

Optimierung und Messung beschreibungsllogischer Beweisverfahren

Projektbericht

Roland Illig

14. Februar 2008

Im Wintersemester 2006/07 fand im Rahmen des Diplomstudienganges erstmalig das Hauptstudiumsprojekt „Reasoning Services“ statt. In diesem Projekt wurde ein Tableaubeweiser für Aussagenlogik so erweitert, daß er auch eine einfache Beschreibungslogik behandeln kann. Beschreibungslogiken sind ein entscheidbares Fragment der Prädikatenlogik, aber sie sind auch mächtiger als Aussagenlogik. Aufgrund dieser Mächtigkeit sind einfache Algorithmen, wie sie für die Aussagenlogik verwendet werden können, nicht mehr so interessant, da sie für viele Anwendungen zu rechenintensiv werden. Durch den Einsatz verschiedener Optimierungstechniken wurde die benötigte Rechenzeit verkürzt. Die Auswirkungen jeder dieser Optimierungen können durch eine integrierte Testumgebung gemessen und dadurch zuverlässig festgestellt werden.

Inhaltsverzeichnis

1	Einleitung	2
1.1	Beschreibung des Projekts	2
1.1.1	Vergleich mit Tableau 98	3
1.1.2	Organisation des Quellcodes	3
1.2	Teilbereiche im Projekt	4
1.2.1	Repräsentation der Ausdrücke sowie Parser	4
1.2.2	Reasoning Service und Beweiser	6
1.2.3	Testumgebung	6
2	Design und Implementation der Testumgebung	6
2.1	Grundlegende Konzepte	6
2.1.1	Benchmark (testing.t98.Benchmark)	6

2.1.2	Globale Option (main.config.Option)	7
2.1.3	Konfiguration (testing.conf.Configuration)	7
2.1.4	Messung (testing.Measurement)	7
2.1.5	Messpunkt (testing.MeasuringPoint)	7
2.2	Methoden zur Zeitmessung	8
2.3	Ausgabe der Testumgebung	9
2.3.1	Das kurze Ausgabeformat	9
2.3.2	Das ausführliche Ausgabeformat	9
3	Testdaten	10
4	Aufgetretene Probleme und deren Lösung	10
4.1	Zu wenig Speicher	10
5	Ergebnisse	12
5.1	Vergleich mit Tableau 98	12
5.1.1	Bewertung	12
5.2	Weitere Optimierungen	15
5.2.1	Unnötiger Ballast in ExpansionRule	15
5.2.2	Viele gleiche Objekte in Branchpoint	15
6	Ausblick	16
6.1	Weitere Optimierungen am Beweiser	16
6.2	Messung der einzelnen Optimierungstechniken	16

1 Einleitung

1.1 Beschreibung des Projekts

Im Wintersemester 2006/07 fand erstmalig das Projekt „Reasoning Services: Tableau-Beweiser für Beschreibungslogiken“ statt. Ziel des Projektes war, einen einfachen aussagenlogischen Tableaubeweiser so zu erweitern, dass er auch mit Beschreibungslogik umgehen kann.

Die Basis für unseren Tableaubeweiser war ein in Java geschriebener aussagenlogischer Tableaubeweiser mit graphischer Benutzeroberfläche. Das Kernstück des Beweisers ist ein Satz von Expansionsregeln, die die erlaubten Änderungen an den logischen Ausdrücken festlegen. Aus diesen Regeln werden so lange Expansionen generiert und auf die Ausdrücke angewendet, bis ein Widerspruch gefunden wird oder keine Expansionen mehr möglich sind. Über ein Menü kann der Benutzer die Art der Logik, den zu verwendenden Parser, den Regelsatz und verschiedene Optimierungen einzeln auswählen.

Die Optimierungstechniken stammen aus dem Artikel „Optimising Description Logic Subsumption“ [HorPat 1999]. Dort werden sechs Optimierungstechniken beschrieben, von denen wir vier ausgewählt und implementiert haben. Unser Ziel war, zu messen,

welchen Einfluss die einzelnen Techniken auf die Gesamtperformance (Zeit- und Speicherbedarf) haben und wie gut sie sich kombinieren lassen. Diese Messungen werden von der integrierten Testumgebung durchgeführt.

1.1.1 Vergleich mit Tableau 98

Auf dem „1998 International Workshop on Description Logics (DL’98)“¹ gab es einen Vergleichswettbewerb von verschiedenen Theorembeweisern. Für diesen Wettbewerb wurden zahlreiche Testdaten in mehreren Testreihen generiert. Jede dieser Testreihen besteht aus 21 beschreibungslogischen Ausdrücken mit gleicher Struktur, aber steigender Komplexität. Die Verfahren, mit denen die Formeln erzeugt werden, sind in [HeuSchw 1996] dokumentiert. Die daraus entstandenen Formeln sind in [dl98-test] veröffentlicht, zusammen mit den Testergebnissen verschiedener anderer Beweiser, so dass wir anderen Beweiser damit vergleichen können.

1.1.2 Organisation des Quellcodes

Der Beweiser ist in verschiedene Java-Pakete aufgeteilt. Die folgende Aufstellung ist so geordnet, dass weiter unten stehende Pakete generell auf die weiter oben stehenden aufbauen und möglichst wenig Abhängigkeiten in die andere Richtung bestehen.

expressions enthält die Repräsentationen der Ausdrücke sowie Parser, um sie aus Dateien oder Eingabefeldern zu erzeugen.

tableau enthält die Datenstrukturen für Tableaus (**Tableau**), Tableauzweige (**TBranch**), Tableaeinträge (**TEntry**) und Expansionen (**TExpansion**).

prover enthält zwei Tableaubeweiser, einen für Aussagenlogik (**ALProver**) und einen für Beschreibungslogik (**DLProver**).

reasoningServices benutzen die Beweiser, um „sinnvolle“ Fragen zu beantworten, ohne dass der Fragende sich um die Tableaus kümmern muss. Das Interface **ReasoningService** definiert, was ein Reasoning Service leisten muss, und die meisten anderen Klassen implementieren spezielle Services.

main ist eine Auffangstelle für Klassen, die sonst nirgends reinpassen.

main.config enthält die „globalen Optionen“, eine Möglichkeit, das Verhalten des Beweisers zur Laufzeit zu ändern. Insbesondere können hier die einzelnen Optimierungstechniken ausgewählt werden.

testing stellt automatisierte Messungen (**Measurement**) zur Verfügung, um die Reasoning Services ohne weitere Benutzerinteraktion zu testen.

testing.conf definiert Konfigurationen (**Configuration**), um den Beweiser mit verschiedenen globalen Optionen zu testen.

¹<http://dl.kr.org/dl98/>

testing.t98 enthält Benchmarks, die denen aus Tableau 98 entsprechen. Sie werden benutzt, um einigermaßen vergleichbare Testergebnisse zu produzieren.

userInterface schließlich stellt eine graphische Benutzeroberfläche bereit, um mit den Reasoning Services zu arbeiten.

Die Benennung der Klassen folgt weitgehend den etablierten Java-Konventionen². Klassen, deren Name mit AL beginnt, behandeln die Aussagenlogik, während Klassen, deren Name mit DL beginnt, für die Beschreibungslogik (engl.: description logics) zuständig sind. Bei allen anderen Klassen ist das langfristige Ziel, sie völlig unabhängig von der verwendeten Logik zu machen, um einen modularisierten und damit leicht erweiterbaren und wartbaren Beweiser herzustellen.

1.2 Teilbereiche im Projekt

1.2.1 Repräsentation der Ausdrücke sowie Parser

Die Ausdrücke, die der Beweiser verarbeiten soll, liegen in vielen Fällen in Textform vor und müssen vor dem Verarbeiten eingelesen werden. Diese Aufgabe erledigt das Interface `Parser`, und speziell für Beschreibungslogik die Klasse `DLCSParser`. Die Ausgabe des beschreibungslogischen Parsers ist ein Objekt der Klasse `DLExpression`. So ein Objekt repräsentiert einen beliebig verschachtelten beschreibungslogischen Ausdruck.

Für die Syntax der beschreibungslogischen Ausdrücke wählten wir die in Anhang A von [DLHB 2003] verwendete, in der Hoffnung, dass es sich bei diesem Buch um ein Standardwerk handelt und dementsprechend viele Softwareprojekte sich danach richten. Wir hatten jedoch nicht berücksichtigt, dass das Buch erst 2003 in der ersten Auflage erschienen war, so dass alle Softwareprojekte, die vorher entstanden, es gar nicht in genau dieser Form kennen konnten. Unsere hauptsächlich verwendeten Testdaten waren aus dem Jahr 1998 und verwenden eine sehr ähnliche, aber eben nicht identische Syntax, und so mussten wir den Parser an deren Syntax anpassen. Das geschieht über die globale Option `parser.case-sensitive`.

Aus der Syntax der beschreibungslogischen Ausdrücke ([DLHB 2003, Anhang A]) wurde deutlich, dass es verschiedene Typen von Ausdrücken gibt: Konzepte, Rollen, Konstanten. Die naheliegende Idee, jeden dieser Typen auf eine Java-Klasse abzubilden, um Typfehler weitgehend auszuschließen, führt zu viel Redundanz, da der Code für die syntaktischen Kategorien weitgehend identisch ist. Auf lange Sicht besser ist es deshalb, nur eine Klasse für alle Arten von Ausdrücken zu verwenden und die Typüberprüfung im Konstruktor dieser Klasse durchzuführen. Die gültigen Argumente für die Operatoren werden mit Hilfe der Aufzählungstypen `DType` und `DOperator` definiert. So hat zum Beispiel der Operator `And` beliebig viele Argumente vom Typ `Concept`, und auch das Ergebnis ist vom Typ `Concept`. Der Operator `Related` hat Argumente vom Typ `Constant`, `Constant` und `Role` und erzeugt daraus ein `ABoxAxiom`. Ein weiterer praktischer Grund für die Verwendung von nur einer Klasse ist, dass moderne Java-Compiler eine Warnung erzeugen, wenn ein Aufzählungstyp in einer `switch`-Anweisung unvollständig abgefragt

²<http://java.sun.com/docs/codeconv/>

wird. Dadurch zeigt der Compiler bei späteren Erweiterungen von `DType` gleich an, wo der Code geändert werden muss.

Nachdem ein Ausdruck (`DExpression`) erzeugt wird, kann er nicht mehr geändert werden. Dadurch ist es möglich, Ausdrücke mehrfach zu verwenden, ohne sie jedes Mal neu zu erzeugen. Bei so einem Vorgehen muss man in Java aufpassen, dass wirklich alle Teile des Ausdrucks gegen versehentliches Überschreiben geschützt sind. Bei Arrays kann das leicht übersehen werden, da deren Inhalt jederzeit überschrieben werden kann, selbst wenn die Array-Variable als `final` deklariert ist. Dieses Array muss daher als „privat“ behandelt werden und darf außerhalb der Klasse `DExpression` nicht änderbar sein. Insbesondere darf die Methode `getSubexpressions` nicht einfach das Array zurückgeben. Um dieses Problem zu umgehen, gibt es vier Möglichkeiten:

1. Bei jedem Aufruf von `subExpressions` wird mit der Methode `clone()` eine Kopie des Arrays angelegt und zurückgegeben. Der Aufrufer kann dieses Array manipulieren, ohne den ursprüngliche Ausdruck zu beeinflussen. Allerdings kostet dieses Verfahren viel Zeit, vor allem, wenn die Methode häufig benutzt wird. Als kleine Abhilfe gibt es die Methode `getNthSubEx`, die nur einen einzigen Teilausdruck zurückgibt, denn in diesem Fall muss nichts kopiert werden.
2. Es wird dokumentiert, dass das von der Methode zurückgegebene Array nicht verändert werden darf. Alle Aufrufer halten sich an die Dokumentation. Dieses Verfahren ist sehr schnell, da keine zusätzlichen Kopien des Arrays erzeugt werden müssen. Allerdings kann man in der Praxis nicht davon ausgehen, dass Programmierer diese Dokumentation jederzeit kennen und sich daran halten, so dass die Daten doch irgendwann inkonsistent werden. Da die Fehlersuche in diesem Fall sehr schwierig ist, ist dieses Verfahren nicht zu empfehlen.
3. Wir bitten Sun, das sowieso schon reservierte Schlüsselwort „`const`“ endlich in die Sprache zu integrieren genau so zu definieren, wie wir es gerade brauchen. Dieses Verfahren ist sehr zeitaufwendig und für ein einsemestriges Projekt daher nicht geeignet. Außerdem haben das andere auch schon versucht, aber Sun ist dagegen.³
4. Wir verwenden einen Dialekt der Sprache Java, der zusätzlich zu den Objekttypen sogenannte „Werttypen“ unterstützt, denn Werttypen haben genau die Eigenschaften, die wir von `DExpression` erwarten:
 - Werte sind zeitlos: Es gibt keinen expliziten Konstruktor, sondern die Werte werden als „irgendwo vorhanden“ angesehen, und man kann jederzeit auf die Werte zugreifen.
 - Werte können nicht verändert werden.
 - Bei Vergleichen werden die Werte elementweise verglichen.

Solche Sprachen sind in Arbeit (siehe [Rathlev 2006]), aber noch nicht allgemein etabliert, so dass wir uns erstmal in dieses Konzept hätten einarbeiten müssen. Es

³http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4211070

war uns zu Beginn des Projekts auch noch nicht bewusst, dass es solche Sprachen gibt.

1.2.2 Reasoning Service und Beweiser

Die Schnittstelle eines logischen Beweisers wird oft auf der Ebene der *Reasoning Services* definiert. Ein Reasoning Service ist ein Dienst, der eine logische Frage beantworten kann, im Falle der Beschreibungslogik zum Beispiel:

- Ist das Konzept C erfüllbar? (SatisfiabilityService)
- Ist das Konzept C ein Teilkonzept von D ? (SubsumptionService)
- Schließen sich die Konzepte C und D gegenseitig aus? (DisjointnessService)
- Welche Objekte erfüllen das Konzept C ? (RetrievalService)

Die einzelnen Reasoning Services benutzen den eigentlichen Beweiser, indem sie ein „initiales Tableau“ erzeugen, das der Beweiser zu beweisen versucht. Je nachdem, ob das klappt oder nicht, fällt auch die Antwort des Reasoning Service aus.

1.2.3 Testumgebung

Die Testumgebung misst, wie viel Zeit der Beweiser für einen Ausdruck benötigt und stellt Methoden bereit, Messreihen durchzuführen und deren Ergebnisse kurz und prägnant darzustellen. Das wird insbesondere für die Optimierungstechniken benötigt, um deren Auswirkungen einzuschätzen.

2 Design und Implementation der Testumgebung

2.1 Grundlegende Konzepte

Die Testumgebung stellt Methoden bereit, um automatisiert Benchmarks durchzuführen. Die Benchmarks laufen in einer bestimmten *Konfiguration* ab, die vor dem Start eingestellt wird. Dann werden die Einzelmessungen durchgeführt und die Messergebnisse in Dateien festgehalten.

2.1.1 Benchmark (testing.t98.Benchmark)

Ein Benchmark ist eine Sammlung von einzelnen Messungen in einer bestimmten Konfiguration und auf einem bestimmten Rechnersystem. Die Ergebnisse der Messungen werden einmal in verdichteter Form (für die Projektleitung) und einmal in der ausführlichen Form (für die Entwickler) ausgegeben. Das Interface ist in dem Paket `testing.t98` und nicht etwa `testing`, da wir nur für den Tableau-98-Test so umfangreiche Testdaten haben, dass es sich lohnt, dort etwas „Benchmark“ zu nennen.

2.1.2 Globale Option (`main.config.Option`)

Um zu messen, welchen Einfluss die einzelnen Optimierungstechniken auf die benötigte Zeit und den Speicherbedarf haben, können die Techniken einzeln aktiviert oder deaktiviert werden. Das geschieht mit den sogenannten „globalen Optionen“, die in der Java-Klasse `main.GlobalOptions` zusammengefasst sind und in der GUI unter dem Menüpunkt `Preferences` → `Global options` eingestellt werden können.

2.1.3 Konfiguration (`testing.conf.Configuration`)

Die Einstellungen der Optionen werden als Konfiguration bezeichnet. Es gibt einige vordefinierte Konfigurationstypen:

`testing.conf.DefaultConfiguration` stellt alle Optionen auf ihre Standardwerte ein.

`testing.conf.BooleanOptionConfiguration` stellt eine einzelne Option auf einen bestimmten Wert ein.

`testing.conf.CombinedConfiguration` kombiniert eine beliebige Anzahl von Konfigurationen.

`testing.conf.OptimizedConfiguration` stellt alle Optimierungen an.

Mit Hilfe des Typs `CombinedConfiguration` kann man verschiedene Konfigurationen verketteten und erhält dadurch eine flexible Möglichkeit, beliebige Konfigurationen zu definieren.

2.1.4 Messung (`testing.Measurement`)

Eine Messung ist eine kleinste Einheit eines Benchmarks. Sie enthält Programmcode mit dazu passenden Daten und führt diesen aus. Dabei wird die Zeit gemessen, die der Rechner für die Ausführung benötigt. Überschreitet die Zeit eine gewisse Grenze, wird die Messung abgebrochen und als Zeitüberschreitung gewertet. Andernfalls wird überprüft, ob der Code das erwartete Ergebnis berechnet hat.

Das Java-Interface `testing.Measurement` definiert, was eine Messung tut, aber stellt keine Möglichkeit bereit, die Messung auch durchzuführen. Die Durchführung wird von der Klasse `testing.MeasurementRunner` erledigt.

Die so erhaltenen Zeitmessungen sind sinnvoll, um unseren Beweiser mit anderen Beweisern zu vergleichen. Dazu müssen allerdings beide Beweiser auf der gleichen Rechnerart ausgeführt werden, damit die Ergebnisse aussagekräftig verglichen werden können.

2.1.5 Messpunkt (`testing.MeasuringPoint`)

Ein Messpunkt ist ein Zähler, der für verschiedene Aufgaben eingesetzt werden kann. Momentan ist der Haupteinsatzzweck zu zählen, wie oft bestimmte Methoden aufgerufen werden. Gegenüber expliziten Zählervariablen haben Messpunkte den Vorteil, dass sie beim Erzeugen automatisch bei einer zentralen Stelle (`testing.AllMeasuringPoints`)

registriert werden, so dass sich die einzelnen Klassen nicht um die Ausgabe der Messwerte kümmern müssen. Außerdem beschränken die Methoden die Aktionen, die mit dem Messpunkt durchgeführt werden können, so dass es unwahrscheinlicher ist, durch Programmierfehler falsche Messergebnisse zu bekommen.

Die Aufgabe, die die Messpunkte erfüllen, könnte man auch mit Hilfe eines Java-Debuggers über das „Java Virtual Machine Tools Interface“ (JVMTI) erledigen. Mit aktuellen Java-Umgebungen wird ein Programm namens `hprof` mitgeliefert, das diese Funktion erfüllt. Es hat jedoch den großen Nachteil, dass es nach dem Prinzip „alles oder nichts“ misst. Das heißt, es werden entweder alle Methodenaufrufe gezählt oder gar keine. Und wenn alle Methodenaufrufe gezählt werden, geht das sehr zu Lasten der Ausführungsgeschwindigkeit. Der Vorteil dabei ist, dass man nicht schon vorher wissen muss, an welchen Stellen man die Messpunkte anbringen will.

Die Messpunkte sind hier etwas flexibler: Nach einer ersten Analyse mit `hprof` kann man sie gezielt an den Stellen anbringen, die man testen möchte. Diese Stellen sind auch nicht auf einzelne Methodenaufrufe beschränkt, sondern es können auch mehrere Messpunkte in einer Methode eingesetzt werden, um verschiedene Ausführungspfade zu untersuchen (Beispiel: `DLExpression.equals`).

Das Programm `hprof` bietet noch eine zweite Möglichkeit des Profilings: In regelmäßigen Zeitabständen (Standardeinstellung: 10 ms) wird ermittelt, in welcher Methode sich die laufenden Threads gerade befinden und darüber eine Übersicht erstellt. Die Ergebnisse dieses Verfahrens hängen vom Zufall ab, da nicht sichergestellt werden kann, dass die Messungen immer zu den gleichen Zeitpunkten stattfinden. Daher sind die so erhaltenen Zahlen nicht uneingeschränkt reproduzierbar, aber genau diese Eigenschaft hätte ich gerne.

Messpunkte sind nur für Vergleiche innerhalb eines Beweisers sinnvoll, da andere Beweiser oft völlig anders strukturiert sind. Sie sind jedoch sehr hilfreich, um bei Codeänderungen festzustellen, welche Methoden (oder allgemein Programmteile) gute Kandidaten für weitere Optimierungen sind.

2.2 Methoden zur Zeitmessung

Um die Ausführungszeit der einzelnen Testfälle zu messen, bieten sich mehrere Verfahren an.

- Messung der insgesamt benötigten Zeit (Wartezeit). Hierzu wird jeweils am Anfang und am Ende eines Testdurchlaufs die aktuelle Systemzeit ermittelt, und die Differenz dieser Zeitpunkte wird als die benötigte Zeit angesehen. Das Messergebnis ist praktisch relevant, da genau die Zeit ermittelt wird, die ein Benutzer auf das Ergebnis warten muss.
- Messung der benötigten Rechenzeit. Hierzu muss das Betriebssystem oder die verwendete Hardware eine Methode bereitstellen, die in einem Thread oder Prozess benötigten Maschinenbefehle mitzuzählen. Dieses Ergebnis ist dann interessant, wenn das Programm parallel mit anderen Programmen läuft, um die Auslastung des Rechners vorherzusagen.

Da der Beweiser nur einen einzigen Thread benutzt, um die Arbeit auszuführen, und während des Beweises nicht auf langsame Speichermedien (Datenträger, Netzwerk) zugreift, sind die Ergebnisse beider Verfahren in diesem Fall sehr ähnlich. Bei Verwendung mehrerer Threads ist die Rechenzeit üblicherweise größer als die Wartezeit, besonders auf Rechnern mit mehreren Prozessoren. Bei Programmen, die auf langsame Speichermedien zugreifen müssen, entstehen Rechenpausen, die sich nur auf die Wartezeit, nicht aber auf die Rechenzeit auswirken. Da die Ergebnisse der beiden Verfahren sehr ähnlich sind, habe ich mich für das erste entschieden, da es in Java leichter zu realisieren ist.

Ein wichtiger Punkt bei allen physikalischen Messungen ist die Messgenauigkeit. Die Zeitmessungen unterliegen mehr oder weniger zufälligen, auf jeden Fall aber unvermeidbaren Schwankungen. Um diese Schwankungen herauszufinden, wurden die Messungen `t_dlexpression_equals_nosame{2,3,4}` durchgeführt, direkt nacheinander und ohne den Rechner dabei anderweitig zu benutzen. Die Gesamtlaufzeit der Messung variierte dabei um ca. 1%. Bei der Interpretation der Messergebnisse muss diese Ungenauigkeit stets berücksichtigt werden.

2.3 Ausgabe der Testumgebung

Die Ausgabe eines Benchmarks besteht aus zwei Dateien: einer kurzen, die nur die wirklich essentiellen Ergebnisse enthält, und einer langen, die insbesondere für die Entwickler des Tableaubeweisers interessant ist, um zu sehen, welche Teile des Beweisers besonders häufig ausgeführt werden.

2.3.1 Das kurze Ausgabeformat

Das kurze Ausgabeformat entspricht dem Format, das auch in den Beispieldateien in `[dl98-test]` verwendet wird. Eine explizite Dokumentation des Formats lag nicht vor, so dass als einzige Quelle der LISP-Quellcode in `dl98-tests.lsp` blieb. Das Format besteht aus mehreren Zeilen. Eine Zeile besteht aus Feldern, die jeweils durch Leerzeichen getrennt sind. Wie viele Felder eine Zeile hat, ist von dem jeweiligen Test abhängig. Das erste Feld ist jedoch bei allen Tests gleich: Es ist der Name der Testdaten.

Bisher sind nur der Erfüllbarkeitstest für beschreibungslogische Ausdrücke ohne Berücksichtigung einer TBox (T98sat) und die Konsistenzprüfung einer TBox (T98kb) implementiert. Das Ausgabeformat für diese beiden Tests hat 5 Felder pro Zeile: `Testname`, `Anzahl-p`, `Ergebnis-p`, `Anzahl-n`, `Ergebnis-n`. `Anzahl-p` ist die Anzahl der nicht erfüllbaren Ausdrücke, die der Beweiser innerhalb der vorgegebenen Zeit (100 Sekunden) berechnen konnte. `Ergebnis-p` ist `Y`, wenn alle berechneten Ergebnisse korrekt waren, sonst `N`. Die Felder `Anzahl-n` und `Ergebnis-n` haben die analoge Bedeutung für die nicht-erfüllbaren Ausdrücke.

2.3.2 Das ausführliche Ausgabeformat

Das ausführliche Ausgabeformat ist eine Eigenentwicklung. Wie auch das kurze Ausgabeformat ist es zeilenbasiert, enthält aber mehr redundante Informationen, so dass auch Leser, die bisher nicht damit gearbeitet haben, schneller damit zurecht kommen sollten.

Jede Zeile besteht aus einem Typen (erstes Wort) und den Argumenten (alle folgenden Wörter). Die Ausgabe für jeden Test besteht aus mehreren Zeilen.

Die erste Zeile hat den Typ `test` und ein Argument, den Namen der Messung. Dieser Name ist üblicherweise der Java-Klassenname, ergänzt um Informationen, um gleichartige Messungen zu unterscheiden. Wenn zum Beispiel aus einer Datei 10 Ausdrücke eingelesen werden und dann von der gleichen Messung verarbeitet werden, sollten der Dateiname und die Nummer des Ausdrucks in den Namen integriert werden.

Nach dem Namen der Messung folgen die Einstellungen der globalen Optionen (Typ: `option`). Pro Zeile wird jeweils der Wert einer Option ausgegeben. Die Argumente sind der Optionsname und der Wert. Die Bedeutung der einzelnen Optionen ist in der Java-Klasse `main.GlobalOptions` dokumentiert.

Der Test geht davon aus, dass mit den obigen Optionen alle relevanten Parameter eingestellt sind. Daher werden keine weiteren Details zur Systemkonfiguration (etwa der verwendete Rechnertyp, die Größe des Speichers, etc.) ausgegeben. Damit sind jetzt alle relevanten Einstellungen getan, und die Messung kann durchgeführt werden. Das Ergebnis der Durchführungen wird in den nächsten beiden Zeilen erfasst. Die erste Zeile (Typ: `state`) gibt an, ob die Messung erfolgreich durchgelaufen ist (`CorrectResult`), ein falsches Ergebnis geliefert hat (`WrongResult`), wegen Zeitüberschreitung (`TimedOut`) oder aus einem anderen Grund abgebrochen wurde (`Aborted`). Die zweite Zeile des Ergebnisses (Typ: `time`) gibt die benötigte Zeit in Millisekunden an. Dies ist die Differenz der Uhrzeiten zwischen Start und Ende der Messung. Eine andere Möglichkeit wäre, die tatsächlich benötigte Rechenzeit zu erfassen, aber das ist mit Java-Bordmitteln nicht machbar.

Nach dem Ergebnis folgen die Messpunkte (Typ: `point`), jeweils mit Name und Zählerstand als Argumenten. Ein Beispiel für das ausführliche Ausgabeformat ist in Abbildung 1 auf Seite 11 zu sehen.

3 Testdaten

Zum Testen der Geschwindigkeit haben wir praktisch nur die Testdaten aus Tableau '98 verwendet. Die Testdaten befinden sich in `[dl98-test]` und sind dort in der Datei `krss.ps` beschrieben, nur leider stimmt die Beschreibung nicht mit den tatsächlichen Testdaten überein. Die Beschreibung definiert (`define-primitive-concept CN C`), während die Testdaten nur (`define-primitive-concept CN`) verwenden. Wahrscheinlich ist mit der letzteren Form so etwas wie eine Konzeptnamendeklaration gemeint, und so wird es von unserem Parser auch behandelt.

4 Aufgetretene Probleme und deren Lösung

4.1 Zu wenig Speicher

Beim Durchführen der Messungen kam es immer wieder vor, dass sich die Java-VM mit den Worten „`java.lang.OutOfMemoryError: Java heap space`“ verabschiedete, obwohl im Code schon Vorkehrungen getroffen waren, um gerade diesen Fehler zu vermeiden:

test	testing.SatisfiabilityMeasurement(0)	
option	dlexpansionrulebool.semanticbranching	true
option	dlexpansionrulebool.simplification	true
option	dlexpression.normalising	true
option	expansion:branching_before_roles	false
option	expansion:roles_add_before_modify	true
option	parser.case-sensitive	false
option	prover.nonatomicclosure	false
option	prover.use_ddb	true
option	prover.verbose	false
option	tableau.fulltableautostring	false
option	useTableauStatusThreads	false
state	CorrectResult	
time	3478	
point	Token.constructor	0
point	DLEExpression.new	8559
point	DLEExpression.constructor	8559
point	DLEExpression.operatorName	281865
point	DLEExpression.compareTo	249144
point	DLEExpression.hashCode	56128
point	DLEExpression.subExpressions	18267
point	DLProver.expand(Tableau)	1438
point	DLProver.expand(TBranch)	1438
point	tableau.expand	1438
point	tableau.constants	1638

Abbildung 1: Die ausführliche Ausgabe einer Messung enthält viele Details, die für die Entwickler interessant sind.

- Die Durchführung des eigentlichen Tableaubeweises findet in einem eigenen Thread statt, so dass eine Exception oder sonstige Fehler nicht gleich zum Abbruch des Hauptprogramms führen.
- Das Tableau ist die Datenstruktur, an der alle Zweige, Einträge und Ausdrücke hängen. Bei einem Abbruch wegen Speichermangel sollten diese ganzen Objekte sofort aufgeräumt werden können. Das heißt, der Zugriff darauf sollte nur über lokale Variablen erfolgen, nicht über Felder von anderen Objekten, etwa den Reasoning Services. Dadurch werden die Objekte zur Garbage Collection freigegeben, sobald die Methode beendet ist. Genauso war es auch schon programmiert, das konnte also nicht der Grund für den anhaltenden Speichermangel sein.

Der anhaltende Speichermangel trat bei der optimierten Konfiguration auf. Und tatsächlich war das auch die Ursache, denn während der Normalisierung der Ausdrücke werden alle Ausdrücke, die erzeugt werden, in einer Tabelle gespeichert, damit keine überflüssigen Objekte mit den gleichen Eigenschaften erzeugt werden. Diese Tabelle wurde jedoch während eines Programmlaufs niemals aufgeräumt, so dass sie zwangsläufig immer größer wurde und schließlich den kompletten Speicher eingenommen hatte. Nachdem dieses Problem behoben wurde (vor und nach jeder Messung wird die Tabelle gelöscht), liefen die Messungen problemlos durch.

5 Ergebnisse

5.1 Vergleich mit Tableau 98

Ein Ziel des Projekts war der Vergleich unseres Beweisers mit den Programmen, die am Tableau 98-Wettbewerb teilgenommen haben. Zwei dieser Programme sind FaCT und Kris. Die Programme und deren Testumgebungen sind in den Artikeln in [dl98-test] im Verzeichnis `LaTeX` beschrieben. Der Testrechner war ein Intel Pentium II mit 200 MHz und 64 MB Hauptspeicher. Unser Testrechner war ein AMD Athlon mit 1800 MHz und 1 GB Hauptspeicher, wobei die Java-VM allerdings auf 128 MB beschränkt wurde. Die folgenden Tabellen sind daher nicht direkt vergleichbar. Für einen direkten Vergleich müsste bei unseren Tests die Zeitschranke für jeden einzelnen Ausdruck von 100 Sekunden auf 10 Sekunden reduziert werden. Und auch dann gibt es noch signifikante Änderungen, da der Prozessortyp und die verwendete Programmiersprache unterschiedlich sind. Zum Beispiel ist FaCT in C++ geschrieben, einer Sprache, die viel Wert auf effizienten Code legt. Java hingegen läuft in einer virtuellen Maschine, und obwohl es in den letzten Jahren viel Entwicklung im Bereich der Just-in-Time-Compiler gab, ist C++ generell immer noch schneller.

5.1.1 Bewertung

Unser Beweiser ist insgesamt sehr viel langsamer als FaCT und Kris, da man beim Vergleich der Ergebnisse noch die unterschiedliche Rechnergeschwindigkeit und die Entwicklungen bei der Code-Optimierungen der Java-VM berücksichtigen muss. Trotzdem ist

Abbildung 2: Testergebnisse für T98sat von FaCT 1.5

Test	Incoherent		Coherent	
	Size	Correct	Size	Correct
k_branch	6	Y	4	Y
k_d4	21	Y	8	Y
k_dum	21	Y	21	Y
k_grz	21	Y	21	Y
k_lin	21	Y	21	Y
k_path	8	Y	6	Y
k_ph	6	Y	7	Y
k_poly	21	Y	21	Y
k_t4p	21	Y	21	Y

Abbildung 3: Testergebnisse für T98sat von Kris

Test	Incoherent		Coherent	
	Size	Correct	Size	Correct
k_branch	3	Y	3	Y
k_d4	8	Y	7	Y
k_dum	15	Y	21	Y
k_grz	13	Y	21	Y
k_lin	6	Y	9	Y
k_path	4	Y	12	Y
k_ph	4	Y	5	Y
k_poly	11	Y	21	Y
k_t4p	7	Y	6	Y

das Ergebnis für ein einsemestriges Projekt nicht schlecht. Da wir uns bisher nur auf die Umsetzung der Algorithmen in Code gekümmert haben und nicht um programmiertechnische Optimierungstechniken (verwendung effizienter Datenstrukturen, schnellen Code gegenüber lesbarem Code bevorzugen, Profiling), ist auch noch Potential vorhanden, unseren Beweiser schneller zu machen.

Interessant ist vor allem der Vergleich unseres unoptimierten Beweisers (4) mit dem optimierten (5). Die Optimierungstechniken haben sich insgesamt positiv auf die Tests ausgewirkt. Bei den unerfüllbaren Formeln schneiden nur einige der Testfälle besser ab, insbesondere k_ph mit einer Steigerung von +9 sticht hier heraus. Bei den erfüllbaren Formeln sind die Auswirkungen der Optimierungen deutlicher zu erkennen: In fast allen Testreihen konnten mehr Ausdrücke innerhalb der Zeitschranke bearbeitet werden. Auffällig ist auch die enorme Steigerung in der Reihe k_d4 von 2 auf 21. ist der Unterschied deutlicher Bei k_d4 und k_grz konnte der optimierte Beweiser alle 21 Ausdrücke bearbeiten, der unoptimierte sehr viel weniger. Nur in einem einzigen Fall (k_poly, inco-

Abbildung 4: Testergebnisse für T98sat von unserem unoptimierten Beweiser

Test	Incoherent		Coherent	
	Size	Correct	Size	Correct
k_branch	1	Y	1	Y
k_d4	2	Y	2	Y
k_dum	12	Y	0	Y
k_grz	21	Y	0	Y
k_lin	21	Y	21	Y
k_path	2	Y	2	Y
k_ph	3	Y	2	Y
k_poly	12	Y	12	Y
k_t4p	1	Y	0	Y

Abbildung 5: Testergebnisse für T98sat von unserem optimierten Beweiser

Test	Incoherent		Coherent	
	Size	Correct	Size	Correct
k_branch	4	Y	4	Y
k_d4	1	Y	21	Y
k_dum	12	Y	3	Y
k_grz	21	Y	21	Y
k_lin	21	Y	21	Y
k_path	2	Y	4	Y
k_ph	12	Y	6	Y
k_poly	12	Y	11	Y
k_t4p	1	Y	1	Y

herent) war das Ergebnis schlechter. Aus den ausführlichen Testergebnissen kann man ablesen, dass hier der optimierte Beweiser durchgängig 20 bis 50 Prozent mehr Zeit benötigt.

Ganz allgemein kann man feststellen, dass verschiedene Beweiser die Messreihen unterschiedlich gut handhaben können. In der Testreihe `k_d4` ist FaCT zum Beispiel bei nicht-erfüllbaren Ausdrücken schneller als bei erfüllbaren (21 : 8), während das bei unserem Beweiser gerade umgekehrt ist (1 : 21).

Wie gut die Optimierungen sich auf praktisch relevante Testdaten auswirken, können wir nicht beurteilen, da wir bisher nur die Testdaten aus Tableau 98 benutzt haben. Diese Testdaten sind, wie in [HeuSchw 1996] beschrieben, speziell konstruierte Formeln, um Beweiser zu testen und zu vergleichen. Sie stammen zwar teilweise aus praktischen Beispielen, haben aber nicht den Anspruch, den typischen Anwendungsbereich eines Beweisers abzudecken. Zudem kann dieser „typische“ Anwendungsbereich auch je nach Anwendung sehr unterschiedlich sein.

5.2 Weitere Optimierungen

5.2.1 Unnötiger Ballast in ExpansionRule

Mit Hilfe der aus der Testumgebung gewonnenen Daten lässt sich bei Codeänderungen jetzt genau feststellen, welche Auswirkungen sie haben. Ein Beispiel ist eine kleine Änderung im Interface `ExpansionRule`. Bisher gibt die Methode `getExpansions` eine `Collection<TExpansion>` zurück. Wenn man sich allerdings den Kommentar dazu ansieht („... liefert *eine* Expansion ...“), fällt auf, dass die Verwendung der `Collection` völlig überflüssig ist, da sie eh nur ein Element enthält. Die Änderung ist daher, die Methode so zu ändern, dass sie die `TExpansion` direkt zurückgibt.

Bevor jedoch der Code geändert wird, muss zunächst eine Referenzmessung durchgeführt werden. Damit die Messungen nachvollziehbar sind, wurde zu jeder der folgenden Messungen ein CVS-Tag angelegt, und die Ergebnisse sind unter einem entsprechenden Namen im Verzeichnis `testresults` abgelegt.

Für die Messung `t_expansionrule_ref` wurde das Programm `T98SatisfiabilityDemo` mit der optimierten Konfiguration ausgeführt. Nach der Änderung wurde mit der gleichen Konfiguration die Messung `t_expansionrule_simple` durchgeführt. Die Ergebnisse sind sehr ähnlich, aber insgesamt ist der Beweiser 4 Prozent schneller geworden. Diese Zahl ergibt sich, wenn man die sich entsprechenden einzelnen Messungslaufzeiten miteinander vergleicht, sofern sie nicht wegen Zeitüberschreitung abgebrochen wurden. Von allen Quotienten der Laufzeit wurde nun der Median genommen, da wegen eines Ausreißers der Mittelwert nicht sehr aussagekräftig war.

Diese 4 Prozent sind zwar nicht viel, aber wenn man bedenkt, dass zu diesem Zeitpunkt der Beweiser durch einen völlig anderen Programmteil massiv ausgebremst wird (`DLExpression.equals` wird zu häufig aufgerufen, bedingt durch `TBranch.containsNegatedEntry`), ist das schon eine gute Leistung. Sehr viel interessanter wird der Vergleich jedoch, wenn das eben genannte Problem behoben ist.

5.2.2 Viele gleiche Objekte in Branchpoint

Die Klasse `Branchpoint` ist technisch gesehen nur ein Wrapper um eine kleine Zahl, mit der Einschränkung, dass es nur eine Nachfolger-Operation gibt, keine beliebige Manipulierbarkeit wie bei einer einfachen `int-Variable`.

Im Laufe eines Beweises werden viele dieser Objekte erzeugt, die mit `equals()` nicht unterscheidbar sind. Bei einem Testlauf von `T98SatisfiabilityDemo` wurden 24 924 060 davon erzeugt. Der höchste dabei auftretende Index war 371. Es ist fraglich, ob der Beweiser schneller wird, wenn man die Objekte nicht jedes Mal neu erzeugt, sondern sie nur einmal erzeugt, in einem Cache abspeichert und dann daraus ausliefert. Um die Frage zu beantworten, wurden die zwei Messungen `t_branchpoint_cache_before` und `t_branchpoint_cache_after` durchgeführt. Das Ergebnis war, dass der Beweiser dadurch etwas langsamer geworden ist (ca. 1 %). Auch nach einer zweiten Optimierung (Ersetzen der `ArrayList` durch ein einfaches `Array`), wurde das Ergebnis nicht besser, sondern noch schlechter. Die Messung (`t_branchpoint_cache_array`) ergab, dass das Programm da-

durch noch langsamer wurde (ca. 3 %). Deshalb sind diese Messergebnisse jetzt im Code kommentiert, die Änderungen am Code sind wieder rückgängig gemacht.

6 Ausblick

6.1 Weitere Optimierungen am Beweiser

Unser Tableaubeweiser hat noch viel Optimierungspotential. Insbesondere sollte der gesamte Code auf effiziente Algorithmen und geeignete Datenstrukturen untersucht werden.

Zum Beispiel ist das lineare Suchen in einer `ArrayList` sehr zeitaufwendig, andererseits sind die Iteratoren in dieser Klasse effizienter als bei anderen Containertypen. Das betrifft vor allem die Klasse `TBranch` und dort die Methoden `findConflict` und `containsNegatedEntry`. Die Listen der Tableaueinträge enthalten häufig mehr als 1000 Einträge, und das ist weit über der Grenze, ab der sich eine nicht-lineare Suche lohnt. Erste Optimierungen in dieser Richtung waren nicht sonderlich erfolgreich (siehe die globale Option `useFindConflict`, aber ich erwarte hier noch eine gute Geschwindigkeitssteigerung. Die Auswahl geeigneter Datenstrukturen ist zum Beispiel in [Shirazi 2000] beschrieben und erläutert.

Die in Java 1.5 eingeführte erweiterte For-Schleife sieht im Quellcode sehr schön und effizient aus, die Frage ist jedoch, ob das in der Laufzeit ähnlich aussieht. Beim Iterieren über beliebige Java-Collections muss jedes Mal ein `Iterator` erzeugt werden, und da nicht alle Datenstrukturen intern so einfach aufgebaut sind wie eine `ArrayList`, könnte das eine weitere Stelle für Optimierungen sein. Zudem sind die Iteratoren für unsere Zwecke viel zu mächtig, man kann mit ihnen nämlich auch Elemente löschen. Die dafür benötigten Informationen müssen von jedem Iterator mitgeführt werden.

6.2 Messung der einzelnen Optimierungstechniken

Bisher habe ich nur zwei Konfigurationen des Beweisers getestet: Einmal die `UnoptimizedConfiguration`, die den Stand des Beweisers zur Hälfte des Semesters repräsentiert, und `OptimizedConfiguration`, in der alle Optimierungen aktiviert sind. Ob letztere die wirklich effizienteste Konfiguration ist, habe ich nicht gemessen. Weitere interessante Konfigurationen sind die, in denen jeweils nur eine der Optimierungstechniken aktiviert ist und die, in denen jeweils eine Optimierungstechnik deaktiviert ist.

Literatur

[dl98-test] Ian Horrocks, Peter F. Patel-Schneider, *Instructions for DL'98 Systems Comparison*, Datei `README` in <http://dl.kr.org/dl98/comparison/dl98-test.tar.gz>.

[DLHB 2003] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, Peter F. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation, and Applications*, Cambridge University Press, 2003.

- [HorPat 1999] Ian Horrocks, Peter F. Patel-Schneider, *Optimising Description Logic Subsumption*, Journal of Logic and Computation, 9:3, June 1999, pages 267-293.
- [HeuSchw 1996] A. Heuerding and S. Schwendimann, *A benchmark method for the propositional modal logics K, KT, S4*, Technical report IAM-96-015, University of Bern, Switzerland, 1996.
- [Rathlev 2006] Jörg Rathlev, *Ein Werttyp-Konstruktor für Java*, Diplomarbeit am Fachbereich Informatik der Universität Hamburg, 2006.
- [Shirazi 2000] Jack Shirazi, *Java Performance Tuning*, O'Reilly & Associates, 2000.
- [Solth 2007] Arved Solth, *Effiziente Expansion durch Redundanzvermeidung in einem beschreibungslogischen Tableau-Beweiser*, Baccalaureatsarbeit am Fachbereich Informatik der Universität Hamburg, 2007.