-Parser Die folgenden Regeln zeigen, wie typische `step`-Ausdrücke auf mathematische Weise interpretiert werden:

- `(and_elim A)` $\rightarrow$ Aus $A \wedge B$ folgt A oder B
- `(or_intro A)` $\rightarrow$ Aus A folgt $A \vee B$
- `(contra A ¬A)` $\rightarrow A, \neg A \Rightarrow \bot$
- `(unit_res A ¬A)` $\rightarrow A, \neg A \Rightarrow \bot$
- `(refl a)` $\rightarrow a = a$
- `(trans a = b, b = c)` $\rightarrow a = b, b = c \Rightarrow a = c$
- `(not_not A)` $\rightarrow \neg\neg A \Rightarrow A$
- `(implies_elim A` $\Rightarrow B, A) \rightarrow$ $A \Rightarrow B, A \Rightarrow B$

Die hier dargestellten Transformationsregeln stellen nur eine Auswahl der im System implementierten Umwandlungen dar. Weitere Regeln, insbesondere für komplexere logische oder datenspezifische Ausdrücke, wurden ebenfalls berücksichtigt, können jedoch aus Platzgründen in dieser Arbeit nicht im Detail dargestellt werden.
Durch die konsequente Anwendung dieser Regeln wird die technische Beweisdarstellung in eine mathematisch lesbare Form überführt. Dies bildet die Grundlage für die spätere graphische Visualisierung und weiterführende Analyse.

3.4 Visualisierung mit Graphviz

Die Umwandlung einzelner Alethe-Ausdrücke in mathematische Notation ermöglicht zwar ein Verständnis der zugrunde liegenden logischen Struktur jedes Ausdrucks, reicht jedoch nicht aus, um den gesamten Beweisfluss oder die Abhängigkeiten zwischen den einzelnen Schritten nachvollziehen zu können. Für ein vollständiges Verständnis ist eine explizite und intuitive Darstellung der Vorher-Nachher-Beziehungen zwischen den Beweisschritten erforderlich. Aus diesem Grund wurde in diesem System eine graphbasierte Darstellung im DOT-Format gewählt, um die logischen Verknüpfungen zwischen den Beweisschritten visuell darzustellen. Graphviz, ein etabliertes Open-Source-Werkzeug zur Erstellung von Graphen, ermöglicht dabei die Generierung solcher Darstellungen auf Basis einfacher textbasierter Beschreibungen. Diese Entscheidung wurde nicht nur aufgrund der technischen Möglichkeiten, sondern auch aufgrund der Benutzerfreundlichkeit getroffen: Insbesondere bei längeren Beweisstrukturen ist die graphische Darstellung einer rein textuellen weit überlegen, da sie eine schnelle und intuitive Erfassung des Beweisflusses erlaubt.

Im Kontext von SMT-Beweisen kommen hauptsächlich `assume`- und `step`-Konstrukte zum Einsatz. `Assume`-Ausdrücke fungieren häufig als Prämissen für `step`-Schritte und werden zur besseren visuellen Unterscheidung in Violett dargestellt. Der abschließende Knoten eines Beweises, der typischerweise die Schlussfolgerung oder einen Widerspruch darstellt, wird in Rot visualisiert. Der resultierende Graph besteht aus Knoten, die einzelnen Beweisschritten entsprechen, und gerichteten Kanten, die logische Abhängigkeiten zwischen den Schritten ausdrücken. Prämissen erscheinen dabei als eingehende Kanten zu einem `step`-Knoten, während ausgehende Kanten die Weiterführung der Beweisstruktur anzeigen. Die gerichtete Anordnung der Kanten unterstützt dabei eine klare Lesbarkeit der Argumentationskette, typischerweise von oben nach unten oder von links nach rechts.

Trotz der Vorteile der graphischen Darstellung bestehen gewisse Einschränkungen. Da in der erzeugten Visualisierung aus Platzgründen lediglich die Namen der verwendeten Symbole angezeigt werden, ist für ein vollständiges Verständnis weiterhin ein Abgleich mit der textbasierten Form notwendig. Darüber hinaus steigt bei komplexeren Beweisen die Anzahl der Knoten rasch an, was die Übersichtlichkeit beeinträchtigen kann. Um diesen Herausforderungen zu begegnen, wären zukünftige Erweiterungen denkbar, beispielsweise die Integration einer Funktion zur Anzeige von Knoteninhalten per Klick oder die Möglichkeit, Teilgraphen bei Bedarf ein- und auszublenden (Folding). Wie diese Visualisierungsstrategie konkret im Beweisprozess zur Anwendung kommt, wird im folgenden Abschnitt anhand eines Beispiels demonstriert.

3.5 Beispiele zur Methodologie

Ziel dieses Kapitels ist es, die in der Methodologie vorgestellte Transformationspipeline anhand konkreter SMT-Beweise zu evaluieren. Dabei wird ***geprüft***, inwieweit das entwickelte System in der Lage ist, maschinell erzeugte Beweise im Alethe-Format **zuverlässig** in mathematisch verständliche Strukturen zu *transformieren* und deren logische Abhängigkeiten graphisch darzustellen.

Hierzu wurden mehrere repräsentative Beispiele aus dem Bereich der SMT-Verifikation ausgewählt, darunter sowohl einfache Widerspruchsbeweise als auch komplexere Instantiierungs- und Gleichungsbeweise. Diese Beispiele entstammen realen Ausgaben des SMT-Solvers `cvc5`, der mithilfe der Option `--dump-proofs` vollständige Beweisstrukturen im Alethe-Format erzeugt.

Die Analyse jedes Beispiels umfasst dabei mehrere Dimensionen: Zunächst wird der seman-

tische Gehalt der verwendeten `assume`-, `define`- und `step`-Konstruktionen rekonstruiert und in mathematischer Notation erläutert. Darauf aufbauend wird die vom System erzeugte graphische Repräsentation betrachtet, um zu bewerten, ob die zugrunde liegende Beweisstruktur adäquat und nachvollziehbar wiedergegeben wurde.

Besonderes Augenmerk gilt dabei der Nachvollziehbarkeit komplexer Beweisschritte, der Klarheit der erzeugten graphischen Strukturen sowie der Frage, ob die Transformation konsistent mit den logischen Intentionen des Originalbeweises ist. Die im Folgenden vorgestellten Experimente sollen aufzeigen, welche Stärken und Grenzen das System in der praktischen Anwendung besitzt.

3.5.1 Beispielanalyse: Widerspruchsbeweis (Beispiel 1)

*SMT-Dump-Ausgabe

```
(declare-const p Bool)
(assume @p1 (and p (not p)))
(step @p2 :rule and_elim :premises (@p1) :args (1))
(step @p3 :rule and_elim :premises (@p1) :args (0))
(step @p4 false :rule contra :premises (@p3 @p2))
```

Dieses Beispiel dient dazu, die Funktionsweise des Systems an einem minimalen, aber logisch vollständigen Widerspruchsbeweis zu demonstrieren.

Zunächst wird mit `assume` eine widersprüchliche Aussage angenommen:

$$@p1 := (p \wedge \neg p)$$

Die Konjunktion $p \wedge \neg p$ ist klassisch widersprüchlich, da sie gleichzeitig die Wahrheit von p und $\neg p$ behauptet, was gemäß der Aussagenlogik nicht möglich ist. Im nächsten Schritt wird diese Konjunktion mit der Regel `and_elim` in ihre beiden Teilterme zerlegt:

$$@p2 := \neg p \quad \text{und} \quad @p3 := p$$

Diese Schritte basieren auf der folgenden Regel der natürlichen Deduktion:

$$\frac{A \wedge B}{A}\ \wedge\text{-Elim}_1 \qquad \frac{A \wedge B}{B}\ \wedge\text{-Elim}_2$$

Es liegen damit zwei Aussagen vor, die einander widersprechen. Gemäß dem Prinzip der Explosion (*ex contradictione sequitur quodlibet*) folgt aus einem Widerspruch jede Aussage – in unserem Fall wird jedoch gezielt die Falschaussage $\bot$ hergeleitet, was durch die Regel `contra` formalisiert wird:

$$\frac{A \quad \neg A}{\bot}\ \text{Kontradiktion}$$

$$@p4 := \bot$$

Die resultierende mathematische Struktur des Beweises lässt sich folgendermaßen zusammen-

fassen:

$$@p1 := (p \wedge \neg p)$$
$$@p2 := \neg p$$
$$@p3 := p$$
$$@p4 := (p \wedge \neg p) \Rightarrow \bot$$

Die folgende Abbildung zeigt die durch das System erzeugte graphische Darstellung des Beweisflusses. Jeder Knoten stellt eine Aussage dar, und die gerichteten Kanten visualisieren die logische Abhängigkeit zwischen den Beweisschritten.

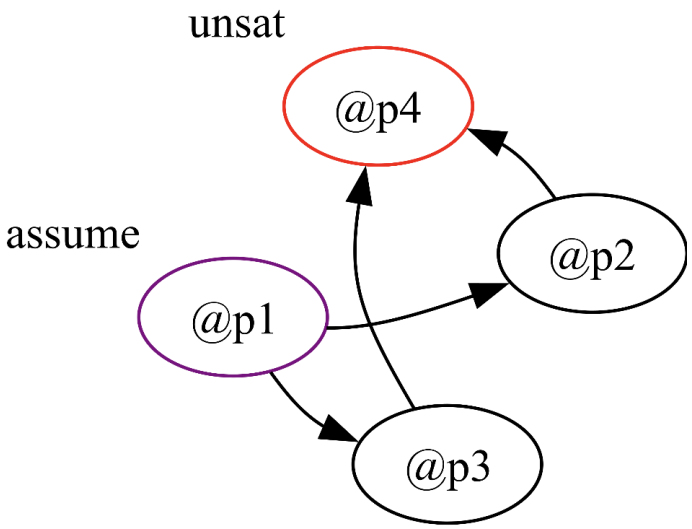

Abbildung 2: Graphische Repräsentation des Widerspruchsbeweises aus Beispiel 1

Diese Visualisierung verdeutlicht auf anschauliche Weise, wie durch Anwendung einfacher logischer Regeln aus einer inkonsistenten Annahme systematisch ein Widerspruch abgeleitet wird. Aus der Herleitung von $\bot$ folgt unmittelbar die Unerfüllbarkeit der ursprünglichen Annahmen, was in der SMT-Terminologie mit `unsat` bezeichnet wird. Gerade in didaktischen Kontexten bietet dieses Beispiel eine besonders klare Demonstration der logischen Struktur und der Stärken der Transformationspipeline.

3.5.2 Beispielanalyse: Instantiierung und Quantorauflösung (Beispiel 2)

*SMT-Dump-Ausgabe

```
(declare-const P (-> Int Bool))
(declare-const a Int)
(define @t1 () (P a))
(define @t2 () (eo::var "x" Int))
(define @t3 () (forall (@list @t2) (P @t2)))
(assume @p1 (and @t3 (and (< 0 a) (< a 2)) (not @t1)))
```

```
(step @p2 :rule and_elim :premises (@p1) :args (0))
(step @p3 :rule and_elim :premises (@p1) :args (2))
(assume-push @p11 @t3)
(step @p5 :rule instantiate :premises (@p2) :args ((@list a)))
(step-pop @p11 :rule scope :premises (@p5))
(step @p6 :rule process_scope :premises (@p11) :args (@t1))
(step @p8 :rule implies_elim :premises (@p6))
(step @p9 :rule reordering :premises (@p8) :args ((or @t1 (not @t3))))
(step @p10 false :rule chain_resolution :premises (@p9 @p3 @p2) :args ((@list true false
```

Dieses Beispiel veranschaulicht die Herleitung eines Widerspruchs durch die Kombination von Allquantoren, Instantiierung und mehrstufiger Resolventenanwendung. Ausgangspunkt ist eine Konjunktion, die sowohl eine allgemeingültige Aussage ($\forall x.\ P(x)$) als auch Bereichsbedingungen für eine Variable a und die Negation einer Instanz $\neg P(a)$ enthält:

$$@p1 := (\forall x.\ P(x)) \wedge (0 < a \wedge a < 2) \wedge \neg P(a)$$

Im ersten Schritt wird mit `and_elim` der allquantifizierte Teil $@p2 := \forall x.\ P(x)$ und die Negation der Instanz $@p3 := \neg P(a)$ extrahiert. Anschließend wird durch die Regel `instantiate` aus $@p2$ die konkrete Aussage $P(a)$ erzeugt:

$$@p5 := P(a)$$

Die Instantiierung erfolgt innerhalb eines lokalen Kontexts, der mit `assume-push` eingeführt und `process_scope` wieder verlassen wird. Letztere Regel ergibt in diesem Fall eine triviale Implikation:

$$@p6 := P(a) \Rightarrow P(a)$$

Durch Anwendung von `implies_elim` entsteht eine Disjunktion:

$$@p8 := \neg P(a) \vee P(a)$$

Diese wird durch `reordering` in die Form $@p9 := P(a) \vee \neg\forall x.\ P(x)$ gebracht, was die Voraussetzung für den abschließenden `chain_resolution`-Schritt ist. Im letzten Schritt werden die Literale $P(a)$ und $\neg P(a)$ sowie $\forall x.\ P(x)$ und $\neg\forall x.\ P(x)$ gegenseitig aufgelöst. Dies führt zum Widerspruch:

$$@p10 := \bot$$

Die Beweisstruktur im Überblick:

$$
\begin{aligned}
@p1 &:= (\forall x.\, P(x)) \wedge (0 < a < 2) \wedge \neg P(a) \\
@p2 &:= \forall x.\, P(x) \\
@p3 &:= \neg P(a) \\
@p5 &:= P(a) \\
@p6 &:= P(a) \Rightarrow P(a) \\
@p8 &:= \neg P(a) \vee P(a) \\
@p9 &:= P(a) \vee \neg \forall x.\, P(x) \\
@p10 &:= \bot
\end{aligned}
$$

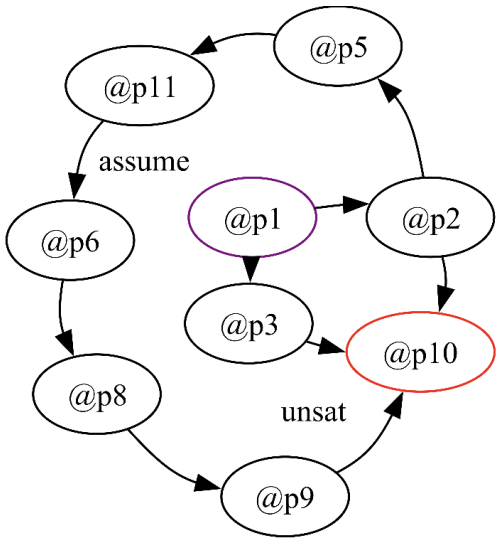

Abbildung 3: Graphstruktur der Transformation in Beispiel 2

Diese Visualisierung zeigt anschaulich, wie durch gezielte Instantiierung und logische Ableitung ein Widerspruch konstruiert wird. Besonders hervorzuheben ist die Rolle der `assume`- und `scope`-Konstrukte, die formale Kontexte kapseln und kontrollierte Implikationen ermöglichen. Zusammen mit `chain_resolution` ergibt sich ein vollständiger Beweis für die logische Unerfüllbarkeit der Ausgangsannahme.

3.5.3 Beispielanalyse: Arithmetischer Gleichungswiderspruch (Beispiel 3)

*SMT-Dump-Ausgabe

```
(declare-const x Int)
(declare-const y Int)
(define @t1 () (- y))
(define @t2 () (- x y))
(define @t3 () (* -1 y))
(define @t4 () (+ 1 x @t3))
```

```
(define @t5 () (+ x @t3))
(assume @p1 (= @t2 (+ x @t1 1)))
(step @p2 :rule trust :premises () :args ((= (= @t5 @t4) false)))
(step @p3 :rule trust :premises () :args ((= (+ x @t3 1) @t4)))
(step @p4 :rule refl :args (1))
(step @p5 :rule trust :premises () :args ((= @t1 @t3)))
(step @p6 :rule refl :args (x))
(step @p7 :rule nary_cong :premises (@p6 @p5 @p4) :args (+))
(step @p8 :rule trans :premises (@p7 @p3))
(step @p9 :rule trust :premises () :args ((= @t2 @t5)))
(step @p10 :rule cong :premises (@p9 @p8) :args (=))
(step @p11 :rule trans :premises (@p10 @p2))
(step @p12 false :rule eq_resolve :premises (@p1 @p11))
```

Dieses Beispiel demonstriert, wie arithmetische Gleichungen durch syntaktische Transformationen und Gleichungsregeln schrittweise zu einem logischen Widerspruch geführt werden können. Ausgangspunkt ist eine Annahme, die zwei arithmetisch äquivalente Ausdrücke gleichsetzt:

$$@p1 := (x - y) = (x + (-y) + 1)$$

In den darauf folgenden Schritten werden mehrere Terme definiert und durch vertrauensbasierte Regeln (`trust`) gleichgesetzt. Wichtig dabei ist die semantische Gleichheit:

$$\begin{aligned} @t1 &:= -y \\ @t2 &:= x - y \\ @t3 &:= -1 \cdot y \\ @t4 &:= 1 + x + (-1 \cdot y) \\ @t5 &:= x + (-1 \cdot y) \end{aligned}$$

Die Gleichheit $@t1 = @t3$ wird als vertrauenswürdige Tatsache angenommen (`trust`). Gleichzeitig werden triviale Gleichungen wie $x = x$ und $1 = 1$ durch `refl` eingeführt. Die Regel `nary_cong` kombiniert diese zu einer größeren Kongruenz:

$$@p7 := x + (-y) + 1 = x + (-1 \cdot y) + 1$$

`trans` verbindet die Gleichungskette weiter zur Form:

$$@p8 := x + (-y) + 1 = 1 + x + (-1 \cdot y)$$

und durch Kombination mit $x - y = x + (-y)$ ergibt sich:

$$@p10 := (x - y) = (1 + x + (-1 \cdot y))$$

Im Gegensatz dazu wurde früher im Beweis mit `trust` eine Ungleichheit eingeführt:

$$@p2 := (x + (-1 \cdot y)) \neq (1 + x + (-1 \cdot y))$$

Der abschließende Schritt `eq_resolve` erkennt den Konflikt zwischen $@p1$ und $@p11$, was zur Ableitung eines Widerspruchs führt:

$$@p12 := \bot$$

$$\begin{aligned}
@p1 &:= (x-y) = (x+(-y)+1)\\
@p2 &:= (x+(-1\cdot y)) \neq (1+x+(-1\cdot y))\\
@p3 &:= (x+(-1\cdot y)+1) = (1+x+(-1\cdot y))\\
@p5 &:= (-y) = (-1\cdot y)\\
@p6 &:= x = x\\
@p7 &:= x+(-y)+1 = x+(-1\cdot y)+1\\
@p8 &:= x+(-y)+1 = 1+x+(-1\cdot y)\\
@p9 &:= (x-y) = (x+(-1\cdot y))\\
@p10 &:= (x-y) = (1+x+(-1\cdot y))\\
@p11 &:= (x-y) \neq (1+x+(-1\cdot y))\\
@p12 &:= \bot
\end{aligned}$$

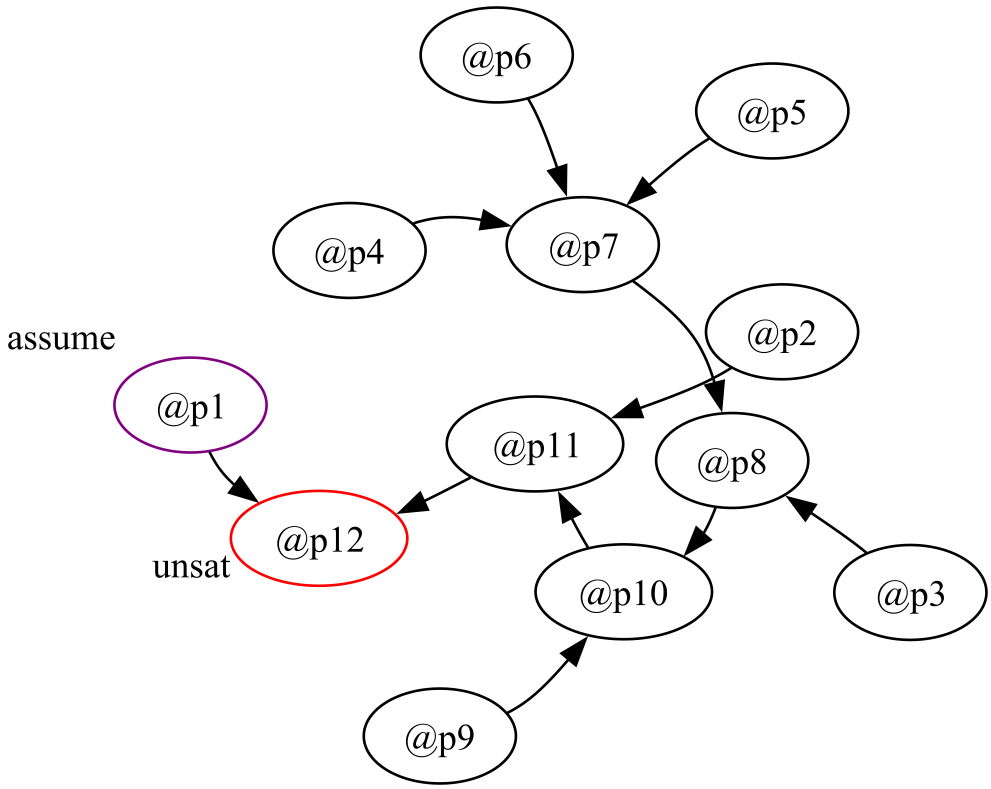

Abbildung 4: Graphstruktur der Gleichungsableitung in Beispiel 3

Die Visualisierung zeigt deutlich, wie scheinbar äquivalente arithmetische Ausdrücke durch syntaktische Manipulation in Konflikt geraten können. Besonders die Rolle von `trust`-Schritten und `eq_resolve` wird hier sichtbar: Obwohl die Transformationen mathematisch korrekt erscheinen, führt ihre Kombination zur Ableitung von $\bot$, also zur Unerfüllbarkeit. Dies illustriert die Fähigkeit des Systems, auch in arithmetischen Kontexten Widersprüche aufzudecken.

4 Evaluation

Das System wurde anhand von insgesamt elf SMT-Eingabedateien evaluiert. Die in den Experimenten verwendeten SMT-Beispiele umfassen verschiedene Logikfragmente wie `QF_UF`, `QF_LIA`, `UFLIA` und `AUFLIA`. Einige davon basieren auf komplexer Mengenlogik aus der TPTP-Bibliothek. Diese Dateien wurden mithilfe des SMT-Solvers `cvc5` in das Alethe-Format konvertiert und danach mit dem entwickelten System geparst, in mathematische Ausdrücke überführt und abschließend grafisch dargestellt. Die ausgewählten Testdateien umfassten logische Widersprüche, arithmetische Gleichungen, Quantoreninstanziierung, Mengentheorie sowie die Extraktion von Unsat-Cores. Für besonders komplexe Probleme mit Mengentheorie (z. B. `TOP001-1`) konnte in der aktuellen Version jedoch keine vollständige Transformation durchgeführt werden.

Die getesteten Beispiele deckten ein breites Spektrum ab, angefangen bei einfachen logischen Ausdrücken bis hin zu komplexeren SMT-Beweisen. Von den elf durchgeführten Tests konnten zehn erfolgreich geparst werden; eine Datei führte zu einem Fehler bei der Verarbeitung. Die folgende Tabelle fasst die Testkategorien und deren Ergebnis zusammen.

Tabelle 2: Getestete SMT-Dateien mit Ergebnis

Dateiname	Kategorie	Ergebnis
test1.smt2	Boolescher Widerspruch	✓
test2.smt2	Arithmetische Gleichung	✓
test3.smt2	Quantoreninstanziierung	✓
test4.smt2	Unvereinbare Bereichsbedingungen	✓
test6.smt2	Unsat-Core	✓
test7.smt2	Quantoreninstanziierung	✓
mbqifo.smt2	Quantoren & Bereich	✓
ARI001-1.smt2	Arithmetik (TPTP)	✓
gen03.smt2	Quantoreninstanziierung	✓
parser.smt2	Arithmetische Bedingungen	✓
top001-1.smt2	Mengenlogik (komplex)	✗

Die derzeitige Version des Systems ist ein Prototyp. Die Aufteilung der Eingabe in passende Parser, die Transformation in mathematische Notation und die grafische Darstellung funktionieren grundsätzlich wie gewünscht. Es wurde jedoch festgestellt, dass es bei bestimmten Regeln des `step`-Konstrukts zu Fehlern kommen kann, insbesondere wenn für eine Regel nicht die benötigten Prämissen vorhanden sind. Diese Schwäche ist unter anderem auf eine bisher begrenzte Anzahl an Testdateien zurückführbar und soll in Zukunft durch weitere Tests und Code-Erweiterungen behoben werden.

Bei zehn der elf Dateien konnte der Beweisprozess korrekt dargestellt werden. Dabei gelang es, komplexe Ausdrücke in eine vereinfachte mathematische Form zu überführen und die Beweisstruktur grafisch nachvollziehbar zu machen. Erfolgreich geparste Fälle zeichneten sich dadurch aus, dass alle Konstrukte (`declare`, `define`, `assume`, `step`) korrekt erkannt und in passende mathematische Ausdrucksformen übersetzt wurden. Die Visualisierung des Beweisflusses erfolgte dabei über `Graphviz`. Im Fehlerfall (`top001-1.smt2`) zeigte sich, dass komplexe Operatoren wie `element_of_set`, `equal_sets`, `subset_sets` oder `in_1st_set` aktuell noch nicht unterstützt werden. Dies verdeutlicht, dass das System bislang nur einen Teil der Alethe-Spezifikation abdeckt. In Hinblick auf die Visualisierung wurde festgestellt, dass diese bei kurzen Beweisen besonders effektiv ist. Bei längeren und verzweigteren Beweisen kann es jedoch zu visueller Komplexität und eingeschränkter Lesbarkeit kommen.

Zusammenfassend konnte festgestellt werden, dass das System bei einfachen SMT-Problemen wie boolescher Logik oder elementarer Arithmetik sehr zuverlässig arbeitet. Auch für mittlere Komplexität, etwa bei Quantoreninstanziierung, zeigte es sich robust. Für komplexe Theorien wie Mengenlehre oder Bitvektoren ist jedoch eine Erweiterung erforderlich. Diese Erkenntnis unterstreicht die Notwendigkeit einer breiteren Regelabdeckung in künftigen Systemversionen.

5 Ausblick

Das vorliegende System ist eine frühe Prototyp-Version und weist mehrere Verbesserungspotenziale sowie Erweiterungsmöglichkeiten auf. Wie bereits in den vorherigen Abschnitten erläutert wurde, verfügt das System derzeit noch über ein unvollständiges Beweissystem. Solange der Beweis kurz ist oder keine seltenen Beweisregeln verwendet werden, funktioniert das Parsing in der Regel ohne Probleme. Wenn jedoch bislang nicht implementierte oder selten genutzte Regeln auftreten, führt dies zu Fehlern und das Parsing schlägt fehl. Ein weiterer Nachteil besteht darin, dass die graphische Darstellung mit zunehmender Anzahl an Symbolen schwerer intuitiv zu erfassen ist. Zusätzlich müssen Benutzer:innen zwischen der graphischen Darstellung und dem Text wechseln, um den Beweis vollständig nachvollziehen zu können.

Deshalb ist es notwendig, den Fehleranteil durch kontinuierliche Erweiterung des Regelsatzes und durch die schrittweise Analyse neuer Beispiele zu reduzieren. Darüber hinaus ist geplant, eine Folding-Funktion in die Graphdarstellung zu integrieren, um den Beweis kompakter darzustellen und es den Nutzer:innen zu ermöglichen, nur die für sie relevanten Prämissen gezielt zu verfolgen. Zusätzlich soll eine interaktive Tooltip-Funktion eingebaut werden, durch die beim Klicken auf ein Symbol dessen mathematische Bedeutung eingeblendet wird.

6 Schlussfolgerung

In dieser Arbeit wurde ein System vorgestellt, das Beweise im Alethe-Format, wie sie vom SMT-Solver `cvc5` erzeugt werden, in eine für Menschen verständliche mathematische Notation überführt und den Beweisprozess mittels Graphen visualisiert. Dadurch wird es auch Personen ohne tiefere Kenntnisse der SMT-Syntax ermöglicht, solche Beweise nachzuvollziehen.

Zunächst ermöglicht das Parsen dem Benutzer, die einzelnen Beweisschritte in einer kompakten mathematischen Form sofort zu erfassen. Gleichzeitig bietet die graphische Darstellung einen klaren

Überblick über den gesamten Beweisablauf. Anhand der Beispiele im Methodikteil konnte gezeigt werden, dass das System benutzerfreundlich arbeitet. Die mathematischen Ausdrücke, die im Zuge der Transformation entstehen, beschränken sich auf Konzepte, wie sie in der gymnasialen Schulbildung vermittelt werden. Daher ist es für Personen mit einer abgeschlossenen allgemeinen Schulbildung möglich, die einzelnen Schritte des Beweises ohne größere Schwierigkeiten zu verstehen.

Das System kann somit insbesondere im Bereich der Lehre und für einfache Debugging-Zwecke hilfreich sein. Zwar ist das System derzeit noch nicht vollständig ausgereift und stößt bei komplexeren logischen Ausdrücken an seine Grenzen, doch zeigt gerade diese Tatsache, dass es großes Erweiterungspotential besitzt. Mit der zukünftigen Unterstützung weiterer Regeln und der Integration interaktiver Funktionen

Abbildungsverzeichnis

1 Ablaufschema der Beweistransformation 4
2 Graphische Repräsentation des Widerspruchsbeweises aus Beispiel 1 12
3 Graphstruktur der Transformation in Beispiel 2 14
4 Graphstruktur der Gleichungsableitung in Beispiel 3 16

Tabellenverzeichnis

1 Vergleich von cvc5-Beweisformaten 3
2 Getestete SMT-Dateien mit Ergebnis 17

Literatur

[1] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Henning Lachnitt, Mathias Mann, Ayman Mohamed, Mudathir Mohamed, Andreas Niemetz, Andres Nötzli, Aina Niemetz, Mathias Preiner, Andrew Reynolds, Yoni Zohar, Cesare Tinelli, and Yi Sheng. cvc5: A versatile and industrial-strength smt solver. In Dana Fishman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2022)*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[2] Haniel Barbosa, Marie Kerjean, Pascal Fontaine, and Christoph Sticksel. Alethe: Towards a generic smt proof format. In *10th International Workshop on the Pragmatics of SAT/SMT Solving (PxTP)*, volume 336 of *EPTCS*, pages 49–54, 2021.

[3] Clark Barrett, Haniel Barbosa, Martin Brain, Gereon Kremer, Aina Niemetz, Mathias Preiner, Andrew Reynolds, and Cesare Tinelli. cvc5: A versatile and industrial-strength smt solver. In *Computer Aided Verification (CAV)*, volume 13371 of *Lecture Notes in Computer Science*. Springer, 2022.

[4] Clark Barrett, Christopher L. Conway, Morgan Deters, and et al. Cvc4: A new efficient smt solver. In *International Conference on Computer Aided Verification (CAV)*, pages 171–177. Springer, 2011.

[5] Johan Bengtsson and Wang Yi. *Timed Automata: Semantics, Algorithms and Tools*. Springer, 2004.

[6] Tjark Bouton, David C. B. de Oliveira, David Déharbe, and Pascal Fontaine. verit: An open, trustable and efficient smt-solver. In Renate A. Schmidt, editor, *Automated Deduction – CADE 22*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[8] John Ellson and et al. *Graphviz - Open Source Graph Drawing Tools*, 2001.

[9] Lydia Kondylidou, Andrew Reynolds, and Jasmin Blanchette. Augmenting model-based instantiation with fast enumeration.

[10] Kenneth L. McMillan. Interpolation and sat-based model checking. In *Computer Aided Verification (CAV)*, pages 1–13. Springer, 2003.

[11] Andreas Niemetz and Mathias Preiner. Bitwuzla. In Cristina Enea and Akash Lal, editors, *Computer Aided Verification (CAV 2023)*, volume 13965 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2023.

[12] Andreas Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014.

[13] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract dpll procedure to dpll(t). *Journal of the ACM*, 53(6):937–977, 2006.

[14] Aaron Stump, Clark Barrett, and David L. Dill. Producing certified sat proofs with a lightweight proof checker. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2013.

[15] Cesare Tinelli and Clark Barrett. The smt-lib standard: Version 2.6. http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf, 2016.