Institut für Softwaretechnologie

Diplomarbeit Nr. 3003

Identifikation und Extraktion relevanter Kenngrößen aus IML-Graphen

Ozan Kilic

Studiengang: Softwaretechnik

Prüfer: Prof. Dr. Erhard Plödereder

Betreuer: Dipl-Inf. Steffen Keul

begonnen am: 01. Februar 2010

beendet am: 02. August 2010

CR-Klassifikation: D.2.2

Inhaltsverzeichnis

Abbildungsverzeichnis **3**

Tabellenverzeichnis **4**

1 Einführung **1**
1.1 Bauhaus . . . 1
1.1.1 Intermediate Language (IML) . . . 2
1.2 Aufgabenstellung . . . 3
1.2.1 Vorgehensweise . . . 5
1.3 Inhalt des Dokuments . . . 6

2 Analysen in Bauhaus **7**
2.1 Einführung in die Analysen . . . 7
2.1.1 Zeigeranalyse in Bauhaus . . . 7
2.1.2 Kontrollflussanalyse in Bauhaus . . . 9
2.1.3 SSA Form in Bauhaus . . . 10
2.1.4 IML Analyse nebenläufiger Programme . . . 10
2.2 Static Single Assignment Form (SSA) . . . 12
2.2.1 Dominanz . . . 14
2.2.1.1 Dominanzbaum . . . 16
2.2.1.2 Dominanzgrenze . . . 17
2.2.2 Phi Platzierung . . . 21
2.2.3 Umbenennung . . . 23
2.2.4 Seiteneffekt . . . 25
2.2.5 Metriken der SSA Analyse . . . 27

3 Das Tool relevant_stats **30**
3.1 Einfuhrung in das Tool relevant_stats . . . 31
3.2 Datenbank . . . 32
3.2.1 SQL . . . 32

3.2.1.1 Data Manipulation Language (DML) 36
3.2.1.2 Data Definition Language (DDL) 37
3.2.1.3 Datensichten (Views) in SQL 38
3.2.2 Datenbankmodell 40
3.2.3 SQLite 42
3.2.3.1 SQLite Schnittstelle in Ada 44
3.2.4 DB Operationen 50
3.2.4.1 Check 50
3.2.4.2 List 51
3.2.4.3 Insert 52
3.2.4.4 Update 54
3.2.4.5 Delete 55
3.2.4.6 ToCSV 55
3.2.4.7 ShowTables 56
3.3 Plot 56
3.3.1 GNUplot 57
3.3.2 Orientierung 61
3.4 Implementierung 66

4 Vorgehensweise für eine neue Analyse **71**
4.1 Änderungen in globals.ads 72
4.2 Änderungen im Paket *Analysis* 75
4.3 Änderungen in *Db_ Operations* 78

5 Bewertung dieser Arbeit **86**
5.1 Ergebnisse 86
5.2 Ausblick 88

Literaturverzeichnis **89**

A relevant_stats **92**
A.1 Kommandozeile 92
A.2 Abhängigkeiten 93

Abbildungsverzeichnis

1.1 Generierung und Verarbeitung von IML 3

2.1 Analysen in Bauhaus 8
2.2 Beispielcode für Fluss-Sensitivität 9
2.3 Beispielprogramm und zugehöriger Kontrollflussgraph(CFG) 10
2.4 Das Philosophen-Problem [SM99] 11
2.5 Straight-Line-Code und seine SSA-Version 13
2.6 Phasen der SSA Analyse 13
2.7 Beispiel für Dominator 15
2.8 Beispiel für ein Dominanzbaum 17
2.9 Beispiel-1 für die Dominanzgrenze 19
2.10 Beispiel-2 für die Dominanzgrenze 21
2.11 Build-SSA 27

3.1 Das Modell von relevant_stats 31
3.2 DML-Anweisungen [Sch10] 37
3.3 Datenbankmodell von relevant_stats 41
3.4 Beispiel zu einer graphischen Darstellung 57
3.5 Ausgabe der Plot-Befehle 58

Tabellenverzeichnis

3.1 Datentypen in SQL [GM] 35
3.2 Beispieltabelle 61
3.3 Beispieltabelle-2 64
3.4 Beispieltabelle-3 66

1 Einführung

Dieses Dokument beschreibt die Durchführung und Ergebnisse der Diplomarbeit "Identifikation und Extraktion relevanter Kenngrößen aus IML Graphen". Sie wurde ausgeschrieben von der Abteilung für Programmiersprachen und Übersetzerbau des Instituts für Softwaretechnologie an der Universität Stuttgart.

1.1 Bauhaus

Wartung und Evolution von Softwaresystemen verursachen 50-80% der Softwarekosten. Das Projekt **Bauhaus** hat das Ziel, Wartungsingenieuren durch analytische Methoden und Werkzeuge das Programmverstehen auf Code- und Architekturebene zu erleichtern, um damit Qualität und Effizienz der Wartungsprozesse zu verbessern.[UB]

Bauhaus ist ein langjähriges und fortlaufendes Rahmenprojekt der Abteilung Programmiersprachen und Übersetzerbau der Universität Stuttgart, welche in enger Zusammenarbeit mit der Universität Bremen und der Axivion GmbH steht.
Die genauen Ziele von Bauhaus sind die Entwicklung von Beschreibungsmitteln, Analysemethoden und zugehörigen Werkzeugen, die es dem Wartungsingenieur erleichtern, semi-automatisch

- Fehlererkennung, z.B. Race Condition
- diverse Sichten auf die Architektur von Altsystemen herzuleiten,
- wiederverwendbare Komponenten zu identifizieren und
- Auswirkungen von Programmänderungen abzuschätzen.

Um dieses Ziel zu erreichen werden Beschreibungsmittel sowie Analysemethoden und -werkzeuge erforscht, die es ermöglichen, Architekturen zu extrahieren, darzustellen und zu manipulieren. [Pro]
Um die Erweiterbarkeit auf verschiedene Quellsprachen zu gewährleisten, wird allerdings nicht der Quelltext selbst, sondern die Zwischendarstellung InterMediate Language (IML) analysiert. Die IML wurde für das Projekt Bauhaus konzipiert und wird im Rahmen dieses Projekts benutzt, um die wichtigen Metriken von einer vorgegebenen Analyse zu messen.

1.1.1 Intermediate Language (IML)

Intermediate Language (IML) ist die im Rahmen des Projekts Bauhaus entwickelte graphbasierte Interndarstellung, die syntaktischen und semantischen Informationen über das zu analysierende Programm repräsentiert. In der IML werden verschiedene Sourcen, die in verschiedenen Programmiersprachen implementiert sind, einheitlich dargestellt.

Die Entwicklung der IML hat mit der Studienarbeit [WUE96] angefangen. In dieser Studienarbeit wurde die Grundlage gelegt, die die Zwischendarstellung für Ada 93 spezifiziert und einen Übersetzer implementiert. Dann kamen die Erweiterungen der IML zustande. Die Erweiterungen für die Sprache C wurden in [ROH98], für Java in [KNA02], für C++ in [KAR03] und für Ada95 in [KEU05] realisiert.

Der Aufbau der IML ist vergleichbar mit dem Aufbau des Abstract-Syntax-Tree(AST) eines Übersetzers. Es handelt sich um einen Graph, der die Programminformationen in seinen Knoten speichert, die für die Programmanalysen nützlich sind. Während bei im AST die einzelne Übersetungseinheiten dargestellt werden, wird im IML das ganze Programm dargestellt. IML hat zwei zentrale Eigenschaften, eine davon ist die Abstraktion der Programmiersprachen, was bedeutet, dass die Darstellung der IML unabhängig von der Programmiersprache geschieht. Die zweite Eigenschaft bezieht sich auf die quellennahe Darstellung. Quellennah bedeutet, dass die Struktur des Quellenprogrammes wiedererkennbar ist.

In der Abbildung 1.1 wird dargestellt, wie IML generiert und verarbeitet wird. Mit Hilfe von den Bauhaus-Werkzeugen jafe für Java-Sourcen und cafeCC für C-,Assembler-,C++ Sourcen wird eine IML Darstellung der einzelnen Übersetzungseinheiten generiert. Mit

dem Tool imllink werden dann die IML-Dateien zu einer gesamten Darstellung zusammengefasst. Nachdem die IML-Datei erstellt wurde, können die Analysen in Bauhaus durchgeführt werden, um das Projekt zu analysieren. Momentan gibt es in Bauhaus verschiedene Analysen wie z.B. SSA Analyse, Points-To-Analyse , Kontrollflussanalyse, Synchronisationsanalyse und einige andere.

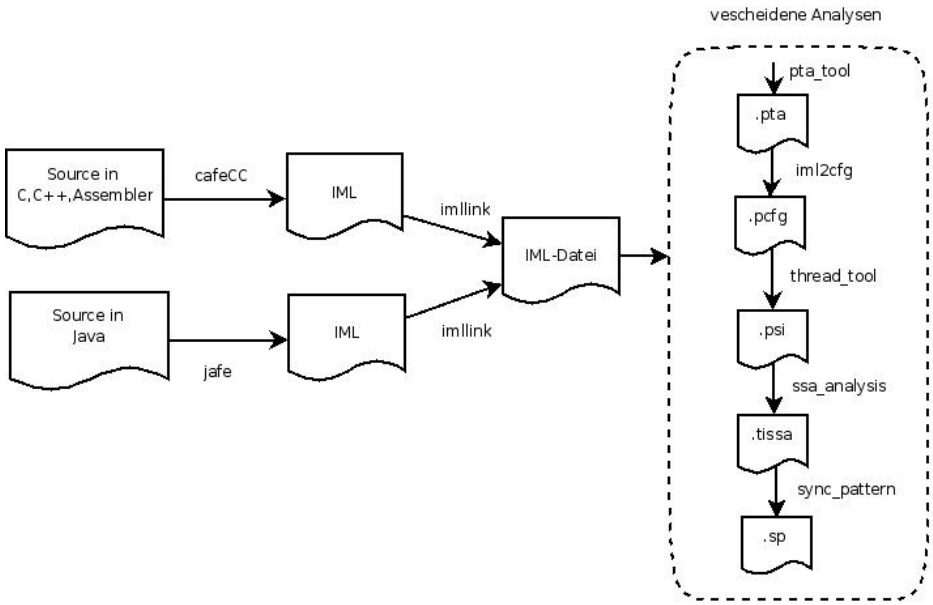


Abbildung 1.1: Generierung und Verarbeitung von IML

1.2 Aufgabenstellung

Analysen, die in Bauhaus zur Verfügung stehen, sind zum Beispiel:

- Erzeugung von Kontrollflussinformation
- Auffinden nicht verwendeter Routinen
- Zeigeranalysen(Points-To-Analysis)
- Auffinden von Seiteneffekten
- Auffinden der Verwendung uninitialisierter Variablen

Diese Analysen haben je nach Algorithmus verschiedene Komplexitäten. Manche Algorithmen brauchen sogar exponentielle Zeit [rnEP09]. Während ein kleines Projekt beispielsweise für eine SSA Analyse 20 Sekunden braucht, dauert die selbe Analyse bei einem großen Projekt tagelang. Für diesen Fall stellen sich die folgenden Fragen:

- Welche Faktoren haben Einfluss auf diese Analyse?
- Kann man diese Faktoren identifizieren?
- Kann man diese identifizierten Faktoren in verschiedenen Projekten vergleichen?
- Kann man diese oben genannten Anforderungen ohne zusätzliche Komplexität erfüllen?

Die Diplomarbeit ist aus oben genannten Fragen entstanden und die genauen Ziele dieser Diplomarbeit sind:

- Identifizieren und Extrahieren relevanter Kennzahlen aus IML Graphen für eine vorgegebene Analyse. Dadurch sollen die Faktoren bestimmt werden, die Einfluss auf diese Analyse haben. Im Rahmen der Diplomarbeit sind diese Faktoren als Metriken genannt.
- Diese Messergebnisse von verschiedenen Analysevorgängen sollen ermittelbar, vergleichbar und damit anwendbar sein. Daher wurde es entschieden, dass die Messergebnisse in eine relationale Datenbank gespeichert werden.
- Die Möglichkeit zur Generierung von Diagrammen soll entworfen und implementiert werden.
- Die Diagramme sollen in wissenschaftliche Veröffentlichungen eingebunden werden können

Diese Aufgaben können in zwei Schwerpunkte unterteilt werden. Zum einen sind die vorgegebene Analyse wie z.B. die SSA Analyse zu analysieren und die identifizierte Metriken festzustellen, die Einfluss auf die vorgegebene Analyse haben, und zum anderen ein Tool zu programmieren, das die Messwerte aus der IML-Datei liest und in eine separate Datenbank schreibt, damit die Informationen hinterher nutzbar werden. Der zweite Schwerpunkt besteht wieder aus zwei Aspekten, der eine ist der Datenbankteil, der die Auflistung, Aktualisierung und das Hinzufügen und Löschen von Messwerten ermöglicht, und der andere Teil besteht darin die aus der Datenbank erreichbaren Informationen im Diagramm ausgeben zu können.

Die Realisierung der oben genannten Anforderungen erfolgt durch das separate Werkzeug **relevant_stats**, welches im Rahmen dieser Diplomarbeit implementiert wurde. Mit Hilfe des Tools relevant_stats soll es möglich werden, zum einen eine Aussage darüber zu treffen, welche Faktoren Einfluss auf die vorgegebene Analyse haben, welche Laufzeit (und auch welcher Speicherbedarf) diese Analyse hat. Dies ist allerdings nur dann möglich, wenn vorhergehend eine Analyse in Bauhaus durchgeführt wird. Darüber hinaus soll untersucht werden, ob eine Voraussage über die Laufzeit durch gemessene Informationen der identifizierten Metriken möglich ist. Diese Aussage kann wie folgt aussehen: "Die SSA Analyse für diese Analyse wird voraussichtlich 2 Stunden dauern, weil ...". Daher muss die Korrelation zwischen den Metriken bestimmt werden. Eine Korrelation beschreibt die Beziehung zwischen zwei oder mehreren statistischen Variablen. Nachdem die Metriken ausgewählt worden sind, die irgendeinen Einfluss auf diese Analyse haben, sollte für jede Metrik bestimmt werden, wie groß dieser Einfluss im Vergleich zu den anderen Metriken ist. Bei der Bestimmung einer Korrelation ist es schwer zu erkennen, ob sich überhaupt ein Kausalzusammenhang folgern lässt. Irgendeine Metrik kann beispielsweise andere Metriken kausal beeinflussen oder mehrere Metriken von einer anderen Größe kausal abhängen. Da im Rahmen der Diplomarbeit das Thema *die Bestimmung der Korrelation zwischen den Messwerten* nicht zu meiner Aufgabenstellung gehört, wurde auf das Thema nicht eingegangen, sondern als Erweiterung angesehen.

1.2.1 Vorgehensweise

Es muss für jede Analyse die folgende Vorgehensweise erfüllt werden.

I. Bestimmung der Metriken bei einer vorgegebenen Analyse:

Diejenigen, die eine vorgebebene Analyse untersuchen möchten, müssen im ersten Schritt die Metriken bestimmen. Metriken sind die Faktoren, die für diese Analyse eine relevante Information enthalten. Diese Metriken können Einfluss auf die Zeitkomplexität oder den Speicherbedarf haben. Im Rahmen dieser Diplomarbeit wird Static Single Assignment (SSA) in Betracht gezogen.

II. Berechnung der Messwerte:

Nachdem die Metriken für diese Analysen bestimmt sind, sollen sie aus der entsprechenden IML Datei gemessen werden. Aus diesem Grund sollen die Module für die Messung implementiert werden.

III. Speicherung dieser Messwerte:

Wenn die ersten beiden Schritte implementiert sind, soll die Implementierung bzw Änderung im Tool relevant_stats erfolgen, damit die Messwerte in die Datenbank gespeichert werden können.

Nachdem die Messwerte in die Datenbank gespeichert worden sind, ist das Tool relevant_stats in der Lage weitere Datenbankoperationen wie List, Update oder Delete durchzuführen und die graphische Darstellung zu erzeugen.

1.3 Inhalt des Dokuments

Nach dieser Einführung wird in Kapitel 2 auf die Analysen eingegangen. In diesem Kapitel wird die SSA Analyse und die dazugehörigen Metriken diskutiert. In Kapitel 3 wird die Architektur und die wichtige Komponente des Tools relevant_stats beschrieben. In Kapitel 4 werden schließlich die Erweiterungen an relevant_stats diskutiert und eine Vorgehensweise angegeben, wie eine neue Analyse in relevant_stats hinzugefügt werden kann. Das Kapitel 5 diskutiert, was in dieser Arbeit erreicht wurde, und welche Aufgaben in der Zukunft noch gelöst werden müssen.

2 Analysen in Bauhaus

In Bauhaus werden die Analysen durchgeführt, um die Kontroll- Datenflussinformationen zu erstellen. Diese Analysen werden in Bauhaus beispielsweise für das Auffinden von nicht verwendeter Routinen, für Erzeugung von Kontrolflussinformationen, für das Auffinden von Seiteneffekten und die Verwendung nicht initialisierter Variablen benutzt. Um dies zu ermöglichen werden verschiedene Analysemethoden in Betracht gezogen und ein Tool für die entsprechende Analyse implementiert. Dieses Kapitel gibt eine kurze Einführung in die Analysen, die in Bauhaus existieren. In den nächsten Schritten wird anschließend auf die SSA Analyse und die Metriken der SSA eingegangen.

2.1 Einführung in die Analysen

In der Abbildung 2.1 wird dargestellt, welche Analysen in welcher Reihenfolge in Bauhaus durchgeführt werden. Jede Analyse enthält die Informationen von den davor durchgeführten Analysen. Das bedeutet beispielsweise, dass während die SSA Analyse läuft, Kontrollflussinformationen zur Verfügung stehen. Dies hat zur Folge, dass die Analysen nachfolgende Analyse beschleunigen können.

Die Analysen in Bauhaus sind statische Analysen. Sie werden vor dem Ausführen eines Programms durchgeführt, und grenzen sich damit von den dynamischen Analysen die während eines Programmablaufs durchgeführt werden, ab. In den folgenden Schritten wird auf jede einzelne Analyse eingegangen.

2.1.1 Zeigeranalyse in Bauhaus

Ziel der allgemeinen Zeigeranalysen ist:

- Worauf kann ein Zeiger an einem bestimmten Punkt zeigen.

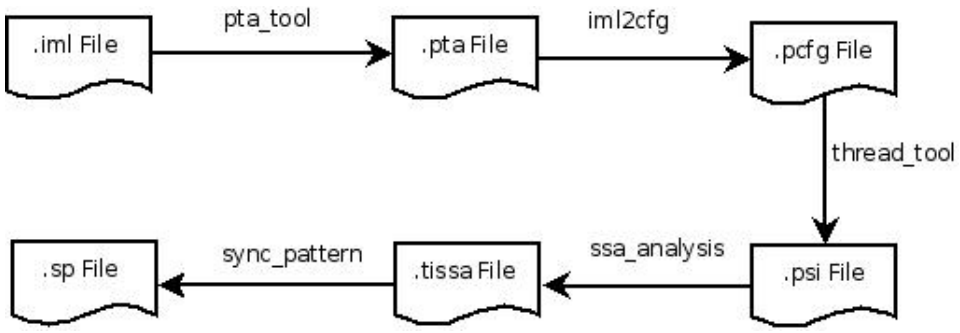


Abbildung 2.1: Analysen in Bauhaus

Die Antwort auf diese Frage ist die Aufgabe der Zeigeranalysen. Daher kann eine weitere Frage "welchen Wert kann ein bestimmter Ausdruck annehmen?" für Zeiger erst dann beantwortet werden, wenn die erste Frage gelöst werden kann. Zeigeranalysen werden angewendet, um präzise Datenflussergebnisse herzuleiten, und um die Optimierungen durchzuführen (beispielsweise Verzicht auf NULL-Zeiger, Prüfungen oder Code-Verschiebung) [rnEP09].

Es genügt nicht, dass die einfachen Zeiger beachtet werden, da alle Zeigerpfade beachtet werden müssen. Falls der Zeiger X beispielsweise auf Zeiger Y zeigt, der auf andere Objekte oder Zeiger zeigt, hat X auch Verweis auf diese Objekte bzw. Zeiger. Diese Transitivität bringt einige Schwierigkeiten mit sich. Da Zeigeranalysen so wichtig sind, wurde viel darüber geforscht. Es gibt sehr unterschiedliche Verfahren wie Steensgrad, Das, Andersen, Wilson und noch viele weitere. Der *worst case* Fall von Zeigeranalyse ist "Alle Zeiger können auf alles zeigen". Das Verhalten ist im Normalfall aber stark unterschiedlich als *worst case*.

Zeiger Analysen können in folgende Teilen klassifiziert werden:

- **Fluss-Sensitivität:** Wird die Reihenfolge der Anweisungen beachtet?
- **Struktur-Sensitivität:** Wie werden Felder von Strukturen beachtet?
- **Kontext-Sensitivität:** Wie werden die Aufrufpfade beachtet?

Das Tool pta_tool ermöglicht Fluss-Insensetive Zeigeranalyse in Bauhaus Projekt. Die Abbildung 2.2 zeigt ein Beispiel, um den Unterschied zwischen Fluss-Sensitiver und Fluss-Insensitiver Zeigeranalyse zu veranschaulichen. Die Frage ist, worauf der Zeiger p zeigt, während die Funktion CallMe aufgerufen wird. Falls es um die Fluss-Sensitive Zeigeranalyse geht, beachtet die Zeigeranalyse die Reihenfolge und erkennt, dass der Zeiger nur das Ziel a hat, während p bei der Fluss-Insensitive Zeigeranalyse die Ziele a und b hat, da die Fluss-Insensitive Analyse alle Zuweisungen an p betrachtet.

```
int a;
int b;

p = &a;
CallMe(p);
p = &b;
```

Abbildung 2.2: Beispielcode für Fluss-Sensitivität

2.1.2 Kontrollflussanalyse in Bauhaus

Ziel der Kontrollflussanalyse ist herauszufinden, von welchem Block im Programm die Kontrolle an welchen Block übergeben wird, und welche Funktionen dabei erreichbar sind. Des Weiteren überprüft die Kontrollflussanalyse welche Werte für eine Funktion möglicherweise zur Ausführung kommen [Her09].

Zur Analyse des Kontrollflusses wird ein gerichteter Graph G(N,E) mit zwei ausgezeichneten Knoten, Entry und Exit, erstellt. Die Knoten im Graph repräsentieren die Grundblöcke also Folgen von Anweisungen, in die hinein es keine Sprünge gibt und die nur sequentiellen Kontrollfluss besitzen, während eine Kante aus der Menge der Kanten den Kontrollfluss zwischen zwei Anweisungen beschreibt. Die ausgezeichneten Knoten Entry und Exit stellen die Anfangs- und Endanweisung eines Programms dar. In der Abbildung 2.3 wird zu einem gegebenen Programmstück der zugehörige Kontrollflussgraph dargestellt.

Das Tool iml2cfg wandelt das IML-Format in ein Kontrollfluss-Format um. Währenddessen werden auch der Dominanzbaum und die Dominanzgrenzen bestimmt. Durch iml2cfg gibt es zusätzlich folgende Möglichkeiten:

- Die Berechnung, wie oft eine Funktion aufgerufen wird
- Existiert ein Block, der nicht erreichbar ist. Die Möglichkeit zum Löschen und zu der Ausgabe solche Knoten. (durch -remove_unreachable bzw. -output_unreachable).

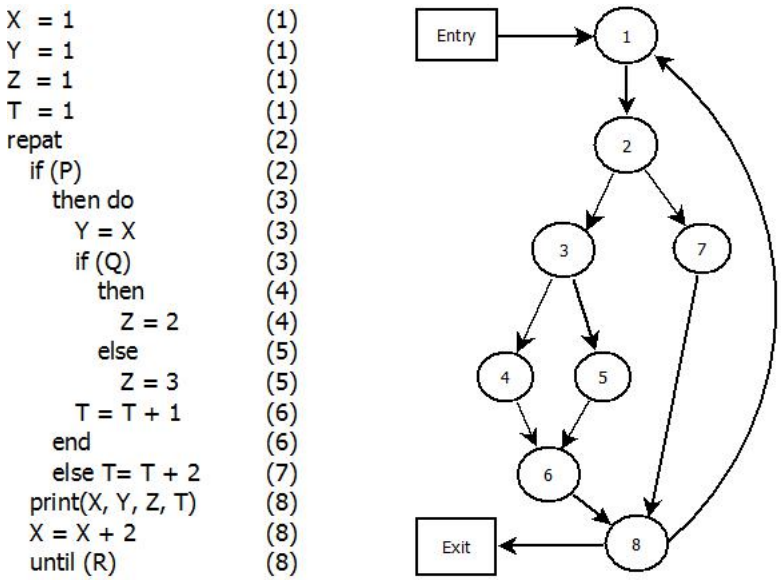


Abbildung 2.3: Beispielprogramm und zugehöriger Kontrollflussgraph(CFG)

2.1.3 SSA Form in Bauhaus

Static Single Assignment Form wird durch das Tool ssa_analysis realisiert. Mehr zum Thema SSA-Form wird in dem nächsten Kapitel erläutert.

2.1.4 IML Analyse nebenläufiger Programme

Durch Threads können Programmabläufe parallelisiert werden. Die Parallelisierung kann zu Performanceverbesserungen führen. Es gibt verschiedene Ansätze, um die Parallelisierung zu ermöglichen. Low-Level Sprachen wie C benötigen für die Erstellung und Manipulation von Threads, eine zusätzliche Schnittstelle wie *pthreads* [But97]. Höhere Programmiersprachen hingegen bieten von sich aus an, wie sie mit mehreren Threads

umgehen. Java ermöglicht es durch syncronized-Blöcke die Shared Memory zu schützen und Methoden threadssicher zu machen [BGL06]. Ada bietet mit seinem Task die parallele Ausführung mehrerer Threads. Wenn die mehrere Threads auf die gemeinsamen Daten zugreifen sollten, kann mit Hilfe von Protected Objects (wie protected function, protected procedure) Synchronisation gewährleistet werden. [Bar06].

Durch die parallele Programmierung mit Threads entstehen, aber auch Probleme, wie Deadlocks und schwer zu erkennende Race Conditions. Deadlocks treten auf, wenn zwei oder mehr Prozesse gegenseitig darauf warten, dass eine gemeinsam gebrauchte Ressource freigegeben wird. Dadurch bleiben beide Prozesse in einer endlosen Wartestellung und kein Prozess kann jemals auf die Ressource zugreifen [Sin89]. Um einen Deadlock anschaulich zu erläutern verwendet man in der Grundlagenliteratur meist das Beispiel der denkenden und essenden Philosophen.

Mehrere Philosophen verbringen ihre Zeit mit Denken und Essen. Sie sitzen an einem runden Tisch, jeder Philosoph hat einen Teller vor sich, und zwischen den Tellern liegt eine Gabel. Wenn ein Philosoph essen will, benötigt er zwei Gabeln. Da er sich jedoch beide mit seinem Nachbarn teilen muss, können nie zwei benachbart sitzende Philosophen gleichzeitig essen. Während die Philosophen denken, sind sie ausschließlich mit sich selbst beschäftigt und stehen nicht in Kontakt mit ihrer Umwelt. Wenn sie jedoch vom vielen Denken hungrig geworden sind, müssen sie versuchen, zwei Gabeln zu bekommen, um dann zu essen. Damit das Essen gerecht verteilt wird und kein Philosoph verhungert, muss man sich eine Regelung für die Vergabe des Bestecks einfallen lassen.

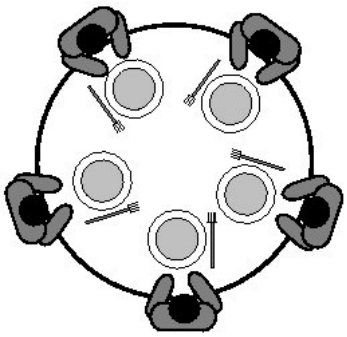


Abbildung 2.4: Das Philosophen-Problem [SM99]

Man könnte auf die Idee kommen, dass jeder Philosoph zunächst das Besteck zu seiner Rechten ergreift, dieses also in Besitz nimmt, und dann darauf wartet, dass das entsprechende andere Teil zu seiner Linken frei ist. Wenn dabei zufällig alle Philosophen gleichzeitig hungrig werden, dann nimmt jeder das Teil zu seiner Rechten in Besitz. Damit hält jeder eine Gabel, und auf dem Tisch ist kein Besteck mehr übrig. Alle Philosophen warten gegenseitig darauf, dass ein anderer sein Besteck wieder ablegt. In diesem Zustand würden alle Philosophen ewig warten und so verhungern. Man nennt diesen Zustand Deadlock.[SM99]

Race conditions entstehen wenn mehrere Prozesse/Threads auf gemeinsamen Speicher zugreifen, und deren Ergebnisse davon abhängen, in welcher Reihenfolge die Operationen der einzelnen Prozesse abgearbeitet werden [NM92].

In Bauhaus gibt es zwei Analysen, thread_tool und sync_pattern, um die nebenläufigen Programme zu analysieren. Im Vergleich zu thread_tool bekommt sync_pattern als Eingabe die durch ssa_analyse in Bauhaus erzeugte SSA-Form.

2.2 Static Single Assignment Form (SSA)

In optimierenden Compilern hat die Entscheidung von Datenstruktur Einfluss auf die Leistung und Effizienz der Programmoptimierung. Eine schlechte Wahl kann die Optimierung verhindern und dadurch die Kompilierung verlangsamen. Def-Use-Informationen werden als Basis vieler Analyse- und Optimierungstechniken verwendet, um die Programme zu verifizieren und Fehler zu erkennen. Moderne Compiler nutzen die SSA Form als kompakte Alternative zu den klassischen Informationsketten. Ein Quellcode steht in SSA Form wenn jede Variable im ganzen Programm nur einmal einen Wert zugewiesen bekommt. Dadurch erhalt man das Verhalten, was dieser Form ihren Namen gab - Static Single Assignment Form. Die Einmalzuweisung bezieht sich darauf, dass jede Variable einen Index bekommt, der bei jeder Definition inkrementiert wird. Für folgende Verwendung gilt die Variable mit dem neuen Index.

Der von Cytron vorgeschlagene Ansatz bezieht sich auf Straight-Line-Code. Straight Line Code ist eine reduzierte Sprache, die nur einfache Zuweisungen, Bedingungen und

die While-Schleife kennt. In Abbildung 2.5 wird ein Beispiel zu dieser einfachen Sprache dargestellt.

```
X = 3;              X1 = 3;
  = X + 3;             = X1 + 3;
X = 4;              X2 = 4;
  = X + 4;             = X2 + 4;
```

Abbildung 2.5: Straight-Line-Code und seine SSA-Version

Die Static Single Assignment Form in Bauhaus besteht aus drei Phasen. Während die SSA Analyse in Bauhaus durchgeführt wird, hat die entsprechende IML-Datei den Kontrollflussgraph zur Verfügung, was in der Kontrollflussanalyse durch das Tool iml2cfg erzeugt wurde. Das bedeutet, dass die SSA Analyse keinen zusätzlichen Aufwand zur Erstellung des Kontrollflussgraphes hat. Die Abbildung 2.6 zeigt ein Beispiel für die Phasen der SSA Analyse.

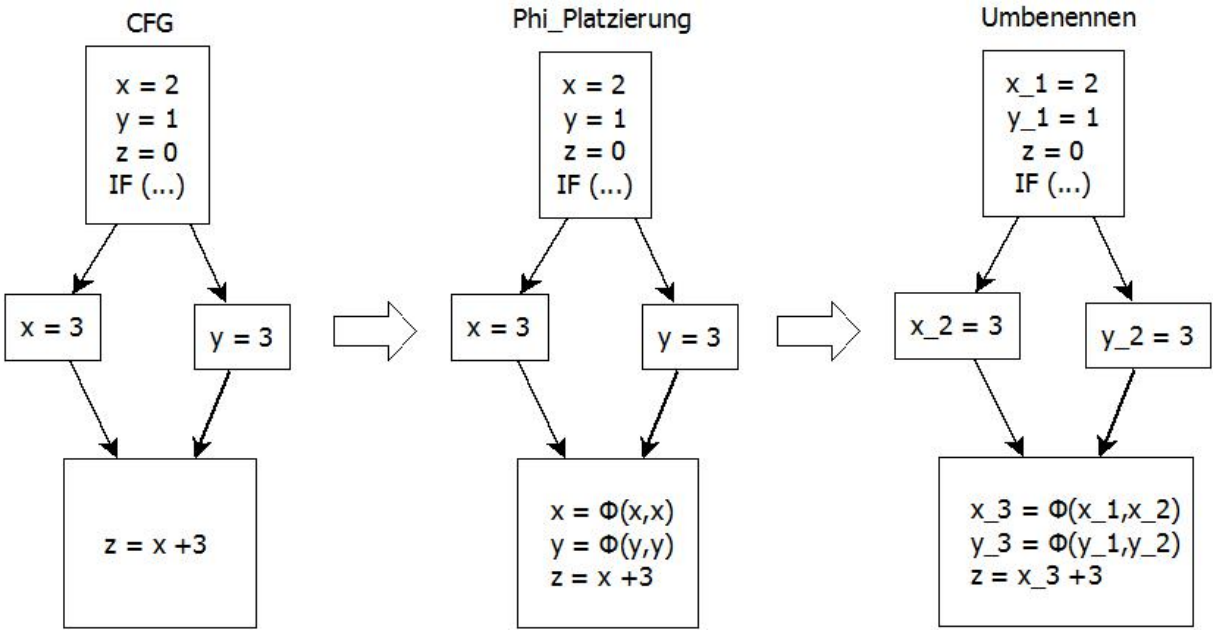


Abbildung 2.6: Phasen der SSA Analyse

In der ersten Phase werden die Punkte im Kontrollflussgraph bestimmt, an denen der Kontrollfluss von verschiedenen Pfaden zusammenkommt und mindestens zwei Definitionen hat. Phi Knoten werden entsprechend der SSA-Art an diesen Punkten platziert. Wenn es beispielsweise um minimale SSA geht, werden sie an jedem Punkt platziert, während bei der pruned-SSA untersucht wird, ob die Variable noch tiefer verwendet wird. Danach werden in der zweiten Phase die Variablen umbenannt, damit sie exakt einen Definitionspunkt haben und die Verwendung dieser Variablen wird entsprechend neuer Namen aktualisiert. Die ersten beiden Phasen der SSA Analyse in Bauhaus sind nach [RCDU91] implementiert. Zusätzlich werden in der dritten Phase die Seiteneffekte aktualisiert, was nicht in der Literatur [RCDU91] steht. Die Algorithmen zu diesen Phasen werden in den nächsten Kapiteln detailliert erläutert.

Im Rahmen dieser Diplomarbeit wurde die SSA Analyse, die In Bauhaus implementiert wurde, tiefer untersucht. Zunächst möchte ich den Begriff *Locator* erläutern, die häufig in dieser Ausarbeitung vorkommt. Dann werden die wichtige Komponente der SSA in folgenden Kapiteln dargestellt. Daraufhin werden die einzelne Phasen und die Metriken der SSA Analyse vorgestellt.

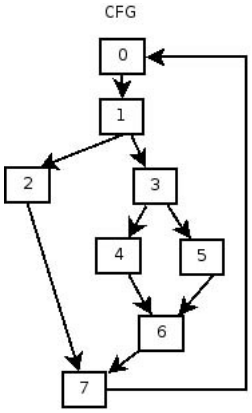
Ein Lokator ist eine abstrakte Variable zur Darstellung von Datenflussinformationen und kann damit als ein atomares Objekt angesehen werden, das nicht aus weiteren Teilen besteht und zur Laufzeit einen Wert enthält. Bei einem Lokator kann es sich beispielsweise um eine vollständige globale Variable oder nur um einen bestimmten Teil davon oder um Teile von lokalen Variablen, formalen Parametern oder Objekten auf der Halde handeln.

2.2.1 Dominanz

Neue Algorithmen für verschiedene Probleme bei der Optimierung und Codegenerierung haben das Konzept Dominanz benutzt. [KDCK]. Durch die Entdeckung der SSA Form hat man mehr Interesse an Dominanz und verwandte Konzepte. Die Antwort auf so eine Frage "Wo soll ich PHI-Knoten platzieren?", kann man mit Hilfe vom Dominanzkonzept bestimmen. Das Konzept gibt die Möglichkeit, die Beziehung zwischen den Knoten im Kontrollflussgraph zu bestimmen. In der Literatur von [RCDU91] wurde diese Beziehung für die Phi-Platzierung und Umbenennung verwendet. In dieser Ausarbeitung werden Dominanz und seine verwandte Konzepte Dominanzbaum und Dominanzgrenze vorgestellt.

Die Algorithmen, die in SSA Form verwendet wurden, brauchen sowohl den Dominanzbaum als auch die Dominanzgrenze. Zuerst werden jedoch Dominator für jeden Block bestimmt. Mit Hilfe von diesen Informationen kann dann der Dominanzbaum erstellt werden. Zuletzt werden Dominanzgrenzen für jeden Block durch Dominanzbaum und Kontrollflussgraph bestimmt.

Gegeben sei ein Kontrollflussgraph G =(V, E, r), ein gerichteter Graph mit Knotenmenge V und Kantenmenge E. Der Knoten r sei Wurzelknoten und r $\in$ V. Ein Knoten u **dominiert** einen Knoten v, wenn u auf allen Pfaden von r zu v liegt. u wird dann **Dominator** von v genannt. Jeder Knoten im Flussgraph besitzt mindestens einen Dominator. Beispielsweise dominiert der Kopf einer Schleife den Rumpf, wenn es nur einen Eintritt in die Schleife gibt [rnEP09]. Falls u ein Dominator von v ist, schreiben wir u dom v (bzw. $u \gg v$). Falls u ein Dominator von v und u $\neq$ v ist, wird es als strikte Dominanz bezeichnet. In der Abbildung 2.7 werden für jeden Block in einem gegebenen Kontrollflussgraph alle Dominatoren dargestellt.



Node	Domiantors
0	{0}
1	{0,1}
2	{0,1,2}
3	{0,1,3}
4	{0,1,3,4}
5	{0,1,3,5}
6	{0,1,3,6}
7	{0,1,7}

Abbildung 2.7: Beispiel für Dominator

Die Dominanzrelation ist also eine Halbordnung, weil die Relation:

- **reflexiv:** Jeder dominiert sich selbst.
- **antisymmetrisch:** u dom v und v dom u $\rightarrow$ u = v.

- **transitiv:** u dom v und v dom w → u dom w. D.h. Jeder Dominator eines Dominators von v dominiert selbst auch v.

ist. Die Menge der Dominatoren von u wird mit D(u) bezeichnet. Dann gilt:

$$\forall u \in D(v),\ u \neq v:\ D(u) \subset D(v). \tag{2.1}$$

Das bedeutet beispielsweise, dass der Knoten 7 in der Abbildung 2.7 von den Knoten dominiert wird, welche Knoten 1 dominieren, weil der Knoten 1 den Knoten 7 dominiert. Es gibt in jeder Dominanzmenge D(v) einen besonderen Dominator. Derjenige Knoten, der gewissermaßen von allen Dominatoren am nächsten an v liegt, nennt sich **direkter Dominator**. Er wird als idom(v) geschrieben.

Das Problem, wie alle Dominatoren für jeden Block im Kontrollflussgraph aufgefunden werden sollen, hat eine lange Geschichte in der Literatur. Prosser erläutert den Begriff Dominator im Jahre 1959 in einer Arbeit zur Analyse von Flussdiagrammen, definierte sie als wie folgt:

"We say box i dominates box j if every path (leading from input to output through the diagram) which passes through box j must also pass through box i. Thus box i dominates box j if box j is subordinate to box i in the program" [Pro59].

Zehn Jahre später zeigten Lowry und Medlock einen Algorithmus zur Berechnung des Dominators [LM69]. Im Jahre 1972 haben Allen und Cocke eine iterative Lösung für dieses Problem gefunden, die $O(N^2)$ Komplexität hat [AC72]. Drei Jahre später veröffentlichte Hecht und Ullman eine Analyse der iterativen Algorithmen mittels Reverse Postorder Traversierungen. Sie zeigten, dass die Dominanz Gleichungen in linearer Zeit auf reduzierbaren Graphen gelöst werden können [HU75].

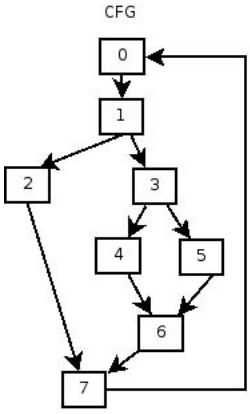
2.2.1.1 Dominanzbaum

Direkte Dominanz bildet einen Baum, den sogenannten Dominanzbaum, wobei Vater von einem Knoten v idom(v) ist. Die Wurzel von Dominanzbaum ist gleich der Wurzel des Kontrollflussgraphes. Der Dominanzbaum hat genau die gleiche Menge von Knoten wie der Kontrollflussgraph, aber dagegen eine sehr unterschiedliche Menge von Kanten.

In dieser Ausarbeitung beziehen sich die Wörter *predecessor, successor* auf den Kontrollflussgraph und die Wörter *parent, child* auf den Dominanzbaum.

Der Pfad von v zur Wurzel im Dominanzbaum liefert alle Dominatoren von v und der Teilbaum unter v enthält alle Knoten, die von v dominiert werden. In der Abbildung 2.8 gibt es zu einem gegebenen Kontrollflussgraph eine Darstellung zu dem dazugehorigen Dominanzbaum. Die Knoten 0,1,7 dominieren beispielsweise den Knoten 7, weil sie auf den Pfad von Knoten 7 zur Wurzel im Dominanzbaum sind. Der Knoten 1 dominiert ebenso alle Knoten (3,4,5,6) im Teilbaum, welcher wiederum den Wurzelknoten 3 enthält.

Der Dominanzbaum kann mit einem top-down-Aufbau erstellt werden. Dafür werden die Knoten des Graphen in Postorder-Reihenfolge nummeriert. Die Knoten, die im Dominanzbaum weiter oben liegen, haben eine größere Postorder-Nummer. Für einen top-down Aufbau des Dominanzbaumes werden die Knoten gemäß absteigenden Postorder-Nummern traversiert [rnEP09].



Node	Domiantors
0	{0}
1	{0,1}
2	{0,1,2}
3	{0,1,3}
4	{0,1,3,4}
5	{0,1,3,5}
6	{0,1,3,6}
7	{0,1,7}

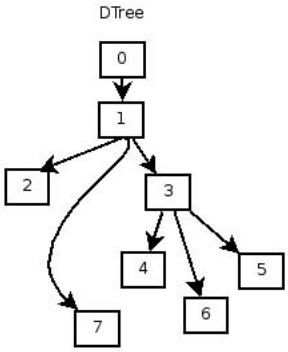


Abbildung 2.8: Beispiel für ein Dominanzbaum

2.2.1.2 Dominanzgrenze

Der Begriff Dominanzgrenze spielt eine wichtige Rolle in der SSA Form. Es wird durch die Dominanzgrenzen bestimmt, an welchen Punkten die Phi-Knoten platziert werden. Es seien X und Y Knoten im Kontrollflussgraph. Y ist in der Dominanzgrenze von X,

falls Y der Knoten ist, der von X nicht strikt dominiert wird, aber dessen Vorgänger von X dominiert werden.

$$DF(X) = \{\ Y \mid (\exists P \in Pred(Y))\ (X \in D(P)\ and\ X \notin D(Y)\ )\} \tag{2.2}$$

Der Knoten 7 in der Abbildung 2.8 liegt beispielsweise in der Dominanzgrenze vom Knoten 3, weil der Knoten 3 den Knoten 7 nicht dominiert, während der Vorgängerknoten 6 (Vorgänger von 7) vom Knoten 3 dominiert wird.

Die Anwendung der Dominanzgrenzen liegt nun darin, dass sie für Definitionen eine interessante Eigenschaft haben. Verlässt man den Dominanzbereich (also an der Dominanzgrenze), so kann dies nur an solchen Stellen im Kontrollfluss passieren, an denen mindestens eine weitere Definition diesen Punkt erreicht. Dies sind aber genau die Stellen an denen ϕ Knoten platziert werden sollen. [Bis08] Eine direkte Berechnung von DF(X) ist in O(N^2) möglich, effizientere Verfahren laufen aber bereits in O(E * log N) [VS95]. In dieser Ausarbeitung werden erst der Algorithmus von Cooper, Harvey und Kennedy zur Berechnung der Dominanzgrenze [PB99], und danach der zweite Ansatz von Cytron [RCDU91] näher betrachtet.

1 Berechnung der Dominanzgrenzen

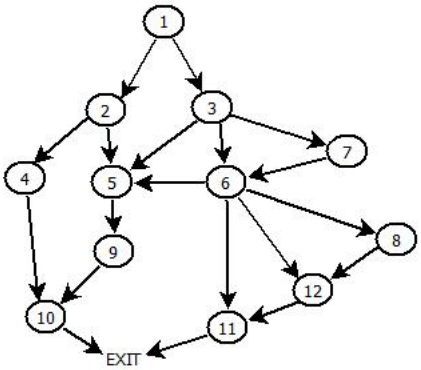
```
for all nodes y do
  if | pred(y) | ≥ 2 then
    for all x ∈ pred(y) do
      runner = x
      while runner ∉ Dominators(y) do
        DF(runner) = DF(runner) ∪ { y}
        runner = Dominators(runner)
      end while
    end for
  end if
end for
```

Der Algorithmus von Cooper, Harvey und Kennedy ist ein recht einfaches Verfahren und setzt den Dominanzbaum voraus, um den direkten Dominator zu jedem Knoten zu wissen. In der vorherigen Abbildung wurde der Algorithmus in Pseudocode dargestellt. In dem Algorithmus werden alle Knoten betrachtet, die den Eingangsgrad größer als

2 haben. Für diese Knoten wird bestimmt, zu welchen Dominanzgrenzen sie gehören. Dazu werden alle Knoten x $\in$ pred(y) betrachtet, wenn der Knoten y mindestens zwei Vorgänger hat. Es gibt zwei mögliche Fälle:

- Entweder x dominiert y
- Oder y liegt in der Dominanzgrenze von x also DF(x) = {y}

Damit ist klar, welche Knoten in DF(x) hinzugefügt werden sollen. Alle Vorgänger, die den Knoten y nicht dominieren und wiederum alle Knoten, die den Vorgängerknoten dominieren aber nicht y dominieren, enthalten den Knoten y in ihrer Dominanzmenge. Der Algorithmus steigt im Dominanzbaum ab jedem Vorgänger x auf und prüft stets, ob der aktuelle Knoten y dominiert. Der Aufstieg kann abbrechen, sobald der erste Knoten gefunden wurde, der y dominiert. Alle weiteren Knoten, die näher an der Wurzel sind, würden ebenfalls y dominieren. Dies bedeutet, dass nicht mehr Knoten als unbedingt nötig verglichen werden.



v	DF(v)
1	-
2	5,10
3	5,Exit
4	10
5	10
6	5,Exit
7	6
8	12
9	10
10	Exit
11	Exit
12	11
Exit	-

Abbildung 2.9: Beispiel-1 für die Dominanzgrenze

In der Abbildung 2.9 werden zu den Knoten im Graph die dazugehörigen Dominanzgrenzen dargestellt. Laut dem Algorithmus von Cooper, Harvey und Kennedy wird der Knoten 10 zunächst in die Dominanzgenze von seinen Vorgängern (die Knoten 4,9) hinzugefügt, weil diese den Knoten 10 nicht dominieren und danach wird der Knoten in die Dominanzgrenze von Knoten 5 (Vorgänger vom Knoten 9) und Knoten 2 (Vorgänger vom Knoten 4) hinzugefügt, weil sie den Knoten 10 nicht dominieren. Letztendlich wird der Algorithmus stoppen, weil der Vorgänger von Knoten 2 also der Knoten 1 den Knoten 10 dominiert. Dies bedeutet, dass die Dominanzgrenze von Knoten 1 nicht den

Knoten 10 enthält.

2 Berechnung der Dominanzgrenzen DF(v):

```
for all X in a bottom-up traversal of the dominator tree do
  DF(X) ← ∅
  for all Y ∈ Succ(X) do
    if idom(Y) ≠ X then
      DF(X) ← DF(X) ∪ {Y} { /* local */}
    end if
  end for
  for all Z ∈ Children(X) do
    for all Y ∈ DF(Z) do
      if idom(Y) ≠ X then
        DF(X) ← DF(X) ∪ {Y} { /* up */}
      end if
    end for
  end for
end for
```

Der andere Ansatz zur Berechnung der Dominanzgrenzen, der in dieser Arbeit vorgestellt wird, ist von [RCDU91]. Dieser Ansatz unterteilt die Dominanzgrenze zu einem Knoten mit zwei Komponenten.

$$DF(X) = DF_{\text{local}}(X) \ \cup \bigcup_{Z \in Children(X)} DF_{\text{up}}(Z) \tag{2.3}$$

$DF_{\text{local}}(X)$ enthält alle Nachfolgerknoten in Kontrollflussgraph, die von X nicht direkt dominiert sind. $DF_{\text{up}}(X)$ enthält dagegen alle Knoten, die in der Dominanzmenge von Kindknoten Z des Knotens X im Dominanzbaum, und nicht direkte Dominanz von X sind. Der Pseudocode vom Algorithmus wurde in der vorherigen Abbildung dargestellt und eben genannter Vorgang wird für jeden Knoten in der Bottom-Up Traversierung angewandt.

DF_local für einen Knoten X:

$$DF_{\text{local}}(X) = \{ \ Y \in Succ(X) \mid X \not\gg \ Y\} \tag{2.4}$$

DF_up für einen Knoten Z, der Kindknoten von X im Dominanzbaum ist:

$$DF_{\text{up}}(Z) = \{\ Y \in DF(Z) \mid idom(Z) \not\gg\ Y\} \tag{2.5}$$

Der Knoten 2 in der Abbildung 2.10 hat beispielsweise in seiner Dominanzgrenze die Knoten 5 und 10. Der Knoten 5 ist $DF_{\text{local}}(2)$, weil dieser Knoten erstens Nachfolgerknoten von 2 im Kontrollflussgraph ist und der Knoten 2 nicht der direkte Dominanz zu 5 ist. Der Knoten 10 ist $DF_{\text{up}}(4)$, weil der Knoten 4 Kindknoten von 2 im Dominanzbaum ist, der Knoten 10 in der Dominanzgrenze von 4 ist und der Knoten 2 den Knoten 10 nicht direkt dominiert.

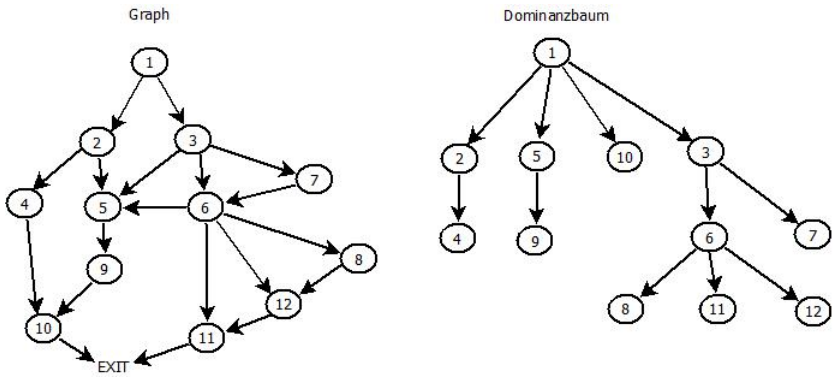


Abbildung 2.10: Beispiel-2 für die Dominanzgrenze

2.2.2 Phi Platzierung

Um die wichtige Metriken besser herauszufinden, wird zunächst vorgestellt, wie die Phi-Platzierung in Bauhaus funktioniert. Aus diesem Grund ist das Algorithmus für die Phi-Platzierung Gegenstand dieser Arbeit. Nach dieser Darstellung des Algorithmus in Bauhaus für die Phi-Platzierung, werden die einzelne Metriken in dem entsprechenden Kapitel näher untersucht. Der in Bauhaus implementierte Algorithmus für Phi-Platzierung (bzw. Umbenennung und Aktualisierung des Seiteneffekts) hat einige Faktoren, die einen wesentlichen Einfluss auf die Komplexität von diesen Algorithmen haben. Mit “Metriken” sind in dieser Ausarbeitung also eben diese Faktoren gemeint.

Die Phi-Platzierung in Bauhaus wurde genauso in dem in [RCDU91] vorgestelltem Algorithmus implementiert. Deswegen wird in dieser Ausarbeitung auf den Algorithmus von [RCDU91] eingegangen. Die Antwort auf die Frage, wo die Phi-Knoten platziert werden sollen, gibt Cytron folgende Bedingung:

"If two non-null paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ converge at a node Z, and nodes X and Y contain assignments to V (in the original program), then a trivial ϕ function $V \leftarrow \phi(V,\ldots V)$ has been inserted at Z (in the new program). " [RCDU91]

Nun platziert SSA Form an jedem Punkt einen Phi-Knoten, an dem der Kontrollfluss von verschiedenen Pfaden zusammenkommt und mindestens zwei Definitionen für eine Variable mitbringt, wenn es um *minimale SSA* geht. Es gibt aber noch eine andere Art von SSA Form (*pruned SSA*), die bei diese Punkten untersucht, ob eine Variable weiter benutzt wird, für die ein Kontrollfluss entsteht.

3 Phi Placement (pruned SSA):

```
for all Locator L do
  W ← { B | L ∈ Defs (B) }
  for all Block X ∈ W do
    compute DF(X)
    for all Block Y ∈ DF(X) do
      if L live-in at Y then
        insert Phi for L at Y
      end if
    end for
  end for
end for
```

Der Algorithmus für die Platzierung, der in Bauhaus implementiert ist, läuft für alle Lokator durch und fügt jeden Block in eine Worklist, der eine Definition mit diesem Lokator hat. Nachdem die Worklist gebildet wurde, läuft die innere Schleife für jeden Block X durch, das in der Worklist steht. Für diesen Block X wird seine Dominanzgrenze bestimmt und anschließend in allen Blöcke ein Phi-Knoten hinzugefügt, die in dieser Dominanzgrenze von Y stehen. Der Algorithmus fügt den Phi-Knoten ohne zusätzliche Überprüfung hinzu, falls es um die *minimale SSA Form* geht. Bei *pruned SSA* wird noch überprüft, ob der Lokator weiter benutzt wird(Live-at Block Y). In [RCDU91] wurde im

Algorithmus anstatt des Lokators nur lokale Variablen benutzt. Dies bedeutet, dass der Algorithmus nur für alle lokalen Variablen durchläuft.

In Bauhaus sind die Aufgaben in verschiedenen Analysen verteilt. Dies sehen wir ebenfalls in der SSA Analyse. Das Tool ssa_analysis braucht beispielsweise keinen zusätzlichen Aufwand, um die Dominanzgrenze von jedem Knoten zu berechnen, weil diese Information während der Erzeugung des Kontrollflussgraphs durch das Tool iml2cfg bereitgestellt wird. Aus diesem Grund beachten wir dies während der Auswahl der SSA Metriken. Damit ist die Berechnung der Dominanzgrenze Gegenstand der Kontrollflussanalyse.

Wir betrachten erneut die Abbildung 2.3. Ein Phi-Knoten soll für die Variable Y im Block 8 hinzugefügt werden, weil der Block 3 eine Zuweisung an die Variable Y hat und der Block 8 in der Dominanzmenge vom Block 3 steht. In den Blöcke 4 und 5 wurde die Variable Z zugewiesen und DF(4) und DF(5) enthalten den Block 6. Das bedeutet, Block 6 wird in der Theorie zwei Mal als gleicher Phi-Knoten (wegen 4 und 5) hinzugefügt(nicht jedoch in der Praxis). Der Algorithmus für die Phi-Platzierung hat nämlich ein Array von Flag für jeden Knoten (bzw. Block), um so festzustellen, ob der Phi-Knoten schon hinzugefügt wurde. Dadurch wird festgestellt, falls ein Phi-Knoten für die Variable Z im Block 6 einmal beispielsweise wegen Block 4 platziert wird, dass kein Phi-Knoten mehr für diese Variable wegen Block 5 nicht mehr notwendig ist, weil schon existiert.

2.2.3 Umbenennung

Nachdem die Phi-Knoten hinzugefügt wurden, müssen im zweiten Schritt die Variablen umbenannt werden, damit die Variablen im ganzen Programm nur einmal einen Wert zugewiesen bekommen. Dadurch wird die SSA Eigenschaft erfüllt. Der Algorithmus zur Umbenennung in Bauhaus wurde wieder nach [RCDU91] implementiert. Der einzige Unterschied ist, dass im Bauhaus die Umbenennung der Varaiblen durch Setzen von Datenflußkanten realisiert wird, während diese Variablen in [RCDU91] durch ihren neuen Namen direkt umbenannt werden. Da das Bauhaus und [RCDU91] sich nur im Hinblick auf die Umbenennung der Variablen unterscheiden, wird in diesem Kapitel der Algorithmus zur Umbenennung nach [RCDU91] erläutert.

4 Umbennenung:

```
for all variable V do
  C(V) ← 0
  S(V) ← EmptyStack
end for
call SEARCH (Entry)
```

Dieser Algorithmus setzt den Kontrollflussgraph und den Dominanzbaum voraus und hat zwei arrays S(*) und C(*). S(*) ist ein array von Stacks. D.h. für jede Variable gibt es ein Stack und C(*) ist dagegen ein array von Integer. Das wiederum bedeutet, dass für jede Variable ein Counter existiert. C(V) zeigt, wie viele Zuweisungen für eine Variable V existieren. Der Algorithmus initialisiert erst die beiden Arrays und beim Aufruf der Funktion *SEARCH* wird Entry-Block übergeben.

5 Search(part1):

```
for all statement A in X do
  if A is an ordinary assignment then
    for all variable V used In RHS(A) do
      get i from Stack S(V)
      replace use of V by use of Vi
    end for
  end if
  for all V in LHS(A) do
    i ← C(V)
    replace V by new Vi in LHS(A)
    push i onto S(V)
    C(V) ← i + 1
  end for
end for
```

In der Funktion *SEARCH* werden zuerst alle Zuweisungen im gegebenen Block durchgelaufen. Die Variablen, die auf der rechten Seite einer Zuweisung stehen, werden durch einen neuen Namen ersetzt. Neue Namen bekommen die Variablen durch ihren eigenen Stack. Dann werden die Variablen, denen ein Wert zugewiesen wird, durch ihren neuen Namen ersetzt.

Nachdem alle Zuweisungen bearbeitet worden sind, wird in den Nachfolgeblöcken im Kontrolflussgraph betrachtet, ob die Blöcke einen Phi-Knoten für diese neue benannte Variable auf der rechten Seite (RHS) enthalten. Falls ja, werden diese Variablen im Phi-Knoten durch ihren neuen Namen ersetzt. Wenn das erreicht ist, ist die Umbenennung für diesen Block abgeschlossen. Dieser Vorgang muss aber für jeden Block durchgeführt werden. Anschließend wird die Funktion *SEARCH* rekursiv für alle Kinderblöcke im Dominanzbaum aufgerufen.

6 Search(part2):

```
for all Y ∈ SUCC(X) do
  j ← WhichPred(Y,X)
  for all φ function F in Y do
    get i from Stack S(V)
    replace the j-th operand V in RHS(F) by Vi
  end for
end for
for all Y ∈ Children(X) do
  call SEARCH(Y)
end for
```

2.2.4 Seiteneffekt

Der letzte Schritt, der in der SSA Analyse durchgeführt wird, ist die Aktualisierung der Seiteneffekte, falls sie existieren. Seiteneffekte hängen möglicherweise vom Aufruf-Kontext ab. Wenn eine Funktion eine Variable verändert (direkt oder über ein Alias), die nach Beendigung der Funktion anderswo noch gelesen werden kann, so geht es um einen modifizierenden Seiteneffekt, während eine Funktion eine Variable liest (direkt oder über ein Alias), die nicht zuvor von der Funktion selbst verändert wurde, so geht es um einen lesenden Seiteneffekt [rnEP09]. Jede Routine kann Seiteneffekte produzieren.

Die Folgende Zugriffe auf Lokatoren werden beim Aufbau einer SSA Form unterschieden:

- **Must-Def**: Der Wert eines Lokators wird in jedem Fall vollständig überschrieben.

- **May-Def**: Der Lokator wird potentiell oder unvollständig überschrieben.
- **May-Use**: Der Wert eines Lokators wird potentiell oder garantiert ganz oder nur teilweise gelesen.

Zur Behandlung der Seiteneffekte von Aufrufen in Bauhaus werden die Werte vom Lokator zwischen der aufrufenden und aufgerufenen Routine ausgetauscht. Die Werte von Lokatoren der aufrufenden Routine werden in dem ersten Schritt in die aufgerufene kopiert, bevor in die aufgerufene Routine eingesprungen wird. Um diesen Vorgang in Bauhaus zu realisieren wurde zwei Knoten eingeführt. Diese Knoten sind *Link_ In_ Definition und Link_ Out_ Uses*.

Im Bauhaus gibt es eine Lokatorabbildung, die eine Abbildung zwischen Lokatormengen der aufrufenden und der aufgerufenen Routine bei einem Routinenaufruf beschreibt. Es handelt sich dabei im Prinzip um einen Copy-In und Copy-Out Parameterübergabemechanismus, der die in den Lokatoren gespeicherten Werte vor dem Eintritt in die Routine überträgt und nach der Ausführung die Werte der geänderten Lokatoren zurückschreibt. Bei einem Copy-In Parameter kann es sich beispielsweise um eine globale Variable oder nur um einen bestimmten Teil davon oder um Teile von lokalen Variablen oder Objekten auf der Halde handeln.

Link_ In_ Definition repräsentiert den formalen Copy-In Parameter. Dieser Knoten wird zu Beginn der Entry-Blocks von der aufgerufenen Routine in den Kontrolflussgraphen platziert. *Link_ Out_ Uses* repräsentiert dagegen den formalen Copy-Out Paramater und befindet sich am Ende des Exit-Blocks der aufgerufenen Routine im Kontrolflussgraph. Die beiden Knoten verweisen auf die Definitionstabelle, die Definitionen der Lokatoren enthält, die potentiell von der aufrufenden Routine gelesen werden.

Zu Beginn der SSA Analyse steht keine Information über Must-Def oder May-Def zur Verfügung. Deshalb wird es davon ausgegangen, dass durch die nicht-lokaler Lokatoren immer einen potentiellen Seiteneffekt entsteht. Deswegen werden zuerst als May-Def markiert. Es wird aber später *Link_ Out_ Uses* untersucht, ob es sich bei diesen Lokatoren um ein Must-Def handelt.

2.2.5 Metriken der SSA Analyse

In diesem Kapitel werden die ausgewählten Metriken der SSA Form vorgestellt. Diese Metriken sind:

- Die Anzahl der Pattern
- Die Anzahl der Lokatoren
- Die Anzahl der Blöcke
- Die Anzahl der Zuweisungen
- Die gesammte Anzahl der LHS (left-hand side) und RHS (right-hand side)
- Der maximale Ausgangsgrad im CFG
- Die Anzahl der PHI Funktionen
- Die Tiefe des Dominanzbaumes
- Die Anzahl der Variablen in Link_In_Definitionen
- Die Anzahl der Variablen in Link_Out_Uses

Diese Metriken sollen für jede Routine berechnet und in die Datenbank gespeichert werden. Im folgenden Teil wird ausdrücklich erläutert und begründet, warum diese Metriken ausgewählt worden sind.

```
Pattern_Iterator := Routines.Make_Iterator_For_Patterns (Routine);
while Storables.Lists.More (Pattern_Iterator) loop
  Storables.Lists.Next (Pattern_Iterator,
                        Storables.Storable (Pattern));
  Pattern_Number := Pattern_Number + 1;
  Build_SSA_For_Routine
        (The_System, Routine,Pattern, Pattern_Number,
         Calc_Must_Defs, Semi_Pruning, Routine_Changed);
  Changed := Changed or else Routine_Changed;
end loop;
```

Abbildung 2.11: Build-SSA

Das Tool ssa_analysis beginnt mit der Post-Order Traversierung, um die SSA Analyse für die Routinen aller Threads zu berechnen. In dieser Traversierung wird SSA zuerst die aufgerufene Routine (Callees) berechnen. Für Jede Routine werden die Phasen der SSA Form sowie die Anzahl der Pattern durchgeführt (siehe Abbildung 2.11). Die Anzahl der Pattern bezieht sich darauf, wie oft die Routine aufgerufen werden. Aus diesem Grund ist **die Anzahl der Pattern** einer der wichtigsten Metriken der SSA Form.

In der ersten Phase, also bei der Phi-Platzierung wird für jeden Lokator eine Worklist mit den Blöcke erstellt, die eine Definition zu einem Lokator haben. Dadurch gehört **die Anzahl der Lokatoren** zu den Metriken. Es wird dann für jeden Block untersucht. In allen Blöcken, die in der Dominanzgrenze von einem Block B (DF(B)) stehen, werden Phi-Knoten platziert oder es wird weiter analysiert, ob es weitere Benutzungen des Lokators gibt und dann erst platziert (je nach der Art der SSA). Dominanzgrenze von einem Knoten und einer Worklist können höchstens soviele Blöcke enthalten wie die Anzahl der gesamten Blöcke. Deswegen gehört **die Anzahl der Blöcke** zu den Metriken.

In der zweiten Phase also bei der Umbenennung müssen all diejenigen Lokatoren umbenannt werden, die mehrmals einen Wert zugewiesen bekommen. In dieser Phase wird es für jede Zuweisung durchgelaufen. Somit ist **die Anzahl der Zuweisungen** eine der wichtigen Metriken. Die Lokatoren, die auf der rechten Seite von einer Zuweisung vorkommen, werden zuerst umbenannt und anschließend jene umbenannt, die auf der linken Seite vorkommen. Deswegen gehören **die Anzahl der LHS und RHS** zu wichtigen Metriken.

Nachdem alle Lokator in dem übergebenen Block umbenannt wurden, müssen in den nachfolgenden Blöcke (Nachfolger im CFG) betrachtet werden, ob sie mit diesem Lokator eine Phi-Funktion haben. **Der maximale Ausgangsgrad im CFG** ist eine der wichtigen Metriken, die zur SSA Analyse gehören. Falls ein Block mindestens eine Phi-Funktion mit diesem Lokator hat, werden diese Lokatoren auch entsprechend Ihren neuen Namen umbenannt. Damit ist dieser Umbenennungsprozess für den Block fertig. Im schlimmsten Fall (worst case) enthalten alle Phi-Funktionen die Lokatoren, die umbenannt werden sollen. Deswegen gehört **Die Anzahl der Phi-Funktionen** auch zu den Metriken der SSA Form.

Dieser Prozess muss ebenfalls für alle anderen Blöcke durchlaufen werden. Dies wird mit dem Durchlauf im Dominanzbaum ermöglicht. Die Funktion wird für die Kinderblöcke

rekursiv aufgerufen. Die Tiefe des Dominanzbaum ist aus diesem Grund die Rekursionstiefe. Deswegen gehört **Die Tiefe des Dominanzbaumes** zu den Metriken der SSA Form.

In dem Letzten Schritt werden Seiteneffekte aktualisiert, um die Effekte zu behandeln werden die Werte zwischen der aufgerufenen und aufrufenden mit Hilfe von Knoten **Link_In_Definition** und **Link_Out_Uses** ausgetauscht, die Zugriff auf die Definition von Lokatoren haben. Die beiden Knoten gehören aus diesem Grund zu den Metriken.

Somit sind die Metriken der SSA Analyse ausgewählt worden. Damit ist der erste Schritt der Vorgehensweise fertig. Der zweite Schritt soll jetzt das Messen dieser Messwerte sein. Im Rahmen der Diplomarbeit wurde das Paket *ssa* implementiert. Dieses Paket kann diese Metriken messen und in das Objekt *T_Project* speichern. Mehr zu diesem Objekt wurde in dem Kapitel 3.4 erläutert. Der letzte Schritt unserer Vorgehensweise ist die Speicherung der Messwerte in die Datenbank, was die Aufgabe des Tools relevant_stats ist. Im folgenden Kapitel wird das Tool relevant_stats näher betrachtet.

3 Das Tool relevant_stats

Nachdem wir die SSA Form näher untersucht haben, behandeln wir nun das Tool relevant_stats. Das Tool relevant_stats wurde im Rahmen der Diplomarbeit implementiert, um Messwerte in die Datenbank speichern bzw. weitere Datenbankoperationen anwenden und daraus graphische Darstellung erstellen zu können. Mit diesen Informationen bzw. graphischen Darstellungen könnte die Aussage getroffen werden (siehe 5.2), warum eine bestimmte Analyse in unserem Beispiel die SSA Analyse lange dauert bzw. eine Vorausschätzung gemacht, wie lange es dauern wird. In diesem Kapitel wird erläutert, welche Eigenschaften dieses Tool hat und wie dieses Tool implementiert wurde. Zuerst werden die Begriffe definiert, die in den nächsten Kapiteln häufig benutzt werden.

In Folgenden wird das Verzeichnis, das die ganze Sourcecode enthält, nicht als Projekt genannt, sondern **ProjektVerzeichnis**, um Missverständnissen vorzubeugen. Nachdem eine Analyse in Bauhaus wie SSA-Analyse durch den entsprechenden Bauhaus-Werkzeug durchgeführt wurde, enthält die IML-Datei alle relevante Informationen zu dieser Analyse. Die Messwerte werden dann aus IML-Datei gelesen und in die Datenbank gespeichert. Dieser Prozess kann für ein ProjektVerzeichnis mehrmals durchgeführt werden. Es kann beispielsweise Sourcecode geändert werden, um zu vergleichen, ob es um eine Verbesserung bzw. Verschlechterung (Bsp. ssa_analysis läuft schneller!) geht. Um dieses zu verwirklichen, wird ein **Projekt** wie folgt zusammengefasst. Alle aus IML gemessene Informationen aus einem ProjektVerzeichnis sind in der Datenbank zu einem Projekt zugeordnet. Dieses Projekt bekommt den Namen des Projektverzeichnisses. Die Projekte werden aber nicht anhand ihrer Namen identifiziert. Dadurch können zwei unterschiedliche Projekte aus einem ProjektVerzeichnis erstellt und verglichen werden. Der letzte Begriff, der in dieser Ausarbeitung häufig vorkommt, ist **plot**. Mit dem Begriff plot ist die graphische Darstellung von den Messwerten gemeint. Die in der Datenbank gespeicherten Informationen können als Histogramm graphisch dargestellt werden. Die Begriffe Plot und plotten sollen sich auf diese Darstellung beziehen.

3.1 Einfuhrung in das Tool relevant_stats

Das Tool relevant_stats bietet verschiedene Möglichkeiten. Sowohl das Hinzufügen der Messwerte in die Datenbank sowie Auflisten, Aktualisieren, Löschen dieser Messwerte aus der Datenbank als auch die graphische Darstellung der Messwerte und die Ausgabe der Messwerte in CSV-Format (Comma Separated Values) gehören zu diesen Möglichkeiten. In der Abbildung 3.1 wird das Modell von relevant_stats graphisch dargestellt. Es müssen zuerst die Messwerte aus der entsprechenden IML-Datei gelesen und in die Datenbank gespeichert werden. Dies wird durch **insert** ermöglicht, erst dann können diese Messwerte durch **list** aufgelistet, durch **update** aktualisiert oder durch **delete** aus der Datenbank gelöscht werden. Die graphische Darstellung ist durch **plot** möglich.

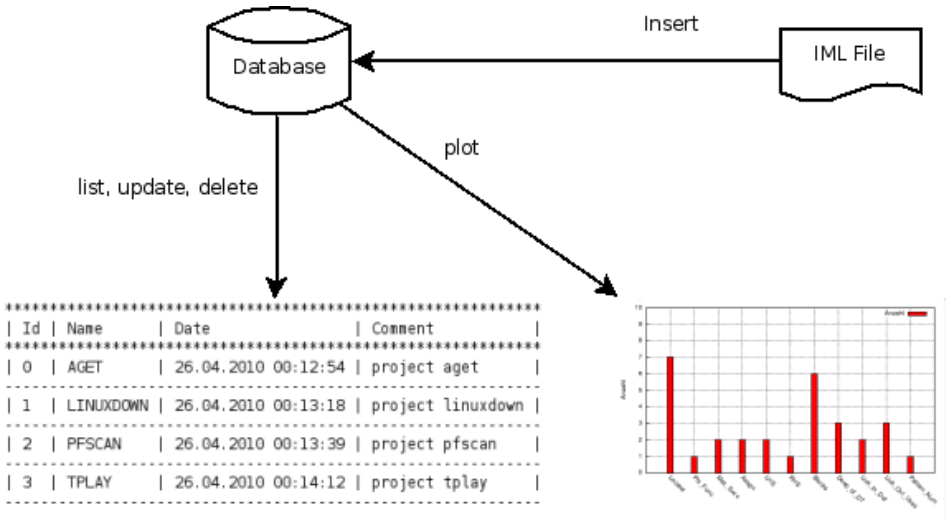


Abbildung 3.1: Das Modell von relevant_stats

Nachdem die Messwerte in der Datenbank gespeichert worden sind, benötigt das Tool nur die Datenbankdatei (DB-File). Das bedeutet, dass die IML-Datei nicht mehr für die Nutzung der anderen Funktionalitäten benötigt wird. Wie in Abbildung 3.1 erkennbar ist, wurden für das Tool zwei wichtige Konzepte eingeführt, das Datenbank- und das Plotkonzept. Als Datenbankkonzept ist es mit Hilfe von einer Schnittstelle von SQLite möglich, die SQL-Abfragen in relevant_stats ausführen zu können. Es besteht beispielsweise die Möglichkeit, die Informationen, die in der Datenbank als Messwert gespeichert sind, durch eine SQL-Abfrage aufzulisten. Das Plotkonzept nutzt das Da-

tenbankkonzept. Es ermöglicht die graphische Darstellung der in der Datenbank stehenden Informationen durch eine passende SQL-Abfrage. Um diese graphische Darstellung zu erstellen benutzt dieses Konzept GNUplot. In den folgenden Kapiteln werden diese beiden Konzepte detailliert erläutert.

3.2 Datenbank

In diesem Kapitel wird das Datenbankkonzept behandelt. Zunächst sind SQL und Views in SQL zu erläutern, die im Datenbankmodell benutzt werden. Nach der Erklärung des Datenbankmodells wird auf die in Rahmen der Diplomarbeit benutzte Datenbank SQLite eingegangen. Zuletzt müssen die DB-Operationen wie Insert, Update, Delete etc. in relevant_stats näher betrachtet werden, besonders wie sie implementiert sind.

3.2.1 SQL

SQL ist eine Datenbanksprache zur Definition, Abfrage und Manipulation von Daten in relationalen Datenbanken. SQL ist von ANSI und ISO standardisiert und wird von fast allen gängigen Datenbanksystemen unterstützt.

Die Syntax von SQL ist relativ einfach aufgebaut und semantisch an die englische Umgangssprache angelehnt. SQL stellt eine Reihe von Befehlen zur Definition von Datenstrukturen nach der relationalen Algebra zur Manipulation von Datenbeständen (Einfügen, Bearbeiten und Löschen von Datensätzen) und zur Abfrage von Daten zur Verfügung. Durch seine Rolle als Quasi-Standard ist SQL von großer Bedeutung, da eine weitgehende Unabhängigkeit von der benutzten Software erzielt werden kann.

Die meisten heute verbreiteten Datenbanksysteme implementieren mehr oder weniger große Teile des SQL-Sprachstandards. Dadurch ist es möglich, Anwendungsprogramme zu erstellen, die vom verwendeten Datenbanksystem unabhängig sind. In der Vor-SQL-Zeit gelang dies mit dem System der kompatiblen Schnittstellen. Viele SQL Implementierungen bieten darüber hinaus noch herstellerspezifische Erweiterungen, die nicht dem Standard-Sprachumfang entsprechen.

Ein kleiner Überblick über die Historie von SQL: [GM]

- 1974 D. Chamberlain et. al. definieren SEQUEL.
- 1977 Revision: SEQUEL/2; IBM-Prototyp System R.
- 1977 Oracle beginnt mit der Implementierung von SEQUEL auf Mainframes.
- 1979 Oracle liefert ein erstes relationales Datenbanksystem mit SQL aus.
- 1981 IBM liefert SQL/DS aus.
- 1983 IBM stellt DB2 vor.
- 1985 Ingres stellt auf SQL um.
  Informix stellt auf SQL um.
- 1986 Sybase wird ausgeliefert.
- 1986 X/OPEN entscheidet sich für SQL.
- 1987 Der Standard ISO 9075 Database Language SQL (SQL-86) wird veröffentlicht.
- 1988 dBASE IV erhält SQL als Abfrage-Sprache. Adabas erhält SQL.
- 1989 ISO 9075 Database Language SQL with Integrity Enhancement (SQL-89) wird veröffentlicht. SESAM und UDS erhalten SQL.
- 1992 ISO 9075 Database Language SQL (SQL2 oder SQL-92) wird veröffentlicht.
- 1993 RDA-Standard (Remote Data Access) der Database Access Group (z.B. Microsoft ODBC) wird veröffentlicht.
- 1996 SQL/PSM (Persistent Stored Modules); Datenbankprozeduren werden normiert (vgl. Kapitel 7.3).
  SQL:1999 wird veröffentlicht und löst damit SQL-92 ab.
- 1999 SQL:1999 wird veröffentlicht und löst damit SQL-92 ab.
- 2003 Eine abermals überarbeitete und teilweise neu strukturierte Version SQL:2003 löst SQL:1999 ab. Zum Zeitpunkt der Drucklegung dieses Buchs befindet sich die neue Norm aber noch im Entwurfsstadium und wird von Pessimisten (oder Realisten) als SQL:200n bezeichnet.

SQL bezieht seinen Namen aus dem englischen Wort **query**, zu Deutsch Abfrage. Mit Abfragen werden die in der Datenbank gespeicherten Daten abgerufen, d.h. dem Benutzer oder einer Anwendersoftware zur Verfügung gestellt. Das Ergebnis einer Abfrage sieht wiederum aus wie eine Tabelle und kann oft auch wie eine Tabelle angezeigt, bearbeitet und weiterverwendet werden.

Zwischen dem SQL-Standard und den in kommerziell verfügbaren Datenbanksystemen angebotenen SQL-Implementierungen gibt es Differenzen in beiden Richtungen. Teilweise übersteigt der verfügbare Sprachumfang die Forderungen der Norm, teilweise werden Forderungen der SQL-Norm nicht erfüllt. Es ist aber grundsätzlich zu erkennen, dass die Hersteller sich mehr und mehr um die Einhaltung des SQL-Standards bemühen.

Die Standardisierung von SQL ist aus folgenden Gründen wichtig: [GM]

- Herstellerunabhängigkeit ist gefordert.
- Die Verfügbarkeit von Front-Endsystemen, also Programmen, die herstellerunabhängig auf Datenbanken zugreifen können, nimmt zu.
- In Client-Server-Umgebungen kommuniziert die Anwendersoftware über Schnittstellen mit dem DBMS.
- Anwendungen sollen portabel sein, damit man beispielsweise auf ein leistungsfähigeres DBMS umsteigen kann oder damit die Anwendungsentwicklung in einer anderen Umgebung erfolgen kann als der produktive Einsatz.
- Verschiedene RDBMs können nebeneinander im Betrieb existieren. Die Anwendungsentwicklung und Datenbankadministration sollten möglichst wenig voneinander abweichen.

Es gibt die im Folgenden angegebenen Datentypen in SQL. Die Skalierungsgrößen in der Klammer hinter dem Datentyp können in der Regel weggelassen werden. Es werden dann Standardwerte eingesetzt, die bei numerischen Datentypen implementationsabhängig sind, bei Zeichen- und Bitketten jeweils 1, bei TIME 0 (d.h. keine Unterteilung der Sekunde) und bei TIMESTAMP 6 (d.h. Genauigkeit auf Mikrosekunden).

INTEGER	üblicherweise vier Byte
SMALLINT	üblicherweise zwei Byte
BIGINT	mindestens so groß wie INTEGER
NUMERIC(p,q)	Dezimalzahlen mit genau p Stellen, davon q hinter dem Dezimalpunkt
DECIMAL(p,q)	Dezimalzahlen mit mindestens p Stellen, davon q hinter dem Dezimal
REAL	Gleitpunktzahlen, einfache Genauigkeit
DOUBLE PRECISION	Gleitpunktzahlen, doppelte Genauigkeit
FLOAT(p)	Gleitpunktzahlen, mindestens p Stellen Genauigkeit
CHARACTER(n)	Zeichenketten mit genau n Zeichen
CHARACTER VARYING(n)	Zeichenketten mit höchstens n Zeichen
VARCHAR	Synonym zu CHARACTER VARYING
NATIONAL CHARACTER(n)	Zeichenketten mit genau n Zeichen mit nationalen Besonderheiten wie z.B. Zeichensatz, Sortierreihenfolge
NATIONAL CHARACTER	Zeichenketten mit höchstens n Zeichen mit nationalen Besonderheiten
CHARACTER LARGE OBJECT (n)	Große Textobjekte. Die maximal mögliche Länge ist implementationsabhängig
CLOB	Synonym zu CHARACTER LARGE OBJECT
BINARY LARGE OBJECT(n)	Binäre Daten mit einer maximalen Länge entsprechend n Zeichen
BLOB(n)	Synonym zu BINARY LARGE OBJECT
DATE	Kalenderdaten vom Jahr 1 bis zum Jahr 9999
TIME(p)	Uhrzeit in Stunden, Minuten, Sekunden, wobei die Sekunden noch p Stellen nach dem Komma haben
TIMESTAMP(p)	Datum und Uhrzeit
BOOLEAN	Wahrheitswerte TRUE und FALSE sowie UNKNOWN

Tabelle 3.1: Datentypen in SQL [GM]

SQL-Befehle lassen sich in drei Kategorien unterteilen, die allerdings im Standard nicht festgeschrieben sind:

- Data Manipulation Language (DML): Befehle zur Datenmanipulation (Ändern, Einfügen, Löschen)
- Data Definition Language (DDL): Befehle zur Definition des Datenbankschemas
- Data Control Language (DCL): Befehle für die Rechteverwaltung und Transaktionskontrolle.

Im Rahmen der Diplomarbeit wird DML sowie Ändern, Einfügen, Löschen von Daten, und DDL sowie Views angewandt. In den folgenden Kapiteln werden DML und DDL weiter erläutert.

3.2.1.1 Data Manipulation Language (DML)

Die Data Manipulation Language (DML, englisch für Datenbearbeitungssprache) ist derjenige Teil einer Datenbanksprache, der verwendet wird, um Daten zu lesen, zu schreiben, zu ändern und zu löschen. DML ist die Datenver- oder Datenbearbeitungssprache einer Datenbank und schließt die Formulierung von Abfragen ein.

Während in den ersten zwanzig Jahren der Datenbanktechnik die DML hauptsächlich zur Programmierung verwendet wurde, wird heute auch Wert auf die direkte Verwendung durch Benutzer gelegt. So kann z.B. die SQL-DML in den meisten Systemen auch interaktiv als Kommandosprache verwendet werden. Auch ist heute in der Regel das Wissen über interne technische Speicherstrukturen nicht mehr notwendig, um entsprechende Verarbeitungsbefehle formulieren zu können. In diesem Fall spricht man von beschreibenden Sprachen.

Im Folgenden gibt es Syntax von Select, Insert, Update und Delete in SQL, die im Rahmen der Diplomarbeit im Tool relevant_stats Anwendung finden.

```
select-exp ::= SELECT [ALL | DISTINCT] select-item-commalist
               FROM table-ref-commalist
               [WHERE cond-exp]
               [GROUP BY column-ref-commalist]
               [HAVING cond-exp]

insert-exp ::= INSERT INTO table [ (column-commalist) ]
           { VALUES row-constr.-commalist | table-exp | DEFAULT VALUES }

update-exp ::= UPDATE table
               SET update-assignment-commalist
               [WHERE cond-exp]

delete-exp ::= table DELETE FROM [WHERE cond-exp]
```

Abbildung 3.2: DML-Anweisungen [Sch10]

3.2.1.2 Data Definition Language (DDL)

Zu Data Definition Language gehören: [GM]

- die Anweisungen zur Anlage und Verwaltung von Schemata (Zusammenfassung von Tabellen, Datensichten etc.), für die ein bestimmter Benutzer zuständig ist
- die Anweisungen zur Definition von Relationen einschließlich der dazugehörigen Konsistenzbedingungen
- die Anweisung zur Anlage von Datensichten (Views)
- die Verwaltung von Benutzern und deren Datenzugriffsrechten.

Die oben aufgelisteten Eigenschaften spielen für relevant_stats Datenbankmodell eine wichtige Rolle. In Bauhaus können verschiedene Analysen gleiche Metriken enthalten. Aus diesem Grund werden die Informationen, die unterschiedliche Analysen betreffen, nicht für jede Analyse (mehrmals) gespeichert, sondern in einer gemeinsamen Tabelle hinzugefügt und aus dieser Tabelle eine Sicht für jede Analyse gebildet. Weil die Daten nur einmal gespeichert sind, kommt es zu keinem Inkonsistenzen Zustand. Mehr zu diesem Thema wird im Folgenden Kapitel erläutert.

Eine zentrale Forderung an eine relationale Datendefinitionssprache ist die Unterstützung von Integritätsbedingungen. SQL unterstützt Schlüssel (Primär- und Kandidatenschlüssel) und Fremdschlüssel. Dies sind Integritätsregeln, die sich durch logische Bezüge von Daten auf andere Daten ergeben. Für die Zuverlässigkeit der Daten ist beispielsweise die Eindeutigkeit aller Primär- und Kandidatenschlüsselwerte unverzichtbar - wie soll sonst ein bestimmtes Projekt identifiziert werden? Seit 1992 wird eine Form von benutzerdefinierten Datentypen unterstützt. SQL:2003 enthält auch objektorientierte Elemente, die die so genannten **objektrelationalen Datenbanken** charakterisieren sollen. Dazu gehört die Möglichkeit, echte benutzerdefinierte Datentypen inklusive Zugriffsmethoden und Vergleichsoperatoren zu erzeugen. Allgemein gilt, dass die Erzeugung neuer Datenbankobjekte mit dem Schlüsselwort CREATE eingeleitet wird. Anweisungen zur Entfernung von Datenobjekten beginnen mit dem Schlüsselwort DROP.

3.2.1.3 Datensichten (Views) in SQL

Eine Sicht (View) ist eine logische Relation (auch virtuelle Relation oder virtuelle Tabelle) in einem Datenbanksystem. Diese logische Relation wird über eine im Datenbankmanagementsystem (DBMS) gespeicherte Abfrage definiert. Der Datenbankbenutzer kann eine Sicht wie eine normale Tabelle abfragen. Wann immer eine Abfrage diese Sicht benutzt, wird diese zuvor durch das Datenbankmanagementsystem berechnet. Eine Sicht stellt im Wesentlichen einen Alias für eine Abfrage dar. Die folgende Abbildung ist ein Beispiel dafür, wie eine Sicht (Views) in SQL erzeugt werden kann.

```
CREATE VIEW SSA_Table
   AS SELECT Metrik_1, Metrik_2, Metrik_3
   FROM Routine_Table
```

Die Aufgabe einer Sicht ist es, den Zugriff auf das Datenbankschema zu vereinfachen. Normalisierte Datenbankschemata verteilen Daten auf zahlreiche Tabellen mit komplexen Abhängigkeiten. Dies führt zu aufwändigen SQL-Abfragen. Außerdem wird ein hohes Maß an Wissen über das Schema vorausgesetzt, um solche Abfragen zu erstellen. Das Bereitstellen geeigneter Sichten erlaubt einen einfachen Zugriff, ohne Kenntnis des darunter liegenden Schemas und ohne Aufweichung der Normalisierung.

Ein weiterer Vorteil von Sichten ist, dass das DBMS keinen zusätzlichen Aufwand zur Vorbereitung der Abfrage benötigt. Die Sicht-Abfrage ist vom Parser bereits bei der Erstellung syntaktisch zerlegt und vom Anfrageoptimierer vereinfacht. Ein Nachteil von

Sichten kann sein, dass die Komplexität der dahinter liegenden Abfrage unterschätzt wird. Der Aufruf einer Sicht kann zu sehr aufwändigen Abfragen führen und der unbedachte Einsatz solcher wiederum, zu erheblichen Performanceproblemen.

Sichten können anhand der verwendeten Anweisungen in verschiedene Klassen eingeteilt werden, die unterschiedliche Aufgaben haben: Selektionssicht, Projektionssicht, Verbundsicht und Aggregationssicht. Während die *Selektionssicht* aus einer Tabelle bestimmte Zeilen herausfiltert, filtert die Projektionssicht bestimmte Spalten. Die Projektionssicht wurde in relevant_stats angewandt und aus den Metriken beispielsweise, die sich in der Routine_Table befinden und für SSA Form bestimmt worden sind, wird eine Tabelle mit dem Name SSA_Table gebildet. Falls mehrere Tabellen verknüpft werden sollen, wird die Verbundsicht angewandt. Bei der Aggregationssicht gibt es die Möglichkeiten die Aggregationsfunktionen (MIN, MAX, COUNT etc.) anzuwenden.

Neben den herkömmlichen Sichten gibt es noch so genannte materialisierte Sichten (materialized views), auch Indexed Views (Microsoft) genannt. Dies sind Sichten, die physikalisch gespeichert werden, um Zugriffe zu beschleunigen. Das findet vor allem bei den großen Datenmengen bei OLAP Anwendung. Mittlerweile spielen diese Views auch in herkömmlichen Datenbanken eine Rolle, da sie bei der Erstellung und Optimierung von QEPs (Query Execution Plan) berücksichtigt werden.

Updates auf eine Sicht sind im Allgemeinen nicht möglich, da sie zu Anomalien führen können. Auf die Sicht kann dann nur lesend zugegriffen werden. In besonderen Fällen, in denen das DBMS eine eindeutige Zuordnung zwischen den in der Sicht zu ändernden Daten und einer physikalischen Tabelle, zu der sie gehören, herstellen kann, ist ein Update möglich. Beispiel für so eine nicht updatable view wäre folgende triviale Sicht:

```
create view FleißigeStudenten (MatrNo, Durchschnittsnote)
  as select MatrNo, avg(Note)
  from Student
  where Note < 2.0
```

Diese Sicht ist nicht veränderbar, da sie das berechnete Attribut Durchschnittsnote enthält. Eine Änderungsoperation lässt sich nicht mehr auf die ursprüngliche Basistabelle *Student* zurückpropagieren. Die folgende Operation würde also von DBMS zurückgewiesen.

```
update FleißigeStudenten
       set Durchschnittsnote=1.0
       where MatrNo = ( select MatrNo
                        from Student
                        where Name='Müller')
```

In SQL-92 ist nur die Änderung reiner Selektionssichten erlaubt. Die Option WITH CHECK OPTION im CREATE VIEW-Statement gibt dann an, ob Änderungen, die zum Verschwinden des Datensatzes aus der Sicht führen würden, verboten sein sollen. In SQL-99 wurde die Menge an änderbaren Sichten zwar deutlich erweitert, sie bleibt aber dennoch immer hinter der theoretisch möglichen zurück. In neueren SQL-Dialekten ist es möglich, Trigger dazu einzusetzen, Updateoperationen auf Sichten von Hand zu implementieren.

In relevant_stats wurden die definierten Sichten nur zum Lesen verwendet. Aus Ihnen können die gewünschten Zeilen selektiert werden. Falls sie aktualisert (bzw. gelöscht oder neue Zeilen hinzugefügt) werden, sollen diese in der Haupttabelle (Routine_Table) realisiert werden, um keine Gedanken mehr über Anomalien zu machen.

3.2.2 Datenbankmodell

In diesem Kapitel wird das Datenbankmodell des Tools relevant_stats diskutiert. In der Abbildung 3.3 wird das Datenbankmodell von relevant_stats dargestellt. Das Tool relevant_stats enthält zwei Tabellen: Project_Table und Routine_Table. Die Project_Table enthält die Informationen von Projekten. Der Begriff Projekt bezieht sich auf nach einem Durchlauf gemessene Informationen aus einem Projektverzeichnis. Angenommen wir haben das ProjektVerzeichnis **AGET**, das die Sourcecode (in C) enthält. Aus diesem ProjektVerzeichniss AGET sollen die Informationen, sowie die Anzahl der Locator gemessen und in die Datenbank gespeichert werden. In der Datenbank gibt es dann ein Projekt mit dem Namen AGET. Falls Sourcecode vom ProjektVerzeichnis geändert werden soll, dann können die neuen Informationen nochmal in die Datenbank gespeichert werden, ohne dass alte Informationen verlorengehen. Dadurch können sowohl verschiedene Projekte miteinander als auch ein Projekt mit seinen Änderungen verglichen werden. Projekte in der Project_Table sind durch ihre ID_Nummer identifiziert. Falls ein Projekt in der Datenbank mehrmals gespeichert wird, hat das Duplikat in der Datenbank den gleichen Name wie das Projekt, aber eine andere ID_Nummer. Zusätzlich enthält die Project_Table die Informationen darüber, wann das Projekt erstellt

wurde. Es können noch Kommentare zu dem Projekt hinzugefügt werden. Die Spalten Comment und Name können durch das Tool relevant_stats aktualisiert werden. Das letzte Attribut, das ein Projekt in der Project_Table enthält, ist eine Information zur Laufzeit. In der Datenbank muss gespeichert werden, wie lange die Analyse für dieses Projekt gedauert hat.

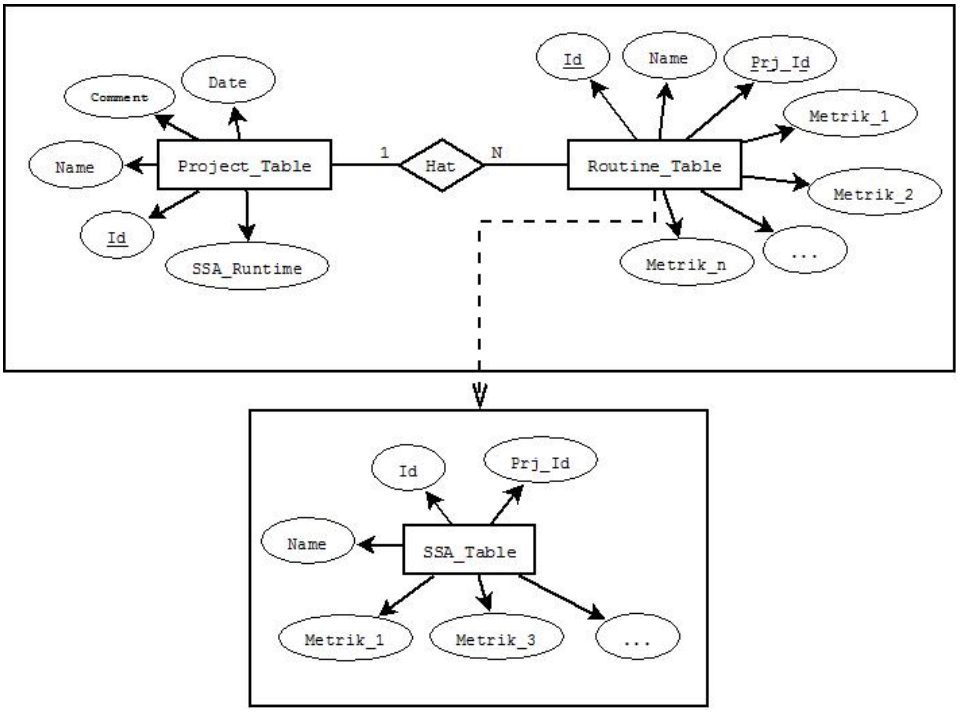


Abbildung 3.3: Datenbankmodell von relevant_stats

Die Routine_Table enthält dagegen alle Metriken, die für eine, bzw. mehrere vorgegebenen Analysen bestimmt worden sind. Falls eine gleiche Metrik für andere Analysen ausgewählt werden soll, wird sie nur einmal in der Routine_Table gespeichert. Dadurch werden mögliche Anomalien bzw. Inkonsistente Zustände verhindert. Die Routine_Table ist auch durch ihre ID-Number identifiziert und hat Prj_Id als Foreign-Key. Mit dieser Prj_Id wird bestimmt, zu welchem Projekt diese Routine gehört.

Im Tool relevant_stats wurde die Projektionssicht benutzt. Es wird für jede Analyse eine virtuelle Tabelle aus den Metriken gebildet, die in der Routine_Table gespeichert sind

aber zu der/den Analyse/n (in unserem Beispiel zur SSA Form) gehören. *Virtuell* heißt im Zusammenhang, dass keine neuen Tabellen angelegt werden. Vielmehr werden sie bei jeder Verwendung neu berechnet. Im Kapitel 3.2.1.3 wurden einige Probleme beim Aktualisieren (Update) von Sichten erläutert. Die Sichten sollen aktualisierbar (update) erzeugt werden, damit Änderungsoperation in der virtuellen Tabelle durchgeführt werden können. Alle Änderungsoperationen, die Metriken der vorgegebenen Analyse (in unserem Beispiel SSA_Table) betreffen, werden in der Routine_Table durchgeführt. Die virtuellen Tabellen sind nur aus diesen Gründen zu den Leseoperationen angewandt. Weil die virtuellen Tabellen bei jeder Verwendung neu berechnet werden, bekommen sie die Änderungen von der Routine_Table mit, als würden die Änderungen in dieser virtuellen Tabelle SSA_Table durchgeführt.

3.2.3 SQLite

SQLite ist eine Bibliothek, die ein relationales Datenbanksystem enthält. Sie unterstützt einen Großteil der im SQL-92-Standard festgelegten SQL-Sprachbefehle. SQLite implementiert Transaktionen, Subselects, Sichten (views), Trigger und benutzerdefinierte Funktionen. Das System ist vor allem für die Embedded-Systeme entworfen, daher fehlen Funktionen wie die Möglichkeit, Objektberechtigungen zu verwalten (GRANT, REVOKE). Der Code für SQLite ist im öffentlichen Bereich (public domain) und somit frei für die Verwendung für gewerbliche oder private Zwecke. Für die wichtigen Programmiersprachen wie C, C++ existieren passende Datenbankschnittstellen. SQLite entält ein einfaches Frontend, das in der Konsole und in Shell-Skripten verwendbar ist. Im Vergleich zu anderen Datenbanksystemen lässt sich die SQLite-Bibliothek direkt in entsprechende Anwendungen integrieren, so dass eine weitere Server-Software nicht benötigt wird. Durch das Einbinden der Bibliothek werden die Datenbankfunktionalitäten erweitert angewandt, ohne auf externe Softwarepakete angewiesen zu sein. Dies ist entscheidender Unterschied gegenüber anderen Datenbanksystemen.

SQLite wurde im Jahr 2000 von Richard Hipp entwickelt und ist in C geschrieben. Auf der Projektseite wird außerdem eine Bibliothek für die Benutzung unter Tcl angeboten. Für andere Programmiersprachen wie C++ und Java gibt es Wrapper externer Entwickler, ebenso stehen ODBC- und JDBC-Treiber zur Verfügung. SQLite erzwingt keine Typsicherheit. Das heißt fehlerhafte Eingaben werden in der Regel akzeptiert und

in Zeichenketten gewandelt.

Die Bibliothek ist nur wenige hundert Kilobyte groß. Mit allen Features ist die Bibliothek 300 KiB gross. Sie kann aber je nach der Compiler-Optimierung kleiner werden (unter 180 KiB reduziert werden [Homc]). Die Datenbanken können verteilt auf mehrere Dateien, bei Bedarf auch in eine einzelne Datei, gespeichert werden, was den Austausch zwischen verschiedenen Systemen (zwischen big-endian and little-endian Architekturen), sogar zwischen Systemen mit unterschiedlichen Byte-Reihenfolgen (zwischen 32-Bit und 64-Bit), vereinfacht. Jede Spalte kann Daten beliebiger Typen enthalten. Erst zur Laufzeit wird nötigenfalls konvertiert.

SQLite nutzt die Lese- und Schreibsperre auf der gesamten Datenbankdatei. Das heißt, wenn irgendein Prozess einen Teil der Datenbank liest, wird verhindert, dass alle anderen Prozesse den anderen Teil der Datenbank ändern. Ähnlich ist es bei den Schreiboperationen. Während ein Prozess mit der Änderungsabsicht aus der Datenbank liest, können alle anderen Prozesse auf irgendeinen Teil der Datenbank nicht zugreifen. Aus diesem Grund müssen die Anwendungen, die mehr Parallelität erfordern, möglicherweise eine andere Lösung suchen.

Es gibt einige in anderen Datenbanksystemen vorhandene Features, die SQLite nicht bietet oder in anderen Datenbanksystemen besser funktionieren. Der Befehl ALTER TABLE existiert nur mit eingeschränktem Funktionsumfang und erst seit Version 3.2. Man kann derzeit lediglich Tabellen umbenennen und Spalten zu den Tabellen hinzufügen. SQLite verfügt über keine Verwaltung von Benutzer- und Zugriffsberechtigungen auf Datenbank-Ebene. Es gelten für die Datenbank-Dateien die Zugriffsberechtigungen des Dateisystems. Falls mehrere Client-Programme über das Netzwerk auf die gemeinsame Datenbank zugreifen sollten, sollte man daher client/server Datenbank-Engine benutzen, weil SQLite zurzeit parallele Dateisysteme nicht unterstützt. Falls es um eine größe Datenmenge geht, sollte auf SQLite verzichtet werden, da SQLite nur mit bis zu 2 Terabytes (2^{41}) Daten umgehen kann. SQLite kann zwar doch mit größeren Daten umgehen, doch es ist offensichtlich ein Problem, die ganze Datenbank in einer Datei zu speichern. Die maximale Dateigröße bei vielen Dateisystemen ist um etwas weniger als diese Größe. Aus diesem Grund sollten in so einem Fall andere Datenbanksysteme verwendet werden.[Homc]

Eingesetzt wird SQLite in Betriebssystemen für Mobiltelefone, wie Symbian OS oder Android. Außerdem nutzt der Mozilla-Firefox-Browser die Version 3 SQLite beispielsweise für Lesezeichen und Cookies, nachdem er schon die Version 2 SQLite für programminterne Datenbanken benutzt hatte. Für diesen Browser und alle anderen Mozilla-basierten Anwendungen gibt es zudem das Add-on SQLite Manager, mit dem man die SQLite-Datenbanken der Anwendung, aber auch jede beliebige andere SQLite-Datenbank betrachten, bearbeiten und abfragen kann.

Des Weiteren benutzt der Browser Safari von Apple intern SQLite, sowohl die Desktop- als auch die iPhone-Variante sowie das E-Mail-Programm Mail von Apple. Innerhalb von Mac OS X wird SQLite auch zur Verwaltung von Cache-Daten wie z.B. die der Software-Aktualisierung eingesetzt. SQLite kommt auch in Adobe AIR als Datenbankanwendung zum Einsatz. Ab Version 1.5 von Adobe AIR können SQLite-Daten mittels der EncryptedLocalStore-Klasse auch verschlüsselt abgespeichert und gelesen werden. Die VoIP/Messenger-Software Skype ab Version 4 speichert Kontakte und alle anfallenden Nutzerdaten (Chatprotokolle, usw.) in einer unverschlüsselten SQLite-Datenbank ab.

Im Rahmen der Diplomarbeit wurde die Datenbank SQLite3 in Absprache mit dem Betreuer eingesetzt, um die gemessene Metriken zu speichern und noch weitere Datenoperationen auszuführen. Es wurde eine Schnittstelle für SQLite in Ada implementiert. Die andere möglichkeit wäre *GNADE* zu benutzen. Das Projekt *GNADE* bietet eine komplette Lösung für die Entwicklung datenbankgestützter Applikationen in Ada 95. Da alle PSLx-Rechner dieses Binding nicht zu Verfügung hatten, wurde es mit der Absprache vom Betreuer darauf verzichtet, das Binding *GNADE* einzusetzen. Das folgende Kapitel gibt mehr Information zum Thema einer SQlite-Schnittstelle in Ada.

3.2.3.1 SQLite Schnittstelle in Ada

SQLite Schnittstellen lassen sich in drei Kategorien unterteilen:

- Stabil
- Experimentell
- Veraltet

Die Anwendungen, die stabile Schnittstellen benutzen, sind in der Lage stabile Schnittstellen in die neuere Version von SQLite zu binden (relink). Experimentelle Schnittstellen sind dagegen versionsabhängig. Das bedeutet, die Anwendungen, die diese Schnittstellen benutzen, müssen ihre Schnittstellen aktualisieren, um eine neue Version von SQLite zu benutzen. Die neuen Schnittstellen werden als experimentell eingesetzt. Nach einer Weile kann eine experimentelle Schnittstelle als stabil markiert werden, falls sie von Entwicklern als zuverlässig bezeichnet wird. Veraltete Schnittstellen wurden durch neue Methoden abgelöst und auf ihre Verwendungen sollte verzichtet werden. Die veralteten und experimentellen Schnittstellen können jederzeit entfernt werden.

Im Rahmen der Diplomarbeit wurde aus der C-Schnittstelle eine Ada Schnittstelle implementiert. Daher gibt es eine kleine Einführung in die C-Schnittstelle von SQLite.

Die C-Schnittstelle hat zwei Objekte *sqlite* und *sqlite_stmt*. Diese Objekte sind unter der Kontrolle einer kleinen Gruppe von Routinen *sqlite3_open*, *sqlite3_prepare*, *sqlite3_step*, *sqlite3_column*, *sqlite3_finalize*, *sqlite3_close* ausgeführt, um die SQL-Anweisungen zu bewerten, wobei das Objekt *sqlite* für die Datenbankverbindung zuständig ist. Diese sechs Routinen und zwei Objekte bilden den Kern der C-Schnittstelle von SQLite.

sqlite3_open öffnet eine Datenbankverbindung zu einer Datenbankdatei und liefert Datenbankobjekt zurück. *sqlite3_open* ist meistens der erste Aufruf in SQLite-API und Voraussetzung für die meisten Funktionen, um die SQL-Abfrage sinvoll zu bewerten. *sqlite3_prepare* wandelt den SQL-Text in ein prepared Statement Objekt, die so genannte *sqlite_stmt*, um und gibt einen Zeiger auf dieses Objekt zurück. Diese Routine braucht wie fast alle andere Routine eine Datenbankverbindung und deswegen als Übergabe die Datenbankverbindungsobjekt *sqlite* und setzt einen vorherigen Aufruf von *sqlite3_open* voraus. Diese Routine bereitet die SQL-Anweisung für die Bewertung vor. Es wird aber von [Homc] nicht empfohlen diese veraltete Routine zu benutzen. Stattdessen gibt es die Routine *sqlite3_prepare_v2* als Ersatz. Die Routine *sqlite3_step* bewertet das Objekt *sqlite3_stmt*, was in der vorherigen Routine *sqlite3_prepare* bzw. *sqlite3_prepare_v2* erzeugt wurde. Die Bewertung mit dieser Routine betrifft nur eine Zeile. Um die nächste Zeile zu bewerten, soll diese Routine nochmal aufgerufen werden. Bei den Anweisungen, die kein Ergebnis zurückgeben (INSERT, UPDATE, DELETE), wird der einmalige Aufruf ausreichen. *sqlite3_column* gibt eine einzelne Spalte aus der Zeile, die mit dem Aufruf von *sqlite3_step* erzeugt wurde. Diese Routine muss auch mehrmals aufgerufen

werden, um alle Spalten einer Zeile zu bewerten. *sqlite3_finalize* gibt das Objekt *sqlite3_stmt* frei, während *sqlite3_close* die Datenbankverbindung schließt.

Die C-Schnittstelle von SQLite hat einige Routine wie *sqlite3_exec*, die von den Programmierern mehr benutzt werden. Die oben genannte Routine (prepare, step und column) sind in der Routine *sqlite3_exec* gekapselt. Falls es irgendeinen Fehler auftritt, kann die Fehlermeldung durch den Aufruf von der Funktion *sqlite_errormsg* erfahren werden. Im Folgenden wird ein Beispiel gezeigt, wie eine SQL-Anweisung in C durchgeführt wird.

```
rc = sqlite3_open(myDB_File, &db);
if( rc ){
  fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
  sqlite3_close(db);
  exit(1);
}
rc = sqlite3_exec(db, mySQL_STMT, callback, 0, &zErrMsg);
if( rc != SQLITE_OK ){
  fprintf(stderr, "SQL error: %s\n", zErrMsg);
  /* This will free zErrMsg if assigned */
  if (zErrMsg)
    free(zErrMsg);
}
sqlite3_close(db);
```

Im Rahmen der Diplomarbeit wurde die Ada Schnittstelle (sqlite.ads und sqlite.adb) für SQLite implementiert. Dafür wurden die Routinen von der C-Schnittstelle in Ada importiert. Die importierten Routinen sind sqlite3_column_text, sqlite3_finalize, sqlite3_column_type, sqlite3_column_count, sqlite3_free_table, sqlite3_get_table, sqlite3_column_name, sqlite3_close, sqlite3_errmsg, sqlite3_open_v2.

Die Ada Schnittstelle enthält zwei wichtige Objekte *Database* und *Result_Table*. Das Objekt *Database* ist für die Datenbankverbindung zuständig und muss bei fast allen Routinen übergeben werden. Das zweite Objekt *Result_Table* dagegen, enthält die Tabelle, die nach Ausführung der Anweisung ausgegeben werden. Das ist immer der Fall, wenn es um die SELECT-Anweisung geht. Bei den anderen Anweisungen wie INSERT, UPDATE, DELETE ist dieses Objekt nicht zu gebrauchen. Es gibt in dem Paket zusätzliche Objekte *Sqlite_Types* und *Result_Codes*. *Sqlite_Types* enthält die Typen in SQL, die erlaubt sind. Die sind Sqlite_Integer, Sqlite_Float, Sqlite_Text, Sqlite_Blob, Sqlite_Null. Das Objekt *Result_Codes* hat die mögliche Ergebnisse wie Sqlite_OK,

Sqlite_Error oder Sqlite_Locked, die nach der Ausführung einer Anweisung auftreten können.

In sqlite.ads wurden die folgenden Routinen implementiert, um die SQL-Abfragen in Ada zu ermöglichen. Die Routinen sind im Folgenden dargestellt.

- Open
- Get_Table
- Error_Msg
- Get_Value
- Get_Rows_Count
- Get_Columns_Count
- Get_Column_Name
- Free_Table
- Close

Die Routine *Open* ermöglicht die Datenbankverbindung und gibt eine Referenz auf das Objekt *database* zurück. Um eine Datenverbindung herstellen zu können, muss die entsprechende Datenbankdatei und sein Zweck übergeben werden. Der Zweck kann entweder das Lesen und/oder Ändern der Datenbankdatei (bzw. in der Datenbank) sein oder das Erzeugen der Datenbankdatei. Falls der Verbindungaufbau gescheitert ist, wird die Variable Status mit der entsprechenden Fehlermeldung gefüllt, die in Result_Codes definiert wurde. Ansonsten hat der Status immer den Wert Sqlite_OK. Der Datenbankabbau erfolgt durch die Routine *Close*. Daher braucht diese Routine als Übergabe nur das Database Objekt.

```
---------------------------------------------------
-- PROCEDURE : Open                               --
-- DEF       : Opening a connection.              --
---------------------------------------------------
procedure Open
  (DB       : out Database;
   Filename : String;
   Flags    : Open_Flags := Open_Readwrite or
```

```
                                Open_Create;
   Status   : out Result_Codes);
--------------------------------------------------
-- PROCEDURE : Close                            --
-- DEF       : Close the connection             --
--------------------------------------------------
procedure Close (DB : Database);
```

Die Routine *Get_ Table* bewertet die SQL-Anweisung und gibt als Antwort eine Tabelle (Result_Table) zurück. Diese Routine darf erst ausgeführt werden, nachdem die Datenbankverbindung mit dem Aufruf von Open erfolgreich war. Das Objekt *database*, das nach der Ausführung von *Open* erstellt wurde, und die SQL-Abfrage müssen bei dem Aufruf von dieser Routine übergeben werden. Ob die Anweisung erfolgreich war, kann mit Hilfe der Variable Status erkannt werden. Falls sie gescheitert ist, wird die Fehlermeldung in Error_Msg gespeichert.

```
--------------------------------------------------
-- PROCEDURE : Get_Table                        --
-- DEF       : Execute a query on the database  --
--------------------------------------------------
procedure Get_Table
  (DB     : Database;
   SQL    : String;
   Result : out Result_Table;
   Status : out Result_Codes;
   Error  : out Interfaces.C.Strings.chars_ptr);
```

Die Routinen *Get_ Value*, *Get_ Rows_ Count*, *Get_ Columns_ Count* und *Get_ Column _ Name* sind in der Lage, die Tabelle zu bewerten, die durch den Aufruf von *Get_ Table* erzeugt wurde. *Get_ Value* kann den Wert an der Stelle (Row,Column) in der Tabelle angeben. Während die Anzahl der Spalten und Zeilen durch den Aufruf von *Get_ Rows_ Count* oder *Get_ Columns_ Count* ermittelt werden können, kann der Name einer bestimmten Spalte durch den Aufruf von *Get_ Column_ Name* erfahren werden.

```
--------------------------------------------------
-- FUNCTION  : Get_Value                        --
-- DEF       : Return the value at a specific   --
--             row/column. Column and Row start --
--             at 0                             --
--------------------------------------------------
function Get_Value
  (Result : Result_Table;
```

```
   Row, Column : Natural)
  return Interfaces.C.Strings.chars_ptr;

--------------------------------------------------
-- FUNCTION  : Get_Rows_Count                   --
-- DEF       : Return the number of rows        --
--------------------------------------------------
function Get_Rows_Count (Result : Result_Table)
  return Natural;
--------------------------------------------------
-- FUNCTION  : Get_Columns_Count                --
-- DEF       : Return the number of columns     --
--------------------------------------------------
function Get_Columns_Count (Result : Result_Table)
  return Natural;
--------------------------------------------------
-- FUNCTION  : Get_Column_Name                  --
-- DEF       : Return the name of a specific    --
--             column. Column starts at 0       --
--------------------------------------------------
function Get_Column_Name
  (Result : Result_Table; Column : Natural)
  return String;
```

Letztendlich kann die Tabelle (Result_Table) mit Hilfe eines Aufrufs der Routine *Free_Table* freigegeben werden, falls keine weitere Anweisung ausgeführt werden soll. Das folgende Beispiel zeigt, wie eine SQL-Abfrage mit diesen Komponenten ausgeführt und bewertet wird.

```
  -------------------
  -- Execute_Query --
  -------------------
  procedure Execute_Query(pFile : Unbounded_String)
    is
      DB      : SQlite.Database     := SQLite.No_Database;
      Result  : SQlite.Result_Table := SQLite.No_Table;
      Status  : SQlite.Result_Codes := SQLite.Sqlite_Error;
      sLocal  : Unbounded_String    := Str("");
      Error   : Interfaces.C.Strings.chars_ptr;
      sText   : Interfaces.C.Strings.chars_ptr;
      nColumn : Natural := 0;
      nRow    : Natural := 0;
  begin
    SQLite.Open(DB, Str(pFile), SQLite.Open_Create, Status);
```

```
   if Status = SQLite.Sqlite_OK then
      sLocal := Str("select * from Project_Table");
      SQLite.Get_Table (DB, Str(sLocal), Result, Status, Error);
   end if;

   if Status = SQLite.Sqlite_OK then
      nColumn := SQLite.Get_Columns_Count(Result);
      nRow    := SQLite.Get_Row_Count(Result);
      for i in 0 .. nRow - 1 loop
         for j in 0 .. nColum - 1 loop
            sText := SQLite.Get_Value(Result, i, j);
            sLocal := Str(Interfaces.C.Strings.Value(sText));
            Ada.Text_IO.Put(Str(sLocal));
         end loop;
      end loop;
   else
      sLocal := Str(Interfaces.C.Strings.Value(Error));
      Ada.Text_IO.Put("Error: " & Str(sLocal));
   end if;

   SQLite.Free_Table(Result);
   SQLite.Close(DB);

end Execute_Query;
```

3.2.4 DB Operationen

In diesem Kapitel werden die Datenbankoperationen diskutiert, die in relevant_stats möglich sind. Diese Operationen sind im Rahmen von relevant_stats entsprechend der gewünschten Anforderungen ausgewählt und auch definiert. Es soll die Möglichkeiten geben, die Messwerte in der Datenbank zu speichern (insert), nach dem Einfügen dieser Messwerte aus der Datenbank zu lesen, aufzulisten (list) und einige Informationen zu aktualisieren (update) oder zu löschen (delete). Im Folgenden soll der Aufbau und die Implementierung jeder Datenbankoperation erklärt werden.

3.2.4.1 Check

Die erste Datenbankoperation, die in dieser Ausarbeitung vorgestellt wird, ist *Check*. Sie überprüft, ob die angegebene Datenbankdatei konsistent ist. Diese Funktionalität wird intern bei den anderen Datenbankoperationen (z.B. List, Update, Delete) benutzt. Es

wird an erster Stelle beispielsweise nach Konsistenz der Datenbankdatei abgefragt, um eventuell ein Projekt aus der Datenbank zu löschen. Es kann aber direkt durch das Tool relevant_stats abgefragt werden, ob eine Datei konsistent ist oder nicht. Die Syntax zu dem Check lautet folgendermaßen.

```
****************************************
* relevant_stats -db check <DB_FILE>  *
****************************************
```

Eine Datenbankdatei ist dann konsistent, wenn die Projekttabelle (Project_Table), Routinetabelle (Routine_Table) und Analysentabellen (in Unserem Beispiel SSA_Table) mit den entsprechenden Spalten in der Datenbank existieren. Diese Überprüfung erfolgt durch SQL-Anweisungen. Es wird eine Select-Anweisung für jeweils eine Projekttabelle, Routinetabelle und Analysentabelle mit ihren Spalten durchgeführt und überprüft, ob ein Fehler auftritt.

```
select Name,Id,Prj_Id,Metrik_1, Metrik_2 ... Metrik_n from Routine_
    Table;
```

Es bestehen folgende Möglichkeiten:

- Die Tabellen existieren, aber mindestens eine der Spalten fehlt. Die Datei ist also inkonsistent.
- Eine der Tabellen fehlt. Die Datei ist inkonsistent.
- Alle Tabellen existieren mit entsprechenden Spalten. Die Datei konsistent.

3.2.4.2 List

Es besteht in relevant_stats die Möglichkeit, die Informationen aufzulisten, die in der Datenbank existieren.

```
*******************************************************
* relevant_stats -db "list <ID> | <SQL_STMT>" <DB_FILE> *
*******************************************************
```

Die Auflistung kann entweder durch eine ID-Nummer oder SQL-Anweisung erfolgen. Die ID-Nummer bezieht sich auf die ID-Nummer des Projekts in der Projekttabelle. Dort werden alle Informationen zu diesem Projekt aufgelistet. Im Folgenden wird der Pseudocode gezeigt, wie list in relevant_stats implementiert wurde.

```
if Check(DbFile) = False then
   return;
end
if Is_A_SQL then
   führe STMT durch
else
   if ID = * then
      STMT := Select * from Project_Table
   else
      STMT := Select * from Routine_Table where Prj_Id=ID
   end
   führe STMT aus
end
show_result()
```

Nachdem die Übergaben geparst wurden, soll festgestellt werden, ob die Auflistung durch eine gültige SQL-Anweisung oder durch eine ID-Nummer erfolgen soll. Falls es sich um eine SQL-Anweisung handelt, wird die SQL-Anweisung ausgeführt, die vom Anwender angegeben ist. Dann wird die daraus erstellte Ergebnisstabelle auf der Konsole angezeigt. Falls es aber um die ID-Nummer geht und als ID-Nummer * angegeben wird, werden alle Projekte aufgelistet. Wenn eine ID-Nummer eine gültige Zahl ist, werden alle Informationen zu dem Projekt mit dieser ID aufgelistet. Eine gültige SQL-Anweisung bei der Auflistung soll eine SELECT-Anweisung sein. Dies wird beim Parsen festgestellt, falls es nicht um eine SQL-Anweisung geht, wird die Auflistung nicht funktionieren. Falls die SQL-Anweisung intern falsch aufgebaut wurde, wird dies bei der Ausführung von SQL festgestellt und eine entsprechende Fehlermeldung angezeigt. Eine gültige ID soll dagegen entweder * oder die ID-Nummer eines Projekts, das in der Datenbank gespeichert ist, sein. Die Datenbankdatei soll immer konsistent sein, damit die Auflistung erfolgreich funktioniert. Die Konsistenzprüfung wird durch die Routine *Check* gewährleistet.

3.2.4.3 Insert

Die nächste Möglichkeit in relevant_stats, ist das Einfügen der Messwerte in die Datenbank. Das Einfügen kann entweder durch die IML-Datei oder CSV-Datei erfolgen.

```
*****************************************************************
* relevant_stats -db "insert <IML_FILE> | <CSV-FILE>" <DB_FILE> *
*****************************************************************
```

Falls die Messwerte aus der IML-Datei gelesen werden, muss darauf geachtet werden, welche IML-Datei ausgewählt wurde. Wenn als IML-Datei die Datei angegeben wird, die nach der Pointeranalyse (.pta Datei) erzeugt wurde, kann nicht erwartet werden, dass die Routinetabelle in der Datenbank zum Beispiel die Anzahl der Phi-Knoten von einer Routine enthält, da diese Knoten erst nach der SSA Analyse erzeugt werden können. Was man in diesem Fall erwarten kann, ist, dass diese Tabelle die Anzahl der Anweisungen einer Routine enthält. Wenn die Datei CSV Format hat, dann muss relevant_stats in diesem Fall zwei Tabellen Routine_Table und Project_Table hinzufügen. Aus diesem Grund müssen die beiden CSV-Dateien, die jeweils diese Tabellen darstellen, durch den Separator **;** angegeben werden. Diese CSV-Dateien sollen <ProjectName>_project_table.csv und <ProjectName>_routine_table.csv heißen. Falls eine dieser Tabellen aus irgendeinem Grund nicht hinzugefügt werden kann, wird die andere Tabelle auch nicht hinzugefügt, falls sie davor erfolgreich hinzugefügt wurde, wird sie bereingt Dies bezieht sich auf die Eigenschaft *Atomarität*. *Atomarität* ist eine der vier Eigenschaften des Transaktionskonzepts. Diese Eigenschaft verlangt, dass eine Transaktion als kleinste nicht mehr weiter zerlegbare Einheit behandelt wird, d.h. entweder werden alle Änderungen der Transaktion in der Datenbasis festgeschrieben oder gar keine [GM]. Dieses Prinzip wurde im Rahmen der Diplomarbeit in relevant_stats eingesetzt. Das bedeutet, dass das Einbringen der Werte in die Datenbank entweder vollständig oder gar nicht erfolgen soll. Die in der CSV-Datei dargestellten Tabellen müssen die Spalten mit ihren Namen enthalten, wie sie in der Datenbank gespeichert sind. Es können einige Spalten fehlen. Falls die CSV-Datei mindestens eine Spalte enthält, die nicht in der Datenbank der zuständigen Tabelle steht, wird der Einfügenprozess erfolglos beendet. Erst dann wird der Bereinigungsprozess gestartet.

Das Einfügen aus einer CSV-Datei ist auch aus dem Grund sinnvoll, wenn die Messwerte während der Analyse berechnet werden. Die Messwerte können dann im Ablauf berechnet und in eine CSV-Datei gespeichert werden. Um diese Messwerte in der Datenbank zu speichern, muss nicht aus der IML-Datei alles neu gemessen werden, sondern nur die Messwerte aus CSV-Dateien in die Datenbank übertragen werden. Im Rahmen der Diplomarbeit wurde das Tool relevant_stats auch als Schnittstelle implementiert und in SSA Analyse eingesetzt. Damit werden die Messwerte von SSA Analyse im Ablauf in die CSV-Dateien gespeichert. Weil das Tool relevant_stats das Einfuügen aus CSV-Datei anbietet, werden die Messwerte in die Datenbank recht einfach übertragen. Die Datenbankdatei, in der die Messwerte gespeichert werden, muss immer konsistent sein, sonst wird nichts in der Datenbank hinzugefügt.

3.2.4.4 Update

Es besteht die Möglichkeit in relevant_stats einige Informationen zu aktualisieren. Daher muss erst die ID-Nummer angegeben werden, die zu einem Projekt gehört, das in der Datenbank existiert. Die Aktualisierung wird für dieses Projekt durchgeführt. Es können in der Project_Table die Spalten *Name* und/oder *Comment* aktualisiert werden, falls als Key *Name* oder *Comment* eingegeben wird. Der Grund dafür ist, dass man von einem Projektverzeichnis mehrere Projekte erstellen kann. Das Projekt bezieht sich auf einen Durchlauf einer Analyse, um die Messwerte zu berechnen. Die Projekte in der Project_Table sind durch eine ID-Nummer identifiziert. Aus diesem Grund kann es sein, dass zwei Projekte mit gleichen Namen in der Datenbank sind. Daher gibt es die Möglichkeit durch Name und/oder Kommentar zwei dieser Projekte zu unterscheiden, und dadurch wiederum ein Projekt mit seiner Änderung zu vergleichen.

```
*************************************************************
* relevant_stats -db "update <ID>                           *
*           <KEY=VALUE> | <IML-FILE> | <CSV_FILES>" <DB_FILE> *
*************************************************************
```

Die weiteren möglichen Aktualisierungen sind in der Routine_Table möglich. Das bedeutet, dass die Messwerte auch aktualisiert werden. Falls die Messwerte nicht vollständig in der Datenbank hinzugefügt wurden, kann es durch ein *update* vervollständigt werden. Dies ist beispielsweise der Fall, wenn die Messwerte mit der IML-Datei in der Datenbank hinzugefügt werden, die nach der Pointeranalyse (.pta-Datei) erzeugt wurde. Daher können einige Metriken nicht in der Datenbank sein, wenn es um die SSA Analyse handelt. Aus diesem Grund gibt es die Möglichkeit, die fehlenden Metriken wie die Anzahl der Phi-Knoten in der Datenbank zu dem entsprechenden Projekt hinzuzufügen. Dies kann entweder durch eine entsprechende IML-Datei oder CSV-Datei erfolgen. Wenn die möglichen Metriken vor der Analyse in der Datenbank sind, könnte mit diesen Metriken eine Voraussage über den Zeitbedarf des Ablaufs der Analyse gemacht werden. Zumindest ist dies das Ziel der Idee.

Die CSV-Dateien sollen genauso aussehen, wie in *insert* erklärt wurde. Die Datenbankdatei muss konsistent sein, um irgendeine Aktualisierung durchführen zu können.

3.2.4.5 Delete

Des weiteren gibt es die Möglichkeit, ein Projekt aus der Datenbank zu löschen. Dafür ist die ID-Nummer des Projekts ausreichend.

```
**********************************************
* relevant_stats -db "delete <ID>" <DB_FILE> *
**********************************************
```

Durch *delete* wird das Projekt und die dazu gehörigen Routinen aus der Datenbank entfernt. Die Analyse Tabellen sind in der Datenbank als Sicht aus Routine_Table definiert. Dies hat zur Folge, dass alle Informationen zu jeder Analyse entfernt werden. Das Tool gibt keine Möglichkeit, nur die Metriken einer bestimmten Analyse aus der Datenbank zu entfernen, wenn es um mehrere Analysen (in normalen Fall) geht. Das ist die Atomarität (alles oder nichts Eigenschaft) der Datenbank. Wie bei den anderen Eigenschaften, muss die Datenbankdatei konsistent sein.

3.2.4.6 ToCSV

Die nächste Möglichkeit ist eine Art der Ausgabe aus der Datenbank zu erstellen. Es wird eine CSV-File aus der Datenbank erstellt.

```
*******************************************************************
* relevant_stats -db "toCSV <ID> | <SQL_STMT> <FileName>" <DB_FILE> *
*******************************************************************
```

Aus einem Projekt oder aus der, durch die aus der SQL-Anweisung erzeugte Tabelle werden CSV-Dateien bzw. eine CSV-Datei erstellt. Falls nur eine ID-Nummer angegeben wurde, die zu einem Projekt gehört, wird eine CSV-Datei für die Tabelle Project_Table mit dem Namen <PrjName>_Project_Table.csv und eine CSV-Datei für die Tabelle Routine_Table mit dem Name <PrjName>_Routine_Table.csv erstellt. Die zweite Möglichkeit ist die durch SQL erzeugte Tabelle in der CSV-Datei auszugeben. Daher sollte der Name der CSV-Datei angegeben werden. Die Datenbankdatei muss wie immer konsistent sein.

3.2.4.7 ShowTables

Die Eigenschaft *ShowTables* ist für den Anwender des Tools relevant_stats gedacht. Mit ShowTables werden für alle Datenbanktabellen folgende Informationen vorgestellt:

- Wie sie heißen
- Welche Spalten sie haben
- Wie sie aufgebaut sind (ob sie eine Sicht einer Tabelle darstellen)

Es ist bei vielen Aufgaben wichtig, den genauen Namen der Spalten einer bestimmten Tabelle zu wissen. Diejenigen, die mit dem Tool umgehen sollen, können mit ShowTables die Informationen zu den Tabellen haben, ohne auf den Quellcode zugreifen zu müssen.

3.3 Plot

Eine der wichtigsten Eigenschaften in relevant_stats ist die graphische Darstellung der Daten, die in der Datenbank stehen. Sie wird in dieser Ausarbeitung immer Plot genannt. Im Rahmen der Diplomarbeit wurde diese Eigenschaft eingeführt, um verschiedene Messwerte von einem Projekt bzw. mehreren Projekten graphisch darstellen zu können. Dadurch können verschiedene Projekte verglichen oder Informationen zu einem Projekt veröffentlicht werden. Diese graphische Darstellungen sollen genau in dem passenden Format Encapsulated PostScript (.eps) sein, damit sie in wissenschaftliche Veröffentlichungen eingebunden werden können. Die graphische Darstellung bezieht sich auf Diagramme, die verschiedene Messwerte darstellen. Die Abbildung 3.4 ist ein Beispiel für eine graphische Darstellung einer Routine. Mit dieser Darstellung sind alle Metriken einer Routine in einem Histogramm dargestellt.

Die graphische Darstellung wird in relevant_stats mit Hilfe von GNUplot realisiert. Daher wurde die C-Schnittstelle von GNUplot benutzt. Die Funktion zur Erstellung eines Histogramms wurde in C implementiert und dann in Ada importiert. Das Tool relevant_stats ruft nur diese Funktion an den entsprechenden Stellen auf, um die graphische Darstellung zu erzeugen. Das folgende Kapitel ist eine Einführung in GNUplot und die Probleme, die bei dieser graphischen Darstellung auftreten können. Anschließend gibt es eine Lösung für dieses Problem.

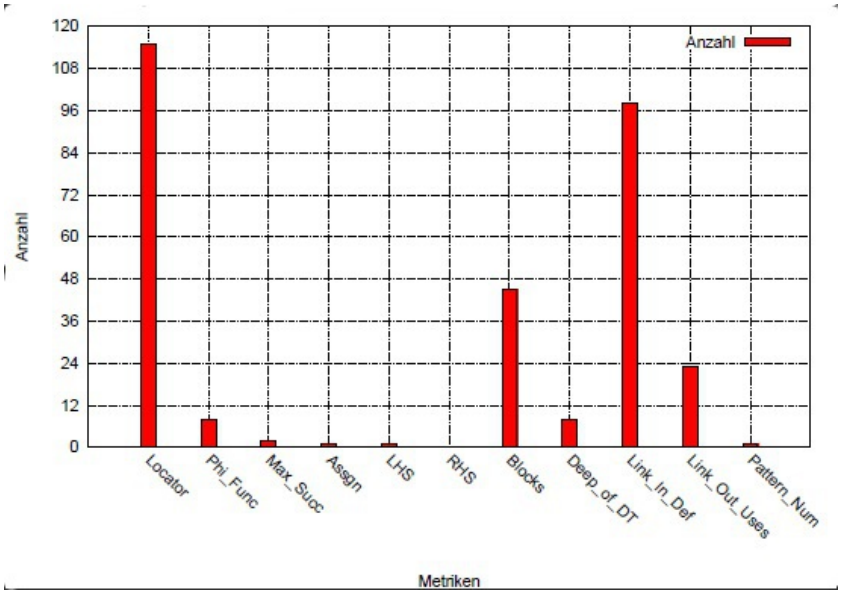


Abbildung 3.4: Beispiel zu einer graphischen Darstellung

3.3.1 GNUplot

GNUplot ist ein skript- bzw. kommandozeilengesteuertes Computerprogramm zur graphischen Darstellung von Messdaten und mathematischen Funktionen (Funktionenplotter). Das Projekt Gnuplot wird seit 1986 kontinuierlich von einem internationalen Team ehrenamtlicher Entwickler vorangetrieben. Der Quellcode wird seit 2000 über SourceForge verwaltet [Homa]. Mit Gnuplot lassen sich Funktionen und Daten (Messwerte) in den drei zweidimensionalen kartesischen Koordinatensystemen oder in einem 3D-Raum darstellen. Flächen werden dabei als Netzgittermodell im drei dimensionalen Raum dargestellt oder als Kontur in die x-y-Ebene projeziert. Für 2D-Darstellungen stehen zahlreiche Arten der Darstellung (Styles) zur Verfügung, wie z.B. Linien, Punkte, Linien mit Punkten und Balken. Die Kurven und Achsen können mit Werten versehen werden, Markierung, Überschriften, Datum-und Zeitangaben etc. sind möglich. Die Hauptanwendung besteht vor allem darin, Wertepaare, die man aus Berechnungen, Messungen etc. erhalten hat darzustellen. Zusätzlich zu dieser Aufgabe können beliebige Funktionen (Polynome, trigonometrische Funktionen ...) geplottet werden und Interpolationen durchgeführt werden, das heißt Polynome als best-fit durch gemessene Wertepaare gelegt werden. Zusätzlich zum üblichen kartesischen Koordinatensystem kennt Gnuplot auch polare Koordinatensysteme. 3D-Interpolation (gridding) zwischen unregelmässig

verteilten Datenpunkten (ein häufiges Problem) ist mit einem einfachen Gewichtungsverfahren möglich. Andere Programme (z.B. Surfer) bieten in dieser Hinsicht zwar weit mehr Möglichkeiten, aber für viele Ansprüche dürfte es bereits genügen [Homb].

GNUplot ist ein kommandozeilen-orientiertes, interaktives Programm, so beim Aufruf erscheint ein Kommandofenster mit folgendem Prompt: **(gnuplot>)**. Es verlangt eine Eingabe, z.B. einen der Plotbefehle, die Definition einer Funktion, Angaben zur Formatierung einer Achse oder etwas Ähnliches. Ist eine Eingabe erfolgt, wird sie ausgeführt, was entweder sofort bemerkt wird (bei einem Plotbefehl) oder sich später auf folgende Plotbefehle auswirkt (bei Funktionsdefinitionen z.B.)

```
gnuplot> a=0.25
gnuplot> f(x)=sin(x)*a
gnuplot> plot f(x)
```

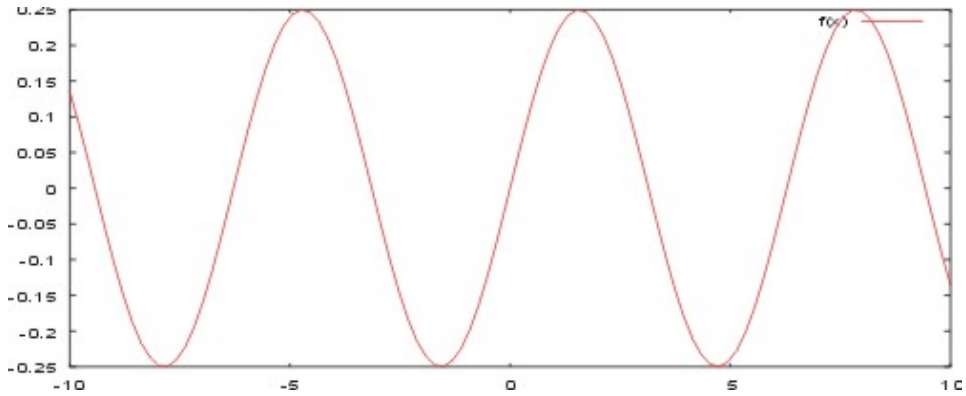


Abbildung 3.5: Ausgabe der Plot-Befehle

Praktischerweise wird man diese Befehlsfolgen nicht immer wieder neu eingeben, sondern in eine Stapeldatei oder Script-Datei (eine einfache Textdatei) schreiben, die dann aufgerufen werden kann (Befehl: load 'Dateiname') und, die die Befehlsabfolge in Gang setzt. So kann man eine Datei immer weiter editieren, z.B. um die Achsenlänge zu verändern, das selbe Grafikformat auf unterschiedliche Daten loszulassen etc.

Es gelten schon einige vordefinierte Standard-Einstellungen über den Stil der Abbildung z.B. dass die erste Grafik in rot gehalten wird etc. Angezeigt werden die gültigen Einstellungen mit dem Befehl *show all*. Diese Grundeinstellungen stehen in einer Datei

namen .gnuplot (UNIX, LINUX), bzw. GNUPLOT.INI(Windows). Diese defaults kann man natürlich überschreiben. Der prinzipielle Befehl, um Änderungen an Achsen, Label, Wertebereiche etc. vorzunehmen lautet *set*. Eine Änderung wirkt sich solange aus, bis man die Einstellungen neu festgelegt hat. Mit reset kehrt man zu den Standardeinstellungen zuück.

Im Rahmen der Diplomarbeit wurde die C-Schnittstelle(*gnuplot_i.c und gnuplot_i.h*) von GNUplot zur graphischen Darstellung benutzt; daher nun eine kleine Einführung in diese Schnittstelle. Die C-Schnittstelle von GNUplot enthält ein Handle-Objekt *gnuplot_ctrl*, das alle Informationen in sich hält. Dieses Objekt wird durch die Funktion *gnuplot_init()* initialisiert und bei fast jeder Funktion in diesem Module übergeben werden. Die Freigabe dieses Objekts erfolgt durch die Funktion *gnuplot_close*.

```
typedef struct _GNUPLOT_CTRL_ {
    /** Pipe to gnuplot process */
    FILE    * gnucmd ;

    /** Number of currently active plots */
    int       nplots ;
    /** Current plotting style */
    char      pstyle[32] ;

    /** Name of temporary files */
    char      to_delete[GP_MAX_TMP_FILES][GP_TMP_NAME_SIZE] ;
    /** Number of temporary files */
    int       ntmp ;
} gnuplot_ctrl ;
```

Es besteht die Möglichkeit mit der Funktion *gnuplot_set_xlabel* bzw. *gnuplot_set_ylabel* X bzw. Y Achsen zu benennen. Der Style eines Plots wird durch die Funktion *gnuplot_setstyle* bestimmt. Die mögliche Stilart ist als String(char *) anzugeben und muss eine der folgenden Styles sein:

- lines
- points
- linespoints
- impulses
- dots

- steps
- errorbars
- boxes
- boxeserrorbars

Die weitere Funktion, die in relevant_stats häufig angewandt wurde, ist *gnuplot_cmd*. Durch diesen Befehl kann eine direkte Verbindung gesetzt werden. Das bedeutet beispielsweise, um einen Style zu setzen, wird die Funktion *gnuplot_cmd* mit dem String **set style ...** aufgerufen, anstatt *gnuplot_setstyle* mit dem entsprechenden Style aufzurufen. Der Nachteil von *gnuplot_cmd* ist, dass die Fehlermeldungen von GNUplot nicht von der Schnittstelle abgefangen werden. Mit *gnuplot_resetplot* kann die Behandlung zurückgesetzt werden. Es wird nun ein kleines Programmstück von der C-Schnittstelle dargestellt.

```
void Plot_Me()
{
    gnuplot_ctrl    *Handle;

    Handle = gnuplot_init() ;
    gnuplot_cmd (Handle, "set terminal postscript eps color");
    gnuplot_cmd (Handle, "set output \"my_output.eps\"");
    gnuplot_cmd (Handle, "set boxwidth 1.5 absolute");
    gnuplot_cmd (Handle, "set style fill solid 1.00 border -1");
    gnuplot_cmd (Handle, "set style histogram gap 5");
    gnuplot_cmd (Handle, "set style data histograms");
    gnuplot_cmd (Handle, "set xtic rotate by -45 scale 0");
    gnuplot_cmd (Handle, "set ytics 100 nomirror");
    gnuplot_cmd (Handle, "set xtics  (\"Function-1\" 0, \"Function-2\"
         1)");
    gnuplot_cmd (Handle, "set ytics 2");
    gnuplot_cmd (Handle, "set yrange [0:100]");
    gnuplot_cmd (Handle, "set xlabel  "x achse");
    gnuplot_cmd (Handle, "set ylabel "y achse");
    gnuplot_cmd (Handle, "plot 'data' using 2 t \"Anzahl\"");
    gnuplot_close(Handle) ;
}
```

3.3.2 Orientierung

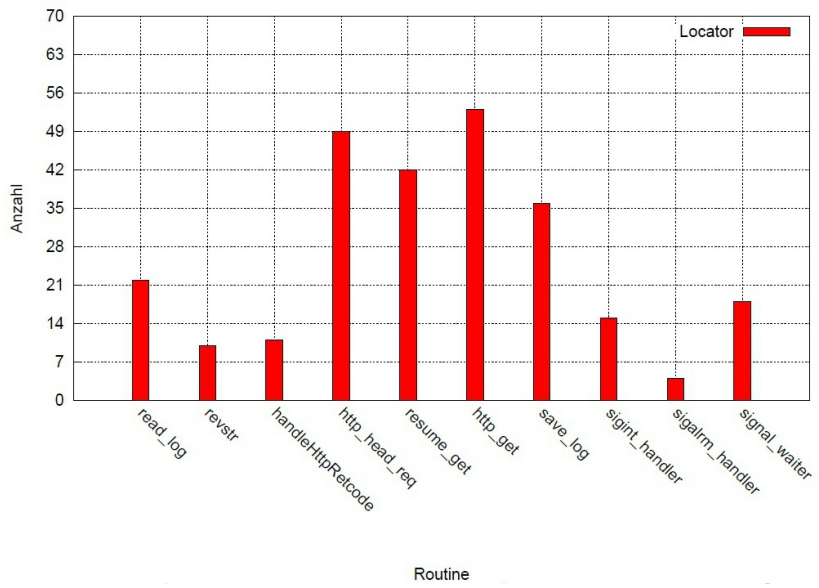
Es soll in relevant_stats die Möglichkeit bestehen, dass eine (oder mehrere) graphische Darstellung aus den Metriken erstellen, die in der Datenbank stehen. Die Erstellung graphischer Darstellung ist nur durch eine SQL-Anweisung möglich, die aus SELECT-Anweisung besteht. Nachdem eine SQL-Anweisung durchgeführt wurde, wird eine Ergebnistabelle im Hintergrund gebildet. Das Tool relevant_stats soll aus dieser Tabelle die entsprechende graphische Darstellung erzeugen. Das Problem ist, wie soll so eine Ergebnistabelle bewertet werden, wenn der Anwender aus dieser Tabelle graphische Darstellung haben möchte. Die Abbildung 3.2 zeigt ein Beispiel zu einer Ergebnistabelle, die nach einer SQL-Anweisung erstellt wurde. Es ist auch gleich zu erkennen, dass es sich nur um ein Projekt mit der ID-Nummer 0 handelt.

Name	Prj_Id	Locator	Phi_Func	Assgn	Blocks
read_log	0	22	1	2	12
revstr	0	10	2	3	8
handleHttpRetcode	0	22	1	2	12
http_head_req	0	49	3	6	19
resume_get	0	42	6	4	29
http_get	0	53	6	9	42
save_log	0	36	0	3	8
sigint_handler	0	15	1	0	6
sigalrm_handler	0	4	0	0	3
signal_waiter	0	18	2	1	11

Tabelle 3.2: Beispieltabelle

Die Lösung für dieses Problem, die Im Rahmen der Diplomarbeit gefunden und implementiert wurde, hat den Zweck, mit wenigem und einfachem Aufwand möglich umfangsreiche bzw. gezielte Informationen aus der Datenbank graphisch darstellen zu können. Mit dieser Lösung kann der Anwender mit einer einzigen SQL-Anweisung (bzw. SELECT-Anweisung) eine graphische Darstellung haben, die beispielsweise alle Metriken von einer Routine oder die Anzahl einer bestimmten Metrik von verschiedene Routine enthält. Damit ist es viel einfacher, Informationen zu einer Routine zu ermitteln bzw. die Routine zu vergleichen.

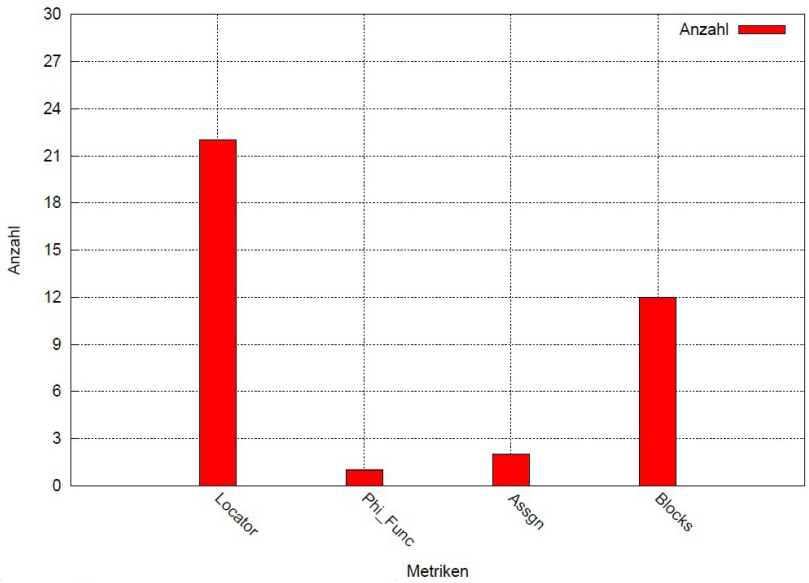
Es bestehen drei mögliche Interpretation einer Ergebnistabelle, wenn es nur um ein Projekt geht. Eine Möglichkeit wäre, dass der Anwender jede Metrik in verschiedenen Routinen vergleichen möchte. Diese Möglichkeit wird in dieser Ausarbeitung spaltenorientierte Ansicht genannt. Die folgende Abbildung ist eine graphische Darstellung von der Spalte *Locator*. Dies ist eine graphische Darstellung, in dem von allen Routinen, die in der Ergebnistabelle existieren, die Anzahl der Locator angezeigt wird. Es ist zu beachten, falls es um die spaltenorientierte Ansicht geht, muss die Spalte *Name* immer da sein. Mit Hilfe der Spalte *Name* wird in der graphischen Darstellung angedeutet, zu welcher Routine diese Werte gehören. Die zweite Spalte, die unbedingt auch dabei sein soll, ist *Prj_Id*. Es muss bei jeder SQL-Anweisung festgestellt werden, ob es um eine bzw. mehrere Projekte geht. Dies kann man nur mit der Hilfe der Spalte *Prj_Id* feststellten. In relevant_stats gibt es eine interne Prüfung, ob eine SQL-Anweisung an dieser Stelle die Spalten *Name* und *Prj_Id* enthält. Falls die Anweisung sie nicht enthält, wird sie vervollständigt. Aus diesem Grund ist es kein Problem, wenn die Spalten *Name* und *Prj_Id* fehlen. Die Plot-Dateien bekommen den Name jeder Spalte. In unserem Beispiel heißt die Datei Locator.eps. Das Tool relevant_stats nicht in der Lage zu entscheiden, welche Spalte(n) angesprochen werden soll(en). Aus diesem Grund wird jeweils für jede Spalte die entsprechende graphische Darstellung erzeugt. D.h. die Dateien Phi_Func.eps, Assgn.eps, Blocks.eps stehen in unserem Beispiel auch zur Verfügung.



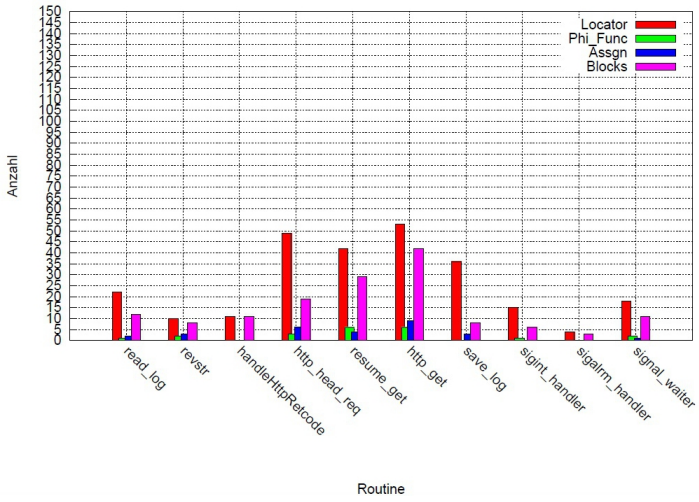
Die zweite Möglichkeit bei der graphischen Darstellung ist, dass der Anwender jeweils

für jede Routine alle in der Tabelle stehende Informationen bzw. Messwerten in einer graphischen Darstellung haben möchte. Dieser Fall nennt sich in dieser Ausarbeitung *zeilenorientierte Ansicht*.

Es wird für jede Zeile eine Plot-Datei erstellt. Es ist ganz wichtig in der SQL-Anweisung die Spalten *Name* und *Prj_Id* zu haben, weil die Datei <Name der Routine>.eps heißt. Falls die SQL-Anweisung diese Spalten nicht enthält, wird sie wieder vervollständigt. Wiederum wird dieser Prozess wie vorher für alle Zeilen angewandt. D.h. Es gibt dann für jede Routine eine Plot-Datei mit den entsprechenden Informationen. Mit der Spalte *Prj_Id* wird es festgestellt, ob es um einen Projekt bzw. mehrere Projekte geht. Es folgt ein Beispiel zu zeilenorientierter Darstellung.



Die letzte Möglichkeit zur Darstellung besteht aus beiden vorherigen Ansichten. Das heißt, der Anwender möchte alle in der Tabelle existierenden Metriken von verschiedenen Routinen vergleichen. Diese Ansicht ist sowohl spaltenorientiert als auch zeilenorientiert (BOTH). In der folgenden Abbildung gibt es ein Beispiel von einer Darstellung, die beide Ansichten (BOTH) enthält. Die Spalten *Name* und *Prj_Id* müssen immer in der Tabelle existieren, sonst wird sie zusätzlich hinzugefügt. Der Name der Plot-Datei ist immer out.eps.



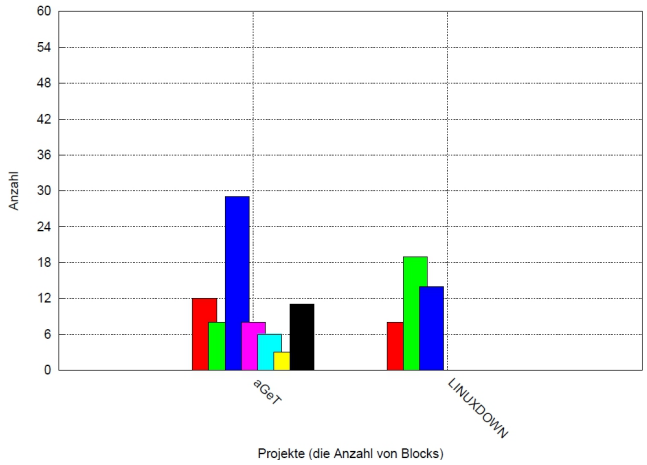
Es gibt eine andere mögliche Tabelle, die nicht nur aus einem Projekt besteht. In diesem Fall gibt es wieder drei mögliche Interpretation.

Name	Prj_Id	Locator	Phi_Func	Assgn	Blocks
read_log	0	22	1	2	12
revstr	0	10	2	3	8
resume_get	0	42	6	4	29
save_log	0	36	0	3	8
sigint_handler	0	15	1	0	6
sigalrm_handler	0	4	0	0	3
signal_waiter	0	18	2	1	11
ReadEn	1	9	2	5	8
real_calc_response	1	18	7	6	19
SaveThreadStat	1	6	3	1	14

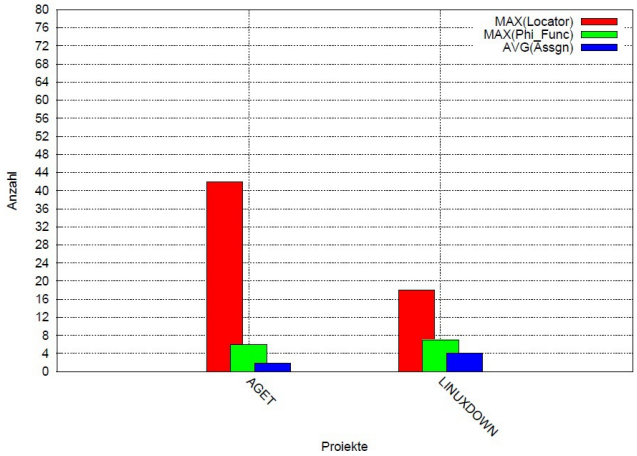
Tabelle 3.3: Beispieltabelle-2

Die erste mögliche Interpretation ist die spaltenorientierte Hinsicht. Durch diese Implementierung kann der Anwender eine graphische Darstellung von jeder Spalte haben. Aber die Routinen werden nach Prj_Id gruppiert dargestellt. In jeder Gruppe werden, wie im normalen Fall, alle Routinen dargestellt. Dadurch kann man verschiedene Projekte miteinander vergleichen. In der folgenden Darstellung wurde ein Beispiel zu der

spaltenorientierten Hinsicht für die vorherige Tabelle dargestellt, die mehrere Projekte enthält. Die daraus erstellte Plot-Datei heißt wieder wie der Name der zugehörigen Spalte.



Die zweite Möglichkeit ist die zeilenorientierte Hinsicht. Dies funktioniert genau so als ginge es um ein Projekt. Die letzte mögliche Darstellung besteht aus sowohl spaltenorientierter, als auch zeilenorientierter Hinsicht (Both). Jede Zeile wird mit allen zugehörigen Spalten gruppiert dargestellt. In der folgenden Abbildung ist ein Beispiel für die Darstellung *Both*. In dem Beispiel wird die maximale Anzahl des Lokators, maximale Anzahl der Phi-Knoten und durchschnittliche Anzahl der Zuweisungen in beiden Projekten verglichen.



Nachdem die Orientierungen geklärt wurde, kann jetzt darauf eingegangen werden, wie Plot in relevant_stats aufgebaut ist.

```
************************************************************
* relevant_stats -db "plot <ORIENT> <SQL_STMT>" <DB_FILE>  *
************************************************************
```

Hiermit bezieht sich <ORIENT> auf die Hinsichten bwz. Orientierungen, die zuvor vorgestellt wurden. Die Orientierung kann *COLUMN*, *ROW* oder *BOTH* sein. Die Orientierung *COLUMN* bezieht sich auf die spaltenorientierte Hinsicht, während *ROW* sich auf die zeilenorientierte Hinsicht bezieht. *BOTH* enthält dagegen sowohl *COLUMN* als auch *ROW*. D.h. falls es um eine graphische Darstellung geht, soll es sich entschieden werden, um welche Ansicht es sich handelt. Daher ist für diese Aktion empfehlenswert, erst die Ergebnisstabelle von einer SQL-Anweisung mit dem Befehl *list* aufzulisten, damit die SQL-Anweisung in dem ersten Schritt getestet werden kann und auch um die Entscheidung zu treffen, um welche Ansicht es gehen soll.

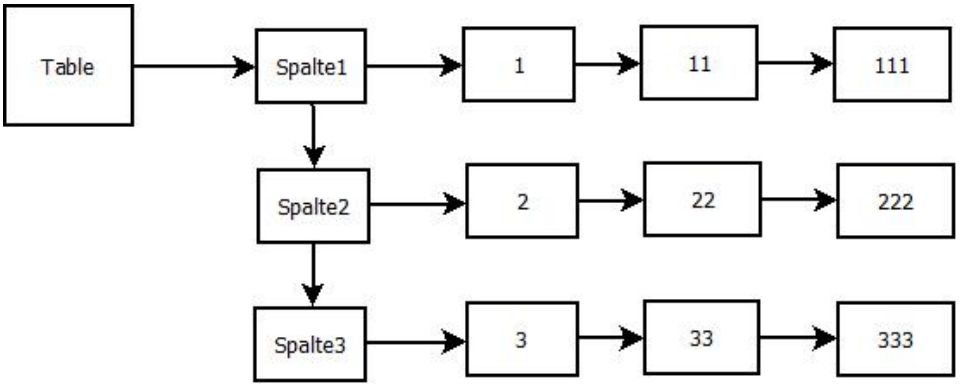
3.4 Implementierung

In diesem Kapitel wird erläutert, wie relevant_stats implementiert wurde. Das Tool relevant_stats hat zwei wichtige Objekte *Table* und *T_Project*. Das Objekt *Table* repräsentiert eine Tabelle. Dieses Objekt wurde in Datenbankoperationen immer verwendet. Wenn eine SQL-Anweisung durchgeführt wird, wird die daraus entstehende Tabelle in das Objekt *Table* gespeichert. Das Objekt *Table* kann eine sowohl spaltenorientierte, als auch zeilenorientierte Tabelle enthalten.

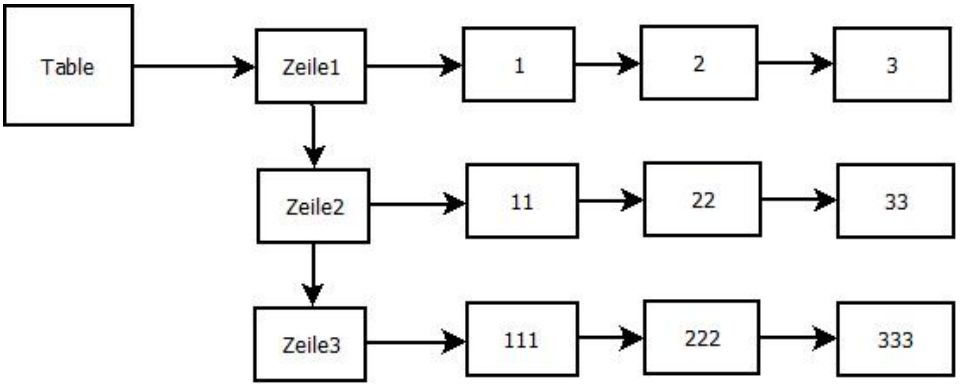
Spalte1	Spalte2	Spalte3
1	2	3
11	22	33
111	222	333

Tabelle 3.4: Beispieltabelle-3

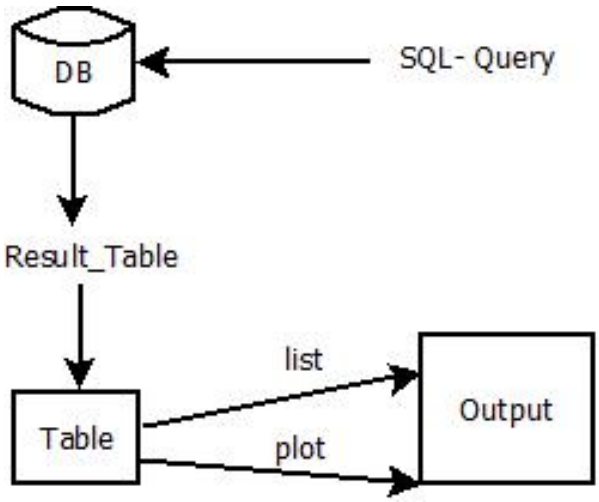
Das bedeutet beispielsweise, wenn die folgende Tabelle als Ausgabe erzeugt wird, kann diese Tabelle in dem Objekt *Table* wie im Folgenden spaltenorientiert gespeichert werden.



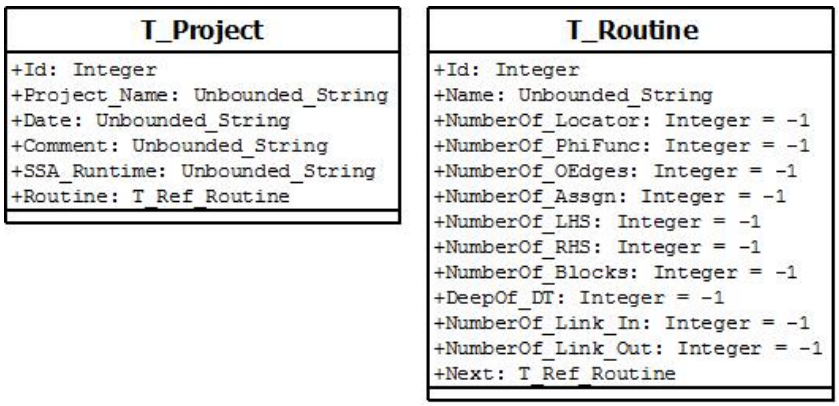
Wenn diese Tabelle zeilenorientiert gespeichert werden soll, wird sie wie im folgenden Beispiel gespeichert.



Die Tabellen können gleichzeitig zeilenorientiert und spaltenorientiert gespeichert werden. Zusätzlich enthält das Objekt *Table* die Informationen, wie die Anzahl der Spalten und Zeilen sein soll. Das Objekt *Table* besteht aus *Colum* bzw. *Row* Objekten, die zuständig für die Spalten bzw. Zeilen sind. Diese Objekte enthalten das Objekt *Value*. Das Objekt *Column* bzw. *Row* enthält die Informationen, wie die Nummer der Spalte bzw. Zeile, die maximale Länge einer Spalte bzw. Zeile, der Name der Spalte sein soll. Das Objekt *Value* repräsentiert einen Wert an einer bestimmten Position in der Tabelle. Dieses Objekt hat die Informationen wie der Datentyp einer Spalte.



Das Objekt *Table* wurde in relevant_stats bei der Auflistung (list) und graphischer Darstellung (plot) benutzt. Die beiden Funktionalitäten erfolgen durch eine SQL-Anweisung. Nach dieser SQL-Anweisung bekommt das Tool relevant_stats eine Tabelle als Ergebnis, falls diese Anweisung erfolgreich durchgeführt wurde. Diese Tabelle wird in das Objekt *Table* umgewandelt. Mit Hilfe vom Objekt werden die Auflistung und die graphische Darstellung einfacher. Das Objekt *Table* ist in der Lage spaltenweise und/oder zeilenweise die Werte durchzulaufen. Dies hilft bei der graphischen Darstellung viel, da sie entweder zeilenweise oder spaltenweise (oder kombiniert) erfolgen soll.



Das zweite Objekt in relevant_stats, das eine wichtige Rolle spielt, ist *T_Project*. Dieses Objekt repräsentiert das Datenbankmodell des Tools relevant_stats. Dieses Objekt hat

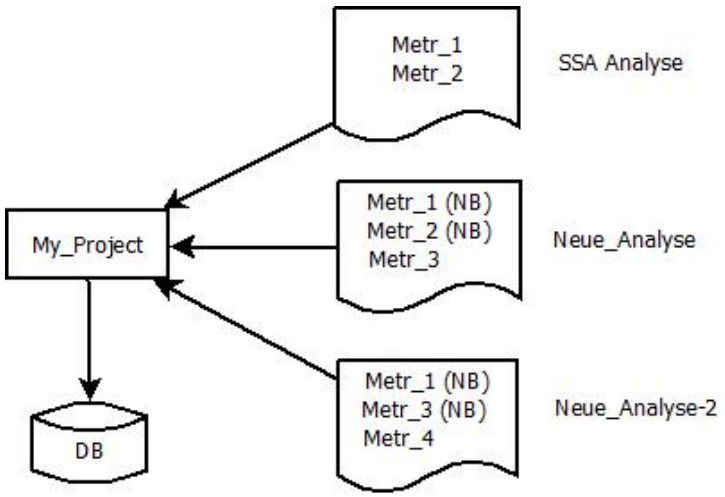
einen Verweis auf das Objekt *T_Routine*. Die beiden Objekte enthalten alle Spalten der Datenbanktabellen *Project_Table* und *Routine_Table*. Ein *T_Project* kann viele *T_Routine* enthalten.

Wie in der vorherigen Abbildung erkennbar ist, sind die Metriken aus der SSA-Analyse in dem Object *T_Routine* gespeichert, weil zurzeit mit dem Tool relevant_stats nur die SSA-Analyse untersucht wird. Falls die neue Analyse hinzugefügt werden soll, muss das Objekt *T_Routine* entsprechend erweitert werden. Diese Objekte sind in dem Paket Analysis als *private* deklariert und es gibt zu jedem Attribut set und get Funktionalitäten.

```
---------------------
-- Set_NO_LSH      --
---------------------
procedure Set_NO_LHS
  (pRoutine : Ref_T_Routine;
   pNO_LHS  : Integer);

---------------------
-- Get_NO_LHS      --
---------------------
function Get_NO_LHS
  (pRoutine: Ref_T_Routine)
 return Integer;
```

Diese Objekte werden benutzt, falls ein neues Projekt beispielsweise in die Datenbank durch *Insert* hinzugefügt wird. Zuerst müssen die Metriken für jede Routine aus IML-Datei gemessen werden (bzw. aus CSV-Dateien gelesen werden), die in dem zu untersuchenden Projekt existieren. Daher werden alle Messwerte und entsprechenden Informationen wie der Name der Routine oder der Kommentar des Projekts in das Objekt *T_Project* gepackt. Das Hinzufügen in die Datenbank erfolgt erst danach mit Hilfe von diesen Objekten. Der gleiche Prozess hat das Tool relevant_stats, falls die Messwerte aus den CSV-Dateien bzw. aus IML-Datei aktualisiert werden sollen. Sie werden zuerst in das entsprechende Objekt gespeichert. Erst dann können die Werte mit Hilfe von diesen Objekten in die Datenbank übertragen werden.



Diese Objekte dienen dann auch der Erleichterung der Erweiterbarkeit des Tools relevant_stats. Falls eine neue Analyse in relevant_stats hinzugefügt wird, ist es beispielsweise für die Funktionalität *insert* nur einen Aufruf ausreichend, der die Objekte durch die entsprechenden (neue) Messwerte aktualisiert. Falls irgendeine Messwerte vorhanden ist, weil sie zu einer vorhandenen Analyse gehört, soll sie gar nicht aktualisiert werden. Dieser Aufruf soll nur die neuen Metriken messen. In der vorherigen Abbildung gibt es ein Beispiel zu diesem Thema. Dabei soll **NB** heißen, dass diese Messwerte nicht nochmal berechnet werden müssen, weil sie in vorherigen Aufruf schon berechnet wurden. Eine weitere Diskussion auf die Erweiterbarkeit wird im Kapitel 4 erläutert.

4 Vorgehensweise für eine neue Analyse

Ein modularer Aufbau des Systems und somit leichte Erweiterbarkeit spielen im Software Engineering eine starke Rolle. Systeme, die eingesetzt werden, müssen dann korrigiert, vor allem den sich ändernden Anforderungen angepasst werden. Auch wenn man beim Architekturenentwurf die Zukunft nicht genau kennt, kann man doch Bereiche des Systems, die wahrscheinlich angepasst oder erweitert werden müssen, so entwerfen, dass der Aufwand dafür akzeptabel ist. Im Rahmen der Diplomarbeit wurde das Tool relevant_stats dementsprechend implementiert. In dieser Ausarbeitung wurde die SSA-Analyse als Beispiel eingesetzt. Es ist offensichtlich klar, dass das Tool relevant_stats um eine oder mehrere Analyse erweitert wird. Deswegen sollte das Tool in der Lage sein, mit neuen Analysen umgehen zu können. Aus diesem Grund habe ich mir darüber Gedanken gemacht, wie dieses Tool implementiert werden kann, damit die Erweiterung um eine neue Analyse mit geringem Aufwand geht. Geringer Aufwand soll damit heißen, mit möglichst wenigen Änderungen im Code alle Funktionalitäten beizubehalten, die das Tool relevant_stats zurzeit mit der SSA-Analyse ermöglicht. In dieser Ausarbeitung wird es erläutert, wie eine neue Analyse in relevant_stats hinzugefügt werden kann. Daher möchte ich in diesem Kapitel eine Vorgehensweise vorlegen, in dem alle Schritte mit einer Beispielanalyse aufgelistet werden.

My_Analysis sei eine neue Analyse, die in relevant_stats eingesetzt werden soll. Angenommen sind die Faktoren *Metrik-1*, *Metrik-2* und *Metrik-3*, die einen wesentlichen Einfluss auf die Analyse *My_Analysis* haben. Diese Metriken haben keinen Einfluss auf andere Analyse, die im Bauhaus implementiert sind. Es sollen die folgenden Änderungen durchgeführt werden.

4.1 Änderungen in globals.ads

Die Datei *globals.ads* enthält alle globalen Variablen, die in relavant_stats verwendet werden. Sie sind meistens als Unbounded_String definiert. Sie lassen sich in folgenden Teilen trennen:

- Die Namen der Datenbanktabellen wie Routine_Table, Project_Table und SSA_Table.
- Die in relevant_stats erlaubte Typen wie INT und CHAR, wobei INT Integer und CHAR varchar(100) in SQL entsprechen.
- Datenbankoperationen sind die SQL-Schlüsselwörter, die in relevant_stats häufig benutzt werden. wie O_SELECT (select in SQL), O_VIEW (view in SQL) und noch viele weitere.
- Die Datenbanktabellen, die in der Datenbank existieren. Sie sind als Array definiert. Für jede Analyse existieren zwei Arrays. Eins davon ist für die Abbildung der Tabelle zuständig und enthält alle Spalten der Tabelle, die zuständiger Analyse gehört. Das zweite Array beinhaltet die Datentypen der Spalten dieser Analyse. Es gibt beispielsweise für SSA Analyse eine Tabelle mit dem Namen SSA_Table. Um diese Tabelle in der Datenbank abzubilden gibt es die Arrays SSA_Table_Rows und SSA_Table_Types. SSA_Table_Rows enthält die Spalten der Tabelle SSA_Table (sowie *Id*, *Prj_Id*, *Name*, *Locator*), wobei SSA_Table_Rows die Typen dieser Spalten (sowie INT, INT, CHAR, INT) beinhaltet.

für die neue Analyse muss im ersten Schritt *My_Analysis* der Name dieser Tabelle definiert werden, die *My_Analysis* in der Datenbank repräsentiert. In unserem Beispiel wird die Tabelle als *MyAnalysis_Table* genannt.

```
Project_Table    : constant Unbounded_String := Str("Project_Table");
Routine_Table    : constant Unbounded_String := Str("Routine_Table");
SSA_Table        : constant Unbounded_String := Str("SSA_Table");
MyAnalysis_Table :  constant Unbounded_String := Str("MyAnalyse_Table");
```

Im nächsten Schritt sind die Spalten und Typen dieser Spalten zu definieren. Aus diesem Grund sind die Metriken, die zur neuen Analyse *My_Analysis* gehören, müssen zuerst bestimmt werden. Sie sind in unserem Beispiel *Metrik_1*, *Metrik_2* und *Metrik_3*. Die Tabelle *MyAnalysis_Table* hat dann die Spalten *Metrik_1*, *Metrik_2*, *Metrik_3* und

zusätzlich die Spalten *Id*, *Prj_Id*, *Name*. Die Spalten *Id* der Id-Nummer, *Prj_Id* des Projekt-IDs , *Name* dem Name der zuständigen Routine entsprechen. Auf diese zusätzlichen drei Spalten kann verzichtet werden. Sowohl bei der Erstellung graphischer Darstellung als auch bei der Auflistung der Messwerten dienen sie zur Erleichterung einer SQL-Anweisung. Während die Messwerte beispielsweise in relevant_stats graphisch dargestellt werden, müssen bei der SQL-Anweisung die Join-Operation zwischen der Tabellen MyAnalysis_Table, Project_Table, Routine_Table durchgeführt werden, wenn die Spalte Prj_Id nicht existeren sollte, um auf die Messwerte eines bestimmten Projekts zugreifen zu können. Das gleiche gilt näturlich für die anderen Spalten. Mit unserer Lösung besteht die Möglichkeit, aus der MyAnalysis_Table die Informationen zu lesen, zu welchem Projekt und zu welcher Routine diese Messwerte gehören, ohne eine Join-Operation durchführen zu müssen. Im Folgenden gibt es SQL-Anweisung, um die Messwerte zu einer Routine *HelloWorld* aufzulisten, die zum Projekt 1 gehört.

```
SELECT * from MyAnalysis_Table
    WHERE Prj_Id = 1 AND
          Name   = 'HellWorld'
ANSTAT:
SELECT * from MyAnalysis_Table M,
              Routine_Table R,
              Project_Table P
    WHERE P.Id = 1 AND
          R.Name   = 'HellWorld'
```

Für die Spalten und Typen der Spalten von der Tabelle *MyAnalysis_Table* muss erst ein Typ *MyAnalysis_Rows* definiert werden. Da *MyAnalysis_Table* in unserem Beispiel aus 6 Spalten besteht, muss unser Typ *MyAnalysis_Rows* auch aus 6 Felder bestehen. Dieser Typ ist wie im Folgenden definiert.

```
type Project_Rows is array(1..6) of Unbounded_String;
type Routine_Rows is array(1..17) of Unbounded_String;
type SSA_Rows  is array(1..14) of Unbounded_String;
type MyAnalysis_Rows is array(1..6) of Unbounded_String;
```

Die Arrays, die die Tabelle *MyAnalysis_Table* darstellen, können dann wie im Folgenden definiert werden. Sie sind vom Typ *MyAnalysis_Rows*. In *MyAnalysis_Table_Rows* werden die Spalten dargestellt, während *MyAnalysis_Table_Types* die Typen dieser Spalten enthält. Die Typen können entweder INT oder CHAR, die in globals.ads definiert sind. Es muss auch bestimmt werden, ob die Metriken in Routine_Table schon existieren. Das ist nur der Fall, wenn eine der Metriken von *My_Analysis* auch zu einer

anderen Analyse gehört, die schon in relevant_stats existiert. Dann müssen die Metriken, die nur zur *My_Analysis* gehören, in Routine_Table auch hinzugefügt werden. Eine Metrik, die auch zur einer anderen Analyse gehört, müsste schon in der Routine_Table existieren. In unserem Beispiel ist es davon ausgegangen, dass die Metriken *Metrik_1*, *Metrik_2* und *Metrik_3* nur zur *My_Analysis* gehören. Aus diesem Grund müssen sie in Routine_Table hinzugefügt werden. Die Tabellen, die zu den Analysen gehören, werden immer als Sicht (View) definiert, die aus den Spalten der Tabelle Routine_Table besteht. alle Metriken, die zu irgendeiner Analyse gehören, müssen aus diesem Grund zusätzlich in Routine_Table existieren. Es wird in unserem Beispiel eine Spalte *MyAnalysis_Runtime* in Project_Table hinzugefügt, um zu speichern, wie lange die Analyse *MyAnalysis* dauert.

```
Project_Table_Rows : Project_Rows
                 := (Str("Id"),
                     Str("Name"),
                     Str("Date"),
                     Str("Comment"),
                     Str("SSA_Runtime"),
                     Str("MyAnalysis_Runtime")
                    );

Project_Table_Types : Project_Rows
                 := (INT, CHAR, CHAR, CHAR, CHAR, CHAR);

Routine_Table_Rows : Routine_Rows
                    := (Str("Id"),
                        Str("Prj_Id"),
                        Str("Name"),
                        Str("Locator"),
                        Str("Phi_Func"),
                        Str("Max_Succ"),
                        Str("Assgn"),
                        Str("LHS"),
                        Str("RHS"),
                        Str("Blocks"),
                        Str("Deep_of_DT"),
                        Str("Link_In_Def"),
                        Str("Link_Out_Uses"),
                        Str("Pattern_Num"),
                        Str("Metrik-1"),
                        Str("Metrik-2"),
                        Str("Metrik-3")
                      );
```

```
Routine_Table_Types : Routine_Rows
                     := (INT, INT, CHAR, INT, INT, INT,
                         INT, INT, INT, INT, INT, INT,
                         INT, INT, INT, INT, INT);

MyAnalysis_Table_Rows :  MyAnalysis_Rows
                 := (Str("Id"),
                     Str("Prj_Id"),
                     Str("Name"),
                     Str("Metrik-1"),
                     Str("Metrik-2"),
                     Str("Metrik-3")
                    );

MyAnalysis_Table_Types :  MyAnalysis_Rows
                 := (INT, INT, CHAR, INT, INT, INT);
```

4.2 Änderungen im Paket *Analysis*

Die Objekte *T_Project* und *T_Routine* sind im Paket *Analysis* vorhanden. Wenn eine neue Analyse hinzugefügt werden soll, sollen die Objekte in Analysis.ads um die neue Metriken der Analyse erweitert werden. In unserem Beispiel wird das Objekt *T_Project* um die Metrik *MyAnalysis_Runtime* und das Objekt *T_Routine* um die Metriken *Metrik_1*, *Metrik_2* und *Metrik_3* erweitert.

```
type T_Project is record
  Id           : Integer          := -1;
  Project_Name : Unbounded_String := Str("NULL");
  Routine      : Ref_T_Routine    := NULL;
  Date         : Unbounded_String := Str("NULL");
  Comment      : Unbounded_String := Str("NULL");
  SSA_Runtime  : Unbounded_String := Str("NULL");
  MyAnalysis_Runtime :  Unbounded_String := Str("NULL");
  Next         : Ref_T_Project       := NULL;
end record;

type T_Routine is record
  Id               : Integer          := -1;
  Routine_Name     : Unbounded_String := Str("NULL");
  Name             : Unbounded_String := Str("NULL");
```

```
    -- Metrics
    NumberOf_Locator : Integer := -1;
    NumberOf_PhiFunc : Integer := -1;
    NumberOf_OEdges  : Integer := -1;
    NumberOf_Assgn   : Integer := -1;
    NumberOf_LHS     : Integer := -1;
    NumberOf_RHS     : Integer := -1;
    NumberOf_Blocks  : Integer := -1;
    DeepOf_DT        : Integer := -1;
    NumberOf_Link_In : Integer := -1;
    NumberOf_Link_Out: Integer := -1;
    Pattern_Number   : Integer := -1;
    Metrik_1         :  Integer := -1;
    Metrik_2         :  Integer := -1;
    Metrik_3         :  Integer := -1;
    -- pointer
    Next              : Ref_T_Routine    := NULL;
  end record;
```

Die Objekte sind als *private* definiert. Um die Werte ins Objekt speichern bzw. auslesen zu können, müssen die entsprechenden Set-Get Funktionalitäten für jede Metrik existieren.

```
  function Get_Metrik_1
    (pRoutine : Ref_T_Routine)
      return Integer;

  procedure Set_Metrik_1
    (pRoutine : Ref_T_Routine;
     pMtr     : Integer);
```

Die nächste Änderung in Paket *Analysis* soll in der Funktion *Get_ Values* erfolgen. Diese Funktion gibt es jeweils für die beiden Objekte *T_ Project* und *T_ Routine*. Die Funktion bereitet in der Datenbank für die entsprechende Tabelle eine Ausgabe für eine Insert-Anweisung vor. Damit wird die Insert-Anweisung in relevant_stats einfacher ermöglicht. Die Reihenfolge der Messwerte in dieser Ausgabe muss genauso sein, wie die Reihenfolge von Spalten der entsprechenden Tabelle (in unserem Beispiel in *MyAnalysis_ Table_ Rows*), die in globals.ads definiert sind. Die entsprechende Messwerte von Objekten werden durch die entsprechenden GET-Funktionen erhalten.

```
  -- Get_Values for T_Project --
  function Get_Values(pProject: Ref_T_Project)
    return String is
  begin
```

```
  return
    Get_Id(pProject)'img & "," &
    "'" & Str(Get_Name(pProject))        & "'" & "," &
    "'" & Str(Get_Date(pProject))        & "'" & "," &
    "'" & Str(Get_Comment(pProject))     & "'" & "," &
    "'" & Str(Get_SSA_Runtime(pProject)) & "'" & "," &
    ","& Str(Get_MyAnalysis_Runtime(pProject)) & ",";
end Get_Values;

-- Get_Values for T_Routine --
function Get_Values(pRoutine : Ref_T_Routine;
                    pProject_Id: Natural)
  return String is
begin
  return
    Get_Id(pRoutine)'img              & "," &
    pProject_Id'img                   & "," &
    "'" & Str(Get_Name(pRoutine))     & "'" & "," &
    Get_NO_Locator(pRoutine)'img      & "," &
    Get_NO_PhiFunc(pRoutine)'img      & "," &
    Get_NO_OEdges(pRoutine)'img       & "," &
    Get_NO_Assgn(pRoutine)'img        & "," &
    Get_NO_LHS(pRoutine)'img          & "," &
    Get_NO_RHS(pRoutine)'img          & "," &
    Get_NO_Blocks(pRoutine)'img       & "," &
    Get_DO_DT(pRoutine)'img           & "," &
    Get_NO_Link_In_Defintions
      (pRoutine)'img                  & "," &
    Get_NO_Link_Out_Uses
      (pRoutine)'img                  & "," &
    Get_PatternNumber(pRoutine)'img   & "," &
    Get_Metrik_1(pRoutine)'img & ","&
    Get_Metrik_2(pRoutine)'img & ","&
    Get_Metrik_3(pRoutine)'img;
end Get_Values;
```

Die letzte Änderung im Paket Analysis bezieht sich auf die Funktion *Set_Attribute*. Diese Funktion existiert jeweils für die Objekte *T_Project* und *T_Routine*. Wenn ein Attribut aus CSV-File gelesen und in die Datenbank gespeichert werden soll, wird diese Funktion jeweils für jedes Attribut aufgerufen, das aus der CSV-Datei gelesen wird. Die Reihenfolge der Spalten, die in Project_Table_Rows und Routine_Table_Rows gespeichert sind, spielen eine wichtige Rolle, weil in der Funktion *Set_Attribute* die Reihenfolge vorgenommen worden sind. D.h. Es wird beispielsweise angenommen, dass die ersten Spalte *Id* ist. Sobald die Funktion feststellt, dass das Attribut, das aktualisiert

werden soll, der erste Spalte entspricht, dann wird die Spalte *Id* im Objekt aktualisiert.

```
if pAttr = Globals.Project_Table_Rows(1) then
   Set_Id(pProject, Integer'Value(Str(pValue)));
...
```

In unserem Beispiel soll eine Anpassung zuerst für die Metrik *MyAnalysis_ Runtime* in der Funktion Set_Attribute durchgeführt werden, die für das Objekt *T_ Project* zuständig ist. Weitere Anpassungen sollen für die Metriken *Metrik_ 1*, *Metrik_ 2*, *Metrik_ 3* in der Funktion Set_Attribute erfolgen, die für das Objekt *T_ Routine* zuständig ist.

```
procedure Set_Attribute
  (pProject : Ref_T_Project;
   pAttr    : Unbounded_String;
   pValue   : Unbounded_String) is
begin
  if (pProject /= No_T_Project) then
     if To_Upper(Str(pAttr)) = To_Upper(Str(Globals.Project_Table_Rows
         (1))) then --
        Set_Id(pProject, Integer'Value(Str(pValue)));
     ...
     elsif To_Upper(Str(pAttr)) =
           To_Upper(Str(Globals.Project_Table_Rows(6))) then
        Set_MyAnalysis_Runtime(pProject, pValue);
     end if;
  end if;
end Set_Attribute;
```

4.3 Änderungen in *Db_ Operations*

Das Paket *Db_ Operations* enthält alle Funktionalitäten des Tools relevant_stats. Diese Funktionalitäten sind:

- *Get_ PrjName* gibt zu einem gegebenen ID den Name des Projeks.
- *Check* überprüft die Datenbankdatei, ob sie Konsistent ist oder nicht.
- *Create* erzeugt eine neue Datenbankdatei mit entsprechenden Datenbanktabellen.
- *List* listet alle Projekte von einer Datenbankdatei auf.
- *Insert* fügt neue Messwerte in die Datenbankdate ein.

- *Update* aktualisiert die Messwerte.
- *Delete* löscht ein Projekt aus der Datenbank.
- *Plot* stellt eine graphische Darstellung zu einer gegebenen SQL-Anweisung dar.
- *ToCSV* speichert die Datenbanktabellen in die entsprechende CSV-Dateien.
- *ShowTables* zeigt Informationen von den Tabellen an, die in der Datenbank gespeichert worden sind.

Neben den globalen Funktionen gibt es die lokal benutzten Funktionen, die im Rahmen der Diplomarbeit als Hilfsfunktionen bezeichnet werden. Die sind:

- *Get_ SelectStmt* gibt zu einem gegebenen Tabellennamen den Text für die Select-Anweisung an.
- *Get_ Select_ Rows* gibt die Spaltennamen zu einem gegebenen Tabellennamen als Text an. Diese Funktion wird in der Funktion *Get_ SelectStmt* aufgerufen, um zu bestimmen, aus welchen Spalten die Selektion ausgeführt werden soll. [1]
- *Get_ CreateStmt* gibt zu einem gegebenen Tabellenname den Text für die Create-Anweisung an.
- *Get_ Create_ Rows* gibt die Spaltennamen zu einem gegebenen Tabellennamen als Text an, wie sie definiert worden sind. Diese Funktion wurde Get_CreateStmt aufgerufen, um eine CREATE-Anweisung erzeugen zu können. Die Funktion gibt die Spalten zum Beispiel in der folgenden Form **Id INT, Name VARCHAR(100), ...**
- *Get_ ViewStmt* gibt den Text zu einem gegebenen Tabellenname für die Erstellung einer Sicht(View) aus der Routine_Table an. Diese Funktion nutzt auch die Funktion *Get_ Select_ Rows*.
- *Get_ InsertStmt* gibt zu einem gegebenen Tabellenname den Text für die Select-Anweisung an. Die Spalten, welche selektiert werden sollen, müssen auch übergeben werden.

[1] Selektion wird normalerweise mit allen Spalten der angegebenen Tabelle durchgeführt. An dieser Stelle könnte dieses Zeichen **'*'** verwendet werden (select * from ...). Es wurde aber verzichtet. Die Spalten einer Tabelle sollten natürlich in globals.ads definiert sein (in unserem Beispiel *MyAnalse_ Rows*)

- *Get_ UpdateStmt* gibt zu einem gegebenen Tabellenname den Text für die Update-Anweisung an. Zusätzlich müssen die Informationen übergeben werden, welche Spalten es gibt und wie sie aktualisiert werden sollen.
- *Get_ DeleteStmt* gibt zu einem gegebenen Tabellenname den Text für die Delete-Anweisung an. Die Bedingung, die das Löschen erfolgen soll, muss auch angegeben werden.
- *Get_ DropStmt* gibt zu einem gegebenen Tabellenname den Text für die Drop-Anweisung an.
- *AddToCSV* speichert die Werte einer Tabelle, die durch eine SQL-Anweisung erzeugt wurde, in die übergebene Datei.
- *Exist_ Project* überprüft, ob die ein Projekt mit der übergebenen ID-Nummer in der übergebenen Datenbankdatei existiert.
- *Get_ LastID* gibt die letzte ID-Nummer einer gegebenen Datenbanktabelle in der gegebenen Datenbankdatei an.
- *Check_ STMT* überprüft, ob eine Anweisung die Spalten *Prj_ Id* und *Name* enthält. Wenn nicht, wird die Anweisung durch diese Spalten vervollständigt. Dies ist vorausgesetzt, wenn es um die graphische Darstellung geht.
- *Is_ A_ MultiColum* stellt fest, ob die durch SQL-Anweisung erstellte Tabelle mehrere Projekte enthält.
- *InsertIntoDB* speichert in der Datenbank die Werte einer Datenbanktabelle.
- *AddValues_ FromCSV* fügt die Werte, die in der gegebenen CSV-Datei stehen, in das gegebene Objekt *T_ Project* hinzu.
- *Save_ Statistics* speichert in die gegebene Datenbankdatei die Werte aus dem gegebenen Objekt *T_ Project*.
- *Update_ ProjectTable* aktualisiert in der Project_Table die im Objekt *T_ Project* stehenden Werte .
- *Update_ RoutineTable* aktualisiert in der Routine_Table die im Objekt *T_ Project* stehenden Werte.

Wenn eine neue Analyse in relevant_stats hinzugefügt werden soll, müssen zunächst die Hilfsfunktionen *Get_ Select_ Rows*, *Get_ Create_ Rows*, *Update_ ProjectTable* und *Update_ RoutineTable* angepasst werden. Die Anpassung für die Funktion *Get_ Select_ Rows* ermöglicht, dass die Spalten der neuen Tabelle *MyAnalysis_ Table* für eine Select-Anweisung vorbereitet werden, wenn diese Funktion mit dem neuen Tabellenname also *MyAnalysis_ Table* aufgerufen wird. Diese Funktion erhält den Namen dieser Spalten, die in der neuen Tabelle von *MyAnalysis_ Rows* erstellt wurde.

```
if pTableName = Globals.Project_Table then
...
elsif pTableName = Globals.SSA_Table then
   sLocal := Globals.SSA_Table_Rows(1);
   nLast  := Globals.SSA_Table_Rows'Last;
elsif pTableName = Globals.MyAnalysis_Table then
   sLocal := Globals.MyAnalsis_Rows(1);
   nLast := Globals.MyAnalsis_Rows'Last;
end if;

for i in 2 .. nLast loop
  if pTableName = Globals.Project_Table then
  ...
  elsif pTableName = Globals.SSA_Table then
     sLocal := Globals.SSA_Table_Rows(i);
  elsif pTableName = Globals.MyAnalysis_Table then
     sLocal := Globals.MyAnalysis_Rows(i);
  end if;
  ...
end loop;
```

Die zweite ähnliche Änderung bezieht sich auf die Erzeugung einer neuen Tabelle (Create) in der Datenbank. Diese Änderung soll in der Funktion *Get_ Create_ Rows* erfolgen. Mit Hilfe dieser Funktion wird eine Create-Anweisung vorbereitet. Daher müssen die Spalten mit ihren Typen durch den Separator Koma (ID INT, PRJ_ID INT, Name VARCHAR ...) angegeben werden.

```
if pTableName = Globals.Project_Table then
...
elsif pTableName = Globals.SSA_Table then
   sRow   := Globals.SSA_Table_Rows(1);
   sType  := Globals.SSA_Table_Types(1);
   nLast  := Globals.SSA_Table_Rows'Last;
elsif pTableName = Globals.MyAnalsis_Table then
   sRow := Globals.MyAnalsis_Table_Rows(1);
   sType := Globals.MyAnalsis_Table_Types(1);
```

```
      nLast := Globals.MyAnalsis_Table_Rows'Last;
   end if;
   ...
   for i in 2 .. nLast loop
     if pTableName = Globals.Project_Table then
     ...
     elsif pTableName = Globals.SSA_Table then
        sRow   := Globals.SSA_Table_Rows(i);
        sType  := Globals.SSA_Table_Types(i);
     elsif pTableName = Globals.MyAnalsis_Table then
        sRow := Globals.MyAnalsis_Table_Rows(i);
        sType := Globals.MyAnalsis_Table_Types(i);
     end if;
     ...
   end loop;
```

Die weiteren Änderungen sollen für die Funktionen *Update_ ProjectTable* und *Update_ RoutineTable* erfolgen. Dies sind die letzten Hilfsfunktionen, die angepasst werden sollen. Solche Funktionen sind dazu da, wenn es um eine Aktualisierung durch eine CSV- oder IML-File geht. Die Messwerte, die aktualisiert werden sollen, werden erst aus der entsprechenden Datei gelesen und in das Objekt *T_ Project* bzw. *T_ Routine* gespeichert. Diese Funktionen aktualisieren in der Datenbank diese Messwerte, falls sie im Objekt einen Wert haben. [2] Für jede Metrik, die dazu gekommen ist, muss diese Aktualisierungsphase im Code angepasst werden. In unserem Beispiel soll für *My-Analyse_ Runtime* in *Update_ ProjectTable* für *Metrik_ 1*, *Metrik_ 2* und *Metrik_ 3* in *Update_ RoutineTable* Anpassungen durchgeführt werden. Das folgende Beispielmuster zeigt, wie diese Aktualisierung erfolgen soll.

```
-- update Metrik_1
if Analysis.Get_Metrik_1(pRoutine) /= -1 then
   sKeyValues := Str("Metrik_1=") & Analysis.Get_Metrik_1(pRoutine)'IMG;
   sLocal := Get_UpdateStmt
              (Globals.Routine_Table, -- UPDATE from Routine_Table
               sKeyValues,            -- SET Metrik_1=Value
               Str("Id=") & pID'img); -- WHERE Id= pID
   SQLite.Get_Table (DB, Str(sLocal), Result, Status, Error);
   Bauhaus.Debug.Verbose("_SQL_Stmt : " & Str(sLocal));
   if Status /= SQLite.Sqlite_OK then
      Bauhaus.Debug.Verbose
        ("update NO_Locator failed.");
```

[2] *Einen Wert haben* bedeutet, dass sie nicht den Default-Wert enthalten. Default-Wert ist für Integer -1 und für String (Unbounded_String) NULL definiert und jede Metrik ist mit diesen Werten vorinitialisiert.

```
      sLocal := Str(Interfaces.C.Strings.Value(Error));
      Bauhaus.Debug.Verbose("_SQL_Error:" & Str(sLocal));
   end if;
end if;
```

Somit ist bei den Hilfsfunktionen keine Änderung mehr nötig. Wenn in der Datenbank die Messwerte vollständig vorliegen, ist bei den globalen Funktionen wie *List*, *Update*, *ToCSV* und *Plot* auch nichts zu ändern.
Die Tabelle *MyAnalysis_ Table*, die zur neuen Analyse *MyAnalysis* gehört, soll durch das Tool relevant_stats als eine Sicht von Routine_Table erzeugt werden können. Dies soll in der Funktion *Create* möglich sein.

```
      -- create MyAnalyse_Table as a view
      if Status = SQLite.Sqlite_OK then
         Bauhaus.Debug.Verbose("_create MyAnalyse_Table");
         sLocal := Get_ViewStmt(Globals.MyAnalyse_Table);
         SQLite.Get_Table (DB, Str(sLocal), Result, Status, Error);
         Bauhaus.Debug.Verbose("_SQL_Stmt:" & Str(sLocal));
      end if;
```

Um danach die Konsistent einer Datenbank zu gewährleisten, muss die Anpassung in der Funktion *Check* erfolgen. Wenn in der Datenbank Project_Table, Routine_Table und die Analysentabellen (in unserem Beispiel MyAnalyse_Table und SSA_Table) existieren, ist somit diese Datenbank Konsistent. Aus diesem Grund ist es notwendig, dass die Existenzprüfung in der Funktion *Check* enthalten ist.

```
      -- check MyAnalyse_Table
      Bauhaus.Debug.Verbose("_check MyAnalyse table");
      sLocal := Get_SelectStmt
                  (Globals.MyAnalyse_Table,
                   Get_Select_Rows(Globals.MyAnalyse_Table));
      SQLite.Get_Table (DB, Str(sLocal), Result, Status, Error);
      Bauhaus.Debug.Verbose("_SQL_Stmt : " & Str(sLocal));

      if Status /= SQLite.Sqlite_OK then
         Bauhaus.Debug.Verbose("_check MyAnalyse_Table failed.");
         sLocal := Str(Interfaces.C.Strings.Value(Error));
         Bauhaus.Debug.Verbose("_SQL_Error:" & Str(sLocal));
         return False;
      end if;
```

Eine der wichtigen Operationen in relevant_stats ist das Einfügen der Messwerte in die Datenbank, die aus einem ProjektVerzeichnis entnommen wurden. Diejenigen, die eine

neue Analyse in relevant_stats einführen möchten, sollen eine Funktion implementieren. Diese Funktion liest die ganzen Messwerte aus IML-Datei und speichert in das Objekt *T_Project*. Mit Hilfe der Funktion *Save_Statistics* werden dann die Messwerte in die Datenbank gespeichert. Daher ist nur ausreichend das Objekt entsprechend (nach oder vor der Aktualisierung der SSA-Analyse) zu aktualisieren.

```
if Is_IML_File then
   Bauhaus.Debug.Verbose("Is_IML_File=TRUE");
   SSA.Get_Statistics(P_Table, Str(pIN_FILE));
   MyPackage.Get_MyAnalyseStatistics(P_Table, Str(pIN_FILE));
   ...
end if;

bReturn := Save_Statistics(pFile        => pDBFile,
                           pProject     => P_Table,
                           pProjectName => pPrj_Name);
```

Nachdem die Möglichkeit besteht, die neuen Messwerte in der Datenbank zu speichern, sollen die neuen Messwerte der Tabelle bzw. Sicht (View) MyAnalysis_Table gelöscht werden können, falls ein Projekt entfernt wird. Dies erfolgt in der Funktion *Delete*. In dieser Funktion muss aber dagegen keine Anpassung bzw. Änderung mehr notig sein, weil die Messwerte nur aus der Routine_Table gelöscht werden dürfen. Da die neue Metriken auch in der Routine_Table gespeichert worden sind, ist das Tool relevant_stats in der Lage, diese Messwerte aus der Datenbank zu entfernen.

Die letzte Änderung bezieht sich auf die Funktion *ShowTables*. Diese Funktion gibt die wichtige Informationen von allen Tabellen an. Somit wird gezeigt, ob es sich um eine Tabelle, eine Sicht oder Informationen von ihren Spalten (ihren Namen und ihre Typen) handelt.

```
-- MyAnalyse_Table
Ada.Text_IO.Put_Line("View          : " & Str(Globals.MyAnalyse_Table));
nLast := Globals.MyAnalyse_Table_Rows'Last;
Ada.Text_IO.Put_Line("Count of Rows : " & nLast'IMG);
Ada.Text_IO.Put_Line("Rows          : ");
for i in 1 .. nLast loop
  Ada.Text_IO.Put("  Name=" & Str(Str_Operations.Get_ShowValue
                              (Globals.MyAnalyse_Table_Rows(I), 20)));
  Ada.Text_IO.Put_Line("Type=" & Str(Globals.MyAnalyse_Table_Types(I)));
end loop;
```

Somit ist keine Änderung mehr nötig. Das bedeutet, dass das Tool relevant_stats in der Lage ist, die Messwerte aus der IML-Datei bzw. CSV-Datei zu lesen und in die Datenbank zu speichern. Wenn diese Daten in der Datenbank existieren, können die Funktionalitäten wie die Auflistung (List), die Aktualisierung aus CSV-Datei (Update), die graphische Darstellung (Plot), die Speicherung der Messwerte in die CSV-Datei gewährleistet werden.

Nach dem Hinzufügen einer neuen Analyse in *relevant_ stats*, tritt das Problem auf, dass die Datenbanken veralten, die vor dem Einfügen existierten. Der Grund dieser Veraltung ist, dass das Tool *relevant_ stats* nach der Hinzufügen der neuen Tabelle in der Datenbank, alle Tabellen sowie *Project_ Table*, *Routine_ Table*, *SSA_ Table* und *MyAnalysis_ Table* auf ihre Existenz überprüft. Da die alte Datenbanken die neue Tabelle *MyAnalysis_ Table* nicht beinhalten, werden sie vom Tool *relevant_ stats* als Inkonsistent bewertet. Darüber hinaus sollte das Tool *relevant_ stats* auch in der Lage sein, die Informationen der veralteten Datenbank beizubehalten, um die Übertragung dieser Informationen in die neue Datenbank zu ermöglichen.

Die Übertragung erfolgt wieder durch das Tool *relevant_ stats*, da dieses Tool in der Lage ist, die Projekttabellen der Datenbanken in CSV-Format umzuwandeln und durch die CSV-Dateien erzeugte Projekttabellen in die Datenbank zu speichern. In diesem Sinne erfolgt die Übertragung für jedes Projekt in der veralteten Datenbank wie folgt. Als erstens werden Projekttabellen durch den Befehl *ToCSV* in CSV-Format umgewandelt.

```
****************************************************
* relevant_stats -db "ToCSV <Prj_ID>" My_Old_DB     *
****************************************************
```

Danach wird durch den Befehl *insert* das durch CSV-Dateien dargestellte Projekt in die neue Datenbank gespeichert.

```
*********************************************************************
* relevant_stats -db "insert                                        *
*  <Prj_Name>_Project_Tablecsv;<Prj_Name>_Routine_Table.csv" My_New_DB *
*********************************************************************
```

5 Bewertung dieser Arbeit

Dieses Kapitel schildert die Sicht des Autors auf die Ergebnisse des Projekts und gibt einen kurzen Ausblick auf die Zukunft von *relevant_stats*.

5.1 Ergebnisse

In diesem Teilkapitel werde ich die einzelnen Phasen der Diplomarbeit näher beschreiben und die daraus resultierenden Ergebnisse im Projektverlauf erläutern.

Der Aufwand für das Entwicklungsprojekt dieser Arbeit war von Anfang an erheblich gut geplant worden. In der Anfangsphase der Arbeit war viel Zeit für eine umfangreiche Einarbeitung in die Sprache Ada95 und in die Analysen, die in Bauhaus implementiert worden sind, notwendig. Zusätzlich war die Einarbeitung in das im Bauhaus implementierten Komponenten und in die Themen SQLite und GNUplot zu bewältigen.

Nach der Einarbeitungsphase habe ich das Projekt in Absprache mit dem Betreuer in vier Schritte aufgeteilt und der Reihenfolge nach bearbeitet. Im ersten Schritt wurde die SSA-Analyse als Beispielanalyse näher betrachtet. Dabei wurden als erstens die Metriken bestimmt, die Einfluss auf diese Analyse haben. Danach erfolgte die Implementierung zur Messung dieser Metriken. Im zweiten Schritt habe ich mir darüber Gedanken gemacht, wie diese Messwerte in der gespeichert werden sollen. Zuerst wurde ein Datenbankmodell erstellt, das den Umgang des Tools relevant_stats mit verschiedenen Analysen erleichtert. In Absprache mit dem Betreuer wurde die Datenbank SQLite ausgewählt, weil SQLite auf allen PSLx-Rechner zu Verfügung steht. Danach habe für SQLite eine Interface in Ada implementiert, um die notwendigen Datenbankoperationen (wie z.b. OpenDb, CloseDB etc.) ausführen zu können. Zuletzt wurde die Funktionalität der Datenbankoperationen (z.b. Select, Insert, Update, Delete) für das Tool relevant_stats mit

Hilfe von Interface implementiert. Im dritten Schritt habe ich für die Erstellung graphischer Darstellung GNUplot ausgewählt und eine Recherche über die Diagrammerstellung in GNUplot geführt. Im Anschluss wurde eine Funktion (Plot_Histogramm) zur Erstellung von Diagramme in C mit Hilfe von GNUplot-Interface implementiert, um diese Funktionalität in Ada benutzen zu können. Daher wurde ein Binding in Ada für diese Funktion erstellt. Als Letztes wurde die Operationen zur graphischen Darstellung mit zusätzlichen Funktionalitäten (z.b spaltenorientierte Ansicht, zeilenorientierte Ansicht) für relevant_stats implementiert. Im letzten Schritt wurde alle bisherigen Komponenten in relevant_stats zusammengeführt, damit das Tool in der Lage ist, die Funktionalitäten der jeweiligen Komponenten auszuführen.

Ergebnisse dieser Diplomarbeit sind:

- Die Implementierung der Schnittstelle für SQLite in Ada.
- Das Ada-Binding für die Interface zu GNUPlot.
- Die Implementierung der Funktionalität zur Erzeugung von Histogramme mit Hilfe von der Schnittstelle von GNUplot.
- Werkzeug zur Messung von Analysenmetriken.
- Werkzeug zur Administration der Messwerte.

Im Zwischenvortrag nach der Hälfte der Bearbeitungszeit wurden erste detaillierte Ideen für den Tool-Entwurf mit einer Beispielanalyse (SSA Analyse) vorgestellt. Besonders werden die Metriken der SSA Analyse im Zwischenvortrag diskutiert, wie sie im Rahmen des Tool angewandt werden können.

In dem verbleibenden Bearbeitungszeitraum sind die Verbesserungsvorschläge und einige Erweiterungen für das Tool relevant_stats in Absprache mit dem Betreuer realisiert worden. Daher wurden neue Eigenschaften wie z.B. toCSV (die Messwerte in CSV-File speichern), das neue Datenbankmodell für das Tool relevant_stats implementiert. Währenddessen wurde es auf die Erweiterbarkeit des Tools geachtet. Zuletzt wurde das Tool relevant_stats als Schnittstelle implementiert, das bedeutet, dass die Metriken statisch während des Ablaufs der SSA Analyse gemessen werden können.

Der Autor bewertet die erzielten Ergebnisse positiv. Neben einem erheblichen persönlichen Gewinn, wurde das Ziel der Arbeit stärker erreicht. Nach Überzeugung des Autors

sind die Kernprobleme bzw. die Anforderungen und auch die Erweiterungen der Aufgabe gelöst.

5.2 Ausblick

In diesem Kapitel werden die Erweiterungen für relevant_stats erläutert.

Eine Erweiterung wäre, die Bestimmung der Korrelation zwischen den Messwerten. In diesem Projekt gibt das Tool relevant_stats die Möglichkeit, die mögliche Messwerten schon vor der SSA-Analyse zu bestimmen. Falls die Korrelation bekannt ist, kann man mit Hilfe von Messwerten eine Voraussage treffen, wie lange die SSA-Analyse ca. dauern wird.

Im Rahmen der Diplomarbeit wurde nur die SSA-Analyse näher untersucht und somit in das Tool relevant_stats hinzugefügt. Man kann jedoch die Vorgehensweise dieser Diplomarbeit auch auf anwenden Analyse anwenden.

Literaturverzeichnis

[AC72] F. E. Allen and J. Cocke. *Graph-theoretic constructs for program flow analysis. Technical Report RC3923 (17789).* IBM Thomas J. Watson Research Center, July 1972.

[Bar06] John Barnes. *Programming in Ada 2005.* Addison Wesley, 2006.

[BGL06] Joshua Bloch Joseph Bowbeer David Holmes Brian Goetz, Tim Peierls and Doug Lea. *Java Concurrency in Practice.* Addison Wesley, 2006.

[Bis08] Marc Bischof. *Interprozedurale SSA-Form Das Konzept der statischen Einmalzuweisung.* Hauptseminar Programmanalysen Universität Stuttgart, 19. April 2008.

[But97] David R. Butenhof. *Programming with POSIX(R) Threads.* Addison Wesley Reading, Mass., 1st edition,, 1997.

[GM] Michael Unterstein Günter Matthiessen. *Relationale Datenbanken und SQL Konzepte der Entwicklung und Anwendung.* Addison-Wesley, ISBN 3-8273-2085-2.

[Her09] Bernhard Hering. *Kontrollflussanalyse 2.* LMU München Institut für Informatik, http://www2.tcs.ifi.lmu.de/lehre/SS09/ProSem/Ausarbeitungen/, Juni 2009. Haupt-/Bachelorseminar: Programmanalyse.

[Homa] Homepage. *Gnuplot.* http://www.gnuplot.info/.

[Homb] Homepage. *Grundkurs Gnuplot.* http://userpage.fu-berlin.de/ voelker/gnuplotkurs/.

[Homc] Homepage. *SQLite.* http://www.sqlite.org.

[HU75] M. S. Hecht and J. D. Ullman. *A simple algorithm for global data flow analysis problems.* SIAM J. Comput., 4(4):519-532, Dec. 1975.

[KAR03] Sebastian KARACA, Tahir; SETZER. *Erweiterung und Generierung der Zwischendarstellung IML für C++ Programme.* Universität Stuttgart, Institut für Softwaretechnologie, Dimplomarbeit Nr. 2048, März 2003.

[KDCK] Timothy J. Harvey Keith D. Cooper and Ken Kennedy. *A Simple, Fast Dominance Algorithm.* Rice University Houston, Texas, USA.

[KEU05] Steffen KEUL. *Generierung der Zwischendarstellung IML für Ada95 Programme.* Universität Stuttgart, Institut für Softwaretechnologie, Dimplomarbeit Nr. 2323, 2005.

[KNA02] Markus KNAUSS. *Erweiterung und Generierung der Zwischendarstellung IML für Java-Programme.* Universität Stuttgart, Institut für Softwaretechnologie, Diplomarbeit Nr. 2006, Oktober 2002.

[LM69] E. Lowry and C. Medlock. *Object code optimization.* Communications of the ACM, pages 13-22, Jan. 1969.

[NM92] Robert H. B. Netzer and Barton P. Miller. *What are race conditions? some issues and formalizations.* LOPLAS, 1992.

[PB99] T. J. Harvey und L. T. Simpson P. Briggs, K. D. Cooper. *Practical improvements to the construction and destruction of static single assignment form.* Softw. Pract. Exper., 28(8):859-881, 1999.

[Pro] Bauhaus Projekt. http://www.bauhaus-stuttgart.de/bauhaus/.

[Pro59] R. Prosser. *Applications of boolean matrices to the analysis of flow diagrams In Proceedings of the Eastern Joint Computer Conference, pages 133-138.* Spartan Books, NY, USA, Dec. 1959.

[RCDU91] Barry K. Rosen Ron Cytron, Jeanne Ferrante, Mark N. Wegman IBM Research Division, and F. Kenneth Zadeck Brown University. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.* 4, October 1991.

[rnEP09] Prof. Dr. rer. nat. Erhard Plödereder. *Vorlesung Programmanalysen und Compilerbau.* Abteilung Programmiersprachen und Übersetzerbau, http://www.informatik.uni-stuttgart.de/iste/ps/, WS 2008/2009.

[ROH98] Jürgen ROHRBACH. *Erweiterung und Generierung einer Zwischendarstellung für C-Programme.* Universität Stuttgart, Institut für Softwaretechnologie, Studienarbeit Nr. 1662, 1998.

[Sch10] Holger Schwarz. *Grundlagen von Datenbanken und Informationssystemen.* Anwendersoftware (AS) Univesität Stutgart, http://www.ipvs.uni-stuttgart.de/abteilungen/as/, Wintersemester 2009/10.

[Sin89] Mukesh Singhal. *Deadlock detection in distributed systems.* COMPUTER, 1989.

[SM99] Reiner Singer Stefan Middendorf. *Java Programmierhandbuch.* dpunkt.verlag, 1999.

[UB] Abteilung Softwaretechnik Universität Bremen. http://www.informatik.uni-bremen.de/st.

[VS95] und Y. Lee V. Sreedhar, G. Gao. *Efficient data flow analysis using DJ graphs: Elimination methods revisited.* 1995.

[WUE96] Martin WUERTHNER. *Entwurf und Implementierung einer Interndarstellung für die Analyse von Ada-Programmen.* Universität Stuttgart, Institut für Softwaretechnologie, Studienarbeit Nr. 1567, 1996.

A relevant_stats

A.1 Kommandozeile

relevant_stats -db <param> [-config_dir <dir>] [-config <key>=<value>] [-verbose] [-trace] [-timing] [-version] [-license] [-usage] [-help] <inputfile>

- -db <param>: Datenbankoperationen von relevant_stats

```
*************************************************************
* -db "check                                              " *
* -db "showtables                                         " *
* -db "list   <ID>              | <SQL-STMT>              " *
* -db "insert <IML-FILE>        | <CSV-FILE>              " *
* -db "update <ID> <TYPE=VALUE> | <IML-FILE> | <CSV-FILE> " *
* -db "delete <ID>                                        " *
* -db "plot   <ORIENT> <SQL-STMT>                         " *
* -db "toCSV  <ID>   | <SQL-STMT> <FileName>              " *
*************************************************************
```

- -config_dir <dir>: Benutzt dir als Konfigurationsverzeichnis.
- -config <key>=<value>: Benutzt den Wert value für den Konfigurationsparamater key.
- -verbose: Gibt Informationen über die durchgeführten Aktionen aus.
- -trace: Gibt detaillierte Informationen über die durchgeführten Aktionen aus.
- -version: Gibt die Version-Information aus.
- -licence: Gibt die Lizenz-Information aus.
- -usage or -help: Gibt Informationen über die Benutzung von relevant_stats aus.

A.2 Abhängigkeiten

Das Tool relevat_stats benutzt die Bibliotheken von SQLite3 und GNUplot, die in der Tabelle erläutert wurden. Während SQLite-Bibliothek ist für die Datenbankopearationen zuständig ist, ist GNUplot für die graphische Darstellung in relevant_stats zuständig.

Bibliothek	Version	Lizenz	Beschreibung
SQLite	3.5.19	Public domain	SQLite ist eine Programmbibliothek, die ein relationales Datenbanksystem enthält.
GNUplot	4.2	Gnuplot Copyright	Gnuplot ist ein skript- bzw. kommandozeilengesteuertes Computerprogramm zur grafischen Darstellung von Messdaten und mathematischen Funktionen (Funktionenplotter)

Erklärung der Urheberschaft

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

_____

(Ozan Kilic)