Universität Dortmund

Fachbereich Informatik

Lehrstuhl für Programmiersysteme und Übersetzerbau

PG 480: Bug-Finder
Enhanced Error Detection in Concurrent Computer Programs

Endbericht

Betreuer: Priv.-Doz. Dr. Stefan Edelkamp
M. Sc. Shahid Jabbar

Teilnehmer: Alexander Goloub
Mark Kellershoff
Yang Liu
Chavdar, Marinov
Dino Midzic
Goran Milouchev
Daniel Rikowski
Damian Sulewski
Eric Tamokoue
Zakaria, Yfrah

eingereicht am: 29. September 2006

Inhaltsverzeichnis

Kapitel 1

Vorwort

Der Trend, gerichtete Suchverfahren der KI zur Fehlerfindung in nebenläufigen Systemen einzusetzen, hat weiter um sich gegriffen und gilt mittlerweile als etabliert. Die Durchschlagskraft dieses Ansatzes liegt sowohl in der grundlegenden Erforschung und Entwicklung von Suchverfahren als auch in der Realisierung von experimentellen und letztendlich auch in der Praxis einsetzbaren Werkzeugen.

Beiden Zielen hat sich die Projektgruppe *Bugfinder* gestellt. Dabei ist eine *Projektgruppe* eine am Fachbereich der Informatik der Universität Dortmund angesiedelte Hauptdiplomsveranstaltung, bei der mehrere Studenten ein Jahr lang an einem Softwareprojekt arbeiten.

1.1 Aufgabe und Teilgruppen

Die Projektaufgabe ist Entwicklung einer integrierten Entwicklungsumgebung zur Validation von nebenläufiger Software. Dabei sollen Fehler in verteilten Systemen durch eine gerichtete Suche gefunden werden und dem Programmierer als Debugging-Unterstützung interaktiv angezeigt werden.

Auf der algorithmischen Seite sollte die Funktionalität durch die Externalisierung des Programmprüfers und (mindestens) einen bestehenden Ansatz zur Abstraktion verwirklicht werden.

Desweiteren stand die Stabilität, Verfügbar und Nutzbarkeit des Werkzeuges an oberster Stelle.

1.2 Abstraktion

Gruppenmitglieder Alexander Goloub, Maik Kellershoff, Tchavdar Marinov, Goran Milouchev

Programmabstraktionen sind zur effektiven Traversierung von Zustandsräumen unerlässlich. In der Programmverifikation versuchen sie den Nachteil der erweiterten Repräsentation zu einem Modellspezifikation aufzuwiegen. Üblicherweise wird das abstrakte System so konstruiert, dass die Verifikation des abstrakten Systems die Verifikation des ursprünglichen Systems nach sich zieht. Im gegenteiligen Fall entstehen *unechte* Fehler, die zwar im abstrakten, jedoch nicht im konkreten Modell existieren.

Ein bekanntes Werkzeug für die “On-the-fly-Abstraktion” und Gegenbeispielanalyse ist BLAST, für Berkley Abstraction Software Toolkit. Jeder Zeuge eines unechten Fehlers führt zu einer verfeinerten Abstraktion, bis letztendlich die Korrektheit nachgewiesen werden kann oder der Fehler echt ist.

Die wichtigsten Möglichkeiten, Programme zu abstrahieren, sind *Program Slicing*, das Code-Fragmente eliminiert, die unabhängig zur Spezifikation zum Beweis der Korrektheit sind, *Datenabstraktionen*, die den

Wertebereich von Variablen einschränken und *Prädikatsabstraktionen*, die Programmbedingungen aus einer Menge von Prädikaten auswählen.

In der Projektgruppe wurden zwei der drei Abstraktionen untersucht und verwirklicht. *Program Slicing* und *Datenabstraktion*. Desweiteren hat die Gruppe die Prädikatabstraktion studiert, aber aufgrund des hohen Implementationsaufwandes nicht realisiert.

1.3 Externalisierung und Maintenance

Gruppenmitglieder Daniel Rikowski, Dino Midzic, Damian Sulewski

Die Externalisierung betrifft den Gebrauch von Sekundärspeicher wie einer Festplatte oder einem USB-Speicher.

Die Gruppe implementiert unterschiedliche Entwürfe zur Kompression, in dem ein kleiner Teil eines Zustandes im RAM gehalten, während der Rest auf der Festplatte gespeichert wird. Ein Cache garantiert wenige I/O Operationen, um die Zustände von der Festplatte zu lesen. Die Gruppe wertete verschiedene Cache-Algorithmen aus und verglich sie auf unterschiedlichen C++ Programmen. Das Ergebnis ist eine Erweiterung von StEAM, die viel größere Modelle zu überprüfen kann als vorher.

Die ursprünglichen Version von StEAM basierte auf dem GCC-Compiler 2.9. Um den neuen Entwicklungen in der Compiler-Technologie anzuwenden, machte die Gruppe StEAM und seine virtuelle Maschine kompatibel zu der GCC Version 3.3. Sie studierte außerdem auch eine Kreuzkompilation auf einer unterschiedlicher Prozessorarchitektur.

StEAM unterstützt Parallelität durch eine eigene *Thread*-Klasse, die abgeleitet in einem C++ Programm verwendet werden kann. Der Nachteil ist die Inkompatibilität, C++ Programme zu überprüfen, die die Standard-*P-Thread*-Implementierung verwenden. Dieses Problem wurde erfolgreich gelöst, indem die *Thread*-Klasse so erweitert wurde, dass sie völlig kompatibel mit dem Standard ist.

1.4 Integration

Gruppenmitglieder Eric Bertrand Tamokoue, Zakaria Yafrah, Liu Yang

Die Teilaufgabe war die Integration von StEAM in eine Entwicklungsumbung mit Editor, Übersetzer, Klasseneditor und Validierer anzubieten, damit auch Praxiserfahrungen in der Fehlerfindung und -korrektur gemacht werden können.

Als Ausgangspunkt zur Implementierung wurde die State-of-the-Art Entwicklungsumgebung Eclipse gewählt, für die im Baukastenprinzip Erweiterungen implementiert wurden. So wird die Adaption des aktuellen Projekts zum Modellprüfer automatisch übernommen.

Die Verifikationsparameter können vom Benutzer in einem Auswahl-Fenster verändert und abgespeichert werden. Gegenbeispiele werden dem Programmierer durch Syntax Highlighting interaktiv zur Verfügung gestellt werden.

Desweiteren können der Fehlerpfad und statistische Information über die Exploration in der Konsole als auch über eine XML-Schnittstelle betrachtet werden.

1.5 Internetpräsenz

Die Entwicklungen in dem Projekt sind unter der Internet-Adresse:

```
http://bugfinder.sourceforge.net
```

weltweit in englischer Sprache verfügbar.

Hier sind neben Download und Installationsbeschreibungen auch weitere Informationsquellen zum aktuellen Stand des Projekts, ein Benuztzerhandbuch und eine technische Referenz zugänglich.

1.6 Ergebnisse

StEAM ist einer der ersten Programmverifikationswerkzeuge für Programme für C++-Programme. Es analysiert das Programm durch Simulation der Maschinensprache.

Dieser Endbericht gibt ein Überblick der erzielten Ergebnisse zur Weiterentwicklung von StEAM. Alle Teilgruppen haben ihre Mindestziele erreicht und gemeinsam den Weg zur erfolgreichen Programmverifikation weiter geebnet.

Programmabstraktionen helfen, den Zustandsraum zu reduzieren. Die erfolgreiche Externalisierung des Modellprüfers ermöglicht, sehr viel größere Explorationsfragestellungen zu lösen. Die Benutzerfreundlichkeit, Stabilität und Verfügbarkeit des integrierten Werkzeuges ermöglicht, auch an eine industrielle Weiterentwicklung zu denken.

Kapitel 2

Motivation für Sofware Modelchecking

Ist es in unserer heutigen Zeit eigentlich noch nötig zu beschreiben, warum Software getestet werden muss?

Schauen wir um uns, sehen wir immer mehr Lebensbereiche, die durch immer mehr Technik beherrscht werden. Während man vor noch nicht zu langer Zeit den Kaffee in einem Gerät gemahlen, dann heißes Wasser zubereitet, und beides in einer Tasse vermischt hat, um an seine tägliche Dosis Koffein zu kommen, drückt man heute einen Knopf. Nur die Tasse sollte man darunter stellen, der Rest geht automatisch.

Nun ist es bei einer Kaffeemaschine nicht gerade lebensgefährlich, wenn sich dort ein Programmfehler einschleicht, aber was ist, wenn das AntiBlockierSystem in meinem Auto meint, es dürfte jetzt gar nicht mehr bremsen? Es sind nicht nur die sicherheitsrelevanten Einsatzgebiete, in denen man auf Fehlerfreiheit achten sollte. Dadurch, dass heutzutage versucht wird, "unsichtbare" Computer zu bauen, fallen diese erst auf, wenn sie nicht mehr funktionieren. Immer mehr stellt sich heraus, dass die Firmen, die in der Lage sind, verlässliche kleine Helferlein zu bauen, am Markt die Nase vorn haben werden. Informiert man sich etwa nicht, ob das Handy das man kaufen möchte, größere Softwarefehler hat?

Leider war und ist das Testen von Software eine sehr aufwendige Angelegenheit. Um ein Programm zu testen, mussten alle Möglichkeiten der Interaktion ausprobiert werden. Dieses dauerte oft sehr lange und war somit mit hohen Kosten verbunden. Meistens fand man die Fehler erst nach der Implementationsphase heraus und es wurde nötig, größere Teile der Software neu zu implementieren. Besonders bei sicherheitsrelevanter Software, z.B. Software zur Beförderung von Personen, oder Software im Bankwesen, ist es wichtig eine hohe Fehlerfreiheit zu gewährleisten. Hier wird bis heute eine so genannte Metasprache eingesetzt. Dabei wird das Modell in diese Sprache übersetzt, auch heute noch zum größten Teil von Menschenhand, diese Übersetzung wird in einen automatischen Prüfer eingespeist. Wird in der Übersetzung ein Fehler gefunden, so heißt das noch lange nicht, dass auch das Original-Modell einen Fehler enthält. Vielleicht ist dieser ja bei der Übersetzung entstanden. Dieses Problem existiert aber auch in der umgekehrten Richtung. Enthält die Übersetzung keinen Fehler, wissen wir nur mit großer Wahrscheinlichkeit, dass das Modell fehlerfrei ist. Es bleibt immer noch das Restrisiko, dass der Fehler bei der Übersetzung entfernt wurde.

Beim Modell Checking geht man einen anderen Weg. Hier versucht man, direkt das Modell zu überprüfen. Es entfällt die Fehlerquelle "Übersetzung", aber es kommen weitere Schwierigkeiten auf. Die Software muss "laufen" während sie geprüft wird, d.h. wir müssen den Ablauf des Programms zumindest simulieren. Weiterhin ist es auch notwendig - dies sollte klar sein - die Fehler zu finden, d.h. das Überprüfende Programm muss die Fehler kennen, nach denen es sucht. Die Vorteile, die Modell Checking mit sich bringt, überwiegen jedoch alle Nachteile. Prinzipiell kann jeder Programmierer einen sogenannten Modell Checker verwenden, es ist nicht mehr notwendig, eine zusätzliche Metasprache zu lernen. Man gibt einfach seinen gerade erstellten Quellcode ein und kriegt entweder die Meldung, es sei fehlerfrei, oder den Fehler, der gefunden wurde.

Fehler können schon früh während der Implementierung gefunden werden. Da es nicht so aufwändig ist wie eine Übersetzung, kann der Modell Checker öfter eingesetzt werden, z.B. in kurzen regelmäßigen Abständen, so dass Fehler bei der Implementation früh erkannt und beseitigt werden können, bevor größere Maßnahmen notwendig werden.

Kapitel 3

Einführung in StEAM

3.1 StEAM

3.1.1 Einleitung

„STate Exploring Assembly Model Checker“, kurz StEAM genannt, ist ein experimenteller „assembly level model checker“ für nebenläufige C++-Programme. Ähnlich wie vergleichbare Programme, z.B. Java Path Finder [VHBP00a], basiert auch StEAM auf einer virtuellen Maschine[VHBP00b], geht jedoch über die üblicherweise vorhandene Funktionalität hinaus.

In diesem ersten Teil wird deshalb zunächst die zugrunde liegende „Internet C Virtual Machine“ (ICVM) [1]beschrieben, sowie die durch StEAM eingeführten Erweiterungen.

3.1.2 Internet C Virtual Machine

Ziel der ICVM war es, plattformunabhängige Programme zu entwickeln, ohne jedoch die Programmiersprache von C++ nach z.B. C# oder Java zu wechseln. Bei der Entwicklung wurde Wert darauf gelegt, auch Spiele ausführen zu können, so dass Ausführungsgeschwindigkeit ein entscheidender Faktor war.

Die virtuelle Maschine simuliert eine 32-Bit-CISC-CPU mit ca. 64.000 Anweisungen. Die aktuelle Version ist bereits in der Lage, komplexe Programme in akzeptabler Geschwindigkeit auszuführen, z.B. auch das kommerzielle Spiel „Doom“ [2].

ICVM benutzt eine modifizierte Version des GNU C-Compilers (gcc), um die Objektdateien aus dem C++-Quellcode zu erzeugen. Der Compilat dieses Compilers sind sogenannte ELF Binaries. Diese ausführbare Datei hat mehrere Segmente, wobei für uns für die Zustandsaufnahme die DATA und BSS Segmente wichtig sein werden, weil sie die globalen Variablen repräsentieren.

Zustand der ICVM

Zu einem bestimmten Zeitpunkt ist der Zustand der ICVM *implizit* gegeben durch den Zustand des gerade ausgeführten Programmes. Dabei werden die Objektdateien mit Programm-Code, initialisierten (DATA-Segment) und nicht initialisierten Variablen (BSS-Segment) sowie die Maschine (CPU) mit sämtlichen Registern und der Stack modelliert. Zusätzlich werden die im Programmverlauf zugewiesenen Werte an die nicht initialisierten Variablen sowie der dynamisch erzeugte Speicher des ausgeführen Programmes verwaltet.

[1] http://icvm.sourceforge.net
[2] www.doomworld.com/classicdoom/ports

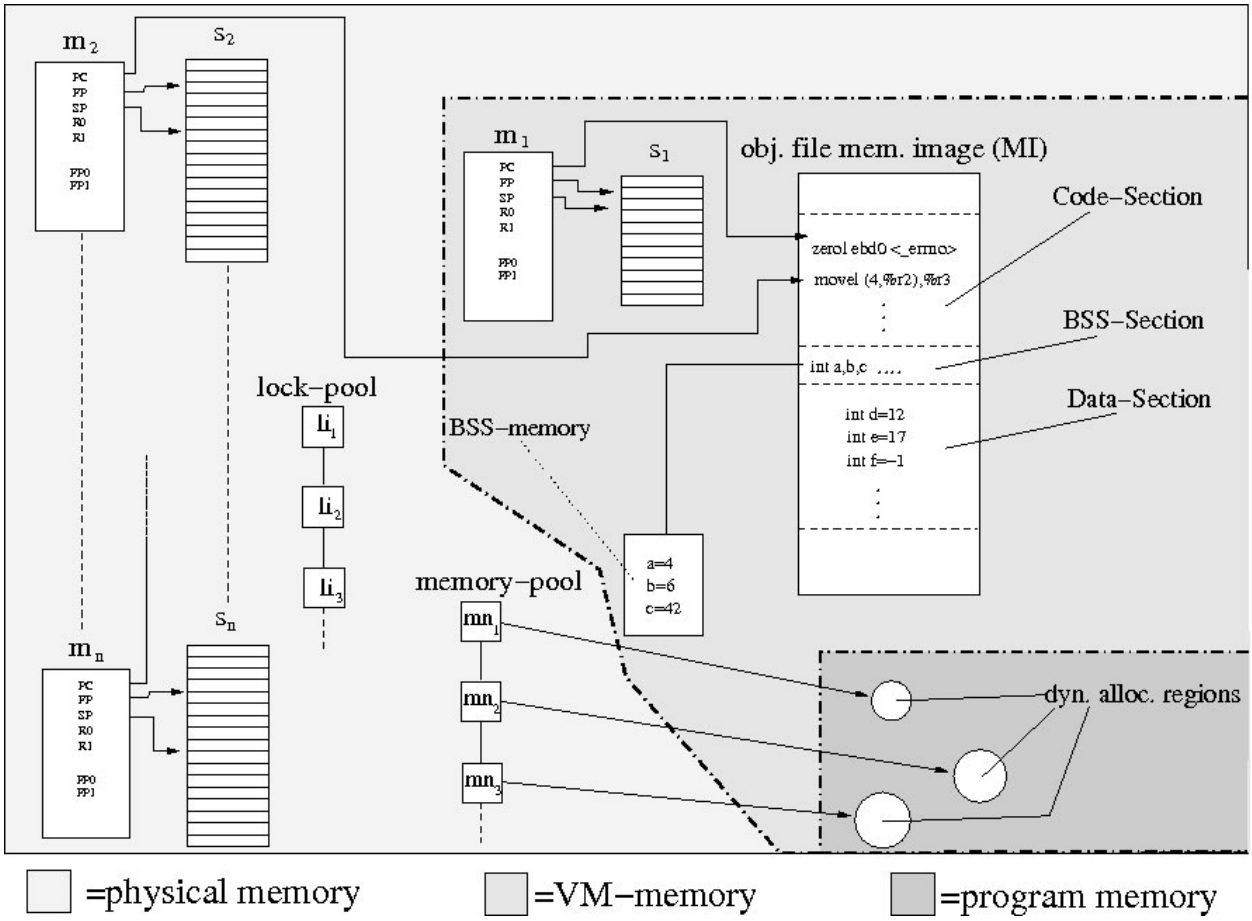

Abbildung 3.1: StEAM-erweiterte ICVM

3.1.3 StEAM-Erweiterungen

Aufbauend auf den gerade beschriebenen Eigenschaften erweitert StEAM die ICVM um viele weitere Funktionen. Am wichtigsten sind Multi-Threading und ein Befehlsatz zur Beschreibung und Durchführung der Fehlersuche.

Aufbau der erweiterten ICVM

Die erweiterte ICVM organisiert den Speicher in einen physischen, virtuellen und Programm-Speicher.

Der virtuelle Speicher beinhaltet den Hauptprozess (main thread), mit zugehöriger Machine m1 und Stack s1. Dazu hat er noch die Objekt-Code Datei, mit zugehörigen Code, BSS und DATA Teilen. Da die globale Variablen in BSS beim Programmstart nicht initialisiert wurden, braucht man die sog. BSS-Speicher, um dessen Werte zu speichern. Ein Teil des virtuellen Speichers ist auch der Programm-Speicher, dessen Teile das Programm selber allozieren kann.

Um das Multi-Threading zu realisieren, erweitert der StEAM ICVM um n zusätzliche Stacks und Machines, die im sog. physischen Speicher residieren. Jedem Prozess gehört eine Maschine und ein Stack, wobei der Programmzähler der Maschine auf die tatsächliche Object-Code Datei zeigt. Dazu fügt der StEAM noch einen Lock- und Speicher-Pool ein. Der Lock-Pool beinhaltet Informationen über gelockte Ressourcen der zugehörigen Prozesse, um die Deadlocks zu vermeiden. Der Speicher-Pool dagegen beinhaltet einen Zeiger auf die zugehörigen Teile des Programm-Speichers.

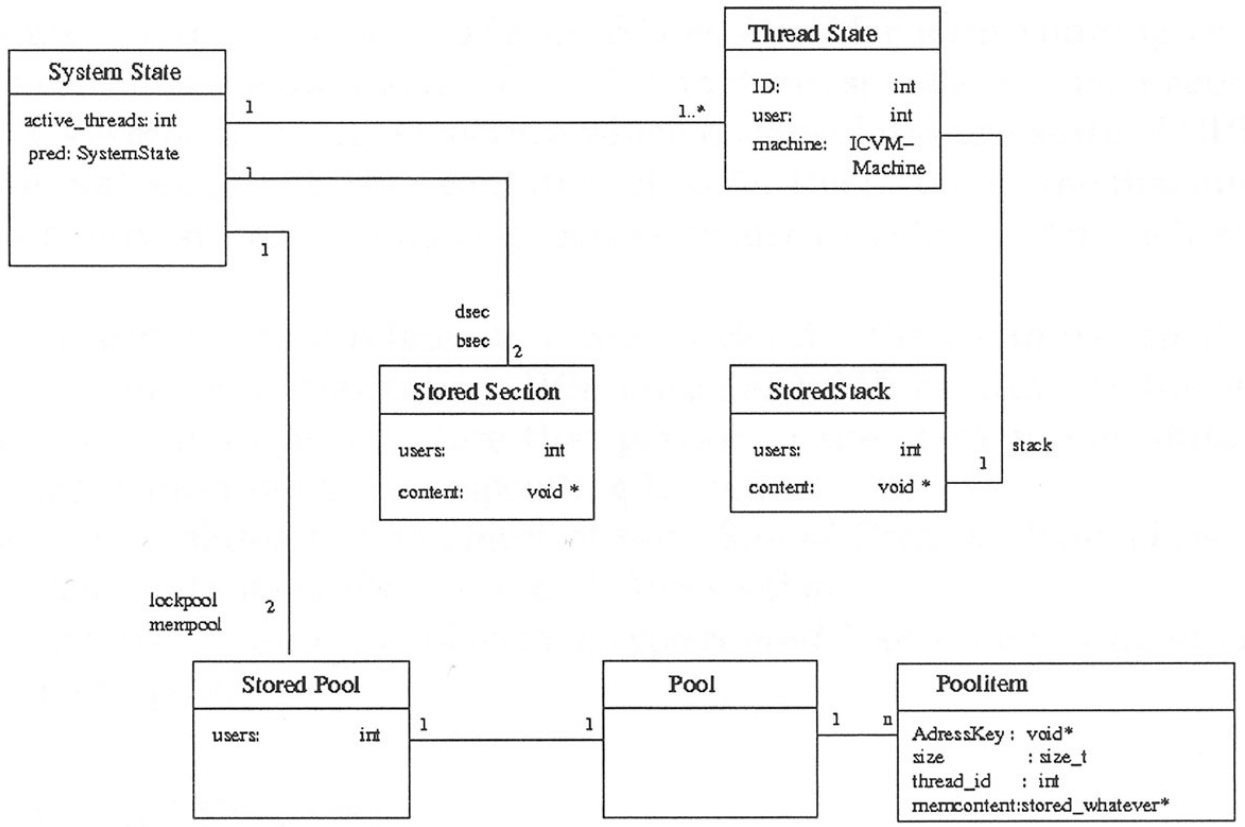

Abbildung 3.2: Klassendiagramm Systemzustand

Zustand von StEAM

Eines der ersten Ziele von StEAM war es, seinen Zustand in einer expliziten Datenstruktur abzubilden. Abbildung 3.2 zeigt das Klassendiagramm einer Zustandsbeschreibung von StEAM. Während der Zustandserweiterung werden für unveränderte Zustände jedoch nicht alle Komponenten kopiert, sondern lediglich ein Zeiger auf die vorhergehende (unveränderte) Komponente. Um Probleme beim Löschen zu verhindern, wird ein Referenzzähler mitgeführt (user).

Multi-Threading

Um Multi-Threading realisieren zu können, benutzt man in StEAM die Klasse `ICVMThread`. Alle Prozesse erben von dieser Klasse. Jede geerbte Klasse muss die Methoden `start()`, `run()` und `die()` implementieren. Ein Beispiel ist auf Seite 12 mit „StEAM-Thread“ gegeben.

Listing 3.1: StEAM-Thread

```
class IVMThread{
  MyThread::MyThread():IVMThread::IVMThread(){
  }

  void MYThread::start(){
    run();
    die();
  }
```

```
  void MYThread::run(){
    ...
  }

  void MyThread::die(){
  }
}

class MyThread{
  void main(){
    a=new MyThread();
    a->start();
  }
}
```

Befehlsmuster

Befehlsmuster sind die grundlegende Technik, die Funktionalität von ICVM zu erweitern und den Programm-Ablauf zu kontrollieren. Vom Prinzip her sind Befehlsmuster Maschinenbefehle, die den Model Checker mit Informationen über das Programm und seine Eigenschaften versorgen.

Ein Befehlsmuster ist im Prinzip eine „sinnlose“ Folge von Anweisungen, die von StEAM als Befehl erkannt werden. Das Beispiel „Programm 2“ auf Seite 13 gibt einen Befehl „17“ mit Parameter „42“ an StEAM.

Listing 3.2: Beispiel Befehlsmuster

```
{
  int i;

  i++;
  i--;
  i++;
  i--;

  i = 17;
  i = 42;

}
```

Konkrete Befehle können bequem mit C++-Makros erzeugt werden, um verschiedene Aktionen durchzuführen, z.B.

- Informationen zu Dateinamen und Zeilen, um den Bezug zwischen Maschinenprogramm und C++-Quelltext herzustellen
- Haupt-Thread festlegen
- Atomare Bereiche festlegen. (Ähnlich kritische Abschnitte)
- Invarianten angeben (Assertions)
- „Künstlicher“ Nichtdeterminismus einführen

3.1.4 Zustandsexpansion

Der Zustandsraum des Programmes wird von StEAM mit den folgenden Schritten erzeugt:

1. Einen der laufenden Threads auswählen
2. Inhalte der CPU-Register, des Stacks und der Variablen-Bereiche wiederherstellen (Kontextwechsel)
3. Eine einzelne Anweisung (oder automaren Block) ausführen (Zustandstransition)
4. Inhalte der CPU-Register, des Stacks und der Variablen-Bereiche einlesen, um den Nachfolgezustand zu erzeugen
5. Neuen Zustand in Hash-Tabelle speichern, falls der Zustand neu ist
6. Die ersten fünf Schritte für alle laufenden Threads wiederholen

3.2 Heuristiken

Einer der wichtigsten Ansatzpunkte für die Effizienz-Verbesserung von StEAM ist die Verbesserung der Such-Heuristiken. Eine Heuristik h gibt zu einem gegebenen Ausgangszustand s die geschätze Distanz $h(s)$ zu einem Fehlerzustand e an. Ein Problem beim Model Checking ist die Tatsache, dass fast nie der gesamte Zustandsraum des Programms bekannt ist und deshalb nicht immer klar zu entscheiden ist, ob eine bestimmte Heuristik zulässig oder konsistent ist.

3.2.1 Fehlerspezifische Heuristiken

Fehlerspezifische Heuristiken sind solche Heuristiken, die darauf zielen, einen Fehler eines bestimmten Typs zu finden.

Most Blocked

Die Most-Blocked-Heuristik schätzt die Distanz mit der Anzahl der nicht-blockierten Prozesse. [ELL01], [GV02]

Lock and Block

Die Lock-and-Block-Heuristik schätzt die Distanz als Anzahl der nicht-blockierten Prozesse plus Anzahl der nicht blockierten Ressourcen.

Formelbasierte Heuristik

Die formelbasierte Heuristik schätzt die Distanz als Anzahl der Transitionen, die nötig sind, um eine gegebene Formel wahr (falsch) werden zu lassen. Diese Heuristik kann von StEAM nicht ohne weiteres eingesetzt werden, weil es keine benannten Variablen(-namen), Funktionen(-namen) und Parameter(-namen) gibt, sondern nur Speicheradressen.

3.2.2 Strukturelle Heuristiken

Im Gegensatz zu fehlerspezifischen Heuristiken suchen die strukturellen Heuristiken nicht nach einem bestimmen Fehlertyp, sondern schätzen ab, welche Fehlerpfade ganz allgemein anfällig für Fehler sind. [GV02] Ein formaler Unterschied ist, dass die fehlerspezifischen Heuristiken eine Distanz *minimieren*, während strukturelle Heuristiken eine Fehlerwahrscheinlichkeit (im umgangssprachlichen Sinne) *maximieren*, sodass wir den gegebenen maximierten Funktionswert von einer Konstante c subtrahieren müssen, die „hoffentlich“ groß genug ist.

Interleaving

Interleaving bevorzugt die Pfade, in denen sich die laufenden Threads in ihrer Ausführung häufig abwechseln.

Branch-Coverage

Branch-Coverage bevorzugt die Pfade, in denen viele Verzweigungsoperationen (Sprünge, Schleifen) stattfinden.

Read-Write

Read-Write bevorzugt die Pfade, in denen die meisten Lese- und Schreiboperationen auf die gleiche Speicherzelle stattfinden.

Alternating Access

Alternating Access bevorzugt die Pfade, in denen die meisten abwechselnden Lese- und Schreiboperationen auf die gleiche Speicherzelle stattfinden.

Threads Alive

Threads Alive bevorzugt die Pfade, in denen die meisten laufenden Threads vorhanden sind.

3.2.3 Pfadbasierte Heuristiken

Die genannten Heurstiken finden Fehler zwar schnell, die zugehörigen Pfade sind jedoch oft sehr lang. Weiter sind diese Heuristiken nicht geeignet, um sie in heuristischen Suchalgorithmen (z.B. A*) [HNR68] einzusetzen, um dem optimalen (=kürzesten) Pfad zu finden. Eine pfadbasierte Heurstik dient dazu, zu einem mit einer der vorhergehenden Heuristiken gefundenen Fehlerzustand den kürzesten Pfad zu finden.

Hamming-Distanz

Die Hamming-Distanz ist weiterhin nicht für heuristische Suchalgorithmen geeignet, in bestimmten Spezialfällen, z.B. beim Vergleich von Schleifenvariablen, geben sie jedoch eine bessere Abschätzung für die echte Distanz zum Zielzustand.

FSM-Distanz

Die FSM-Distanz arbeitet auf einer Abstrahierung ϕ des ursprünglichen Zustandsraums. Bei einem gegebenen Zustand s und einem Zielzustand g ist die FSM-Distanz definiert als kürzester Pfad von $\phi(s)$ nach $\phi(g)$ im abstrakten Zustandsraum.

In diesem Fall verwenden wir den Programmzähler (PC) als Abstrahierung. Dadurch ist es nicht mehr nötig, den gesamten Zustandsraum „abzuwandern“, sondern die Distanzen können direkt aus dem Programmquelltext ermittelt werden. Weiterhin wird der abstrakte Zustandsraum für jeden Thread getrennt betrachtet und als Transitionsgraph bezeichnet. Um die Gesamt-FSM-Distanz zu berechnen, müssen die Einzeldistanzen nur wieder addiert werden, da sichergestellt ist, dass sich bei einer Zustandsänderung der Programmzähler in höchstens einem Thread ändert.

Die FSM-Heurstik gibt es in zwei Varianten, eine operationsbasierte und eine verzweigungsbasierte. Die Operationsbasierte FSM-Distanz betrachtet jeden einzelnen Befehl im Maschinen-Code als Zustandsübergang, während die verzweigungsbasierte FSM-Distanz die Befehle zwischen Verzweigungsbefehlen im Maschinen-Code zusammenfasst.

3.3 StEAM-Heuristiken

In seinem Ansatz von Heuristiken realisiert StEAM entweder einfache Strategien, die nur einen Parameter im Zustand betrachten, oder kombinierte, die mehrere Prioritäten gleichzeitig verfolgen.

3.3.1 Einfache Heuristiken

Lock

Lock ist die Strategie, die Zustände mit mehr blockierten Variablen und mehr aktiven Prozessen bevorzugt. Locks sind offensichtlich die Voraussetzungen, dass ein Prozess eventuell blockiert werden könnte. Natürlich, nur ein aktiver Prozess kann geblockt werden.

Shared Variable

In diesem Fall bevorzugen wir, den aktiven Prozess zu wechseln, nachdem eine globale Variable gelesen oder geschrieben wurde. Die Grundidee ist, dass durch ihre Änderung andere Prozesse beeinflusst werden könnten.

Thread-ID

Wenn die Prozesse identisch sind, und sich nur in der Prozess-ID unterscheiden, unterscheidet sich ihr Verhalten nur unwesentlich. In so einem Fall betrachten wir nur den zuletzt ausgeführten Prozess, d.h.den mit der größten ID.

3.3.2 Komplexere Heuristiken

Da StEAM ein Assembly Level Model Checker ist, ist er in der Lage mehrere Parameter bei der Programmausführung zu betrachten, um aufgrund dessen zu entscheiden welcher Pfad am wahrscheinlichsten genommen werden könnte. Hier ist ein Überblick über die zur Zeit realisierten Heuristiken.

Prefered alive and blocked

Addiert die Anzahl der blockierten Prozesse hoch drei, die Anzahl der lebendigen Prozesse hoch zwei und für jede Gruppe der lebendigen Prozesse die Anzahl der Prozesse hoch zwei gewichtet mit dem wahrscheinlichsten Prozess.

Prefered blocked

Addiert die Anzahl der blockierten Prozesse hoch drei, die Anzahl der lebendigen Prozesse hoch zwei und für jede Gruppe der geblockten Prozesse die Anzahl der Prozesse hoch zwei gewichtet mit dem wahrscheinlichsten Prozess.

Read and Write

Bevorzugt den Schreibzugriff auf globale Variablen, und unterwertet den Lesezugriff ohne direkten Schreibzugriff für jeden Prozess.

Lock and Block

Zählt die Anzahl der Locks in allen Prozessen und die Anzahl der blockierten Prozesse.

Prefered locked 1

Zählt die Anzahl der Locks in allen Prozessen und die Anzahl der blockierten Prozesse hoch zwei.

Prefered locked 2

Zählt die Anzahl der Locks in allen Prozessen hoch zwei und die Anzahl der Gruppen von blockierten und lebenden Prozessen hoch zwei, wobei die wahrscheinlichste Gruppe dadurch gewählt wird, dass die Gruppe mit ihrer größten ID untereinander verglichen werden.

Alternating Read and Write

Bevorzugt abwechselnde Lese- und Schreibezugriffe.

3.4 Bestandsaufnahme

Neben den obene beschriebenen Eigenschaften von StEAM gibt es auch einige Unzulänglichkeiten:

1. Die Zustands-Exploration findet vollständig im Arbeitsspeicher (und ggf. im Swap-Bereich) statt. Ist dieser voll, kann das Modell nicht weiter überprüft werden.
2. Die proprietäre Thread-Syntax erfordert manuelle Anpassungen bei „Real Life“-Programmen, in denen üblicherweise POSIX-Threads oder C-Threads verwendet werden.
3. Auf Grund der bisherigen Konzentration auf den wissenschaftlichen Aspekt von StEAM, ist der für das Überleben eines Software-Produktes unverzichtbare Aspekt der Benutzerfreundlichkeit überhaupt nicht berücksichtigt worden. (Das betrifft insbesondere die komplizierte Installation)

4. Probleme bei der Ausführung des stdout Befehls ließen keinerlei Möglichkeit zu die Ausgabe in Modellen zu machen

Kapitel 4

Integrierte Enwicklungsumgebung für StEAM

4.1 Motivation

StEAM ist ein Model Checker für C/C++ Programme. StEAM verfügt jedoch bis jetzt über keine Arbeitsoberfläche. Das Starten, Steuern und Ausgeben von StEAM erfolgte bislang im Shell-Fenster. Ziel der Gruppe ist die Entwicklung einer GUI für eine integrierte Entwicklungsumgebung, die es dem Software-Entwickler gewährleistet, seine C/C++ Programme in einer einheitlichen Entwicklungsumgebung zu schreiben, Compilieren und anschließend mit StEAM zu überprüfen. Die GUI soll mindestens neben einem Editor über eine Parameterfenster und eine Ausgabekonsole verfügen.

Eclipse bietet als eine mächtige IDE-Plattform eine gute Möglichkeit neue Funktionalitäten zu integrieren und kann dabei eine nahtlose Benuzeroberfläche und eine einheitliche Benuzerführung gewährleisten. Diese Integration läuft über Plugins und Plugin-Erweiterungen.

4.2 Eclipse - Aufbau und Architektur

4.2.1 Einführung

Eclipse ist der Nachfolger von IBM Visual Age for Java 4.0. Seine Entwicklung soll mehr als 40 Millionen Dollar gekostet haben. Der Quellcode für Eclipse wurde Ende 2001 von IBM freigegeben. Die Verwaltung und Entwicklung von Eclipse wurde von der Eclipse-Foundation übernommen.

Eclipse ist ein Integrated Development Environment (IDE) zur Integration verschiedenster Anwendungen. Eine solche Anwendung ist z.B. die mitgelieferte Java Entwicklungsumgebung JDT (Java Development Toolkit). Diese Anwendungen werden in Form sogenannter Plug-Ins zur Verfügung gestellt und von der Eclipse-Plattform automatisch erkannt und integriert.

Das zentrale Element von Eclipse ist der sog. **Workspace**, ein spezielles Verzeichnis im Dateisystem, der verschiedene Ressourcen (Dateien, Ordner und Projekte) enthält . Die Benutzeroberfläche **Workbench** ist aufgeteilt in sog. **Views** und **Editoren**, die sich über **Perspectives** gruppieren lassen. So lässt sich durch Wechseln der Perspektive die gesamte Benutzeroberfläche auf einen Schlag umkonfigurieren.

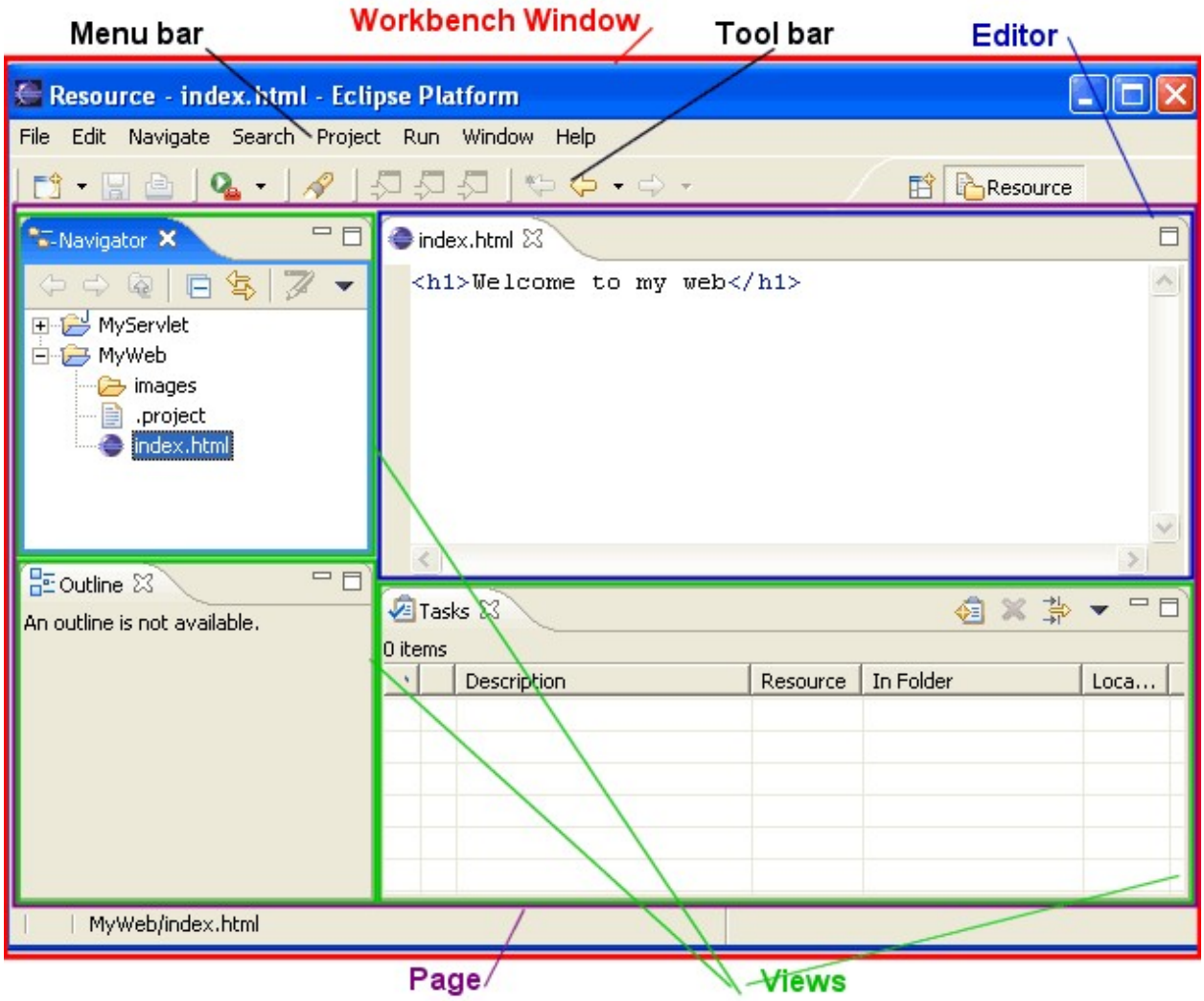

Abbildung 4.1: Eclipse, Die Workbench

4.2.2 Die Workbench

Die Abbildung 4.1 stellt den lauffähigen Applikationsrahmen dar und dient der Interaktion zwischen dem Benutzer und Eclipse. Sie verwaltet sowohl die Ressourcen als auch die Steuerelemente, die die Funktionalität der Plug-Ins steuern. Jede Workbench in Eclipse besteht zusätzlich aus den Komponenten Perspektiven, Views und Editoren, die im Folgenden vorgestellt werden.

Ressourcen

Die Workbench verwaltet drei Arten von Ressourcen: Dateien, Verzeichnisse und Projekte.

Editoren

Ein Editor ist ein Teil einer Workbench und dient zum Bearbeiten einer Ressource. Hierbei wird ein strikter Laden-Verändern-Speichern Lebenszyklus eingehalten.

Perspektiven

Eine Perspektive ist eine Zusammenstellung von Views und Editoren in der Workbench, sowie die zur Verfügung stehenden Werkzeugleisten und Menüs. Mit der Perspektive wird eine von der Aufgabe abhängige, optimale Anordnung von Editoren und Views bereitgestellt. Man kann in einer geöffneten Perspektive jeden anderen View und jeden Editor hinzufügen - auch wenn diese in einem ganz anderen Plug-In definiert wurden. Dies ermöglicht es dem Benutzer, eine auf ihn zugeschnittene Entwicklungsumgebung zusammenzustellen. Eclipse stellt in der Standardversion folgende Perspektiven zu Verfügung:

- Ressource
- CVS Repository Exploring
- Debug
- Java
- Java Browsing
- Java Type Hierarchy
- Plugin Development
- Install/Update

Views

Ein View bietet meist eine Sicht auf die vorhandenen Ressourcen. Je nach View werden nur Teile oder auch innere Zusammenhänge der Ressourcen angezeigt. Er unterstützt Editoren indem er alternative Darstellungen anbietet, oder bei der Navigation und der Verwaltung von Workbenchinformationen hilft. Beispielsweise bietet die Navigator-View Informationen über die Ressourcen der Projekte, während der Properties-View editierbare Objekteigenschaften, z.B. von aktuell editierten GUI-Elementen, bietet. Man kann zwischen folgende Views unterscheiden:

Package Explorer: Im Package Explorer sind alle Projekte mit darin enthaltenen Paketen, Klassen und Methoden zu sehen. Pakete sind keine realen Ressourcen, sondern virtuelle Objekte. Die Paket-Struktur eines Projekts ergibt sich aus den Paket-Deklarationen am Beginn jeder Java-Quelldatei. Die Java-Spezifikation verlangt allerdings, dass sich die Paket-Struktur isomorph auf eine Verzeichnisstruktur abbilden lässt.

Navigator: Der Navigator zeigt Projekte, Verzeichnisse und Dateien und vermittelt somit einen Überblick über die von der Eclipse-Workbench verwalteten Ressourcen und gestattet die Navigation in der Menge der Ressourcen.

Hierarchie: Die Hierarchie zeigt die Super- und Subtypen für Klassen und Interfaces an. Dabei besteht die Möglichkeit, entweder die Sicht nur auf Super- oder Subtypen zu beschränken oder die komplette Hierarchie anzuzeigen. Mit der History-Funktion kann man rasch zwischen den verschiedenen Sichten wechseln oder vorher angezeigte Hierarchien noch einmal anzeigen. Der Hierarchie-Browser besteht aus zwei Fenstern. Im oberen Fenster wird die Typenhierarchie angezeigt, im unteren Fenster werden die Felder und Methoden des selektierten Typs angezeigt.

Outline: Der Outline-View gestattet die Navigation innerhalb einer Quelldatei. Generell ist der Outline nicht auf Java-Programme beschränkt, sondern steht - je nach installierten Plug-Ins - auch für andere Dateitypen zur Verfügung. Bei Java-Programmen stellt der Outline-View sowohl die Felder und Methoden als auch die Import-Anweisungen dar. Sind innere Klassen definiert, werden auch diese Klassen dargestellt: die Hauptklasse und die inneren Klassen bilden eine Baumstruktur. Durch einen einfachen Klick auf ein im Outline dargestelltes Feld oder eine Methode wird der Editor auf die Definition des Feldes bzw. der Methode positioniert. Die verschiedenen Buttons der Outline-Toolbar erlaubt es, bestimmte Elemente aus dem Outline-View herauszufiltern. Mit dem Sortieren-Button z.B. können Felder und Methoden alphabetisch geordnet werden (andernfalls gilt die Reihenfolge in der Quelldatei).

Probleme und Konsole: Im View Problems erscheinen alle Probleme, die beim Kompilieren entstehen und in der Konsole wird der in die Standardausgabe (d.h. System.out.print()) umgeleitete Text gedruckt.

Debug-, Breakpoint-, Variables-, und Expression-View: Diese Views sind ein Bestandteil der Debug-Perspective. Diese Perspektive stellt in Eclipse umfangreiche Debugging-Funktionalität zur Verfügung und ermöglicht eine gute Übersicht über den aktuellen Zustand der virtuellen Maschine. Durch die Evaluierung von Einzelstatements oder eine Überprüfung von Attributen lassen sich viele weitere Informationen über die VM erhalten. Eclipse bietet, wie fast jede andere IDE, natürlich auch eine Möglichkeit zum Remote Debugging, die besonders bei Server-Anwendungen hilfreich sein kann. Im Debug-View sieht man beim Debugging die aktuell ausführende Zeile. Gleichzeitig sieht man im Breakpoints-View alle Breakpoints und im Variables-View die aktuelle Variabelenbelegung. Ausserdem lässt sich ein beliebiger Ausdruck im Expressions-View beobachten.

4.2.3 Eclipse Architektur

Eclipse ist in Java geschrieben und daher weitgehend betriebssystemunabhängig. Die Benutzeroberfläche ist entwickelt in SWT, einem parallel mit Eclipse entstandenen Open-Source-Projekt, das die problembehafteten GUI-Bibliotheken AWT und Swing Stück für Stück ersetzt.

Eclipse ist eine generische Applikationsplattform mit einer ausgeklügelten Plug-In Architektur 4.2. Architektonisch gesehen besteht Eclipse aus einem relativ kleinen sprachunabhängigen Kern, um den in unterschiedlichen Abstraktionsschichten Plug-Ins gelegt werden. Eclipse selbst ist sprachneutral - auch die Java-Entwicklungsumgebung ist lediglich ein Plug-In -, auch wenn die Mehrheit der Plug-Ins bislang noch Java-zentriert ist.

4.3 Eclipse und C/C++: CDT-Plugin

Das CDT (C Development Toolkit) Plug-In erlaubt, C oder C++ Applikationen mit Eclipse zu entwickeln. Neben dem Plug-In muss der gcc-Compiler auf dem Rechner installiert sein. Dieser ist in mingw oder cygwin enthalten. Das Plug-In ist in eine zip-Datei verpackt.

4.3.1 Installation und Konfiguration des CDT Plug-Ins:

Unter Windows

1. MinGW (Gnu Compiler für Windows) und MSYS installieren

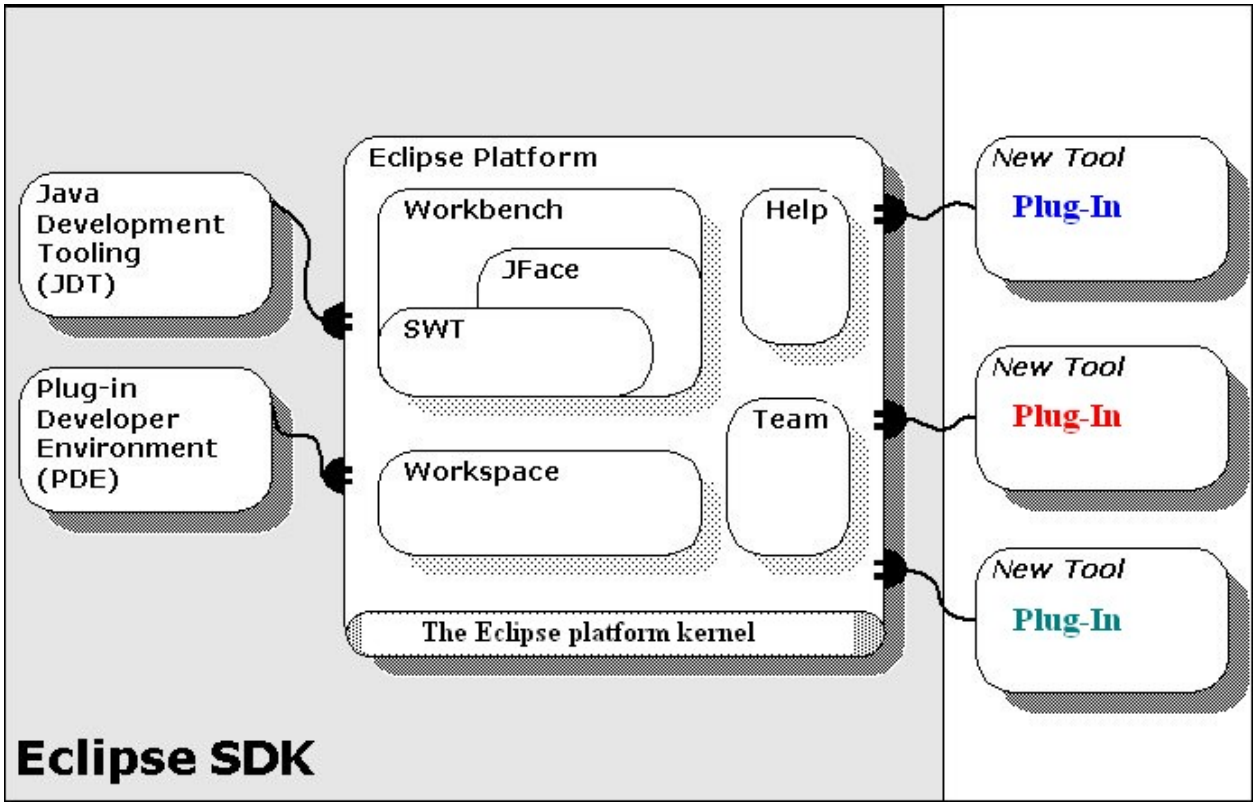

Abbildung 4.2: Eclipse, Plattform-Architektur

2. Umgebungsvariable-Path anpassen:
   Rechtsklick auf Arbeitsplatz -> Eigenschaften -> Erweitert -> Umgebungsvariablen -> Systemvariablen -> Path
   dann folgende Werte:
   C:\MinGW\bin\; C:\MSYS\bin\; C:\MSYS\mingw\;
   anhängen.

3. CDT installieren:
   Eclipse starten -> Help -> Software Updates -> Find and Install... -> New Remote Site -> New features to Install -> New Remote Site:
   http://download.eclipse.org/tools/cdt/releases/eclipse3.1 eintragen.

4. Rechtsklick auf das zu compilierendes C++ Projekt -> Propertys -> C/C++-Build -> Build-Settings:
   "Use Default Command" deaktivieren und stattdessen folgenden Befehl eintragen:
   mingw32-make -k dann Apply drücken (um sich zu ersparen diesen Befehl bei jedem Projekt einzugeben, kann man diese Einstellungen unter Window -> Preferences -> Managed Make durchführen.

Unter Linux

Unter Linux ist nur Schritt 3 erforderlich. Falls die Umgebungsvariable PATH alle Pfade zum Compiler enthält, sollte Eclipse CDT diese automatisch erkennen.

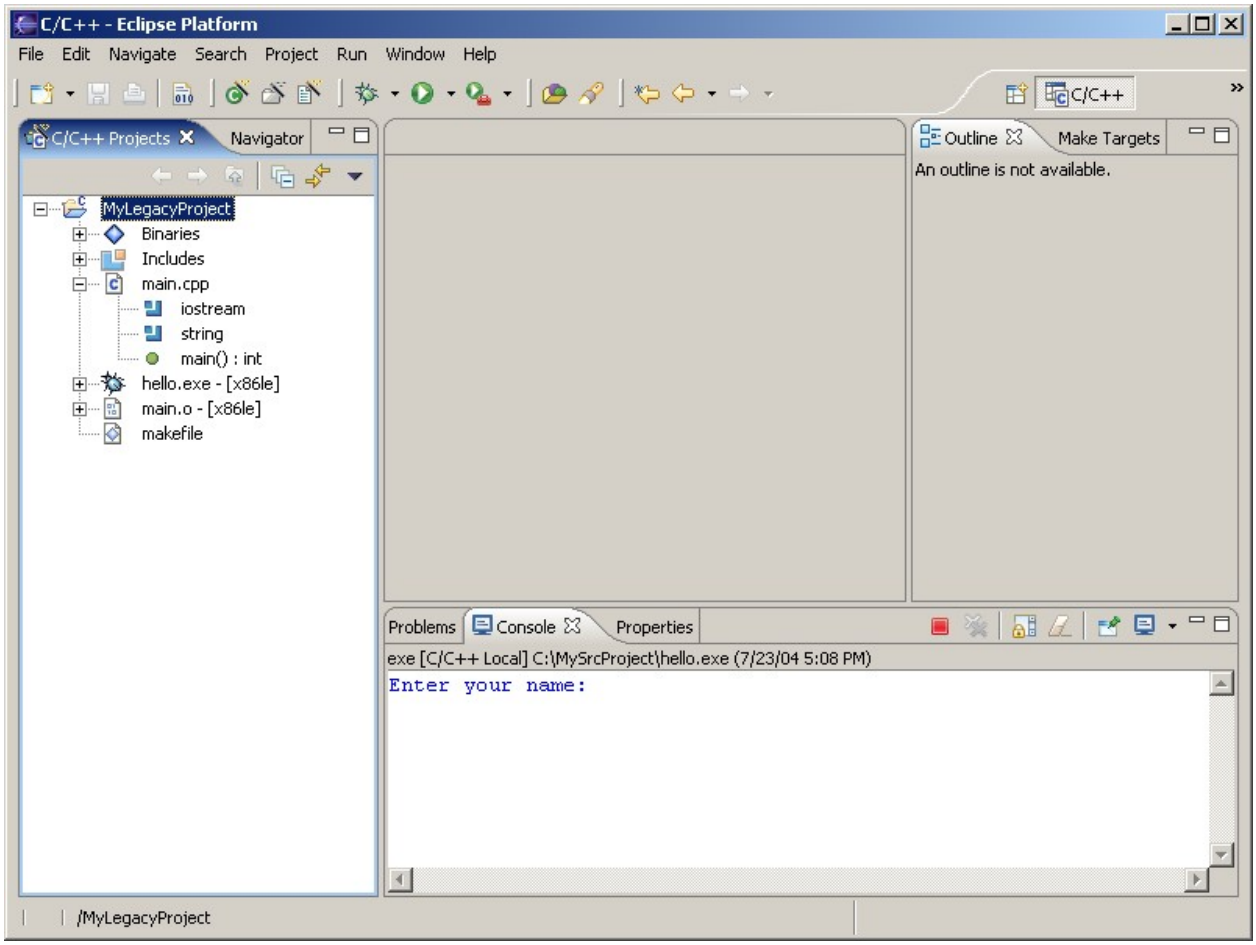

Abbildung 4.3: Eclipse, C++ Perspective

4.4 Plugin Entwicklung

Plugins sind Erweiterungsmodule für Eclipse, die zusätzliche Funktionalitäten zu Verfügung stellen. Eclipse ist bis auf einen kleinen Kernel eine Sammlung von Plugins. Diese Plugins dienen der Erweiterung und Ergänzung der Funktionalität von Eclipse. Dabei ist die Erweiterbarkeit und Komponentenorientierung eines der Basiselemente von Eclipse.

4.4.1 Die Plugin Struktur

Eclipse ist als eine offene, erweiterbare Plattform entwickelt worden. Dies ist realisiert worden, in dem man einen kleinen Plattformkern geschaffen hat, der dazu dient, Plugins zu starten. Die Workbench selbst ist nur ein Plugin der Eclipse Plattform. Jede Eclipse Installation besitzt ein Plugin-Verzeichnis, in dem sich die installierten Plugins befinden. Die einzelnen Plugins besitzen dort ein Unterverzeichnis, in dem sich die Ressourcen und die das Plugin beschreibende Manifestdatei plugin.xml befinden.

4.4.2 Initialisierungsvorgang

Beim Starten von Eclipse, werden vom Plattformkern die Informationen aus den im Plugin-Verzeichnis befindlichen Manifestdateien ausgelesen und anschließend in einem Teil des Plattformkerns, der Plugin-Registry, registriert. Die Plugins werden erst bei Bedarf aus diesem Repository geladen. Diese Methode des Ladens bezeichnet man als Lazy Loading, was eine Verbesserung der Ladezeit der Entwicklungsumgebung gewährleistet.

Die Manifestdateien werden ausschließlich während des Startvorgangs gelesen, deswegen können die Plugins nicht während der Laufzeit installiert werden. Daher ist ein Plug-In Entwickler gezwungen, immer eine zweite Eclipse-Instanz zu starten, in der das Plugin getestet werden kann.

Ein wichtiges Prinzip von Eclipse ist, dass Plugins nicht isoliert existieren. Sie bauen aufeinander auf und ergänzen ihre Fähigkeiten. Dies bedingt die Kommunikation der Plugins untereinander. Dabei muss die Kommunikation folgendes beinhalten: um welche Funktionalität sie die Plattform erweitern und um welche Funktionalität andere das Plugin erweitert können. Um diese Anforderungen zu erfüllen, ist das Kernkonzept der Plugin Architektur, die Extension Points, entstanden. Zur Zusammenarbeit ist es notwendig, die Informationen über diese Extension Points zu kommunizieren. Ein Extension Point definiert für ein Plugin, um welche Funktionalität das Plugin von anderen Plugins erweitert werden kann. Da Plugins Extension Points verwenden und selbst welche publizieren, entsteht ein komplexes Netz von Abhängigkeiten untereinander.

4.4.3 Plugin Manifest-Datei

Das Plugin Manifest ist ein XML-Dokument, welches die Informationen über den strukturellen Aufbau des Plugins für die Plattform beinhaltet. Einer der notwendigen Bestandteile eines Plugins ist die Manifestdatei plugin.xml im Verzeichnis des Plugins. Üblicherweise beginnt die Entwicklung eines Plugins mit der Erstellung dieses XML-Dokuments. Diese Datei beinhaltet neben einer allgemeinen Beschreibung des Plugins auch die zuvor erwähnten Relationen zu anderen Plugins. Die Manifest-Datei muss mindestens folgende Einträge besitzen: einen Namen für das Plugin, einen eindeutigen Identifikator (ID) und die Versionsnummer des Plugins. Darüber hinaus werden in der Manifest-Datei die Extensions und Extension-Points deklariert.

Listing 4.1: Manifest-Datei

```
<?xml version="1.0" encoding="UTF-8" ?>
 <?eclipse version="3.0"?>
 <plugin>
   <extension point="org.eclipse.ui.views">
```

```
      <category
        name="BugFinder Category"
        id="BugFinder">
      </category>
      <view
        name="Bugs View"
        icon="icons/bugview.gif"
        category="BugFinder"
        class="steam4clips.views.BugsView"
        id="steam4clips.views.BugsView">
      </view>
    </extension>
    <extension point="org.eclipse.ui.actionSets">
      <actionSet label="Steam" visible="true" id="steam4clips.actionSet">
      <menu label="&StEAM" id="steam4Clips.menu">
      .
      .
      .
    </extension>
  </plugin>
```

4.4.4 Plugin Installation

Um das erstellte Plug-In Projekt zu installieren, muss man es zu einem ZIP-Archiv machen, indem man

>>***File*** >> ***Export*** >>**...**>>***Deployable Plugins and fragments***

wählt.

Anschließend muss das erstellte ZIP-Archiv im Verzeichnis:

$ECLIPSE_HOME/plugins

entpackt werden. Die Plug-Ins können allerdings erst nach einem Neustart von Eclipse verwendet werden. Je nach installiertem Plugin sind jetzt neue Perspektiven auswählbar, neue Einträge in Menü- und Werkzeugleiste vorhanden oder in untergeordneten Strukturen neue Funktionalitäten eingetragen.

4.5 Oberfläche

Eclipse wurde als intergrierte Entwicklungsumgebung für StEAM genutzt und wurde deshalb so erweitert, dass StEAM-fähige Programme damit entwickelt und getestet werden können.

Mit Hilfe der verschiedenen API, die von Eclipse zur Verfügung gestellt werden, haben wir einige Änderungen in Eclipse vorgenommen. Dabei wurde für die Entwicklung der GUI hauptsächlich das *Standard Widget Toolkit - SWT* benutzt. SWT stellt allgemein gebräuchliche Komponenten(Widgets) wie zum Beispiel Schaltflächen, Menüs, Bäume, Tabellen bereit. Der Grund der Benutzung von SWT und nicht SWING, wie bei gewöhnlichen Java-Programmen, liegt an der prägnantesten Definition des SWT, die auf der Komponenten-Hompage nachzulesen ist:[1]*Die SWT-Komponente ist dafür ausgelegt, effizienten portablen Zugriff auf die Einrichtungen der Benuzeroberflächedes Betriebssystems zu bieten, auf dem sie implementiert ist.*Um dieses Ziel zu erreichen, implementiert das SWT eine dünne Schicht über der nativen Komponenten des Betriebs-

[1] http://dev.eclipse.org/viewcvs/index.cgi/ checkout /platform-swt-home/main.html

systems. Wenn eine Komponente auf einer bestimmten Plattform nicht verfügbar ist, implementiert das SWT diese Komponente in Java.

Die Realisierung einer GUI in SWT beruht auf der Klasse *Shell*. Ein Shell-Objekt repräsentiert ein Fenster auf dem Desktop. Der Rest der Komponenten wird durch das Zusammenstellen von BasisKomponenten mit *Composite*-Komponenten gebildet.Um das Ganze konkreter darzustellen, geben wir hier ein Teil des Codes unserer Oberfäche wieder:

Listing 4.2: Parsing report.xml

```
private Shell shell;

    public SWTGUI(Shell parent, int style){
        super(parent, style);
    }

    public void open(){
        Shell parent = getParent();
        shell = new Shell(parent, SWT.BORDER | SWT.MIN | SWT.CLOSE);
        shell.setText("StEAM Plugin");

        shell.setLayout(null);
        shell.layout();
        shell.pack();
        shell.setSize(465, 532);

        Group grouphashing = new Group(shell, SWT.NONE);
        .
        .
        Button Hashing = new Button(grouphashing, SWT.CHECK | SWT.LEFT);
        .
        .
        Button full = new Button(grouphashing, SWT.RADIO | SWT.LEFT);
        .
    }
}
```

Mit Hilfe von *org.eclipse.ui.actionSets* erweiterten wir zunächst die Menüleiste mit einem neuen Menü Namens *StEAM*, siehe Abbildung 4.4, was uns ermöglicht alle Parameter von *StEAM* auswählen zu können. Die Parameterauswahl findet über ein Steuerungsfenster 4.5 statt. Dabei wird die zum Start von *StEAM* benötigte ivm-Datei bei Starten der GUI automatisch vom aktuellen Projekt ermittelt und ihren Pfad in dem dafür vorgesehen Textfeld eingetragen. Hilfe 4.6, wie die Parameter zusammengesetzt werden können bekommt man auch mit durch das neu eingebaute Menü, und ist somit unabhängig von der Shell-Console von Linux.

Sobald alle benötigte Parameter ausgewählt wurden, kann man dann das *Modell Checker StEAM* mit dem dafür vorgesehen Knopf starten.

4.6 mfgen Integration

Mfgen ist ein Tool, der dazu dient StEAM-geeignete Makfiles zu erzeugen. Ein so erzeugtes Makefile beinhaltet alle sich im Projektordner befindenen Dateien mit den Endungen *.c* oder *.cc*.

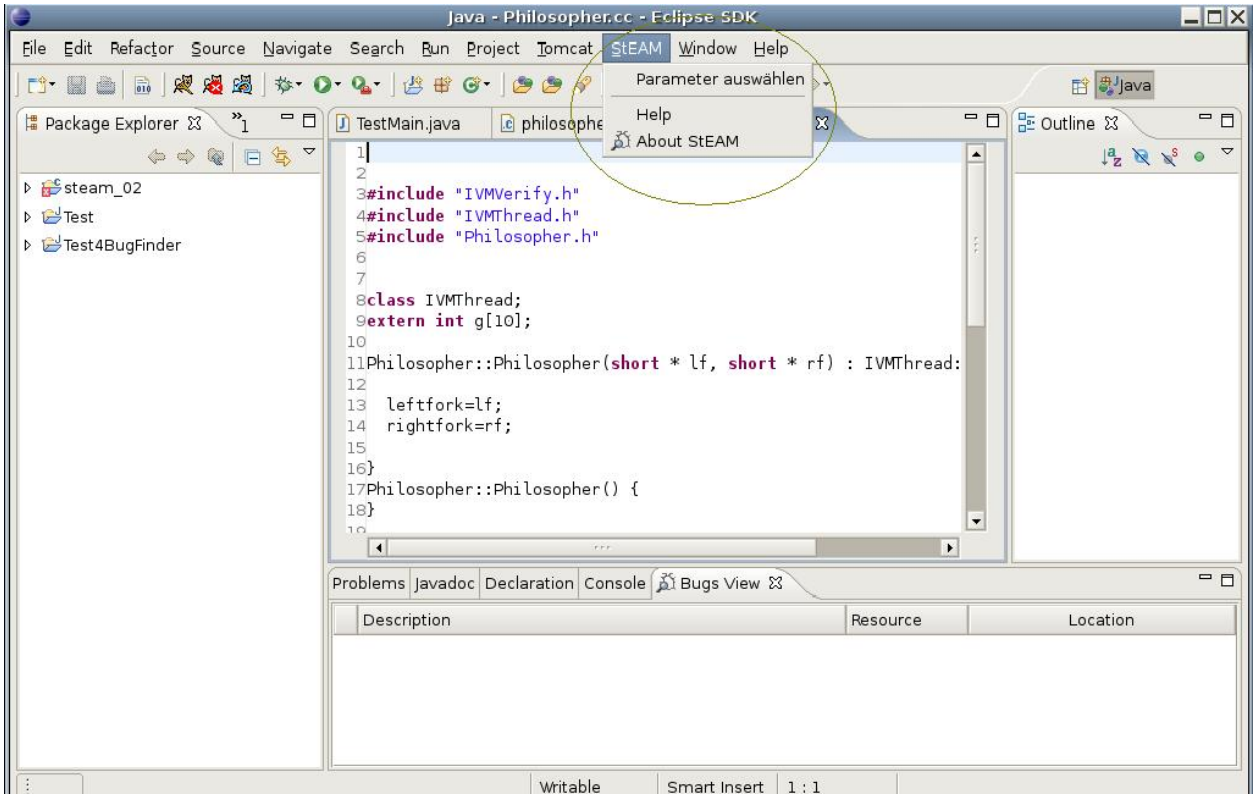

Abbildung 4.4: Eclipse StEAM-Menu

Mfgen wurde in Eclipse integriert als eine Erweiterung des Toolbar-Menus. Diese Integration erfolgt in der Datei *plugin.xml*. Hier wurde, genauso wie im vorigen Abschnitt schon, als Extension Point die Klasse *org.eclipse.ui.actionSets* verwendet, bei der eine ObjectContribution durchgeführt wird. Dadurch kann genau gesteuert werden, für welche Objekte *mfgen* in die PopUp-Menus integriert werden.

Listing 4.3: Shell-Objekt

```
<extension point="org.eclipse.ui.actionSets">
    <actionSet
        label="steam"
        visible="true"
        id="steam4clips.actionSet"
    >
    <menu id="steam4clips.menu" label="&amp;StEAM"/>
    <separator name="SteamGroup"/>
    <action
        id="steam4clips.MfgenAction" label="mfgen"
        tooltip="Makefile erzeugen" class="steam4clips.actions.MfgenAction"
        menubarPath="steam4clips.menu/SteamGroup"
        style="push" state="false"/>
</extension>
```

Hier sehen Sie einen Ausschnitt des XML-Codes des *Menus*, den wir eingefügt haben.

Von großer Bedeutung ist hier allein ein Objekttyp, der gesamte Projektordner. Bei der Anwendung des erzeugten Knopfs, werden alle Ressourcen des aktuellen Projektordners durchsucht und alle für die Erzeugung der Makefile notwendigen Dateien ermittelt und anschließend wird die Makefile im Ordner angelegt.

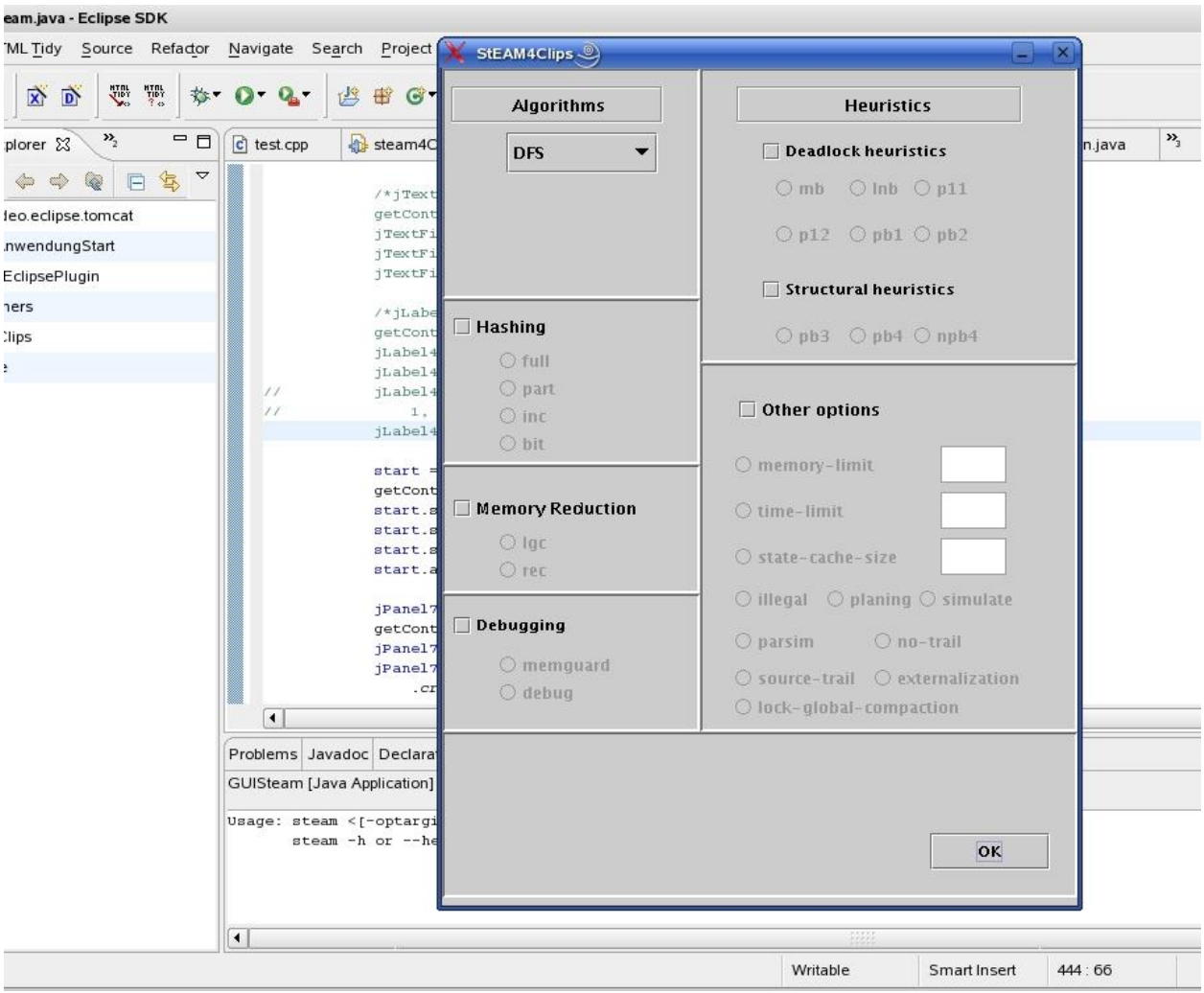

Abbildung 4.5: Steuerungsfenster von StEAM

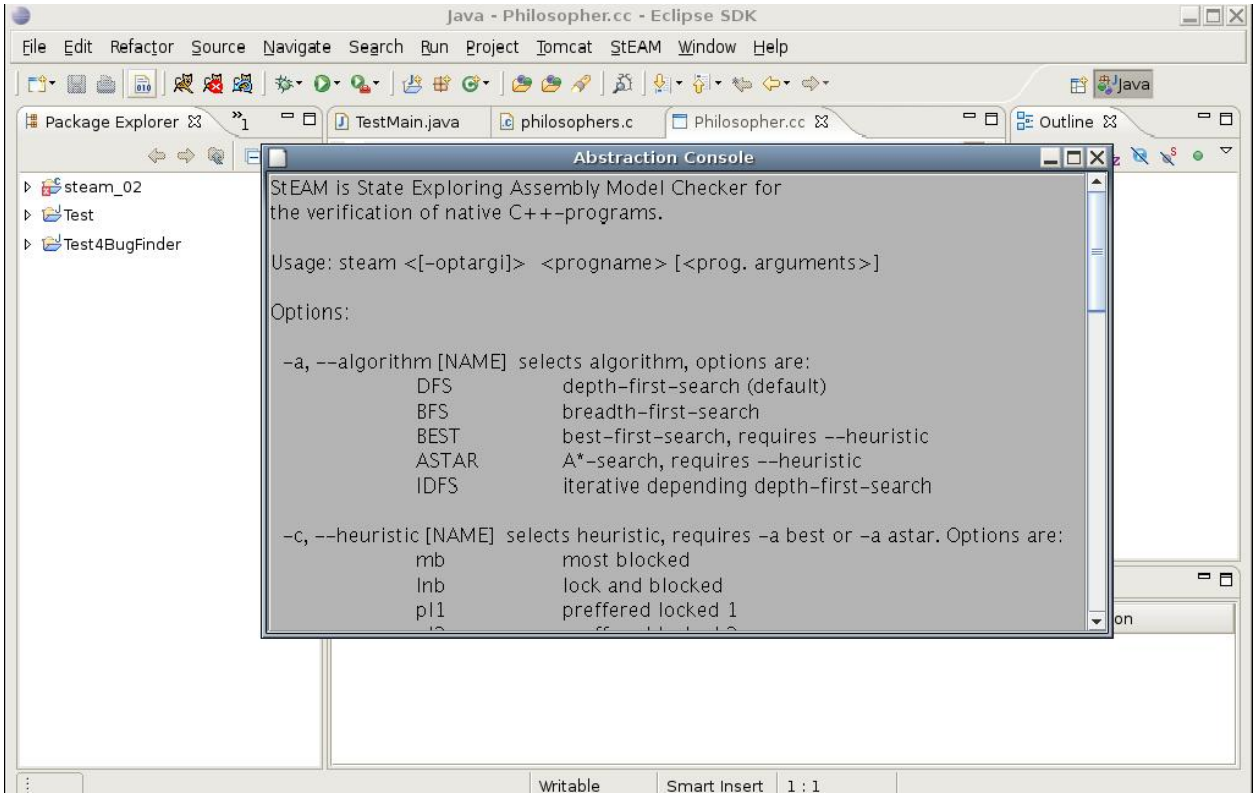

Abbildung 4.6: Hilfe für StEAM

Um das Ausführen des PlugIns auf Grund der Auswahl im PopUp-Menu anzustoßen, ist in der Datei *plugin.xml* auf eine Klasse zu verweisen, die von der Klasse IObjectActionDelegate erbt. Diese hat Zugriff auf das Objekt, für das das PopUp-Menu geöffnet wurde, und ermöglicht somit das Anlegen des Makefiles in den entsprechenden Ordner.

4.7 Error-Trailer

Nach einem StEAM-Aufruf wird auf die Eclipse-Konsole zusätzlich zu der StEAM-Ausgabe ein Bericht im XML-Format (report.xml) erzeugt, was verschiedene Informationen, wie z.B. Laufzeit, Speicherbenutzung und Fehlerpfad enthält. Unser Ziel ist es das XML-Dokument zu parsen und anschließend die Auswertungen des XML-Parsers wie z.B. Fehlerpfad zu highlighten.

4.7.1 XML (Extensible Markup Language)

XML definiert, wie Daten strukturiert in Textdateien gespeichert werden. XML-Datenstrukturen sind auch für andere als die ursprüngliche Anwendung verständlich. Daten können plattformunabhängig zwischen verschiedenen Anwendungen ausgetauscht werden. XML ist als Metasprache erweiterbar und ist ebenso wie HTML und XHTML eine Untermenge von SGML. XML 1.0 wurde Anfang 1998 vom W3C als Standard verabschiedet.

XML-Dokumente können beliebig strukturierte Daten enthalten. Die Elemente in XML bilden eine Baumstruktur, welche die interne Struktur des Dokuments abbildet. Dieser Baum kann durch einen XML-Parser aufgebaut werden. Auf diese Weise wird für eine Anwendung der Zugriff auf einzelne Baumknoten ermöglicht.

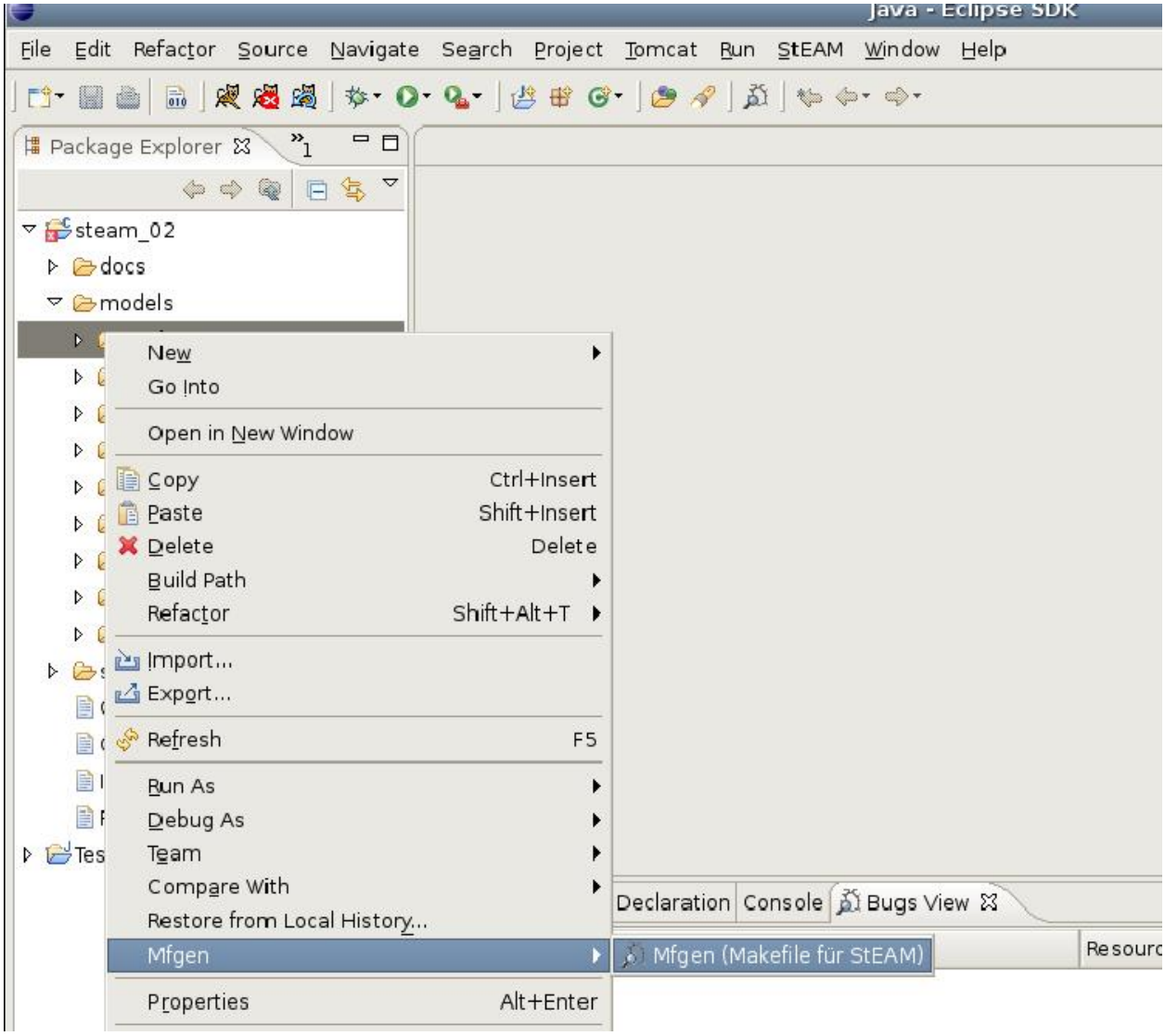

Abbildung 4.7: StEAM-Mfgen

4.7.2 XML-Parser

Ein XML-Dokument kann nicht direkt von einer Anwendung verarbeitet werden, sondern wird in der Regel erst von einem Parser in seine Bestandteile zerlegt. Dabei wird es auf Fehlerfreiheit überprüft. Der Parser liest ein Dokument ein und übermittelt an die Anwendung die Eigenschaften und die Struktur der Daten.

XPath

Die XML Path Language (XPath) ist eine vom W3C-Konsortium entwickelte Anfragesprache, um Teile eines XML-Dokumentes zu adressieren. XPath dient als Grundlage einer Reihe weiterer Standards wie XSLT, XPointer und XQuery. Ein XPath-Ausdruck adressiert Teile eines XML-Dokuments, das dabei als Baum betrachtet wird, wobei einige Unterschiede zum "klassischen"Baum der Graphentheorie zu beachten sind:

- Knoten (nodes) des Baumes sind XML-Elemente, -Attribute, -Kommentare, -Namensräume und -Verarbeitungsanweisungen.

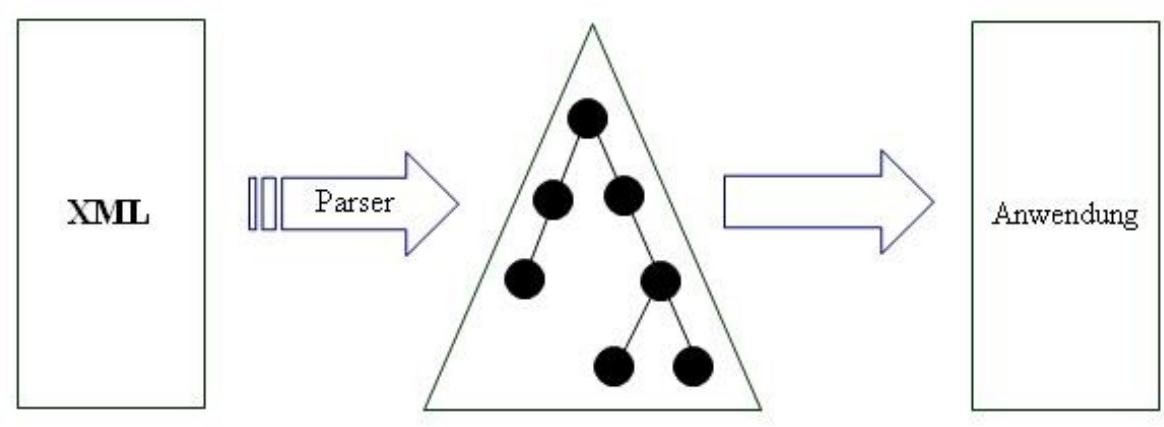

Abbildung 4.8: Der DOM-Parser erstellt eine Baumstruktur aus XML-Daten

- Nur XML-Elemente gelten als child im Sinne einer XPath-Achse. Alle anderen Knoten stehen auf der gleichen Hierarchie-Stufe des Baumes wie der Knoten, der sie enthält.
- Die Achsen preceding, following, preceding-sibling und following-sibling orientieren sich nicht allein an der Baumstruktur, sondern auch an der Reihenfolge der Deklaration der Elemente im XML-Dokument (Linked-Tree).

Document Object Model

Das Document Object Model (DOM) -definiert vom World Wide Web Consortium (W3C)- ist ein plattform- und sprachneutrales Interface (API), welches es erlaubt auf die Struktur und den Inhalt eines XML-Dokuments dynamisch zuzugreifen und diese zu verarbeiten. Der DOM-Parser liest das XML-Dokument, analysiert und validiert es und baut es in Form einer Baumstruktur im Speicher. Dabei wird jeder Bestandteil des XML-Dokuments als Knoten in die Struktur aufgenommen.

Im ersten Schritt wird ein bestehendes Dokument durch das Programm eingelesen und ein Dokument-Objekt erzeugt. Anhand dieses Objekts kann mittels der Methoden des API auf die Inhalte, Struktur und Darstellung zugegriffen werden. Insbesondere erlaubt DOM:

- die Navigation zwischen den einzelnen Knoten eines Dokuments,
- das Erzeugen, Verschieben und Löschen von Knoten sowie
- das Auslesen, Ändern und Löschen von Textinhalten.

Nach erfolgreichem Abschluss des Parsens stehen die XML-Daten in Baumform im Speicher zur Weiterverarbeitung zur Verfügung.

4.8 Highlighting

Eclipse unterstützt ein *Syntax coloring* API, welches unabhängig von jedem möglichen Parsing und AST Management ist. Diese API beruht auf einem separaten Parser, welcher auf die Erkennung üblicher Elemente, die gefärbt werden müssen, begrenzt ist.

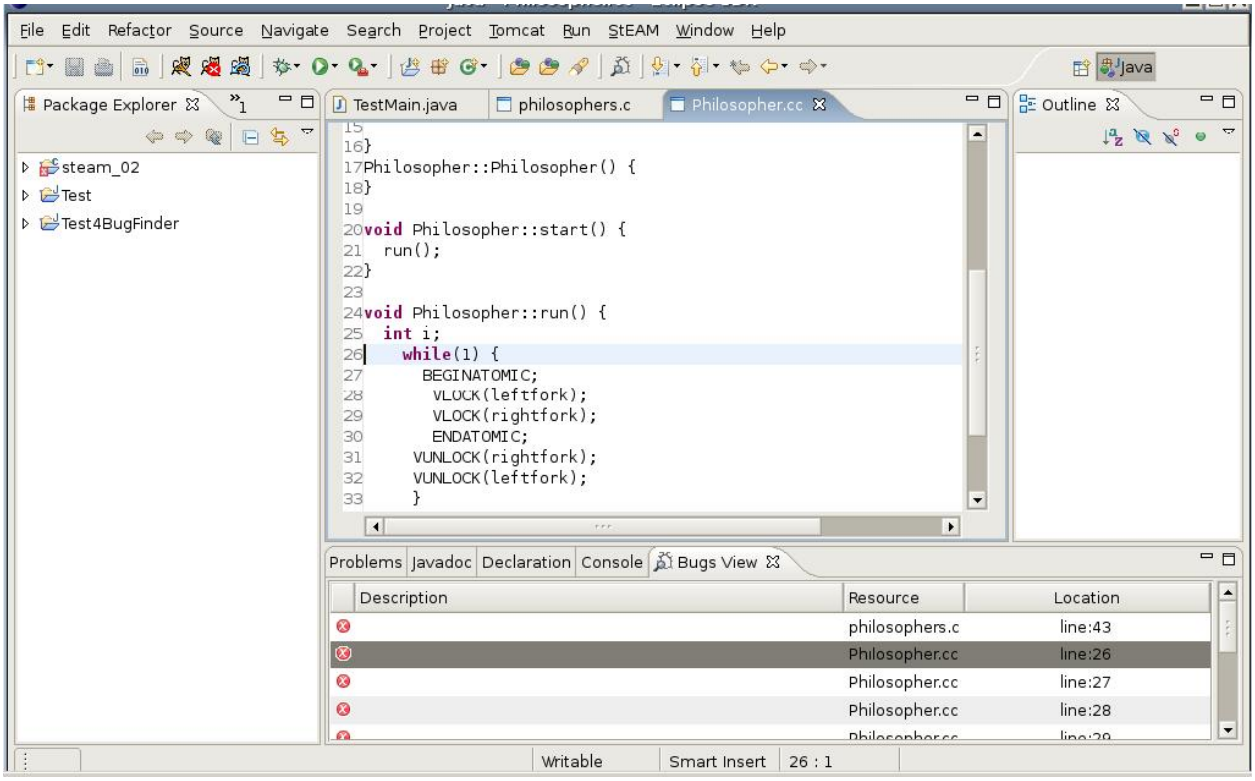

Abbildung 4.9: Highlighting und Error-Report für StEAM

StEAM erzeugt nach einem erfolgreichen Durchlauf eine XML-Datei report.xml, die sowohl den Errorpfad als auch Statistiken enthält. Diese Datei wird anhand der im vorigen Abschnitt erläuterten techniken geparst und anschließend für das Highlighting benötigt.

Um das Syntax-Highlighting zu realisieren wurde zunächst ein neues *View*, siehe Abb. , erzeugt, die zur Anzeige des Fehlerpfads dient. Hier folgt der dazu benötigt XML-Code:

Listing 4.4: Syntax Highlighting

```
<extension point="org.eclipse.ui.views">
    <category name="BugFinder Category"
        id="BugFinder">
    </category>
    <view
        name="Bugs View"
        icon="icons/bugview.gif"
        category="BugFinder"
        class="steam4clips.views.BugsView"
        id="steam4clips.views.BugsView">
    </view>
</extension>
```

Unter

Window > Show View > Other > BugFinder Category > BugsView

kann die neue Erweiterung an der dafür vorgesehenen Stelle in Eclipse angezeigt werden.

Das View besteht hauptsächlich aus eine Tabelle. Hier werden Informationen angezeigt, die aus der Report.xml geparst wurden. Mit Hilfe von verschiedenen Methoden, die von SWT zu Verfügung gestellt werden,

haben wir dann das Syntax Highlighting erweitert. Der folgende Quellcode zeigt uns zum Teil die von uns benutzte Vorgehensweise:

Listing 4.5: Parsing report.xml

```
File xmlfile = new File("report.xml");
XMLParser inst = new XMLParser();
String str = inst.fehlerPfad();
BugsView.table.removeAll();

if(xmlfile.exists()){
 for(int i=0; i<=xmlfile.length();i++){
  String filename = str.substring(str.indexOf("␣")+1,str.indexOf(":"));
  String thread = str.substring(0,str.indexOf("␣"));
  String zeile = str.substring(str.indexOf(":")+1,str.indexOf("\n"));
  str=str.substring(str.indexOf("\n")+1,str.length());
  BugsView.bugsList.addEntry(
   "error", "thread:␣"+thread, filename, "line:"+zeile);
 }
} else {
 MessageDialog.openInformation(
  PlatformUI.getWorkbench.getActiveWorkbenchWindow().getShell(),
   "StEAM␣Plugin", "Fehler␣beim␣Highlighting,
␣␣␣␣report.xml␣nicht␣gefunden!");
}

.
.
.
private void makeActions(){
 Action doubleClickAction = new Action(){
  public void run(){
   ISelection selection = new TableViewer().getSelection();
   Object obj =((IStructuredSelection)selection).getFirstElement();
   String lineStr = ((BugEntry)obj).getLocation();
   .
   .
   .
  }
 }
}
```

Die zur Realisierung des Highlightings notwendigen Techniken wurden zum größten Teil mit Scarpino [Sca05] erlernt.

4.9 Ausblick: Der Eclipse-Debugger

Mit Hilfe eines Debuggers lässt sich ein Programm an einer bestimmte Stelle anhalten und die Ausführung des Codes zeilenweise weiterverfolgen. Die Debug Perspective in Eclipse stellt umfangreiche Debugging-Funktionalität zur Verfügung und ermöglicht eine gute Übersicht über den aktuellen Zustand der virtuellen Maschine. Durch die Evaluierung von Einzelstatements oder eine Überprüfung von Attributen lassen sich

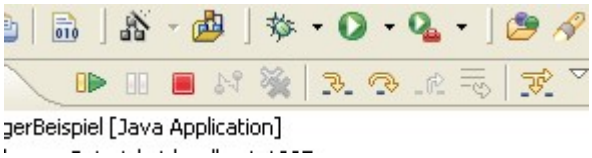

Abbildung 4.10: Eclipse, Debug Buton-Leiste

viele weitere Informationen über die VM erhalten. Die Debug View dient der Überwachung der gestarteten Prozesse. In der Werkzeugsleiste dieser View sowie im Kontextmenü kann man Aktionen ausführen, die sich auf einen selektierten Eintrag, d.h. einen laufenden Thread beziehen.

Im Debug-View hat man folgende Button-Leiste zur Verfügung:

Mit dem grünen ,,Play"-Button kann man die Programmausführung fortsetzen, so dass das Programm erst beim nächsten Breakpoint wieder stoppen würde. Mit dem ,,Pause"-Button kann man einen Thread anhalten zum Zweck der Analyse oder zum Bearbeiten von Werten von Variablen zur Laufzeit. Mit dem roten ,,Stop"-Button kann man das Programm beenden (bspw. dann, wenn man einen Fehler gefunden hat).

Diese umfangreiche Debugging-Funktionalität kann man in unserem Plugin für StAEM integrieren, um das Plugin zu erweitern und um eine gute Navigation im Fehlerpfad zu gewährleisten.

Kapitel 5

Abstraktion

5.1 Parsing

Der Parser bildet das vordere Ende der Abstraktions-Engine. Er soll die einem C++-Project zugehörigen Dateien einlesen und vorbereiten. Dabei sollen so viele Informationen wie möglich erhalten bleiben.

Das Problem an dieser Aufgabe bestand darin, dass die quelloffenen Parser, die verfügbar sind, nur überprufen, ob eine oder mehrere C++-Dateien korrekt sind. Das heißst, dass das ,,Output" des Parsers lediglich aus ,,true" , der C++-Code ist korrekt, und ,,false", der Code enthält syntaktische Fehler, besteht.

Für die Abstraktion auf Quellcodeebene reicht es nicht aus zu wissen, ob eine Datei syntaktische Fehler enthält, vielmehr ist es wichtig, die in dieser Datei enthaltenen Strukturen, das "Programm", zu erkennen.

JavaCC - ein Parser Generator

Da lediglich einer der quelloffenen Parser-Generatoren schon ein C++-Language-File enthält, wird er in diesem Projekt verwendet. Dieser Generator ist JavaCC. JavaCC steht für ,,Java Compiler Compiler". JavaCC ist ein Programm, dass aus einer speziellen Sprachdatei einen Parser, der Programme der Sprache liest, generiert und entscheidet, ob sie syntaktisch korrekt sind. Dabei liest er die Datei ,,Wort für Wort", oder spezieller ,,Token für Token", ein und gibt notfalls entsprechende Fehlermeldungen aus.

Da, wie oben erwähnt, die Ausgabe ({true,false}) des generierten Parsers uninteressant ist, wurde der von JavaCC generierte C++-Parser erweitert. Ziel ist es während des Parsens alle verfügbaren Datenstrukturen an der entdeckten Stelle zu speichern. Um das zu erreichen, hat jedes Objekt ein ,,Gedächnis" in Form eines Vectors[1] erhalten. Diese Daten werden zur Generierung einer universellen Programm-Datenstruktur benutzt, was unter Datenstrukturkonvertierung etwas näher erläutert wird.

Language-File Cpp.jj für JavaCC

Diese Datei ist enstanden nachdem einige Veränderungen an der Sprachdatei Cplusplus.jj, die online verfügbar ist, vorgenommen wurden. Die Sprachdatei ist sehr alt (ca. 10 Jahre), so dass viele Neuerungen nicht enthalten sind. Sie hatte unter anderem den Fehler, dass eine Zeichensequenz für Zeilenende gefehlt hat, was dazu führte, dass der Parser unter Angabe falscher Meldungen abbrach.

5.1.1 Aufbau des Abstraktionsparsers

Der Parser ist eigentlich eine Kollektion von verschiedenen Programmen. Dabei bildet der von JavaCC generierte C++-Parser das Backend der Programme. Er arbeitet auf einem InputStream und leistet die

[1] Datenstruktur in Java, die alle Eigenschaften von Liste und Array besitzt

Erkennungsarbeit. Er besteht aus zwei Teilen. Der erste Teil ist das Einlesen von Zeichenketten auf Streamebene und der zweite Teil arbeitet auf erkannten Worten und versucht zu erkennen, ob es sich bei der Eingabe um eine gültige C++-Eingabe handelt. Diese beiden Teile wurden in diesem Projekt aber kaum verändert, so dass hier auch nicht genauer darauf eingegangen wird.

Nachdem der C++-Parser die Datei eingelesen und seine interne Datenstruktur aufgebaut hat, konvertiert der Abstraktionsparser in mehreren Schritten die interne C++-Parser-Datenstruktur in die Datenstruktur, die an die Abstraktionsalgorithmen übergeben wird.

5.1.2 Interne DS

Die interne Datenstruktur des C++-Parsers besteht aus den erwarteten Elementen `Token` und `Scope`. `Token` ist dabei ein erkanntes Wort oder Zeichen. In der Klasse CPPParserConstants.java sind die ,,Arten" aufgelistet, die ein Token sein kann. Ein Token enthält seine String-Repräsentation und seine Art. Die folgende Tabelle enthält eine Zeile Quellcode in Stringrepräsentation und Art. Die Zeile besteht insgesamt aus sieben Token.

String:	int	a	=	b	+	4	;
Token Art:	INT	ID	ASSIGNEQUAL	ID	PLUS	DECIMALINT	SEMICOLON

Abbildung 5.1: Tokenzerlegung einer Zeile Quellcode

`Scope` ist ein Gültigkeitsbereich. Dieser Gültigkeitsbereich wird auch häufig Körper genannt. Beispiele sind hier der Methodenkörper oder Schleifenkörper. Er sammelt eine Abfolge von geparsten `Token`. Da ein Scope aber auch Unterscopes, SubScopes, haben kann, ensteht eine baumartige, rekursive Struktur. Jedes Scope enthält eine Liste von Elementen, die in ihm gefunden wurden. Die Elemente können entweder Scopes oder Tokens sein. Start ist immer das ROOT-Scope. Eine Erweiterung dieses Projekts war, jedem Scope seinen Typ zuzuordnen. Vorher sah ein Scope mit dem Inhalt einer Methode genau so aus wie das Scope einer Klasse. Um unterscheiden zu können wurden die folgenden Typen bestimmt:

- Root
- Class
- Constructor
- Destructor
- Function
- Structure
- If
- Else
- For
- While
- Do
- Unknown

5.1.3 Datenstrukturkonvertierung

Um die Abstraktion von Quellcode effizient durchführen zu können, wurde eine Metadatenstruktur entworfen, die im Kapitel ,,Datenstruktur" näher erklärt wird. Da die interne Datenstruktur des Parsers nicht ausreicht um Abhängigkeiten darzustellen, wie sie im Kapitel ,,Dependency-Graph" erklärt werden, musste der Parser so erweitert werden, dass ein Abstraktionsparser entstand. Der Abstraktionsparser nimmt die interne Datenstruktur des C++-Parsers und erzeugt daraus die Metadatenstruktur, auf der die Abstraktion dann ausgeführt werden kann. Dazu wurden zwei Hauptfunktionen erstellt. Die eine erzeugt für jedes Scope in der internen Datenstruktur ein entsprechendes Objekt. Die andere fügt diesem Objekt seinen Inhalt ein.

5.1.4 Ausblick

Der Parser ist im Moment noch nicht sehr robust gegenüber den Eingabedateien. Er ist sehr anfällig für schlecht formatierte Eingaben. Deshalb ist der Benutzer dieses Systems gut beraten, wenn er den zu prüfenden Code nach Pretty-Print-Standards schreibt. Eine gute Erweiterung wäre deshalb den Parser robuster gegenüber den Eingabe zu machen. Da zur Generierung des Parsers via JavaCC nur ein stark veraltetes Sprachfile zur Verfügung stand, besteht eine Möglichkeit in der Erweiterung des Parsers auf einen aktuellen C++-Standard. Dazu muss man das Sprachfile so umschreiben, dass alle neuen Konstrukte ünterstützt werden. Leider war es nicht möglich eine Grammatikbeschreibung für Gnu CC zu erhalten, was der wichtigste Compiler dieses Projektes ist, da er eng mit StEAM verknüpft ist. Ein Beispiel für ein nicht unterstütztes Konstrukt ist ,,`using namespace <name>;`".

5.2 Datenstruktur

Für die Abstraktion von Quellcode ist es wichtig das Programm und seine Struktur verarbeiten zu können. Diese Struktur ist in jeder imperativen,objektorientierten Programmiersprache ähnlich. Ein Programm besteht im Wesentlichen aus:

- Variablen
- Startmethode
- Klassen
  - Klassenvariablen
  - Klassenmethoden

Die Methoden enthalten das eigentliche Programm. Die obige Strukturierung dient im Prinzip nur der Übersichtlichkeit. ,,Nur" in den Methoden wird Code definiert. Dieser Code wird zusätzlich durch Konstrukte strukturiert. Beliebtes Beispiel hierfür ist die `If`-Anweisung.

```
if (Bedingung) {Code}
```

Die in dem Code-Block definierten Variablen sind nur in diesem Block gültig. Für die Abstraktion ist es notwendig die Abhängigkeiten zwischen diesen Strukturen zu erkennen. Die hier entworfene Datenstruktur bildet diese Eigenschaften ab. Die Datenstruktur enthält die folgenden Struktur-Klassen:

- `AbsProgram` - Das zu abstrahierende Programm
- `AbsClass` - Eine Klasse des Programms
- `Method` - Eine Methode
- `AbsStructure` - Ein Gültigkeitsbereich
- `Variable` - Eine Variable

Diese Klassen reichen aus eine stark vereinfachte Version eines objektorientierten Programms dar zu stellen. Die Abstraktion soll aber auch einen Output in Form eines Programms liefern. Dazu sind noch mehr Informationen notwendig. Diese Informationen sind durch zusätzliche Klassen abgebildet. Diese Klassen sind:

- `AbsBaseElement` Verwaltung der Zeilen einer Struktureinheit
- `AbsFile` Verwaltung der Zeilen einer Datei
- `Line` Darstellung einer Zeile
- `LineManager` Zuordnung zwischen Zeile und Datei
- `Statement` Knoten des Srukturgraphen

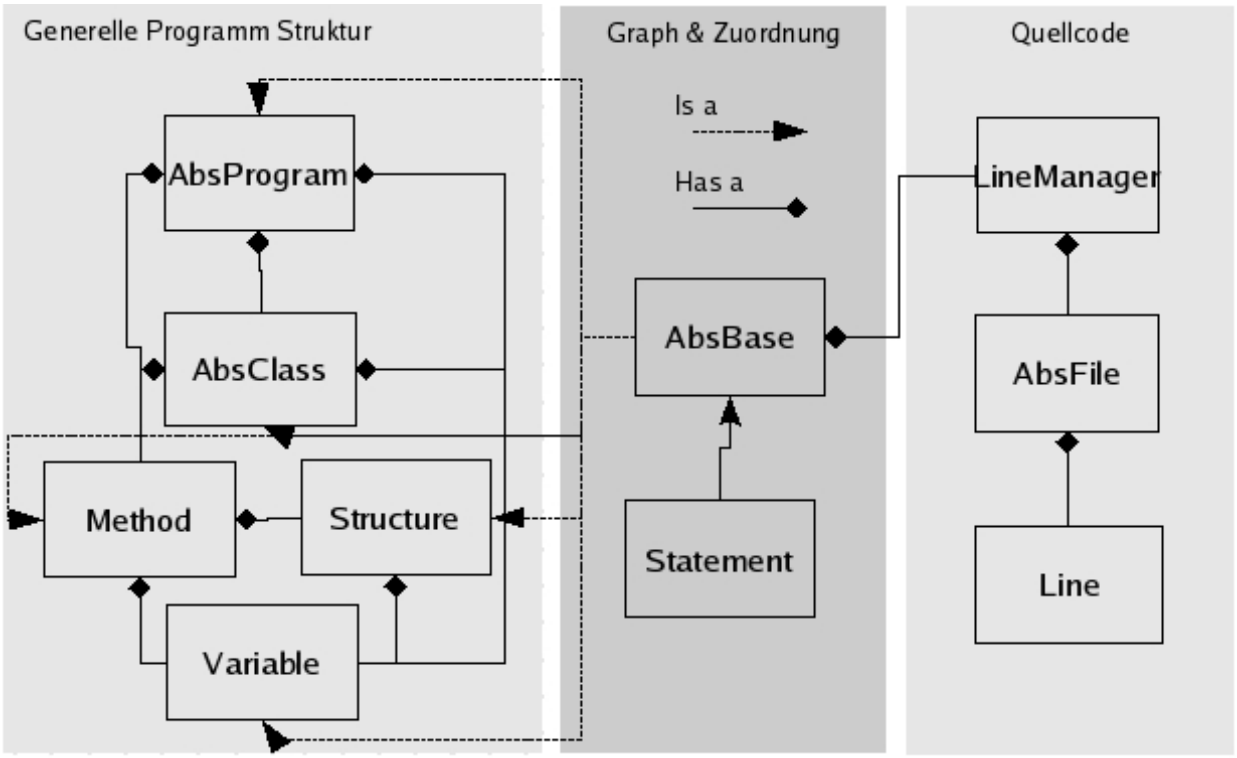

Abbildung 5.2: Metadatenstruktur - Ein Überblick

5.2.1 Gliederung der Datenstruktur

Wie in der Abbildung 5.2 zu sehen ist, gliedert sich die Datenstruktur in drei Teile. Diese drei Teile stellen verschiedee Zusammenhänge dar. Der erste Teil, die “generelle Programmstruktur”, beinhaltet eine objektorienterte Version der einzelnen Programmteile. Hier wird eine Variable zum Beispiel in der Methode gespeichert, in der sie definiert wird. Der zweite Teil gliedert sich in zwei verschiedene Bereiche. Ein Teil ist das AbsBaseElement, dass dazu dient einem abstrakten Objekt aus dem ersten Teil Quellcode zuzuordnen. Nur so kann man bei Bedarf den Quellcode eines Objektes verändern. Dazu geht man wie folgt vor: Man “reduziert” das zu verändernde Objekt auf sein “AbsBaseElement”-Objekt und ruft darauf die entsprechende Funktion zur Veränderung des Quellcodes auf. Der andere Teil des zweiten Blocks ist “Statement”. Diese Klasse dient zur Verwaltung verschiedener Abhängigkeiten zwischen den verschiedenen abstrakten Objekten. “Statement”-Objekte werden dem Abhängigkeitsgraph als Knoten dienen. Nur durch diese Datenstruktur ist es möglich den Quellcode zu verändern, denn “Statement” ist die Verbindung zwischen dem Abhängigkeitsgraphen und dem Quellcode in Form von Dateien. Der dritte Teil ist die Verwaltung des Quellcodes für die abstrakten Objekte. Der “Linemanager” verwaltet alle eingelesenen Dateien und die Veränderungen an diesen. Nach einer erfolgreichen Abstraktion wird er den Quellcode in entsprechende Dateien schreiben. Diese Dateien befinden sich im gleichen Pfad, wie ihre “Vorfahren”, jedoch mit dem Präfix “Abs”.

5.3 Objektorientierter Abhängigkeitsgraph

Die meisten slicing Algorithmen basieren auf Programm-Abhängigkeitsgraphen. Mit Hilfe dieser Graphen kann das Programm-Slicing zu einem Graph-Reachablility Problem reduziert werden. Das Programm wird als eine Menge von Knoten repräsentiert, die von einem ausgewählten Knoten erreichbar sind. Die Slicing-Techniken für prozedurorientierte Programme unterscheiden sich von den Techniken für objektorientierte Programme. Für das Slicing von nicht objektorientierten Programmen genügt der so genannte „Program Dependency Graph“ (PDG), der die Control- und Data- Dependencies darstellt. Der implizierte Control Dependency Graph beinhaltet die Programmanweisungen und deren Reihenfolge. Die Datenabhängigkeiten zwischen den Anweisungen werden durch gerichtete Kanten zwischen den Knoten der Anweisungen repräsentiert.

Der Algorithmus für Slicing von objektorientierten Programmen läuft auf einem obektorientierten Programm-Abhängigkeitsgraph (ODG), der die Struktur der objektorientierten Prgramme repräsentiert.

Die Definition des Graphen basiert auf einem Property-Multigraph, der einen erweiterten gerichteten Graphen repräsentiert. Der Property-Multigraph unterscheidet sich durch mehrere Kantenarten, Knoteneigenschaften und Eigenschaftsrelationen. Die verschiedene Kantenarten repräsentieren verschiedene Arten von Abhängigkeiten zwischen den Entitäten, die Knoteneigenschaften repräsentieren die eingehenden Objekte einer Entität (z.B. Attribute von einem komplexen Objekt), und die Eigenschaftsrelationen repräsentieren die internen Abhängigkeiten zwischen den eingegangenen Objekten. Der Vorteil dieses Graphen ist, dass die verdeckten Abhängigkeiten, die aus der Objekt Aggregation resultieren, beim Slicing eliminiert werden können. Der Algorithmus, der auf einem objektorientierten Abhängigkeitsgraph basiert, kann präzisere Ergebnisse liefern, indem er die internen Abhängigkeiten des Objektes untersucht.[CW97]

Definition1: Ein Property-Multigraph ist ein Tupel $(V, (E_1, E_2, ..., E_n, A)$, wo

1. V ist eine endliche Menge von Knoten.
2. $E_i \subseteq V \times V$ ist eine endliche Menge von gerichtete Kanten. Für eine Kante e, $e \in E_j$, $1 \leq j \leq n$, von einem Knoten v_1 zu einem Knoten v_2, v_1 ist Startknoten von e, notiert als $IV(e)$, und v_2 ist Endknoten von e, notiert als $TV(e)$.
3. A ist eine endliche Menge von Knoteneigenschaften. $A = A(X)$, wo $A(X)$ die Eigenschaften beinhaltet, die mit dem Knoten X assoziiert sind. $A(X)$ besteht aus zwei endlichen Mengen: Auf der einen Seite die benutzten Knoteneigenschaften $U(X)$ und auf der anderen Seite die definierten Knoteneigenschaften $D(X)$, d.h., $A(X) = U(X) \cup D(X)$.

Ein Objektorientiertes Programm besteht aus Anweisungen, Methoden, Attributen (auch als Objekte referenziert) und Klassen. Diese Einheiten werden in dem Graphen mit den folgenden Knoten dargestellt: Anweisungen (auch Statements), Formal-In und Formal-Out Parameter, Methoden, Attribute und Klassen. Sie sind wie folgt definiert (die Shapes von den Knoten werden in der Abbildungen gezeigt):

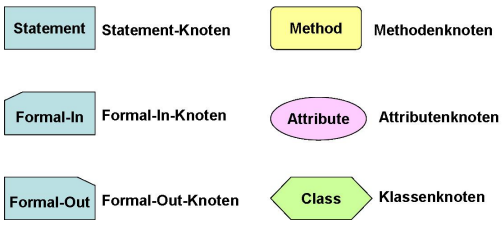

Abbildung 5.3: ODG Knoten

- Der Statement-Knoten repräsentiert eine Programmanweisung, Funktionsaufruf oder Control-Anweisungen (wie z.B. if-then-else oder while Schleifen) innerhalb einer Methode oder Funktion.

- Der Formal-in Knoten repräsentiert den Zustand eines formalen Eingabe-Parameter am Anfang der Methode.
- Der Formal-out Knoten repräsentiert den Zustand eines formalen Ausgabe-Parameter am Ende der Methode.
- Ein Methodenknoten repräsentiert eine Programm- oder Klassenmethode oder Funktion.
- Ein Attributenknoten repräsentiert ein Klassenattribut (Variable).
- Der Klassenknoten repräsentiert eine Klasse im Programm.

Die Abhängigkeiten zwischen den Programmentitäten beinhalten Vererbung (bei den Klassen), Membership, Friend-Beziehung, Methodenaufruf, Parameter-, Control- und Daten-Abhängigkeiten. Jede solche Abhängigkeit wird als gerichtete Kante von einer Entität zur Anderen repräsentiert. Die Ziel-Entität ist von der Quell-Entität abhängig. Diese Abhängigkeiten sind unten genauer beschrieben und in der Abbildung dargestellt.

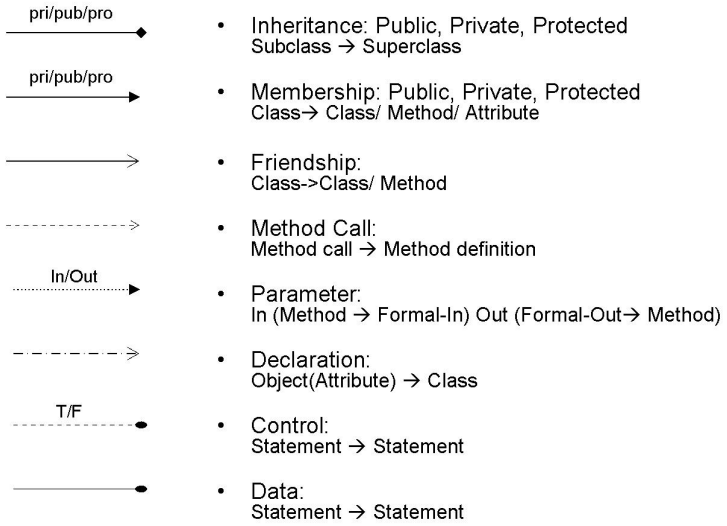

Abbildung 5.4: ODG Kanten

- Eine Vererbung-Abhängigkeitskante ist von einer Superklasse zu einer Subklasse. Diese Kante kann mit *Pub*, *Pro*, oder *Pri* beschriftet werden. *Pub* repräsentiert public, *Pro* - protected, und *Pri* - eine private Vererbungsabhängigkeitskante.
- Eine Membership-Abhängigkeitskante ist von einer Klasse zu einem Attribut oder Methode. Diese Kante kann auch mit *Pub*, *Pro*, oder *Pri* beschriftet werden.
- Eine Friend-Abhängigkeitskante ist von einer Klasse C zu einer Methode M, was bedeutet, dass M Freund von C ist. M darf auf die privaten Attribute und Methoden zugreifen, die in C deklariert sind.
- Eine Methodenaufruf-Abhängigkeitskante von einer Programmanweisung zu einer Methode, die von der Anweisung aufgerufen wird.
- Eine Formal-In oder Formal-Out Abhängigkeitskante repräsentiert Input- bzw. Output- Parameter-Abhängigkeiten. Formal-In ist von einer Programmanweisung zu einem Formal-Parameter der aufgerufenen Methode. Formal-Out ist von einem Formal-Out-Parameter der aufgerufenen Methode zu der Aufrufanweisung.
- Eine Deklaration-Abhängigkeitskante ist von einer Klasse zu einem Objekt, um zu zeigen, dass das Objekt eine Instanz von der Klasse ist.

- Eine Control-Abhängigkeitskante repräsentiert die Control-Abhängigkeiten zwischen den Programmanweisungen (Statements), Funktionsaufrufen und den Control-Anweisungen, wie Schleifen und Entscheidungskonstrukte.
- Eine Daten-Abhängigkeitskante repräsentiert die Datenabhängigkeiten zwischen den Programmanweisungen und den Formal-In/Out Parametern in einer Methode. Eine Kante von Knoten X zu Knoten Y bedeutet, dass Y abhängig ist von X.

Definition2: Angenommen dass $P = (M, \{C_1, C_2, ...C_p\})$ ein OO-System ist, wobei man mit M das Main-Programm notiert, und C_i eine Klasse ist für $1 \leq i \leq p$. Ein objektorientierter Abhängigkeitsgraph (ODG) von P ist definiert als $G_{odg}(P) = (V_{odg}, E_{odg}, A_{odg}, R_{odg})$, wo $(V_{odg}, E_{odg}, A_{odg}, R_{odg})$ ein Property-Multigraph ist, und

1. $V_{odg} = V_c \cup V_m \cup V_a \cup V_s \cup V_{f-in} \cup V_{f-out}$ wo V_c eine Menge ist von Klassenknoten, die $C_1, C_2, ..., und C_p$ repräsentieren.

   V_m ist eine Menge von Methodenknoten, die die in M definierten Funktionen und die in $C_1, C_2, ..., und C_p$ definierten Methoden repräsentiert,

   V_a ist eine Menge von Attributenknoten, die die Attribute in $C_1, C_2, ..., and C_p$ repräsentiert,

   V_s ist eine Menge von Statement-Knoten, die die Programmanweisungen repräsentieren,

   V_{f-in} ist eine Menge von Formal-in Knoten, die die Initialzustände der Formal-Parameter, globalen Objekte und Attribute repräsentieren, auf die zugegriffen wird,

   V_{f-out} ist eine Menge von Formal-out Knoten, die die Endzustände der zurückgegebenen Objekte, Formal-Parameter, der geänderten Attribute und globalen Objekte repräsentieren.

2. $E = (E_{i-pub}, E_{i-pro}, E_{i-pri}, E_{m-pro}, E_{m-pri}, E_{m-fri}, E_l, E_f, E_{p-in}, E_{p-out}, E_c, E_d)$ wobei

   $E_{i-pub} \subseteq V_c \times V_c$ eine Menge ist von $public - inhritance\ dependency$ Kanten,

   $E_{i-pro} \subseteq V_c \times V_c$ eine Menge ist von $protected - inheritance\ depedency$ Kanten,

   $E_{i-pri} \subseteq V_c \times V_c$ eine Menge ist von $private - inheritance\ depedency$ Kanten,

   $E_{m-pub} \subseteq V_c \times (V_m \cup V_a)$ eine Menge ist von $publcmembership\ dependency$ Kanten,

   $E_{m-pro} \subseteq V_c \times (V_m \cup V_a)$ eine Menge ist von $protectedmembership\ dependency$ Kanten,

   $E_{m-pri} \subseteq V_c \times (V_m \cup V_a)$ eine Menge ist von $privatemembership\ dependency$ Kanten,

   $E_{m-fri} \subseteq V_c \times V_m$ eine Menge ist von $friend - relateionship\ dependency$ Kanten,

   $E_l \subseteq V_c \times (V_s \cup V_a)$ eine Menge ist von $declaration\ dependency$ Kanten,

   $E_f \subseteq (V_s \cup V_{f-in}) \times V_m$ eine Menge ist von $procedure - call\ dependency$ Kanten,

   $E_{p-in} \subseteq (V_s \cup V_{f-in}) \times V_{f-in}$ eine Menge ist von $parameter - in\ dependency$ Kanten,

   $E_{p-out} \subseteq V_{f-out} \times (V_s \cup V_{f-in})$ eine Menge ist von $parameter - out\ dependency$ Kanten,

   $E_c \subseteq (V_s \cup V_m \cup V_{f-in}) \times (V_s \cup V_{f-in} \cup V_{f-out})$ eine Menge ist von $control\ dependency$ Kanten,

   $E_d \subseteq (V_s \cup V_{f-in}) \times (V_s \cup V_{f-out})$ eine Menge ist von $data\ dependency$ Kanten.

Beispiel:

Calc.cpp Test.cpp

```
class Calc {
public:
   Calc();
   int addNumbers(int a,int b);
   int calculateSquare(int d);
   int addAndSquare(int a,int b);
   int type;
};
Calc::Calc(){
    type=1;
}
int Calc::addNumbers(int a,int b){
    a = a+b;
    return a;
}
int Calc::calculateSquare(int d){
    int r=0;
    r = d*d;
    return r;
}
int Calc::addAndSquare(int a,int b){
    int r=0;
    r = addNumbers(a,b);
    r = calculateSquare(r);
    return r;
}
int main4(int o){
    Calc calc;
}
```

Abbildung 5.5: C++ Beispielsprogramm

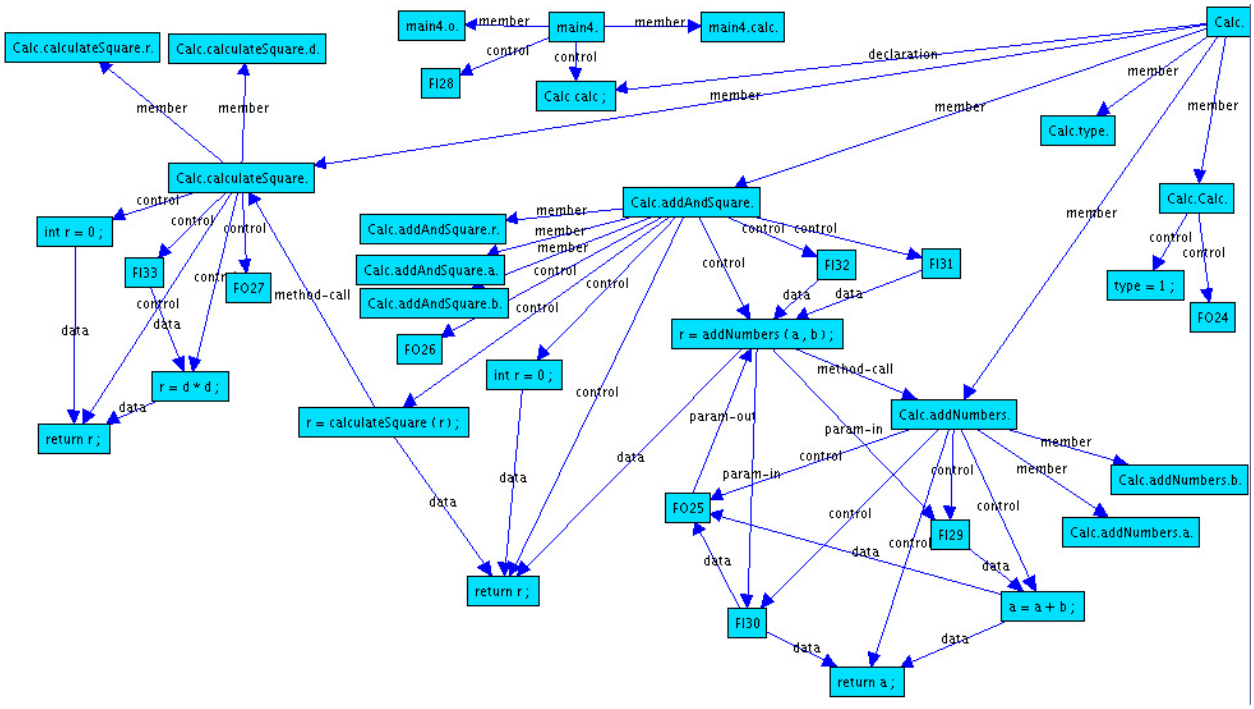

Abbildung 5.6: Dependency-Graph von dem Beispielsprogramm

5.4 Slicing

Unter *Slicing* (genauer: Program Slicing) versteht man die Analyse eines Computerprogramms mit dem Ziel herauszufinden, welche Anweisungen eines Programms eine bestimmte Anweisung in einem bestimmten Programmpunkt beeinflussen oder umgekehrt von ihr beeinflusst werden. Ausgehend von einer Programmeigenschaft reduziert Slicing das Programm auf eine minimale Form die immer noch diese Eigenschaft erfüllt. Program Slicing findet Anwendung in der Softwareentwicklung, unter anderem beim Debugging, Verständnis neuer Programme, Programmtest und Model-checking. Beim Model-checking können z.B. große Programme erst gar nicht geprüft werden, da ihr Zustandsraum exponentiell groß sein kann und das zu inakzeptablen Laufzeiten führen kann. Im Unterschied zu *Data Abstraction* werden bei *Slicing* bestimmte Variablen und Anweisungen wegabstrahiert. Die meisten Slicing-Algorithmen basieren auf Programmabhängigkeitsgraphen. Auf einem Graphen kann Slicing auf das Erreichbarkeitsproblem reduziert werden, wobei der Slice eine Menge von Knoten ist, die über bestimmte Arten von Kanten mit dem Knoten, der das Slicing-Kriterium repräsentiert, direkt oder über andere Knoten verbunden sind. Weiser definierte das Slicing-Kriterium als:

> A slicing-criterion of a program P is a tuple(i,V), where i is a statement in P and V is a subset of the variables in P

Sein Algorithmus benutzt Datenflussanalyse auf Kontrollabhängigkeitsgraphen um intra und interprozedurale Slices zu berechnen. Die meisten Techniken für Slicing sind an prozeduralen Programmen orientiert.

In der Abbildung 5.7 ist ein einfaches Beispielprogramm und der dazugehörige Abhängigkeitsgraph.

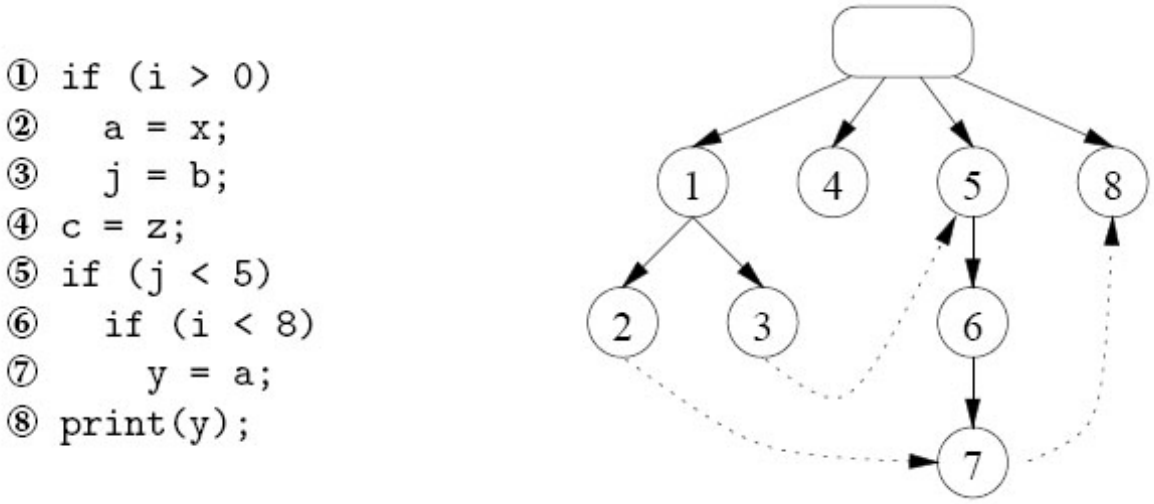

Abbildung 5.7: Beispielprogramm mit Abhängigkeitsgraph

Sei Zeile 8 das Slicing-Kriterium. Der zugehörige Slice enthält den gesamten Ausschnitt bis auf Zeile 4(siehe Abbildung 5.8)

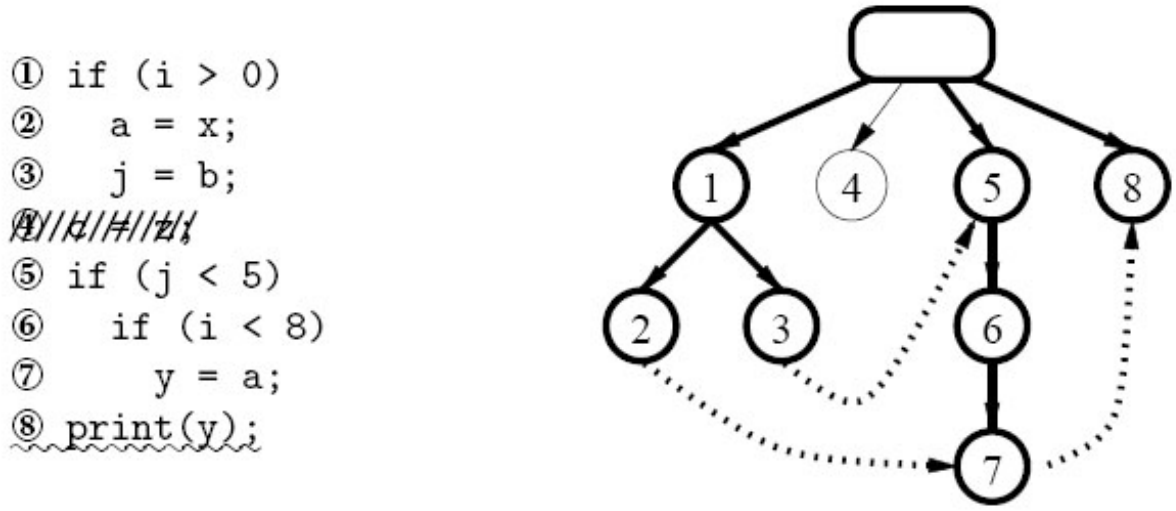

Abbildung 5.8: Slice des Beispielprogramms

Man stellt fest, dass der Wert der Variablen i im Prädikat der If-Anweisung in Zeile 1 den Wert der Variablen y in Zeile 8 beeinflusst. Mit Hilfe des Abhängigkeitsgraphen können wir alle Pfade zwischen den Zeilen 1 und 8 verfolgen, das sind 1→2→7→8 und 1→3→5→6→7→8

Bei objektorientierten Programmen muss man Eigenschaften wie Vererbung, Polymorphie und Verkapselung berücksichtigen. Bei Vererbung erhalten die erbenden Klassen die Methoden und Eigenschaften der Vaterklasse und können sie erweitern oder durch eigene ersetzten. Polymorphie ist eine wichtige Eigenschaft von objektorientierten Programmen, die es erlaubt zur Laufzeit eine aus mehreren Definitionen einer Methode anhand der Parameter zu wählen oder zwischen Variablen gleichen Namens und verschiedenen Typs zu unterscheiden. Als Kapselung bezeichnet man den kontrollierten Zugriff auf Methoden bzw. Attribute

von Klassen. Klassen können den internen Zustand anderer Klassen nicht in unerwarteter Weise lesen oder ändern. Eine Klasse hat eine Schnittstelle, die darüber bestimmt, auf welche Weise mit der Klasse interagiert werden kann. Dies verhindert das Umgehen von Invarianten des Programms. Ein Objekt wird von einer Klasse instantiniert, die Attribute und Methoden beinhaltet. Eine Klasse kann aber auch andere Klassen als Attributen enthalten. Der Programmabhängigkeitsgraph muss also alle diese Eigenschaften berücksichtigen.

Wenn der Modellchecker Steam auf die Anweisung VASSERT trifft überprüft er, ob die angegebene Bedingung eingehalten ist. Von daher ist es naheliegend diese Bedingungen für die Auswahl der Variablen, nach enen abstrahiert wird, zu benutzen. Ein Program enthält unter Umständen mehrere VASSERT-Anweisungen. Beim Parsen wird eine Liste aller Properties erzeugt, für die dann alle Abhängigkeiten überprüft werden. Die Abhängigkeiten werden als Kanten im Graphen dargestellt. Wichtig für das Slicing sind vor allem die Kontrollabhängigkeitskanten und die Datenabhängigkeitskanten.

Kontrollabhängigkeiten entstehen, wenn Anweisungen die Ausführung von anderen Anweisungen kontrollieren (If-, While- u. a. Anweisungen). Datenabhängigkeiten entstehen zwischen Anweisungen, die Variablen definieren und Anweisungen, die diese Variablen verwenden.

Als Eingabe erhält der Algorithmus das Programm und die Property-Liste mit den Variablen aus den VASSERT-Anweisungen.

Listing 5.1: Beispielprogramm

```
class Computer
{
public:
        Computer();
        ~Computer();
        void doSomething();
        void setspeed ();
        void readspeed();
protected:
        int processorspeed;
        double d;
};
Computer::Computer()
{
        double h = d;
}
Computer::Computer2()
{
  processorspeed=4;
}
void Computer::doSomething()
{
        int a=3;
}
void Computer::setspeed ()
{
        processorspeed = 3;
        VASSERT (processorspeed == 3);
}
void Computer::readspeed()
{
        return processorspeed;
```

```
}
int main()
{
        Computer compute;
        Computer compute2;
        compute = compute2;
}
```

Als erstes werden die Knoten im Graphen, die die Variablen aus dem Vektor repräsentieren, herausgesucht, markiert und in ein Array gespeichert. Für jeden markierten Knoten wird überprüft, ob ein Pfad zu irgendeinem anderen Knoten aus dem Graphen über Kontroll-, Datenabhängigkeitskanten oder eine beliebige andere Abhängigkeitskante existiert. Falls das der Fall ist, wird der Knoten markiert. Dies wird so lange wiederholt, bis kein neuer Knoten mehr markiert wird.

In einem zweiten Durchlauf werden alle nicht markierte Knoten entfernt. Das geschieht in dem die einzelnen Lines der zugehörigen Objekte aus der Datenstruktur auskommentiert werden. Dadurch ändern sich die Zeilennummern im originalen Programm nicht.

Auf diese Weise haben wir den Zustandsraum des Programms reduziert. Das Programm muss dann in Steam-C++ zurückgeschrieben werden und anschliessend auf Fehler überprüft werden. Wird ein Fehlerpfad ausgegeben, so können wir, da wir ja die Zeilennummern nicht verändert haben, den Fehler im originalen Programm simulieren.

Listing 5.2: Das Program nach dem Slicing

```
class Computer
{
public :
Computer ( ) ;
~ Computer ( ) ;
void doSomething ( ) ;
void setspeed ( ) ;
void readspeed ( ) ;
protected :
int processorspeed ;
//double d ;
} ;
Computer :: Computer ( )
{
//double h = d ;
}
Computer :: Computer2 ( )
{
processorspeed = 4 ;
}
void Computer :: doSomething ( )
{
//int a = 3 ;
}
void Computer :: setspeed ( )
{
processorspeed = 3 ;
VASSERT ( processorspeed == 3 ) ;
```

```
}
void Computer :: readspeed ( )
{
return processorspeed ;
}
int main ( )
{
//Computer compute ;
//Computer compute2 ;
//compute = compute2 ;
}
```

5.5 Integration in Entwicklungsumgebung

Die Abstraktionsengine wurde als Plugin für Eclipse entwickelt. Für diese Entscheidung gab es mehrere Gründe. Auf der einen Seite ist es für den potentiellen ,,User" viel einfacher das Programm zu bedienen und zu installieren, auf der anderen Seite ist es für Entwickler viel einfacher aus einem Eclipse-Plugin die notwendigen Ressourcen im Dateisystem zu verwalten.

5.5.1 Benutzer-Sicht

Aus Sicht des ,,Users" ist die Eingabe von komplizierten Pfadangaben und Parametern zusätzlicher Arbeitsaufwand, der hier vermieden werden soll.

Eines der Ziele dieser PG bestand in der Vereinfachung der Bedienung des Modelcheckers Steam. Durch dieses Ziel motiviert wurde auch versucht die Bedienung der Abstraktionseinheit so simpel wie möglich zu gestalten. Unter diesen Umständen wurde der Begriff ,,One-Klick-Abstraktion" geprägt. Dieses Ziel wurde nahezu erreicht. Der Benutzer sollte in der Lage sein mit höchtens drei ,,Klicks" am Ziel zu sein, je nach Abstraktionsart.

Durch die ,,generische" Projektverwaltung von Eclipse ist der Zugriff auf alle notwendigen Dateien möglich ohne Eingaben des Benutzers ab zu fragen. Die einzige Interaktion die notwendig ist, um zu indentifizieren, welches Projekt er ,,bearbeiten" will, ist das Öffnen einer beliebigen Datei dieses Projektes im Eclipse-Editor.

Durch die Eclipse-Plugin-Architektur wird die Installation des Abstraktionsplugins auf ein Minimum an Aufwand reduziert. Alle notwendigen Klassenpfade sind dann schon gesetzt. Dies wäre sonst nur durch eine Installationssoftware möglich, die im Allgemeinen aufwändig zu erstellen ist, vor allem wenn man Plattform unabhängig arbeiten möchte.

5.5.2 Entwickler-Sicht

In der Regel besteht ein Eclipse-Projekt aus mehreren Dateien, die mit Hilfe von verschiedenen Funktionen des Eclipse-Frameworks leicht identifiziert werden können. So ist es dem Plugin möglich, schnell und effizient die notwendigen Dateien für die Abstraktion im Dateisystem zu finden.

Die Alternative wäre das Einlesen von Startparametern gewesen. Allerdings wäre dann das Auffinden der notwendigen Dateien nur schwierig realisierbar. Man m"ßte das Dateisystem durchsuchen und wissen ob man auf Linux, Unix oder Windows arbeitet, um den Pfadseperator korrekt zu benutzen. Eclipse bietet diese Funktionalität schon an. Trotz dieser Unterstützung durch Eclipse wurde bei der Entwicklung darauf geachtet so unabhängig wie möglich von Eclipse zu sein.

5.5.3 Aufbau des Abstraktionsplugins

Das Plugin besteht im Wesentlichen aus der Klasse Abstraction, die eine Steuerungsfunktion für das ganze Abstraktionsprogramm übernimmt. Sie ist die Klasse, die Eclipse bei Aktivierung des Plugins aufruft. Sie delegiert alle Aufrufe an Subprogramme und beinhaltet die Datenstrukturen während eines Abstraktionsdurchlaufs. Sie ruft zuerst den Parser auf, um die Dateien einzulesen und vorzubereiten. Danach führt sie die einzelnen Abstraktionsalgorithmen aus. Wenn der Quellcode erfolgreich abstrahiert wurde, sorgt Abstraction dafür, dass der ,,neue" Code kompiliert wird und an den Modelchecker übergeben wird. Sie soll das Ergebnis im Eclipse-Framework anzeigen.

5.6 Dataabstraktion

5.6.1 Vorwort

Für das Überprüfen realistischer Software sind Ansätze, den Zustandsraum möglichst klein zu halten unumgänglich. Der Ressourcenverbrauch und der Zeitaufwand für einen Modell - Check sind sonst einfach zu groß. Es sind mittlerweile schon viele Arbeiten entstanden, die verschiedene Methoden und ihre Anwendung auf Systeme behandeln, doch es fehlt an Tools, die es einem Entwickler ermöglichen, diese Zustandsraumverkleinerungstechniken für große Code-Segmente anzuwenden. In unserem Ansatz bauen wir auf dem C++ Model - Checker StEAM auf. Dieser Modell-Checker erlaubt es, von einem C++ Code einen Zustandsraum aufzubauen, durchzulaufen und das Programm auf vordefinierte Eigenschaften zu überprüfen. Mit Vorverarbeitung des Quellcodes versuchen wir für die Überprüfung der gewünschten Eigenschaften unnötige Details aus dem Programm zum Teil Benutzer-gesteuert zu entfernen. Dies wird mit Slicing und Data Type Abstraktion erreicht. Die Erweiterung kommt in Form eines Eclipse-Plugins, so dass auf diesem Wege das Model-Checking möglichst eng an den eigentlichen Entwicklungsprozess gebunden ist – die Benutzer können schon aus der C++ Entwicklungsumgebung Eclipse/CDT in wenigen Schritten Annahmen über Programmeigenschaften überprüfen. In diesem Kapitel wird Data Type Abstraktion behandelt.

5.6.2 Data Type Abstraktion

Für das Überprüfen von bestimmten Programmeigenschaften werden oft Datentypen mit einem großem Wertebereich nicht gebraucht. Sie bringen zu viel Informationen mit sich und lassen so auch den Zustandsraum wachsen. Im Sinne der Data Type Abstraktion werden diese Datentypen durch einfachere ersetzt, sofern dies das Überprüfen der Programmeigenschaften nicht behindert. Inwieweit die Datentypen vereinfacht werden dürfen, hängt von der zu prüfenden Eigenschaft ab: Ist z.B. eine Annahme über das Programm nur im begrenztem Maße von einer Integer Variable abhängig, so wird der Integer Wert durch einen abstrakten Datentyp ersetzt, der die möglichen Wertbelegungen der Variable drastisch reduziert. Auf diese Weise reduziert sich die Anzahl der möglichen Zustände des Programms [DHJ+01].

Zum Beispiel: Wir haben einen Code-Segment, der wie folgt aussieht:

Listing 5.3: Beispiel

```
int x;
x = 0;
...
if (x == 0) x = x + 1;
```

Für zu prüfende Eigenschaften des Programms ist die genaue Wertbelegung der Variable x nicht vom Belang. D.h. die möglichen Pfade, auf denen die Eigenschaft entweder tatsächlich im Programm existiert oder nicht erfüllt wird, werden nicht von einer geringfügigen Änderung der Variable x beeinflusst [DHJ+01].

Ein sinnvoller abstrakter Datentyp für die Variable x ist zum Beispiel folgender: x = NEG, ZERO, POS Die Wertebelegung ändert sich wie folgt:

Listing 5.4: Abbildung auf Werte eines abstrakten Datentyps

```
Integer x < 0 --> x = NEG;
Integer x = 0 --> x = ZERO;
Integer x > 0 --> x = POS;
```

Es gibt jetzt also nur drei mögliche Wertbelegungen der Variable x: $x = POS$, $x = ZERO$ und $x = NEG$. Im Vergleich zu der Anzahl der Zustände, die bei einer Integer Variable aufgebaut werden mussten, bedeutet diese Ersetzung eine drastische Verkleinerung des vom Model-Checker aufzubauenden Zustandsraums.

Da das Programm die Operatoren und Methoden verwendet, die mit neuen abstrakten Datentypen nicht umgehen können, müssen jetzt neue abstrakte Operatoren und Methoden eingeführt und die alten damit überladen werden.

Der auf Integer Werte anzuwendende + Operator wird jetzt durch einen abstrakten + Operator überladen, der auch mit Variablen vom Typ *Sign* umgehen kann. x und y sind Variablen, deren Datentyp durch *Sign* (wie oben beschrieben) ersetzt wurde.

Listing 5.5: Rechnen mit Sign

```
x = NEG, y = NEG —>  x + y = NEG;
x = NEG, y = ZERO —> x + y = NEG;
x = ZERO, y = ZERO —> x + y = ZERO;
x = ZERO, y = POS —> x + y = POS;
x = POS, y = POS —> x + y = POS;
x = NEG, y = POS —> x + y = {NEG, ZERO, POS};
```

Wir können annehmen, daß das Überprüfen eines abstrakten Programm wie eines konkreten die gleichen Ergebnisse liefert, da es gilt, daß ein Pfad des abstrakten Programms einem Pfad des konkreten identisch ist, wenn alle Entscheidungen auf diesem Pfad deterministisch sind [Sai00]. Im letzten Fall $x = NEG, y = POS - - > x + y = NEG, ZERO, POS$ kann man das Ergebnis der Addition von abstrakten Datenwerten nicht genau feststellen, da durch Ersetzung des Datentyps Informationen verloren gegangen sind. Dieser Fall muss beim Model – Checker als als eine nicht-deterministische Entscheidung behandelt werden. Es wird jedes mögliches Ergebnis des Programms beachtet. Da die Pfade eines abstrakten Programms mit einer nicht-deterministischen Entscheidung eine Obermenge über die Pfade eines konkreten bilden, impliziert die Korrektheit des abstrakten Programms die Korrektheit des konkreten Programms.

Die zu abstrahierende Variable und dazu passende abstrakte Interpretation dieser Variable sollte vom Benutzer sinnvoll gewählt werden. Die oben vorgestellte abstrakte Interpretation der Integer-Werte für eine Variable durch POS, $ZERO$ und NEG – ("SIGN") wäre zum Beispiel angemessen, wenn die Spezifikation der zu überprüfender Eigenschaften $x == 0$ enthält. Mit "SIGN" ist es dann immer noch möglich, die Gültigkeit dieses Ausdrucks zu überprüfen.

Über die Abbildung der Integer Variablen auf die positive, negative und gleich Null Bereiche können zahlreiche andere abstrakte Interpretationen definiert werden, die auch nicht nur auf Basis-Datentypen angewendet werden können. Es können zum Beispiel bei nicht-Basistypen Arrays oder Listen auf ihre Länge abstrahiert werden.

5.6.3 Umsetzung

Bei der Abstraktion und Überprüfen eines Programms kann man das Vorgehen in vier Schritte unterteilen:

1. Definieren der zu prüfenden Annahmen über Programmeigenschaften
2. Sinnvolle Auswahl der zu abstrahierenden Variablen und dazu passender abstrakter Interpretationen
3. Transformation des konkreten Programms in ein abstraktes
4. Überprüfen der Gültigkeit der Annahmen über das Programm mit dem Modell – Checker

Wie schon erwähnt – wurde die Abstraktion auf Basis eines Java - Eclipse-Plugins und das Model-Checking mit StEAM umgesetzt. Der Code des in Eclipse entwickelten Programms wird mit Hilfe des Entwicklers über das Plugin modifiziert / abstrahiert. Der abstrahierte Code wird schließlich mit StEAM auf vordefinierte Annahmen über die Programmeigenschaften geprüft.

Funktionsweise des Plugins:

Um die oben beschriebene Abstraktion auf einem Programmcode auszuführen, sind folgende Operationen notwendig:

- Alle Programmkomponenten für die Weiterverarbeitung identifizieren
- Dem Entwickler alle Programmvariablen und abstrakte Datentypen zur Auswahl anbieten
- Über die Datenabhängigkeiten im Programm feststellen, welche anderen Variablen von der Auswahl betroffen sind
- Ersetzen des Datentyps der Variablen durch den ausgewählten abstrakten Datentyp
- Zurückschreiben der Veränderungen

Für das aktuell in dem Editor Fenster in Eclipse bearbeitete Programm kann auf Wunsch durch Drücken eines DataAbs – Buttons das BugFinder Plugin gestartet werden. Intern passiert folgendes: Der Programmcode wird geparst und in eine für die Weiterverarbeitung bequemere Datenstruktur gefüllt. Mit den Informationen aus dieser Datenstruktur wird ein Datenabhängigkeitsgraph aufgespannt.

Mit Hilfe dieses Datenabhängigkeitsgraphen ist es nun möglich Abhängigkeiten zwischen den Variablen zu überprüfen: Bei dem Ausdruck x = y + z; ist die Variable x von y und z abhängig. Es existiert also auch eine Variablenabhängigkeitskante zwischen x und y und eine weitere zwischen x und z. Weiter bestehen Abhängigkeiten zum Beispiel bei Parameterübergaben und Methoden-Rückgabewerten. Möchte man bei einer bestimmten Variable den Typ durch einen abstrakten ersetzen, so muss man drauf achten, dass der Datentyp auch bei allen von ihr abhängigen Variablen verändert werden muss:

Wir haben zum Beispiel einen Quellcode, der wie folgt aussieht:

Listing 5.6: Zu abstrahierender Quelltext

```
class Test {
public:
    Test();
    Test1();
    int progVariable1;
    int progVariable2;
    int progVariable3;
    int result;
    int otherVariable;
};

Test::Test(){
        progVariable3=progVariable1+progVariable2;
        Test1(progVariable3);
}

Test::Test1(int parameter){
        int methodVariable;
        methodVariable=parameter;
        result = methodVariable;
}
```

Beim Starten des Abstraktionsplugins werden dem Entwickler alle Variablen aus dem Quelltext zur Auswahl bereitgestellt. (5.9)

Durch einen Doppel-Klick auf eine Variable wird die Liste der mitbeinflussten Variablen, deren Datentyp bei der aktuellen Auswahl der Variable auch geändert werden müsste, mitangezeigt. Auf diese Weise kann der Entwickler sofort die Auswirkungen der Änderung des Datentyps einer Variable in einen abstrakten sehen. Die abstrakte Datentypen können unten links im Fenster ausgewählt werden. Ist die Auswahl der Variable und des zugehörigen Datentyps getroffen, so werden die Änderungen durch ein Klicken auf den Button "Change" durchgeführt. Beim Schließen des Fensters werden sie in die Interface - Datenstruktur zurückgeschrieben.

In dem aktuellen Projekt in Eclipse wird eine neue C++ Quellcode Datei mit dem abstrahierten Code erstellt. Für das Beispiel aus 5.6 sieht der Code wie folgt aus:

Listing 5.7: Abstrahierter Quelltext

```
#include "Sign.h";

class Test {
public :
Test ( ) ;
Test1 ( ) ;
Sign progVariable1 ;
Sign progVariable2 ;
Sign progVariable3 ;
Sign result ;
int otherVariable ;
} ;
Test :: Test ( ) {
progVariable3 = progVariable1 + progVariable2 ;
Test1 ( progVariable3 ) ;
}
Test::Test1(Sign parameter){
Sign methodVariable ;
methodVariable = parameter ;
result = methodVariable ;
}
```

Oben im Quelltext aus 5.7 wurde der Code um den Befehl include Sign.h erweitert. Sign beinhaltet abstrakte Operatoren, durch die die Standard Operatoren überladen werden, wenn eine Ersetzung des Datentyps durch einen abstrakten durchgeführt wird.

Für den Operator "+" wird zum Beispiel in der "Sign" Bibliothek die Operation auf Werte vom abstrakten Datentyp "Sign" wie folgt definiert:

Listing 5.8: Rechnen mit "Sign"

```
Sign::Sign(int Eingabe)
{
if (Eingabe < 0)  Wert=NEG;
else if (Eingabe == 0)  Wert=ZERO;
else Wert=POS;
}

void Sign::setzeWert(int Eingabe)
{Wert=Eingabe;}
```

```
int Sign::gibWert()
{return Wert;}

Abstr Abstr::berechneWert(Abstr A, Abstr B)
{
int i;
if (A.gibWert()==NEG&&B.gibWert()==NEG) i=-1;
if (A.gibWert()==NEG&&B.gibWert()==ZERO) i=-1;
if (A.gibWert()==ZERO&&B.gibWert()==ZERO) i=0;
if (A.gibWert()==ZERO&&B.gibWert()==POS) i=1;
if (A.gibWert()==POS&&B.gibWert()==POS) i=1;
if (A.gibWert()==NEG&&B.gibWert()==POS) {RANGE(i, -1, 1);}
Abstr abs(i);
return abs;
}


Abstr operator+(Abstr A, Abstr B)
{
Abstr temp;
temp.setzeWert(temp.berechneWert(A,B));
return temp;
}

Sign operator+(Sign A, int B)
{
Sign temp;
Sign temp1(B);
temp.setzeWert(temp.berechneWert(A,temp1);
return temp;
}

Sign operator+(int A, Sign B)
{
Sign temp;
Sign temp1(A);
temp.setzeWert(temp.berechneWert(temp1,B));
return temp;
}
```

Der Nichtdeterminismus bei Operationen, bei denen es nicht klar ist, welcher Wert zurückgegeben werden muss, wird durch durch *StEAM* Anweisung "Range(i, -1, 1)" gesichert!

Wir haben ein Programm erhalten, in dem konkrete Datentypen durch abstrakte ersetzt wurden. Das abstrahierte Programm kann jetzt mit den formulierten Annahmen zusammen über das GUI - Plugin mit *igcc* kompiliert und mit StEAM überprüft werden.

Für das fernere Benutzen von dem Data Type Abstraktion Eclipse - Plugin ist es notwendig, dass die Bibliothek der Operatoren/Funktionen für abstrakte Datentypen gefüllt wird. Bei dem aktuellen Entwicklungsstand existieren in der Bibliothek nur wenige Operatoren für den einfachen abstrakten Datentyp *Sign*.

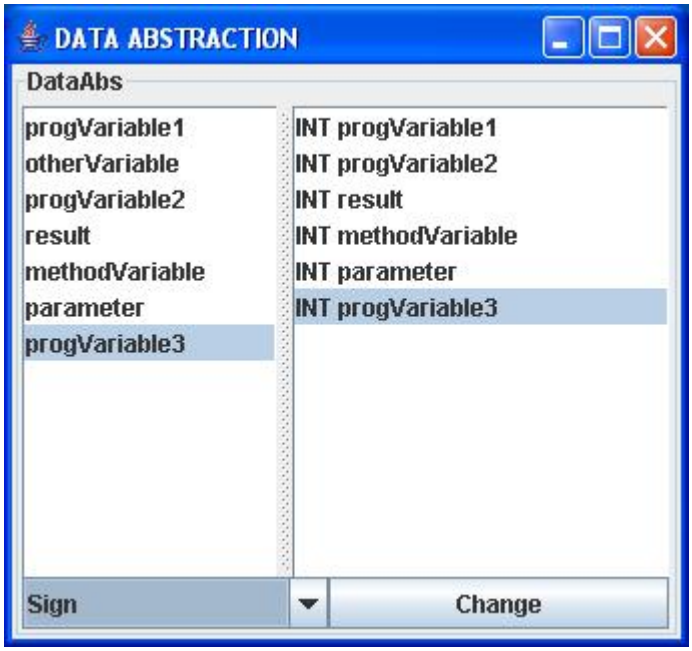

Abbildung 5.9: Auswahl der Variablen

Kapitel 6

Externalisierung

6.1 Problembeschreibung

In diesem Kapitel wollen wir eine weitere Möglichkeit vorstellen dem Problem der Zustandsexplosion zu begegnen.

Wir kennen einige Ansätze um den Zustandsraum bei einer Expandierung zu minimieren[CGP99]. Hierzu gehören verschiedene Such-Heuristiken[ELLL04] oder auch Techniken wie Hashen. All diese Ansätze verkleinern lediglich den Suchraum, sind die einzelnen Zustände sehr groß oder gibt es sehr viele von Ihnen, so expandieren wir bereits nach kurzer Suche einen Zustandsraum der nicht mehr in den Hauptspeicher passt. Eine Lösung für dieses Problem stellt die Externalisierung dar[SMS02]. Hierbei werden Zustände nicht im Hauptspeicher gehalten, sondern auf ein sekundäres Medium, meist sehr viel langsamer, aber auch um einiges größer, z.B. die Festplatte, ausgelagert. Besonders wichtige Leistungen auf diesem Gebiet sind [SD98, KM03].

Wir wollen natürlich nicht verschweigen, dass dies bereits durch das System eingesetzt wird. Verbraucht ein Programm mehr RAM als zur Verfügung steht wird ausgelagert, das nennt man *swapping*. Das System benutzt hierfür den so genannten *Virtuellen Speicher*. Dies ist eine spezielle Datei, oder sogar Partition auf der Festplatte in die das Betriebsystem bei Bedarf nicht benutzte Speicherbereiche speichert. Werden Informationen aus diesen Bereichen wieder benötigt so werden s.g. *Speicherseiten* in den RAM geladen, andere falls notwendig ausgelagert, und die Informationen zur Verfügung gestellt. Hierbei ergeben sich aber einige Nachteile, gegenüber einer direkten Externalisierung der Zustände. Erstens benutzt das System eine „standard“ Strategie um zu entscheiden was ausgelagert werden soll. Eine direkt auf das Problem zugeschnittene Externalisierung hat hingegen mehr Informationen und kann so besser entscheiden welche Zustände nicht benötigt werden. Zweitens kennt das System die Größe der Zustände nicht, es lagert immer konstant große Speicherseiten aus. Eine Methode die, die Größe eines Zustandes kennt, kann diesen komplett auslagern, und wieder in den Speicher laden.

Bei der strikten Externalisierung benutzt man den RAM ausschließlich zum generieren neuer Zustände und finden von Duplikaten. Dadurch das alle Informationen, sowohl die Struktur des Zustandsraumes als auch die vollständigen Zustände ausgelagert werden, wird konstant viel Speicherplatz verbraucht. Man lädt den benötigten Zustand von der Festplatte, expandiert ihn und schreibt dann die entstandenen Zustände wieder auf die Festplatte. Nach der Expandierung werden durch Sortierung Duplikate gefunden und entfernt (Vergl. [EJS04]). Aufgrund der sehr dynamischen Struktur der Zustände, aber auch der komplexen Speicherung in StEAM haben wir uns entschieden die Vorgaben an eine Externalisierung etwas zu relativieren. Wir belassen einige Informationen im RAM, wodurch dessen Verbrauch nicht mehr konstant ist. Allerdings ist die Menge der Informationen sehr gering, so können immer noch sehr große Probleme gelöst werden. Weiterhin ist die Größe dieser MiniStates konstant, im gegensatz zu Ihrem vollständigen Repräsentanten, wir sind also in der Lage die Einschränkung im vorraus zu bestimmen. So erreichen wir ein Maximum aus Speicherplatzgewinn

und Geschwindigkeitsverlust.

6.2 Ablauf

Die erste Aufgabe bestand darin eine Möglichkeit zu finden die Zustände überhaupt auszulagern. Dabei stießen wir auf mehrere Herausforderungen. Erstens wird in StEAM, um Speicherplatz zu sparen, eine Methode namens „Collapse Compression“[Hol04, LV01] verwendet. Zweitens musste eine Stelle im Code gefunden werden um die Schnittstelle zwischen Externalisierung und Expandierung zu schaffen. Weiterhin stellten sich die hohe Anzahl an Zugriffen und die Notwendigkeit, Beziehungen zwischen den Zuständen zu speichern als größere Hürden heraus.

Eines der größten Probleme ist die langsame Zugriffszeit bei externen Speichern. Die Fragmentierung eines Zustandes hat im RAM praktisch keine Bedeutung. Hier können wir in konstanter Zeit auf alle Speicherstellen zugreifen. Beim lesen von einer Festplatte ist aber genau dies das Problem. Jedesmal wenn der kontinuierliche Lesefluß unterbrochen wird muss der Schreib-Lesekopf an eine andere Stelle bewegt werden(es wird ein *seek* ausgeführt), dies kostet Zeit. Optimal wäre eine Externalisierung die keine seek Befehle benötigt. Nun der Leser kann sich sicherlich denken das dies nicht möglich ist.

Collapse Compression ist eine Methode um beim Expandieren großer Zustände Speicherplatz zu sparen. Hierbei „teilen“ sich einige Zustände die Sektionen die unter ihnen gleich sind. Bleibt z.B. beim expandieren eines Zustandes der Stack unverändert, wird dieser nicht neu für den Folgezustand erzeugt, sondern lediglich ein Vermerk auf den schon vorhandenen Stack gespeichert. Diese Art der Komprimierung hat sich als sehr effizient gezeigt, falls nur RAM Speicher genutzt werden soll. Möchte man allerdings externalisieren, so stößt man auf einige Probleme.

Wurde ein Zustand auf die Festplatte geschrieben, dürfen nur die Teile aus dem RAM gelöscht werden, die ausschließlich von diesem Zustand benutzt wurden. Würde man einfach alle Teile aus dem Speicher löschen, würde man einige andere Zustände zerstören. Bei der Lösung dieses Problems half uns weiter, das StEAM zu jedem Teil eines Zustandes eine Variable speichert, die die Anzahl der Zustände enthält welche diesen Teil benutzen. Wir löschen also nach einem erfolgreichen externalisieren nur die Teile dessen Anzahl Benutzer gleich 1 ist.

Ein weiteres Problem welches Collapse Compression mit sich bringt ist die Speicherung und Wiederherstellung der Zustände. Bei der Speicherung auf einem externen Medium wird viel mehr Raum benötigt als bei der Speicherung im RAM. Während im RAM ein Segment der von mehreren Zuständen benutzt wird, nur einmal vorhanden ist, speichern wir diesen Teil für jeden Zustand separat. Würden wir uns lediglich merken, wo der Teil auf dem langsameren Medium gespeichert liegt und zu dieser Stelle springen, müssten wir beim Lesen sehr viele seek Befehle in Kauf nehmen um einen Zustand zu rekonstruieren. Speichern wir den kompletten Zustand ab, brauchen wir zwar mehr Platz auf dem externen Medium, das Lesen in „einem Rutsch“ ist aber sehr viel effizienter, als ein „hin und her“ springen. Der höhere Speicherverbrauch stellt kein „Problem“ dar, da bei der externen Speicherung normalerweise genügend Platz vorhanden ist.

Jedoch lenkt uns diese Methode in ein weiteres Problem. Da wir nach dem Lesen nicht feststellen können ob wir einen Teil gelesen haben der bereits im Speicher vorhanden ist, erzeugen wir eine Kopie dieses Teils für den gerade zu rekonstruierenden Zustand. Diese Tatsache zeigt sich besonders dann von großem Nachteil, wenn viele Zustände direkt nach dem erzeugen gespeichert werden und sofort wieder gelesen werden müssen. Hierdurch wird der Vorteil der Collapse Compression praktisch zunichte gemacht.

Eine Stelle zu finden, an der man den Zustand löschen kann war bei StEAM relativ einfach. Aufgrund der weit vorangeschrittenen Implementation der „State reconstruction“ [1] konnten wir hier sehr gut ansetzen. Wurde ein Zustand expandiert, als neu erkannt und in die Hashtabelle eingefügt, so speichern und löschen wir diesen. Auch die Punkte zur Rekonstruktion eines Zustandes sind nicht allzu häufig. Ein vollständiger

[1] Hierbei werden Zustände direkt nach dem Expandieren gelöscht, und bei Bedarf wieder neu erstellt.

Zustand wird lediglich gebraucht um ihn zu expandieren, oder um ihn mit einem gerade generierten zu vergleichen. Dies wird nötig falls die Hashwerte der beiden identisch sind.

Um den Zugriff auf oft benötigte Zustände zu beschleunigen, entscheiden wir uns einen Cache zu benutzen. Hierbei werden Zustände nicht sofort ausgelagert, sondern verbleiben im Speicher. Die Anzahl dieser Zustände kann frei variiert werden, je nach vorhandenen Ressourcen. Wird nun ein Zustand benötigt, schaut man zuerst im Cache nach, ist er dort nicht zu finden, so wird er von dem externen Medium geladen. Der Cache beschleunigt aber auch die Speicherung. So können durch ihn mehrere Zustände auf einmal ausgelagert werden wir müssen nicht nach jedem neu generierten auf das langsame Medium zugreifen.

Aufgrund der hohen Komplexität bei der Speicherung der Beziehungen zwischen den Zuständen, aber auch um einen Weg zu finden, der für alle Suchmethoden möglichst optimal ist, entschieden wir uns die Struktur des Suchbaums im RAM zu belassen.

Die so entstandenen „MiniStates“ werden nicht externalisiert, sie haben eine konstante Größe und enthalten essentielle Informationen zu dem Zustand den sie repräsentieren. Diese Informationen sind: Der Speicherort auf dem externen Medium, ein Link auf den Eltern Zustand und einige wenige Daten mehr. Wir haben uns für diesen Weg entschieden, auch wenn dies theoretisch die Größe des Suchbaums einschränkt, da uns so mehr Möglichkeiten bleiben die Externalisierung an eine Suchmethode anzupassen.

Wie bereits erwähnt benutzen wir einen Cache als Zwischenspeicher. Es ist nun aufgrund der MiniStates allein von diesem Cache abhängig in welcher Reihenfolge die Zustände gespeichert werden. Hat der Cache seine Maximale Größe erreicht, so speichern wir einige Zustände auf die Festplatte. Würden wir nun die Zustände samt ihren Bezügen untereinander Speichern, so wäre eine Reihenfolge vorgegeben. Verzweigt ein Zustand a auf einen Zustand b so muss vor dem Speichern von a bekannt sein wo b auf der Festplatte liegt. Bleiben die Beziehungen aber im Speicher, so können wir beliebige, z.B. die *LeastRecentlyUsed* Zustände auslagern. Diese Tatsache nutzen wir um die verschiedenen Arten der Externalisierung zu verwirklichen.

6.3 Arten der Externalisierung

StEAM unterstützt zwei Arten der Externalisierung. Entstanden ist die zweite Art aufgrund der Weiterentwicklung der Modelle die wir testen, und aufgrund der Einsicht, das es sehr schwer sein wird eine Externalisierungsmethode für alle Modelle zu schaffen. Bei beiden Methoden werden einige Zustände im Speicher, dem o.g. Cache gehalten. Werden mehr Zustände erzeugt, als in diesen Cache passen, so werden die überschüssigen auf Festplatte ausgelagert. Sobald ein Zustand externalisiert werden soll, werden alle Zustände im Cache geprüft und, falls sie noch nicht gespeichert wurden, auf Festplatte geschrieben. Dieses Schreiben geschieht in zwei unterschiedlichen Methoden.

6.3.1 Single File Externalisierung

Bei dieser Methode wird der Augenmerk auf das Speichern gelegt. Um diese Phase so effizient wie möglich abzuarbeiten durchlaufen wir den kompletten Cache und schreiben alle, noch nicht gespeicherten, Zustände hintereinander in eine Datei. Taucht wider ein Zustand auf der gespeichert werden muss, so wird dieser, und natürlich alle anderen die sich im Cache befinden und noch nicht gespeichert wurden, hinten dran gehängt.

Die Vorteile liegen auf der Hand. Es muss nur eine Datei zum schreiben geöffnet werden. Diese kann über die komplette Prüfung hinweg offen bleiben da wir ja nur hinten Zustände anhängen. Sei M die Anzahl der Zustände die sich gleichzeitig im Cache befinden können, so sind wir in der Lage bis zu M Zustände mit nur einer Kopf Bewegung schreiben.

Allerdings bietet die Methode auch einige Nachteile. Werden Zustände benötigt die sich auf der Festplatte befinden, muss für jeden Zustand ein seek Befehl ausgeführt werden. Weiterhin können die Dateien sehr groß werden, was eine Aufteilung auf verschiedene Festplatten, zwecks Beschleunigung, erschwert.

6.3.2 Hash File Externalisierung

Im Gegensatz zur oben genannten Methode wird hier der Aspekt des Lesens bevorzugt. Die Zustände werden in mehren Dateien gespeichert. Die Zustände werden so sortiert, dass in einer Datei nur Zustände mit dem selben Hashwert zu finden sind. Um das Lesen noch einmal zu beschleunigen unterscheiden wir zwei Arten von Zugriffen. Wird nur ein Zustand gebraucht, z.B. um Ihn zu expandieren, so wird auch nur ein Zustand gelesen, dies braucht genau einen seek Befehl. Werden jedoch alle Zustände benötigt, die in einer Datei vorhanden sind, beim überprüfen einer Kollision z.B. , so wird die komplette Datei geladen. Übersteigt die Dateigröße allerdings den Raum im Cache so wird diese Datei stückweise geladen, wobei die Größe der Teile genau der Größe des Caches entspricht. Nachdem ein Teil auf Kollision mit dem zu untersuchenden Zustand geprüft wurde, wird der nächste geladen.

Die Vorteile liegen natürlich bei der kurzen Zugriffszeit auf alle Zustände mit dem selben Hash wert und der Möglichkeit durch Aufteilen der Dateien auf verschiedene Medien, mehr Platz nutzen zu können. Hier könnte man sich auch eine parallele Nutzung mehrerer Medien vorstellen, sei es um Speicherplatz zu gewinnen, sei es um die Zugriffszeit zu erhöhen.

Der größte Nachteil besteht bei der Notwendigkeit einer Sortierung beim Speichern. Stellt n die Kapazität des Caches da, so können bis zu n Datei Öffnungen, incl. seek Befehl, nötig sein um die n Zustände zu speichern.

6.3.3 External Collapse Compression

Zusätzlich zu den oben genannten Methoden haben wir uns entscheiden die „Collapse Compression“ auf die Externalisierung zu erweitern. Hierzu musste folgendes geändert werden. Beim Externalisieren von Zuständen speichert man den ganzen Zustand auf die Festplatte. Würde man diesen Zustand nun einfach zerlegen und nur die Teile schreiben die noch nicht gespeichert wurden, würde man zwar beim Speichern Zeit einsparen, aber beim Lesen des Zustandes diese durch unnötige seek Befehle wieder teuer bezahlen. Um diesen Nachteil auszugleichen muss man dafür sorgen das beim Rekonstruieren des Zustandes die Sektionen die an aderen Stellen der Festplatte liegen, schon im Speicher vorhanden sind. Dies wiederrum bedeutet aber das man einige Teile des Zustandes garnicht externalisieren sollte. Natürlich ist dieses „garnicht“ etwas relativiert anzusehen, wir sollten diese Teile halt äußerst selten auslagern.

Um diesen Ansatz zu realisieren haben wir uns die schon vorhandene interne „Collapse Compression“ Struktur zu nutze gemacht, hier wird der Zustand ja bereits Speicherintern zerlegt. Es wurden mehrere Caches angelegt, jeweils für eine Sektion einer, und mehrere Dateien auf der Festplatte. Wir speichern jetzt keine Zustände mehr, sondern Teile davon. Jeder Cache hat eine maximale Größe, die vom Benutzer natürlich an die vorhandenen Ressourcen angepasst werden kann. jetzt macht es aber nicht nur Sinn die Gesamtgröße der Zustände anzupassen, sondern auch das Problem zu analysieren. Hierdurch kann man nämlich dann die Größenrelationen zwischen den Cache's so festlegen das möglichst selten Teile ausgelagert werden müssen. Weitere Erkenntnisse zu dieser Art der Implementierung präsentieren wir in Kapitel 6.4.2

6.4 Experimente

Wie schon oben erwähnt stellt sich keine der Methoden als optimal für alle Modelle dar. Für einen Vergleich wurden zwei Modelle gewählt bei dem die Gegensätze besonders stark hervortreten.

6.4.1 Die Modelle

Als erstes Modell wurden die speisenden Philosophen gewählt. In diesem Modell befinden sich n Philosophen um einen Esstisch. Vor jedem Philosoph steht ein Teller Spagetti. Zwischen den Tellern liegen Gabeln, nötig

um die Speise zu sich zu nehmen. Jeder Philosoph benötigt sowohl seine rechte als, als auch seine linke Gabel um zu essen. Leider liegt zwischen den Tellern jeweils nur eine Gabel, es befinden sich also genau so viele Gabeln auf dem Tisch wie es Philosophen gibt. Die Philosophen wurden als Threads implementiert, die Gabeln als globale boolean Variablen.

Das Problem welches bei diesem Modell zu finden ist, ist ein Deadlock. Dieser tritt auf, wenn alle Philosophen eine Gabel an sich nehmen, denn in diesem Fall warten alle auf die zweite, die gerade von ihrem Nachbarn blockiert wird.

Beim zweiten Modell, welches zu prüfen war handelte es sich um das bekannte n-Puzzle. Bei diesem Puzzle steht uns ein $n * n$ großes Feld zur Verfügung, auf dem sich $n^2 - 1$ Plättchen befinden. Dadurch entsteht ein leeres Feld, hierher dürfen alle benachbarten Plättchen verschoben werden. Wir haben also bei jedem Zug die Wahl zwischen bis zu 4 Bewegungsrichtungen. „Bis zu„ 4 deshalb, da, wenn sich die freie Stelle am Rand befindet, natürlich nur 3 Bewegungen möglich sind.

Zur Implementierung des Problems wurde die `RANGE(x,y,z)` Anweisung benutzt. Dabei stellt x eine beliebige Integer Variable im Programm dar, y eine untere Grenze und z eine obere Grenze für diese Variable. Erkennt StEAM eine solche Anweisung, so generiert er $z - y$ Kinder vom aktuellen Zustand. In jedem dieser Kinder weist StEAM der Variable x einen anderen Wert zu. Bei dem n-Puzzle wurde $z - y = 4$ gewählt, und jeder Wert bedeutet eine Bewegung eines Plättchens. Eine Bewegung wurde als Tausch zweier Felder in einem Array realisiert. Die Instanz die zu finden war, ist 0123...n, wobei 0 das leere Feld darstellt.

Der Vollständigkeit halber erwähnen wir hier noch, das StEAM zwei Arten zur Verfügung stellt den Hashwert eines Zustandes zu berechnen. Einerseits *volles Hashing* andererseits *partielle Hashing*. Während beim vollen Hashing der Hashwert über den gesamten Zustand errechnet wird, so benutzt StEAM beim partiellen Hashing lediglich die Register der CPU um einen Hashwert zu berechnen.

Alle Experimente wurden auf einem System durchgeführt welches die folgenden Eigenschaften aufweist: 1,8 GHz AMD CPU, 512 MB RAM, 60 Gig IDE Festplatte mit 8 MB Cache. Der interne StEAM Cache fasste bei der Externalisierung 1024 Zustände. Bei allen Durchläufen wurde mit der Breitensuche gesucht. Beim erforschen des Philosophen Modells wurde das partielle Hashen eingesetzt, beim n-Puzzle waren wir gezwungen mit dem vollen Hashen zu arbeiten. StEAM ist aufgrund der internen Architektur in der Lage zu jedem Hashwert ca. 65000 Zustände zu speichern. Beim Anwenden des partiellen Hashen wurden aber ab einer bestimmten n-Puzzle Instanz mehr Zustände gleichen Hash wertes berechnet, dies zwang uns bei diesem Modell zum vollen Hashing.

Die leeren Felder in der Tabelle bedeuten, dass die Ressourcen, sei es Zeit oder Platz, zu Begrenzt waren um überhaupt eine Berechnung durchzuführen.

6.4.2 Ergebnisse der Experimente

In den ersten Experimenten die wir durchgeführt haben, haben wir das Philosophen Problem untersucht. Bei diesen Experimenten wurde ausschließlich die Single File Externalisierung eingesetzt. Wie man der Tabelle 6.1 entnehmen kann erhielten wir sehr gute Ergebnisse. Es sollte hier noch darauf hingewiesen werden das andere ModelChecker, wie z.B. VeriSoft[God05] lediglich in der Lage sind Probleme mit bis zu 10 Philosophen zu lösen. Hingegen schaffte es StEAM bereits ohne Modifikationen bis zu 100 Philosophen im internen Speicher zu lösen, war dann aber gezwungen die Festplatte zu nutzten was die Performance stark verschlechterte. Dies verlangsamte den Prozess natürlich erheblich, da die interne Swapping Funktionalität benutzt wurde. Beim Anwenden der Externalisierung sehen wir deutlich einen Performance Zuwachs an der um ca. 70% gesunkenen Rechenzeit. Natürlich sind wir so auch in der Lage größere Instanzen zu untersuchen.

Besonders bei diesem Problem machte sich die Implementation der in Kapitel 6.3.3 beschriebenen „External Collapse Compression“ bemerkbar. Obwohl diese Methode in erster Linie implementiert wurde um den Speicherplatz auf der Festplatte zu reduzieren, wirkte sich die Reduzierung dermassen stark aus, das sogar

Anz. Philos.	25	50	100	150	200	300
External	150 MB	157 MB	183 MB	256 MB	335 MB	545 MB
Collapse	163 MB	201 MB	566 MB	1256 MB		
Standard	162 MB	239 MB	897 MB			

Tabelle 6.1: Speisende Philosophen - Speicherverbrauch (RAM)

Anz. Philos.	50	100	150	200	250	300
External	8 Min	2,8 Std.	14,25 Std.	34 Std.	3 Tage	4,5 Tage
Collapse	8 Min	2,7 Std	22 Std	2,5 Tage		
Standard	10 Min	3,5 Std.	1,5 Tage			

Tabelle 6.2: Speisende Philosophen - Dauer zur Lösung

die Laufzeit signifikant beeinflusst wurde. Bei der Berechnung einer Lösung für das 6 Philosophen Problem mit Breitensuche konnte der externe Speicherverbrauch von ca. 8 GB auf ca 0,8 GB reduziert werden.

Kommen wir zum n-Puzzle. Auch hier sehen wir die positive Auswirkung der Externalisierung. Während der interne Speicher schon bei kleinen Instanzen aufgebraucht wird, können wir mit Hilfe der Externalisierung weitaus schwierigere Fälle lösen.

Wie wir bereits in Kapitel 6.3.2 berichtet haben, wurde auf Grundlage dieses Problems eine neue Art entwickelt, die Zustände auszulagern. Warum wurde dieses nötig? Untersucht man dieses Problem näher stellt man Fest das es einige signifikante Unterscheide zu dem vorherigen aufzeigt. Bei diesem Modell entstehen weitaus mehr Kollisionen als bei den Philosophen. Dies wird auch klar wenn man sich die Implementierung des n-Puzzles anschaut. An den Rändern werden die Plättchen zwar bewegt, die `RANGE` Anweisung wird ausgeführt, sie führt aber zu einem Zustand der gleich ist mit dem Ausgangszustand. Weiterhin entstehen schon allein durch die Art des Modells Kollisionen. Bewegt man ein Plättchen in die eine Richtung, dann im nächsten Zug wieder zurück entsteht eine Kollision.

Zusätzlich zu all den Kollisionen wird ein weiterer Unterschied aufgezeigt. Während bei dem Philosophen Problem sehr oft die Register der CPU benutzt werden, also auch eine hohe Streuung beim partiellen Hashing erreicht. Beim n-Puzzle Problem wird eher selten von den Registern gebrauch gemacht, die Variablen die das Puzzle repräsentieren liegen ja im Hauptspeicher, also erreicht das partielle Hashing auch eine sehr geringe Streuung. Dies führt abermals zu vielen Kollisionen.

Warum bilden aber diese Kollisionen ein so großes Problem? Kollisionen schwächen eine Externalisierung sehr stark. Im Falle einer Kollision müssen die kollidierenden Zustände in den Hauptspeicher geladen und verglichen werden. Verwendet man nun die erste Methode der Externalisierung so muss jeder Zustand mit einem seek Befehl geladen werden. Am Anfang, solange der Suchraum noch klein ist, passen die Zustände zwar in den Cache, aber sobald Zustände ausgelagert werden müssen, bricht die Performance ein. Dies führte zu der Überlegung alle benötigten Zustände mit einem seek Befehl einzulesen und zur Methode der „Hash File Externalisierung“. Weiterhin hilft hier auch eine Verbesserung der Hashfunktion weiter. Wendet man volles Hashing an so ist der Zeitunterschied mit oder ohne Externalisierung marginal.

Züge	15	16	17	18	19	21
External	292 MB	401 MB	420 MB	435 MB	456 MB	633 MB
Collapse	292 MB	403 MB	503 MB	641 MB	974 MB	2134 MB
Standard	552 MB	861 MB	1125 MB			

Tabelle 6.3: n-Puzzle - Speicherverbrauch (RAM)

Züge	15	16	17	18	19	21
External	1 Std.	1,8 Std.	2,5 Std.	3,5 Std.	5,1 Std.	14,6 Std.
Collapse	1 Std.	1,8 Std.	2,5 Std.	3,5 Std.	6 Std.	16 Std.
Standard	1,1 Std.	1,9 Std.	2,5 Std.			

Tabelle 6.4: n-Puzzle - Dauer zur Lösung

6.5 Ausblick

Wie bereits die zwei von uns untersuchten Probleme aufgezeigt haben gibt es keine optimale Externalisierung. Es bleibt also nicht nur die Aufgabe die hier vorgestellten Varianten weiter zu verbessern, sondern auch weitere zu entwickeln, die für andere Probleme von Nöten sein werden.

Bei der Anwendung der ersten Methode ist aufgefallen das der Speicherplatzverbrauch auf dem Externen Medium sehr groß ist, also Zeit sowohl fürs Schreiben als auch fürs Lesen verbraucht wird. Wir haben mit der „Externen Collapse Compression“ einen ersten Schritt in die Minimierung dieses Speichers getan, hier sollte aber weiter optimiert werden. Man könnte z.B. einen Zustand noch weiter zerlegen, und beim Vergleichen der Zustände die Hashwerte der einzelnen Teilbereiche zuerst vergleichen. Hierdurch würde man die Anzahl der Zugriffe auf das externe Medium weiter reduzieren.

Bei der zweiten Methode fällt vor allem ein Leistungsabfall auf sobald die Größe der einzelnen Dateien die Größe des Cache's übersteigt. Dies ist leicht zu erklären. Erreicht die Datei die zum Hashwert h gehört eine Größe die nicht mehr in den Cache passt, so wird diese in mehreren Etappen geladen. Wird nun ein neuer Zustand z mit Hashwert h erzeugt muss der Anfang der Datei geladen werden. Nachdem keine Kollision gefunden wurde, wird der Rest der Datei geladen, dabei werden natürlich einige der ersten Zustände entfernt. Passiert es nun das Zustand $z + 1$ auch Hashwert h hat, so laden wir die Datei wieder von neuem, usw. Abhilfe würde hier eine Möglichkeit schaffen den Hashwert des nächsten Zustandes schon im vorraus zu wissen, oder zumindes abzuschätzen. So könnten wir mehrere Zustände erzeugen und dann alle (auch untereinander) auf Kollision prüfen. Diese wäre eine Art frühe Delayed Duplicate Detection, es würden keine Doppelten Zustände im Speicher gehalten, aber es könnte vorkommen das gleiche, wohlgemerkt nicht die selben, Zustände im Speicher und auf der Festplatte nebeneinander existieren.

Natürlich bestehen bei diesem Thema noch viele andere Möglichkeiten die Effizienz zu steigern, diese würden aber den Rahmen dieser Arbeit sprengen und bleiben daher der Phantasie des Lesers überlassen.

Kapitel 7

Steigerung des Funktionsumfang

7.1 Installation

7.1.1 Alter Zustand

Zu Beginn der Projektgruppe haben wir lange gebraucht, bis wir StEAM überhaupt auf irgendeinem Linux-Rechner compilieren geschweige denn zum Laufen bringen konnten. Die Probleme waren dabei:

- Fehlende Bibliotheken bzw. uns unbekannte Abhängigkeiten, u.a. von libICE, libstd++.
- Vermischung des ursprünglichen ICVM-Quellcodes und des StEAM-Quellcodes.
- „Veralteter“ Quellcode. (Zu diesem Zeitpunkt war StEAM nur mit GCC 2.95.1 kompilierbar, der schon auf vielen Linux-Systemen nicht mehr zu den standardmäßig verfügbaren Paketen gehört)
- Benötigte Root-Rechte. (Das war besonders in den Rechner-Pools eine große Behinderung)
- Makefiles waren an die individuellen Bedürfnisse der früheren Programmierer angepasst, insbesondere an individuelle Verzeichnisnamen und Pfade der früheren Programmierer.
- Getrennte Binärdateien führten zu großen Schwierigkeiten beim Ausführen und Debuggen von StEAM.
- ...

7.1.2 Neuer Zustand

Nachdem die Probleme identifiziert und beseitigt wurden, haben wir auch dafür gesorgt, dass zukünftige Benutzer nicht über die gleichen Probleme stolpern werden.

Etliche vorhandene Shell-Scripts zur leichteren Entwicklung wurden aufgelöst und in das Makefile integriert. Es wurde außerdem ein frei wählbarer Installationspfad realisiert, und die Tools mfgen und extcml in den Kompilierungsprozess eingegliedert. Dabei wurden die Tools anhand einiger Beispielmodelle überprüft und kleinere Probleme beseitigt. Außerdem wurden die üblichen Dokumentationsdateien eingeführt (INSTALL-, README-, CHANGELOG-Dateien u.ä.)

Beim ICVM-gcc-Compiler und den ICVM-binutils wurden kleinere Fehler korrigiert, so dass die Kompilierbarkeit wenigstens auf GCC 2.95.4 garantiert werden konnte. Dafür wurden auch einige für StEAM nicht relevante Programmteile entfernt, z.B. die OpenGL-Unterstützung. Die Bibliotheken wurden soweit es ging erneuert und die Quellen so umgeordnet, dass das Packaging sowohl als Quellcode- als auch als Binärpakete für die wichtigsten Linux-Distributionen (Debian, Red Hat, SuSE) möglich wurde.

7.2 Dokumentation

Am Anfang hatte StEAM zwei englischsprachige Dokumente, User Manual und Technical Reference. Diese Dokumente wurden am Anfang in das Linuxdoc SGML Format konvertiert. Linuxdoc SGML ist die übliche Form zum Schreiben von Open-Source-Dokumentationen, und hat den Vorteil, dass es mit einem Befehl in alle gängige Formate (pdf, PostScript, RichText, Info, um nur einige zu nennen) konvertierbar ist. Danach wurde die Entwicklung von StEAM verfolgt und in diesen zwei Dokumenten beschrieben.

Das *User Manual* beschreibt die Installation und Nutzung von StEAM anhand eines Modells und alle Parameter und Befehle die StEAM versteht. Es wurde um das Kapitel "Installation von StEAM" erweitert. Auch die Änderungen bei den Programmeingaben wurden aktualisiert, sowohl die Systematisierung der Parameter wie die Einführung der Konfigurationsdateien. Zuletzt wurde die pThreads-Kompatibilität dokumentiert.

Das *Technical Reference* dient der Weiterentwicklung von StEAM. Sie wurde auch laufend aktualisiert und entspricht dem jetzigen Stand der Entwicklung.

Um die Herstellung der Dateien diverser Formate zu erleichtern, wurde ein kleines Skript und eine Readme-Datei geschrieben. Sie ermöglichen eine schnelle und automatische Erzeugung aller gängigen Datei-Formate aus der SGML-Datei.

7.3 Programmeingaben

7.3.1 Zusammenführung

Zu Beginn unserer Arbeit war die Funktionalität von StEAM auf zwei Binärdateien aufgeteilt: „steam“ und „st_icvm“.

„steam“ hatte dabei die Aufgabe die StEAM-, ICVM- und Modell-Parameter, die in der Konsole übergeben wurden, aufzuteilen und an das eigentliche Programm „st_icvm“ zu übergeben, Die StEAM-Parameter wurden mit Umgebungsvariablen übergeben, die Modelldatei und die Modellparameter direkt mit dem Aufruf von „st_icvm“.

Diese Aufteilung hatte den Nachteil, dass bestimmte Entwicklungs-Tools, z.B. „gdb“ und „valgrind“ nicht funktionierten. Diese Programme haben sich nur auf „steam“ bezogen und die eigentliche Funktion von StEAM nicht berücksichtigt. Außerdem war eine Umleitung der Konsolenausgabe nicht möglich: Die Ausgaben von „steam“ wurden zwar korrekt umgeleitet, die Ausgaben von „st_icvm“ jedoch nicht.

Wir haben die Funktionalität der beiden Programme in einer einzigen Binärdatei „steam“ zusammengefasst. Dafür waren zwar einige Eingriffe in den ICVM-Code nötig, aber dafür ist StEAM jetzt „aus einem Guss“.

7.3.2 Parameter

Am Anfang waren Parameter von StEAM unübersichtlich und nur spartanisch durch eine Hilfeatei dokumentiert. Zum größten Teil wurden Programmeingaben durch globalen Variablen gesteuert, was sehr schnell durch Unübersichtlichkeit zu Fehlinterpretationen führte.

Dieses Problem wurde durch die Erstellung einer neuen Klasse „config“ gelöst, die alle Parameter setzt. Es wurde eine neue Parsing-Funktion für Parameter geschrieben, die auch Parsing von Konfigurationsdateien ermöglicht. Die Funktion akzeptiert auch zweistellige Parameter, was eine bessere Strukturierung der Parameter ermöglicht. Beim Parsing werden auch die ungültigen Kombinationen von Parametern erkannt und mit einem Hinweis zur Lösung versehen.

Die Parameter selbst wurden in kurzer und langer Schreibweise strukturiert, so dass beide Angaben z.B. -a oder --algorithm möglich sind. Neben einer ausführlichen Hilfefunktion (--help) wurde auch eine man-Datei

erstellt.

7.3.3 Konfigurationsdateien

Neben den Inline-Parametern kennt StEAM auch Konfigurationsdateien. Es ist möglich, die Konfigurationsdateien für Rechner-, Benutzer- und Projekt-Ebenen zu erstellen. Die Dateien werden in dieser Reihenfolge eingelesen. Während des Parsing werden die Dateien ebenfalls syntaktische Korrektheit überprüft, und im Fall der Unkorrektheit wird auf die Fehlerquelle hingewiesen. Die Dateien werden während der Installation mit Standardoptionen belegt und sind ausreichend, in der für Unix/Linux-Systeme üblichen Weise, kommentiert.

7.4 Programmausgaben

7.4.1 XML-Bericht

Zusätzlich zur Ausgabe auf die Konsole kann auch ein Bericht in XML-Form ausgegeben werden, aus dem verschiedene Informationen, wie z.B. Laufzeit, Speicherbenutzung und Fehlerpfad wesenlich leichter automatisiert verarbeitet werden können. Dadurch wird es inbesondere auch einfacher für Dritt-Programme, über einen StEAM-Aufruf Informationen zu erhalten, ohne dabei eine Konsolenausgabe parsen zu müssen.

Listing 7.1: Beispiel eines von StEAM erzeugten XML-Berichtes

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl" href="steam-report.xsl"?>
<report steam-version="0.1" report-format="1.0.0.0">
  <statistics>
    <memory>
      <item name="max">139087872</item>
    </memory>
    <states>
      <item name="generated">33</item>
      <item name="level">14</item>
    </states>
    <timings>
      <item name="computation">41</item>
      <item name="difference">2</item>
      <item name="search">0</item>
    </timings>
    <hashtable bit-state="0">
      <item name="hash-code-count">31</item>
      <item name="collision-count">3</item>
      <item name="usage-count-most-frequent">2</item>
      <item name="usage-count-least-frequent">1</item>
    </hashtable>
  </statistics>
  <settings>
    <configuration>
      <item name="algorithm">DFS</item>
    </configuration>
  </settings>
```

```
  <error>
    <reason>Deadlock detected</reason>
    <trail>
      <step thread="1" file="philosophers.c" function="main" line="43" />
      <step thread="3" file="Philo.cc" function="Philo::run" line="26" />
      <step thread="3" file="Philo.cc" function="Philo::run" line="27" />
      <step thread="3" file="Philo.cc" function="Philo::run" line="28" />
      <step thread="3" file="Philo.cc" function="Philo::run" line="29" />
      <step thread="3" file="Philo.cc" function="Philo::run" line="30" />
      <step thread="3" file="Philo.cc" function="Philo::run" line="31" />
      <step thread="3" file="Philo.cc" function="Philo::run" line="27" />
      <step thread="2" file="Philo.cc" function="Philo::run" line="26" />
      <step thread="2" file="Philo.cc" function="Philo::run" line="27" />
      <step thread="3" file="Philo.cc" function="Philo::run" line="28" />
      <step thread="2" file="Philo.cc" function="Philo::run" line="28" />
      <step thread="1" file="philosophers.c" function="main" line="44" />
      <step thread="1" file="philosophers.c" function="main" line="44" />
    </trail>
  </error>
</report>
```

Die in diesem Bericht enthaltenen Funktionen sind noch nicht endgültig festgelegt. Es können jederzeit weitere Angaben hinzugefügt werden, auch die Struktur kann ohne großen Aufwand angepasst werden.

7.4.2 Log-Funktion

Bei unserer Arbeit an StEAM ist uns immer wieder aufgefallen, dass bestimmte Programmausgaben zwar sehr nützlich sind, aber für den Endbenutzer völlig uninteressant. Andererseits war es oft beim Debuggen so, dass man sich nur für die Ausgaben interessierte, die für den gerade bearbeiteten Bereich relevant sind.

Daraus wuchs der Wunsch nach einer etwas strukturierteren Programmausgabe im Stile einer Log-Datei mit Datum und Uhrzeit, Prioritäten (Hinweise, Warnungen, Fehler usw.) und verschiedenen Ausgabezielen (Konsole, Log-Datei)

Dabei gibt es jedoch ein Dilemma: Ein Fehler von StEAM und ein Fehler im Modell sind für den Benutzer völlig unterschiedliche Ereignisse. Deshalb ist das Log-System so aufgebaut, dass man einzelne Ausgaben nicht nur verschiedenen Subsystemen zuordnen kann, sondern bei Bedarf jedes Subsystem über die Art und Weise der Ausgabe selbstständig entscheiden kann.

Das Absetzen einer Log-Nachricht könnte z.B. so aussehen:

```
logger.print(
      LSS_EXTERNALISATION,
      LS_WARNING,
      "Only␣\%d\%\%␣hard␣disc␣capacity␣available\n",
      hdFree / hdCap * 100
     );
```

Bis auf die ersten zwei Parameter, die das Subsystem und die „Ernsthaftigkeit“ der Nachricht angeben, entspricht die print-Funktion der Standard-C-Funktion „printf“, d.h. offene Parameter sind wie gewohnt möglich. Der Programmierer kann einen Grenzwert festlegen, unter dem keine Meldungen mit einer bestimmten Priorität ausgegeben werden, und das für jedes Subsystem getrennt. So ist es z.B. möglich, Warnungen

der Externalisierung zu ignorieren, Warnungen des Modells jedoch anzuzeigen. Natürlich kann die Ausgabe ganzer Subsysteme auch komplett abgeschaltet werden. Eine genauere Beschreibung der zur Verfügung stehenden Funktionen befindet sich in der Quelltext-Dokumentation.

Das Log-System ist bisher nur zu einem geringen Teil für den Endbenutzer kontrollierbar. Er kann z.Z. nur ein „Verbosity-Level“ angeben, mit dem unwichtigere Ausgaben ausgeblendet werden können. In zukünftigen StEAM-Versionen werden dem Endbenutzer vielleicht weitere Einflussmöglichkeiten gegeben.

7.5 Fehlerbeseitigung

7.5.1 Compiler-Kompatibilität

Viele Stellen des StEAM-Codes wurden angepasst, um den restriktiveren Anforderungen der neuren Compiler gerecht zu werden. Das häufigste Problem war dabei Zeigerarithmetik, die seit GCC 3.3 vielfach nur unter Verwendung expliziter Typ-Umwandlungen erlaubt ist.

Um z.B. zu überprüfen, ob ein Zeiger innerhalb des durch zwei weitere Zeiger begrenzten Speicherbereiches liegt, musste der bestehende Code...

Listing 7.2: Zeigeraritmetik 1. Beispiel

```
void * lowerBound;
void * upperBound;
void * p;

[...]

if ( (p >= lowerBound) && (p <= upperBound) ) {
```

durch explizite Umwandlung in Integer-Zahlen folgendermaßen korrigiert werden:

Listing 7.3: Zeigeraritmetik 2. Beispiel

```
if ( ((unsigned int) p >= (unsigned int) lowerBound)
  && ((unsigned int) p <= (unsigned int) upperBound) ) {
```

Vielfach waren solche und ähnliche Stellen jedoch durch Makros und andere komplexere Strukturen versteckt, sodass man diese Probleme nicht durch ein einfaches Suchen und Ersetzen löseni konnte. Zumal oft dem Code nicht sofort anzusehen war, was gemeint war. Hierfür mussten teilweise ganze Code-Passagen anders formuliert werden.

Zweithäufigstes Problem war die Zuweisung von einem untypisierten an einen typbehafteten Zeiger. Während z.B.

```
char * x = malloc(...)
```

von GCC 2.95 kommentarlos akzeptiert wird, muss es bei GCC 3.4 so lauten:

```
char * x = (char *) malloc(...)
```

da malloc einen void-Zeiger zurückgibt, die Variable x jedoch ein char-Zeiger ist.

7.5.2 Speicherlecks und -fehler

Wie bei jedem großen Projekt haben sich auch in StEAM Speicherfehler in Form von nicht wieder freigegebenen Variablen sowie Zugriffen auf bereits freigegebene Variablen eingeschlichen. Alle erkennbaren Fehler

dieser Art wurden beseitigt, meist mit Hilfe des Tools „valgrind“, sodass StEAM weitaus solider im Sinne von „Speicher-Hygiene“ geworden ist.

Darüber hinaus wurde an vielen Stellen die Verwendung von Puffern mit fester Größe durch dynamisch berechnete und erzeugte Puffer ersetzt, um zukünftige Probleme im Zusammenhang mit überschriebenem Speicher zu vermeiden.

7.6 Ausblick

StEAM zukunftsicher zu machen, war iebenfalls eine Priorität der Projektgruppe.

7.6.1 SourceForge-Projekt

Es zeigte sich als sehr problematisch, StEAM auf den verschiedenen Platformen zu testen. Einerseits, war es schwer über diverse Hardwarekonfigurationen zu verfügen, andererseits ist StEAM sehr hardwarenah. Zweitens war es mangels Root-Rechte im Pool kaum möglich, die StEAM-Pakete für diverse Linux-Distributionen zu testen, was dazu führte, dass wir meist zu Hause entwickeln mussten. Deswegen brauchten wir eine Platform, auf der wir gemeinsam entwickeln konnten, und die gleichzeitig als Archiv für die Weiterentwicklung von StEAM dienen würde.

Die Entscheidung fiel auf die Open-Source Plattform SourceForge. Sie bietet CVS Server, Webspace, Bug-Tracking Tools, Mailing Listen und diverse andere Hilfsmittel, die uns den Einstieg erleichterten. So konnten wir immer einen Überblick über die Änderungen der anderen behalten, und wesentlich effizienter programmieren. Last but not least, durch die Freigabe des Codes überlassen wir das Testen und Debuggen der Open-Source-Gemeinde, so wie eventuelle Verbesserungen und Erweiterungen in der Zukunft.

Auf unserer Website kann man die aktuelle Dokumentation und die Linux-Pakete für die haufigsten Linux-Distributionen finden. Die Dokumentation umfasst User-Manual und Technical Reference auf Englisch, und eine Anzahl von HowTo's, die die Arbeit und Weiterentwicklung von StEAM erleichtern sollten. Die Dokumentation ist in allen gängigen Formaten verfügbar. Alle diese Dokumente finden Sie in unserem Appendix.

Die Linux-Pakete sind mit Dokumentation wie info, man oder help versehen, und entsprechen den Standards der Linux Distributionen, was die Datei-Organisation und Dokumentation betrifft.

7.6.2 Dokumentation des Quellcodes

Grundsätzliches Problem bei allen Erweiterungen des StEAM-Codes war zu erfahren, was eigentlich gemeint war, sprich: „Was sollte funktionieren, wenn es funktionieren würde?“.

Ein wichtiger Schritt, einen Großteil dieser Fragen bezüglich des Projektes zu klären, war die Erstellung einer detaillierten Code-Dokumentation. Dazu wurden der Reihe nach alle relevanten Funktionen, Methoden, Strukturen etc. auf ihre Funktionalität hin untersucht, in dem der Code „von Hand“ nachvollzogen wurde. Diese wurden dann direkt in den Code geschrieben, jedoch in einem Format, aus dem automatisiert mit Hilfe des Tools „Doxygen“[1] eine Gesamt-Dokumentation des Quelltextes in verschiedenen Formaten, u.a. HTML und LaTeX, erstellt werden kann.

Auch unsere Änderungen und Erweiterungen sind entsprechend dokumentiert, sodass sich der StEAM-Quelltext jetzt in dem seltenen Zustand einer umfassenden Kommentierung befindet. Diese Arbeit ist eine sinnvolle Ergänzung zur technische Dokumentation (S. 85) und wird v.a. späteren Entwicklern viel Arbeit und Zeit sparen.

[1] http://www.stack.nl/ dimitri/doxygen/

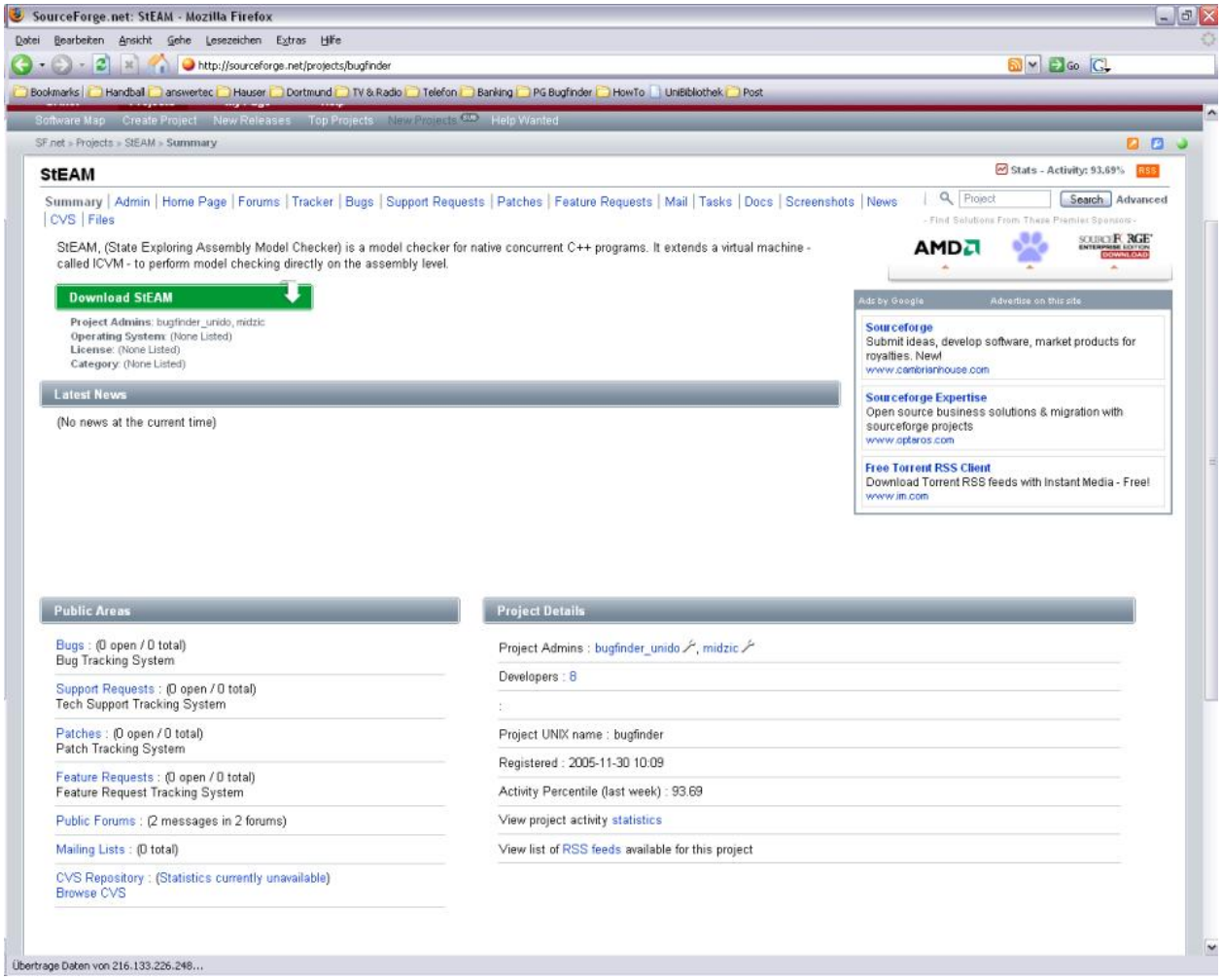

Abbildung 7.1: Bugfinder Sourceforge

7.6.3 Vorbereitungen für IGCC-Ersatz

Eine grundsätzliche Schwachstelle im StEAM ist seine Abhängigkeit von den ICVM-i*-Programmen (igcc, ias, iaddr2line, ...) Deshalb haben wir versucht, die Modelle mit dem normalen GCC zu erstellen, der Code für die Motorola-M68000-CPU kompiliert. Das hätte neben der Auflösung dieser Abhängigkeit auch die Möglichkeit eröffnet, StEAM zu den verschiedenen GCC-Frontends kompatibel zu machen, u.a. auch Java.

Leider gab es dabei einige Schwierigkeiten:

1. Das Erstellen eines GCC-Tools ist eine langwierige und komplizierte Sache. U.a. hat die Compile-Zeit von ca. 60 Minuten auf einem 1.800-MHz-Rechner dazu beigetragen, dass es mehrere Wochen dauerte, bis überhaupt ein funktionierender Motorola-GCC zur Verfügung stand.

2. Selbst wenn der neue GCC erstellt werden konnte, war das keine Garantie dafür, dass er auch funktioniert. Wir mussten feststellen, dass nicht alle beliebigen Kombinationen der beöntigten Teil-Quellen gcc, binutils und libc harmonieren.[2] Auch wenn die Beseitigung dieses Missstands recht schnell ging, dauerte es umso länger, da wir lange nicht herausgefunden haben, dass versionsabhängige Inkompatibilitäten die Ursache waren.

[2]Dieser Umstand ist den Entwicklern dieser Programme bekannt.

Abbildung 7.2: Bugfinder Homepage

3. Das ICVM-ELF-Format ist nur grob am ELF-Standard-Format orientiert, sodass dafür ein neuer Loader geschrieben werden musste.

4. igcc erzeugt Little-Endian-Dateien während der M68k-GCC standardmäßig Big-Endian-Dateien schreibt.[3] Ein Anpassen des GCC-Quellcodes ist zwar möglich, um die Daten im Little-Endian-Format zu erhalten, allerdings ist auf diese Weise unser angestrebtes Ziel, IGCC durch einen „Out-of-the-box“ -GCC zu ersetzen, natürlich nicht mehr zu halten.

Nachdem wir unsere Bestrebungen aus Zeitgründen aufgeben mussten, findet sich im StEAM-Code - natürlich neben unserem deutlichen Erkenntnisgewinn über die Interna von GCC - nur der halbfertige ELF-Loader sowie eine Zusammenfassung unserer Erkenntnisse, die für die zukünftige Weiterverfolgung interessant sind. Diese Zusammenfassung befindet sich in der Quellcode-Datei des neuen Loaders (icvmload_native.h)

[3] Anders als bei anderen GCC-Zielen, z.B. ARM, lässt sich das nicht einfach durch einen Parameter zur Laufzeit umschalten.

Kapitel 8

Appendix

8.1 User Manual

8.1.1 What's new

- v0.1 - by Dino Midzic - Document converted from TeX file, added links and some minor changes
- v0.1.1 - by Dino Midzic - Install section changed, new sources and packages explained
- v0.2 - by Dino Midzic - steam_0.2-1 has new parametars, minor bug fixies in linuxdoc solved, grammar check by Daniel Rikowski (thanks)

8.1.2 Introduction

StEAM, (State Exploring Assembly Model checker) is a model checker for native concurrent C++ programs.It extends a virtual machine - called ICVM - to perform model checking directly on the assembly level. This document gives you the basic information needed to install and use StEAM. The example programs included in the installation package are supposed to show the current capabilities of StEAM. However, we encourage you to try to verify your own programs. This manual will also explain the steps needed to prepare a piece of software for verification with StEAM. For detailed information about the internals of the model checker, we refer to Technical Reference of StEAM.

8.1.3 Installation

The installation instructions assume that StEAM will be installed on a Linux system using gcc. StEAM is based on the Internet Virtual Machine (IVM). The IVM package comes with source code for the compiler and the virtual machine (vm) itself. In the course of the StEAM project only the vm needed to be enhanced, while the compiler was left unchanged. The task of building igcc - the compiler of IVM, will hence be assigned to the user. The rest of the model checker, including the modified virtual machine can either be obtained as a binary package for Linux or as a source package. The source package can be compiled for use on Linux and Cygwin (a Linux-like environment for Windows).

Installing the compiler

The original source package of IVM can be downloaded from the http://bugfinder.sourceforge.net. The package was found to compile with older versions of gcc (e.g. 2.95.4). However, later versions such as 3.4.4

require considerable changes to the makefile in order to build igcc.

We have debuged a few minor bugs in igcc and rewritten some Makefiles to be able to make binary packages for igcc. At a present, you can find Debian and RPM Packages at http:/bugfinder.sourceforge.net. Feel free to test the packeges, considering they are in beta status at a time.

Installing the compiler from source

After unpacking source package, you will find 4 subdirectories: binutils, gcc, lib and include. First check if you have right gcc version on your machine. Output should be something like this:

```
gcc -v
Reading specs from /usr/lib/gcc-lib/i386-linux/2.95.4/specs
gcc version 2.95.4 20011002 (Debian prerelease)
```

Everything fine, active is gcc-2.95. If you have more as one gcc version installed, some other version could be active. Please set correct symbolic links:

```
ln -sf /usr/bin/gcc-2.95 /usr/bin/gcc
ln -sf /usr/bin/g++-2.95 /usr/bin/g++
```

Now you have to install binutils:

```
cd binutils
make
make install
```

After that, you have to install the igcc compiler itself:

```
cd ../gcc
make
make install
```

If everything went fine, you shold be able to compile something. In gcc directory is also the file test.c. There are no libraries installed yet, but you can test by typing

```
igcc -S test.c
```

You should be able to edit and view the machine language file test.s.

Igcc is a cross-compiler, which means it compiles on your machine binary code for steam virtual machine. To do so he need some libraries. Standard libraries are included in lib directory, otherwise you couldn't compile igcc at all. The last step we need to do is to make static libraries:

```
cd ../lib
make
make install
```

Done! Your igcc compiler is ready to use. If something went wrong, reffer to INSTALL and README sections in directories mentioned above, or mail us.

Installing the compiler from packages

There is not much to say here - you have to install first ivm-binutils and then ivm igcc package. Packages are build on i686, but shoul run on any Intel or AMD. You have some other processor - you can still compile the sources, configure file is able to check which processor you have.

Instalation of StEAM

Installing StEAM from source

After unpacking source package, you will find 3 subdirectories: docs, models and src. To compile StEAM, you'll need 3 libraries source packages: libX11, libICE and libxml2 from your favorite linux distribution. Makefile assume that they are in /usr/lib/, or you have to modify Makefile to set a proper path. After installing libraries, we can compile StEAM.

You should be able to compile StEAM with gcc or some other compiler, but it has not be tested enough. We had success with gcc 2.95, 3.3 and 3.4. We suggest gcc-3.4 with libc++6. First check if you have right gcc version on your machine. Output should be something like this:

```
gcc -v
...
gcc version 3.4.4 20050314 (prerelease) (Debian 3.4.3-13)
```

Everything fine, active is gcc-3.4. If you have more as one gcc version installed, some other version could be active. Please set correct symbolic links:

```
ln -sf /usr/bin/gcc-3.4 /usr/bin/gcc
ln -sf /usr/bin/g++-3.4 /usr/bin/g++
```

Now you have to install StEAM:

```
cd src
make
make install
```

If everything was fine, you should be able to start steam:

```
steam
Usage: steam <[-optargi]>  <progname> [<prog. arguments>]
       steam -h or --help for detailed information and examples.
```

Reffer to section 8.1.4 to learn how to use StEAM.

Installing StEAM from packages

After installing igcc packages and downloading steam package for your linux distribution, install it. Packages are i686-build and compiled with gcc-3.4. For other processors recompile from source could be nessesery.

8.1.4 A First Example

If you followed the instructions in section 8.1.3, you should now be able to check one of the example programs. If you obtained source package, you have in STEAM_VERSION/models small collections of test programs. Otherwise, obtain newest models archive from http://bugfinder.sourceforge.net/. After unpacking, change to models directory.

You can see here a few subdirectories. Go to the directory philosophers, which has C++ implementation of well known dining philosophers problem:

```
cd philosophers
```

There are two versions, with and without deadlock. Deadlock is intended error, to show function of StEAM. If you look at the content of Philosopher_err.cc (e.g. with `less Philospher_err.cc`), you will notice, that it basically does locks and unlocks two global variables which were passed to the constructor function. The file `philosophers.c` contains the main() method which generates and starts the threads.Let's see it now:

```
cd deadlock
ls

Philosopher_err.cc
Philosopher_err.h
philosophers.c
```

We have programed a mfgen, the tool that produces a Makefile automaticly. To include all present file in Makefile to be compiled, and compile it after, do

```
mfgen -f *
make
```

make will first apply annotations to the `.c` and `.cc` source files. After this, the example program is compiled and linked with `ig++` - the compiler of IVM. Do another directory listing and you will see the following:

```
Makefile
Philosopher_err.cc
Philosopher_err.cc_ext.c
Philosopher_err.h
Philosopher_err_ex.o
philosophers.ivm
philosophers_ex.o
philosophers.c
philosophers.c_ext.c
```

The source files containing an `ext.c ext.cc` in their name were produced by the source annotation tool extcml. The file `philosophers.ivm` is the IVM-object file compiled from the annotated source files. Start the model checker on the example program by typing:

```
steam philosophers.ivm 2
```

This will start the model checker with the default parameters on the compiled program `philosophers.ivm` parameterized with 2. In the case of the dining philosophers, this means that 2 instances of the thread class Philosopher are created. The typed command will produce the output of some standard information

from the model checker, followed by the report of a detected deadlock, an error trail and some statistics - such as the number of generated states, memory requirements etc. Each line of the error trail tells us the executed thread, the source file and line number of the executed instruction. Here, Thread 1 refers to the main program, while Thread 2 and Thread 3 correspond to the first and second instance of Philosopher. However, the error trail may seem a bit lengthy for such a small program. Now type:

```
steam -a bfs philosophers.ivm 2
```

This tells the model checker to use *breadth-first (BFS) search* (instead of *depth-first*, which is the default search algorithm). The error trail produced by BFS will be significantly shorter than the first one. You may want to compare the source files with the information given in the trail to track down the deadlock. Alternatively, you can add the parameter `--source-trail` to the command line. This prints the error trail as actual source lines - rather than just as line numbers. Apparently, a deadlock occurs if both Philosophers lock their first variable and wait for the respective other variable to be released.

8.1.5 Options

StEAM is invoked by a call:

```
steam <[-optargi]>  <progname> [<prog. arguments>]
```

The options are declared so:

- <`progname`> is the name of the program to be checked.
- <`prog. arguments`> are the parameters passed to the program to be checked
- <`[-optargi]`> are optional model checker parameters.

```
steam -h
```

Prints out the list of available parameters. The parameters come from one of the following categories.

Algorithms

In the current version, StEAM supports *depth-first search* (default), *breadth-first search* (`-a bfs`),*depth-first iterative deepening* (`-a idfs`) and the heuristic search algorithms *best-first* (`-a best`) and (`-a astar`). The latter two additionally require that a heuristic is given (see 8.1.5 (Heuristics)).

Also note, that IDFS is only effective in combination with bitstate-hashing (see below).

Heuristics

For detailed information about each heuristic, we refer to Technical Reference. The heuristic to use is specified by the parameter `-c or --heuristic` <*hname*>, where<*hname*> is from one of the follwing groups.

Deadlock heuristics

These heuristics are specifically tailored to accelerate the search for deadlocks, but will in general not be helpful for finding other errors, such as assertion violations. StEAM currently supports the deadlock heuristics: *mb, lnb, pl1, pl2, pb1, pb2, pb3, pb4* and *npb4*. For example:

```
steam -a best --heuristic lnb philosophers.ivm 30
```

Uses *best-first search* and the *lock and block* heuristic to model check the dining philosophers with 30 instances of the Philosopher class. The `best` parameter may have been omitted in this case, because StEAM uses *best-first* as the default algorithm when a heuristic is given. You may compare the results of the directed search with these of undirected search (type `steam philosophers.ivm 30` or `steam -a bfs philosophers.ivm 30`).

Structural Heuristics

Rather than searching for a specific kind of error, structural heuristics exploit properties of concurrent systems and the underlying programming language, such as the degree of thread interleaving and the number of global variable accesses. The currently supported structural heuristics are *aa, int* and *rw*.

Externalization

When the option `-e or --externalization` is given, StEAM keeps the states external on a hard disc, not in RAM. This option should be used in when the model does not fit in RAM any more. StEAM currently supports two arts of externalisation, *sf* or single-file, where maximum file size will be used for files on disc, and *hf*, where a hash function stores states in many smaller files. Options to be used only with externalization are:

File size

With option *–file-size x*, where x is file size in MB, you can set the maximum size of the file to be storen on harddisc. Default is 1900 MB.

File path

This option can change the path where files are to be stored, for example *–file-path /tmp/StEAM* is the default path.

State-cache size

To improve speed, you can cache certain amount of states in RAM. To set this, you should use *–state-cache-size x*, where x is amount of states. This value's maximum depends on your RAM size.

Hashing

By default, StEAM only hashes the cpu-registers of a program state `-p part`. Other options are:

Full Hashing

The option `-p full` instructs StEAM to hash the entire program state.

Incremental Hashing

Given the option `-p inc`, the entire state description is hashed in an incremental fashion, which is usually faster than `-p full` alone. Note, that `-p inc` implies full hashing.

Bitstate Hashing

When option `-p bit` is given, closed states are not completely stored but mapped to a position in a bit-vector. Bitstate hashing should always be used with full hashing, as this minimizes the chance that StEAM will miss states.

Illegal Memory Accesses

Given the option `-i or --illegal`, StEAM will explicitly look for illegal memory accesses (segmentation faults). As this may significantly slow down the exploration, it is not a default option.

Other Options

Time and Memory limit

You can set the maximum amount of memory in megabytes (`--memory-limit<`*mb*`>`), maximum amount of generated states (`--state-limit<`*n*`>`) and the maximum runtime in seconds (`--time-limit<`*s*`>`) required by the model checking run. When one these limits is exceeded, the model checker prints its final statistics and terminates.

Source Trail

When option `--source-trail` is given, the error trail is printed as actual source lines, rather than as line numbers. StEAM currently uses external Unix-commands to print single lines from source files. As a drawback, the printing of the trail cannot be interrupted using `ctrl-c`, as this terminates the external program rather than the model checker. This problem should be fixed in future versions.

No Trail

The option `--no-trail` suppresses the printing of the error trail - this is useful while doing experiments where the actual trail is of no interest.

8.1.6 Configuration files

As of StEAM Version 0.2-2 is possible to set Parametars in configuration files. There are 3 identical configuration files, and are stored in /etc/steam, /home/user/.steam and SOME_PROJECT_PATH/.steam. Initialy, only /etc/steam will be installed. It is well documented, and has a default values for StEAM. If you want to override this options for one user, you can copy this file in `/home/user/.steam`, and make modifications. This modification will not influence other users, and will make you starting von steam simpler.

Sometimes, we have option that has to be set only for one project. Such options can be set in your project path in file .steam. You can also copy the file from `/etc/steam`, and make settings you need.

In case of syntax error in file, a error line and file name will be shown, and StEAM will be stoped. If configuration file is not present, it will be ignored.

8.1.7 Verifying your own Programs

In this section, we give a small tutorial, how your own code can be verified with StEAM.

Motivation

The desire to verify software may in principle arise from two reasons. Either we already have some faulty implementation of a system and want to use model checking to detect the error; or we are just starting with our implementation and want to use model checking to avoid the generation of errors while our software is being developed. In this tutorial, we want to concentrate on the latter case. Using assembly model checking in the implementation phase yields several advantages. On the one hand, it may reduce or even replace the efforts spent on verifying properties of a formal description of the program. Skipping the formal modeling will not only save a considerable amount of development time, also there is no need for the developer to learn the underlying description language (e.g. Promela). Furthermore, assembly model checking is able to find implementation errors which were not present in the software's specification.

The concurrency in StEAM can also be used twofold. Either, the investigated software itself describes a concurrent system. In this case, we define the processes in the software in terms of threads in StEAM. Or, the software itself is not concurrent. In the latter case, we can use threads to simulate the environment of our system.

The Snack Vendor

As an example, lets us assume we want to develop the control software of snack vendor machine. As its basic functionalities, the machine should accept that coins are inserted to raise the credit and buttons are pressed to buy a certain snack. 8.1.7 (This code) shows the bare bones of the software formed by two minimal functions - `insertCoin()` and `pressButton()`.

```
/*********************************************
 * vendor.c
 *
 * The bare bones of a snack vendor machine
 *
 * written by Tilman Mehler
 *********************************************/

#include "vendor.h"
#include "icvm_verify.h"
#include "Inserter.h"
#include "Presser.h"
#include "Checker.h"

int credit=0;
int coin_values[3] = {10, 50, 100};
int price[3] = {50, 100, 200};
int sold=0;
int got=0;

void main() {

  BEGINATOMIC;
  (new Inserter())->start();
  (new Presser())->start();
  (new Checker())->start();
  ENDATOMIC;
}
void insertCoin(int amount) {
  if(credit+amount<=MAX_CREDIT) {
```

```
    credit+=amount;
    got+=amount;
  }
}
void pressButton(int n) {
  if(price[n]<=credit) {
    credit-=price[n];
    sold+=price[n];
  }
}
```

When a coin is being inserted, the credit is raised by the appropriate value. When a button is pressed, the machine first checks if there is sufficient credit to buy the snack. If this is the case, the price of the snack is subtracted from the credit. Two counters - `got` and `sold` - keep track of how much money was received and the total value of the snacks sold.

A first property we would like to check is that the total value of sold snacks is always less or equal to the amount of money received, i.e. we want to check for the invariant: `(got>=sold)`. Currently, StEAM supports the detection of deadlocks and local assertions. However, we are also able to check for invariants by introducing an observer thread. The source code of this observer is shown in 8.1.7 (This code)

```
/*************************************
 *
 * Checker.h
 *
 * Declarations for the Checker-class
 *************************************/

#ifndef CHECKER_H
#define CHECKER_H
#include "IVMThread.h"

class Checker:public IVMThread
{
 public:
  Checker();
  virtual void start();
  virtual void run();
  virtual void die();

};
#endif


/*************************************
 *
 * Checker.cc
 *
 * Definitions fot the Checker-class,
 * which makes an invariant-check
 *************************************/

#include "Checker.h"
#include "icvm_verify.h"

extern int credit;
```

```
extern int got;
extern int sold;

Checker::Checker() {}

void Checker::start() {
  run();
}
void Checker::run() {
  VASSERT(got>=sold);
}
void Checker::die() {}
```

As can be seen, a thread class in StEAM must be derived from the abstract class `IVMThread` and must implement the methods `start()` `run()` and `die()`. The `start()`-method makes preparations for the generated instance and must finally call the `run()`-method. The call to the `run()` method tells StEAM to insert the new thread into the running system. The `run()`-method itself must contain the actual loop of the thread. The `die()`-method is reserved for future versions of StEAM and currently has no effect - it must be implemented, but currently the body can be left empty. The statement `VASSERT(bool e)` is used to define local assertions. If the program reaches the control point of the statement and the expression `e` evaluates to *FALSE*, StEAM returns the path that leads to this state and reports an error.

Note that we do not have to put our assertion into an infinite loop. We can rely on searching only for those error states, where the observer thread is first executed when the assertion does not hold. Thus, we avoid the generation of equivalent paths to the same error state. Furthermore, we need to simulate the environment of our system - in this case, the users of the machine. One way to achieve this, is to introduce two derived thread classes - one that inserts coins and one that presses buttons. The header and source files for these are shown 8.1.7 (here). Both classes use the statement `RANGE(varname n, int min, int max)`, which causes a value between *min* and *max* to be non-deterministicly chosen as the value of the integer variable *n*.

```
/************************************
 * Inserter.h
 *
 * Declaratiins fot the coin-inserter
 * Thread-class
 ************************************/

#ifndef INSERTER_H
#define INSERTER_H
#include "IVMThread.h"

class Inserter:public IVMThread
{
 public:
  Inserter();
  virtual void start();
  virtual void run();
  virtual void die();
};
#endif



/************************************
 * Inserter.cc
```

```
 *
 * Definitions fot the coin-inserter
 * Thread-class
 ************************************/

#include "vendor.h"
#include "Inserter.h"
#include "icvm_verify.h"

extern int coin_values[3];

Inserter::Inserter() {}

void Inserter::start() {
  run();
}
void Inserter::run() {
  int i=0;
  while(1) {
    RANGE(i, 0, 3);
    insertCoin(coin_values[i]);
  }
}

void Inserter::die() {}
```

Although, in its current form, the source code can already be compiled to an IVM-executable, StEAM will not be able interpret the program correctly. First, the `.c` and `.cc` files need to be annotated with additional information. Fortunately, this remains mostly transparent to the user. The StEAM package comes with the tool `mfgen`. The latter takes as its parameter a set of source files and generates a Makefile which can be used to build the model-checkable file. In the directory where the sources for our vendor machine are located, simply type:

```
mfgen -f *
```

followed by

```
make
```

This will create the binary file `vendor`.

Finding, interpreting and fixing errors

Now, start the model checker by typing:

```
steam -a bfs vendor
```

After a (hopefully short) time, the model checker will print the error trail and report an assertion violation. Note that we have used *`breadth-first-search`* (`BFS`) to detect the error but the default algorithm is *`depth-first`* (`DFS`). In most cases, `DFS` find errors faster than BFS, but the BFS is guaranteed to return the shortest trail.

Take some time for interpreting the error trail before you read on ...

It should be obvious, what happened: The button was pressed immediately after the credit was increased, but before the coin value was added to `got`. We can fix the error by declaring the two variable increases as an atomic block in vendor.c, i.e.:

```
BEGINATOMIC;
credit+=amount;
got+=amount;
ENDATOMIC;
```

When you recompile the program (`make`) - and restart the checker, the error will no longer occur.

8.1.8 Special-Purpose Statements

The special-purpose statements in StEAM are used to define properties and to guide the search. In general, a special-purpose statements is allowed at any place in the code, where a normal `C/C++` statement would be valid. In Section 8.1.7 (Verifying your own Programs) we already used some of these statements for verifying the vendor machine. Now, we summarize statements, that are currently supported by StEAM.

BEGINATOMIC

```
BEGINATOMIC
```

This statement marks an atomic region. When such a statement is reached, the execution of the current thread will be continued until a consecutive `ENDATOMIC`. A `BEGINATOMIC` statement within an atomic block has no effect.

ENDATOMIC

```
ENDATOMIC
```

This statement marks the end of an atomic region. An `ENDATOMIC` outside an atomic region has no effect.

RANGE

```
RANGE (<varname>, int min, int max)
```

This statement defines a nondeterministic choice over a discrete range of numeric variable values. The parameter <*varname*> must denote a variable name that is valid in the current scope. The parameters *min* and *max* describe the upper and lower border of the value range. Internally, the presence of a `RANGE`-statement corresponds to expanding a state to have *max-min+1* successors. A `RANGE`-statement must not appear within an atomic region. Also, the statement currently only works for `int` variables.

VASSERT

```
VASSERT (bool e)
```

This statement defines a local property. When, during program execution, `VASSERT(e)` is encountered, StEAM checks if the corresponding system state satisfies expression *e*. If *e* is violated, the model checker provides the user with the trail leading to the error and terminates.

VLOCK

```
VLOCK (void * r)
```

A thread can request exclusive access to a resource by a `VLOCK`. This statement takes as its parameter a pointer to an arbitrary base type or structure. If the resource is already locked, the thread must interrupt its execution until the lock is released. If a locked resource is requested within an atomic region, the state of the executing thread is reset to the beginning of the region.

VUNLOCK

```
VUNLOCK (void * r)
```

Unlocks a resource making it accessible for other threads. The executing thread must be the holder of the lock. Otherwise this is reported as an access violation, and the error trail is returned.

8.1.9 Status Quo and Future of StEAM

The model checker StEAM is an attempt to realize software model checking without abstraction and the need to manually construct models in specialized languages, such as Promela. In the current form, the tool is already able to find errors in small programs. The large search space induced by real programs remains the main challenge of software model checking, which may inhibit the finding of an error. Therefore, one of the main issues of future development lies in finding good heuristics. As shown in previous work, an appropriate heuristic is able to find errors fast, even if the underlying search space is huge. In version 0.2 from StEAM a externalisation functionability was added, in order to make larger problems solved, though on a cost of speed.

Also we will improve the memory-efficiency of StEAM by optimizing its state representation. Future versions of StEAM will also support memory-efficient techniques such as bitstate hashing. In the long term our vision is to create a user-friendly application, which facilitates software- development and maintainance without the need for in-depth knowledge about model checking technology.

Main focus of the team is making StEAM accept programs which was compiled with gcc, so it would not have to use igcc any more. A plugin for eclipse environment should make checking with steam easier, and give a posibility to work with graphical interface.

8.2 Technical Reference

8.2.1 What's new

- v0.1 - by Dino Midzic - Document converted from TeX file, added links and some minor changes
- v0.2 - by Dino Midzic - Changes to StEAM 0.2 updated

8.2.2 IVM Source Files

The original package of *ivm* includes the virtual machine and a modified version of the gcc compiler *igcc*, which produces the ELF executables to be interpreted by *ivm*. As stated before, the approach of StEAM does not require changes to igcc, so only the source files for the virtual machine are of relevance. Moreover, only the files located in the sub folder `ivm/vm` need to be regarded. These include:

icvmvm.c

In the original package, this file contained the loop which reads and executes the program's opcodes. For StEAM, this was replaced by a loop, which expands program states according to the chosen algorithm. The original program execution is only done until the main-thread registers to the model checker. At this point, the contents of the stack, the machine, the global variables and the dynamic memory are memorized to build the initial state (i.e. the root state in the search tree).

The source file also holds the functions `getNextState`, `expandState` and `iterateThread`. The first function chooses the next open state to be expanded according to the used search algorithm. The second function expands a given state and adds the generated successors to the open list. The third function iterates a given thread one atomic step by executing the corresponding machine instructions.

icvmsup.c

This file implements the instructions of the virtual machine in a huge collection of mini-function. Due to its size of more than 140000 lines, it is compiled to 14 separate object files (icvmsup.o, icvmsup1.o, ... ,icvmsup13.o) before linking. Compiling icvmsup takes quite long (several minutes on a 1.8GHz machine). This is quite inconvenient in cases, where a `make clean` needs to be done (e.g. when a constants in a header file were changed), as the original makefile will also delete the icvmsup object files, even if the machine instructions are not affected by the change. In this case it may be convenient to move the object icvmsup object file to a temporary folder before cleaning up and moving them back before the subsequent `make all`.

In order to do so, you can type `make update` instead of `make clean; make all`. This will do it automaticly.

The source file icvmsup.c was not modified during the development of StEAM. However, the file is useful to find out about the meaning of a certain instruction. In icvmsup.c, an instruction can be located through its corresponding opcode. For example, if we disassemble the dining philosophers from 8.1 (User Manual) with

```
iobjdump -d philosophers | less
```

we find a fraction of machine code:

.
.

```
                .

20:       0200 7c2c 0000  bsrl 2c9c <___do_global_ctors>
26:       a8eb 0800       movel (8,%r2),-(%sp)
2a:       a8eb 0400       movel (4,%r2),-(%sp)

                .
                .
                .
```

We may want to find out about the instruction `brsl <x>` (branch sub routine long). Note that `iobjdump` displays opcodes according to the hi/low order of the underlying machine. In our case, we have an Intel-processor with Little-Endian notation, which means that the corresponding opcode is `0x2` rather than `0x200`. In *icvsump.c* we find a line starting with:

```
_If1(_sCz2,2,s32) ...
```

The instructions for *ivm* are generated through a macro, whose first parameter is the opcode with a leading `\_sCz`. Note that the opcode is appended without any leading zeros. We can easily interpret the implementation of the instruction, even without knowing how the macro translates to compilable c-code:

```
{{(R_SP-=8,WDs32(R_SP,R_PC+6,0));R_PC=(is32(2)+R_PC);}
```

First, the stack-pointer (R_PC) is decremented to reserve space for the return-address of the called sub routine. Then, that address is stored at the new position of stack-pointer through the macro `WDs32` (write signed 32-bit data). The return address corresponds to the current program counter plus 6 - the length of the bsrl instruction. Afterwards, the program counter is set to the the current position plus a 32-bit offset that follows the opcode. Apparently, the macro `is32(y)` reads a 32-bit parameter from address of the program counter plus y.

cvm.h

This header defines the basic data types needed for the virtual machine, such as the machine registers. The macros used by the mini-functions in icvmsup.c - such as `WDs32` - are also defined here. As a big convenience, all memory read- and write-accesses are encapsulated through the macros of this header. This enabled us to record all such accesses by simply enhancing the macros. On the one hand, this allow us to determine changes made by a state transition without fully comparing the pre- and successor state. On the other hand, we can capture illegal memory accesses before they actually happen.

8.2.3 Additional Source files

The following source files were added for StEAM.

avltree.h|.c queue.h|.c

The name is somewhat misleading, as this folder also contains `.c` files. More precisely, it contains the AVL-Tree package used for the lock- and memory pool of states in StEAM. Using the AVL trees may not have been the best choice for storing the pools, as their handling is tedious and a simple hash table may be faster in many cases. Developers are hereby encouraged to replace the AVL trees with a more suitable data structure.

IVMThread.h|.cc

This is the class definition for threads in StEAM. All new thread classes must be derived from IVMThread. A thread registers to the model checker by executing a TREG command-pattern in its start-method. A second TREG statement tells StEAM, it has reached the `run()`-method of the thread.

mfgen.c

This is the source code of the tool, which is parameterized with the names of all involved source files of the checked program. In turn generates a makefile for building the *ivm*-executable that can be model checked by StEAM.

extcmdlines.cpp

This is the source code annotation tool. Before the investigated program is compiled, its source is annotated with line number information and information about the name of the source file through command patterns. The annotation is automatically done in the makefile generated by mfgen and, hence, remains transparent to the user. It must be assured though, that extcmdlines.cpp is compiled to an executable called extcml. It must be in the system path.

The source annotation tool is actually a workaround, as StEAM does currently not use the debug information that can be compiled into ELF files (e.g. using the -g option on gcc). Writing a parser for this information would have cost too much time considering the manpower available for the project. Developers are encouraged to add this functionality to StEAM, as it would make the annotation redundant and speed up the entire model checking process.

pool.h|.c

These files define the functions needed for the lock- and memory-pool of StEAM's state description.

icvm_verify.h|.c

These source files are the core of StEAM as they define the command patterns, data structures and functions needed by the model checker. A command pattern is generated by the macro *INCDECPATTERN*, which translates into a senseless sequence of increment and decrement instructions. The parameters of a pattern are realized by assigning their value to a local variable. After parsing the command pattern, StEAM can obtain the values of the parameters directly from the stack.

Most functions defined in icvm_verify.c relate to program states. This includes, cloning of a state, comparison of two states, deletion of states etc. Moreover, icvm_verify.c implements the functions of StEAM's internal memory management. The management is encapsulated through the functions `gmalloc(int s)` and `gpmalloc(int s, char * p)`, which allocate memory of a given size or of a given size plus a given purpose. When StEAM is invoked with the parameter `-memguard`, all memory regions allocated with `gmalloc` or `gpmalloc` are stored in a private memory pool. NOTE: This refers to the memory pool of the model checker and must not be confused with that of the investigated program. It is possible to print the contents of the memory pool using the function `dumpMemoryPool()`.

StEAM's internal memory management is useful to e.g. detect memory leaks in the model checker. For instance, this was important during the implementation of state-reconstruction. Here, states are frequently deleted after being replaced by a corresponding mini-state. If a few bytes of the state are not freed, the resulting leaks will quickly exceed the available memory.

The source file icvm_verify.c also implements the currently supported heuristics through the function `getHeuristicEstima`
The latter function is called by `getNextState()` in `icvmvm.c`, which chooses the state to be expanded next according to its heuristic value. Note that for simplicity undirected search is also implemented through heuristic values. For BFS, the heuristic estimate of a state corresponds to its depth in the search tree. For DFS the estimate is determined by subtracting the depth from a constant which is greater than any search depth we may ever encounter.

steam.c

This is the frontend of the model checker, which calls the executable of the modified virtual machine. Environment variables are used to pass the parameters.

hash.h|.c

These files implement the (non-)incremental hashing in StEAM. The hash function is parameterized with a bitmask, which determines the hashed components of the state description. The frontend of steam currently only gives two choices: Either only the cpu-registers are hashed (default) or the entire state (`-fullhash`). The cpu registers were chosen for the partial hash option, since they constitute the state component that is most likely to be changed by a state transition. Developers are encouraged to try other subsets of components.

8.2.4 Further Notes

Changes in icvmvm

The functions `getNextState`, `expandState`, `iterateThread` and `printTrail` actually do not belong in the file `icvmvm.c` and it would be a good idea to move them to `icvm_verify.c`.

When StEAM is started with the `-scrtrl` option, the error trail is printed as the actual source lines of the checked program (rather than just the line numbers). For this, StEAM makes use of the Unix-commands *cat* and *grep*. Hence, the option will not work on systems where these commands are not available. It would be desirable to rewrite the `printTrail` function such that it no longer requires any external programs.

As a definite drawback, states must currently be fully expanded. The problem is, that it is not a-priori known, if during thread iteration a non-deterministic statement is encountered. In the latter case, execution of a thread yields more than one successor. For this reason state reconstruction is currently not available for programs involving non-deterministic statements. Also the full expansion is disadvantageous for depth-first variation, such as DFS and IDA*, since at depth d the model checker needs to memorize d o states instead of d - where o is the outgoing degree of a node in the search tree.

Hence, a big improvement would be to add the possibility to directly generate the i-th successor of a state.

8.3 Entwicklung von StEAM unter Eclipse

Die folgende Anleitung bezieht sich auf die Entwicklung von StEAM selbst, nicht auf die Entwicklung anderer Programme, die mit StEAM getestet werden sollen!

Eventuelle Nennungen von Fenstertiteln, Beschriftungen, Schaltflächen u.ä. beziehen sich auf Eclipse 3.1 in Kombination mit CDT 3.0.2. Zum Zeitpunkt des Schreibens ist auch bereits die Nutzung des Release Candidates 3.2 von Eclipse mit dieser Anleitung möglich gewesen.

1. **Eclipse installieren**

   (a) Laden Sie Eclipse 3.1 oder 3.2 herunter und entpacken bzw. installieren Sie es.[1]

   (b) Sofern auf Ihrem System kein Java Runtime Environment verfügbar ist, laden Sie sich auch dieses herunter[2] und installieren Sie es ebenfalls.

2. **CDT installieren**

   (a) Starten Sie Eclipse.

   (b) Installieren Sie nun die *C/C++ Development Tools* (CDT) unter *Help/Software Updates/Find and Install*.

   (c) Wählen Sie die *Option Search for new features to install*.

   (d) Klicken Sie auf *New remote site* und geben Sie danach einen beliebigen Titel („CDT“ ist eine gute Wahl) und die CDT-URL[3] ein und schließen Sie dann das Fenster.

   (e) Stellen Sie sicher, dass der dadurch neu angelegte Eintrag in der Liste *Sites to include in search* markiert ist und klicken Sie dann auf *Finish*.

   (f) Wählen Sie dann in der Liste *Select the features to install* den Eintrag *Eclipse C/C++ Development Tools 3.0.2* aus und folgen Sie den weiteren Anweisungen von Eclipse.

   (g) Nachdem die Installation abgeschlossen ist, müssen Sie Eclipse neu starten, um CDT benutzen zu können.

3. **Das StEAM-Projekt anlegen**

   (a) Starten Sie Eclipse und wechslen Sie auf die *C/C++*-Perspektive.

   (b) Legen Sie im Navigator ein neues Projekt vom Typ *CVS/Projects from CVS* an.

   (c) Wählen Sie *Create a new repository location* und füllen Sie das darauf erscheinende Fenster gemäß den folgenden Angaben aus:

   - Host: bugfinder.cvs.sourceforge.net
   - Repository path: /cvsroot/bugfinder

   Falls Sie ein SourceForge-Konto besitzen:

   - User: Ihr SourceForge-Anmeldename
   - Password: Ihr Kennwort oder frei lassen, wenn Sie Ihr Kennwort nicht speichern wollen.
   - Connection type: extssh

   Falls Sie kein SourceForge-Konto besitzen:

   - User: anonymous
   - Password: Nichts eingeben
   - Connection type: pserver

[1] http://www.eclipse.org/downloads/
[2] http://java.sun.com/
[3] http://download.eclipse.org/tools/cdt/releases/eclipse3.1

(d) Bestätigen Sie das Fenster und wählen Sie danach das CVS-Modul aus. Lassen Sie sich dazu die vorhandenen Module mit der Option *Use an existing module* anzeigen und wählen Sie das gewünschte Modul aus. (Als diese Anleitung verfasst wurde, war das aktuelle Modul `steam_0.2`).

(e) Eclipse fragt danach, auf welche Art das neue Projekt angelegt werden soll. Wählen Sie da die Option *Check out as a project configured using the New Project Wizard*.

(f) Wenn Eclipse Sie nach einem CVS-Tag fragt, übergehen Sie das Fenster einfach.

(g) Wählen Sie als Projekttyp *C++/Standard Make C++ Project* aus. Verwenden Sie auf keinen Fall eines der *Managed Make*-Projekte!

(h) Geben Sie danach den Projektnamen ein und - falls gewünscht - verändern Sie den Projektpfad. Der CVS-Modulname ist eine gute Wahl für den Projektnamen. (z.B. `steam_0.2`)

(i) Dann fragt Sie Eclipse nach den Einstellungen für das neue C++-Projekt. Die wichtigsten Angaben sind hierbei:

- Wenn Ihre Systemsprache eine andere als Englisch ist, legen Sie unter *Environment* eine neue Variable mit dem Namen *lang* und dem Wert *en* an. Andernfalls zeigt GCC Warnungen und Fehlermeldungen in Deutsch an, die Eclipse nicht verarbeiten kann.
- Wählen Sie unter *C/C++ Indexer* entweder *CTags Indexer* (dazu muss CTags auf Ihrem System installiert sein) oder *No Indexer*. Im StEAM-Quellcode befindet sich eine extrem große C-Datei, mit der der vollständige Indexer nicht zurechtkommt. (Wenn Sie einen schnellen Rechner mit viel Arbeitsspeicher besitzen, können Sie es jedoch zu einem späteren Zeitpunkt versuchen)

(j) TODO: Fenster schließen

4. **StEAM kompilieren und ausführen**

(a) Stellen Sie sicher, dass das StEAM-Projekt das einzige geöffnete ist und die C/C++-Perspektive aktiv ist.

(b) Zeigen Sie sich das Fenster "Make Targets" an. (*Window/Show View/Make Targets*) Dieses Fenster zeigt Ihnen die Verzeichnisstruktur des StEAM-Projektes an, in der Sie für jedes Verzeichnis mehrere Make-Ziele anlegen können. Wenn Sie keine Ordner darin sehen können, müssen Sie evtl. die Option *Hide empty folders* in der Menüleiste des Fenster deaktivieren.

(c) Wechseln Sie im Make-Targets-Fenster in das Verzeichnis *src* und legen Sie die folgenden Make-Ziele an: (Kontextmenü des Verzeichnis, *Add Make Target*)

- add
- clean
- update (Funktionsweise s. Technical Reference)

(d) Wenn Sie mit Root-Rechten arbeiten, können Sie auch die folgenden Ziele anlegen:

- install
- uninstall

Andernfalls müssen Sie diese Ziele bei Bedarf in einer Root-Shell ausführen.

(e) Kompilieren Sie StEAM, indem Sie das all-Target ausführen. Je nach Rechnerausstattung kann das etliche Minuten dauern. 10-15 Minuten sind eine plausible Annahme.

(f) Warten Sie, bis der Kompilierungsvorgang abgeschlossen ist.

(g) Installieren Sie danach StEAM, entweder über das entsprechend angelegte Make-Ziel oder über eine Root-Shell. Außerdem sollten Sie sich ein Modell kompilieren, um den weiteren Verlauf des HowTos folgen zu können. (Das Dining-Philosophers-Modell ist für den Anfang gut geeignet)

(h) Legen Sie als nächstes eine neue Run-Konfiguration für StEAM an, indem Sie das Fenster zum Bearbeiten der "Run Configurations" öffnen. (z.B. mit *Run/Run...*)

(i) Legen Sie unter *Local C/C++ Application* eine neue Konfiguration an und nehmen Sie folgende Einstellungen vor:

i. Unter *Main*:

- **Project:** Das oben angelegte StEAM-Projekt
- **C/C++ Application:** "steam". Wenn diese Datei nicht zur Auswahl angeboten wird, ist StEAM nicht erfolgreich kompiliert worden.

ii. Unter *Arguments*:

- **C/C++ Program Arguments:** Den Dateinamen eines zuvor erstellten Modells und seine entsprechenden Parameter. z.B. "philosophers 2"
- **Working directory:** Das Verzeichnis, in dem das Modell liegt. (Das ist wichtig, damit StEAM die Quelldateien des Modells finden kann)

iii. Unter *Debugger*:

- **Debugger:** GDB Debugger

iv. Unter *Common*:

- **Display in favorites menu:** Sowohl *Run* als auch *Debug* auswählen.

(j) Wenden Sie die vorgenommenen Einstellungen mit *Apply* an.

(k) Danach finden Sie in den Favoriten-Listen der Run- und Debug-Konfigurationen neue Einträge, um StEAM mit dem angegebenen Modell ausführen bzw. zu debuggen.

Literaturverzeichnis

[CGP99] Clarke, E. M., O. Grumberg und D. A. Peled: *Model checking*. MIT Press, 1999.

[CW97] Chen, Jiun-Liang und Feng-Jian Wang: *Slicing Object-Oriented Programs*. In: *Proceedings of the 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC'97/ICSC'97)*, December 1997.

[DHJ+01] Dwyer, Matthew B., John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Robby, Hongjun Zheng und W Visser: *Tool-Supported Program Abstraction for Finite-State Verification*. In: *International Conference on Software Engineering*, Seiten 177–187, 2001.

[EJS04] Edelkamp, Stefan, Shahid Jabbar und Stefan Schroedl: *External A**. KI Zeitschrift, Seiten 226–240, 2004.

[ELL01] Edelkamp, Stefan und A. Lluch-Lafuente: *HSF-Spin User Manual*, 2001.

[ELLL04] Edelkamp, S., S. Leue und A. Lluch-Lafuente: *Directed Explicit-State Model Checking in the Validation of Communication Protocols*. STTT, 5(2-3):247–267, 2004.

[God05] Godefroid, P.: *Software Model Checking: The VeriSoft Approach*. Formal Methods in System Design, 26(2):77–101, 2005.

[GV02] Groce, A. und W. Visser: *Model checking Java programs using Structural Heuristics*. In: *ISSTA*, Seiten 12–21, 2002.

[HNR68] Hart, P.E., N.J. Nielsson und B. Raphael: *A formal basis für heuristic determination of minimum path cost*. In: *IEEE Transactions on Systems Science and Cybernetics*, Band 4, Seiten 100–107, 1968.

[Hol04] Holzmann, G.: *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[KM03] Kristensen, L. M. und T. Mailund: *Path Finding with the Sweep-Line Method using External Storage*. In: *ICFEM*, Seiten 319–337, 2003.

[LV01] Lerda, F. und W. Visser: *Addressing dynamic issues of program model checking*. In: *SPIN*, Seiten 80–102, 2001.

[Sai00] Saidi, H.: *Model checking guided abstraction and analysis*. In: *SAS 00: Static-Analysis Symposium*, Seiten 377–396, 2000.

[Sca05] Scarpino, Matthew: *SWT/JFace in action*. Manning, 2005.

[SD98] Stern, U. und D. Dill: *Using Magnetic Disk instead of Main Memory in the Murphi Verifier*. In: *CAV*, Seiten 172–183, 1998.

[SMS02] Sanders, P., U. Meyer und J. F. Sibeyn: *Algorithms for Memory Hierarchies*. Springer, 2002.

[VHBP00a] Visser, W., K. Havelund, G. Brat und S. Park: *Java PathFinder - Second Generation of a Java Model Checker*, 2000.

[VHBP00b] Visser, Willem, Klaus Havelund, Guillaume Brat und Seung-Joon Park: *Model Checking Programs*. In: *Proc. of the 15th IEEE International Conference on Automated Software Engineering*, Seiten 3–12, 2000.

Listings