UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

Bachelorarbeit

Ein typbasierter Ansatz zur Kombination verschiedener Verifikationstechniken

Friedberger Karlheinz

Oktober 2012

Betreuer: Prof. Dr. rer. nat. Dirk Beyer

Abstrakt

Um Fehler in Programmen zu finden, wird von Softwareentwicklern vermehrt Model-Checking eingesetzt. Ein Hauptproblem dabei ist aber der große Zustandsraum, der bei der Analyse komplexer Programme auftreten kann. Es existieren verschiedene Ansätze, um während der Untersuchung die Anzahl an Zuständen überschaubar zu halten.

In dieser Arbeit werden zwei unterschiedliche Analysen des Model-Checkings kombiniert, um ihre jeweiligen Vorteile zu nutzen: die Explizite Analyse und eine BDD-basierte Analyse. Die Grundlage der BDD-basierten Analyse ist eine effiziente Datenstruktur, mit welcher eine große Menge an Zuständen performant verwaltet werden kann. Die Explizite Analyse hingegen erlaubt aufgrund ihrer Schlichtheit eine schnelle Untersuchung eines Programmpfades.

Für die Kombination der Analysen werden die Variablen automatisch in verschiedene Klassen eingeteilt. Die Unterteilung ermöglicht es, jede Variable mit der dafür am besten geeigneten Analyse zu untersuchen. Wichtige Variablenklassen sind hierbei vor allem die booleschen und die diskreten Variablen.

Diese Arbeit behandelt die Integration der Variablenklassifikation und der BDD-basierten Analyse in das Framework CPAchecker. Dabei wird auf die Variablenklassifizierung und die kombinierten Analysen eingegangen. Außerdem werden die Vorteile der Kombination der Analysen anhand verschiedener Testläufe empirisch untersucht.

Inhaltsverzeichnis

1 Einleitung

In Computerprogrammen gibt es häufig Fehler, also unerwünschtes Verhalten unter bestimmten Bedingungen. Diese Tatsache ist jedem Softwareentwickler und vielen Anwendern bekannt. Ein Programmierer verbringt einen Großteil seiner Arbeitszeit damit, Software zu analysieren, Fehler zu finden und diese dann zu beheben, damit ein Programm fehlerfrei läuft.

Es gibt verschiedene Methoden um Software von Fehlern zu befreien.

Ein Ansatz zum Erkennen von Fehlern ist das Testen. Hierbei wird ein Programm mit verschiedenen Eingaben unter verschiedenen Bedingungen durchlaufen und die Ausgaben mit dem jeweils erwarteten Ergebnis verglichen. Leider kann Testen nur die Abwesenheit von Fehlern für die gegebenen Testfälle, nicht aber die Korrektheit des Programms im Allgemeinen bestätigen. Zusätzlich kann die Anzahl der möglichen Eingaben so groß sein, dass aus Zeit- oder Kostengründen nur eine geringer Anteil davon tatsächlich getestet wird.

Ein anderer Ansatz ist die Modellprüfung („Model-Checking“), welche ein automatisches Verfahren zur Verifikation eines Systemmodells gegen eine Spezifikation ist. Der Vorteil des Model-Checkings liegt darin, dass man durch eine mathematische Formulierung vom Programm selbst abstrahieren und durch die Verifikation einen Beweis für die Korrektheit des Programms bezüglich der Spezifikation finden kann. Durch die Abstraktion erhält man ein Model der Software, welches mit verschiedenen Analysen untersucht werden kann.

Beim Model-Checking werden mögliche Zustände für ein Programm erzeugt und diese daraufhin untersucht, ob sie der Spezifikation widersprechen. Ein großes Problem dabei ist die Zustandsexplosion, welche bei komplexen Programmen auftreten kann. Eine große Anzahl an Zuständen verursacht nicht nur eine länger dauernde Analyse, sondern es werden auch immense Rechenressourcen benötigt. Es existieren verschiedene Ansätze um die Anzahl an Zuständen überschaubar zu halten.

Diese Arbeit behandelt die Kombination der Expliziten Analyse mit einer BDD-basierten Programmanalyse. Als Grundlage wird dabei der Ansatz von Rhein, Apel und Raimondi [11, 12] verwendet, bei welchem Programme mit einer hybriden Analyse untersucht werden, so dass annotierte boolesche Variablen mit BDDs und alle weiteren Variablen explizit analysiert werden. Die neuartige Idee dieser Arbeit ist die automatische Erkennung verschiedener Variablentypen und die unterschiedliche Kodierung dieser Typen zur optimalen Nutzung der BDD-basierten Analyse.

In dieser Arbeit wird zuerst auf die Grundlagen von BDDs und auf das Framework CPAchecker eingegangen, welches für die Analysen verwendet wird. Nach einer Beschreibung der typbasierten Klassifizierung von Variablen werden in Abschnitt 4 und 5 die Explizite und die BDD-basierte Analyse vorgestellt. Abschnitt 6 beschreibt die durchgeführten Tests und zeigt anhand von Ergebnissen, welche Vorteile die Kombination der Analysen hat.

2 Voraussetzungen für die Kombination der Analysen

In dieser Arbeit werden die Explizite Analyse und eine BDD-basierte Analyse im Framework CPAchecker kombiniert. Aus diesem Grund werden hier die Grundlagen zu BDDs erklärt und das Programm CPAchecker vorgestellt.

2.1 Grundlagen von Binären Entscheidungsdiagrammen

Ein Binäres Entscheidungsdiagramm („Binary Decision Diagram“, kurz BDD) ist eine Datenstruktur, welche eine Menge boolescher Variablen und deren Verknüpfungen effizient repräsentieren kann.

BDDs wurden von Lee (1959) [10] und Akers (1978) [1] über if-then-else-Programme definiert und zur Darstellung von Schaltkreisen verwendet, da viele logische Schaltungen als boolesche Funktionen dargestellt werden können. BDDs werden aber nicht nur zur Darstellung, sondern auch zur Analyse und Verifikation von Hard- und Software verwendet.

Ein BDD ist ein gerichteter azyklischer Graph $G = (N, E)$ mit einer Menge N an Knoten und einer Menge E an Kanten. Dieser Graph besitzt genau einen Wurzelknoten. Jeder Knoten aus N ist entweder ein innerer Knoten oder ein Endknoten. Die Endknoten enthalten entweder den Wert 0 oder den Wert 1 und besitzen keine ausgehende Kante. Jeder innere Knoten enthält eine boolesche Variable und besitzt zwei ausgehende Kanten, die als Low- und High-Kante bezeichnet werden. Dabei sind die Knoten $high(x)$ und $low(x)$ die jeweiligen Nachfolgerknoten der High- bzw. Low-Kante eines Knotens mit der Variablen x. Eine Variable kann in mehreren Knoten enthalten sein.

Jeder Knoten repräsentiert eine boolesche Funktion, welche durch den Knoten selbst und alle seine Kindknoten definiert ist. Der Funktion, die durch einen Knoten x gegeben ist, lautet

$$ite(x, high(x), low(x)) = \begin{cases} high(x) & \text{if } x \\ low(x) & \text{if } \neg x \end{cases} .$$

Die boolesche Funktion des gesamten BDDs ist durch seinen Wurzelknoten gegeben. In den hier verwendeten Grafiken ist die High-Kante eine durchgezogene Linie, die Low-Kante wird punktiert dargestellt.

Abbildung 1: BDDs der Funktionen x und $\neg x$

2.1.1 Operationen

BDDs unterstützen alle booleschen Operatoren wie beispielsweise Konjunktion ($\wedge$), Disjunktion ($\vee$), Negation ($\neg$), Implikation ($\Rightarrow$) und Äquivalenz ($\Leftrightarrow$) sowie weitere Operationen wie Existenz- und Allquantifizierung ($\exists$ bzw. $\forall$) von Variablen. Die Abbildungen 1 und 2 zeigen die BDDs einfacher Funktionen.

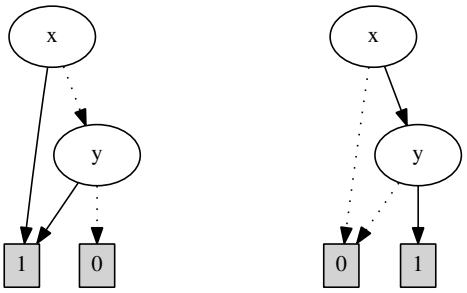

Abbildung 2: BDDs der Funktionen $x \vee y$ und $x \wedge y$

2.1.2 Reduktion und Variablenordnung

Für Analysen wird meist ein reduzierter und geordneter BDD („Reduced Ordered Binary Decision Diagram“, kurz ROBDD) verwendet. Mit ROBDDs sind viele boolesche Operationen in polynomieller Laufzeit $O(n)$ in Abhängigkeit von der Anzahl n der Knoten der Operanden möglich [7]. Da in der Literatur der Begriff BDD meist einen ROBDD bezeichnet, wird auch in diesem Dokument der Begriff BDD anstelle von ROBDD benutzt.

Die Reduktion eines ROBDDs ist durch zwei Regeln festgelegt:

- Alle isomorphen Subgraphen werden zusammengefasst.
- Alle Knoten, deren ausgehende Kanten den gleichen Zielknoten haben, werden eliminiert. Die eingehenden Kanten werden zum Nachfolger geführt.

Bei einem geordneten BDD sind die Variablen auf jedem Pfad von der Wurzel zu einem Endknoten gleich geordnet. Es müssen nicht alle Variablen auf jedem dieser Pfade verwendet werden. Die Ordnung der Variablen hat großen Einfluss auf die Anzahl der Knoten im Graphen. Bei einer guten Ordnung hat ein BDD wenig Knoten. Leider ist eine optimale Variablenordnung im Allgemeinen schwer zu finden.

Die Grafiken 3 und 4 zeigen dieselbe Funktion $(x_0 \Leftrightarrow y_0) \wedge (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ mit verschiedenen Variablenordnungen. Im ersten Fall sind die Variablen alphabetisch geordnet (x_0, x_1, x_2, y_0, y_1, y_2). Dadurch entstehen bei dieser Funktion exponentiell viele Knoten im BDD. Bei einer positiven Anzahl n an Variablenpaaren (x_i, y_i) mit $0 \leq i < n$ gibt es hier genau $3(2^n - 1)$ Knoten im BDD für die Variablen und zusätzlich noch die zwei Endknoten 0 und 1. Die zweite Grafik zeigt den BDD mit alternierenden Variablen (x_0, y_0, x_1, y_1, x_2, y_2). Hierbei entstehen nur linear

viele Knoten für die Variablen, nämlich genau $3n$. Die zweite Ordnung ist für die gegebene Funktion besser und wird in dieser Arbeit später auch für Bitvektoren verwendet.

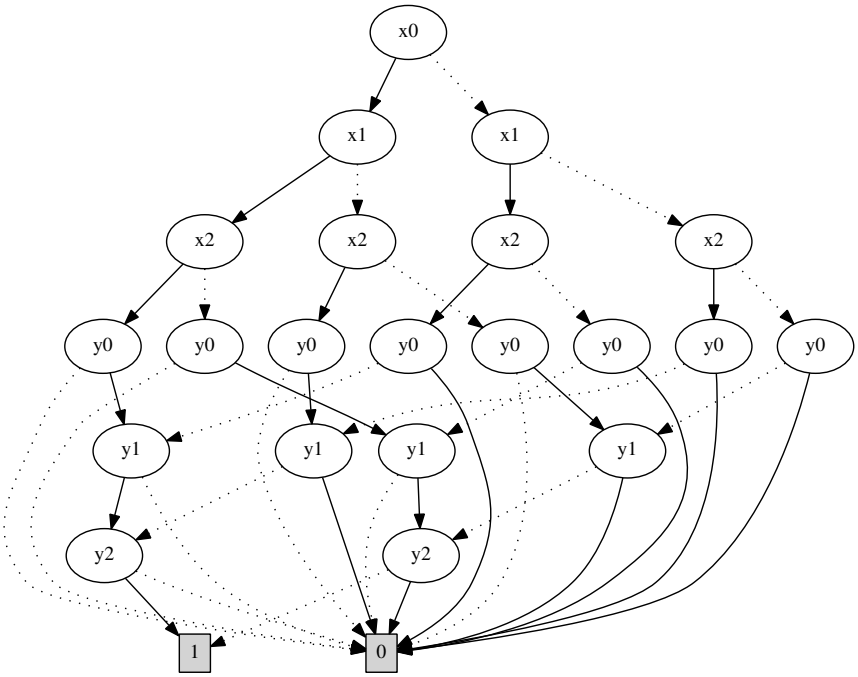

Abbildung 3: BDD der Funktion $(x_0 \Leftrightarrow y_0) \wedge (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ mit schlechter Variablenordnung

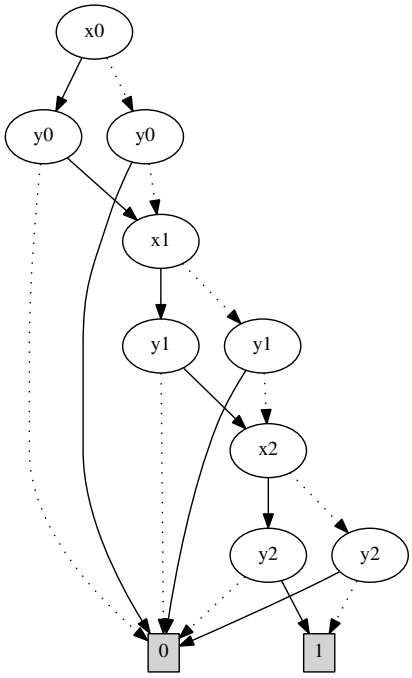

Abbildung 4: BDD der Funktion $(x_0 \Leftrightarrow y_0) \wedge (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ mit guter Variablenordnung

Der Vorteil eines reduzierten und geordneten BDDs ist, dass es für eine gegebene Variablenordnung und eine gegebene Funktion genau eine eindeutige Darstellung als ROBDD gibt.

Wenn in der Literatur von einem BDD die Rede ist, wird dabei meist ein reduzierter und geordneter BDD gemeint, da für Analysen und in Programmen nur diese Art von BDD verwendet wird. Auch für die in dieser Arbeit behandelte Softwareanalyse werden nur ROBDDs verwendet.

2.1.3 BDD-Bibliotheken

Es existieren effiziente Implementierungen wie beispielsweise CUDD[1], BuDDy[2] oder JavaBDD[3], um komfortabel mit BDDs zu arbeiten.

All diese Implementierungen weisen relativ ähnliche Funktionen zur Manipulation von BDDs auf. Sie verwenden zur Speicherung der Knoten aller verwalteten BDDs ein einzelnes Array, das bei Bedarf vergrößert wird. Außerdem werden Caches für die zuletzt berechneten BDDs eingesetzt, um die Operationen zu beschleunigen. Um Speicherplatz zu sparen, werden identische Subgraphen von verschiedenen BDDs nur einmal gespeichert und dann mehrfach referenziert. So ist in Abbildung 2 die Funktion y jeweils als Subgraph in der Funktion $x \wedge y$ bzw. $x \vee y$ vorhanden. Mit einer guten Variablenordnung kann der Anteil an identischen Subgraphen erhöht werden. Dies hat zur Folge, dass unnötige BDD-Operationen verhindert und damit Laufzeit und Speicherplatz gespart werden.

[1] `http://vlsi.colorado.edu/~fabio/CUDD`
[2] `http://buddy.sourceforge.net/manual/main.html`
[3] `http://javabdd.sourceforge.net/`

2.2 Einführung in das Programm CPAchecker

Das Programm CPAchecker ist ein Framework, mit dem es möglich ist Programme zu analysieren und anhand einer Spezifikation zu überprüfen. Eine detaillierte Beschreibung des Programms findet sich bei Beyer und Keremoglu [6].

CPAchecker ist dazu in der Lage, verschiedene Analysen zu kombinieren und dadurch ein Programm mit unterschiedlichen Algorithmen zu untersuchen. Bei der 1. Competition on Software Verification 2012 erreichte CPAchecker mit zwei verschiedenen Konfigurationen, welche jeweils eine andere Analyse benutzten, die ersten beiden Plätze [3].

CPAchecker ist in Java programmiert. Sowohl der Quellcode von CPAchecker als auch einige Codebeispiele sind unter `http://cpachecker.sosy-lab.org` online verfügbar.

2.2.1 Funktionsweise

CPAchecker generiert aus C-Quellcode die Kontrollflussgraphen („Control Flow Automaton“, kurz CFA) für alle Funktionen des Programms. Ein Kontrollflussgraph ist ein gerichteter Graph mit genau einem Startknoten. Jeder Knoten eines CFA stellt eine Position im Programm dar. Der Startknoten entspricht dem Programmstart bzw. dem Funktionsaufruf. Die Endknoten, welche Fehlerpositionen im Programm darstellen, werden als Fehlerzustand markiert. Jede Anweisung im Code wird auf eine oder mehrere Kanten im Graphen abgebildet. In Abbildung 5 wird beispielsweise der CFA zu Code 1 dargestellt.

Für die Analyse des kompletten Programms werden die CFAs der einzelnen Funktionen zu einem großen Graphen verknüpft, indem Funktionsaufrufe innerhalb eines CFAs mit den CFAs der entsprechenden Funktionen verbunden werden. Auf dem Gesamtgraphen können von CPAchecker verschiedene Algorithmen ausgeführt werden. Für die hier verwendete kombinierte Analyse wird der CPA-Algorithmus benutzt.

Code 1: Programm mit einfachem Kontrollfluss

```
1  int main(void) {
2      int a = 1;
3      int b = 0;
4
5      if (a == 1) {
6          b = 10;
7      } else {
8          b = 5;
9      }
10
11     while (a < b) {
12         a = a + 1;
13     }
14
15     return (0);
16 }
```

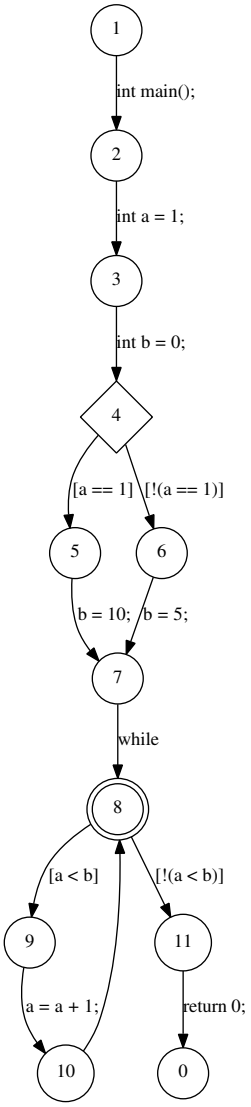

Abbildung 5: CFA zu Code 1

2.2.2 CPA-Algorithmus

Der Hauptalgorithmus von CPAchecker ist der CPA-Algorithmus, dessen Funktion es ist, mit Hilfe einer konfigurierbaren Programmanalyse („Configurable Program Analysis“, kurz CPA) einen Erreichbarkeitsgraphen („Abstract Reachability Graph“, kurz ARG) zu erstellen. Jeder Knoten im ARG ist ein abstrakter Zustand des Programms. Die Kanten stellen die Übergänge zwischen den Zuständen dar. Falls zwei Pfade im ARG dieselben Zustände besitzen, werden sie zusammengefasst. Dies reduziert die Größe des Graphen.

Der Algorithmus prüft die Erreichbarkeit der Fehlerzustände. Bei Erreichen eines Fehlerzustandes gilt ein Programm als fehlerhaft. Terminiert der CPA-Algorithmus ohne einen Fehlerzustand zu erreichen, gilt ein Programm als fehlerfrei.

Der Algorithmus selbst ist ein iteratives Verfahren. Ausgehend von einer initialen

Menge an Zuständen werden durch Traversieren des Kontrollflussgraphen weitere Zustände für den ARG erstellt. Das Erzeugen weiterer Zustände wird von der CPA übernommen. Der Algorithmus terminiert, wenn ein Fehlerzustand erreicht wird oder keine weiteren Zustände mehr für den ARG erzeugt werden können.

Eine genaue Beschreibung des Algorithmus findet sich bei Beyer, Henzinger und Théoduloz [4].

2.2.3 Aufbau einer CPA

Eine einzelne CPA besteht formal aus einer abstrakten Domäne („Domain“), einer Transferrelation, einer Genauigkeit („Precision“), einem Merge- und einem Stop-Operator. CPAchecker enthält für alle diese Elemente Interfaces. Dadurch kann jede CPA eine andere Analyse implementieren und der CPA-Algorithmus selbst kann unabhängig von den Interna der CPAs agieren.

Die abstrakte Domain bestimmt die Grundfunktionen der Analyse und die Struktur der abstrakten Zustände. Außerdem legt die Domain fest, wann ein Zustand s von einem Zustand s' überdeckt. Eine Überdeckung $s \sqsubseteq s'$ besteht genau dann, wenn s' abstrakter ist als s, also wenn der Zustand s eine Teilmenge der Informationen des Zustands s' besitzt.

Die Transferrelation erzeugt ausgehend von einem Zustand und einer Kante des CFA eine Menge an neuen Zuständen. Der Merge-Operator kombiniert einen Zustand s mit einem anderen Zustand s', wobei s' vom Ergebniszustand überdeckt werden muss: $s' \sqsubseteq merge(s, s')$. Der Stop-Operator gibt an, ob ein Zustand von einer Kombination mehrerer Zustände überdeckt wird. Die Kombination von Zuständen kann nur ein einzelner Zustand davon oder auch die Vereinigung der Zustände sein.

Zusätzlich kann für jede CPA eine Genauigkeit angegeben werden, welche überwacht, welche Variablen bzw. welche Kanten analysiert werden sollen. Damit ist es möglich, verschiedene Kanten von unterschiedlichen CPAs behandeln. Es ist möglich, die Precision während der Analyse zu ändern [5] und dadurch den Schwerpunkt der gesamten Analyse zwischen verschiedenen CPAs zu verschieben. Für diese Arbeit wird aber nur eine statische Precision verwendet, welche zu vor Beginn des CPA-Algorithmus erstellt wird und für den Verlauf der Analyse unverändert bleibt.

2.2.4 Kombination von Analysen

Der CPA-Algorithmus operiert mit genau einer CPA. Diese CPA ist in den meisten Fällen die CompositeCPA, welche selbst mehrere unterschiedliche CPAs aufnehmen, Funktionsaufrufe an sie durchreichen und die Ergebnisse entsprechend zusammensetzen kann. Dadurch können verschiedene Analysen kombiniert werden.

Als eine grundlegende Analyse wird häufig die LocationCPA verwendet, welche die Position im Kontrollflussgraphen analysiert. Dadurch müssen andere Analysen keine Rücksicht auf die konkrete Programmposition nehmen.

3 Typbasierte Klassifizierung der Variablen

Für diese Arbeit werden eine Explizite und eine BDD-basierte Analyse kombiniert um die Vorteile beider Analysen zu nutzen. Der Vorteil der BDD-basierten Analyse liegt in der effizienten Verwaltung von booleschen Funktionen. Damit können beispielsweise boolesche Variablen analysiert werden. Um aus einem Programmcode die entsprechenden Variablen zu extrahieren, müssen die Variablen des Programms in mehrere disjunkte Klassen aufgeteilt werden. Nur so ist es möglich, jede Variable mit der dafür am besten geeigneten Analyse zu behandeln.

Im Folgenden wird erklärt, welche Variablenklassen verwendet werden, welcher Struktur diese unterliegen und wie die Variablen automatisch eingeteilt werden.

3.1 Klassifizierung von Variablen

Für die Klassifikation braucht man Informationen über das Laufzeitverhalten der Variablen. Hierfür werden sowohl die Werte der Variablen als auch Verknüpfungen mit anderen Variablen betrachtet. Dabei ist auch die Art der Verknüpfung, also der Operator, von Bedeutung.

Im Folgenden werden die Klassen von Variablen erklärt, welche für die Kombination der Expliziten mit der BDD-basierten Analyse relevant sind.

3.1.1 Boolesche Variablen

Boolesche Variablen werden daran erkannt, dass sie bei Berechnungen nur in Verbindung mit logischen Operatoren verwendet werden. Logische Operationen sind hierbei z. B. Konjunktion (&&), Disjunktion (||) und Negation (!). Außerdem sind Tests auf Gleichheit und Ungleichheit (== bzw. ! =) erlaubt.

In der Programmiersprache C gibt es keinen expliziten Datentyp für boolesche Variablen, sondern es wird der Typ `Integer` dafür verwendet. Dadurch ist bei Initialisierung einer Variablen nicht ersichtlich, ob diese im weiteren Code bei Berechnungen als boolesche Variable verwendet wird oder nicht. Aus diesem Grund sind für die Erkennung von booleschen Variablen im Quellcode nur Zuweisungen und Vergleiche erlaubt, bei denen die Zuweisung bzw. der zweite Operand selbst eine boolesche Variable, eine logische Operation oder die Zahl 0 ist.

Im Code 2 sind alle Variablen boolesch, da keine Variable für eine nicht-boolesche Operation verwendet wird.

Code 2: Quellcode mit booleschen Variablen

```
1 if (A == 0) {
2     D = B && C;
3 } else {
4     D = !B;
5 }
```

3.1.2 Diskrete Variablen

Die Klasse der diskreten Variablen enthält alle Variablen, welche nicht Bestandteil einer Berechnung sind, sondern nur direkt zugewiesen oder auf Gleichheit bzw. Ungleichheit (== bzw. ! =) getestet werden. Die rechte Seite einer Zuweisung darf nur ein Zahlenwert oder eine andere diskrete Variable sein. Als Zahlenwerte sind alle ganzen Zahlen erlaubt.

Diese Einschränkungen gewährleisten, dass alle möglichen Belegungen der Variablen dieser Klasse bekannt sind. Diese Information kann für Optimierungen genutzt werden. Da alle Variablen dieser Klasse nur eine endliche Anzahl an Werten haben, kann diese Klasse auch bei den endlichen Domänen („finite domain“) oder endlichen Automaten („finite state machine“) eingeordnet werden.

Eine diskrete Variable kann auch eine boolesche Variable sein und umgekehrt. Alle Variablen im Code 3 sind diskret und die Variable A zusätzlich boolesch.

Code 3: Quellcode mit diskreten Variablen

```
1 if (A == 0) {
2     B = 3;
3     C = B;
4 } else {
5     C = 0;
6 }
```

3.1.3 Variablen in einfachen Berechnungen

Die Operationen, welche im Zusammenhang mit den in dieser Klasse befindlichen Variablen möglich sind, umfassen sowohl Zuweisungen und logische Operationen als auch Addition (+), Subtraktion (−), Vergleiche (<, >, <=, >=) und binäre Bitoperationen (&, |, ∧). In dieser Klasse sind dadurch auch alle Variablen enthalten, welche boolesch oder diskret sind.

Die von dieser Klasse ausgeschlossenen Operationen sind beispielsweise Multiplikation (∗), Division (/), Modulo (%) und Bitshift (<<, >>).

Code 4: Quellcode mit Berechnungen

```
1 if (A && B) {
2     C = 3;
3 } else {
4     D = E + 5;
5     F = G * 2;
6 }
```

Im Code 4 werden die Variablen A, B, C, D und E in einfachen Berechnungen verwendet. Die Variablen A und B sind boolesch, die Variable C ist diskret und die Variablen F und G befinden sich in keiner der drei genannten Klassen.

Jede Variable, die nicht den oben genannten Kriterien entspricht, wird nicht klassifiziert, sondern sie befindet sich stattdessen in der Restmenge der Variablen. Grafik 6 zeigt die Einteilung der genannten Klassen und deren Zusammenhang.

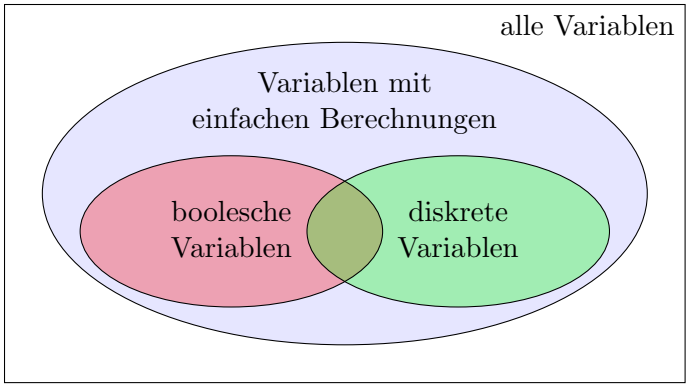

Abbildung 6: Übersicht über die genannten Variablenklassen

3.2 Partitionierung von Variablen

Um Variablen zu klassifizieren, müssen zuerst grundlegende Gruppen von zusammengehörenden Variablen erstellt werden. Diese Gruppen werden Partitionen genannt. Variablen aus derselben Partition befinden sich auch in der gleichen Klasse.

Zur Partitionierung werden Variablen entsprechend ihrer Abhängigkeiten zusammengefasst, sodass sich voneinander abhängige Variablen in der gleichen Partition befinden. Zwei Variablen gelten hierbei genau dann als abhängig, wenn sie Teil derselben Anweisung sind. Bei dieser Einteilung werden Kontrollflussabhängigkeiten ignoriert und nur die einzelnen Anweisungen für sich betrachtet.

Durch die Partitionierung können Variablen, welche im Quellcode selbst keinen expliziten Wert zugewiesen bekommen oder nicht Teil eines mathematischen Ausdrucks sind, klassifiziert werden. Zusätzlich besteht die Möglichkeit die Partitionen später zur Optimierung der BDD-basierten Analyse zu nutzen.

Code 5: Quellcode mit Variablenabhängigkeiten

```
1 A = B;
2 if (X == Y) {
3     D = B + C;
4 } else {
5     X = Z;
6 }
```

Im Code 5 finden sich die beiden Partitionen $\{A, B, C, D\}$ und $\{X, Y, Z\}$. Die Variablen X und Y werden miteinander verglichen und sind daher voneinander abhängig. Der Variable X wird außerdem der Wert Z zugewiesen. Daher sind X, Y und Z in einer Partition. Die Variable A erhält den Wert von B und die Summe von B und C wird der Variablen D zugewiesen. Deshalb müssen A, B, C und D zusammen in einer Partition sein.

3.3 Implementierung der Partitionierung und Klassifizierung

Zur Partitionierung und Klassifizierung der Variablen wurde in CPAchecker eine Analyse implementiert, welche bereits vor dem CPA-Algorithmus aufgerufen wird. Dadurch sind bereits bei der Instanziierung der CPAs Informationen verfügbar, mit denen die spätere Programmanalyse optimiert werden kann.

Die Variablenanalyse iteriert genau einmal über alle Kanten des CFAs und untersucht dabei jede Kante auf Variablen, Zahlenwerte und Operationen. Mit Hilfe dieser drei Kriterien werden die Variablen in Partitionen und oben genannte Klassen eingeteilt. Die Analyse zur Klassifizierung der Variablen speichert zusätzlich ab, welche Kanten des CFAs zu welchen Partitionen gehören. Diese Information wird beispielsweise von der BDD-basierten Analyse verwendet.

3.3.1 Vorgehensweise bei der Partitionierung

Alle Variablen einer Kante des CFAs werden in die gleiche Partition eingefügt. Sollten sich noch keine der Variablen in einer Partition befinden, wird eine neue Partition mit den Variablen erstellt. Falls Variablen bereits in Partitionen enthalten sind, werden alle beteiligten Partitionen zu einer neuen Partition vereinigt und alle Variablen dieser Kante in diese Partition eingefügt. Da sich keine Variable in mehreren Partitionen gleichzeitig befindet, sind alle Partitionen disjunkt.

Einen Sonderfall bildet ein Funktionsaufruf. Bei dieser Kante werden die Parameter getrennt voneinander behandelt, da sie sich in verschiedenen Partitionen befinden können. Anstatt des kompletten Funktionsaufrufs werden hier die Zuweisungen der Argumente an die entsprechenden Parameter einzeln untersucht.

Bei der Funktionsrückkehr wird eine temporäre Variable verwendet, welche für jede Funktion einen eindeutigen Namen hat. Diese Variable wird (wenn nötig) bei der Zuweisung des Rückgabewertes verwendet. Damit ist gewährleistet, dass bei einer Funktion mit mehreren Rücksprungmöglichkeiten jeder zurückgegebene Wert in der gleichen Partition ist. Es wird hierbei angenommen, dass sich eine Variable innerhalb einer Funktion bei jedem Aufruf dieser Funktion in derselben Partition befindet.

3.3.2 Vorgehensweise bei der Klassifizierung

Initial befindet sich jede Variable in jeder der oben genannten Klassen. Gleichzeitig zur Partitionierung werden bei jeder Kante des CFAs die dazugehörigen Operatoren und Zahlenwerte überprüft, um eventuell die Variablen dieser Kante sofort aus einer der oben genannten Klassen auszuschließen. Wenn eine Partition eine von einer bestimmten Klasse ausgeschlossene Variable enthält, wird die gesamte Partition aus dieser Klasse entfernt.

3.3.3 Einsammeln von Zahlenwerten

Da bei der Klassifizierung ohnehin jeder Ausdruck auf Zahlenwerte untersucht wird, ist es sinnvoll, diese Werte zusammen mit der Partition der Variablen abzuspeichern. Hierbei werden Zahlen mit einem negativem Vorzeichen als negative Werte behandelt. Durch Speichern der Werte kann bei späteren Analysen festgestellt werden, mit welchen Zahlen eine Variable möglicherweise verrechnet wird. Diese Information wird z. B. von der BDD-basierten Analyse zur Behandlung von diskreten Variablen genutzt.

4 Explizite Analyse

Bei der Expliziten Analyse („Explicit Analysis“) werden die genauen Werte der Variablen verfolgt und in den Zuständen gespeichert. Bei jedem erreichten Zustand sind die Werte aller Variablen bekannt, welche auf dem bisherigen Pfad durch den CFA zugewiesen wurden. Die Transferrelation wertet bei jeder Zuweisung die rechte Seite aus und speichert das Ergebnis der Zuweisung zusammen mit allen bisherigen Zuweisungen, welche aus dem Vorgängerzustand übernommen werden, in einem neuen Zustand.

4.1 Merge- und Stop-Operator

Die Explizite Analyse verwendet die Operatoren $merge^{sep}$ und den $stop^{sep}$, wie sie bei Beyer, Henzinger und Théoduloz [4] beschrieben werden.

Der Operator $merge^{sep}$ ist durch die Funktion $merge^{sep}(s,\ s') = s'$ definiert. Der Operator $stop^{sep}$ prüft, ob es einen Zustand s' aus den bereits erreichten Zuständen gibt, welcher den neuen Zustand s überdeckt:

$$stop^{sep}(s,\ R) = (\exists\ s' \in R :\ s \sqsubseteq s')$$

Eine Überdeckung $s \sqsubseteq s'$ besteht bei der Expliziten Analyse genau dann, wenn ein Zustand s eine Teilmenge der Zuweisungen eines Zustandes s' enthält.

Die Bedingung für $stop^{sep}$ ist bei der Expliziten Analyse selten erfüllt. Dadurch ist die Menge an erreichbaren Zuständen bei der Expliziten Analyse in vielen Fällen größer als bei einer anderen Analyse.

Bei der Expliziten Analyse wird auf dem CFA ein Algorithmus zur Tiefensuche ausgeführt, mit dem neue Kanten für den CPA-Algorithmus bestimmt werden. In der Regel wird mit Tiefensuche ein Fehlerzustand schneller gefunden als mit Breitensuche.

4.2 Vor- und Nachteile der Expliziten Analyse

Der Vorteil der Expliziten Analyse ist die schnelle Analyse eines Programms, wenn jede Variablen initialisiert wird, also wenn ihr immer ein Wert zugewiesen wird, bevor lesend darauf zugegriffen wird.

Genau hier liegt aber auch der Nachteil der Expliziten Analyse: Bei nicht initialisierten Variablen kann keine Aussage über die Belegung einer Variablen getroffen werden. Daher müssen bei Bedingungen mit einem Zugriff auf eine nicht initialisierte Variable beide ausgehenden Programmpfade weiter analysiert werden, da nicht festgestellt werden kann, ob die Bedingung zu $Wahr$ oder $Falsch$ evaluiert. Bei geeigneten Programmen nimmt die Anzahl der erreichbaren Zustände und damit auch Dauer und Speicherverbrauch der Analyse exponentiell zur Programmlänge zu.

5 BDD-basierte Programmanalyse

Bei der BDD-basierten Analyse werden alle Informationen eines Zustandes in einem BDD kodiert.

Prinzipiell ist es möglich, jede Variable eines Programm generell als Bitvektor mit 32 Bits aufzufassen. Ein solcher Bitvektor wird in einem BDD als Menge von 32 Variablen dargestellt. Bei entsprechender Hardwarearchitektur sind anstatt 32 Bit auch 64 Bit möglich. In diesem Fall werden entsprechend mehr BDD-Variablen benötigt.

Da aber die Größe eines BDDs stark von der Anzahl der Variablen abhängt, erreicht man durch eine geringere Anzahl an BDD-Variablen eine Verkleinerung der BDDs und eine Verbesserung der Laufzeit. Bei der BDD-Analyse werden die verschiedenen Variablenklassen unterschiedlich kodiert, um die Anzahl der notwendigen BDD-Variablen zu verringern. Im Folgenden werden die verschiedenen Kodierungen vorgestellt.

5.1 Analyse boolescher Variablen

Die BDD-Analyse verwaltet boolesche Variablen als einzelne BDD-Variablen. Jede logische Verknüpfung im Quellcode wird exakt auf die entsprechende BDD-Operation abgebildet.

5.2 Analyse diskreter Variablen

Falls eine Variable gleichzeitig boolesch und diskret ist, wird sie als boolesche Variable analysiert und kodiert.

Bei diskreten Variablen sind alle von einer Variablen möglicherweise angenommenen Werte W bekannt. Mit dieser Information ist es möglich, die Anzahl der BDD-Variablen und damit die Größe des BDDs zu reduzieren. Da bei jeder Partition die Zahlenwerte der in der Partition enthaltenen Variablen gespeichert sind, ist sinnvoll, die Partitionen getrennt voneinander zu untersuchen. Dadurch kann für jede Partition eine optimale Kodierung gefunden werden. Das folgende Vorgehen wird für jede Partition separat durchgeführt.

Bei diskreten Variablen reicht für eine Kodierung mit möglichst wenigen BDD-Variablen eine Abbildung aus, bei welcher die Werte aus W einfach durchnummeriert werden. Dieses Verfahren ist ähnlich zur Kodierung finiter Domains bei Bryant [8, S.16].

Die Abbildung ist injektiv, weil verschiedene Werte aus W unterschiedliche Nummern bekommen. Durch die Injektivität der Abbildung haben zwei Variablen genau dann einen identischen Wert, wenn sie der gleichen Nummer zugewiesen werden. Variablen mit unterschiedlichen Werten werden mit verschiedenen Nummern verrechnet.

In der Transferrelation wird bei diskreten Variablen jede Zahl durch ihre Nummer ersetzt. Damit werden im BDD die Variablen anstatt mit den Werten aus W mit den entsprechenden Nummern verknüpft. Eine Zuweisung oder ein Test auf Gleichheit bzw. Ungleichheit zweier Bitvektoren wird bitweise durchgeführt: Jedes Bit des ersten Operanden wird mit dem entsprechenden Bit des zweiten Operanden verrechnet.

Die Menge M enthält alle Nummern, welche für die Nummerierung der Zahlenwerte W benötigt werden. Des Weiteren gibt es ein zusätzliches Element in M, welches repräsentativ für alle nicht verwendeten Zahlenwerte steht, also für $\mathbb{Z} \setminus W$. Dieses Zusatzelement ist notwendig, da eine diskrete Variable auch ungleich allen bekannten Werten sein kann. In Code 6 stehen zwar für die Variable A nur die Werte 0 und 1 explizit im Code, A kann aber dennoch einen davon verschiedenen Wert annehmen, dessen genauer Wert irrelevant ist.

Code 6: Quellcode mit diskreten Variablen

```
1 if (A == 0) {
2     ...
3 } else if (A == 1) {
4     ...
5 } else {
6     ...
7 }
```

Um die Werte der Menge M im BDD eindeutig darzustellen, werden mindestens $log_2(|M|)$ BDD-Variablen benötigt, weil die größte Zahl aus M als Binärzahl mindestens $log_2(|M|)$ Stellen hat. Dieser Wert ist für alle relevanten Testfälle kleiner als die Bitgröße 32 und spart BDD-Variablen ein, wenn nur die Variablen im Programm nur wenige Werte annehmen.

In Code 6 gelte die Annahme, dass die Variable A in keinen weiteren, außer den angegebenen Anweisungen vorkommt. Dann enthält die Wertemenge W, welche zur Partition mit der Variablen A gehört, genau die Zahlen 0 und 1. Damit braucht man zur Kodierung mindestens $log_2(|W| + 1) = log_2(3) \approx 1.58$ BDD-Variablen. Die Variable A kann hier anstatt mit 32 mit nur 2 BDD-Variablen kodiert werden.

5.3 Analyse von Variablen aus einfachen Berechnungen

Variablen aus einfachen Berechnungen – ohne boolesche oder diskrete Variablen – werden als Vektoren mit 32 Bit kodiert und analysiert. Die Länge der Bitvektoren kann in CPAchecker mit einer Konfigurationsoption geändert werden. Zahlenwerte werden im Zweierkomplement dargestellt.

Mit Bitvektoren im BDD sind verschiedenste Rechenoperationen möglich, indem entsprechende Hardwarekomponenten durch BDD-Operationen simuliert werden.

Ein naheliegendes Beispiel ist hier ein Addiernetz, dessen Funktionsweise durch elementare Operationen im BDD nachgebildet werden kann. Dadurch wird eine Ad-

dition zweier Bitvektoren im BDD möglich. Eine BDD-basierte Addition ist natürlich langsamer als eine direkt in Hardware ausgeführte Addition, sie erlaubt es aber, auch symbolische (unbekannte, nicht initialisierte) Werte zu addieren. Das Addiernetz kann durch weitere Operationen erweitert werden um auch Subtraktion und Vergleiche wie $<$ und $>$ zu erlauben.

Eine Zuweisung oder ein Test auf Gleichheit bzw. Ungleichheit zweier Bitvektoren wird bitweise durchgeführt, genauso wie die binären Operationen &, | und $\wedge$. Bei logischen Operationen wie &&, || und ! werden die Bits der Operanden zuerst disjunktiv verknüpft. Dadurch ergibt sich für jeden Vektor ein einzelnes Bit, welches genau dann $False$ ist, wenn der Vektor gleich dem Nullvektor ist, also für alle Bits den Wert $False$ enthält. Mit diesem einen Bit wird die Operation durchgeführt und dann das Ergebnis auf 32 Bit verlängert, indem die vorderen 31 Stellen des Vektors auf $False$ gesetzt werden. Dadurch wird erreicht, dass der Bitvektor einer Binärdarstellung der Zahlen 0 oder 1 entspricht.

5.4 Implementierung der BDDCPA

Die BDDCPA und die dazugehörenden Klassen in CPAchecker ermöglichen es, Programme mit BDDs zu analysieren.

Die BDDCPA basiert auf der bereits vorhandenen FeatureVarsCPA, welche eine Menge boolescher Variablen, welche durch den Benutzer vorgegeben werden, mit BDDs verwalten kann. Die BDDCPA erweitert die FeatureVarsCPA um das Verwenden von Bitvektoren für nicht-boolesche Variablen und die Auswertung von booleschen und arithmetischen Ausdrücken. Somit können alle oben genannten Variablenklassen mit der BDDCPA behandelt werden. Des Weiteren wurde die Unterstützung für Funktionsaufrufe und das damit verbundene Verdecken („Scoping“) von Variablen hinzugefügt, um auch komplexere Programme analysieren zu können. Sowohl globale als auch lokale Variablen können mit BDDs behandelt werden.

Der Zugriff auf die BDD-Implementierung (JavaBDD) erfolgt über eine Wrapper-Klasse, welche eine Zuordnung von Namen an Variablen ermöglicht. Dadurch ist es für den Programmierer komfortabler, BDDs zu erstellen und benutzte Variablen wiederzuverwenden. Die Wrapper-Klasse enthält Methoden, um einen BDD in textueller Form als boolesche Formel oder graphisch als gerichteten Graphen darzustellen. Da diese Methoden eine relativ komprimiert gespeicherte Datenstruktur in eine vom Menschen lesbare Form konvertieren, besteht die Möglichkeit, dass sie zur Darstellung größerer BDDs viel Speicherplatz und Laufzeit benötigen.

5.5 Transferrelation

Während des CPA-Algorithmus entscheidet sich die Transferrelation je nach Kantentyp im CFA für eine oder mehrere Operationen auf dem BDD BDD_s des vorherigen Zustands s und erzeugt dadurch einen neuen BDD $BDD_{s'}$ für den Folgezustand s'. Da sich hinter diesen Operationen eine komplexe Analyse verbirgt, werden im Folgenden die Aktionen für die wichtigsten Kantentypen erklärt.

5.5.1 DeclarationEdge

Eine DeclarationEdge stellt die Deklaration einer Variablen dar. Optional wird ein Initialwert für die Variable angegeben. Der Initialwert kann das Ergebnis einer mathematischen oder logischen Operation, ein Zahlenwert oder eine andere Variable sein.

Die Transferrelation löscht einen möglicherweise vorhandenen Wert der deklarierten Variable aus BDD_s, indem sie ihn durch eine Existenzquantifizierung bindet. Sollte die Variable nicht frei in diesem BDD vorkommen, wird per Definition durch die Existenzquantifizierung nichts verändert. Falls ein Initialwert existiert, wird dieser symbolisch mit BDD-Operationen ausgewertet und der deklarierten Variable zugewiesen. Diese Zuweisung wird durch eine Konjunktion mit dem BDD verknüpft, der aus der Existenzquantifizierung folgt.

Falls die Variable ein Bitvektor ist, werden alle Operationen, also die Existenzquantifizierung, die Zuweisung und die Konjunktion, bitweise durchgeführt.

Code 7: Deklaration einer Variablen mit Initialwert

```
1 int A = 3;
```

Code 7 zeigt die Deklaration der Variablen A mit dem Initalwert 3. Es wird angenommen, dass die Variable A nicht diskret ist und sich in der Klasse der einfachen Berechnungen befindet, da in diesem Fall die Zahl 3 in Binärschreibweise (0011_2) im BDD dargestellt werden kann. Für die Kodierung im BDD werden hier Bitvektoren der Länge 4 verwendet.

Existenzquantifizierung:
$BDD_{exist} = \exists A_0, A_1, A_2, A_3 : BDD_s$

Auswertung der Zuweisung:
$BDD_{decl} = (A_0 \Leftrightarrow 1) \wedge (A_1 \Leftrightarrow 1) \wedge (A_2 \Leftrightarrow 0) \wedge (A_3 \Leftrightarrow 0)$

Verknüpfen der Zuweisung mit dem bisherigen Zustand:
$BDD_{s'} = BDD_{exist} \wedge BDD_{decl} =$
$(\exists A_0, A_1, A_2, A_3 : BDD_s) \wedge ((A_0 \Leftrightarrow 1) \wedge (A_1 \Leftrightarrow 1) \wedge (A_2 \Leftrightarrow 0) \wedge (A_3 \Leftrightarrow 0))$

5.5.2 StatementEdge

Diese Kante enthält die Zuweisung eines Wertes an eine Variable, so dass diese den neuen Wert besitzt. Der Wert selbst kann eine mathematische oder logische Berechnung sein. Es ist hierbei möglich, dass eine Variable sowohl auf der linken Seite als auch als Teil der rechten Seite der Zuweisung vorkommt (Code 8).

Code 8: Zuweisung einer Variablen mit Berechnung

```
1 A = A+B;
```

In diesem Fall wird die Zuweisung in zwei Zuweisungen zerlegt (Code 9): Als erste Operation wird die rechte Seite ausgewertet und einer temporären Variable zugewiesen. Dann wird der Wert der temporären Variable der eigentlichen Variable zugewiesen. Dieses Vorgehen ist notwendig, da vor der Zuweisung eine Existenzquantifikation der Variable durchgeführt werden muss und danach der Variablenwert nicht mehr für die Berechnung der rechten Seite verfügbar wäre.

Code 9: gesplittete Zuweisung

```
1 TMP = A+B;
2 A = TMP;
```

Die BDD-Operationen erfolgen analog einer Variablendeklaration (DeclarationEdge), also wird erst die symbolische Auswertung der rechten Seite der linken Seite zugewiesen und dann die Zuweisung mit dem BDD verknüpft, der aus der Existenzquantifizierung der Variable der linken Seite mit BDD_s folgt. Das Ergebnis ist $BDD_{s'}$.

5.5.3 AssumptionEdge

Eine AssumptionEdge enthält eine Bedingung. Es gibt hierbei immer zwei komplementäre Kanten, da eine Bedingung im Quellcode entweder zu $Wahr$ oder zu $Falsch$ evaluieren kann. Für beide Fälle existiert jeweils eine Kante vom gleichen Vorgängerknoten zu einem Nachfolger.

Die Transferrelation wertet die Bedingung symbolisch mit BDD-Operationen aus und fügt diese Auswertung durch Konjunktion BDD_s hinzu. Das Ergebnis ist der BDD $BDD_{s'}$. Falls $BDD_{s'}$ zu 0 evaluiert, bedeutet dies, dass die Bedingung nicht erfüllbar ist. In diesem Fall gibt es keinen Folgezustand s'. Im Gegensatz mit einer Zuweisung wird bei einer AssumptionEdge keine Existenzquantifikation der Variablen durchgeführt.

Code 10: Bedingung mit Abfrage auf Gleichheit

```
1 if (A == 3) { ... }
```

Code 10 vergleicht die Variable A mit der Zahl 3. Die Variable A sei nicht diskret und befinde sich in der Klasse der einfachen Berechnungen, sodass die Zahl 3 in

Binärschreibweise (0011_2) im BDD kodiert werden kann. Für die Kodierung im BDD werden Bitvektoren der Länge 4 verwendet.

Auswertung der Bedingung: $BDD_{assume} = (A_0 \Leftrightarrow 1) \wedge (A_1 \Leftrightarrow 1) \wedge (A_2 \Leftrightarrow 0) \wedge (A_3 \Leftrightarrow 0)$ Verknüpfen der Bedingung mit dem bisherigen Zustand: $BDD_{s'} = BDD_s \wedge BDD_{assume} =$ $BDD_s \wedge ((A_0 \Leftrightarrow 1) \wedge (A_1 \Leftrightarrow 1) \wedge (A_2 \Leftrightarrow 0) \wedge (A_3 \Leftrightarrow 0))$

5.5.4 FunctionCallEdge

Diese Kante repräsentiert einen Funktionsaufruf einer Funktion im Quellcode mit oder ohne Parametern. Für jedem Parameter der Funktion wird ein Argument übergeben. Jedes Argument kann eine mathematische oder logische Berechnung sein.

Nach symbolischer Auswertung der Argumente werden diese den Parametern zugewiesen. Alle Zuweisungen werden durch Konjunktion mit BDD_s verknüpft. Das Ergebnis dieser Operation ist $BDD_{s'}$.

5.5.5 FunctionReturnEdge

Die FunctionReturnEdge enthält die Berechnung des Rückgabewertes einer Funktion. Der eigentliche Funktionsrücksprung wird dann von der ReturnStatementEdge übernommen.

Nach symbolischer Auswertung des Rückgabewertes wird dieser einer temporären Variable zugewiesen, welche für jede Funktion einzigartig ist. Diese Zuweisung wird durch Konjunktion mit BDD_s verknüpft. Das Ergebnis dieser Operation ist $BDD_{s'}$.

5.5.6 ReturnStatementEdge

Diese Kante entspricht der Rücksprungposition einer Funktion. Hier wird eventuell der Rückgabewert der Funktion einer Variablen zugewiesen. Die BDD-Operationen der Zuweisung erfolgen hierbei analog einer Variablendeklaration an einer DeclarationEdge.

Alle Variablen, welche innerhalb der gerade verlassenen Funktion deklariert wurden, werden durch Existenzquantifizierung gelöscht. Diese Variablen befinden sich nicht im Gültigkeitsbereich der aktuellen Funktion und haben dadurch keinen Einfluss auf weitere BDD-Operationen. Durch die Existenzquantifizierung wird das wiederholte Aufrufen der gerade verlassenen Funktion ermöglicht und die Variablen- und Knotenzahl in $BDD_{s'}$ reduziert.

5.5.7 BlankEdge

Dieser Kantentyp hat keine explizite Bedeutung und wird für allgemeine Verbindungen zwischen Knoten wie z. B. Sprunganweisungen verwendet. Da an dieser Kante keine Berechnung stattfindet, gilt $s = s'$.

5.6 Merge-Operator

Die BDD-basierte Analyse verwendet einen Merge-Operator, welcher in Falle einer Merge-Operation zwei Zuständen miteinander vereinigt. Das Ergebnis der Merge-Operation ist ein Zustand, der die Disjunktion der BDDs der beiden Zustände enthält:

$$BDD_{merge} = BDD_s \vee BDD_{s'}$$

Beyer, Henzinger und Théoduloz [4] bezeichnen diesen Operator als $merge^{join}$.

Da die Merge-Operation bei der BDD-basierten Analyse häufig auftritt, ist für die BDD-basierte Analyse eine Breitensuche auf dem CFA von Vorteil gegenüber der Tiefensuche bei der Expliziten Analyse, da bei einem Ansatz mit Tiefensuche der komplette Pfad im CFA ab der Merge-Operation nochmals berechnet werden muss.

5.7 Stop-Operator

Der Stop-Operator der BDD-basierten Analyse ist ein $stop^{sep}$-Operator, wie er bei Beyer, Henzinger und Théoduloz [4] beschrieben wird. Eine Überdeckung eines Zustandes s durch einen anderen Zustand s' besteht hierbei genau dann, wenn s den Zustand s' impliziert:

$$stop^{sep}(s, R) = (\exists\, s' \in R : BDD_s \implies BDD_{s'})$$

Dadurch wird gewährleistet, dass s' und s identisch sind oder dass s' weniger Informationen enthält als der Zustand s.

5.8 Precision

Die Precision bestimmt, welche Variablen mit einer Analyse behandelt oder von ihr ignoriert werden sollen. In CPAchecker gehört zu jeder CPA eine Precision. Die für die Explizite und BDD-basierte Analyse benutzen Precisions sind komplementär, da jede Variablenklasse, welche mit der BDDCPA behandelt wird, von der Expliziten Analyse ausgeschlossen werden muss und umgekehrt.

Durch zum Programmstart übergebene Konfigurationsoptionen können die jeweils zu beachtenden Variablenklassen bestimmt werden. Damit können die Anteile der Analysen variiert und Tests mit verschiedenen Konfigurationen durchgeführt werden.

5.9 Verbesserung der Variablenordnung

Die Performance und Größe eines BDDs hängt stark von der Variablenordnung ab. Darum wird hier kurz auf diesen Aspekt eingegangen.

5.9.1 Festlegen der Variablenreihenfolge

Bei der verwendeten BDD-Implementierung gibt es zwei Möglichkeiten, die Variablenordnung in einem BDD zu bestimmen. Zum Einen werden die Variablen initial in der Reihenfolge sortiert, in welcher sie angelegt werden. Zum Anderen kann man Variablen zu einem späteren Zeitpunkt manuell (bzgl. einer gegebenen neuen Sortierung) oder automatisch (dynamische Sortierung) umordnen. Ein Vergleich der beiden Methoden kann bei Déharbe und Vidal [9] gefunden werden. Dabei wird deutlich, dass eine initiale Variablenordnung nicht für alle weiteren Berechnungen optimal ist. Das Finden einer idealen Variablenordnung und die Neuordnung (Reordering) der BDDs kann aber mehr Zeit benötigen als die Berechnung mit der falschen Variablenordnung durchzuführen.

In der hier benutzten BDD-basierten Analyse wird die erste Möglichkeit verwendet, da sie ohne großen Berechnungsaufwand bereits gute Ergebnisse für die Testfälle liefert. Bei Instanziierung der für die Analyse benötigten Java-Klassen werden die Variablen in einer Reihenfolge angelegt, welche sowohl auf den Partitionen als auch auf Kontrollflussabhängigkeiten zwischen den Variablen basiert.

5.9.2 Verschränken von Bitvektoren

Bei einer Zuweisung oder einem Vergleich von Bitvektoren, wie sie in dieser Analyse häufig vorkommen, werden meist nur gleiche Bit-Positionen miteinander verrechnet. Aus diesem Grund ist eine Verschränkung der einzelnen Bits von Vektoren in der Variablenordnung von Vorteil. Dadurch wird beispielsweise bei einer bitweisen Zuweisung im BDD eine linear statt exponentiell zur Länge der Bitvektoren wachsende Knotenzahl erreicht. Diese Tatsache wurde schon von Aziz, Taşiran und Brayton [2] im Zusammenhang mit Finite-State-Machines festgestellt.

Hier sei nochmals die Variablenordnung aus den Beispielen 3 und 4 genannt, welche die Zuweisung der Variablen x an die Variable y darstellen. Da jede Variable im Beispiel ein Bitvektor der Länge 3 ist, ergibt sich aus Zuweisung $x = y$ die Formel $(x_0 \Leftrightarrow y_0) \wedge (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$.

Da Zuweisungen nur innerhalb einer Variablenpartition möglich sind, werden die BDD-Variablen jeweils pro Partition bitweise verschränkt. Die Partitionen selbst bzw. deren Variablen werden sequenziell geordnet.

Für zwei Partitionen $P_1 = \{A, B\}$ und $P_2 = \{C\}$ bedeutet dies, falls die Variablen A, B und C durch Bitvektoren der Länge 3 dargestellt werden, folgende Variablenreihenfolge: A_0, B_0, A_1, B_1, A_2, B_2, C_0, C_1, C_2.

5.9.3 Kontrollflussabhängigkeiten

Bei manchen Kontrollflussstrukturen führt eine ungünstige Variablenordnung zu einer sehr großen Knotenzahl im BDD und verlangsamt dadurch die BDD-Operationen. Dies kann vermieden werden, wenn bei der Variablenordnung die grundlegende Kontrollstruktur des Programms beachtet wird. Obwohl die Variablen innerhalb einer Variablenpartition bereits verschränkt sind und somit eine relativ feste Position besitzen, kann die Reihenfolge der Partitionen an die Kontrollflussstruktur angepasst werden.

Hierfür werden Abhängigkeiten zwischen Partitionen betrachtet. Die Abhängigkeiten können durch einen gerichteten Graphen dargestellt werden, dessen Knoten die Partitionen und dessen Kanten die Abhängigkeiten sind. Eine Abhängigkeit zwischen zwei Partitionen P_1 und P_2 besteht genau dann, wenn eine Variable aus P_1 in einer Bedingung und eine Variable aus P_2 in einer darauf (direkt oder indirekt) folgenden Anweisung verwendet wird, welche nur in einem Pfad des CFAs ausgehend von der Verzweigung der Bedingung vorhanden ist. Sollte zwischen ersten Bedingung und zweiten Anweisung im CFA eine weitere Bedingung sein, wird diese Abhängigkeit ignoriert. Auf diesem Abhängigkeitsgraphen wird eine einfache Tiefensuche durchgeführt, um die Reihenfolge der Partitionen zu bestimmen.

6 Empirische Untersuchung des kombinierten Ansatzes mit verschiedenen Konfigurationen

Im Folgenden wird die Kombination aus Expliziter und BDD-basierter Analyse anhand einiger Testfälle mit der Expliziten Analyse verglichen. Zusätzlich werden verschiedene Konfigurationen der kombinierten Analyse untersucht und daraus resultierende Unterschiede im Verhalten der Analysen gezeigt. Es wird vor allem auf die Anzahl der erreichten Zustände und die Laufzeit (Prozessorzeit) der Analysen eingegangen.

6.1 Testbedingungen

Die einzelnen Tests wurden auf einem Rechner ausgeführt, der einen Intel(R) Core(TM) i7-2600 Prozessor mit 3.4 GHz und 16 GB Arbeitsspeicher besitzt. Für jeden Test wurde eine Zeitgrenze von 10 Minuten (600 Sekunden) und ein Speicherlimit von 15000 MB gesetzt. Der Heap für die Java Virtual Machine wurde auf 12500 MB beschränkt. Als Betriebssystem wurde Ubuntu 12.04 (64-bit) mit der Kernelversion 3.2.0 verwendet. Die installierte Java-Version war OpenJDK 1.6.0 (IcedTea6 1.11.4). Es wurde CPAchecker in Revision 6653 benutzt.

Die Ausgabe von CPAchecker wurde durch die Option „-noout“ deaktiviert. Durch Angabe der Option „-stats“ wurden Statistiken über die verschiedenen Analysen erstellt, um z. B. festzustellen, wie viele Zustände während einer Analyse erstellt werden.

6.2 Konfigurationen

Es wurden verschiedene Tests durchgeführt, bei denen jeweils eine andere Konfiguration für die Analysen benutzt wurde. In allen Tests werden die nicht mit einem BDD behandelten Variablen von der Expliziten Analyse untersucht.

6.2.1 Explizite Analyse (`ex`)

Die reine Explizite Analyse dient als Vergleichsanalyse und wird in den Tabellen als `ex` geführt. Diese Analyse führt einen Algorithmus zur Tiefensuche auf dem CFA aus. Alle weiteren Konfigurationen verwenden aufgrund des Merge-Verhalten der BDD-basierten Analyse eine Breitensuche auf dem CFA.

Die Explizite Analyse führt bei Erreichen eines Fehlerzustandes im CFA zusätzlich eine Überprüfung des Fehlerpfades mit den Programm CBMC durch, um falsche Ergebnisse zu verhindern. Wenn CBMC einen Fehlerpfad weder bestätigen noch ausschließen kann, bricht die Explizite Analyse die Verifikation ohne Resultat ab.

6.2.2 Explizite Analyse mit BDD-Analyse boolescher Variablen (`ex-bdd-bool`)

Bei dieser Kombination von Expliziter und BDD-basierter Analyse werden nur die booleschen Variablen mit BDDs behandelt. Sowohl die diskreten Variablen als auch alle übrigen werden explizit analysiert.

6.2.3 Explizite Analyse mit BDD-Analyse boolescher und diskreter Variablen (`ex-bdd-no-sc`)

Die BDD-basierte Analyse übernimmt hier die Behandlung der booleschen und diskreten Variablen. Die restlichen Variablen werden von der Expliziten Analyse übernommen.

6.2.4 Explizite Analyse mit BDD-Analyse einfacher Berechnungen (`ex-bdd`)

Alle Variablen, welche nur in einfachen Berechnungen verwendet werden, behandelt die BDDCPA. Diese Variablenklasse umfasst auch boolesche und diskrete Variablen.

6.3 Ergebnisse der Untersuchung

Die Benchmarks werden mit einer Menge an Quellcodedateien aus dem Testverzeichnis von CPAchecker durchgeführt, deren Dateinamen das jeweils zu erwartende Ergebnis (SAFE bzw. UNSAFE) beinhaltet. Der Quellcode enthält nur Variablen vom Typ *int*, es gibt keine Zugriffe auf Structs oder Pointer. Die Dateien werden aufgrund der verschiedenen Kontrollflüsse in die 3 Gruppen

- locks
- eca-simplified
- ssh-simplified und ntdrivers-simplified

eingeteilt und entsprechend getrennt behandelt.

Die angegebenen Zeiten sind ein Richtwert. Es kommt nicht auf den exakten Wert, sondern vor allem auf die entsprechende Größenordnung an.

6.3.1 locks

Die Dateien dieser Gruppe enthalten eine relativ einfache Kontrollstruktur, welche sich zwischen den Dateien nur in der Länge unterscheidet. Die hier gegebene Kontrollstruktur verursacht bei der Expliziten Analyse eine exponentielle Anzahl an Zuständen in Abhängigkeit zur Programmlänge, wie aus Tabelle 1 ersichtlich wird. Eine vollständig bdd-basierte Analyse benötigt nur linear viele Zustände. Diese Tatsache wirkt sich unmittelbar auf die Laufzeit der Analysen aus.

Alle Dateien in Tabelle 1 wurden entweder korrekt verifiziert oder das Zeitlimit wurde überschritten („TO“).

Die Dateien enthalten zwischen 11 und 31 boolesche oder diskrete Variablen, weitere Variablen gibt es nicht. Damit erübrigt sich die Konfiguration `ex-bdd`, bei der auch Variablen aus einfachen Berechnungen mit BDDs behandelt werden, welche nicht boolesch oder diskret sind. Bei diesen Testfällen liefert sie die gleichen Ergebnisse wie die Konfiguration `ex-bdd-no-sc`.

Obwohl bei diesen Dateien die Konfiguration `ex-bdd-no-sc` die besten Ergebnisse bzgl. Laufzeit und Anzahl der Zustände liefert, wird darauf hingewiesen, dass mit dieser Konfiguration die Explizite Analyse keine Variablen behandelt, sondern alle Variablen automatisch rein BDD-basiert analysiert werden.

	`ex`		`ex-bdd-bool`		`ex-bdd-no-sc`	
	Zeit	Zustände	Zeit	Zustände	Zeit	Zustände
`locks_5_safe`	✓ 1,5	749	✓ 1,5	344	✓ 1,4	79
`locks_6_safe`	✓ 1,8	1697	✓ 1,8	668	✓ 1,4	92
`locks_7_safe`	✓ 2,0	3813	✓ 2,3	1312	✓ 1,4	105
`locks_8_safe`	✓ 2,5	8489	✓ 3,0	2596	✓ 1,4	118
`locks_9_safe`	✓ 3,6	18733	✓ 3,6	5160	✓ 1,4	131
`locks_10_safe`	✓ 6,8	41009	✓ 5,3	10284	✓ 1,5	144
`locks_11_safe`	✓ 25	89141	✓ 8,7	20528	✓ 1,4	157
`locks_12_safe`	✓ 120	192569	✓ 20	41012	✓ 1,4	170
`locks_13_safe`	✓ 600	413757	✓ 60	81976	✓ 1,4	183
`locks_14_safe`	`TO` 600	-	✓ 280	163900	✓ 1,4	196
`locks_14_unsafe`	✓ 160	41156	✓ 270	163901	✓ 1,4	197
`locks_15_safe`	`TO` 600	-	`TO` 600	-	✓ 1,4	209
`locks_15_unsafe`	`TO` 600	-	`TO` 600	-	✓ 1,5	210

Tabelle 1: Ergebnisse der Tests der Dateien locks mit verschiedenen Konfigurationen, Dateien ohne Timeout (TO) wurden korrekt verifiziert, die angegebene Zeit ist die benötigte CPU-Zeit in Sekunden

6.3.2 eca-simplified

Event-Condition-Action-Systeme (ECA) werden in der Industrie beispielsweise verwendet, wenn mit bestimmten Eingaben unter gegebenen Bedingungen eine Aktion durchgeführt werden soll. Bei den hier vorliegenden Dateien handelt es sich um modifizierte Programme, welche von der RERS Grey Box Challenge 2012 [4] stammen. Die ursprünglichen Quellcodes enthalten einfache Pointerzugriffe, welche in den modifizierten Dateien entfernt wurden.

Die Dateien enthalten zwischen 26 und 61 diskrete Variablen und eine unbenutzte boolesche Variable, weitere Variablen gibt es nicht. Damit liefert die Analyse `ex-bdd` die gleichen Ergebnisse wie die Konfiguration `ex-bdd-no-sc`. Weil es nur eine ungenutzte boolesche Variable in jedem Testfall gibt, liegt der Unterschied zwischen `ex` und `ex-bdd-bool` hauptsächlich in der Tiefen- bzw. Breitensuche, welche während der Analyse auf dem CFA durchgeführt werden. Die Analyse `ex-bdd-no-sc` erreicht

[4] http://leo.cs.tu-dortmund.de:8100/isola2012/index.html

nur einen Bruchteil der Zustände im Vergleich zu `ex` und benötigt daher weniger Laufzeit.

Obwohl bei diesen Dateien die Konfiguration `ex-bdd-no-sc` die besten Ergebnisse bzgl. Laufzeit und Zustandsmenge liefert, muss beachtet werden, dass auch in diesem Fall die Explizite Analyse keine Variablen behandelt, sondern alle Variablen automatisch nur BDD-basiert analysiert werden.

	ex		ex-bdd-bool		ex-bdd-no-sc	
	Zeit	Zustände	Zeit	Zustände	Zeit	Zustände
P1_00_simple_safe	✓ 6,6	61440	✓ 9,0	61440	✓ 4,5	1403
P1_44_simple_unsafe	✓ 3,0	5808	✓ 4,4	12684	✓ 3,6	943
P2_05_simple_safe	✓ 4,5	35992	✓ 6,0	35992	✓ 3,7	1016
P2_50_simple_unsafe	✓ 3,6	16003	✓ 5,0	21794	✓ 3,4	955
P3_12_simple_safe	✓ 16	159907	✓ 24	159907	✓ 8,0	3197
P3_13_simple_unsafe	✓ 13	126668	✓ 11	68868	✓ 7,2	3038
P4_44_simple_safe	✓ 270	1564952	✓ 390	1564953	✓ 31	12895
P4_45_simple_unsafe	? 81	757892	✓ 250	1105634	✓ 25	12728
P5_03_simple_safe	✓ 540	3959429	TO 600	-	✓ 32	19815
P5_30_simple_unsafe	✓ 400	3047196	TO 600	-	✓ 31	19677
P6_27_simple_unsafe	✓ 310	2064656	✓ 560	2070764	✓ 26	17561
P6_28_simple_safe	TO 600	3225255	TO 600	-	✓ 29	17744

Tabelle 2: Ergebnisse der Tests der Dateien eca-simplified mit verschiedenen Konfigurationen, die angegebene Zeit ist die benötigte CPU-Zeit in Sekunden

6.3.3 ssh-simplified und ntdrivers-simplified

Die Dateien ssh-simplified und ntdrivers-simplified sind vereinfachte Programme und Treiber von Betriebssystemen. Die Dateien enthalten sowohl boolesche und diskrete Variablen, einfache Berechnungen als auch Variablen, welche nicht Bestandteil oben genannter Variablenklassen sind und daher immer mit der Expliziten Analyse behandelt werden müssen.

In fast allen Dateien von ssh-simplified kommen zwischen 22 und 29 boolesche Variablen und zwischen 11 und 19 diskrete Variablen vor, Es gibt 2 vereinfachte Programme (`s3_srvr_1a_safe.cil.c` und `s3_srvr_1b_safe.cil.c`), welche insgesamt nur 10 Variablen und damit weniger boolesche und diskrete Variablen enthalten.

Die Dateien ntdriver-simplified sind größer als ssh-simplified und enthalten bis zu 220 boolesche und bis zu 200 diskrete, nicht boolesche Variablen. Die Dateien enthalten sowohl einfache als auch schwerere Berechnungen wie Addition bzw. Multiplikation. Variablen einer Multiplikation können nicht mit BDDs behandelt werden, sondern müssen explizit analysiert werden. Die Konfiguration `ex-bdd` ermöglicht es, Additionen mit BDD-Operationen zu simulieren. Damit ergeben die oben genannten Konfigurationen für diese Dateien verschiedene Ergebnisse bzgl. der Anzahl an erreichbaren Zuständen. In Tabelle 3 sind die Ergebnisse der unterschiedlichen Konfigurationen angegeben.

Mit der Konfiguration `ex-bdd-no-sc` ergibt sich ein falsches Verifikationsergeb-

nis bei der Datei `cdaudio_simpl1_safe.cil.c`. Dies liegt an der speziellen Struktur des Programms und einem Schwachpunkt in der Expliziten Analyse. Eine Begründung hierfür wird in Abschnitt 7.1.2 gegeben.

Die Konfiguration `ex-bdd` benötigt, obwohl sie von allen Konfigurationen die wenigsten Zustände erreicht, in vielen Fällen mehr Laufzeit als `ex-bdd-no-sc`. Bei einigen Dateien braucht `ex-bdd` sogar wesentlich länger als andere Analysen. Die Ursache hierfür liegt in den aufwändigeren BDD-Operationen, welche bei der BDD-basierten Simulation einer Addition auftreten. Die Explizite Analyse produziert zwar mehr Zustände, kann aber durch schnellere Operationen Laufzeit sparen. Des Weiteren kann bei einer steigenden Anzahl an Variablen die falsche Variablenordnung ein große Anzahl an BDD-Knoten verursachen.

	ex			ex-bdd-bool			ex-bdd-no-sc			ex-bdd		
		Zeit	Zust.		Zeit	Zust.		Zeit	Zust.		Zeit	Zust.
s3_clnt_1_safe	✓	7,8	41574	✓	3,7	6265	✓	2,8	579	✓	3,1	421
s3_clnt_1_unsafe	✓	3,8	14954	✓	2,4	1831	✓	1,9	444	✓	2,1	305
s3_clnt_2_safe	✓	7,7	41549	✓	3,7	6720	✓	2,7	577	✓	3,1	420
s3_clnt_2_unsafe	✓	3,8	14953	✓	2,3	1410	✓	2,0	446	✓	2,2	307
s3_clnt_3.cil_safe	✓	28	101093	✓	3,5	5954	✓	2,6	430	✓	3,0	404
s3_clnt_3_safe	✓	7,4	41678	✓	3,7	6730	✓	3,0	658	✓	2,9	458
s3_clnt_3_unsafe	✓	3,6	15012	✓	2,4	1567	✓	2,0	521	✓	2,1	341
s3_clnt_4_safe	✓	7,8	41549	✓	3,7	6847	✓	2,7	578	✓	3,1	423
s3_clnt_4_unsafe	✓	3,7	14953	✓	2,3	1536	✓	1,9	445	✓	2,2	307
s3_srvr_10_unsafe	✓	1,6	170	✓	1,6	204	✓	1,6	204	✓	1,9	163
s3_srvr_11_unsafe	✓	2,1	1498	✓	3,1	5052	✓	2,4	531	✓	2,8	399
s3_srvr_12_unsafe	✓	1,9	1048	✓	3,2	5009	✓	2,3	589	✓	2,7	403
s3_srvr_13_unsafe	✓	2,0	1228	✓	1,9	866	✓	1,9	432	✓	2,1	318
s3_srvr_14_unsafe	✓	2,4	3925	✓	1,7	347	✓	1,9	256	✓	1,8	195
s3_srvr_1_safe	✓	2,4	5363	✓	5,1	14152	✓	2,9	560	✓	3,7	423
s3_srvr_1_unsafe	✓	1,6	375	✓	1,8	640	✓	2,0	400	✓	2,1	289
s3_srvr_1a_safe	✓	1,4	318	✓	1,7	358	✓	1,5	358	✓	2,1	127
s3_srvr_1b_safe	✓	1,3	119	✓	1,3	100	✓	1,4	100	✓	1,5	65
s3_srvr_2_safe	✓	2,4	5138	✓	4,8	13170	✓	2,9	556	✓	3,6	419
s3_srvr_2_unsafe	✓	1,6	377	✓	1,8	636	✓	1,9	399	✓	2,0	290
s3_srvr_3_safe	?	2,5	5189	✓	5,0	13752	✓	3,0	557	✓	3,6	420
s3_srvr_4_safe	?	2,6	5177	✓	4,7	13049	✓	2,8	554	✓	3,6	417
s3_srvr_6_safe	✓	250	284031	✓	69	132734	✓	3,4	596	✓	4,0	456
s3_srvr_6_unsafe	✓	250	284035	✓	1,8	119	✓	1,7	119	✓	1,7	119
s3_srvr_7_safe	✓	170	235212	✓	22	65974	✓	3,0	570	✓	3,9	431
s3_srvr_8_safe	✓	2,6	5374	✓	4,9	13356	✓	3,0	562	✓	3,6	425
cdaudio_simpl1_safe	✓	3,4	10159	✓	5,5	9990	X	4,6	7759	TO	600	-
cdaudio_simpl1_unsafe	✓	2,8	3198	✓	5,3	9985	✓	4,6	7759	TO	600	-
floppy_simpl3_safe	✓	2,6	7066	✓	4,2	6950	✓	3,7	4468	✓	3,5	1088
floppy_simpl3_unsafe	✓	2,1	1725	✓	4,0	6705	✓	3,5	4319	✓	3,5	1082
floppy_simpl4_safe	✓	2,7	9301	✓	4,7	9120	✓	4,0	6158	✓	440	1601
floppy_simpl4_unsafe	✓	2,4	2849	✓	4,6	8744	✓	4,0	5972	✓	440	1597
kbfiltr_simpl1_safe	✓	2,1	3706	✓	3,2	3706	✓	2,2	1179	✓	2,5	444
kbfiltr_simpl2_safe	✓	2,5	7179	✓	4,3	6539	✓	3,2	2552	✓	2,8	872
kbfiltr_simpl2_unsafe	✓	2,0	1458	✓	4,2	6498	✓	3,2	2523	✓	2,7	874

Tabelle 3: Ergebnisse der Tests der Dateien ssh-simplified und ntdrivers-simplified mit verschiedenen Konfigurationen, die jeweils erste Spalte jeder Konfiguration gibt das Ergebnis der Verifikation an (korrekt = ✓, unbekannt = `?`, falsch = `X`, Timeout = `TO`), die angegebene Zeit ist die benötigte CPU-Zeit in Sekunden

6.4 Zusammenfassung der Ergebnisse

Die Ergebnisse der Testläufe zeigen, dass die Explizite Analyse in vielen Fällen ein Vielfaches an Zuständen erreicht und oft eine längere Laufzeit benötigt als die Kombination mit der BDD-basierten Analyse. In einzelnen Fällen liefert die Explizite Analyse jedoch schneller ein Ergebnis. Insgesamt überwiegen aber die Vorteile der BDD-basierten Analyse.

Dies kann auch an den Abbildungen 7 und 8 nachvollzogen werden, in denen die korrekt verifizierten Dateien jeweils entsprechend der Anzahl an Zuständen bzw. der Laufzeit sortiert sind. Die Graphen für die Konfigurationen `ex-bdd-no-sc` und `ex-bdd` liegen immer unterhalb des Graphen der Expliziten Analyse. Die Konfiguration `ex-bdd-bool` von Expliziter und BDD-basierter Analyse liefert zwar weniger Zustände als die Explizite Analyse, sie benötigt aber teilweise mehr Zeit für ein Resultat.

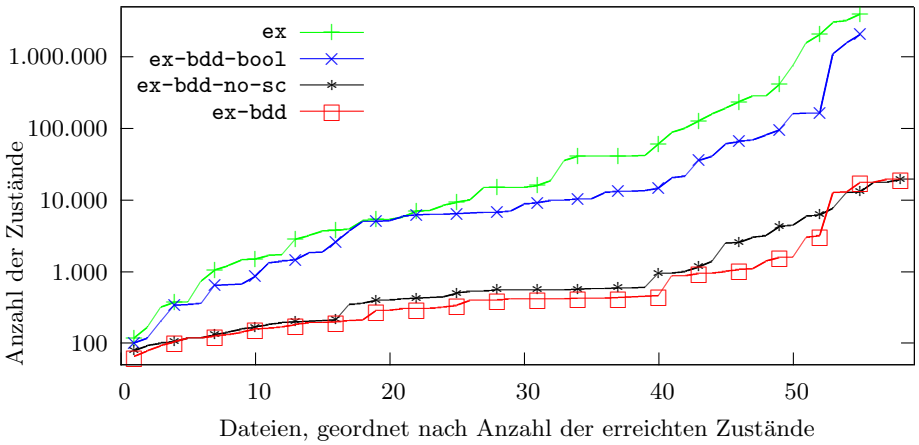

Abbildung 7: Quantilgraph der Dateien mit Anzahl der erreichten Zustände

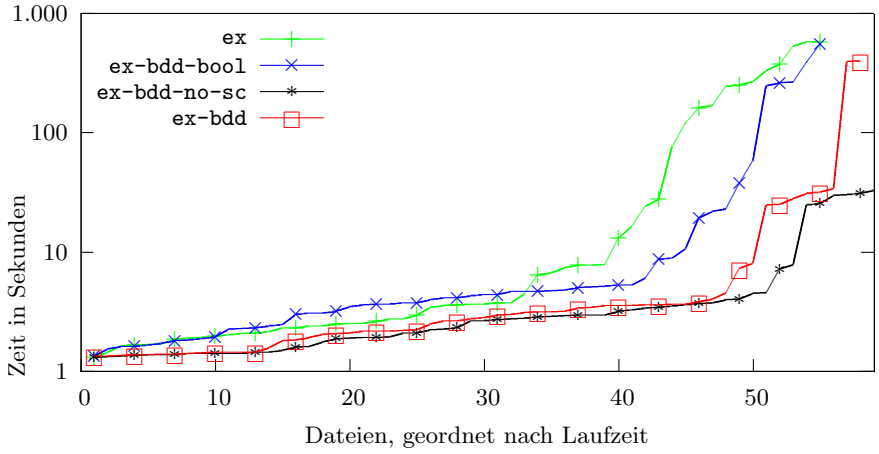

Abbildung 8: Quantilgraph der Dateien mit Laufzeit

7 Fazit

Die Kombination der Expliziten und BDD-basierten Analyse mit automatischer Typenerkennung bei Variablen liefert in vielen Fällen weniger erreichbare Zustände und bessere Zeiten als die Explizite Analyse alleine.

Es muss der Arbeitsaufwand der einzelnen Analysen betrachtet werden, da dieser für einen Großteil der Laufzeit verantwortlich ist. Der Einsatz der BDD-basierten Analyse verursacht ein geringere Zustandsanzahl im ARG im Vergleich zur reinen Expliziten Analyse. Jedoch wird bei schwierigeren BDD-Operationen die Laufzeit der BDD-basierten Analyse Analyse größer. Daher muss je nach Programm abgewogen werden, wie viele Variablen mit BDDs bzw. explizit untersucht werden sollen.

Die Kombination der Analysen hat außerdem weitere Schwierigkeiten gezeigt, welche das Resultat einer Verifikation beeinflussen können.

7.1 Schwierigkeiten verschiedener Analysen

In CPAchecker kann eine Analyse oder eine Kombination verschiedener Analysen unter bestimmten Umständen einen Fehlerzustand erreichen, der bei Ausführung des Quellcodes eigentlich niemals erreichbar ist.

Da von jeder einzelnen Analyse Soundness garantiert wird, führt dieses Verhalten aber nicht dazu, dass ein Fehlerzustand „übersehen“ wird. Soundness bedeutet, dass eine Analyse eine Fehlerpfad nur dann ausschließt, wenn dieser beweisbar unerreichbar ist. Damit auch die Kombination von Analysen sound ist, wird von der CompositeCPA bei der Kombination der Ergebnisse der einzelnen Analysen ein abstrakter Zustand erzeugt, welcher eine Obermenge der tatsächlichen Zustände darstellt.

Wenn aber keine Analyse einen vollständigen Beweis für die Unerreichbarkeit einer Fehlerposition hat, kann es sein, dass eventuell ein unmöglicher Fehlerzustand für erreichbar gehalten wird.

7.1.1 Verschiedene Pfade im Erreichbarkeitsgraphen

Jede CPA entscheidet für sich alleine, wie neue Zustände erstellt werden. Da jede Analyse einen eigenen Merge-Operator hat, kann es sein, dass bei einer Kombination zweier Analysen mit $merge^{join}$-Operator jede Analyse zwar im ARG einen eigenen Pfad zum gleichen Fehlerzustand gefunden hat, diese Pfade selbst müssen aber nicht identisch sein. Dadurch wird für ein entsprechendes Programm, welches eigentlich fehlerfrei ist, das Resultat UNSAFE ausgegeben.

Dieses Verhalten wurde während der Implementierung der BDD-basierten Analyse festgestellt, da anfangs die einzelnen Variablenklassen mit verschiedenen BDD-basierten Analysen getrennt voneinander untersucht wurden. Als Lösung ergab sich die Vereinigung aller BDD-basierten Analysen in einer einzigen CPA.

7.1.2 Defizit der Explizite Analyse bei uninitialisierten Variablen

Die Explizite Analyse kann bei Bedingungen mit einer nicht initialisierten Variablen im Allgemeinen keine Aussage darüber treffen, welchen Wert eine Variable nach der Auswertung der Bedingung hat. Wenn bei einem späteren Lesezugriff auf diese Variable wieder kein Wert vorhanden ist, kann eine spätere, ähnlich aufgebaute Bedingung ebenfalls nicht eindeutig berechnet werden. Dieses Verhalten gibt es nicht nur bei der reinen Expliziten Analyse, sondern auch in jeder Kombination der Expliziten Analyse mit einer anderen Analyse. Dieser Fall wird in Datei `cdaudio_simpl1_safe.cil.c` aus Abschnitt 6.3.3 sichtbar.

Eine mögliche Lösung zum Ausschluss des Fehlerzustandes wäre die weitere Untersuchung des Fehlerpfades im ARG mit einer weiteren Analyse. Dabei kann auf bereits berechnete Zwischenergebnisse zurückgegriffen werden.

Die Konfiguration `ex` verwendet zur Kontrolle jedes Fehlerpfades eine Analyse durch CBMC, damit die Anzahl der falsch erkannten Fehler gering bleibt. Diese Option könnte durch den modularen Aufbau von CPAchecker auch für die kombinierte Analyse aktiviert werden.

7.2 Zusätzliches Optimierungspotential

Die in der BDD-basierten Analyse benutzte Variablenordnung wird relativ einfach erstellt, da nur wenige Abhängigkeiten beachtet werden. Weil die Geschwindigkeit dieser Analyse stark davon abhängt, kann durch eine bessere Sortierung der Variablen die Laufzeit der kombinierten Analyse möglicherweise weiter gesenkt werden.

Des Weiteren ist es nicht notwendig, jede mögliche Variable mit BDDs zu behandeln, sondern es kann durchaus sinnvoll sein, auch boolesche Variablen explizit zu untersuchen.

7.3 Schlusswort

Diese Arbeit zeigt Möglichkeiten einer Kombination von grundsätzlich verschiedenen Analysen für das Model-Checking. Dabei werden Variablen automatisch mit der dafür jeweils am besten geeigneten Analyse untersucht.

Aus den durchgeführten Untersuchungen wird ersichtlich, dass durch die Kombination der Analysen in vielen Fällen das Problem einer möglichen Zustandsexplosion verhindert und zugleich die Laufzeit der Analyse verringert werden kann.

Beim Model-Checking ist die Kombination unterschiedlicher Analysen ein vielversprechender Weg, um die steigende Komplexität von Software handhaben und gleichzeitig eine Beschleunigung der Verifikation erreichen zu können.

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

Literatur

[1] S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 27(6):509–516, June 1978.

[2] Adnan Aziz, Serdar Taşiran, and Robert K. Brayton. BDD Variable Ordering for Interacting Finite State Machines. In *Proceedings of the 31st annual Design Automation Conference*, DAC '94, pages 283–288, New York, NY, USA, 1994. ACM.

[3] Dirk Beyer. Competition on Software Verification (SV-COMP). In C. Flanagan and B. König, editors, *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS 2012, Tallinn, Estonia, March 27-30)*, LNCS 7214, pages 504–524. Springer-Verlag, Heidelberg, 2012.

[4] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007, Berlin, July 3-7)*, LNCS 4590, pages 504–518. Springer-Verlag, Heidelberg, 2007.

[5] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Program Analysis with Dynamic Precision Adjustment. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008, L'Aquila, September 15-19)*, pages 29–38. IEEE Computer Society Press, Los Alamitos (CA), 2008.

[6] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011, Snowbird, UT, July 14-20)*, LNCS 6806, pages 184–190. Springer-Verlag, Heidelberg, 2011.

[7] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[8] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, September 1992.

[9] David Déharbe and Jorgiano Márcio Bruno Vidal. Optimizing BDD-Based Verification Analysing Variable Dependencies. In *Proceedings of the 14th symposium on Integrated circuits and systems design*, SBCCI '01, pages 64–69, Washington, DC, USA, 2001. IEEE Computer Society.

[10] C.Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[11] Alexander von Rhein, Sven Apel, and Franco Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In *Proc. Java Pathfinder Workshop*, 2011.

[12] Alexander von Rhein, Sven Apel, and Franco Raimondi. On-the-fly Hybrid Model Checking for Software Verification. 2012.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich habe die Arbeit nicht in gleicher oder ähnlicher Form bei einer anderen Prüfungsbehörde vorgelegt.

Passau, 11. Oktober 2012, Friedberger Karlheinz