Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 88

Berechnung und Darstellung der Verwendungshäufigkeiten von programmiersprachlichen Konzepten

Julian Ziegler

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Betreuer/in:	Dipl. Ing Torsten Görg
Beginn am:	15. August 2013
Beendet am:	14. Februar 2014
CR-Nummer:	D.2.7, D.2.8

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

1 Einleitung

Dieses Dokument stellt den Weg zur Lösung der Aufgabenstellung „Berechnung und Darstellung der Verwendungshäufigkeiten von programmiersprachlichen Konzepten“ vor. Dazu wird zunächst die Aufgabenstellung im Detail und anschließend die Struktur dieser Arbeit vorgestellt.

1.1 Aufgabenstellung

Wartung ist ein wichtiger Aspekt der Softwaretechnik, dessen Aufwand mit der zunehmenden Größe von Softwaresystemen kontinuierlich steigt. Um diesen Aufwand zu reduzieren macht es Sinn, an einem der Punkte anzusetzen, für den ein Großteil der Zeit von Wartungsarbeiten anfällt: das Verstehen des vorliegenden Programmcodes. Diesen Prozess kann man durch Programme unterstützen, welche dem Benutzer bestimmte Informationen über den vorliegenden Quellcode darlegen.

Ziel dieser Arbeit ist die Entwicklung eines solchen Programms. Es soll dabei eine Statistik über die Nutzung der verschiedenen Sprachkonzepte im vorliegenden Programm erstellen und dem Benutzer anschaulich präsentieren. Diese Statistik soll sich jedoch nicht nur auf die Konzepte der jeweiligen Programmiersprache beziehen, in der der zu analysierende Quelltext vorliegt, sondern auch Bibliotheken umfassen, die mit ihrer bereitgestellten Funktionalität ein elementarer Bestandteil eines Programms sind.

Eine derartige Statistik kann dem Benutzer dabei helfen, vorliegenden Quellcode zu verstehen, indem sich Rückschlüsse von der Häufigkeit bestimmter Konzepte auf die Funktionalität schließen lassen. Die Ergebnisse können darüber hinaus von anderen Programmen weiterverwendet werden und so bei weiteren Analysen helfen.

Damit die Erfassung einer solchen Statistik möglichst unabhängig vom Programm bleibt, soll sie auf Basis der Zwischensprache „Bauhaus IML“ (folgend auch nur „IML“ genannt) durchgeführt werden. Bauhaus ist ein Framework, welches diverse Mittel zur Codeanalyse bereitstellt. Es enthält dabei, neben einer Vielzahl an Analyseprogrammen, mehrere Front-Ends für verschiedene Sprachen—darunter C/C++—mittels derer sich zu analysierende Quelltexte in die Zwischendarstellung IML überführen lassen.

1.2 Aufbau des Dokuments

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 1 — Einleitung: Das erste Kapitel stellt zunächst die Aufgabenstellung sowie die Struktur des Dokumentes vor.

Kapitel 2 — Grundlagen: Dieses Kapitel beschäftigt sich mit ein paar Grundlagen, wie der Zwischensprache Bauhaus IML sowie der Definition einiger Begriffe.

Kapitel 3 – Entwurf und Realisierung: In diesem Kapitel werden die konkreten Anforderungen vorgestellt, die an das zu entwickelnde Programm gestellt sind. Anschließend werden mögliche Lösungswege evaluiert, die letztlich gewählte Lösung sowie deren Konzeption präsentiert, um abschließend auf die Realisierung einzugehen.

Kapitel 4 — Implementierung: Details der Implementierung sind Gegenstand dieses Kapitels. Es erläutert die konkrete Umsetzung einiger Teile des Programms.

Kapitel 5 — Test und Validierung: Hier geht es um durchgeführte Tests um die Qualität und Funktionalität des Programms zu gewährleisten.

Kapitel 6 — Zusammenfassung Abschließend fasst dieses Kapitel die Ergebnisse der Arbeit bewertend zusammen und stellt Möglichkeiten zur Anknüpfung vor.

2 Grundlagen

In diesem Kapitel werden diverse Begriffe geklärt. Nach einem kurzen Überblick über die Zwischensprache, auf Basis der die Erstellung der Statistik zu vollziehen ist, werden zwei zentrale Begriffe der Aufgabenstellung definiert, nämlich *programmiersprachliches Konzept* und *Bibliothekskonzept*.

2.1 Bauhaus IML

Die Bauhaus IML Zwischensprache bildet die Basis für eine Vielzahl der verfügbaren Analyseverfahren im Bauhaus Framework. Quelltext, durch eines der verfügbaren Front-Ends in einen sogenannten „IML Graphen“ übersetzt, wird nun durch die zahlreichen Arten an Knoten der Zwischensprache dargestellt. Die Knoten der IML gehören jeweils einer Klasse an, welche diese definiert. Diese Klassen sind in einer Klassenhierarchie angeordnet, wobei eine Knotenklasse, wie aus dem Paradigma der Objektorientierung bekannt, Eigenschaften der Elternklasse erbt. Es existiert keine Mehrfachvererbung, dafür können Interfaces implementiert werden. Die Wurzel des Klassenhierarchiebaumes bildet die Knotenklasse `IML_Root` und besitzt selbst entsprechend keine Elternklasse. Eine Knotenklasse kann verschiedene Attribute enthalten, die auch an die jeweiligen Kinder vererbt werden und die Knotenklasse charakterisieren. Der Typ eines solchen Attributs kann einer von vielen sogenannten *builtin* Typen sein—beispielsweise Integer, String, oder Identifier—aber auch eine Knotenklasse selbst, was dann als Kante im Graph interpretiert wird. Kanten werden des Weiteren als „semantisch“ oder „syntaktisch“ klassifiziert. Dabei führen semantische Kanten beispielsweise zu Typinformationen oder Definitionen, während die besuchten Knoten beim Traversieren der syntaktischen Kanten des Graphen einen ausgebauten, abstrakten Syntaxbaum ergeben.

IML Klassen können abstrakt sein und damit nicht instanziierbar, um eine einheitliche Vorlage für diverse abgeleitete Klassen zu bilden. Als Beispiel sei hier die abstrakte Klasse `Throw_Statement` mit ihren beiden Spezialisierungen `Cpp_Throw_Statement` und `Java_Throw_Statement` genannt. Dies ermöglicht Analysen und sonstigen Verfahren nah an der ursprünglichen Quellsprache zu arbeiten, gibt aber gleichzeitig die Möglichkeit, auf einer abstrakteren Ebene zu bleiben, wenn einem die Information über ein `Throw_Statement` genügt und weitere Spezialisierungen nicht notwendig sind, was die Kompatibilität von Programmen bei künftigen Erweiterungen der IML verbessert.

Als zwei weitere Beispiele für IML Klassen seien die beiden abstrakten Klassen `O_Node` und `T_Node` genannt. Beide sind in der Regel mit semantischen Kanten an Knoten angehängt und bieten zusätzliche Informationen über diese. Dabei beherbergen Knoten der Klasse `O_Node` Definitionen bezüglich des Knotens, an den sie gebunden sind, während Knoten der Klasse `T_Node` Typinformationen repräsentieren. Dies lässt sich exemplarisch an der Klasse `Routine` sehen, welche eine Routine im Quelltext darstellt. Ein Attribut dieser Klasse nennt sich „Definition“ und stellt eine semantische Kante

zu einem Knoten der Klasse `O_Routine` dar, welcher Informationen über Parameter der Routine enthält. Eine dieser Informationen ist das Attribut „Return_Type“, das wiederum eine semantische Kante zu einem Knoten der Klasse `T_Node` ist, welche die Information über den Typen des Rückgabewertes der Routine darstellt.

Die Knoten, die in einem IML Graphen enthalten sind, besitzen eine eindeutige Identifikationsnummer, nachfolgend ID genannt. Ein IML Graph selbst ist eine Binärdatei, auf die mittels einer in Ada verfassten Bibliothek zugegriffen werden kann. Diese Bibliothek ist zu Teilen automatisch aus der Spezifikation der IML generiert.

2.2 Terminologie

Nachfolgend seien die zentralen Begriffe definiert, um deren Erfassung sich diese Arbeit widmet.

Programmiersprachliche Konzepte

Bei den programmiersprachlichen Konzepten, folgend auch nur Konzepte genannt, handelt es sich um die Konstrukte, die die jeweilige Programmiersprache zur Verfügung stellt, um Funktionen, Methoden, Typen und Objekte. Damit eine Implementierung nicht vollständig abhängig von der Programmiersprache des zu analysierenden Quelltextes ist, dient die IML als Abstraktion, wobei eine jede IML Klasse als programmiersprachliches Konzept zählt. Durch ihre Nähe zur Quellsprache bleiben Details erhalten, während aber gleichzeitig eine Kompatibilität mit anderen Programmiersprachen sowie ein simpler Ausbau zur vollständigen Unterstützung weiterer Sprachen gewährleistet wird. Folgend werden die Worte *IML Klasse* und *Konzept* gleichbedeutend verwendet.

Bibliothekskonzepte

Da ein Programm selten nur die unmittelbar von der Sprache genutzten Konstrukte einsetzt, sind genutzte Bibliotheken in den meisten Programmen allgegenwärtig. Hier erweitern sie die Sprache, indem sie dem Programmierer eine Funktionalität bereitstellen, die so zuvor noch nicht vorhanden war. Aus diesem Grund ist es wichtig, auch Bibliotheken in eine derartige Statistik einzubinden. Ein Bibliothekskonzept sei definiert durch die von der Bibliothek bereitgestellten Funktionen, Typen und Objekte, die ein Programmierer nutzen kann.

3 Entwurf und Realisierung

Dieses Kapitel umfasst, neben festgestellten Vorgaben wie den Anforderungen, auch die Findung und Evaluation des gewählten Lösungsweges. Daran angeschlossen finden sich konkrete Informationen zu der Realisierung der Implementierung des Programms `iml_concept_stats`.

3.1 Anforderungen

Erfassung der Verwendungshäufigkeit aller programmiersprachlichen Konzepte

Das zu implementierende Programm ist dazu in der Lage, die Verwendungshäufigkeiten der in einem Programm genutzten Konzepte zu ermitteln. Die *Verwendungshäufigkeit* bezeichnet hier die Zählung der Nutzung des jeweiligen Konzeptes. Zu beachten ist hierbei, dass das Auffinden eines Konzeptes nicht nur dessen Häufigkeit erhöht, sondern auch die seiner Eltern in der IML Klassenhierarchie.

Kommen in einer Funktion drei `While` Schleifen vor, stellt sich dies im IML Graphen des Quelltextes in Form von drei `While_Loop` Knoten dar, was für diese Funktion, unter dem Gesichtspunkt des Konzeptes `While_Loop`, einen Wert von 3 ergibt. Darüber hinaus zählt dies aber auch zur Häufigkeit der Elternklasse des `While_Loop`s, nämlich dem abstrakten `Loop_Statement`, seines Zeichens Basisklasse für mehrere verschiedene Spezialisierungen von verschiedenen Schleifenarten, siehe Abbildung 3.1.

Erfassung der Verwendungshäufigkeiten von genutzten Bibliothekskonzepten

Auch Bibliothekskonzepte sollen auf diese Art und Weise erfasst werden, da ihr Einsatz und Vorhandensein in vielen Programmen eine Unerlässlichkeit darstellt und eine Statistik ohne sie nicht vollständig wäre. Die Menge der Bibliothekskonzepte wird dabei vom Programm automatisch ermittelt.

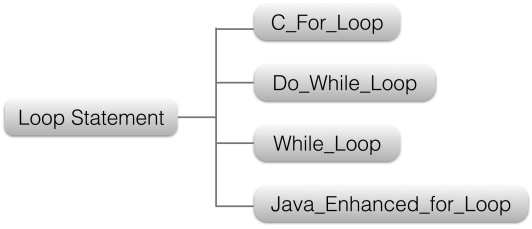

Abbildung 3.1: Ausschnitt aus der IML Klassenhierarchie.

Korrektes Erfassen auch von über Zeiger genutzte Konzepte

Durch die Verwendung von in Bauhaus verfügbaren Zeigeranalyse Verfahren soll auch die Nutzung von Funktionen und Objekten eingebundener Bibliotheken über Zeiger erkannt, sowie deren Nutzung dem eigentlichen Konzept, auf das gezeigt wird, zugerechnet werden. Bauhaus enthält mehrere Zeigeranalyse Verfahren, davon sind die Ergebnisse der „Andersen Analyse“, „Das Analyse“ und „ECR Analyse“ einheitlich durch die Bibliothek `PTA_Query` zugänglich. Entsprechend soll auch diese Bibliothek genutzt werden, womit Kompatibilität zu Ergebnissen der genannten drei Verfahren besteht. Damit ist auch zukünftig eine Unterstützung weiterer Verfahren durch nur geringfügige Anpassungen möglich, sollte die Bibliothek erweitert werden.

Dem Benutzer wird die Möglichkeit gegeben, bei einer Analyse auf einem IML Graphen, auf dem eine der drei Zeigeranalyse Verfahren angewendet wurde, zu wählen, ob er diese Ergebnisse in der zu erstellenden Statistik berücksichtigen will.

Visuelle Darstellung in wählbarer Granularität

Die Darstellung der Ergebnisse sei in frei wählbarer Granularität möglich. Das bedeutet, das Programm, über das die Statistik erstellt wird, ist als der Baum dargestellt, der entsteht, wenn man den syntaktischen Kanten im IML Graphen folgt—beginnend bei der Wurzel. Dieser kann dann an jedem Knoten entweder aus- oder eingeklappt werden, bis hinunter zu den Blättern des Baumes.

Während dieser Baum sich über die Vertikale erstreckt, bezieht sich die wählbare Granularität aber auch auf die Horizontale. In einem weiteren Baum, der sich über die Horizontale erstreckt, werden die Konzepte in Form der IML Klassen abgebildet, welche gemäß der IML Klassenhierarchie ebenfalls aus- und einklappbar sind.

Diese Form der Darstellung des zu untersuchenden Programms erzeugt noch eine weitere Statistik auf einer Metaebene. Neben der Anzahl der allgemeinen Verwendung eines Konzepts ist hier noch eine andere Nutzung sichtbar, wie die Menge an definierten Funktionen der verschiedenen Übersetzungseinheiten, welche man der Baumdarstellung des Programms selbst entnehmen kann.

Spätere Ergebnisbetrachtung ohne erneute Analyse

Eine Anzeige der ermittelten Verwendungshäufigkeiten ist jederzeit ohne einen weiteren Analysevorgang möglich. Dafür kann eine erzeugte Statistik in Form einer CSV (*Comma-separated values*) Datei exportiert werden. Dies ermöglicht darüber hinaus die ermittelten Daten durch andere Programme, wie beispielsweise eine Tabellenkalkulation, weiter zu verarbeiten.

3.2 Evaluation: eigenes Dateiformat

Der zu Beginn angedachte Weg zur Erfüllung der Aufgabenstellung sah zwei getrennte Teile vor. Zunächst würde in einem Programm der zu analysierende IML Graph geladen, die Werte erfasst

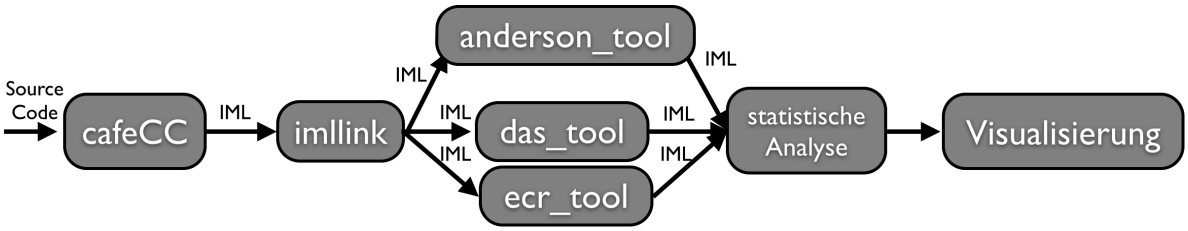

Abbildung 3.2: Ursprünglich angedachte Toolchain.

und in eine separate Datei gespeichert werden. Die Aufgabe eines zweiten Programms wäre dann ausschließlich die Darstellung dieser Datei. In diesem Szenario sähe eine mögliche Toolchain wie in Abbildung 3.2 aus. Hier wird Quellcode mittels eines der in Bauhaus verfügbaren Front-Ends—in der Abbildung das C/C++ Front-End `cafeCC`—in IML überführt, anschließend gegebenenfalls mit dem Linker für IML Dateien `imllink` zu einem anderen Graphen gebunden. Der Benutzer hat nun die Möglichkeit den IML Graphen mit weiteren Informationen anzureichern, wie durch das Ausführen einer der vorhandenen Zeigeranalyse Verfahren. Irgendwann kommt der Graph am Programm zur statistischen Analyse an, welches seine Ergebnisse in eine separate Datei schreibt, welche wiederum von einem Visualisierungsprogramm betrachtet werden kann.

Ein derartiges Verfahren hat mehrere Vorteile. Zum einen wäre für die Darstellung der Ergebnisse praktisch kein Rechenaufwand mehr notwendig, zumindest unter der Prämisse, die zu lesende Datei enthalte vorbereitet alle notwendigen Informationen. Des Weiteren ist das Programm zur Darstellung nicht mehr an das IML Format gebunden, was der Implementierung deutlich mehr Freiraum lässt. Zwar existieren für die in Ada geschriebenen IML Bibliotheken Sprachanbindungen für sowohl C++ als auch Java, allerdings sind diese nicht vollständig und besitzen nicht den gleichen Umfang wie die IML Bibliothek in Ada. Zudem ist ein eigenes Dateiformat von all den vielen zusätzlichen Informationen eines IML Graphen befreit, die für die Statistik irrelevant sind.

Die wohl schwierigste Anforderung, derer ein eigenes Dateiformat gerecht werden muss, ist die beliebige Granularität der Darstellung. Sprechen wir von Granularität bei einem IML Graphen, liegt die feinste Untergliederung, die man schaffen kann, auf Basis der einzelnen Knoten des Graphen, beziehungsweise der Blätter im Baum, aufgespannt durch die syntaktischen Kanten. Dies zwingt einen im Prinzip dazu, alle Informationen eines jeden Knotens vorzuhalten, egal ob dieser später überhaupt betrachtet wird, oder nicht. Ein naiver Ansatz speichert eine große Tabelle, bestehend aus den Knoten als Reihen und den Konzepten als Spalten, mit einem Zelleninhalt, der die ermittelte Häufigkeit enthält. Um die dabei anfallende Datenmenge grob zu überschlagen, gehen wir von einem IML Graphen mit 745447 syntaktischen Knoten aus. Diese Zahl entspricht dem IML Graphen des quelloffenen Raytracers „POV-Ray“ in Version 3.6.1. Das Speichern von einer Zahl pro Knoten, für jede der aktuell 306 IML Klassen, entspricht bei 32bit Ganzzahlen einem Speicheraufwand von grob 1.8GB Daten—nur für die Zahlen selbst. Wählt man anstelle eines Binärformats zur Speicherung ein textbasiertes Format, beispielsweise XML, erhöht sich diese Zahl nochmals—je nach Format—unter Umständen sehr stark. Das wirkt besonders im Hinblick auf den lediglich 36MB großen IML Graphen von „POV-Ray“ indiskutabel groß.

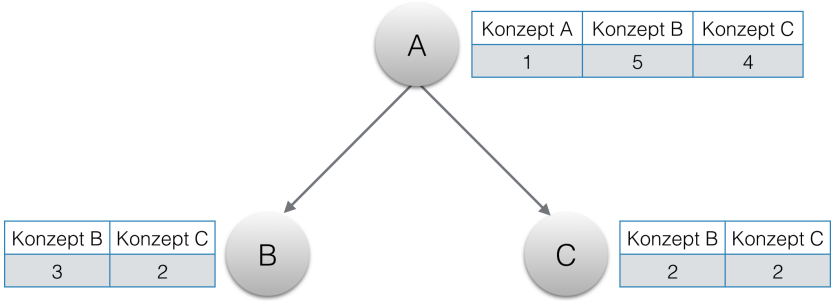

Abbildung 3.3: Implizit vorhandene Informationen bei einem Speichern der Statistik pro Knoten.

3.3 Evaluation: Daten im IML Graphen speichern

Ein weiter Ansatz ist hybrider Natur. Unter Verwendung des bestehenden IML Graphen werden durch das Programm, welches die statistische Analyse durchführt, einem jedem Knoten entsprechende Zahlen als zusätzliche Information hinzugefügt. Damit entfällt nicht nur eine eigene Dateistruktur, sondern auch eine Zuordnung der Daten zu einem Knoten, da dies in einem IML Graphen uns bereits implizit gegeben ist. Doch auch in diesem Ansatz stellt das Speichern von Zahlen für jedes Konzept und für jeden Knoten eine immer noch immense Datenmenge dar. Eine offensichtliche Verbesserung wäre es, anstelle einer vollständigen Tabelle lediglich Zahlen für die Konzepte zu speichern, die in den Knoten unterhalb eines Knotens überhaupt vorkommen. Dies spart eine große Menge an zu speichernden Daten, vor allem für Blattknoten, die nur das Konzept speichern müssten, für das sie selbst stehen.

Bei dieser Überlegung offenbart sich aber auch ein Problem dieses Vorgehens, denn hier werden viele Daten gespeichert, die so schon implizit im Graphen vorhanden sind. Dabei ist das nicht nur der Fall bei Blättern, die lediglich für das Konzept, welches sie ohnehin repräsentieren, einen extra Datensatz anlegen würden, sondern auch für jegliche Vater-Kinder Beziehung im Baum, wie in Abbildung 3.3 sichtbar. Es lassen sich alle Daten, die Knoten A speichert, vollständig durch die Summation der Daten der Kinder (B und C) und dem Konzept, das A selbst repräsentiert, ausdrücken.

Will man das Speichern derartiger Daten vermeiden, führt einen die konsequente Auflösung des Gedankens—der Erhalt der Daten durch Summation der Kindesdaten—unweigerlich dazu, gar keine Daten mehr zu speichern und die Statistik eines Knotens direkt daraus zu errechnen, alle im Baum unter ihm vorhandenen Knoten, beziehungsweise deren Konzepte, bei Bedarf aufzusummieren.

3.4 Konzeption

Die endgültige Konzeption der Implementierung sah nun nur noch ein Programm vor. Da die Speicherung vorbereiteter Daten entfällt, operiert das darstellende Programm nun direkt auf dem IML Graphen, erzeugt seine Ansicht des zu analysierenden Programms und berechnet die entsprechende Statistik ebenfalls selbst.

Damit entfällt zum einen die Erzeugung einer großen, separaten Datei, um die Ergebnisse zur Anzeige zu bringen, zum anderen besteht auch nicht mehr die Notwendigkeit Daten in den IML Graphen zu speichern, die in diesem unlängst implizit vorhanden sind. Da das Programm zur Darstellung nun direkt mit dem IML Graphen arbeitet, erhöht dies auch die mögliche Flexibilität des Programms und verringert den Aufwand künftiger Erweiterungen beträchtlich. Vor allem Änderungen, welche zusätzliche Daten zur Anzeige bringen sollen, hätten ein Abändern des Dateiformates—Ergänzung um die zusätzlichen Daten—und damit ein Ändern des Analyse- sowie Anzeigeprogramms bedingt.

Durch diese Entscheidungen fällt die Wahl der Programmiersprache praktisch direkt auf Ada 2005. Für die grafische Benutzeroberfläche fällt die Wahl auf GtkAda in der vorhandenen Version 2.14. Beide zusammen ermöglichen zudem beispielsweise eine künftige Integration der Implementierung in den IML Codebrowser `cobra`.

Um die grafische Darstellung zu realisieren soll ein sogenannter „Tree View“, ein häufiges Element in verschiedensten Bibliotheken für grafische Benutzeroberflächen, darunter auch GtkAda, genutzt werden. Es vereint eine Baumdarstellung eines Datensatzes in Reihen, welche aufgeklappt werden können, mit einer Tabelle, durch die Möglichkeit, beliebig viele Spalten seitlich der Baumdarstellung anzubringen. Während sich in diesem Oberflächenelement das zu analysierende Programm als Baum dargestellt befindet, wobei einem jeden Knoten dieses Baumes eine Zeile im „Tree View“ Element zugeordnet wird, befinden sich die Konzepte in den Spalten dieses Elements, wodurch sich eine tabellarische Ansicht ergibt.

Die Darstellung in einem Tree View erfüllt mehrere ästhetische Kriterien [BBD09] für die Visualisierung von Graphen, soweit sich diese anwenden lassen. Darunter fällt unter anderem die visuelle Unordnung, die in einem Tree View auch bei einem dichten IML Graphen gering gehalten wird, da diese Form der Darstellung ohne das Zeichnen von Kanten selbst auskommt und sich Dichte nur als einen aufwendigeren Bildlauf niederschlägt. Die Navigation im Tree View, also der Bildlauf sowie das Aus- beziehungsweise Zuklappen von Reihen, verändert nicht die Darstellung der übrigen Elemente, die nicht Teil der Interaktion sind, wodurch die mentale Navigationskarte, die sich ein Benutzer bildet, aufrecht erhalten wird, was wiederum ein weiteres Kriterium erfüllt. Zu guter Letzt besitzt die Darstellung in einem Tree View auch gute Eigenschaften bezüglich der Skalierung mit vielen Knoten sowie Kanten—ebenfalls einem Kriterium entsprechend—da die dargestellten Daten nie allesamt gleichzeitig sichtbar sind, sondern nur die, welche der Benutzer selbst ausgeklappt hat. Auch wenn der Benutzer sehr viel Reihen ausklappt, ist es einzig der steigende Aufwand für den Bildlauf der Darstellung, welcher negativ auf die Bedienbarkeit wirkt. In einer vergleichenden Evaluation mehrerer Visualisierungsmethoden für Bäume [SKLS10] wurden Verbesserungen an einer naiven Darstellung, welche für jeden Knoten simpel eine Kante mit einem jeden Kind zeichnet, hinsichtlich der Benutzernavigation, evaluiert. Zwar entspricht der Tree View nicht exakt dieser Visualisierung, dennoch teilt er vollzogene Erweiterungen der naiven Darstellung, welche in der Evaluation als spürbare Verbesserungen für die Navigation in den Daten hervorgingen, darunter der Verzicht auf das Zeichnen von Kanten. Das beste Ergebnis der Evaluation erzielte jedoch eine Verbesserung, die den notwendigen Bildlauf wesentlich reduzierte, indem Kinder ab einer gewissen Menge auf mehrere Spalten aufgeteilt und nebeneinander dargestellt wurden. Die Adaption dieser Technik ist in diesem Fall leider nicht möglich, da es für den Zweck der hier umzusetzenden Arbeit eine eindeutige Beziehung zwischen einer Reihe und einem Teil des darzustellenden Programms—einem IML Knoten—geben muss.

Um den IML Graphen zu traversieren wird das Entwurfsmuster des „Besuchers“ genutzt, das in der Bauhaus Bibliothek in Form einer abstrakten Klasse `Visitor` vorhanden ist. Dieses erlaubt es, den gesamten syntaktischen Teil des IML Graphen zu traversieren, wobei für jeden Knoten, der besucht wird, eine Prozedur aufgerufen wird, die der Klasse des besuchten Knoten entspricht. Das Muster wird nicht nur für den Aufbau der Darstellung des zu analysierenden Programms, sondern auch für die Erfassung der Statistik selbst genutzt.

Für die Möglichkeit, einmal erzeugte Ergebnisse auch später ohne erneute Analyse wieder ansehen zu können, sorgt eine Funktion zum Export der Daten in ein strukturiertes Format, welches von einer breiten Masse an vorhandenen Programmen verstanden wird. Hier fiel die Wahl auf ein simples CSV (*Comma-separated values*) Format, womit eine große Kompatibilität zu allen Arten von Datenverarbeitungsprogrammen erreicht wird.

3.5 Visualisierung großer Datenmengen

Nachdem mit GtkAda die zu nutzende Benutzeroberflächenbibliothek fest stand, galt es herauszufinden, wie das Tree View Element diese Bibliothek mit einer großen Datenmenge zurecht kommt. Genauer ging es um die Frage, ob ein simpler Ansatz, der den zu visualisierenden Graphen sofort vollständig anzeigt, möglich ist.

Zu diesem Zweck wurden Prototypen gebastelt, welche den Baum der syntaktischen Knoten eines IML Graphen in einem GtkAda Tree View darstellten. Als Eingaben dienten zunächst IML Graphen einer Sammlung an Testprojekten meines Betreuers. Mit einer Anzahl von 220724 SLOC (Source Lines of Code, also die Anzahl der Zeilen Code ohne Kommentare und Leerzeichen) und einem IML Graphen mit insgesamt 1088638 Knoten, davon 745447 über syntaktische Kanten erreichbar, ist der zuvor genannte Raytracer „POV-Ray“ das größte Projekt dieser Sammlung.

Ein Graph dieser Dimension stellte für einen naiven Prototypen, der den ganzen IML Graphen auf einmal in die Anzeige einfügt und vollständig visualisiert, kein Problem dar. Die Darstellung ist in einer akzeptablen Zeit von unter drei Minuten erstellt und kann ab dann flüssig und ohne weitere Wartezeiten oder Verzögerungen betrachtet werden. Das gilt auch für Interaktionen mit der Darstellung, also dem Auf- sowie Zuklappen von Reihen im Tree View.

Um in größeren Dimensionen testen zu können wurde ein Skript entwickelt, das als Eingabe eine gewünschte Anzahl SLOC entgegen nimmt, entsprechend dieser Angabe eine Datei mit Quellcode der Sprache C erstellt und diese anschließend zu einem IML Graphen kompiliert. Bei dem generierten Code handelt es sich um die Wiederholung einer Funktion nach dem immer gleichen Schema, deren Name neben einer fortlaufenden Nummer auch eine Zufallszahl enthält. Der Funktionskörper enthält wenige Zeilen an mathematischen Berechnungen, bevor ein Wert zurückgegeben wird, wie in Listing 3.1 zu sehen. In der Hauptprozedur der Datei werden darüber hinaus alle erstellten Funktionen aufgerufen. Die zusätzliche, zufällige Nummer im Namen der Funktion ermöglicht es später derartig erstellte IML Graphen mit `imllink` zu größeren Graphen zu binden, ohne allzu viel mehrfach vorhandene Funktionen zu haben. Da das Front-End `cafeCC` bei einer Datei ab ungefähr 1400000 SLOC den Übersetzungsvorgang mit einem Fehler quittierend vorzeitig einstellt, wurden größere Graphen entsprechend durch das Binden mehrerer vorhandener Graphen mit `imllink` bewerkstelligt, bevor

Listing 3.1 Schablone der generierten Funktionen. p ist eine fortlaufende Nummer, q eine Zufallszahl.

```
int func_p_q(int x)
{
  int y = x << p;
  int z = x + p;
  float a = (float) x;
  a /= z;
  a *= y*y*y;
  return (int) a;
}
```

irgendwann auch dieses Programm mit einem Speicherfehler seinen Dienst einstellte. Der größte damit erstellte IML Graph besitzt 19650362 Knoten bei einer Dateigröße von 722MB.

Bei Testläufen mit zunehmend größeren IML Graphen als „POV-Ray“, ergibt sich ein erwartetes Bild. Die Zeit zum Erstellen der Darstellung wächst linear und ist anschließend ohne Probleme flüssig nutzbar. Zwar lässt sich in der Tat eine zunehmende Trägheit bezüglich der Interaktion mit der Oberfläche, also eine verzögerte Antwort, feststellen, allerdings befindet sich diese selbst beim größten getesteten Graphen auf einem akzeptablen Niveau.

Großes Augenmerk liegt jedoch auch auf dem Speicherverbrauch des Programms, welcher nicht nur aus dem geladenen IML Graphen besteht, sondern auch aus den Daten, die im Tree View angezeigt werden. Dieser Speicherverbrauch ließe sich deutlich reduzieren, wenn anstelle von allen Knoten jeweils nur diejenigen in der Benutzeroberfläche dargestellt werden, die auch tatsächlich angesehen werden, also im Tree View ausgeklappt sind. Knoten, welche nicht sichtbar sind, weil eingeklappt, wären dann nicht schon im Vorfeld geladen, verbrauchen keinen Speicherplatz und reduzieren auch die Menge an Zeilen, die der Tree View im Hintergrund verwalten muss, wodurch sich die Antwortzeit des Programms auf Interaktionen mit der Oberfläche verbessert. Letzteres wirkt sich dabei zusätzlich förderlich auf den Erhalt der ästhetischen Kriterien bezüglich der Skalierung mit der Anzahl Knoten und Kanten aus.

Aufgrund dessen wurde die Darstellung des IML Graphen so gewählt, dass ein geladener Graph nicht vollständig auf einmal in die Benutzeroberfläche eingefügt wird, sondern notwendige Teile dynamisch nachgeladen werden, wenn der Benutzer diese betrachten will. Details zur Implementierung finden sich in Kapitel 4.1.

3.6 Architektur

Die Architektur der Implementierung lässt sich grob in mehrere logische Blöcke unterteilen, die zum einen aus dem Teil bestehen, der für die grafische Benutzeroberfläche zuständig ist und zum anderen aus den abgeleiteten `Visitor` Klassen, wie in Abbildung 3.4 zu sehen.

Der Programmstart findet zunächst in der Hauptprozedur `Iml_Concept_Stats` statt. An dieser Stelle werden sowohl Kommandozeilenparameter interpretiert als auch der spezifizierte IML Graph ge-

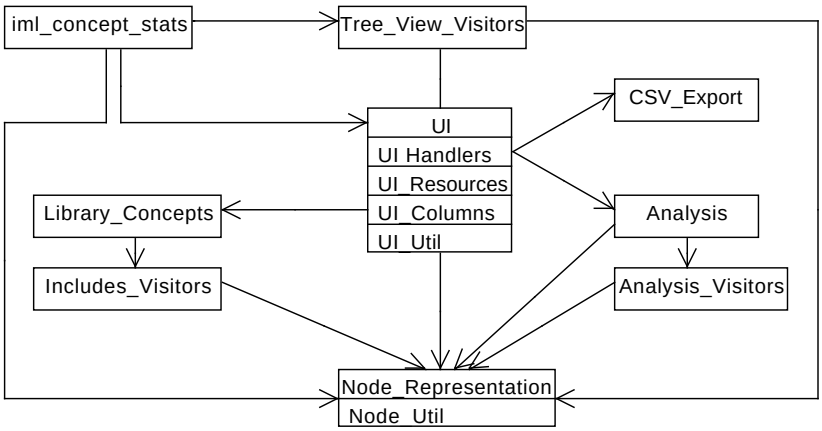

Abbildung 3.4: Skizzierte Architektur der Implementierung `Iml_Concept_Stats`.

laden, um anschließend die Benutzeroberfläche zu initialisieren, welche ab hier den Kontrollfluss übernimmt.

Die gesamte Kontrolle bezüglich der Benutzeroberfläche findet sich in mehreren Teilen partitioniert vor. Das Paket `UI` bildet die Schnittstelle zum Erstellen des Hauptfensters sowie anderer Oberflächenteile und nutzt selbst wiederum die anderen `UI_*` Pakete um dies zu bewerkstelligen, wie in Abbildung 3.5 dargestellt. `UI` kümmert sich um die Instantiierung der notwendigen Oberflächenteile, erstellt und macht sie sichtbar, mit einer Ausnahme: das Erstellen und Verwalten der Konzepte in der Benutzeroberfläche ist an `UI_Columns` übergeben. In `UI_Handlers` sind alle in der Oberfläche registrierten „Handler“ gesammelt. Diese Funktionen werden mit der GtkAda Umgebung registriert und bei Eintreten des Ereignisses, für das man sie registriert hat, von GtkAda aufgerufen. Damit ist dieses Paket für die gesamte Nutzerinteraktion zuständig. Typen und globale Instanzen, welche über die Benutzeroberfläche hinweg verwendet werden, sind gemeinsam in `UI_Resources` gruppiert verfügbar. `UI_Util` bietet Zugriffsmöglichkeiten auf die Benutzeroberfläche, in erster Linie um Daten aus der Ansicht auszulesen sowie einzufügen.

Das Paket `Tree_View_Visitors` bietet die erste von drei Visitor-Klassen. Er dient zum Aufbau der Ansicht des als IML Graph eingelesenen Programms. `Includes_Visitors` bietet einen Visitor zum extrahieren der Bibliothekskonzepte aus dem Graphen. Ein Objekt dieser Klasse wird von `Library_Concepts` verwendet um die aufbereiteten Ergebnisse der Benutzeroberfläche zugänglich zu machen. Mit der Durchführung der Analyse verhält es sich ähnlich. Diese wird über das Paket `Analysis` gesteuert, welches wiederum eine Instanz des `Analysis_Visitor` nutzt.

Daneben gibt es das Paket `CSV_Export`, welches die Funktionalität bereitstellt, die Daten der Benutzeroberfläche zu serialisieren und Pakete, welche Hilfsfunktionalität bieten. Darunter sei genannt `Node_Representation`, welches textuelle Abbildungen für verschiedene Knotenklassen der IML bietet, sowie `Node_Util`, in dem Hilfsfunktionen rund um den IML Graphen gesammelt sind.

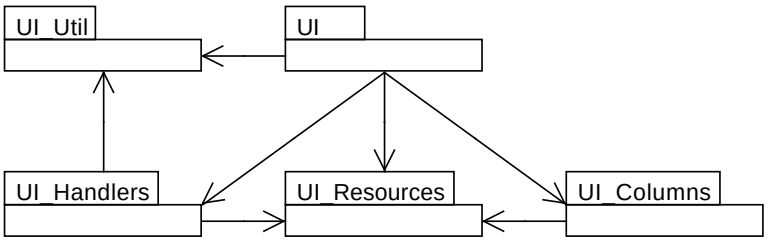

Abbildung 3.5: Detailansicht auf die Benutzeroberflächenkomponente.

3.7 Realisierung

Dieses Kapitel handelt von Details bezüglich der Umsetzung des Entwurfs sowie den realisierten Funktionen und deren Benutzung. Implementierungsspezifische Informationen finden sich im nachfolgenden Kapitel 4.

3.7.1 Durchführung

Die Entwicklung wurde mit dem „GNAT Programming Studio“ durchgeführt, entsprechend liegt die Implementierung in Form eines Projektes dieser Entwicklungsumgebung vor. Notwendige Abhängigkeiten zur Übersetzung des Projektes sind in diesem bereits voreingestellt, entsprechend der vorhandenen Umgebung auf dem Abteilungsrechner `pslx2`. Bei diesen Abhängigkeiten handelt es sich neben diverser IML Bibliotheken, um GtkAda. Die Konfiguration des Projektes geht davon aus, relativ zum Bauhaus Stammverzeichnis, in dem Verzeichnis `projects/tools/iml_concept_stats` zu liegen. Zur Organisation wurde eine simple Dateistruktur genutzt, wie nachfolgend dargestellt.

```
iml_concept_stats/
├── bin/
├── build/
│   └── ...
├── iml_concept_stats.gpr
├── src/
│   └── ...
└── test/
    ├── generate_arbitrarily_sized_graph/
    │   └── ...
    └── test_programms/
        └── ...
```

Bei der Entwicklung wurde darauf geachtet, den „Bauhaus Ada 95 Style Guide“ einzuhalten.

3.7.2 Programmargumente

Beim Start nimmt das Programm ein einziges, zwingendes Argument, nämlich den Pfad zu dem einzulesenden IML Graphen. Alle weiteren Argumente sind optional und nachfolgend aufgelistet.

`-decl-table`
Da die Darstellung des IML Graphen den syntaktischen Kanten folgt, ist der Inhalt der Deklarationstabellen von IML Knotenklassen, die eine solche enthalten, normal nicht in der Darstellung vorhanden. Mit diesem Argument wird in die Ansicht ein Unterpunkt für die entsprechenden IML Knoten eingefügt, der den Inhalt der Deklarationstabelle enthält, mit Ausnahme von `O_Routine` Knoten.

`-with-imported`
Dieses Argument wirkt sich auf das vorherige Argument aus. Sollte es spezifiziert sein, so werden in den angezeigten Deklarationstabellen auch importierte Deklarationen aufgenommen.

`-detach`
Bei Angabe von diesem Argument wird während der Erstellung der Statistik das Datenmodell von der Anzeige getrennt, was eine geringfügig schnellere Berechnung zur Folge hat.

`-short-concepts`
Die angezeigten Namen von Bibliothekskonzepten werden bei Angabe dieses Arguments gekürzt. Es hat keine Auswirkung auf die Namen der Konzepte der Bauhaus IML.

`-init-depth <number>`
Um auch sehr große IML Graphen flüssig anzeigen zu können, werden sie nicht vollständig in die Anzeige eingefügt. Dieses Argument spezifiziert die Tiefe, bis zu der ein neuer Wurzelknoten in der Anzeige eingefügt werden soll. Der Standardwert dieses Arguments liegt bei 4.

`-reload-depth <number>`
Soll ein Teil eines Graphen dargestellt werden, der bislang noch nicht in die Darstellung geladen wurde, erfolgt ein Nachladen der benötigten Teile. Dieses Argument spezifiziert die Tiefe, bis zu der eine Nachladeoperation Knoten hinzufügen soll. Der Standardwert dieses Arguments liegt bei 4.

`-seperator <param>`
Jeder IML Knoten, dessen Klasse in der Klassenhierarchie unter `IML_Root` vorkommt, oder von dieser Klasse ist, besitzt ein Attribut `SLoc`, welches den absoluten Pfad zu der Datei enthält, aus der der Knoten bei der Übersetzung gewonnen wurde. Dieses Argument gibt das Zeichen an, das in diesem Pfad zum Trennen von Verzeichnissen genutzt wird. Es ist hauptsächlich vom Betriebssystem abhängig, auf dem der IML Graph erzeugt wurde. Der Standardwert dieses Arguments ist '/'.

`-include-prefix <param>`
Nur Bibliothekskonzepte, die von einem Verzeichnis aus eingebunden wurden, welches mit dem hier spezifizierten Präfix beginnt, werden berücksichtigt. Der Standardwert dieses Arguments ist '/usr'.

`-dynamic-column-font-size <number>`
Hier kann die Schriftgröße, die für die Darstellung der Konzepte in der Benutzeroberfläche genutzt wird, angegeben werden. Der Standardwert dieses Arguments liegt bei 8.

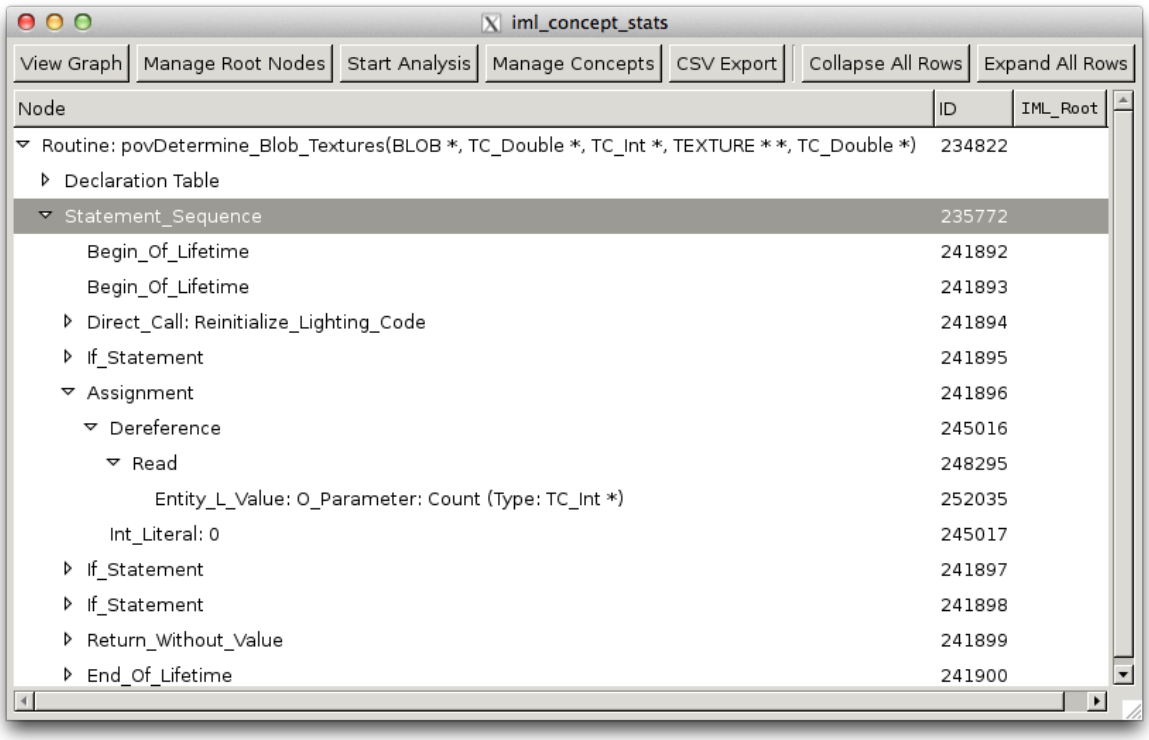

Abbildung 3.6: Benutzeroberfläche mit teilweise dargestelltem IML Graphen.

`-node <param>`
Dieses Argument ermöglicht einen Knoten durch seine ID zu spezifizieren, welcher dann direkt beim Start der Benutzeroberfläche in diese eingefügt wird. Das ist zwar auch über die Benutzeroberfläche selbst möglich, das Argument ermöglicht jedoch eine Automatisierung auf der Konsole. `<param>` ist genau eine IML Knoten ID. Das Argument kann mehrfach genutzt werden.

3.7.3 Funktionen

Die realisierte Benutzeroberfläche besteht aus zwei Teilen: einer Reihe an Schaltflächen und darunter liegend der Bereich der Darstellung des geladenen IML Graphen. Schaltflächen und Eingabefelder sind fast vollständig mit Hilfstexten in Form von „Tooltips“ unterlegt, um die Bedienung zu vereinfachen. Die gesamte Benutzeroberfläche inklusive einer dargestellten Routine lässt sich in Abbildung 3.6 betrachten.

Der Darstellungsbereich ist so gestaltet, dass sich in diesen beliebig viele, beliebige Teilbäume des IML Graphen einfügen lassen. Die Wurzel eines solchen Teilbaumes wird als sogenannter „Wurzelknoten“ in die Tree View Darstellung eingefügt, welcher sich mit dem Dialog hinter der Schaltfläche *Manage Root Nodes* verwalten lässt. In diesem Dialog, zu sehen in Abbildung 3.7, findet sich neben einer Liste

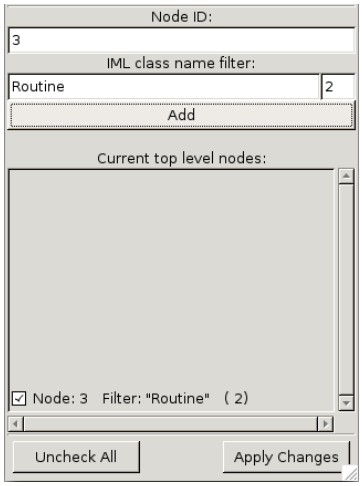

Abbildung 3.7: *Manage Root Nodes* Dialog mit ausgefüllten Eingabefeldern.

aller momentaner Wurzelknoten—wobei man sie durch Abwahl entfernen kann—die Möglichkeit, anhand der Knoten ID weitere Wurzelknoten hinzuzufügen.

Am oberen Ende des Dialogs befindet sich ein Eingabefeld, in dem die ID des Knotens anzugeben ist, der anschließend durch die Betätigung des *Add* Schalters in die Liste der Wurzelknoten aufgenommen werden soll. Zwei weitere Eingabefelder dieses Dialogs ermöglichen das massenhafte Einfügen von Wurzelknoten. Die beiden Felder unter *IML class name filter* sind standardmäßig leer und wirken sich damit nicht aus; in diesem Fall wird lediglich der spezifizierten Knoten in die Liste der Wurzelknoten aufgenommen. Diese Semantik ändert sich jedoch, sobald in das Feld der Name einer IML Klasse eingetragen wird und gegebenenfalls in das kleine Feld zur Rechten eine minimale Tiefe, wie in Abbildung 3.7 zu sehen. Nun wird ab dem spezifizierten IML Knoten—in der Abbildung der Knoten mit der ID 3—eine Breitensuche über die syntaktischen Kanten gestartet, die solange läuft, bis sie einen Knoten gefunden hat, dessen IML Klasse dem angegebenen Filter entspricht. Wird dieser Knoten gefunden, werden alle Knoten der selben Generation, welche ebenfalls dem Filter entsprechen, als Wurzelknoten hinzugefügt.

In Abbildung 3.8 wird illustriert, welche Knoten hinzugefügt werden. Das linke Beispiel der Abbildung entspricht einem imaginären Knoten 3, unter welchem mehrere `Routine` Knoten hängen. Mittels der Nutzung des Filters „Routine“ lassen sich diese alle auf einmal als Wurzelknoten in die Ansicht einfügen. Im rechten Beispiel dagegen sollen alle `Routine` Knoten aller `Unit` Knoten hinzugefügt werden. Dieses Vorhaben konfligiert allerdings zunächst mit dem `Routine` Knoten, der direkt unter dem Knoten 3 hängt und von der Breitensuche als erstes gefunden werden würde, wodurch die Ebene darunter nie exploriert wird. Durch die Eingabe einer Minimaltiefe von 2 lässt sich dieses Problem lösen. Nun werden alle gewünschten Knoten als Wurzelknoten der Ansicht hinzugefügt. Dieser nicht ganz triviale Vorgang ist nochmals in der Oberfläche des Dialogs und den Tooltips der einzelnen Felder erklärt.

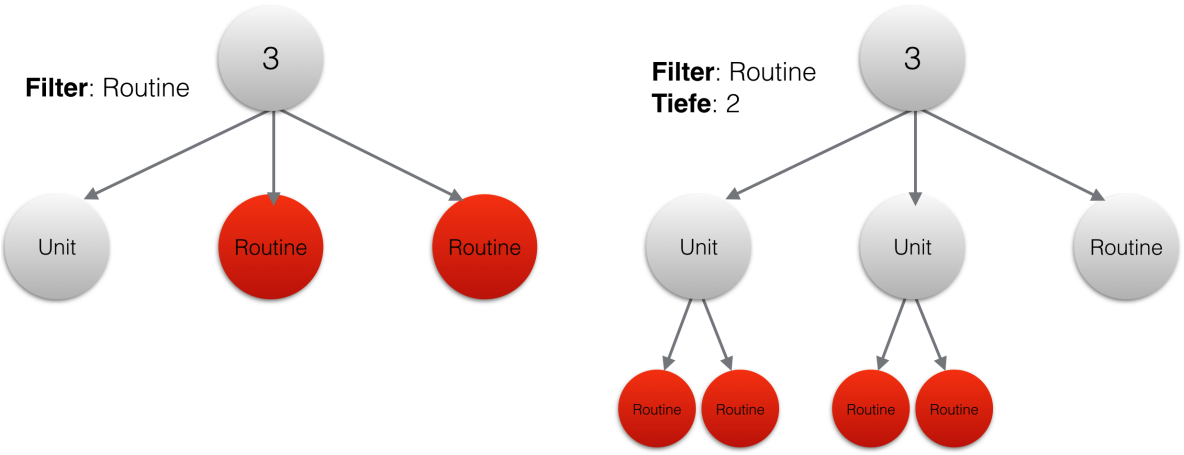

Abbildung 3.8: Veranschaulichung des Hinzufügens mehrerer Knoten durch den *Manage Root Nodes* Dialog. Rot markiert sind die hinzugefügten Knoten.

Die Schaltfläche *View Graph* ist eine Komfortfunktion, welche den Wurzelknoten des IML Graphen als Wurzelknoten in der Ansicht hinzufügt. Zwar wäre dies auch mit dem *Manage Root Nodes*-Dialog möglich, setzt aber die Knoten ID der Wurzel des Graphen als bekannt voraus.

Nach dem Betätigen der Schaltfläche *Start Analysis* wird der Benutzer gefragt, ob er Zeigeranalyse Informationen im IML Graphen bei der Erstellung der Statistik berücksichtigen will. Sollte dies affirmativ beantwortet werden, im Graphen werden aber keine derartigen Informationen vorgefunden, wird die Statistik schlicht ohne sie erstellt. Anschließend wird die Statistik erstellt und das Programm trägt die entsprechenden Werte in all diese Reihen der Darstellung ein, die beim Start der Analyse in der Benutzeroberfläche vorhanden waren.

Der Dialog, der sich hinter der Schaltfläche *Manage Concepts* verbirgt, ermöglicht es Konzepte für die Erstellung der Statistik auszusparen, folgend „deaktivieren“ genannt, indem in einer hierarchischen Darstellung die entsprechenden Konzepte entweder an- oder abgewählt werden können. Dabei ist zu beachten, dass zwischen Konzepten, welche in der horizontalen Baumansicht nicht sichtbar sind, weil das Elternkonzept nicht ausgeklappt wurde und Konzepten, die über den eben erwähnten Dialog deaktiviert wurden, unterschieden wird. Ein Konzept, welches lediglich nicht sichtbar ist, da eingeklappt, zählt nach wie vor als aktiviert und wird bei der zu erstellenden Statistik berücksichtigt, während ein Konzept, das explizit deaktiviert wurde, auch implizit unsichtbar ist, um die dargestellten Konzepte bei einer großen Anzahl übersichtlich zu halten. Aufgrund der Baumdarstellung birgt dies eine kleine Widrigkeit, da ein deaktiviertes Konzept, das damit auch unsichtbar ist, es natürlich unmöglich macht, dessen Kinder auszuklappen. Damit macht das Deaktivieren eines Konzeptes dessen Kinder implizit unsichtbar, deaktiviert sie jedoch nicht, um die Möglichkeit zu erhalten, beispielsweise nur ein Blattkonzept in der Klassenhierarchie der Konzepte für die Statistik zu berücksichtigen. In der Praxis stellt dies kein großes Problem dar, da man entweder die Eltern der zu berücksichtigenden Konzepte aktiviert lassen kann, oder sie einfach wieder aktivieren kann, nachdem die Statistik erstellt wurde.

CSV Export öffnet einen Dialog, der verschiedene Parameter zum Export der dargestellten Daten konfigurieren lässt. Darunter findet sich unter anderem die Möglichkeit, die vorhandenen Reihen im Tree View zu filtern, ebenso wie die Option eine Einrückung zu simulieren, welche der Baumdarstellung im Programm gleich kommt. Deaktivierte Konzepte werden bei diesem Export ausgespart, während die Entscheidung, unsichtbare Konzepte ebenfalls zu ignorieren, auch in diesem Dialog konfigurierbar ist.

Die beiden Schaltflächen *Collapse All Rows* und *Expand All Rows* sind Komfortfunktionen, um sämtliche Zeilen im Tree View entweder aus- oder einzuklappen. Anzumerken sei dabei die Eigenschaft von *Expand All Rows* auch tatsächlich alle Reihen für jeden Knoten in die Benutzeroberfläche einzufügen. Mehr zum dynamischen Einfügen von Reihen findet sich im Kapitel 4.1.

4 Implementierung

Dieses Kapitel widmet sich den Details der Implementierung. Für verschiedene Teile, welche sich für die Erfüllung der Anforderungen, oder der Nutzung allgemein, als kritisch dargestellt haben, wird nachfolgend die Art der Umsetzung vorgestellt.

4.1 Baumdarstellung des Programms

Um ein Programm, vorliegend als IML Graph, zu visualisieren, bedarf es im Groben zweier Tätigkeiten: das Traversieren des Graphen und das Anzeigen dieser Informationen. Während Ersteres vom Paket `Tree_View_Visitors` geleistet wird, ist für Letzteres die Prozedur `UI_Handlers.Row_Expand` zuständig. Das Anzeigen sollte, wie zuvor bereits eruiert, in einer Baumdarstellung mittels eines GTK Tree Views geschehen, während für die Traversierung das Entwurfsmuster des „Besuchers“ dient, welches Bauhaus mit der abstrakten Klasse `Visitor` anbietet und als Elternklasse für `Tree_View_Visitor` dient. Großes Augenmerk lag auf der Performanz der Implementierung, speziell was das Betrachten und Darstellen von sehr großen IML Graphen betrifft. Dabei stellte sich das Traversieren des Graphen in Tests mit Prototypen als absolut unkritisch heraus. Der Großteil der Zeit in den Testläufen ging eindeutig an das Erstellen der Darstellung, welches aus dem Hinzufügen von Reihen für jeden im Graphen besuchten Knoten besteht.

Um diesen Aspekt zu optimieren, musste zunächst strikt darauf geachtet werden, wie Reihen in den Tree View hinzugefügt werden. GtkAda erlaubt lediglich das Hinzufügen neuer Reihen am oberen Ende des Anzeigeelementes mit sub-quadratischer Laufzeitkomplexität [gtk], weshalb die Traversierungsreihenfolge des `Visitor` redefiniert wurde (siehe Listing 4.1), damit die angezeigten IML Knoten in der Reihenfolge vorliegen, mit der sie auch im Bauhaus Codebrowser `cobra` dargestellt werden und damit homogen sind. Die abgeänderte Klasse traversiert den Graphen rekursiv, besucht die syntaktischen Kinder eines Knotens von rechts nach links und fügt einen besuchten Knoten vor dem Traversieren seiner Kinder in die Ansicht ein.

Damit ist ein Grundstein für die effiziente Darstellung von IML Graphen gelegt, aber für eine passable Skalierbarkeit der Anzeige reicht dies noch nicht aus. Um die enorme Menge an Knoten sehr großer IML Graphen—und die daraus resultierende Menge an im Tree View darzustellender Reihen—langfristig bewältigen zu können, ist eine mögliche Lösung diese Knoten schlichtweg nie allesamt zu visualisieren. Es gilt zu visualisieren, was der Benutzer betrachten will, oder benötigt um zu navigieren, alles andere jedoch bringt uns keinen direkten Nutzen. Um dies zu realisieren wird der genutzte `Visitor` nochmals umgeändert, sodass der Knoten lediglich bis zu einer bestimmten Tiefe von seinem Startknoten aus traversiert (siehe Listing 4.1). Tiefe bezieht sich hier auf die Generation der Knoten im Graphen, die pro Pfad beim Besuch der Kinder um je eins zunimmt. Damit Benutzer dennoch den gesamten Graphen explorieren können, müssen Knoten gegebenenfalls nachgeladen

Listing 4.1 Umgekehrte Traversierungsreihenfolge im `Tree_View_Visitor`, außerdem eine Tiefenbeschränkung der Traversierung.

```
if Relative_Depth < Relative_Depth_Limit then
  for I in reverse Vector'Range loop
    IML_Roots.Internal_Accept_Visitor (Vector (I), Self);
  end loop;
end if;
```

werden. GtkAda erlaubt es, eine Funktion zu registrieren, die beim Ausklappen einer Reihe im Tree View aufgerufen wird. Diese Technik erlaubt das dynamische Nachladen von Knoten nach Bedarf.

4.1.1 Nachladen von Knoten

Neue Wurzelknoten der Ansicht werden nur noch bis zu einer bestimmten Tiefe überhaupt in die Ansicht geladen, weitere Knoten erst dann, wenn nötig. Da das Feststellen der Notwendigkeit des Nachladens bei der Aktion des Aufklappens einer Reihe geschieht und dies überhaupt nur möglich ist, wenn eine Reihe bereits Kinder hat, operiert die Logik stets eine Ebene unter der aktuellen Ansicht. Das bedeutet, beim Aufklappen einer Reihe im Tree View besitzt diese bereits Kinder, da GtkAda diese Operation—das Aufklappen—sonst gar nicht zulassen würde. Geprüft wird nun, ob diese Kinder bereits Kinder in Form von Reihen haben. Sollte dem nicht so sein, wird der `Visitor` ab diesem Knoten gestartet, sodass er deren Kinder einfügt, bis erneut die maximale Tiefe erreicht wurde. Finden wir dagegen bereits Kindeskinder vor, so gehen wir davon aus sie selbst in einer vorherigen Operation eingefügt zu haben.

Diese Annahme erlaubt es uns, bereits beim ersten gefundenen Kindeskind die Ausführung dieser Prozedur abzubrechen. Das wiederum ermöglicht eine schnelle Ausführung der Prozedur, selbst bei großen Graphen, ohne die Notwendigkeit die Reihen, für die bereits Kinder eingefügt wurden, in irgendeiner Weiße zu markieren oder zu speichern. Dies verhält sich umso besser, umso höher die Tiefe ist, bis zu der neue Knoten eingefügt werden—je höher, desto öfter treffen wir auf Reihen, die schon eingefügte Kinder haben.

Sollte jedoch kein Kindeskind vorgefunden werden, müssen neue Reihen eingefügt werden. Dafür wird von jedem Knoten aus, der ein Kind des Knotens ist, dessen Reihe soeben aufgeklappt wurde, der `Visitor` gestartet. Wie eingangs erwähnt, ist die Ausführungszeit des `Visitor` selbst trivial, anders jedoch, wenn sehr viele davon gestartet werden. Das war besonders bei den generierten IML Graphen aufgefallen, da sich hier unter einem einzigen `Unit` Knoten viele tausend `Procedure` Knoten befinden. Da zwar die Instanziierung von tausenden `Visitor` Objekten lange dauert, das Einfügen all dieser Knoten von einer Ebene höher aus—durch lediglich eine `Visitor` Instanz—jedoch in gewohnt kurzer Zeit absolviert ist, wurden kurzerhand höhere Werte für die einzufügende Tiefe des `Visitor` festgelegt. Damit werden diese Knoten bereits geladen, wenn der `System` Knoten eines IML Graphen geladen wird. Eine Ansammlung derartig vieler Knoten fand sich in Tests mit den bereits erwähnten Testprojekten unter `Unit` Knoten, aber sonst nicht in ausreichend bedrohlicher Anzahl, womit sich dieses Problem mit der angepassten Tiefe vollständig lösen lies. Nichtsdestotrotz lässt sich die Tiefe, bis zu der der `Visitor` Knoten einfügt, durch einen Programmschalter jederzeit verändern.

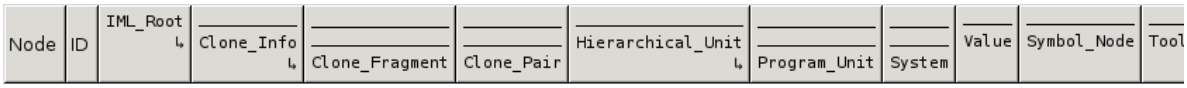

Abbildung 4.1: Konzepte als horizontaler Baum in den Spalten des Tree Views.

Listing 4.2 Intern genutzte Struktur für Tree View Kopfzeilenelemente.

```
type Concept_Column is
  record
    Column  : Gtk.Tree_View_Column.Gtk_Tree_View_Column;
    Class   : IML_Reflection.Class_ID;
    Label   : Gtk.Label.Gtk_Label;
    Column_ID : Integer := -1;
    Generation : Natural := 0;
    Expanded : Boolean := False;
    Visible : Boolean := False;
    Active  : Boolean := True;
    -- Used as temporary storage to speed up
    -- Analysis_Visitors.Insert_Class_Map_Data
    Class_Map_Cursor : Analysis_Visitors.Class_ID_Maps.Cursor
      := Analysis_Visitors.Class_ID_Maps.No_Element;
    -- Temporary storage for UI.Create_Column_Selection_Dialog
    Checkbox : Gtk.Check_Button.Gtk_Check_Button;
  end record;
```

4.2 Darstellung der Konzepte

Die Konzepte, welche bedingt durch die IML Klassenhierarchie entsprechend eine hierarchische Struktur haben, sollten gemäß dieser in den Spalten des Tree View Anzeigeelementes veranschaulicht werden. Wie für die Baumdarstellung des als IML Graph vorliegenden Programmes auch, sollten sie entsprechend traversierbar sein, im Sinne von Kinder können aus- sowie eingeklappt werden.

Da in GtkAda hierfür kein vorgefertigtes Element beziehungsweise eine Klasse existiert, behilft sich die Implementierung durch eine textuelle Darstellung, wobei das grafische Element, auf das diese gezeichnet ist, zur Interaktion fähig ist, also ein Klick das Element entweder aus- beziehungsweise einklappt. Das Resultat der Umsetzung, die sich im Paket `UI_Columns` findet, lässt sich in Abbildung 4.1 betrachten.

Zur Realisierung dessen bedarf es zunächst einer Möglichkeit im Programm Daten über die IML Konzepte zu bekommen. Praktikable Wege umschließen die Extraktion dieser Daten direkt selbst aus der IML Spezifikation, entweder zur Laufzeit oder noch zuvor, nämlich zur Übersetzungszeit, indem eine Quelldatei mit den notwendigen Daten generiert wird. Gewählt wurde letztlich jedoch die Nutzung der in Bauhaus enthaltenen Pakete `IML_Reflection` und `IML_Classes`, welche zur Laufzeit die notwendigen Daten über die Konzepte bereitstellen.

Zum Zeitpunkt der Initialisierung der Benutzeroberfläche werden alle Konzepte der Bauhaus IML in Form von `IML_Reflection.Class_ID` Typen abgerufen, um sie direkt in eine interne Struktur zu verpacken, die in Listing 4.2 zu sehen ist. Die Hauptaufgabe dieser Struktur ist es, diverse Daten zu speichern, die zwar auch aus dem Tree View abrufbar wären—beispielsweise die Sichtbarkeit der Spalte oder das `Label` Text-Element der Kopfzeile—aber dafür einer Iteration über den Tree View bedürften. Der Grund ist also das laufzeiteffiziente Bereithalten diverser Daten, die an anderer Stelle häufig abgefragt werden. Folgend wird für jede IML Klasse eine Spalte erzeugt und in den Tree View eingefügt, bis auf solche Spalten, die keine „Eltern“ haben, werden jedoch alle unsichtbar gemacht. Die Wahl direkt sämtliche Spalten einzufügen vereinfacht deren Handhabung deutlich und erzeugt keinen Nachteil in Sachen Performanz der Ansicht, da eine unsichtbare Spalte keine zusätzliche Last für die Darstellung erzeugt. Dabei sei anzumerken, dass zwischen `Visible` und `Active` ein deutlicher Unterschied besteht. Eine Spalte kann unsichtbar sein, weil sie lediglich nicht ausgeklappt ist, um jedoch nicht aktiv zu sein, muss sie vom Benutzer explizit im entsprechenden Dialog *Manage Concepts* deaktiviert worden sein, was implizit ebenfalls Unsichtbarkeit bewirkt.

Die Reihenfolge, in der die Spalten in den Tree View eingefügt werden, entspricht deren Reihenfolge im Falle eines vollständig ausgeklappten, flachen Baumes. Durch diese Anordnung genügt es zur Laufzeit die notwendigen Spalten anzuzeigen oder zu verstecken, um ein Aus- oder Einklappen von Kindern darzustellen. Die dafür notwendige Information, bezüglich der hierarchischen Beziehungen der Klassen, wird aus der Variable `IML_Reflection.Class_ID.Super` gewonnen, die ein Zeiger auf die Elternklasse ist.

Um die Hierarchie der Klassen auch visuell darzustellen, wird der Text des Kopfelementes einer Spalte bei jeder Aus- sowie Einklappoperation neu generiert. Dieser besteht in seiner Rohform zunächst aus dem Namen des Konzeptes, welcher in Form der Variable `IML_Reflection.Class_ID.Name` bereitgestellt und mit Leerzeilen umschlossen wird, um die Generation der Klasse darzustellen. Oberhalb des Namens einer Klasse werden in eventuell vorhandenen Leerzeilen Hilfslinien gezeichnet, damit der Baum, der durch sein horizontales Wachstum bei vielen Spalten schnell aus dem Bildschirm verschwinden kann, weiterhin übersichtlich bleibt und die Generation der Klassen stets eindeutig sichtbar ist.

4.3 Erfassung der Bibliothekskonzepte

Die Extraktion der Bibliothekskonzepte aus dem vorliegenden IML Graphen stellt einen wichtigen Aspekt dar, um eine möglichst umfassende Statistik über das zu analysierende Programm erstellen zu können. Das Bewerkstelligen der Extraktion findet auf Basis des eingelesenen IML Graphen selbst statt.

Hierbei wird sich dem Fakt beholfen, dass ein jeder Knoten der IML Klassenhierarchie, der entweder vom Typ `IML_Root` ist, oder davon abstammt, ein Attribut `SLoc` besitzt. Dieses Attribut ist eine Struktur, welche diverse Informationen darüber enthält, „woher“ dieser Knoten stammt, also aus welcher Quelldatei. Die Struktur umfasst noch weitere Informationen, unter anderem auch über die Zeile in der Quelldatei und vieles mehr, für diesen Zweck jedoch ist primär einzig der absolute Pfad zur Quelldatei interessant. Die Klasse `Include_Visitor` dient dazu, den IML Graphen zu traversieren und den absoluten Pfad eines jeden Knotens, der aus dessen `SLoc` Attribut gewonnen wurde, in einem Set

Listing 4.3 Veranschaulichung der Erweiterung des Sets an Pfaden mit Teilpfaden.

```
Urspruengliches Set:
  /usr/include/stdint.h
Erweitertes Set:
  /usr
  /usr/include
  /usr/include/stdint.h
```

Listing 4.4 Instanziierung eines neuen Bibliothekskonzeptes auf Basis von gesammelten Pfaden. Das Paket `SU_Sets` bietet Sets des Typs `Unbounded_String`. `Parent` ist ein Parameter der Prozedur.

```
  Current := SU_Sets.Element (C);
  Current_Class := new IML_Reflection.Abstract_Class'(
    (Name_Length => SU.Length (Current),
     Number_Of_Fields => 0,
     Name => SU.To_String (Current),
     Super => Parent, Fields => (others => null), Subclasses => null));
  Class_ID_Lists.Append (CID_List, Current_Class);
```

zu speichern, sofern er mit einem bestimmten Präfix startet. Dabei ist die Natur des Bauhaus `Visitor` zu beachten, der nur syntaktische Kinder besucht, weshalb beim abgeleiteten `Includes_Visitor` eine Erweiterung notwendig war, die auch die Knoten, welche im Attribut `Declaration_Table` von `Unit` Knoten hinterlegt sind, besucht. Das so zusammengestellte Set an Pfaden wird nun durch das Paket `Library_Concepts` aufbereitet, um daraus Bibliothekskonzepte zu gewinnen.

Dieses erweitert das Set an absoluten Pfaden zunächst um Teilpfade. Stellt man sich einen jeden Pfad als Menge vor, deren Elemente durch den Separator für Verzeichnisse getrennt sind, werden nun alle Teilmengen eines Pfades zum Set hinzugefügt, die das Element ganz rechts im Pfad, also dessen Anfang, enthalten, wie illustriert in Listing 4.3.

Damit durch Bibliothekskonzepte nicht unnötiger Aufwand für eine gesonderte Behandlung zu IML Konzepten entsteht, wird für das Set an Pfaden nun jeweils eine neue Instanzen des Typs `IML_Reflection.Abstract_Class` erstellt. Das erlaubt es insbesondere der Logik zur Darstellung der Konzepte in der Benutzeroberfläche, solche Konzepte, die von „echten“ IML Klassen stammen, mit denen, die hier für Bibliothekskonzepte „künstlich“ erstellt werden, gleich zu behandeln. Listing 4.4 zeigt die Instanziierung der neuen Konzepte und das Anhängen an eine Liste, derer sich `UI_Columns` bedient, um diese in der Benutzeroberfläche darzustellen, was dann aussieht wie in Abbildung 4.2.

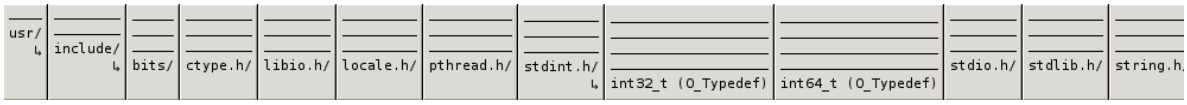

Abbildung 4.2: Bibliothekskonzepte als horizontaler Baum in den Spalten des Tree Views.

Listing 4.5 Eine der `Visit_` Prozeduren, wie sie für jede IML Klasse existieren. Die Klasse trägt sich als vorhanden ein und ruft danach die `Visit_` Prozedur der Elternklasse auf.

```
procedure Visit_Conditional
  (Self : access Analysis_Visitor;
   Node : access Standard.Conditional_Internals.Conditional_Class'Class)
is
begin
   Node_Util.Add_Class (Storables.Get_Node_ID (Storables.Storable (Node)),
                        Conditionals.Get_Class_ID);
   Visit_Value (Self, Node);
end Visit_Conditional;
```

Zwar werden Konzepte von IML Klassen und solche, die für Bibliothekskonzepte erstellt wurden, intern getrennt gespeichert, allerdings nur aus organisatorischen Gründen. Nach diesem Schritt, dem Extrahieren der Bibliothekskonzepte aus dem IML Graphen, ist der Ursprung der Konzepte für die weitere Nutzung der Klassen transparent und muss von der Logik nicht berücksichtigt werden. Durch die Art der Extraktion entsteht, wie bei Darstellung eines Programms selbst auch, ebenfalls eine Statistik auf einer Metaebene, da nicht genutzte Bibliothekskonzepte nicht berücksichtigt werden. Der Präfix, mit dem ein Pfad beginnen muss, um beachtet zu werden, erlaubt dazu eine Verfeinerung der Auswahl, wenn beispielsweise nur eine bestimmte Bibliothek von Interesse ist, oder auch die Deaktivierung der Erfassung jeglicher Bibliothekskonzepte.

4.4 Berechnung der Statistik

Der zentrale Punkt dieser Arbeit ist das Erfassen einer Statistik über die Verwendungshäufigkeit der Konzepte. Auch dies geschieht erneut auf Basis einer `Visitor` Klasse, die sich im Paket `Analysis_Visitors` findet und wiederum von der Prozedur `Calculate_From` des Pakets `Analysis` verwendet wird, welche die besagte Erfassung der Statistik ab einem beliebigen, übergebenen Knoten im IML Graphen starten kann. Dazu veranlasst die Prozedur zunächst den Aufbau einer Hilfsdatenstruktur, wie auch das Eintragen aller Konzepte in eine Instanz des Paketes `Ada.Containers.Ordered_Maps`, welche jedes Konzept mit einer Zahl assoziiert, um diverse Abläufe zu beschleunigen. Bei der Hilfsdatenstruktur handelt es sich um ein Feld an Strukturen, die für einen jeden relevanten Knoten einen Zeiger auf dessen korrespondierende Tree View Zeile enthalten, ebenso wie eine zunächst leere Liste, die später im eigentlichen Analyseprozess genutzt wird. Anschließend wird eine Instanz des `Analysis_Visitor` ab dem übergebenen Knoten gestartet, welcher die Daten erfasst und selbstständig in den Tree View einträgt, wodurch die Notwendigkeit einer Zwischenspeicherung der Ergebnisse entfällt.

Der Kern des `Analysis_Visitor` bilden die `Visit_` Prozeduren, derer eine für jede IML Klasse existiert. Diese Prozeduren haben essentiell immer den gleichen Ablauf, wie in Listing 4.5 zu sehen und haben im Prinzip lediglich nur die Aufgabe, die Tatsache ihres Aufrufs festzuhalten. Zuerst wird für den Knoten, für den diese Prozedur aufgerufen wurde, das Konzept vermerkt, für das diese Prozedur steht—in Falle von Listing 4.5 das `Conditional` Konzept. Vermerkt bedeutet, es wird an

die zuvor erwähnte, eingangs leere Liste der Hilfsdatenstruktur, die `IML_Reflection.Class_ID` des aktuell behandelnden Knotens angehängt. Anschließend ruft die Prozedur die entsprechende Prozedur der Elternklasse auf, wodurch letztlich der Besuch eines jeden Knotens bei der Prozedur `Visit_IML_Root` endet, in der sich die weitere Logik um Traversierung und die Erfassung der Daten befindet.

Der Ablauf von `Visit_IML_Root` ist in Algorithmus 4.1 skizziert. Dieser sichert zu Beginn zunächst einmal die Variable `Global_Class_Map`, deren Sichtbarkeit sich über den Körper des Pakets `Analysis_Visitors` erstreckt und Konzepte mit einer Zahl assoziiert, die für die Häufigkeit des Konzeptes steht. Diese Sicherung erfolgt in eine lokale Variable der Prozedur und muss zwingend eine tiefe Kopie sein, nicht nur eine Referenz auf das gleiche Objekt im Speicher, da anschließend die Zahlen der `Global_Class_Map` auf 0 gesetzt werden. Nun folgt die Traversierung der Kinder des aktuellen Knotens, gefolgt von einem Besuch weiterer Knoten, die der `Visitor` auslassen würde, da er lediglich syntaktische Kinder besucht. Darunter fallen unter anderem die Knoten, welche im Attribut `Declaration_Table` der Klasse `Program_Unit` gespeichert sind, ebenso wie die Knoten, welche den Typen eines besuchten Knotens angeben, sowie die Verfolgung verketteter Typkonstrukte wie `T_Type_Qualifier` oder `T_Pointer`.

Nachdem ein Besuch bei allen Knoten erfolgte, die sich entweder im Baum unter dem momentanen Knoten befinden, oder sonst Teil der Statistik des momentanen Knotens sind, fehlt noch die Eintragung der Klassen des momentanen Knotens in die `Global_Class_Map`. Dazu wird die Liste iteriert, die im Vorfeld über den Aufruf der `Visit_` Prozeduren hinweg aufgebaut wurde, wie in Zeile 6 im Algorithmus 4.1 zu sehen. Für jedes in der Liste enthaltene Konzept wird der korrespondierende Zähler in der `Global_Class_Map` erhöht. Damit auch Bibliothekskonzepte erfasst werden, wird darüber hinaus auf das `SLoc` Attribut des Knotens zugegriffen und direkt in ein Konzept in Form einer `IML_Reflection.Class_ID` umgewandelt, sofern es sich um ein gültiges Bibliothekskonzept handelt. Sollte dem so sein, wird auch dafür der entsprechende Zähler erhöht. Anschließend wird der momentane Datenstand der `Global_Class_Map`, das heißt die Zähler der Konzepte, für den aktuellen Knoten in die Tree View Ansicht eingetragen—in alle Reihen, in denen dieser vorkommt. Zu diesem Zweck wurden schon im Vorfeld Zeiger auf die Reihen im Tree View gesammelt, in denen der Knoten dargestellt wird, um diese zusätzliche Operation—linear in ihrer Laufzeitkomplexität mit der Anzahl an Reihen im Tree View—an dieser Stelle vermeiden zu können. Zuletzt folgt die Verschmelzung der Daten in der aktuellen `Global_Class_Map`, sowie der zu Beginn angelegten Kopie dieser, `Backup`, wobei die Zahlen der Konzepte zusammengezählt werden. Hierin findet sich die Begründung für die Nutzung einer `Ada.Containers.Ordered_Maps`. Diese besitzt zwar eine höhere Laufzeitkomplexität für das Einfügen neuer Elemente, jedoch findet dies lediglich zum Beginn, vor der Analyse selbst, statt. Danach können wir uns die Eigenschaft der Ordnung zu nutze machen und die Operation des Verschmelzens der Daten mit linearer Laufzeitkomplexität bewerkstelligen, da die Datenstruktur stets die selben Elemente enthält, lediglich mit verschiedenen Zählern.

Durch dieses Vorgehen werden letztlich nur für all die Knoten eines IML Graphen Werte in den Tree View eingetragen, die zu Beginn der Analyse auch wirklich im Tree View als Reihe vorhanden waren, egal ob aus- oder eingeklappt. Das ist eine direkte Konsequenz aus der Tatsache, dass ein Graph im Tree View nicht vollständig dargestellt wird, sondern lediglich die benötigten Teile, zuzüglich einer selbst definierbaren Tiefe. Zwar wäre es möglich bereits erfasste Werte für Reihen, die noch nicht

Algorithmus 4.1 Algorithmus für die Erfassung der Statistik. $global\_class\_map$ ist eine paketweit sichtbare Variable, die Konzepte mit einer Zahl assoziiert.

```
1:  procedure Visit_IML_Root(Self, Node)
2:      Backup ← Global_Class_Map                          // Tiefe Kopie
3:      Clear_Occurrences(Global_Class_Map)                // Setzt die assoziierten Zahlen auf 0
4:      Visit_All_Children_Of(Node)
5:      Additional_Visits_For(Node)
6:      for all I ∈ Class_List(Node) do
7:          Increment_Occurrence_Count(i)
8:      end for
9:      if Make_Concept(SLoc) ∈ List_Of_Valid_Concepts then
10:         Increment_Occurrence_Count(Make_Concept(SLoc))
11:     end if
12:     if Has_PTA_Instance then
13:         Process_PTA_Nodes_For(Node)
14:     end if
15:     Insert_Data_Into_Tree_View(Global_Class_Map, Node)
16:     Global_Class_Map ← Union(Global_Class_Map, Backup)
17: end procedure
```

angezeigt werden vorzuhalten, ist aber aufgrund des Speicherverbrauches, der bereits in Kapitel 3.3 erörtert wurde, nicht praktikabel.

Für die Erfassung der Analyse, das Traversieren des IML Graphen, wurde Eingangs die Parallelisierung der Arbeit durch Ada Tasks erwogen. In der Ausführung zeigte sich jedoch durch Messungen, wie der dominierende Teil der Ausführungszeit bei der Eintragung der Werte in die Benutzeroberfläche liegt und die hier erfolgende Traversierung, inklusive des in `Visit_IML_Root` umgesetzten Algorithmus, eine vernachlässigbar geringe Zeit einnimmt.

4.5 Nutzung der Zeigeranalyseinformationen

Zeigeranalyseinformationen werden berücksichtigt, wenn vor dem Start des `Analysis_Visitor` entsprechende Informationen im IML Graphen gefunden wurden und der Benutzer deren Miteinbeziehung wünscht. Dafür wird das in Bauhaus verfügbare Paket `PTA_Querys` genutzt, das einen universellen Zugang auf die Daten der Zeigeranalyse verfahren ECR, Andersen und Das geben soll. Genutzt werden die Knoten der Klasse `Abstract_Object`, welche von einer feld-sensitiven Zeigeranalyse stammen, mittels einem Iterator, der über den Aufruf `PTA_Querys.Make_Representation_Iterator` zu bekommen ist. Die daraus gewonnenen Werte werden in der Form von Reichweiten in der Statistik dargestellt.

An dieser Stelle endet leider der einheitliche Zugriff auf Zeigeranalyseergebnisse mit dem Paket `PTA_Querys`, da die Klasse `Abstract_Object`—entsprechend ihres Namens—lediglich abstrakt ist und

die konkreten Klassen, die die verschiedenen Zeigeranalyse Verfahren nutzen, gänzlich verschieden sind und individuell abgehandelt werden müssen.

Allerdings zeigen sich schon zuvor ein paar Probleme mit diesem Paket, da es noch nicht fertig zu sein scheint. So ist das Resultat des Aufrufs `PTA_Querys.Make_Representation_Iterator` auf einer ECR Instanz ein nicht-initialisierter Iterator und der Aufruf von `PTA_Querys.More`, ebenfalls auf einer ECR Instanz, ruft sich wiederum selbst auf, was eine Endlosschleife erzeugt. Aus diesem Grund ist das Nutzen von Informationen aus ECR Knoten im Programm deaktiviert.

5 Test und Validierung

Das Testen ist ein ein elementarer Bestandteil im Entwicklungsprozess einer Software. In einem Fall wie diesem, wo Werte erfasst werden, bedarf es auch einer Validierung dieser. Nachfolgend wird erläutert, wie dies gelöst wurde, anschließend finden sich Informationen zur letztendlichen Leistungsfähigkeit des entwickelten Programms.

5.1 Vorgehen

Für das Validieren der Ergebnisse wurde vor dem Entwicklungsprozess eine Reihe kleiner Testprogramme in den Quellsprachen C sowie C++ erstellt, die mit dem Ziel entworfen wurden, sowohl exotischere, als auch gewöhnlichere Nutzung der verwendeten Sprachen abdecken zu können. Dabei handelt es sich um die folgenden Dateien mit dem entsprechend angegebenen Hauptzweck.

`arraytest.cpp`
Verschiedene Arten Elemente eines Feldes abzurufen.

`class.cpp`
Simple Nutzung einer C++ Klasse.

`nested_class.cpp`
Mehrfach verschachtelte Klassen.

`class_templates.cpp`
C++ Klassentemplates, darunter auch eine Spezialisierung.

`dyn_ptrs.cpp`
Zur Laufzeit vergebene Zeigeradressen.

`func_ptrs.cpp`
Funktionszeiger und verschiedene Arten deren Nutzung.

`inline.c`
Funktionen die mit dem `inline` Schlüsselwort versehen sind.

`lib_func_ptrs.cpp`
Funktionszeiger auf Funktionen, die von Bibliotheken bereitgestellt werden.

`lib_local_func_calls.c`
Funktionszeiger auf lokale und Bibliotheksfunktionen, zur Unterscheidung derer.

`loop.cpp`
Simple Nutzung verschiedener Schleifenkonstrukte.

`typedeftest.cpp`
Mehrere Typedef und Zeiger miteinander verkettet.

`recfac.c`
Funktion zur rekursiven Berechnung der Fakultät.

`searchforexit.cpp`
Minimales Realweltbeispiel. Suche nach einem Schlüsselwelt in einer gegebenen Datei.

`linematching.cpp`
Kleines Realweltbeispiel. Gruppierung von Zeichenketten nach Gleichheit in einer Eingabe.

Ebenfalls vor dem Entwicklungsprozess wurde, hauptsächlich für das Ermitteln und Prüfen der Leistungsfähigkeit des Programms, eine Sammlung an IML Graphen verschiedener Größe erstellt, welche aus automatisch generiertem C-Quellcode erzeugt wurden. Näheres zu deren Erzeugung findet sich in Kapitel 3.5.

Im Entwicklungsprozess wurden konstant beide Mengen an IML Graphen manuell mit dem entstehenden Programm eingelesen, dargestellt und visualisiert, um mögliche Regressionen und sonstige Veränderungen in der erzeugten Statistik, sowie der Leistungsfähigkeit, direkt bemerken zu können. Zusätzlich wurden in sinnvollen Abständen Programmversionen archiviert, um vergleichende Tests bei gleicher Eingabe gegen ältere Versionen ausführen zu können. Sofern notwendig, wurde umgehend eine Korrektur eines Fehlers vorgenommen, oder die Ursache für eine Verschlechterung der Leistungsfähigkeit ermittelt und behoben. Auf eine Automatisierung dieser Tests wurde dabei aus Zeitgründen verzichtet.

Im Quellcode findet eine große Zahl an Selbsttests bezüglich Konsistenz, Vor- und Nachbedingungen statt. Dies erfolgt durch die Nutzung von `if` Abfragen, wenn es möglich ist wieder einen validen Programmzustand herzustellen, oder der Compiler-Anweisung `pragma Assert`, sofern dem nicht so sein sollte.

Im Verlauf der Entwicklung wurde insbesondere eine Verschlechterung der Leistungsfähigkeit bezüglich des Darstellens sehr großer IML Graphen festgestellt. Dabei handelt es sich um eine Auswirkung der Anzahl an temporärer Zwischenspeicher, welche sich während der Entwicklung stets erhöht hat und einer Beschleunigung der Erfassung der Statistik dienen. Aus diesem Grund wurde die Verschlechterung der Leistungsfähigkeit nicht als Fehler angesehen und damit nicht wieder zu einen früheren Zustand erhöht.

5.2 Performanz

Alle Tests wurden auf dem Abteilungsrechner `pslx2` ausgeführt. Dessen Hardware umfasst einen Intel Xeon Prozessor mit 4 Kernen zu je 3Ghz und 12GB Arbeitsspeicher. In sämtlichen Tests wurde lediglich der Programmschalter `-verbose` als Argument, neben dem zu ladenden IML Graphen, übergeben.

#	Knoten	Syntaktische Knoten	SLOC	Navigieren	Vollständig
0	1088638	745447	220724 (POV-Ray)	Ja	Ja
1	3950281	3600038	450000	Ja	Ja
2	7900281	7200038	900000	Ja	Ja
3	11850281		1350000	Ja	Nein
4	19650362		2250000*	Nein	Nein
5	12233955		2300000*	Ja	Nein
6	14834009		2600000*	Nein	Nein

Tabelle 5.1: Darstellungstest mit generierten IML Graphen und POV-Ray zum Vergleich.
* Diese Graphen wurden durch `imllink` erzeugt, der SLOC Wert bildet sich hier aus den addierten Werten der verbundenen Graphen und entspricht daher keinem realen Wert.

Keiner der für diese Tests genutzten IML Graphen enthält Informationen aus einem Zeigeranalyse Verfahren.

Die endgültige Leistungsfähig des entwickelten Programms, bezüglich der Darstellung eines IML Graphen, ist in Tabelle 5.1 dargestellt. Die Spalte „Navigieren“ bezieht sich darauf, ob es möglich ist, im geladenen IML Graphen ohne nennenswerte Verzögerung zu navigieren, Reihen auszuklappen und beliebige Elemente zu suchen. „Vollständig“ enthält die Antwort auf die Frage, ob es möglich ist, sämtliche Knoten des geladenen IML Graphen auf einmal in den Tree View zu laden. Die dabei entstehende Anzahl an Reihen entspricht mindestens der Anzahl syntaktischer Knoten im Graphen.

Die anfänglichen Prototypen des entwickelten Programms waren dazu in der Lage, selbst den Graphen #3 in der Tabelle 5.1 zur Ansicht zu bringen. Durch die Nutzung mehrere Zwischenspeicher, zur Beschleunigung anderer Prozesse, erhöhte sich jedoch der Speicherverbrauch des Programms, welcher auf dem genutzten Testrechner den Graphen #5 nun zum größten, darstellbaren Graphen macht. Dieser Graph lässt sich erfolgreich laden, darstellen und vom Benutzer navigieren, ist allerdings bereits zu groß, um vollständig—also jeder Knoten—in den Tree View geladen werden zu können. Auch hier handelt es sich um ein Problem mit dem Speicher, von dem sich GtkAda nicht mehr genug allokieren kann und daher die Ausführung des Programms beendet. Der letzte Graph, der sich vollständig in den Tree View laden lässt, ist #2 in der Tabelle 5.1.

Die Graphen benötigen dabei durchaus Zeit um geladen zu werden. Dieser Vorgang, der beim Programmstart stattfindet, wird durch einen Aufruf an die, von Bauhaus bereitgestellte Funktion `IML.IO.Load` bewerkstelligt und benötigt unter 120 Sekunden für den Graphen #3 und unter 11 Sekunden für #0. Das Darstellen des Graphen durch die Schaltfläche „View Graph“ in der Benutzeroberfläche nimmt sich im Falle von Graph #3 nochmals weitere 9 Sekunden, ab dann ist aber ein flüssiges Navigieren im Graphen möglich.

Mit diesen Zahlen kann die Erfassung der Statistik nicht ganz mithalten. Tabelle 5.2 zeigt die Ergebnisse zweier Messungen auf dem IML Graphen des Testprogramms `linematching.cpp`, der 115206 Knoten enthält. Bei der Darstellung wurden in beiden Fällen zusätzlich die Programmschalter `-init-depth 99999` und `-node 3` genutzt, um auf simple Art und Weiße direkt den Wurzelknoten des Graphen

Aktivierte Konzepte	Dauer (Sekunden)
`IML_Root`	480
Alle IML Konzepte	2945

Tabelle 5.2: Dauer der Erfassung der Statistik in Sekunden auf dem IML Graph aus des Testprogramms `linematching.cpp` mit 115206 Knoten.

anzuzeigen, sowie sämtliche syntaktischen Knoten des Graphen direkt in die Ansicht einzufügen. Der durchgeführte Test variiert die Anzahl an Spalten in beiden Tests durch die Deaktivierung von ungewollten Konzepten mittels des Dialogs „Manage Concepts“.

Im Ergebnis in Tabelle 5.2 zeigt sich direkt die enorme Abhängekit der nötigen Laufzeit von der Anzahl an Spalten, für die in der Benutzeroberfläche Werte eingetragen werden. Dies deckt sich mit weiteren Messungen im `Analysis_Visitor` selbst, die aufzeigen, dass der genutzte Algorithmus selbst einen nur sehr geringen Anteil an der Laufzeitdauer zu verantworten hat, während der Großteil bei der GtkAda Operation liegt, die einen Wert in eine Zelle des Tree View einträgt. Von dem Modell zur Datenhaltung, welches sich hinter dem GtkAda Tree View verbirgt, wird lediglich die Möglichkeit angeboten, den Tree View Zelle für Zelle zu befüllen, nicht jedoch eine ganze Zeile auf einmal. Es wird ebenfalls keine direkte Möglichkeit gegeben, die Datenstruktur im Hintergrund selbst zu bearbeiten. Eine effizientere Art um Daten in den Tree View einzufügen, würde die Erfassung der Statistik, beziehungsweise eigentlich deren Darstellung in der Benutzeroberfläche, um ein vielfaches beschleunigen.

GtkAda leidet noch an einer weiteren Unannehmlichkeit, welche in bestimmten Situationen das Nutzungserlebnis verschlechtern kann. Bei Operationen, die sehr schnell enorm viele Zeilen im Tree View in ihrer Ansicht ändern, führt GtkAda im Hintergrund diverse Berechnungen aus, ohne jedoch die Benutzeroberfläche zu blockieren. Während diesem Zustand, also solange die Berechnungen andauern, werden andere Interaktionen mit der grafischen Oberfläche entweder stark verzögert, gar nicht, oder sogar nur teilweise durchgeführt und mitten in ihrer Ausführung angehalten. Provozieren lässt sich dies durch ein Einklappen aller Reihen („Collapse All Rows“) nach einem vorangegangenen Ausklappen dieser („Expand All Rows“).

6 Zusammenfassung

Dieses Kapitel fasst die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick für zukünftige Verbesserungen sowie Erweiterungen.

6.1 Fazit

Das Resultat dieser Arbeit ist das entwickelte Programm `iml_concept_stats`, welches graphisch aufbereitet einen IML Graphen laden und darstellen kann. Diese Darstellung zeigt den Graphen als Baum und kann vom Benutzter frei navigiert werden, auch bei sehr großen Graphen. Die Oberfläche erlaubt es dem Benutzer außerdem beliebig viele, beliebe Teilgraphen, sowie ganze Gruppen an Teilgraphen auf einmal darzustellen. Des Weiteren werden auch die Konzepte eines geladenen Graphen visualisiert, inklusive solcher, die aus genutzten Bibliotheken stammen. Ebenfalls in einer Baumdarstellung, wird die Funktionalität geboten, die verschiedenen Konzepte zu verstecken, oder sogar vollständig zu deaktivieren. Bei Bibliothekskonzepten bieten sich darüber hinaus noch Filtermöglichkeiten, um zu definieren was als Bibliothekskonzept überhaupt in Betracht gezogen wird. Als Kernkompetenz erlaubt das entwickelte Programm die Erfassung einer Statistik über die aktuell dargestellten Inhalte, deren ermittelte Werte ebenfalls in der Benutzeroberfläche dargestellt werden. Zur weiteren Bearbeitung besteht die Möglichkeit, die angezeigten Daten jederzeit in ein tabellenkalkulationskompatibles Format zu exportieren, wobei diverse Filter- und Konfigurationsoptionen verfügbar sind.

Durch die Nutzung von Ada als Programmiersprache, sowie der Benutzeroberfläche durch GtkAda, ergeben sich diverse Optionen zur Zusammenarbeit mit anderen Werkzeugen von Bauhaus. Eine Erweiterung oder Veränderung des vorliegenden Programmes ist einfach gehalten, insbesondere das Abändern zusätzlich besuchter Knoten im `Analysis_Visitor`, wodurch die Art der Statistik, die erfasst wird, geändert werden kann.

Es gibt jedoch noch einige Stellen, die etwas mehr Reife bedürfen. Hauptsächlich durch fehlende Zeit konnten einige Dinge nicht so umgesetzt werden, gewünscht. Allen voran steht dabei die Benutzeroberfläche, welche an vielen Stellen nicht unbedingt intuitiv zu benutzen ist. Weitere Komfortfunktionen können das Nutzererlebnis deutlich verbessern.

Der Aufwand zur Einarbeitung in Bauhaus wurde von Beginn an unterschätzt. Dazu dauerte die Planungsphase zu lange, da zunächst der Versuch unternommen wurde, die Lösung der Aufgabe mit einem eigenen Dateiformat zu erstellen. Eine detaillierte Evaluation, insbesondere mit Prototypen, gab letztlich der gewählten Lösung den Vorzug. Im Zwischenvortrag stellten sich auch noch einige Punkte heraus, deren Entwurf neu überarbeitet werden musste, um allen Ansprüchen genügen zu können.

Ein Punkt, der mehr Beachtung hätte finden sollen, ist speziell die Verifikation der Statistik, möglichst in einem automatisierten Verfahren. Zwar wurden viele Fälle, darunter auch seltene und eher exotische Anwendungen, geprüft, allerdings würde ein deutlich breiteres Prüfverfahren mehr Gewissheit über die Korrektheit der Ergebnisse bringen. Eine derartige Verifikation wäre nicht mehr manuell von Hand ausführbar und würde stark von einer extra entwickelten Rahmenstruktur profitieren.

6.2 Ausblick

Im Wesentlichen seien hier alle Teile zu nennen, die im vorherigen Kapitel als unzureichend genannt wurden. Neben einer umfassenden, automatisierten Rahmenstruktur zur Verifikation, umschließt dies auch eine Überarbeitung der grafischen Benutzeroberfläche hinsichtlich Aspekten der Nutzerfreundlichkeit und der Nutzungserfahrung.

Ein weiterer wichtiger Teil ist die aktuelle Leistungsfähigkeit des Programms. Während die Darstellung von Graphen, durch das Laden von Reihen erst zum Zeitpunkt ihrer Notwendigkeit, auch solche einer durchaus annehmbaren Größe zur Anzeige bringt, gibt es bei der Erfassung der Statistik hier Nachholbedarf. Der zeitkritische Punkt liegt dabei auf Seiten von GtkAda, weshalb eine Lösung wohl mit dem Ableiten und Redefinieren des Datenmodells, der GtkAda Tree View Klasse, möglich wäre. Im Detail geht hierbei um eine effiziente Eintragung von Daten in dieses. Alternativ könnte man auch an einer anderen Stelle ansetzen und einen Weg suchen, um die Berechnungen, die GtkAda hier langsam machen, verzögert ausführen lassen kann.

Die gesamte Performanz des Programms, speziell zu Zeiten von sehr vielen Reihen, die im Tree View angezeigt werden, kann deutlich erhöht werden, wenn das von GtkAda übernommene Rendering der Daten abgeändert wird. Aktuell führt GtkAda im Hintergrund Rendervorgänge und sonstige Berechnungen auch für solche Teile der Darstellung durch, die eigentlich gar nicht im Bild sind. Da es sich bei der Darstellung vieler Reihen je um eine sehr lange Liste handelt, sind die meisten davon nicht im Bild, müssten also auch nicht im Hintergrund aktualisiert werden.

Auch eine ungelöste Sache der dargestellten Statistik ist eine Neuberechnung derer. Aktuell ist es nicht möglich herauszufinden, ob für einen bestimmten Teil bereits eine Statistik erstellt wurde und eine Neuberechnung überflüssig ist. Eine zufriedenstellende Lösung sollte es, wenn möglich vermeiden, auf die Speicherung von Daten zurückzugreifen, da hier wiederum ein potentiell sehr großer Flaschenhals für die Skalierfähigkeit des Programms entsteht. Eine Möglichkeit wäre das Vorhandensein bereits eingetragener Werte als Zeichen zu werten, keine Erneute Berechnung zu vollziehen. Dies gibt aber die aktuelle Architektur des Programms nicht her, da die Berechnung noch vor dem Eintragen der Werte geschieht.

Eine Integration des Programms in den Bauhaus Codebrowser `cobra` ist theoretisch jederzeit machbar. Es wäre auch möglich, die Benutzeroberflächen des `IML_Navigator` mit der von `iml_concept_stats` zu verbinden, um bei der Navigation durch einen dargestellten IML Graphen die zusätzlichen Informationen aus dem `IML_Navigator` betrachten zu können.

Zuletzt wäre auch ein Ausbau der Funktionalität bezüglich der Nutzung von Zeigeranalyse Ergebnissen nützlich. Dabei gilt es zum einen die Breite der unterstützen Verfahren zu erhöhen und zum anderen

mehr Optionen beim Auslesen dieser Informationen zu bieten, wie beispielsweise ob feld-sensitive oder -insensitive Informationen abgefragt werden sollen.

Literaturverzeichnis

[BBD09] F. Beck, M. Burch, S. Diehl. Towards an Aesthetic Dimensions Framework for Dynamic Graph Visualisations. In E. Banissi, L. J. Stuart, T. G. Wyeld, M. Jern, G. L. Andrienko, N. Memon, R. Alhajj, R. A. Burkhard, G. G. Grinstein, D. P. Groth, A. Ursyn, J. Johansson, C. Forsell, U. Cvek, M. Trutschl, F. T. Marchese, C. Maple, A. J. Cowell, A. V. Moere, Herausgeber, *IV*, S. 592–597. IEEE Computer Society, 2009. (Zitiert auf Seite 15)

[gtk] *Reference Manual GtkAda-2.14.0*. (Zitiert auf Seite 25)

[SKLS10] H. Song, B. Kim, B. Lee, J. Seo. A Comparative Evaluation on Tree Visualization Methods for Hierarchical Structures with Large Fan-outs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, S. 223–232. ACM, New York, NY, USA, 2010. doi: 10.1145/1753326.1753359. URL http://doi.acm.org/10.1145/1753326.1753359. (Zitiert auf Seite 15)

Alle URLs wurden zuletzt am 23. 01. 2014 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift