Verifikation von systemnaher Software mittels Bounded Model Checking

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik

der Universität Fridericiana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Hendrik Post

geboren am 26.3.1980 in Münster

Tag der mündlichen Prüfung: 24.04.2009
Erster Gutachter: Dr. rer. nat. C. Sinz, Universität Karlsruhe (TH)
Zweiter Gutachter: Prof. Dr. sc. techn. W. Küchlin, Universität Tübingen

Vorwort

Die vorliegende Arbeit wurde im Rahmen eines Promotionsstipendiums der Stiftung der Deutschen Wirtschaft an der Universität Tübingen begonnen. Abgeschlossen wurde Sie schließlich im Rahmen der Tätigkeit als wissenschaftlicher Mitarbeiter an der Universität Karlsruhe.

Ich möchte mich zunächst beim den Initiatoren der Arbeit, Herrn Professor Küchlin sowie Herrn Dr. Sinz bedanken. Ihre Vision vom Einsatz formaler Methoden zur Verifikation von Teilen eines echten Betriebssystems haben die Faszination ausgeübt, die mich bis zum Schluss der Arbeit motiviert und angetrieben hat. Ich möchte mich bei beiden für die hervorragende fachliche und persönliche Betreuung bedanken. Ich danke ebenso Herrn Dr. Bündgen für seine fachliche Betreuung und die Organisation und Betreuung meines Praktikums bei IBM Deutschland, Forschung und Entwicklung GmbH.

Ich möchte auch Herrn Professor Kaufmann und Frau Dr. Zweig danken, die mir durch die außergewöhnlich gute Betreuung meiner Diplomarbeit den Weg in das wissenschaftliche Arbeiten gewiesen haben. Ich danke auch Herrn Gorges von der Robert Bosch GmbH, der durch seinen Einsatz die Erstellung industrieller Fallstudien ermöglicht hat. Herrn Professor Kropf, ebenfalls von der Robert Bosch GmbH Leonberg, möchte ich danken für die Ermöglichung zweier Diplomarbeiten, die zu dieser Arbeit beigetragen haben.

Herrn Dr. Kroening, Universität Oxford, danke ich für die wissenschaftliche Grundsteinlegung, auf der meine Arbeit aufsetzt. Ebenso danke ich für die Erstellung und die Unterstützung beim Einsatz des Werkzeugs CBMC, das weitreichende Anwendung in dieser Arbeit findet.

Ich danke Herrn Professor Podelski, Herrn Dr. Kroening und Herrn Professor Biere für die Einladungen an Ihre Lehrstühle und für wertvolle Anregungen.

Mein Dank gebührt der Stiftung der Deutschen Wirtschaft, die in der ersten Phase der Promotion finanzielle und ideelle Förderung leistete. Im Besonderen danke ich Herrn Hülshörster für die hervorragende Begleitung meines Stipendiums in den Jahren 2002 bis 2008. Ich danke den Initiatoren der Exzellenzinitiative des Bundes und der Länder zur Förderung von Wissenschaft

und Forschung an deutschen Hochschulen in dessen Rahmen der zweite Teil meiner Promotion an der TU Karlsruhe gefördert wurde.

Ich danke meinen Studenten Friedrich Meissner, Matthias Sauter, Alexander Kaiser und Florian Merz für die Erstellung ihrer hervorragenden Diplomarbeiten. Ebenso danke ich Herrn Daniel Harbarth und Herrn Manuel Meiborg für ihre Studienarbeiten.

Zu guter Letzt möchte ich meinen Eltern danken, die mir meine langjährige Ausbildung ermöglicht haben, und ohne die ich niemals den Weg, den ich bestritten habe, hätte einschlagen können. Weiter möchte ich von ganzem Herzen Amalinda danken, die durch ihre persönliche Unterstützung, wie auch fachliche Diskussionen und Beiträge, die Zeit meiner Promotion bereichert und verschönert hat.

Inhaltsverzeichnis

Einleitung 1

Die Analyse von Software-intensiven Systemen hat eine – für die Verhältnisse der Informatik – lange Geschichte. Eine zentrale Person hierbei ist Alan Turing, der bekannt ist für seine wissenschaftlichen Ergebnisse, die der Analysierbarkeit von programmierbaren Maschinen harte Grenzen aufzeigen.

Turing entwarf zur Analyse ein einfaches Modell eines Computers: die Turingmaschine. Er bewies, dass es nicht in allen Fällen möglich ist festzustellen, ob eine solche Maschine bei der Ausführung eines Programms immer zu einem Ende kommt, also *anhält* oder *terminiert*. Die Frage, ob eine Turingmaschine auf einem Programm immer anhalten wird, nennt man das *Halteproblem*. Durch seinen Beweis, dass es im allgemeinen nicht möglich ist das Halteproblem zu lösen [Tur36] setzte Turing eine der entscheidenden Grenzen für die Analysierbarkeit von Software-betriebenen Systemen.

Trotz des Negativresultats war Turing auch einer der ersten, die ein einfaches Verfahren zur Analyse von Programmen vorschlug. In seiner Publikation *Checking a large Routine* [Tur49] begründet er somit die Anfänge der formalen Programmanalyse. Turing setzt also die Grenzen ebenso wie den ersten Meilenstein, den es zu übertreffen gilt.

In dieser Einleitung klären wir zunächst abstrakt Inhalte, Ziele und Methode dieser Arbeit: Wir erklären warum es zunehmend wichtiger ist, die Fehlerfreiheit von Programmen und Software sicherzustellen. In den folgenden einführenden Abschnitten klären wir die Begriffe *systemnahe Software* und *Verifikation*. Auf die Klärung des Untersuchungsgegenstandes folgt eine Beschreibung und Motivierung der gewählten Methode des Bounded Model Checking. Wir ordnen kurz die Methode in den wissenschaftlichen Kontext ein und beschreiben das wissenschaftliche Ziel der Arbeit.

Das Ziel der Verifikation von systemnaher Software ist die Steigerung des Vertrauens in die Zuverlässigkeit.

1.1 Motivation

Zu Zeiten Alan Turings waren programmierbare Systeme auf wenige Großrechenmaschinen beschränkt. Heute sind viele Gegenstände des täglichen Lebens, wie auch viele Geschäfts- und sicherheitskritische Systeme, mit Soft-

ware programmierbar. Fehler in Softwaresystemen können Geschäftsprozesse in Unternehmen zum Erlahmen oder sogar wichtige Unternehmensdaten in Gefahr bringen. Neben den im Geschäftssektor häufig eingesetzten Großrechnern, finden sich auch zunehmend viele eingebettete Systeme in Mobiltelefonen, Autos und Flugzeugen. Ein modernes Auto enthält dutzende kommunizierende Teilsysteme mit einem eigenen Prozessor. Ein Fehler in einem Programm kann also großen finanziellen und körperlichen Schaden zur Folge haben, und die Abhängigkeit von Programmen scheint weiter zuzunehmen. Von den vielen Einsatzbereichen von Software ist der Bereich der systemnahen Programme besonders sicherheitskritisch: Arbeiten Betriebssysteme und mit Hardware eng interagierende Systeme nicht korrekt, können auch darauf aufbauende Anwendungen nicht zuverlässig arbeiten.

Die in der Industrie heute eingesetzten Verfahren zur Sicherung von Qualität sind das *Testen* sowie der *Codereview*, also das systematische manuelle Durchlesen des Programms. Wächst die Komplexität eines Programms – beispielsweise die Zahl der Zeilen Quellcodes – steigt der Aufwand bisheriger Qualitätssicherungsmaßnahmen. Eine weit verbreitete Hypothese ist, dass bisherige Methoden an der zunehmenden Komplexität aus wirtschaftlichen, aber auch aus technischen Gründen, scheitern werden. Ein denkbarer Ausweg ist die Verifikation von Programmen durch formale Methoden. Wir nehmen an, dass es zumindest wünschenswert ist, eine Alternative zu entwickeln um folgendes Problem beim Testen zu umgehen:

Will man ein Programm testen, werden für die vollständige Prüfung einer Rechenoperation eines 32 Bit Rechners mehr als 4 Milliarden Testfälle benötigt. Bei einer 64 Bit Rechenoperation sind es bereits mehr als $18 * 10^{18}$ Testfälle. Misst man die Komplexität in Zeilen Quellcode muss man zugestehen, dass schon eine zusätzliche Zeile durch eine Fallunterscheidung die Anzahl der Testfälle verdoppeln kann. Zwar kann eine intelligente Auswahl der Testfälle helfen, die Anzahl auf repräsentative Fälle zu beschränken, im Allgemeinen wächst die Anzahl von Testfällen aber exponentiell in der Größe des Programms. Im Gegenzug verringert sich aus wirtschaftlichen Gründen die Zeit, die zum Testen eines Systems zur Verfügung steht. Die Frage ist, ob Verifikation der Komplexität Herr werden kann. Durch den Einsatz von Verifikationstechniken kann die Sicherheit erhöht und bestenfalls Testaufwand eingespart werden.

Die meisten Ansätze zur Verifikation von Programmen basieren auf *formalen Methoden* wie etwa formale Logiken und Beweiskalküle. Ähnlich wie in der Mathematik können so Analysen nahezu beliebig großer Systeme realisiert werden: Die Gültigkeit von Aussagen wie $a - a = 0, a \in \mathbb{N}$ kann man durch algebraische Regeln in wenigen Schritten ableiten. Eine Prüfung durch Testen von Einzelfällen hat hier nur eine Berechtigung zum Auffinden von einfachen Rechenfehlern.

Den genannten Vorteilen stehen jedoch auch Nachteile bei der Anwendung gegenüber. Das allgemeinste Problem ist, dass sich nicht alle Sätze beweisen lassen, sobald der gewählte Formalismus zu aussagekräftig ist. Dies ist eine umgangssprachliche Formulierung des Gödelschen Unvollständigkeitssatzes der, ähnlich der Unentscheidbarkeit des Halteproblems, theoretische Grenzen setzt. Es ist allerdings unklar, ob der Satz konkrete Auswirkungen bei praktischen Fragestellungen hat. In der Praxis ist oft schon die Beschränkung der Laufzeit Hinderungsgrund für komplexe Beweise.

Für die Anwendbarkeit wichtig ist die Frage nach der Automatisierbarkeit der formalen Methoden. Es existieren eine Reihe vollautomatischer Verfahren, beispielsweise für die aussagenlogische Erfüllbarkeitsprüfung (SAT), sowie eine Reihe von interaktiven, also halbautomatischen, Beweissystemen, in denen ein Mensch Teile des Beweises führen muss. Automatische Methoden sind attraktiv, da sie eine höhere Skalierbarkeit durch weniger manuellen, also kostenintensiven, Aufwand versprechen.

Automatische Methoden sind beschränkt auf Formalismen, die einen Algorithmus zur endlichen Berechnung eines Verifikationsproblems für alle Eingaben erlauben. Ob sich diese Formalismen, zum Beispiel das *Bounded Model Checking*, nutzen lassen, um interessante Eigenschaften über nichtakademische Programme berechnen zu können, ist auch Untersuchungsgegenstand dieser Arbeit.

Im Titel wird bereits angedeutet, dass diese Arbeit zum Ziel hat, die Analysemöglichkeiten mittels Bounded Model Checking zu evaluieren und zu verbessern. Eine Definition der Methode Bounded Model Checking wird in Kapitel 2 geliefert. Im folgenden Abschnitt wird zunächst der Untersuchungsgegenstand, also die Begriffe *Verifikation* und *systemnahe Software*, umrissen.

1.2 Allgemeine Einführung

In diesem Abschnitt führen wir den Untersuchungsgegenstand der Arbeit ein.

1.2.1 Über die Verwendung des Begriffes *Verifikation*

Verifizieren ist gleichbedeutend mit der Bestätigung der Richtigkeit durch Überprüfen [SS06]. Es scheint Einigkeit zu herrschen, dass im Kontext von Programmen der Begriff *Verifikation* auf die Prüfung verweist, dass eine Software wie erwartet funktioniert. In der Praxis unterscheidet sich aber das Verständnis der Überprüfung erheblich.

In einer strengen Auslegung muss die Verifikation sicherstellen, dass niemals, das heißt also mit der Sicherheit eines formalen Beweises, Fehler auftreten können. Nun versteht man aber unter einem Softwaresystem sowohl eine formale Beschreibung in einer formalen Sprache (z.B. in einer Modellierungssprache) wie auch eine konkrete Umsetzung (Implementierung). Bei der Verifikation hängt die Korrektheit der Software nur von der Semantik der Programmiersprache und der formalisierten Anforderung (Spezifikation) ab. Ob ein Programm empirisch wie erwartet funktioniert, hängt auch von der Korrektheit der ausführenden Hardware ab. Letzterer Punkt wird in der dieser Arbeit vorausgesetzt und somit wird das Ergebnis einer Verifikation im Sinne der Korrektheitsprüfung als nicht mehr empirisch angesehen, als ein mathematischer Beweis. Eine erfolgreiche Verifikation führt im kantischen Sinne zu einer *analytischen* Aussage *a priori*, deren Inhalt die Korrektheit des Systems bezüglich einer Spezifikation konstatiert. Laut Balzert wird bei der Verifikation folgende Frage beantwortet: "Entspricht die Software den Anforderungen?" (siehe [Bal97]).

Als Gegenpol zur obigen strengen Auslegung der Verifikation tritt häufig der Begriff *Validierung* auf. Frei nach Balzert [Bal97] könnte man Validierung die empirische Überprüfung der, meist in funktionalem Aspekt interpretierten, Richtigkeit verstehen: "Wurde die richtige Software entwickelt?"

Anstatt jedoch eine formal vollständige Überprüfung zu verlangen, ist bei der Validierung eher eine empirische Sicherstellung gemeint. Methoden zur Validierung umfassen das Testen (Unit-Test), können aber im Bereich Qualitätssicherung auch Methoden wie Kundenreviews beinhalten. In einem Versuch der wissenschaftstheoretischen Einordnung würde Validierung eher eine empirische, kritische Überprüfung darstellen (*synthetisch a posteriori*).

Wünschenswert ist es, sowohl Validierung wie auch Verifikation zur Analyse einzusetzen. Beschränkt man den Begriff Validierung auf das Ausführen einzelner Testfälle und nimmt man an, dass eine formale Verifikation alle möglichen Ausführungen umfasst, ergibt sich eine Bandbreite, die mit *Abdeckung* bezeichnet werden kann. Im Rahmen dieser Arbeit wird Verifikation in einigen Fällen vollständige Abdeckung aller möglichen Ausführungen eines Programms erreichen (siehe die AES-Fallstudie in Kapitel 3). In anderen Fällen, beispielsweise in der Verifikation von Linux (Kapitel 4) wird eine Einschränkung vorgenommen. Neben den Ausführungen kann man auch eine Einschränkung der Abdeckung von Spezifikationen vornehmen.

Eine Besonderheit beim Bounded Model Checking ist die Tatsache, dass bei Prüfung einer Software alle Ausführungen nur bis zu einer bestimmten Länge betrachtet werden. Ist die Längenbeschränkung groß genug, erreicht man eine vollständige Abdeckung von Ausführungen. Nähere Erklärungen hierzu finden sich bei der Einführung der Methode in Abschnitt 2.2.7.

In der Verwendung des Begriffes Verifikation werden also Einschränkungen zugestanden. Der Kern unseres Verständnisses von Verifikation bleibt aber, dass alle Aussagen über die Korrektheit einer Software mit Hilfe einer analytischen Methode und nicht durch Ausführung gemacht werden müssen. Hierzu muss das Softwaresystem mittels einer formalen Semantik analysiert werden. Beobachtungen im Sinne eines Tests zählen nicht hierzu, auch wenn diese für die Praxis zur Validierung weiter nützlich bleiben.

1.2.2 Systemnahe Software

Unter systemnaher Software verstehen wir Programme, die in enger Interaktion mit Hardware stehen. Sie wird durch einen Compiler direkt in Maschinencode übersetzt und direkt auf einer physikalischen Hardware ausgeführt. Interpretierte Software oder Systeme wie Java, die Software prinzipiell nur auf einer simulierten beziehungsweise virtuellen Hardware ausführen, bezeichnen wir nicht als systemnah.

Im Rahmen der Arbeit nehmen wir weiter folgende Charakteristiken systemnaher Software an:

1. Nutzung imperativer Programmiersprachen.
2. Hohe Komplexität der Software in dem Sinne, dass keine einfache Abstraktion gefunden werden kann.
3. Die implementierte Funktionalität enthält zumeist keine schon aus theoretischen Gründen harten Problematiken (z.B. die Berechnung algebraischer Probleme oder NP-vollständiger Probleme).

Imperative Programmiersprachen sind die zur Zeit verbreitetste Form zur Implementierung von systemnaher Software. Betriebssysteme, Software für eingebettete Systeme, wie in der Flugzeugindustrie und im Automobilbereich, sowie Implementierungen von laufzeitkritischer Software in Graphikanwendungen sind Beispiele für imperativ implementierte, systemnahe Software.

Der Ausschluss der Verifikation von Programmen, die theoretisch harte Probleme berechnen, ist durch zwei Gründe motiviert. Es ist natürlich mit einfachereren Programmen zu beginnen, da die Softwareverifikation zur Zeit wenig etablierte Standardtechniken kennt. Erst wenn Methoden bekannt sind, um theoretisch einfache Probleme zuverlässig zu lösen, ist eine Erweiterung des Anwendungsfeldes sinnvoll.

Da systemnahe Software häufig strengen Ressourcenbeschränkungen unterliegt, handelt es sich bei den meisten Implementierungen nicht um schwierige

algorithmische Probleme. Ein Treiber eines Betriebssystems soll möglichst geringe Laufzeiten unter wenig Speicherverbrauch leisten. In eingebetteten Systemen sind die Einschränkungen meist noch restriktiver. In Bereichen wie Bildverarbeitung oder der Implementierung von Datenstrukturen zur Verwaltung von Dateisystemen finden sich Ausnahmen - der größte Teil des Quellcodes aus unseren Fallstudien arbeitet auf Eingaben konstanter Größe, die weit unter der eines Dateisystems liegen.

Zusammenfassend kann der Gegenstand der Arbeit beschrieben werden: Die Verifikation von systemnaher Software umfasst die formale, nicht unbedingt erschöpfende, Korrektheitsprüfung einer großen Klasse von Softwaresystemen, die aufgrund ihrer geringen Abstrahierbarkeit, nicht aber aufgrund der Komplexität der von ihnen berechneten Funktionen, eine Herausforderung darstellen.

Im Folgenden konkretisieren wir die Ziele dieser Arbeit.

1.2.3 Ziele der Verifikation

Wir haben nun geklärt, was die Begriffe Verifikation und systemnahe Software bedeuten. In diesem Abschnitt beschreiben wir, welche Eigenschaften oder Spezifikationen wir untersuchen wollen. Informell lassen sich die häufigsten Fragen bezüglich der Korrektheit einer Software wie folgt zusammenfassen:

1. Terminiert die Software (immer)?
2. Berechnet die Software das gewünschte Ergebnis?
3. Ist die Ausführung der Software wohldefiniert?
4. Zeigt eine sequentielle Software das gewünschte Verhalten?
5. Zeigt eine parallele Software das gewünschte Verhalten?

Die Punkte werden nun einzeln erläutert.

Ein Programm heißt *total korrekt*, falls es immer terminiert und die richtigen Ausgaben berechnet. Ohne die Forderung der Termination spricht man von *partieller Korrektheit*. Ein häufiges Beispiel für Software bei der Termination untersucht wird, sind Gerätetreiber (beispielsweise im *Terminator* Projekt [CPR06]), die eng mit dem Betriebssystem interagieren. Kehrt der Programmfluss aus einem Treiber nicht in das aufrufende Betriebssystem zurück, kann eine Blockade entstehen aus der Datenverlust und Nicht-Verfügbarkeit resultieren.

Im Allgemeinen sollen Terminationseigenschaften im Rahmen dieser Arbeit nicht direkt, sondern indirekt untersucht werden: Im Abschnitt 4 finden sich

Beispiele für Regeln, deren Verletzung zu einem Systemstillstand führt. Die Forderung nach einem immerwährenden Fortschritt der Programmberechnung wird reduziert auf das Finden von Fällen, in denen es garantiert keinen Fortschritt mehr geben kann.

Die zweite Frage, ob eine Software das gewünschte Ergebnis berechnet, betrifft so genannte *funktionale Eigenschaften* eines Programms. Im klassischen Sinne werden Programme als Berechnungsmittel mathematischer oder anderer Funktionen verstanden. Hierbei gilt die Analogie, dass ein Programm bei bestimmten Eingaben (z.B. Funktionsparameter einer C Funktion) die richtigen Ausgaben produziert. Eine C Funktion `double sin(double x)` bei einem Argument 2π soll also den Wert `0`, beziehungsweise ein bis auf Rundung äquivalentes Ergebnis, zurückliefern.

Ein Programm ist nicht *wohldefiniert*, falls es grammatikalisch (syntaktisch) oder bezogen auf die Bedeutung (semantisch) nicht eindeutig analysiert werden kann. Die Uneindeutigkeit spielt eine Rolle bei dem Verhältnis von einem *Compiler* (Übersetzer) und einer natürlichsprachlichen Definition der Sprache in einem Standard. Welche Fälle es genau für die Sprache C zu beachten gibt, wird in Abschnitt 4.1.1 technisch erläutert.

Der häufigste Fall für verletzte Wohldefiniertheit ist *implementierungsspezifisches Verhalten* eines Programms: Der Sprachstandard lässt mehrere Möglichkeiten offen und ein Compiler implementiert nur eine hiervon. Man kann eine weitere Unterscheidung treffen, ob die Auswahl syntaktische oder semantische Eigenschaften beeinflusst.

In Kontrast zu implementierungsspezifischem Verhalten kann ein Standard auch eine Möglichkeit vorgeben, die durch verschiedene Compiler unterschiedlich implementiert wird. Auch hier kann man zwischen semantischen und syntaktischen Erweiterungen des Compilers unterscheiden. Tabelle 1.1 listet die vier Möglichkeiten anhand eines Beispiels auf.

Verletzte Wohldefiniertheit ist ein grundlegendes Problem für Verifikationswerkzeuge, da der Begriff Korrektheit erst nach Klärung einzelner Fälle möglich ist: *Ist ein Programm, das den Standard verletzt, aber compiliert und die Anforderungen erfüllt, korrekt?*

Abschließend erläutern wir kurz ein Beispiel, das eine Spracherweiterung durch die *GNU Compiler Collection (GCC)* vorstellt:

```
(1)    x ? : y
(2)    x ? x : y
```

Der Ausdruck (1) ist syntaktisch inkorrekt, da der zweite Operand des `?` Operators ausgelassen wurde. Der GCC Compiler lässt solche Ausdrücke jedoch

Tabelle 1.1: Beispiele für verletzte Wohldefiniertheit in C

	Syntaktisch	Semantisch
Erweiterung (z.B. GNU C)	`x ? : y` (Fehlender 1. Operand) *"Syntatic sugar"*	`int a[(1.99<2.0)?5:6]` (Je nach Version und Compiler sind beide Größen möglich)
Einschränkung (z.B. GNU C)	Designierte Initialisierer (laut CIL[NMRW02] Dokumentation in GNU C nicht voll unterstützt.)	`((char) x) << 1` (Neues, niedrigstes Bit undefiniert) *"Implementierungs-spezifisch"*

zu und übersetzt sie in ein sprachkonformes Äquivalent (2). Die konkrete Implementierung berechnet für den Wert des Ausdrucks `x`, falls `x` wahr ist, und anderenfalls `y`.

Weitere Dokumentation über GCC Spracherweiterungen finden sich in der Beschreibung des Transformations-Werkzeugs CIL [NMRW02]. CIL wird in Kapitel 4 behandelt.

Unter der Verifikation von *Verhaltenseigenschaften* einer Software versteht man die Untersuchung, ob ein Programm Zustandsfolgen enthält, die einem bestimmten Muster folgen. Im Gegensatz zur funktionalen Verifikation sind also auch Zwischenzustände einer Berechnung von Bedeutung. Die Spezifikation von Verhaltensmustern wird häufig in der Form von endlichen Automaten angegeben. Etwa kann ein Programm, welches ein Taktsignal simulieren soll, durch einen Automat mit zwei Zuständen und unbedingten Übergängen formuliert werden.

Alle vier bisher genannten Fragen lassen sich um die Dimension paralleler Programme erweitern. So kann beispielsweise eine funktionale Eigenschaft für einen parallel, zum Beispiel mittels kommunizierender *Threads of Control*, implementierten Sortieralgorithmus geprüft werden. Die Zahl der möglichen Ausführungen steigt nochmals exponentiell an, da jeder Thread möglicherweise am Zuge ist, seinen oder einen globalen Zustand zu ändern.

Im Rahmen dieser Arbeit werden hauptsächlich Verhaltenseigenschaften geprüft. In Ansätzen werden zusätzlich funktionale Eigenschaften untersucht (siehe Abschnitt 3). Die Behandlung paralleler Programme wird in Abschnitt 4.2.4 erörtert.

Nach der Klärung des Gegenstands und der Ziele folgt nun die Wahl der Methoden basierend auf der obigen Abgrenzung.

1.2.4 Wahl der Methode

Eine genauere Erläuterung verschiedener Verfahren zur Verifikation von Software wird in Kapitel 2 gegeben. Im Rahmen der allgemeinen Einleitung wird nur die gewählte Methode beschrieben und anhand der Fragestellung motiviert.

Unter systemnaher Software wird, wie in Abschnitt 1.2.2 beschrieben, ein in einer imperativen Programmiersprache geschriebenes Programm verstanden. Weiter gehen wir davon aus, dass folgende Charakteristika auf systemnahe Software zutreffen:

1. nicht, oder nur teilweise spezifizierte Schnittstellen
2. endlicher Speicherverbrauch, so dass eine Beschränkung der maximalen Größe von Datenstrukturen keine große Einschränkung bedeutet.
3. eine Programmgröße in Zeilen, die keine manuelle Modellierung des Programms in einer anderen Sprache zulässt.
4. vielfache Verwendung von C-spezifischen Programmkonstrukten wie Zeigerarithmetik, Typumwandlungen, `goto`-Anweisungen und modulare Arithmetik.
5. eine Korrektheit, die von dem genauen Ergebnis arithmetischer Operationen abhängt.

Die im Rahmen dieser Arbeit behandelten Softwaresysteme aus dem Bereich Kryptologie, Betriebssysteme und eingebettete Software erfüllen die oben genannten Einschränkungen. Programme, die die obigen Eigenschaften nicht erfüllen, sind unter anderem

- *Algorithmenbibliotheken*,
- *Computeralgebra*-Implementierungen,
- *Funktionale Programme*.

Durch die Nähe zur Hardware ist es ein naheliegender Schritt, Methoden der Hardwareverifikation auf die Ebene der systemnahen Software anzuheben. In der Hardwareverifikation ist die derzeit vorherrschende Methode, die in der letzten Dekade fast ausschließlich industrielle Anwendungen dominiert hat, die Methode des *Model Checking* (MC) zu der bereits ausführliche Einführungen existieren [CGP99]. Als spezielle Variante hiervon ist die Technik des *Bounded Model Checking*(BMC) [BCCZ99] zu nennen. Beide Verfahren

werden in den Abschnitten 2.2.6 und 2.2.7 eingeführt. Im Folgenden nennen wir deren abstrakte Arbeitsweise.

Ein besonderes Kennzeichen beider Verfahren ist, dass Sie vollständig automatisch arbeiten. Beide Methoden verifizieren Schaltnetze und -werke durch die Exploration des vollständigen Raumes aller möglichen Zustände. Zustandsräume werden hierbei durch die Menge aller Speicherbits, beispielsweise Flipflops, definiert. Die Schaltlogik definiert mögliche Übergänge von einem Zustand in einen Folgezustand. In der einfachsten Ausprägung, dem Model Checking mit expliziter Aufzählung von Zuständen, wird mittels einer Breiten- oder Tiefensuche der Zustandsraum abgesucht. Selbst diese einfache Variante hat bereits für Softwareverifikation Anwendbarkeit bewiesen: so konnten Engler et al. Fehler im Linux Kern nachweisen, indem sie ein vollständiges Dateisystem Schritt für Schritt ausführten [YTEM06].

Im Gegensatz zu Model Checking analysiert Bounded Model Checking ein System nur bis zu einer bestimmten Grenze vollständig. Diese Grenze kann eine Anzahl von Taktzyklen, Instruktionen oder Elemente in einer Datenstruktur sein. Aufgrund der Reduktion auf endliche Ausführungen kann so ein Laufzeitvorteil und die Entscheidbarkeit des Problems erzielt werden. Gerade bei komplexen Hardwaresystemen ist man auf den Laufzeitvorteil angewiesen.

Systemnahe Software kann als eine spezielle Ausführung eines Hardwaresystems angesehen werden. Ein Programm belegt Speicherplätze mit festen Werten, Register nehmen Variablen auf und Hardware-Operationen berechnen einfachste arithmetische Ausdrücke. Es liegt folglich nahe, dass ein Verfahren, welches für Hardwareverifikation Ergebnisse liefert, auch für die Verifikation von Softwaresystemen geeignet sein könnte.

Wir erwarten, dass die Verifikation der gewählten Systeme schwerer als die Verifikation gängiger Hardware ist. Gleichzeitig besteht aber eine prinzipielle Übereinstimmung in der Repräsentation von Arithmetik und endlichem Speicher, so dass die Anwendbarkeit der Methoden des Bounded Model Checking zu vermuten ist. Aus diesen beiden Gründen wurde die Entscheidung getroffen Bounded Model Checking als Verifikationsmethode im Rahmen dieser Arbeit einzusetzen.

1.3 Wissenschaftlicher Kontext

Wir beginnen zunächst mit einer Einordnung und Erläuterung vorhergehender Arbeiten. Im zweiten Unterabschnitt wenden wir uns den wissenschaftlichen Fragestellungen dieser Arbeit zu.

1.3.1 Wissenschaftliche Einordnung

Die Prüfung diskreter Modelle von Hardware und anderen Systemen mit endlichem Zustandsraum hat eine lange Tradition. Historisch gesehen war jedoch die Notwendigkeit, den Zustandsraum eines solchen Systems explizit repräsentieren zu müssen, eine große praktische Einschränkung für die Anwendung auf industrielle Systeme oder gar auf Software. Burch et al. präsentierten 1990 eine Methode, die es ermöglicht Zustandsräume von Modellen symbolisch darzustellen [BCM+90]. Die symbolische Kodierung ermöglichte es erstmals, große Systeme mit bis zu 10^{20} Zuständen auf Korrektheit untersuchen zu können. Ermöglicht wurde dieser Fortschritt durch den Einsatz von aussagenlogischen Entscheidungsdiagrammen, englisch *Binary Decision Diagram* (BDD) genannt. Der Einsatz von BDDs führte dazu, dass Modellprüfung als Standardmethode in dem Gebiet der Hardwareverifikation eingesetzt wurde.

Trotz des Einsatzes von BDDs war der Speicherverbrauch beim Einsatz von Modellprüfungstechniken enorm. Dies ist darauf zurückzuführen, dass ein BDD schlimmstenfalls linear viel Speicher in Abhängigkeit von der Größe des Zustandsraums benötigt. Bei Zustandsräumen die durch Bitvektoren definiert werden, wird schnell eine exponentielle Steigerung erreicht.

Eine pragmatische Technik, um mit der Explosion des Zustandsraumes zurecht zu kommen, ist die Beschränkung auf eine maximale Anzahl von Schritten (*Bound*) in dem Modell. Modelliert man Hardware und fragt danach, ob nach beliebig vielen Schritten ein bestimmter Zustand, z.B. ein Fehler, erreichbar ist, dann wird zunächst geprüft, ob nach einer kleinen Anzahl von Schritten (Taktzyklen) ein Fehler vorliegen kann. Wird ein Fehler gefunden, kann der Verifikationsversuch abgebrochen werden. Wird kein Fehler gefunden, kann das Verfahren mit einer größeren Schranke wiederholt werden bis entweder ein Fehler vorliegt, oder keine neuen Zustände mehr entdeckt werden. Diese schrittweise Technik wird Bounded Model Checking genannt und hat den Vorteil, dass die eigentliche Analyse der Frage, ob eine Eigenschaft gilt, auf das aussagenlogische Erfüllbarkeitsproblem (*SAT*) zurückgeführt werden kann. Die heute häufigste und auch in der Arbeit verwendete Interpretation basiert auf der Arbeit von Biere et al. von 1999 [BCCZ99].

SAT-basiertes Bounded Model Checking bietet den Vorteil, dass es für Schaltungen einsetzbar ist, für die eine Darstellung als klassischer BDD nicht mehr in den Speicher eines leistungsfähigen Rechners passen würde. Aus diesem Grund wiegt der Nachteil nicht schwer, dass gegebenenfalls nur die Korrektheit eines Systems für eine beschränkte Anzahl von Schritten gewährleistet werden kann.

2003 schlugen Clarke, Kroening und Yorav [CKY03] den Einsatz von Boun-

ded Model Checking für die Äquivalenzprüfung von Schaltungen in Verilog und deren vorhergehender abstrakter Beschreibung in C vor. Das Werkzeug CBMC [CKL04] wurde 2004 von Clarke, Kroening und Lerda vorgestellt. Neben der oben beschriebenen Äquivalenzprüfung wird auch direkt die Prüfung von C Programmen unterstützt.

In der neueren Forschung zur Softwareverifikation dominieren bei den automatischen Verfahren die Methode der Modellprüfung [BR02, CCG+03, CKSY05, CDW04, CCK+06], Implementierungen der Technik der abstrakten Interpretation [Pol08, CCF+05], sowie eine Reihe von individuellen Verfahren der statischen Analyse [YTEM06, ABD+02].

Das Verfahren der Softwareverifikation durch SAT-basiertes Bounded Model Checking verspricht dieselben Vorteile wie in der Hardwareverifikation: Speicherplatzsparende Modellprüfung unter Zuhilfenahme der seit Jahrzehnten stark verbesserten Erfüllbarkeitsprüfer für SAT.

1.3.2 Wissenschaftliche Fragestellung

Es herrscht der weitreichende Eindruck, dass formale Methoden endgültige Sicherheit über die Korrektheit eines Systems bestätigen können. Selbst wenn man die wissenschaftstheoretischen Probleme dieser Aussage ignoriert, stellen sich jedoch weitreichende Einschränkungen schon allein aus praktischen Gründen. In der Arbeit Seven Myths of Formal Methods liefert Hall [Hal90] eine Zusammenfassung der Probleme, unter anderem:

- Eine formale Darstellung eines real existierenden Software basiert auf - möglicherweise falschen - Modellierungsannahmen.
- Jedes Werkzeug zur Verifikation kann Fehler enthalten.
- Zu beweisende Eigenschaften (Spezifikationen) können unvollständig oder falsch sein.

Die Frage ist also, was formale Methoden zur Qualität beitragen, wenn sie keinen endgültigen Nachweis bringen können, dass ein System sich wie erwartet verhalten wird. Dies wirft die These auf, die dieser Arbeit zugrunde liegt und bisher nicht in wissenschaftlichen Arbeiten behandelt wurde: *Was tragen formale Methoden, im Besonderen die Methode des Bounded Model Checking, zur Qualität beziehungsweise Zuverlässigkeit von Software effektiv bei*? Abbildung 1.1 liefert eine bildliche Darstellung der Frage und deren Unterpunkte.

Diese Frage wird in zwei Unterpunkten behandelt:

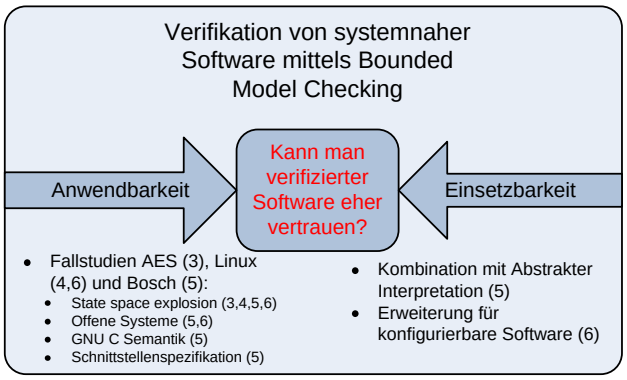


Abbildung 1.1: Struktur der wissenschaftlichen Fragestellung. Die Zahlen geben die Kapitelnummern an, in denen die Aspekte behandelt werden.

- *Ist Bounded Model Checking als Methode für die Softwareverifikation praktisch* **anwendbar***?*

- *Wie muss Bounded Model Checking erweitert werden, damit es für auch für große, konfigurierbare Software* **einsetzbar** *ist?*

Zur praktischen *Anwendbarkeit* gilt es eine Reihe von möglichen Problemen zu untersuchen. Zunächst muss untersucht werden, ob die Technik gut genug skaliert um Ergebnisse liefern zu können (*State space explosion*). Eine weitere Frage ist, ob offene Systeme mit vielen spärlich dokumentierten Schnittstellen eine sinnvolle Anwendung erlauben. Aufgrund des unzureichenden Wissens über Eingaben an Schnittstellen besteht die Gefahr unnötige Warnungen zu generieren. Als dritter Punkt besteht die Frage, wie sich Schnittstellenspezifikationen einfach formalisieren und prüfen lassen.

Zur Prüfung dieses Themenbereichs dienen drei Fallstudien, in denen die einzelnen Fragen empirisch geprüft werden.

Die *Einsetzbarkeit* von Bounded Model Checking betrifft die Frage, ob bei der Analyse großer Systeme, bei der die gewählte Schranke (*Bound*) nicht hinreichend groß gewählt werden kann, Bounded Model Checking sinnvoll angewendet werden kann. Des Weiteren besteht das Problem, dass moderne Software nicht nur aus einer Sammlung von Quelldateien (*Sourcecode*) besteht, sondern hoch adaptierbar und konfigurierbar ist. Kann Bounded Model Checking erweitert werden, um auch solche Systeme abzudecken?

Zwei neue Verfahren dienen dazu die obigen Probleme zu lösen und die Einsetzbarkeit herzustellen. Analog zum ersten Themenbereich werden diese auch durch Fallstudien evaluiert.

Durch die erhöhte Einsetzbarkeit und die umfassenden Fallstudien auf Software und Eigenschaften unterschiedlicher systemnaher Bereiche ist schließlich die Frage nach dem Beitrag von Bounded Model Checking zu beantworten. Erwähnenswert ist, dass die Resultate in Zusammenarbeit mit Entwicklern des Linux Kerns, sowie Testmanagern der Robert Bosch GmbH auch aus der Sicht des praktischen Einsatzes beurteilt werden.

Zusammengefasst enthält diese Arbeit die umfassende Studie der Anwendbarkeit der von Clarke et al. vorgeschlagenen Technik sowie deren Erweiterung zur Unterstützung von großen Softwareprojekten und Software Produkt Linien.

1.4 Aufbau und Gliederung der Arbeit

In Kapitel 2 wird eine Einordnung und eine grobe Vorstellung aller in der Arbeit verwendeten Verifikationsverfahren gegeben. Unter anderem definieren wir genauer was es heißt, dass ein Programm *korrekt* ist. Das Verfahren des Software Bounded Model Checking bildet die Grundlage der Kapitel 3 bis 6. Hier verwenden wir die Methode zum Beweis, dass zwei Implementierungen eines kryptologischen Algorithmus immer gleiche Ergebnisse liefern (Kapitel 3). Nach diesem Basisfall erweitern wir den Anwendungsbereich auf ein ganzes Betriebssystem: In Kapitel 4 untersuchen wir die Effektivität anhand des Linux Betriebssystems und erweitern das Verfahren um eine praktische Einsetzbarkeit zu erreichen.

Da sich bei der Anwendung von Bounded Model Checking auf ein offenes und sehr großes Betriebssystem nur eine sehr ausführliche Suche nach Fehlern, nicht aber der Beweis der Fehlerfreiheit erreichen lässt, stellen wir in Kapitel 5 eine Kombination aus Bounded Model Checking und einem weiteren Verfahren vor: abstrakte Interpretation. Abstrakte Interpretation ermöglicht es uns in diesen Fällen Fehlerfreiheit feststellen zu können ohne auf die Präzision der Methode des Bounded Model Checking verzichten zu müssen.

Kapitel 6 behandelt das Problem, dass viele Softwaresysteme heute nicht mehr als eine bloße Sammlung von Dateien vorliegen: sie werden generiert. Eine Methode zur effizienten Verifikation von generierten Familien von Programmen wird vorgestellt und anhand des Linux Betriebssystems empirisch als plausibel aufgezeigt.

Abbildung 1.1 beschreibt grob die Aufteilung der wissenschaftlichen Fragen auf die einzelnen Kapitel.

Im Folgenden stellen wir nun die Grundlagen der Arbeit vor. Neben den allgemeinen Definitionen von Programmen und deren Korrektheit werden auch die Verfahren, die in den Fallstudien zum Einsatz kommen, detailliert

vorgestellt.

Grundlagen 2

Ziel dieser Arbeit ist die Verifikation von Programmen. Verifiziert werden sollen hierbei vor allem Eigenschaften, die sich auf die Bedeutung eines Programms beziehen. Eine reine Analyse der Form eines Programms, ähnlich der grammatikalischen Analyse von natürlichsprachlichen Sätzen, wird nicht behandelt, beziehungsweise als bereits geschehen vorausgesetzt.

Für die Verifikation muss also zunächst die Bedeutung eines Programms formal definiert werden. Oft wird hierzu ein Modell aller Programmausführungen als formale Bedeutung definiert. Die Verifikation behandelt dann die Frage, ob in dem Modell eine Eigenschaft gilt.

2.1 Programmsemantik

Bei Programmen unterscheidet man zwischen der *Syntax* (Form) und der *Semantik* (Bedeutung) (siehe zum Beispiel [NN92]). Die Syntax eines Programms betrifft die grammatikalische Struktur eines Programms, das heißt die Frage, ob ein Programm wirklich Element einer konkreten Programmiersprache ist. Ein anderer Ausdruck für syntaktische Korrektheit ist Wohlformuliertheit. Die Prüfung der Syntax wird überlicherweise von einem Compiler vorgenommen.

Die Komplexität einer solchen Prüfung ist für viele gängige Sprachen effizient möglich. Effizient heißt, dass der theoretische und praktische Berechnungsaufwand vernachlässigbar in Bezug auf den Aufwand der Verifikation von semantischen Eigenschaften ist. Im Weiteren gehen wir davon aus, dass alle Programme syntaktisch korrekt vorliegen.

Ein interessanter Gedanke ist, dass eine formale Semantik eine syntaktische Darstellung der Bedeutung eines Programms liefert. Auf dieser kann dann eine Analyse, beispielweise ein Ableitungskalkül angewendet werden. Quelle für alle semantischen Modellierungen ist die Sprachdefinition, das heißt für die Sprache C die ISO Standards [ISO99].

Im ISO/IEC 9899:1999 Standard [ISO99], auch bekannt unter den Namen C99 sowie ANSI-C, wird zu jedem Sprachbestandteil eine syntaktische wie auch semantische Beschreibung gegeben. Im Standard wird die Semantik in der Form einer sprachlichen Beschreibung des Verhaltens einer abstrakten

Maschine definiert:

> "5.1.2.3 Program execution
> [#1] The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant.
> [...]
> [#3] In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced [...]."
> (ISO/IEC 9899:TC2 WG14/N1124 Committee Draft — May 6 2005, [ISO99])

Diese Art der Definition einer Semantik wird als *operationell* bezeichnet. Die vermutlich bekannteste abstrakte Maschine ist sicherlich die von Neumann Maschine die eine einfache Modellierung von Rechenmaschinen vornimmt, die bis heute aussagekräftig moderne Hardwarearchitekturen abstrahiert.

Die im Standard definierte Semantik lässt jedoch absichtlich wie auch unabsichtlich Lücken, die für eine abstrakte Maschine mehrere Zustände nach einer Anweisung zulassen. Im Standard ISO/IEC 9899:1999 heißt es:

> "3.4.1
> 1 implementation-defined behavior is unspecified behavior where each implementation documents how the choice is made.
> 2 EXAMPLE An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right."
> (ISO/IEC 9899:TC2 WG14/N1124 Committee Draft — May 6 2005, [ISO99])

Übersetzt heißt dies, dass ein Compiler der den Standard umsetzt (implementiert), selbst festlegen muss wie das höchstwertigste Bit belegt wird, falls die Bitvektordarstellung einer Zahl nach rechts rotiert beziehungsweise verschoben wird.

```
int c1 = 1;
int c2 = c1 >> 1;
```

Eine Implementierung der Sprachsemantik in einem Compiler muss also festlegen ob `c2` gleich `0 := 0000 0000` oder `128 := 1000 0000` sein muss.

Die Semantik, wie sie im ISO Standard angegeben ist, ähnelt sehr einer Beschreibung wie sie auf eine abstrakte Registermaschine der theoretischen Informatik oder auch auf eine von Neumann Maschine zutreffen könnte. Ein Nachteil ist jedoch, dass sich diese nicht direkt zur formalen Analyse eignet. Deshalb wird für die Verifikation, wie sie von CBMC vorgenommen wird, noch eine – meist implizit verwendete – Zwischenstufe eingefügt. Diese beruht darauf, dass eine von Neumann Maschine oder allgemein eine Registermaschine als Zustandsübergangssystem modelliert werden kann. Solche Zustandsübergangssysteme haben eine sehr viel einfachere Struktur, so dass sie einfacher analysiert oder weiter transformiert werden können.

2.1.1 Programme als Zustandsübergangssystem

Der Ansatz, der im Rahmen der Arbeit verwendet wird, ist eine spezielle Form operationeller Semantik, welche auf einem *Labelled Transition System* basiert.

Definition 1 *Ein* Labelled Transition System (LTS) *ist ein Fünftupel, geschrieben* $(\mathcal{S}, \mathcal{S}_i, \mathcal{S}_e, \Sigma, \mathcal{T})$*, wobei* $\mathcal{S}$ *eine endliche Menge von Zuständen beschreibt.* Σ *ist eine Menge von Labels und* $\mathcal{T}$ *ist die dreistellige Übergangsrelation* $\mathcal{T} \subseteq \mathcal{S} \times \Sigma \times \mathcal{S}$ *mit* $\mathcal{S} \times \Sigma \times \mathcal{S} := \{(s, \alpha, s') | s, s' \in \mathcal{S} \wedge \alpha \in \Sigma\}$*.* $\mathcal{S}_i \subseteq \mathcal{S}$ *ist eine nicht-leere Menge von Startzuständen und* $\mathcal{S}_e \subseteq \mathcal{S}$ *ist eine nicht-leere Menge von Endzuständen.*

$(s, \alpha, s') \in \mathcal{T}$ bezeichnet einen Übergang vom Zustand s in einen Zustand s', der durch α ermöglicht wird. Die Verkettung $\mathcal{T}^2$ ist definiert als

$$\mathcal{T} \circ \mathcal{T} := \left\{(s, \alpha_1\alpha_2, s'') | \exists s' \in \mathcal{S}.(s, \alpha_1, s') \in \mathcal{T} \wedge (s', \alpha_2, s'') \in \mathcal{T}\right\}$$

$\mathcal{T}^n, n \in \mathbb{N} \setminus \{0, 1\}$ ist die Verallgemeinerung der obigen Verkettung. $\mathcal{T}^*$ ist der transitive Abschluss von $\mathcal{T}$. Die Menge der Zustände, die von einem beliebigen Zustand aus S unter beliebigen Labels in einem Übergang erreicht werden kann, bezeichnet $img_{\mathcal{T}}(S), S \subseteq \mathcal{S}$. Die Erweiterung auf die Menge der Zustände die in beliebig vielen Schritten erreicht werden kann, wird mit $img_{\mathcal{T}^*}(S)$ bezeichnet.

Eine Alternative zum LTS ist eine Kripke Struktur, die später zusammen mit dem Verfahren des Bounded Model Checking eingeführt wird. Eine Kripke-Struktur ist einem Labelled Transition System sehr ähnlich (siehe [MOSS99]).

Das Grundproblem bei der Modellierung von Systemen wie beispielsweise Programmen ist die Anzahl der benötigten Zustände. Zur Programmierung

eines Zählers reicht beispielsweise die Angabe eines n-stelligen Bitvektors und die Semantik einer Inkrementfunktion, die einen Bitvektor um eins erhöht. Zusammen ist dieses System darstellbar in einer Schaltung von $O(n)$ Gattern und $O(n)$ Flipflops. Der Zustandsraum einer solchen Schaltung ist jedoch $O(2^n)$ weil er alle möglichen Werte des Bitvektors umfassen muss. Will man also die Semantik eines Programms in einem LTS darstellen, ist das LTS in der Regel exponentiell größer als die Programmbeschreibung in Bits. Dieser Effekt wird als *State space explosion* bezeichnet.

Das sequentielle Verhalten eines LTS lässt sich durch Pfade beschreiben. Ein *Pfad* $\pi := s_0, ..., s_n$ ist eine geordnete Sequenz von Zuständen. $\pi(i)$ bezeichnet hierbei den i-ten Zustand der Sequenz und π_i die Teilsequenz, die ab dem i-ten Zustand startet $(s_i, ..., s_n)$. Ein Pfad *kommt in einem Programm vor*, genau dann wenn für alle Paare $s_i, s_{i+1}, 0 \leq i < n$ ein Übergang in $\mathcal{T}$ existiert. Ein *einfacher Pfad* ist ein Pfad, bei dem jeder Zustand nur einmal vorkommen darf. Ein Pfad enthält einen *Zyklus* genau dann, wenn er nicht einfach ist.

Ist bezogen auf einen Zustand und ein Label der Nachfolger eindeutig definiert, spricht man von einem deterministischen LTS:

Definition 2 *Ein* Deterministisches Labelled Transition System (DLTS) *ist ein LTS für das folgende Einschränkung gilt:* $\forall(t_1, t_2) \in \mathcal{T} \times \mathcal{T}$ *mit* $t_1 := (s', \alpha, s_1)$ *und* $t_2 := (s', \alpha, s_2)$ *gilt* $s_1 = s_2$.

Einschränkungen

Eine operationelle Semantik kann als Menge von Pfaden einer abstrakten Registermaschine aufgefasst werden. Diese kann in der Form von Zustandsübergängen eines DLTS oder LTS angegeben werden. Um dies zu vereinfachen, werden zwei Annahmen über die Menge der zu analysierenden Programme gemacht:

1. Der Zustandsraum, das heißt die maximale Anzahl von verwendeten Bits zur Kodierung des Stacks, des Heaps und der statischen Variablen eines Programms, ist *endlich*.

2. Die durch den Standard definierte Semantik beschreibt das Verhalten einer *deterministischen* abstrakten Maschine.

Die erste Einschränkung ist motiviert durch die pragmatische Feststellung, dass für systemnahe Programme eine maximale Speicherverwendung angegeben oder angenommen werden kann. Die zweite Einschränkung abstrahiert

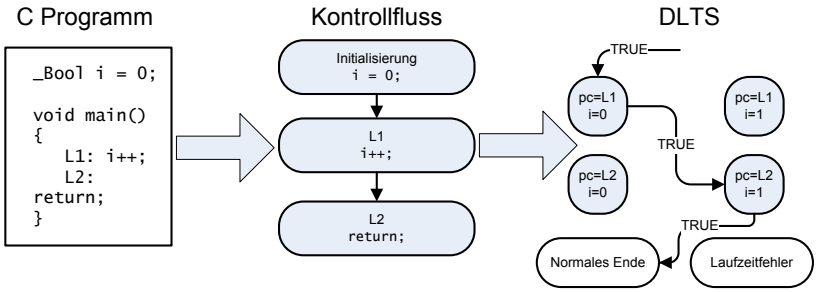


Abbildung 2.1: Das C Programm auf der linken Seite implementiert einen 1 Bit Zähler, der nach einmaligem Hochzählen die Ausführung abbricht. In der Mitte findet sich eine Darstellung des Programms als Kontrollflussblöcke, die die einzelnen Schritte des Programms abbilden. Auf der rechten Seite wird die Semantik des Programms in der Form eines DLTS gegeben. Hiebei kodiert eine neue Variable `PC` die Information, welches der nächste auszuführende Programmblock ist. Im LTS sind also die Zustände zwischen den Programmschritten beschrieben.

von der Tatsache, dass die natürlichsprachliche Definition der Sprache C unspezifiziertes Verhalten zulässt. Diese Art von Nichtdeterminismus kann man jedoch umgehen: Die Konkretisierung des Verhaltens wird durch implementierungsspezifische Eigenschaften des Compilers und eine konkrete Hardwareplattform vorgegeben. Die einzige praktisch vorkommende Art von Nichtdeterminismus ist die Ausführungsreihenfolge von parallelen Programmen. Bis auf Abschnitt 4.2.4 wird aber keine Untersuchung parallel laufender Programme vorgenommen werden, so dass Annahme 2 für alle anderen Abschnitte gilt.

Annahme 1 bedingt, dass der Zustand eines Programms immer durch einen endlichen Vektor $\vec{b}$ Boolescher Variablen kodiert werden kann: $\vec{b} \in \mathcal{S}$ mit $\mathcal{S} := \mathcal{B}^n, n \in \mathrm{N}$. Die Mengen, die durch $\mathcal{S}_i$ und $\mathcal{S}_e$ beschrieben sind, lassen sich folglich als n-stellige Boolesche Relation ausdrücken. Zur Modellierung bedingter Übergänge soll Σ ebenso die Form einer n-stelligen Booleschen Relation annehmen. Zusammen ergibt sich für $\mathcal{T}$, die Übergangsrelation eines LTS, eine $3 \cdot n$-stellige Boolesche Relation.

Annahme 2 garantiert deterministische Übergänge zwischen Zuständen. Der Übergang kann durch eine abstrakte Modellierung der Registermaschine entsprechend des Standards definiert werden (siehe die Verifikations-Werkzeuge CBMC [CKL04] oder C32SAT [BB07]).

Ein Beispiel für die Umwandlung eines Programms in ein Transitionssystem, das die Semantik eines 1 Bit Zählers definiert, ist in Abbildung 2.1 gegeben.

2.1.2 Kompakte Semantik

Die obige Definition von Transitionssystemen ist zur Darstellung großer Programme ungeeignet. Aus Gründen der Übersichtlichkeit können folgende Vereinfachungen gemacht werden:

- Beim Übergang von einem Zustand $\vec{b}$ in einen Folgezustand $\vec{b'}$ werden üblicherweise nur Angaben über geänderte oder relevante Zustandsbits gemacht. Das Weglassen von Zustandsinformation bei $\vec{b'}$ ist also so zu interpretieren, dass alle nicht neu definierten Bits gleich bleiben.
- Aus Gründen der Darstellung können konkrete Zustände zu abstrakten Zustandsmengen zusammengefasst werden. Ebenso können die Label in einer informellen abstrakten Schreibweise wiedergegeben werden (Beispiel 1).

Beispiel 1 *Ein Zählerprogramm inkrementiert eine Variable i um genau eins und lässt eine andere Variable u unverändert. Das Inkrement lässt sich als Übergang von Bitvektoren darstellen, da sich i als endlicher Bitvektor kodieren lässt. Für einen 8-Bit Zähler und zwei Variablen werden 2^{16} Übergänge benötigt. Zur kompakten Darstellung schreiben wir einfach $i' = i + 1$. Dass u nicht verändert werden soll ist implizit durch Weglassen formuliert.*

Es ist wichtig zu bemerken, dass sich die obigen Regeln nur auf die Darstellung, nicht aber auf die Semantik beziehen.

2.1.3 Die Sprache C

C ist eine imperative Programmiersprache. Der aktuelle Standard der die Syntax und Semantik der Sprache C definiert wird ISO/IEC 9899:1999 genannt [ISO99]. Im Weiteren wird der Begriff ANSI-C als Kurzreferenz für obigen Standard verwendet.

Für die Verifikation ist es notwendig folgende Merkmale der Sprache zu unterstützen und zu modellieren:

Abgeleitete Datentypen In C gibt es eine Reihe von *Basistypen* wie `int, char` und *abgeleitete Datentypen*, die durch die Schlüsselworte `struct` und `union` definiert werden können. Die Basis-Datentypen umfassen ganze Zahlen (`short,int,...`) sowie Fließkommawerte (`float, double,...`). Eine `struct` Deklaration ermöglicht es verschiedene Basis- oder abgeleitete Typen zu einem neuen Typ zusammenfassen. Das Schlüsselwort `union` beschreibt einen Mischtyp, der es ermöglicht ein Element eines der unter `union`

aufgeführten Typen zu umfassen. Indirekt lassen sich über `unions` Typumwandlungen (*Typecasts*) realisieren.

Eine sehr grundlegende und, zumeist als selbstverständlich angenommene, Einschränkung der Datentypen ist die Tatsache, dass alle C Datentypen nur endliche Quellbereiche oder Domänen haben. Der Datentyp `int` kodiert je nach Architektur Zahlen als 16, 32 oder 64 Bitvektor. Als Folge kann es zu arithmetischen Überläufen kommen, falls die maximal oder minimal darstellbare Größe überschritten wird. Die meisten C Basistypen basieren auf *modularer Arithmetik*.

Zeiger Ein Zeiger realisiert das Konzept der Referenz in der Sprache C. Anstatt beispielsweise Parameter einer Funktion kopieren zu müssen, kann eine Referenz auf Daten übergeben werden. Durch Dereferenzierung kann man Zugriff auf das referenzierte Objekt erhalten, welches nachfolgend manipuliert werden kann. Zeiger haben in C einen Typ, der den Typ des Objektes angibt, welches man durch Dereferenzierung erhalten kann. Zusätzlich ist mit einem Zeiger die Information assoziiert wie viele Referenzierungsstufen enthalten sind. Ein Zeiger auf einen Zeiger auf eine `int` Variable, im Programm als `int **` deklariert, hat zwei Indirektionsstufen.

Ein großes Problem ist, dass Zeiger in C auf beliebige Speicherplätze verweisen können. Das Ziel eines Zeigers `p` kann willkürlich gesetzt oder per arithmetischen Operationen manipuliert werden (etwa `p = p + 1`). Dies führt dazu, dass die Manipulation von Zeigern unbefugten Zugriff auf sensible Daten (Passwörter) oder privilegierte Operationen (Löschen von Daten) ermöglichen kann. Ein weiteres Problem ist die Einführung von undefiniertem Verhalten durch Zeiger auf bisher nicht initialisierte Objekte. Die Überprüfung, dass Zeiger nur auf gültige Speicherplätze zeigen, wenn sie benutzt werden, ist ein Standardproblem der Programmanalyse. Der ANSI-C Standard definiert das Symbol `NULL` als eine besondere Zeigerkonstante, die verwendet wird um zu kodieren, dass ein Zeiger auf ein ungültiges Objekt zeigt. Meist ist `NULL` als Zeiger `(void *)0` definiert. Man kann aber nicht immer durch einen Vergleich mit `NULL` feststellen, dass ein Zeiger auf ein gültiges Objekt zeigt:

```
int* foo() {
    int i;
    return &i;
}
```

Nach Aufruf der Funktion `foo` zeigt der Rückgabewert auf die lokale Variable `i`. Da `i` lokal ist, wird sie bei Aufruf der Funktion auf dem Stack angelegt.

Nach Ende der Funktion ist ein Zugriff auf diese Speicherstelle ungültig. Der Rückgabewert ist aber nicht `NULL`.

Ein weiterer Punkt ist, dass durch Zeiger Typumwandlungen und generische Funktionen implementiert werden. So ist der Typ `void *` umwandelbar in jeden anderen Zeigertyp. Ebenso ist jeder Zeigertyp nach `void *` konvertierbar. Eine Funktion, die einen Parameter von diesem Typ akzeptiert, ist also in der Regel für Parameter von unterschiedlichen Typen ausgelegt. Dies ist ein Problem für die Analyse von offenen Systemen, da nicht automatisch festgestellt werden kann, von welchem Typ Eingaben an die Funktion übergeben werden. Lässt man zu viele Typen zu, kann es zu unrealistischen Eingaben kommen, die zu einer unerwünschten Fehlermeldung des Verifikationswerkzeugs führen können:

```
void increment(void * v) {
     ((char *)v)++;
}
```

Die Funktion `increment` referenziert die 8 Bits, die an der Adresse `v` liegen. Die Funktion kann benutzt werden um eine `char` Variable, aber auch eine Struktur oder ein Array, deren erstes Feld eine `char` Variable beinhaltet zu modifizieren. In Linux würde die Funktion auch verwendet, um beispielsweise die ersten 8 Bits einer Variable von anderem Typ wie `long` zu modifizieren. Sind die Aufrufe der Funktion nicht bekannt, kann man nicht auf die wirklich verwendeten Eingabetypen zurückschließen.

Funktionen und Rekursion Das Konzept von Funktionen zur Strukturierung von Programmen ist hinlänglich bekannt und soll nicht weiter erläutert werden. Eine Besonderheit von C ist die Formulierung von Funktionen mit einer variablen Anzahl formaler Parameter. Das bekannteste Beispiel ist die `printf` Funktion, welche Textausgaben aller Art realisiert. Es ist wichtig in Erinnerung zu behalten, dass sich C Funktionen signifikant von mathematischen Funktionen unterscheiden:

1. C Funktionen können nicht nur die formalen Parameter lesen und schreiben.
2. Als Folge hiervon sind C Funktionen nur in dem Sinne deterministisch, dass sie bei gleichem *Gesamtzustand* die gleichen Ausgaben produzieren (*funktionale Konsistenz*). Bezogen auf die Menge der formalen Eingabeparameter der Deklaration gilt der Determinismus nicht.

Eine weitere Besonderheit ist das Rechnen mit Funktionen zur Manipulation des Programmflusses wie weiter unten erläutert wird.

`goto` und berechnete Sprünge Die `goto L;` Anweisung bewirkt einen Sprung im Programmfluss zu einer Programmstelle, die mit einem *Label* `L` gekennzeichnet wurde. Labels sind im Bereich von Funktionen eindeutig definiert. Erweiterungen des ANSI-C Standards erlauben es, das Ziel eines Sprungs über so genannte Labelvariablen zu berechnen. Solche *computed gotos* sind durch viele Kodierrichtlinien verboten und kommen in systemnahen Programmen fast nie vor und werden im Rahmen dieser Arbeit deshalb nicht behandelt. Ein Beispiel für eine Sammlung von Kodierrichtlinien ist MISRA-C [Aut04] für eingebettete Programme im Automobilsektor. Zu erwähnen ist, dass Schleifen sich vollständig durch `goto` Anweisungen ausdrücken lassen.

Eine besondere Direktive zur Berechnung von Sprungzielen, die auch vom ANSI-C Standard unterstützt wird, ist die Verwendung von *Funktionszeigern*. Ein Funktionszeiger repräsentiert eine Adresse an die durch Dereferenzierung gesprungen wird. Die weitere Typinformation definiert die Art der Funktion, wie etwa den Rückgabeparameter und die Anzahl und Typen der formalen Eingabeparameter. Nur wenige Werkzeuge der Programmverifikation unterstützen die Verwendung von Funktionszeigern (CBMC [CKL04], SATABS [CKSY05], Static Driver Verifier [BBC+06]).

Dynamische Speicherallokation Fast alle systemnahen Programmiersprachen erlauben die Anforderung von Speicher außerhalb des Stacks durch ein Programm. In Java können neue Objekte durch das Schlüsselwort `new` erzeugt werden. In C++ wurden hierfür Konstruktoren eingeführt. In C jedoch muss solcher Speicher immer explizit *alloziert* (reserviert) und *dealloziert* (freigegeben) werden.

Die Verwendung von Speicherbereichen ohne Allokation oder nach einer Deallokation ist ein schwerer Fehler, der zu undefiniertem Programmverhalten führen kann. Ein weiterer Fehler ist das Fehlen einer Deallokation nach der letzten Nutzung. Fehler dieser Art werden Speicherlecks (*memory leaks*) genannt und sind vor allem in lange laufenden Programmen, wie Webservern, ein Problem. Als Vorgriff auf die nächsten Abschnitte sei erwähnt, dass die Prüfung auf die Abwesenheit von Speicherlecks eine Lebendigkeitseigenschaft darstellt. MISRA-C [Aut04] verbietet auch die Nutzung von dynamischer Speicherallokation. Außerhalb dieses speziellen Bereichs ist sie jedoch eine verbreitete Technik.

Nach der Darstellung der Mächtigkeit der Sprache C befassen wir uns nun mit drei gängigen Ansätzen zur Verifikation von C Programmen.

2.2 Verifikation - Statische Analyse bis Modellprüfung

In diesem Abschnitt werden zunächst die grundlegenden Verifikationstechniken präsentiert.

Was ist die Verifikation eines Programms? In der Regel soll gezeigt werden, dass für alle Eingaben und alle Programmpfade eine Eigenschaft gilt. Alan Turing jedoch zeigte, dass eine fundamentale Eigenschaft aller Programme nicht berechenbar ist: Hält ein Programm oder fährt es mit einer Berechnung unendlich fort. Alle nicht-trivialen Eigenschaften sind ähnlich schwer zu analysieren wie das so genannte *Halteproblem* (Satz von Rice). Ein einfaches Programmbeispiel motiviert diese Aussage:

```
P :=  P1; Fehler;
```

Das Programm `P` ist die sequentielle (hintereinander ausgeführte) Komposition der Programme `P1` und `Fehler`. In letzterem Programm kann ein Fehler auftreten, der dazu führt dass eine Eigenschaft nicht erfüllt ist. Die Frage, ob es in `P` zu einem Fehler kommen kann hängt also davon ab, ob `P1` immer oder auch nur manchmal beendet wird. Führt `P1` immer zu einer unendlichen Berechnung, wird das zweite Programm niemals erreicht und die Eigenschaft ist trivialerweise erfüllt.

Dieses Beispiel erläutert, dass Eigenschaften, die direkt oder indirekt die Erreichbarkeit bestimmter Zeilen in einem Programm betreffen, gleichschwer zu analysieren sind wie das Halteproblem. Der Satz von Rice ist das berühmteste Statut, welches umgangssprachlich besagt, dass jede Aussage, die die Gültigkeit einer nicht trivialen Eigenschaft über ein Programm betrifft, nicht berechenbar ist. Im Folgenden studieren wir zunächst die Schlupflöcher, die in der Praxis effiziente Programmverifikation ermöglichen. Nachfolgend geben wir eine Übersicht über Eigenschaften, um im Folgenden grundlegende Techniken für die Verifikation einiger dieser Eigenschaften darzustellen. Die Differenzierung der verschiedenen Methoden wird die Wahl von Bounded Model Checking im Hauptteil der Arbeit motivieren und gleichzeitig auch Grenzen der gewählten Methode aufzeigen.

2.2.1 Die Grenzen der Grenzen

Die Programmiersprache C ist, von einem abstrakten Standpunkt aus betrachtet, Turing vollständig. Somit ist die Berechnung von nicht-trivialen Eigenschaften eines Programms (z.B. ob es hält) im Allgemeinen nicht entscheidbar. Betrachtet man nur eine Teilklasse der möglichen C Programme,

kann jedoch Entscheidbarkeit hergestellt werden. Eine der grundlegenden Einschränkungen ist beispielsweise die Forderung, dass das Programm nur endlich viel Speicher und somit nur endlich viele Zustände nutzen darf. In einem solchen Fall wäre das Haltproblem trivialerweise lösbar durch eine Tiefensuche auf dem Zustandsraum mit Markierung bereits besuchter Zustände auf dem aktuellen Pfad.

2.2.2 Eigenschaften

Unter der *Korrektheit* eines Systems versteht man eine Relation, die beschreibt ob ein System eine Spezifikation erfüllt. Im Bereich formaler Systeme ist die Relation ebenfalls in einer formalen Sprache abgefasst. Zusammen mit einer formalen Repräsentation des Systems kann dann ein rigoroser Beweis geführt werden, ob ein System M eine Spezifikation p erfüllt (in Formeln $M \models p$).

Klassisch wird geprüft, ob ein Programm eine Anforderung oder Spezifikation erfüllt. Ein anderer Ansatz ist es, erst die formale Spezifikation eines Systems auszuarbeiten und dann aus dieser den Quellcode automatisch zu generieren: In der B-Methode [Abr96] beschreibt Abrial eine Methode, die auf der Verfeinerung von Modellen bis hin zur Implementierung basiert. Ausgehend von einer abstrakten Spezifikation wird Stück für Stück Implementierungsdetail hinzugefügt.

Ein Beweis, dass ein neues Modell ein altes verfeinert, trägt die Korrektheit bis hin zum generierten Quellcode. Ein solches Vorgehen bezeichnet man auch als *Correctness by Design* (CoD). Bei Programmen, die bereits geschrieben sind oder bei denen nur teilweise Eigenschaften formal erfasst werden sollen, kann CoD nicht angewendet werden. In dieser Arbeit verwenden wir nur den entgegengesetzten Ansatz, der auf der Korrektheitsprüfung auf Implementierungsebene basiert.

Eigenschaften, die der formalen Überprüfung bedürfen, sind häufig in einer der folgenden Kategorien zu finden:

1. **Funktionale Korrektheit:** Entspricht der Rückgabewert einer Funktion dem erwarteten?

2. **Zeitliches Verhalten:** Benötigt die Ausführung eines Programms weniger als 5 Sekunden beziehungsweise 1000 Zyklen?

3. **Sicherheit**[1]**:** Kann es jemals passieren, dass z.B. eine Division durch Null stattfindet?

[1] *Sicherheit* wird hier im Sinne von Ausfallsicherheit verwendet.

4. **Lebendigkeit:** Wird eine Anfrage irgendwann beantwortet werden?

5. **Fairness:** Wird eine Anfrage unendlich oft positiv beantwortet?

Funktionale Korrektheit kann unterschiedlichste Ausprägungen haben. Die häufigsten Ausprägungen sind die Prüfung auf Äquivalenz (zum Beispiel in der Hardwareverifikation) oder die Prüfung, ob ein Modell ein anderes verfeinert (Event-B [Abr96]). Zeitliches Verhalten ist vor allem im Bereich der eingebetteten Systeme von Bedeutung und wird in dieser Arbeit nicht behandelt. Ein Beispiel ist hier die Prüfung, ob eine Funktion unter allen Umständen in 100 Millisekunden beendet wird.

Sicherheitseigenschaften gehören zu den häufigsten Spezifikationen für sicherheitskritische Software und werden im Folgenden eine besondere Behandlung bekommen. Insbesondere ist die Methode des Bounded Model Checking spezialisiert für die Prüfung von Sicherheitseigenschaften – Lebendigkeitseigenschaften sind in der Praxis nur eingeschränkt mit Bounded Model Checking prüfbar.

Im nächsten Abschnitt stellen wir eine Klasse von Algorithmen vor, die dazu dient Eigenschaften über Programme zu berechnen. Die Klasse der statischen Analysemethoden verwendet hierbei zumeist eine Abstraktion des Programmes, so dass die Gültigkeit von Eigenschaften schnell berechenbar und entscheidbar ist.

2.2.3 Statische Analyse

Verfahren der statischen Analyse von Quellcode verwenden als Eingabe ein Programm in einer, meist imperativen, Programmiersprache und berechen, ob eine Eigenschaft für das ganze Programm oder für jede Zeile gilt.

Die Entscheidbarkeit wird durch Abstraktion, Beschränkung der Programme und Eigenschaften oder durch konservative Ergebnisunschärfe ermöglicht.

Klassische Programmanalyse

Nielson et al. [NNH99] beschreiben die Natur der (klassischen) Programmanalyse wie folgt:

> Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer.

Zusammengefasst heißt dies, dass in der Programmanalyse sichere und berechenbare Annäherungen vom Laufzeitverhalten eines Programms gefunden werden. *Sicher* meint, dass nicht weniger Verhalten oder mögliche Werte von Ausgaben oder Variablen angenommen können, als das Programm wirklich hat. Es handelt sich also um eine *Überapproximation*, die garantiert, dass das Programm wirklich sicher ist, wenn die Programmanalyse dies ausgibt.

Zu den bedeutendsten klassischen Ansätzen der Programmanalyse rechnen die Autoren *Datenfluss- und Constraint-basierte Analyse*, sowie *abstrakte Interpretation* und *Typ- und Effektsysteme*. Allen Ansätzen ist gemeinsam, dass sie *schlüssige* oder *konservative* Ergebnisse bezüglich der Programmkorrektheit, beziehungsweise des vorhergesagten Programmverhaltens, liefern. Dies entspricht der obigen Überapproximation des Verhaltens.

Alle obigen Ansätze betreiben *statische Analyse*, analysieren also Programme ohne sie auszuführen. Sie stehen im Gegensatz zu *dynamischen* Verfahren, wie etwa konventionellem Testen eines Programms. Um den Gegensatz zu letzteren Ansätzen auszudrücken, spricht man von Verfahren der *statischen Analyse*.

Die oben genannten Verfahren besitzen eine gemeinsame Basis [NNH99]: Zu jedem der Verfahren existieren Varianten, die nach folgenden Kriterien aufgeschlüsselt sind (siehe [JR00]). Jedes der Kriterien erlaubt es, Geschwindigkeit und Präzision der Analyse gemäß der notwendigen Anwendung zu balancieren.

Fluss-Sensitivität Eine *flusssensitive* Analyse berücksichtigt die Reihenfolge, in der atomare Statements in einem Programm ausgeführt werden können. Als Folge hiervon kann *flussinsensitive* Analyse keine Aussagen beweisen, die von der Ausführungsordnung abhängen. Weiter noch ist das Ergebnis einer flusssensitiven Analyse meist an bestimmte Programmstellen gekoppelt: Variable `x` hat möglicherweise den Wert `1` in Zeile 5.

Eine insensitive Analyse kann nur eine Aussage über alle möglichen Werte für `x` an beliebigen Stellen im Programm liefern. Model Checking und Bounded Model Checking sind im diesen Sinne flusssensitive Analysen. Flusssensitive Analyse wird beispielsweise eingesetzt, um vor der eigentlichen Verifikation mögliche Ziele von Zeigern in Programmen zu approximieren [ABD+02, CKL04, CKSY05].

Pfad-Sensitivität *Pfadsensitive* Analysen berücksichtigen nicht nur die Reihenfolge der Ausführung, sondern genaue Pfade im Programm. Eine `if (expr) then P1; else P2;` Anweisung ist das einfachste Beispiel hierfür. Alle Pfade, die in das Programm `P1` führen, müssen berücksichtigen, dass

`expr` gilt. Alle Ergebnisse für Programmstellen in `P2` müssen `!(expr)` annehmen. *Pfadinsensitive* Analysen betrachten Pfade, die in echten Ausführungen nicht vorkommen können. Dies führt zu inkorrekten Warnungen (false positives).

Kontext-Sensitivität Moderne imperative Programmiersprachen wie Java, C und C++ unterstützen das Konzept von Funktionen beziehungsweise Prozeduren. Ähnlich wie in der Mathematik werden Funktionen verwendet um die Berechnung von Ausgaben aus Eingaben zu kapseln. Man kann eine Funktion verstehen als Abbildung von Eingaben auf Ausgaben, wobei nicht alle Ausgabe- und Eingabeparameter syntaktisch kenntlich gemacht werden müssen. Eine Funktion, die sich selbst – direkt oder indirekt über Aufrufe anderer Funktionen – aufruft, nennt man *rekursiv*.

Zu jedem zur Laufzeit vorkommenden Aufruf einer Funktion können eine Reihe von Angaben über den *Aufrufkontext (Kontext)* gemacht werden: In welcher Zeile fand der Aufruf statt? Wohin springt die Ausführung nach Beendigung der Funktion? Welche Funktionen wurden zuvor aufgerufen? Wie ist die Belegung der in der Funktion gelesenen Speicherinhalte?

Für rekursive Funktionen, die pro Schritt mehr als einen Aufruf ausführen, steigt die Zahl der Kontexte exponentiell in der Tiefe des Rekursionsbaums.

Eine *kontextinsensitive* Analyse erstellt für jede Funktion genau eine Teilanalyse. Diese soll allgemein genug sein, dass sie in allen Kontexten verwendet werden kann. Üblicherweise bedeutet dies, dass keine Informationen über mögliche Aufrufkontexte verwendet werden dürfen.

Im Gegenzug existieren zwei Ansätze einer kontextsensitiven Analyse: Zum Einen besteht die Möglichkeit, eine Teilanalyse einer Funktion genau einmal, dafür aber parametrisiert zu erstellen. Die Parameter dienen dazu, das Verhalten bei jeder Benutzung je nach Kontext verfeinern oder einschränken zu können. Solch eine Analyse wird *kompositionell* genannt. Zum Zweiten besteht die Möglichkeit eine Funktion für jeden Aufrufkontext neu zu analysieren.

Eine besondere Variante des zweiten Verfahrens ist das so genannte *Inlining* welches in manchen Model-Checking Varianten verwendet wird (zum Beispiel in [CKL04]). Inlining beschreibt den Prozess jede Funktion komplett in alle Aufrufkontexte einzufügen[2]. Hierzu sei ein kurzes Beispiel gegeben:

[2] Dies entspricht genau der Technik, die in Compilern Anwendung findet.

```
void addiere(int & zahl,
             int konstante) {
   *zahl += konstante;
}

void main() {
   int i = 0;
   addiere(&i,1);
   addiere(&i,2);
}
```

```
void main() {
   int i   = 0;
   zahl_1  = &i;
   konstante_1 = 1;
   *zahl_1 += konstante_1;
   zahl_2  = &i;
   konstante_2 = 2;
   *zahl_2 = konstante_2;
}
```

Es ist leicht zu sehen, dass Inlining die Größe des Programms exponentiell anwachsen lassen kann. Zudem besteht das Problem, dass bei Programmen, deren Rekursionstiefe prinzipiell oder nicht praktisch erkennbar beschränkt ist, Inlining keine erschöpfende Kontext-Sensitivität bieten kann.

Vollständigkeit Ein Verifikationsverfahren wird als *vollständig* bezeichnet, falls es für alle korrekten Programme das Ergebnis *Programm ist korrekt* ausgibt.

Bezogen auf ein Programm und seiner LTS Semantik, kann die Analyse eine Semantik annehmen, die höchstens alle Zustandsübergänge erlaubt, die auch die originale Semantik erlauben würde. Im üblichen Sprachgebrauch findet sich auch der Ausdruck, dass die Programmsemantik die von der Analyse berechnete Semantik *simuliert*.

Schlüssigkeit Gibt ein *schlüssiges* Verifikationsverfahren die Meldung *Programm ist korrekt* aus, dann ist dieses Programm korrekt.

Ein schlüssiges Verfahren kann also die Semantik eines Programms überapproximieren. In einem LTS wären also mehr Übergänge möglich als die tatsächliche Semantik eines Programms es zulassen würde. Man kann also sagen, dass die abstrakte Programmsemantik die orginale Programmsemantik simuliert.

Im Folgenden wird eine schlüssige Anwendung des Verfahrens der abstrakten Interpretation von Programmen behandelt, wie sie durch Cousot und Cousot [CC77] standardisiert wurde.

2.2.4 Abstrakte Interpretation

Abstrakte Interpretation [CC77] (AI) bezeichnet sowohl einen Rahmen zur Approximation von mathematischen Strukturen wie auch die Anwendung

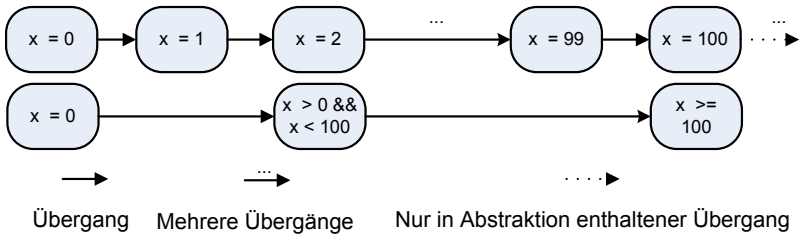


Abbildung 2.2: Die Modellierung der Obermenge der Semantik eines Programms ist nicht eindeutig. Lässt man zusätzliches Verhalten und Zustände zu, ergeben sich unterschiedliche Transitionssysteme, die je nach zu beweisender Eigenschaft ausreichen können.

dieser Technik zur statischen Analyse von Programmen. Überlicherweise wird abstrakte Interpretation zum schlüssigen Beweis von Sicherheitseigenschaften und generellen Datenflusseigenschaften eingesetzt.

Zunächst folgt eine Definition von Programmen wie sie in der AI verstanden werden. Anschließend wird aufgezeigt wie sich die Programmanalyse effizient als Fixpunktanalyse formulieren lässt.

Programme werden in der statischen Analyse oft als Transitionssysteme (TS) dargestellt.

Definition 3 *Ein* (Unlabelled) Transition System τ *ist ein Tripel* $\langle \Sigma, \Sigma_i, t \rangle$ *wobei* Σ *eine Menge von Zuständen umfasst.* $\Sigma_i \subseteq \Sigma$ *ist die Menge der Startzustände und* $t \subseteq \langle \Sigma, \Sigma' \rangle$ *bezeichnet die Übergangsrelation zwischen einem Zustand und möglichen Nachfolgern.*

Betrachten wir beispielhaft das folgende Programm:

```
x := 0; while (x < 100) do x := x + 1;
```

Die Semantik des obigen Programms lässt sich kompakt als Transitionssystem darstellen:

$$\tau = \langle \mathbb{Z}, \{0\}, \{\langle x, x' \rangle | x, x' \in \mathbb{Z} \wedge x' = x + 1\} \rangle$$

Abbildung 2.2 illustriert zwei alternative Abstraktionen des Programms. Die genaue Wahl des Transitionssystems und der Abstraktion hängt von den zu beweisenden Eigenschaften ab. So wird die Tatsache, dass in C oder Java Programmen endliche Wertbereiche vorliegen, in τ ignoriert. Diese Abstraktion ist in Abbildung 2.2 durch den punktierten Übergang dargestellt. Dieser existiert im abstrakten Transitionssystem, nicht aber im konkreten

Programm. Angenommen, die zu untersuchende Eigenschaft sei die Frage, ob `x` nach Ausführung des Programmfragmentes positiv oder negativ sei, dann enthält τ unnötige Details, die möglicherweise zum Beweis der Eigenschaft nicht gebraucht werden. Im unteren Teil der Abbildung 2.2 findet sich ein abstrakteres Transitionssystem welches nur drei Zustände benötigt. Man kann also unendliche Transitionssysteme wie τ zu endlichen Transitionssystemen abstrahieren. Auf diese Weise können Fragen der Korrektheit eines Programms berechenbar gemacht werden.

Abstraktion von Programmen

Die Semantik eines Programms zu berechnen ist im Allgemeinen unmöglich. Abstrahiert man die Semantik geeignet, zum Beispiel durch eine endliche Repräsentation, wird die Semantik berechenbar. Hierzu muss man Überapproximationen zulassen, die die Präzision der berechneten Semantik einschränken. Das Ergebnis kann also Pfade und Zustände enthalten, die nicht im wirklichen Programm vorkommen. Die Analyse ist aber noch immer schlüssig, nicht aber vollständig.

Das Ziel von abstrakter Interpretation ist es, so präzise wie möglich die Semantik – im Sinne der Menge aller möglichen Ausführungen – zu berechnen. Diese Information kann nicht gewonnen werden, indem das Programm auf allen möglichen Eingaben ausgeführt wird. Es kann jedoch möglich sein ein Programm auf abstrakten Werten statt auf konkreten Werten. Statt also `x := x +1;` für alle möglichen Werte von `x` zu berechnen, wird der Wert symbolisch repräsentiert, etwa durch eine mathematische Gleichung $x' = x + 1$. Weiter kann die symbolische Repräsentation der Werte und Zuweisungen einer Abstraktion unterworfen werden welche die Berechenbarkeit erleichtert. Statt beispielsweise obige Zeile symbolisch für alle reelle Zahlen zu interpretieren, kann man abstrakte Zustände einführen: positiv $+$, negativ $-$ und unbestimmt ? (positiv oder negativ). Interpretiert man das Programm als Transitionssystem ist die Übergangsrelation nach Abstraktion folgende

$$\{\langle +, + \rangle \langle -, ? \rangle, \langle ?, ? \rangle\}$$

Durch Abstraktion hat man also erreicht, dass die abstrakte Zustandsmenge und vor allem die Größe der Übergangsrelation kleiner geworden ist. Geht man davon aus, dass `x'` und `x` ganze natürliche Zahlen sind, so hat man sogar das Transitionssystem von einer unendlichen Zustandsmenge auf eine endliche Simulation verkleinert. Möglich ist dies durch die Eigenschaft, dass die Domäne der abstrakten Zustände *Abs* im Allgemeinen die Form einer Potenzmenge der konkreten Zustände hat. Jeder abstrakte Zustand ist also eine Menge von konkreten Zuständen. Folglich ist die abstrakte Übergangs-

relation eine Relation auf dem Kreuzprodukt der Potenzmenge der konkreten Zustände. Im Allgemeinen werden jedoch der Form halber abstrakte Zustände $(+, -, ?)$ neu eingeführt und mittels einer Abstraktionsfunktion $\alpha : \wp(\Sigma) \rightarrow \mathit{Abs}$ sowie einer Konkretisierungsfunktion $\gamma : \mathit{Abs} \rightarrow \wp(\Sigma)$ definiert.

Abstrakte Interpretation nach Cousot

Nach den informellen Einführungen folgt nun die formellere Klärung der Begriffe und Grundideen der abstrakten Interpretation. Die folgende Ausführung hält sich eng an den Formalismus den Cousot [Cou07b] beschreibt. Eingeführt wurde der einheitliche Rahmen der abstrakten Interpretation in [CC77]. Die Publikation enthält jedoch einen heute nicht mehr verwendeten Formalismus, sowie einige Einschränkungen die heute nicht mehr gelten. Wir folgen den Ausführungen in [Cou07b].

Cousot [Cou07b] definiert die *Semantik* $\mathcal{S}[p]$ einer konkreten Software p als formales Modell der Ausführung eines Programms $p \in \mathbb{P}$. Die *semantische Domäne* $\mathcal{D}$ ist die Menge aller solchen formalen Modelle, also $\forall p \in \mathbb{P} : \mathcal{S}[p] \in \mathcal{D}$.

Ein Beispiel ist eine operationelle Semantik eines Programms, die durch die Menge aller maximalen Pfade bestimmt wird. Die Menge aller Pfade ist gleich der Menge aller endlichen und unendlichen Sequenzen von Zuständen aus Σ – wobei zwei aufeinanderfolgende Zustände einem elementaren Programmschritt entsprechen:

$\Sigma^n := [0, n[\mapsto \Sigma$ bezeichne die Menge von Pfaden der Länge n. Die semantische Domäne ist $\mathcal{D} := \wp(\mathcal{T})$ mit $\mathcal{T} := \cup_{n=1}^{\infty} \Sigma^n$. Dieses Verständnis entspricht der Definition der Semantik aus Kapitel 2.

Es gibt jedoch auch andere Definitionen der Semantik die genau auf Prüfeigenschaften zugeschnitten sind und weniger Informationen beinhalten.

Eine Spezifikation ist gemäß Cousot eine verlangte Eigenschaft der Semantik eines Systems. Eine Eigenschaft beschreibt indirekt eine Menge von semantischen Modellen in der sie gilt. Die Menge der Eigenschaften $\mathcal{P}$ ist folglich gleich der Potenzmenge der semantischen Domäne $\mathcal{P} := \wp(\mathcal{D})$. Die stärkste Eigenschaft eines Programms ist seine Semantik $S[p], p \in \mathbb{P}$. Diese Semantik wird von Cousot *Collecting Semantics* (CS) genannt[3]. Letztere wird mit $\mathcal{C}[p]$ bezeichnet.

Ein Programm p *erfüllt* eine Eigenschaft $P \in \mathcal{P}$ falls $\mathcal{S}[p] \in P$. Aufgrund der

[3] Die Terminologie ist nicht einheitlich. In frühen Werken verwendet Cousot den Begriff *Static Semantic*. Neuere Werke verwenden den Begriff *Collecting Semantics*. Jones und Nielson [JN94] führen den Begriff *Accumulating Semantics* ein.

Definition der CS reicht es aus zu zeigen, dass $\mathcal{C}[p] \subseteq P$.

Der Beweis der obigen Aussage steht die Unentscheidbarkeit sowie der praktische Berechnungsaufwand entgegen. $\mathcal{C}^\sharp[p]$ bezeichnet eine schlüssige Überapproximation von $\mathcal{C}[p]$. Es gilt somit $\mathcal{C}[p] \subseteq \mathcal{C}^\sharp[p]$. In Worten ausgedrückt, lässt die Überapproximation also Pfade und Programmverhalten zu, die im ursprünglichen Programm unmöglich wären. Die umgedrehte Richtung ist jedoch ausgeschlossen.

Zusätzlich bezeichne $\mathcal{P}^\sharp \subseteq \mathcal{P}$ eine schlüssige Unterapproximation der Eigenschaft $\mathcal{P}$. Zum Beweis, dass ein Programm eine Eigenschaft erfüllt, reicht es nun zu zeigen, dass $\mathcal{C}^\sharp[p] \subseteq P^\sharp$.

Zur Sicherstellung der Berechenbarkeit der Programmsemantik und Effizienzsteigerung wird in der abstrakten Interpretation eine *abstrakte Domäne* eingeführt, die die *konkrete Domäne* der konkreten Semantik abstrahiert. Eine *abstrakte Domäne* ist ein vollständiger Verband $L = (\wp(\Sigma), \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ mit der partiellen Ordnung $\sqsubseteq$. Letztere wird üblicherweise so gewählt, dass kleinere Elemente *präziser* im Sinne der Programmausführung sind. Meist wird die Domäne punktweise erweitert, so dass sie für jeden Programmpunkt (zum Beispiel nach jeder Anweisung) definiert ist. Beispielsweise kann man auf diese Weise für jeden Programmpunkt angeben, welche Werte die Variablen haben, anstatt global die Wertebereiche zu berechnen.

Das Verhältnis der *konkreten Domäne* $\langle \mathcal{P}, \subseteq \rangle$ und der *abstrakten Domäne* $\langle \mathcal{P}^\sharp, \sqsubseteq \rangle$ wird durch zwei Funktionen $\alpha \in \mathcal{P} \mapsto \mathcal{P}^\sharp$ und $\gamma \in \mathcal{P}^\sharp \mapsto \mathcal{P}$ definiert. γ wird *Konkretisierungsfunktion* genannt. Ihre wichtigste Eigenschaft ist

$$\forall Q_1, Q_2 \in \mathcal{P}^\sharp : (Q_1 \sqsubseteq Q_2) \Rightarrow (\gamma(Q_1) \subseteq \gamma(Q_2))$$

Als Folge hiervon gilt

$$(\mathcal{C}^\sharp[p] \sqsubseteq P^\sharp) \Rightarrow (\gamma(\mathcal{C}^\sharp) \subseteq \gamma(P^\sharp))$$

Aufgrund der Schlüssigkeit gilt $\mathcal{C}[p] \subseteq \gamma(\mathcal{C}^\sharp[p])$ und $\gamma(\mathcal{P}^\sharp) \subseteq \mathcal{P}$. Somit kann die ursprüngliche Beweisverpflichtung $\mathcal{C}[p] \subseteq P$ indirekt bewiesen werden.

Für die Korrektheitsanalyse muss nun die CS eines Programms berechnet werden. Diese ist unberechenbar was sich in der Tatsache äußert, dass zwar die Berechnungsvorschrift zur Fixpunktberechnung angegeben werden kann, jedoch die Konvergenz des Berechnungsverfahrens nicht sichergestellt ist. Eine Fixpunktberechnung iteriert eine Berechnungsvorschrift bis *Konvergenz* erreicht wird, also sich der aktive berechnete Wert, in unserem Fall die berechnete Semantik, nicht mehr ändert. Für die Berechnung des Programmverhaltens ist dies optimalerweise erreicht wenn die berechnete Semantik nur noch das konkrete Verhalten des Programms zulässt.

Für die Konvertierung des Halteproblems in eine Fixpunktberechnung der CS ändert sich also wie erwartet nichts bezogen auf die Problemhärte. Konvergenz bei der Berechnung des Fixpunktes zu erreichen ist im Allgemeinen nicht möglich. Durch geeignete Abstraktion wird die Konvergenz und Effizienz sichergestellt jedoch führt die Abstraktion vom echten Programmverhalten zu unrealistischen Warnungen, dass ein Programm eine Eigenschaft nicht erfüllt.

Ein Beispiel für eine abstrakte Programmdomäne wäre für ein Programm mit nur einer Variable die Potenzmenge möglicher Werte, die die Variable annehmen kann. Die Ordnung $\sqsubseteq$ entspricht dann der Inklusionsbeziehung der Teilmengen. Das größte Element ist gleich der Vereinigung aller Teilmengen, das heißt also der Potenzmenge der konkreten Werte selbst. Das kleinste Element ist der Schnitt aller Teilmengen, das heißt im kleinsten Fall die leere Menge. Gesucht wird nun für jeden Programmpunkt das kleinste Element, das heißt die kleinste Menge von Werten, die eine Variable in einer Zeile annehmen kann.

```
1: x := 0;
2: while (x < 100) do
3:     x := x + 1;
4: ;
```

Für dieses Beispielprogramm ergibt sich folgende CS sortiert nach Zeilen:

Zeile 1: $\{x \to 0\}$

Zeile 2: $\{x \to 0, 1, ..., 100\}$

Zeile 3: $\{x \to 0, 1, ..., 100\}$

Zeile 4: $\{x \to 100\}$

Die Einschränkung der Semantik auf die möglichen Variablenwerte zu jeder Stelle im Programm ist bereits eine Abstraktion einer vollen Pfad-basierten operationellen Semantik – die Menge der Zwischenzustände ist nicht repräsentiert. Man kann jedoch die Domäne noch weiter vereinfachen, indem die Wertebereiche der Variablen selbst abstrahiert werden. Wendet man die abstrakte Domäne $VZ := \{+, -, ?\}$ an ergibt sich folgende CS:

Zeile 1: $\{x \to +\}$

Zeile 2: $\{x \to +\}$

Zeile 3: $\{x \to +\}$

Zeile 4: $\{x \to +\}$

In beiden Fällen erfüllt die abstrakte Domäne VZ den Zweck den Wertebereich aller Variablen an allen Programmpunkten zu erfassen. Im abstrakteren Fall ist die Genauigkeit ausreichend um die Spezifikation *x ist positiv am Ende der Ausführung* zu beweisen.

Als zweite Möglichkeit die Fixpunktberechnung zu beschleunigen oder erst zu ermöglichen bieten sich so genannte *Widening* und *Narrowing* Operatoren an. Für eine genaue Definition der beiden Operatoren sei auf Standardwerke verwiesen [CC77, JN94].

Ein Beispiel hierfür ist die Anwendung eines *join* Operators falls zwei Pfade in einem Programm zusammenlaufen: Hat eine Variable auf beiden Pfaden unterschiedliche Werte, so kann sie nach Zusammenführung einen der beiden möglichen Werte haben. Die beiden Mengen von Werten werden vereinigt (*join*). In dem Beispiel ist die erhaltene Abstraktion pfadinsensitiv, da die Information, auf welchem Pfad ein bestimmter Wert erreicht wird, nicht mehr enthalten ist. Das häufigste Beispiel ist hier die Implementierung einer pfadinsensitiven Datenflussanalyse. Das folgende Programm illustriert diesen Fall. Auf der rechten Seite sind die Werte in der abstrakten Domäne VZ angegeben.

```
1:      if (x > 0)
2:          y = x;                      x := {+}, y := {x}
3:      else
4:          ;                           x := {?}, y := {?}
        // join
5:      ;                               x := {?}, y == {?}

6:      assert( x > 0 => (y == x));
```

Zum Beweis der im `assert` formulierten Bedingung ist es notwendig eine Abstraktion zu wählen, die den Zusammenhang zwischen `x > 0` und den beiden möglichen Definitionen von `y` erhält. Im konkreten Programm gilt die Bedingung. In einer abstrakten Ausführung auf der Domäne VZ mit einem pfadinsensitiven Zusammenführen von abstrakten Werten gilt die Bedingung nicht, da aus beiden Mengen `x := {+}` und `x := {?}` die Menge `x := {?}` resultiert.

Zusammenfassend soll gesagt werden, dass abstrakte Interpretation ein generisches Framework zur Analyse bereitstellt. Die Wahl der Abstraktionen bestimmt die Genauigkeit (Ausdrucksstärke) und Entscheidbarkeit (oder Geschwindigkeit) der Analyse. Viele der klassischen Verfahren, wie zum Beispiel

klassische Datenflussanalysen, lassen sich im Rahmen von abstrakter Interpretation formulieren.

Fallstudien abstrakter Interpretation und statischer Analyse

Astreé [BCC+03b] ist ein Werkzeug, das abstrakte Interpretation für die statische Programmanalyse von Sicherheitseigenschaften von C Programmen implementiert. Analysiert werden können Programme ohne dynamische Speicherallokation und ohne Rekursion. Abgedeckt werden folgende mögliche Probleme in C Programmen:

- Das Vorhandensein von undefiniertem Verhalten gemäß ANSI-C (zum Beispiel eine mögliche Division durch null) [ISO99].
- Die Nutzung von implementierungsspezifischem Verhalten ohne welches das Programm nicht korrekt wäre (beispielsweise die Annahme wie viele Bits zur Kodierung eines Integers verwendet werden).
- Die Verletzung von Benutzer-definierten Programmierrichtlinien.
- Die Einhaltung von Benutzer-definiertem Laufzeitverhalten, das mittels `assert`-ähnlichen Direktiven spezifiziert wurde.

Cousot [Cou07a] schreibt, dass Astreé für Systeme mit Millionen von Codezeilen skaliert. Die Domäne aus der die Programme stammen wird als *Kontroll-Kommando*-orientierte Systeme angegeben.

Polyspace [Pol08] ist eine kommerzielle Implementierung eines zu Astreé vergleichbaren Werkzeugs. Zwar sind die genauen Implementierungsdetails nicht verfügbar, jedoch konnten in einer Kooperation mit der Robert Bosch GmbH Daten über die Leistungsfähigkeit von AI-Ansätzen in der industriellen Anwendung unabhängig von der obigen Autorenfallstudie gewonnen werden (Kapitel 5): Eine Software aus dem Bereich der eingebetteten Systeme mit weniger als hunderttausend Zeilen Code benötigt mit voller Präzision bereits 12 Tage zur Analyse mittels Polyspace. Ein zweites Problem besteht in der Frage welche Analysequalität mit den genannten Laufzeiten erreicht werden kann. Die Leistung abstrakter Interpretation skaliert mit dem Abstraktionsgrad der CS. Mehr Abstraktion führt aufgrund der Unvollständigkeit der Methode jedoch zu mehr Falschmeldungen, die einer Inspektion bedürfen. In der in Kapitel 5 beschriebenen Fallstudie wurden mehr als tausend Warnungen von Polyspace ausgegeben.

Wir betrachten die Skalierbarkeit weiter als offen, da nicht geklärt ist wie Programme eingeschränkt werden müssen um die von Cousot propagierte

Skalierbarkeit zu erreichen. Es ist ungeklärt, ob dynamische Speicherallokation und Rekursion nicht doch in der Praxis zu intensiv genutzt werden, als dass man sie, wie in Polyspace und Astreé, ignorieren könnte. Zusammenfassend betrachtet der Autor dieser Arbeit abstrakte Interpretation als eine ausgereifte Technik zur Programmverifikation. Skalierbarkeit wird jedoch mittels Überapproximation erreicht – dies führt wiederum dazu, dass sich viele konkrete Eigenschaften nicht beweisen lassen. Eine Verringerung der Abstraktion wirft die Frage nach effizienteren Techniken auf, die gerade für eine Detailanalyse optimiert scheinen. Ein Beispiel dieser Art von Techniken ist das Model Checking welches im nächsten Kapitel eingeführt wird.

2.2.5 Model Checking

Model Checking (MC) behandelt die Frage ob eine endliche Struktur M (im logischen Sinne) ein Modell einer Eigenschaft f ist. Die Notation für diese Fragestellung ist $M \models f$. Die Struktur, die untersucht wird, ist das Modell und abstrahiert in unserer Anwendung die operationelle Semantik eines Programms.

Die spezielle Form des Model Checking Problems, die mittels *Bounded Model Checking* im nächsten Abschnitt behandelt wird, ist das *existentielle Model Checking*. Gefragt ist, ob es einen Ablauf oder *Pfad* in einem Modell gibt, auf dem eine Eigenschaft f gilt: $M \models Ef$.

Die Kombination dreier Charakteristiken zeichnet Model Checking und Bounded Model Checking aus (siehe beispielsweise [BCC+03a]):

- Automatisierbarkeit.
- Beschränkung meist auf endliche Systeme.
- Prüfung von Eigenschaften in einer temporalen Aussagenlogik.

Die Erstellung eines Modells (*Modellierung*) ist ein aufwändiger Prozess. Sobald jedoch das Modell erstellt ist, kann dessen Prüfung automatisch vorgenommen werden (siehe Standardlehrbücher wie [CGP99]). Die Komplexität der Prüfung, ob eine Eigenschaft in einem Modell gilt, ist abhängig von der Menge möglicher Zustände und von der Eigenschaft, die geprüft werden soll. Bei endlichen Systemen terminiert ein Model Checking Algorithmus immer – es ist jedoch nicht für alle Programme und Eigenschaften möglich ein geeignetes endliches Modell zu finden.

Eigenschaften, die in einem Modell gelten sollen, werden in temporalen Logiken ausgedrückt. Die häufigsten verwendeten Logiken sind *(propositional) linear temporal logic* (LTL) und *(propositional) computational tree logic*

Tabelle 2.1: Operatoren linearer temporaler Logiken

Symbol	Englischer Name	Bedeutung
Xf	Next	Im nächsten Zustand gilt f
Gf	Globally	Auf allen folgenden Zuständen gilt f
Ff	Finally	Jetzt oder in irgendeinem folgenden Zustand gilt f
$f_1 U f_2$	Until	Bis f_2 gilt, ist f_1 wahr
$f_1 R f_2$	Release	f_2 ist wahr, falls für alle vorherigen Zustände f_1 nicht galt

(CTL). Beide Logiken sind Erweiterungen klassischer Aussagelogiken. In dieser Arbeit werden Eigenschaft untersucht werden, die sich in LTL ausdrücken lassen.

Bisher wurden die Formalismen des Labelled Transitions Systems (siehe Abschnitt 2.1.1) sowie die des Transitions Systems aus der abstrakten Interpretation (Abschnitt 2.2.4) eingeführt. In der Literatur sind Modelle für Model Checking aber meist als *Kripke Strukturen* gegeben:

Definition 4 *Eine* Kripke Struktur *M ist ein Viertupel $M = (\mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$. $\mathcal{S}$ ist wie beim LTS eine Menge von Zuständen. $\mathcal{I} \subseteq \mathcal{S}$ definiert eine Menge von Startzuständen und $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ ist die Übergangsrelation. $\mathcal{L} : \mathcal{S} \mapsto \mathcal{P}(\mathcal{A})$ bezeichnet eine Menge von Labels, die im Unterschied zu LTS,* an Zuständen *gelten oder nicht gelten können. $\mathcal{P}(\mathcal{A})$ bezeichnet hierbei die Potenzmenge der atomaren Aussagen die in einem Zustand gelten können.*

Ein LTS kann in eine Kripke Struktur überführt werden, solange die Menge der Zustände endlich ist. Müller-Olm, Schmitt und Steffen liefern eine ausführliche Einführung in der die Modellierung mittels LTS und Kripke Strukturen verglichen wird [MOSS99].

Zunächst muss jedoch noch der Begriff eines *Pfades* in einer Kripke Struktur eingeführt werden: Analog zu einem Pfad in einem LTS ist ein Pfad π in einer Kripke Struktur definiert als geordnete Sequenz $s_0, s_1, ..., s_i$ $(s_j, 0 \leq i \leq j \in \mathcal{S})$ von Zuständen die der Transitionsrelation $\mathcal{T}, \mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ gehorchen. $\pi(i), i \in \mathbb{N}$ bezeichnet den Zustand an i-ter Stelle in der Sequenz. Ein *Präfix* eines Pfades ist eine zusammenhängende Teilsequenz eines Pfades, die s_0 beinhaltet. Pfade ermöglichen es über das sequentielle Verhalten von Kripke Strukturen zu sprechen.

Im einfachsten Fall wird eine Spezifikation verlangen, dass ein Model niemals einen Fehlerzustand annehmen kann (zum Beispiel eine Division durch null). Übertragen auf Programme bedeutet dies, dass kein (möglicher) Pfad,

Tabelle 2.2: Die formale Semantik einer LTL Formel in Bezug auf einen Pfad π.

$\pi \models a$	gdw.	$a \in \mathcal{L}(\pi(0))$
$\pi \models \neg f$	gdw.	$\pi \not\models f$
$\pi \models f_1 \wedge f_2$	gdw.	$\pi \models f_1$ und $\pi \models f_2$
$\pi \models Xf$	gdw.	$\pi(1) \models f$
$\pi \models Gf$	gdw.	$\pi(i) \models f$ für alle $i \geq 0$
$\pi \models Ff$	gdw.	$\pi(i) \models f$ für mindestes ein $i \geq 0$
$\pi \models f_1 U f_2$	gdw.	$\pi(i) \models f_2$ für mindestes ein $i \geq 0$ und $\pi(j) \models f_1$ für alle $0 \leq j < i$
$\pi \models f_1 R f_2$	gdw.	$\pi(i) \models f_2$ falls für alle $j < i, \pi(j) \not\models f_1$

der von einem Startzustand ausgeht, jemals einen Zustand enthalten (erreichen) kann, der fehlerhaft ist. In komplizierteren Fällen hängt aber eine Spezifikation von endlichen oder unendlichen Sequenzen von Zuständen ab: *Nach einem Zustand in dem f gilt, muss irgendwann ein Zustand folgen, in dem f' gilt.* f und f' sind hierbei Boolesche Aussagen über die Menge der durch $\mathcal{L}$ zugewiesenen atomaren Aussagen. Um die temporale Spezifikation ausdrücken zu können, muss eine Spezifikationssprache Mittel zur Verfügung stellen, um über vorhergehende beziehungsweise nachfolgende Zustände sprechen zu können. In LTL wird hierzu zunächst der Operator **X** eingeführt. Eine Formel Xf drückt aus, dass im nächsten Zustand die Formel f gelten soll. f hängt in einem LTS von dem aktuellen Zustand, also den atomaren Aussagen (Labels), die $\mathcal{L}$ diesem Zustand zuweist, ab. Eine Übersicht der verwendeten Operatoren mit umgangssprachlicher Erklärung findet sich in Tabelle 2.1.

In Erweiterung zu **X** gibt es den **G** Operator. Die Semantik von Gf ist, dass f in allen Zuständen auf einem Pfad gilt. Mittels **G** lassen sich *safety properties* ausdrücken, das heißt Eigenschaften, die besagen, dass ein Ereignis (Fehler) niemals eintritt. Komplementär hierzu sind *liveness properties*, die ausdrücken, dass etwas (positives) irgendwann auf einem Pfad passieren muss. Zur Formulierung letzterer Eigenschaften gibt es in LTL den Operator **F**. Ff ist wahr, falls f im aktuellen oder einem folgenden Zustand wahr wird. $f_1 U f_2$ drückt aus, dass f_1 gilt bis f_2 wahr ist. Danach kann f_1 weiter wahr sein, muss es aber nicht. Beim Operator R gilt der umgedrehte Fall, also $f_1 U f_2 \Leftrightarrow f_2 R f_1$.

Die formale Semantik von LTL Formeln wird mit Bezug auf Pfade in einer Kripke Struktur, beziehungsweise eines LTS, wie in Tabelle 2.2 definiert.

Wir definieren nun die Äquivalenz von Formeln. Zu beachten ist, dass eine Formel f in einer Kripke Struktur M genau dann gilt ($M \models f$), falls $\pi \models f$ für alle initialisierten Pfade gilt.

Definition 5 *Zwei Formeln f und g sind* äquivalent*, in Symbolen $f \equiv g$, falls $M \models f \leftrightarrow M \models g$ für alle Kripke Strukturen M gilt.*

Es gilt somit $\neg F \neg f \equiv Gf$. F und G sind werden *duale* Operatoren genannt.

LTL Formeln sind definiert über alle Pfade einer Struktur M. Modelliert die Struktur ein Programm, betreffen die LTL Eigenschaften alle Pfade in dem Programm. Das Finden eines Gegenbeispiels entspricht der Angabe eines Pfades in dem die Eigenschaft nicht gilt. Ein solcher Pfad wird *Zeuge (Witness)* oder Gegenbeispiel genannt.

In der Logik CTL werden zusätzlich Pfadquantoren eingeführt: Um auszudrücken, ob eine Formel für alle oder nur für einen Pfad gilt, werden für die folgende Präsentation die *Pfadquantoren* E (ein Pfad) und A (alle Pfade) eingeführt. $M \models Af$ bedeutet also, dass auf allen Pfaden f gilt.

Das existentielle Model Checking Problem wird als $M \models Ef$ formalisiert: *Gibt es in einem Modell M einen Pfad auf dem f gilt?* Außer zur Differenzierung des existentiellen Model Checking Problems von anderen Fragstellungen ist die Verwendung der Pfadquantoren E und A im LTL Model Checking nicht erlaubt.

Die Standardtechnik für das Model Checking von LTL-Formeln [LP85] ist die Bildung eines Produktautomaten aus der Kripke Struktur und eines Automaten der die negierte Spezifiktion kodiert. Ist der Produktautomat leer, das heißt kein Fehlerzustand ist erreichbar, ist das System korrekt in Bezug auf die Spezifikation. Eine genauere Beschreibung wird von Clarke et al. in ihrem Standardlehrbuch gegeben [CGP99].

Das Problem welches mit LTL Model Checking gelöst wird, ist die Frage, ob ein LTS oder eine Kripke Struktur Pfade enthält, die einer Spezifikation entsprechen. Das größte Problem von Model Checking ist hierbei die *State space explosion*, die die Größe der untersuchbaren Systeme stark begrenzt. Um Systeme industrieller Komplexität im Bereich der Hardwareverifikation untersuchen zu können wurde von Biere et al. das Verfahren des Bounded Model Checking vorgeschlagen [BCCZ99].

2.2.6 Bounded Model Checking

Bounded Model Checking beantwortet analog zum Model Checking die Frage, ob das sequentielle Verhalten eines Systems Pfade enthält, beziehungsweise zulässt, die einer Spezifikation entsprechen. Im Bounded Model Checking werden jedoch nur *Präfixe* $\pi(0), ..., \pi(i), i \leq k$ eines Pfades π betrachtet. k wird als *Schranke* bezeichnet und fließt in die Formulierung des Bounded Model Checking Problems ein: $M \models_k f$. Als *Suffix* π_i eines Pfades π bezeichnen

wir die Teilsequenz $\pi(i), ..., \pi(k), i \leq k$

Ein wichtiger Gesichtspunkt ist, dass ein endlicher Präfix der *Schleifen* enthält Aussagen über unendliche Pfade zulässt.

Definition 6 *Für $l \leq k$ nennen wir einen Pfad π eine (k,l)-Schleife, falls es einen Übergang $T(\pi(k), \pi(l))$ von $\pi(k)$ nach $\pi(l)$ gibt und der Pfad sich als $\pi = u\dot{v}^{\omega}$ darstellen lässt. Hierbei ist $u = (\pi(0), ..., \pi(l-1))$ und $v = (\pi(l), ..., \pi(k))$. v^{ω} ist die unendliche Wiederholung von v. Wir nennen π eine k-Schleife, falls es ein $l, k \geq l \geq 0$ gibt für das π eine (k,l)-Schleife ist.*

Biere et al. [BCC+03a] berichten, dass die initiale Motivation des Bounded Model Checking die Aussicht war von den großen Fortschritten des SAT-Solving zu profitieren. 1999 war die Größe der Systeme, die mittels BDD basierten Verfahren des symbolischen Model Checking untersucht werden konnten, auf mehrere hundert Flipflops beschränkt. Zur gleichen Zeit waren aussagenlogische Formeln mit tausenden Variablen und Millionen von Klauseln lösbar. Auch wenn die Zahl von Flipflops nicht direkt mit der Zahl von Variablen eines aussagenlogischen Erfüllbarkeitsproblems vergleichbar ist, so bestand laut Biere et al. die Hoffnung durch SAT-Solving bisher unbehandelbare Probleme lösen zu können.

Weiterhin, folgend den Darstellungen in [BCC+03a], nehmen wir an, dass alle Formeln in einer Negationsnormalform vorliegen: Negationen dürfen nur vor atomaren Aussagen stehen, nicht aber vor Operatoren. Negationsnormalformen lassen sich über Dualität der Operatoren und De Morgansche Gesetze effizient berechnen.

Im Gegensatz zum Model Checking kann im Bounded Model Checking nur über Sequenzen einer beschränkten Länge gesprochen werden. Die Definition der Semantik eines beschränkten Modells muss also berücksichtigen, dass der Zustand $\pi(k)$ keinen Nachfolger hat.

Die Notation $\pi \models_k^i f, i \leq k$ drückt aus, dass f an der aktuellen Position i von π gilt. Eine Formel $f := Fp$ ist *gültig* falls es einen Suffix $\pi_i, i \geq 0$ von π gibt, in dem p gilt. Für Fälle in denen ein Pfad zyklisch ist entspricht die Semantik der des unbeschränkten Model Checking. Für *Schleifen* ist die *beschränkte Semantik* wie folgt definiert:

Definition 7 *Sei $k \geq 0$ und π eine k-Schleife. Eine LTL Formel f* gilt *für einen Pfad π mit Schranke k genau dann wenn $\pi \models f$. Wir schreiben $\pi \models_k f$.*

Ist ein Pfad π keine k-Schleife, dann ist eine Formel $f := Fp$ gültig in der *unbeschränkten* Semantik, falls es ein $i \geq 0$ gibt, so dass p im Suffix π_i von

Tabelle 2.3: Die formale beschränkte Semantik einer LTL Formel in Bezug auf einen schleifenfreien Pfad π

$\pi \models_k^i p$	gdw.	$p \in \mathcal{L}(\pi(i))$
$\pi \models_k^i \neg p$	gdw.	$p \notin \mathcal{L}(\pi(i))$
$\pi \models_k^i f \wedge g$	gdw.	$\pi \models_k^i f$ und $\pi \models_k^i g$
$\pi \models_k^i Xf$	gdw.	$i < k$ und $\pi \models_k^{i+1} f$
$\pi \models_k^i Gf$		falsch
$\pi \models_k^i F\phi$	gdw.	$\exists j, i \leq j \leq k . \pi \models_k^j f$
$\pi \models_k^i fUg$	gdw.	$\exists j, i \leq j \leq k . \pi \models_k^j g$ und $\forall n, i \leq n < j . \pi \models_k^n f$
$\pi \models_k^i fRg$	gdw.	$\exists j, i \leq j \leq k . \pi \models_k^j f$ und $\forall n, i \leq n < j . \pi \models_k^n g$

π gilt. Der letzte Zustand auf einer beschränkten Sequenz, $\pi(k)$, hat keinen Nachfolger. Aus diesem Grund kann man die obige rekursive Definition nicht direkt übertragen. Für die Definition der beschränkten Semantik eines Pfades führen wir die Notation $\pi \models_i^k f$ ein. i ist die aktuelle Position im Präfix und $\pi \models_i^k f$ gilt, falls der Suffix π_i die Formel f erfüllt. Aufbauend definieren Biere et al. die *beschränkte Semantik* für Pfade ohne Schleifen wie folgend [BCC+03a]:

Definition 8 *Sei $k \geq 0$ und π ein schleifenfreier Pfad. Eine LTL Formel f ist* gültig *bezüglich eines Pfades π und einer Schranke k – geschrieben $\pi \models_k f$ – genau dann wenn $\pi \models_k^0 f$.*

$\pi \models_k^0 f$ ist rekursiv über die Position im Pfad in Tabelle 2.3 definiert.

Durch die vorangegangenen Definitionen kann nun das Problem des existentiellen Model Checking auf ein beschränktes Problem reduziert werden. Die Beweise für die folgenden Lemma 1 und Lemma 2, sowie für Theorem 1 finden sich in [BCCZ99].

Lemma 1 *Sei f eine LTL Formel und π ein Pfad, dann gilt*

$$\pi \models_k f \Rightarrow \pi \models f$$

Das Lemma folgt aus der konservativen Definition der beschränkten Semantik. $\pi \models_k f$ gilt nur wenn sichergestellt ist, dass die unbekannten Zustände, die durch die Schranke ausgeblendet werden, keinen Einfluss auf die Gültigkeit von f haben. Für eine Eigenschaft wie $f := Gp$ kann die beschränkte Semantik keine Garantien geben - die Eigenschaft gilt also möglicherweise nicht: $\pi \not\models_k f$.

Und weiter

Lemma 2 *Sei f eine LTL Formel und M eine Kripke Struktur. Falls $M \models Ef$ dann gibt es ein $k \geq 0$ so dass $M \models_k Ef$.*

Zyklische Pfade enthalten mindestens eine (k,l)-*Schleife*. Da Bounded Model Checking auf Strukturen mit endlichem Zustandsraum definiert ist, folgt, dass jeder *unendliche Pfad* zyklisch sein muss. Jeder zyklische Pfad hat aber einen endlichen Präfix, der die Schleifen nur einmalig enthält. Aus diesem Grund folgt in Theorem 1, dass immer ein ausreichendes k existiert so dass Bounded Model Checking den Pfad, der eine Eigenschaft erfüllt, findet falls es ihn gibt:

Theorem 1 *Sei f eine LTL Formel und M eine Kripke Struktur. $M \models Ef$ gilt genau dann wenn es ein $k \geq 0$ gibt so dass $M \models_k Ef$.*

Die Semantik des Bounded Model Checking Problems ist somit hinreichend definiert. Im Folgenden wird nun eine Entscheidungsprozedur für die Gültigkeit von LTL Formeln der Form Ef vorgestellt. Motiviert wird die Entscheidungsprozedur durch das Problem, dass im Model Checking binäre Entscheidungsdiagramme (*Binary-Decision-Diagram*, BDD) verwendet werden, um die Gültigkeit von Formeln zu entscheiden. Durch ein Entscheidungsdiagramm kann man effizient erfüllbare Belegungen für boolesche Formeln berechnen. In der praktischen Anwendung ist jedoch der benötigte Speicherplatz – linear in der Anzahl möglicher Belegungen – zu groß um erreichbare Zustände für komplexe Hardwareschaltungen kodieren zu können. Im BMC kann auf die Verwendung von Entscheidungsdiagrammen durch den Einsatz moderner Verfahren des SAT-Solving verzichtet werden. Zwar ändert sich die zeitliche Komponente der Berechnungskomplexität nicht, jedoch ist der benötigte Speicherplatz in praktischen Anwendungen logarithmisch in der Anzahl der Zustände.

Im Folgenden stellen wir dar wie die Gültigkeitsprüfung in ein aussagenlogisches Erfüllbarkeitsproblem übertragen werden kann [BCCZ99]. Das effiziente Lösen des Erfüllbarkeitsproblems (SAT) ist in der Literatur beschrieben [PBG05].

Die Reduktion verlangt als Eingabe eine Kripke Struktur M, eine Formel f und eine Schranke k. Als Ausgabe wird eine aussagenlogische Formel $[\![M, f]\!]_k$ produziert. Sei $s_0, ..., s_k$ eine endliche Sequenz von Zuständen auf dem Pfad π. s_i kodiert einen Zustand zum Zeitpunkt i mittels Zuweisungen zu aussagenlogischen Variablen, die die endliche Menge der atomaren Zustandsinformationen gemäß $\mathcal{L}$ kodieren, so dass $[\![M, f]\!]_k$ erfüllbar ist genau dann wenn π ein Zeuge für f ist.

Die Konstruktion der aussagenlogischen Formel erfolgt in drei Teilen. Zunächst wird die Menge der Pfade, die in der Kripke Struktur möglich sind,

durch eine Formel $[\![M]\!]_k$ beschrieben. Nachfolgend wird eine Bedingung L_k formuliert, die wahr ist, falls der Pfad π eine Schleife enthält. Als dritter Bestandteil wird die Frage kodiert, ob ein Pfad π eine Eigenschaft f erfüllt.

Definition 9 *Für eine Kripke Struktur M und eine Schranke $k \geq 0$ ist die* Entfaltung $[\![M]\!]_k$ *der Transitionsrelation T definiert als*

$$[\![M]\!]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

T muss also bereits als aussagenlogische Kodierung der möglichen Zustandsübergänge als Relation vorliegen.

Als nächstes betrachten wir die *Schleifenbedingung* L_k, die kodiert ob eine Schleife von einem Zustand s_k zu sich selbst oder einem früheren Zustand existiert.

Definition 10 *Die Schleifenbedingung L_k ist wahr genau dann wenn ein Übergang vom Zustand s_k zu sich selbst oder zu einem vorhergehenden Zustand existiert:*

$$L_k := \bigvee_{l=0}^{k} T(s_k, s_l)$$

Die Übersetzung der LTL Formel hängt davon ab, ob ein Pfad schleifenfrei ist. Zunächst behandeln wir den Fall, dass eine Schleife vorliegt. Bei der Notation von Zwischenformeln ${}_l[\![p]\!]_l^i$ steht l für den Schleifenanfang und i für die momentane Position in π.

Definition 11 *Der* Nachfolger $succ(i)$ *in einer (l,k)-Schleife ist definiert als $succ(i) := i+1$ für $i < k$ und $succ(i) := l$ für $i = k$.*

Definition 12 *Die* Übersetzung $[\![f]\!]$ *einer LTL Formel f für eine k-Schleife π ist definiert als*

${}_l[\![p]\!]_k^i := p(s_i)$	${}_l[\![Gf]\!]_k^i :=_l [\![f]\!]_k^i \wedge_l [\![Gf]\!]_k^{succ(i)}$
${}_l[\![\neg p]\!]_k^i := \neg p(s_i)$	${}_l[\![Ff]\!]_k^i :=_l [\![f]\!]_k^i \vee_l [\![Ff]\!]_k^{succ(i)}$
${}_l[\![f \vee g]\!]_k^i :=_l [\![f]\!]_k^i \vee_l [\![g]\!]_k^i$	${}_l[\![fUg]\!]_k^i :=_l [\![g]\!]_k^i \vee ({}_l[\![f]\!]_k^i \wedge_l [\![fUg]\!]_k^{succ(i)})$
${}_l[\![f \wedge g]\!]_k^i :=_l [\![f]\!]_k^i \wedge_l [\![g]\!]_k^i$	${}_l[\![fRg]\!]_k^i :=_l [\![g]\!]_k^i \wedge ({}_l[\![f]\!]_k^i \wedge_l [\![fRg]\!]_k^{succ(i)})$
	${}_l[\![Xf]\!]_k^i :=_l [\![f]\!]_k^{succ(i)}$

Enthält ein Pfad keine Schleife findet ein Spezialfall der obigen Definition Anwendung.

Definition 13 *Die* Übersetzung $[\![f]\!]$ *einer LTL Formel* f *für Pfade ohne Schleife ist induktiv definiert als*

Schritt für $0 \leq i \leq k$	
$[\![p]\!]_k^i := p(s_i)$	$[\![Gf]\!]_k^i := [\![f]\!]_k^i \wedge [\![Gf]\!]_k^{i+1}$
$[\![\neg p]\!]_k^i := \neg p(s_i)$	$[\![Ff]\!]_k^i := [\![f]\!]_k^i \vee [\![Ff]\!]_k^{i+1}$
$[\![f \vee g]\!]_k^i := [\![f]\!]_k^i \vee [\![g]\!]_k^i$	$[\![fUg]\!]_k^i := [\![g]\!]_k^i \vee ([\![f]\!]_k^i \wedge [\![fUg]\!]_k^{i+1})$
$[\![f \wedge g]\!]_k^i := [\![f]\!]_k^i \wedge [\![g]\!]_k^i$	$[\![fRg]\!]_k^i := [\![g]\!]_k^i \wedge ([\![f]\!]_k^i \wedge [\![fRg]\!]_k^{i+1})$
	$[\![Xf]\!]_k^i := [\![f]\!]_k^{i+1}$
Basisfall:	
	$[\![f]\!]_k^{k+1} := 0$

Die drei Komponenten der Übersetzung des Problems in eine aussagenlogische Formel können nun zusammengefügt werden.

Definition 14 *Die (aussagenlogische)* Übersetzung eines Bounded Model Checking Problems *für eine Kripke Struktur* M*, eine Schranke* $k \geq 0$ *und eine LTL Formel* f *ist definiert als*

$$[\![M, f]\!]_k := [\![M]\!]_k \wedge \left((\neg L_k \wedge [\![f]\!]_k^0) \vee \bigvee_{l=0}^{k} \left(T(s_k, s_l) \wedge_l [\![f]\!]_k^0 \right) \right)$$

Es gilt nach Biere et al. das folgende Theorem [BCCZ99]

Theorem 2 $[\![M, f]\!]_k$ *ist erfüllbar genau dann wenn* $M \models_k Ef$*.*

Die *Übersetzung* von $M \models_k Ef$ in ein aussagenlogisches Erfüllbarkeitsproblem ist also schlüssig und vollständig bezüglich der *beschränkten* Semantik. Im Folgenden beschreiben wir wann Bounded Model Checking selbst vollständig und schlüssig bezüglich der *unbeschränkten Semantik* sein kann.

Vollständiges BMC

Biere et al. behandeln neben den bisher besprochenen Fällen auch die Frage wie hoch die Schranke k gewählt werden muss, um sich sicher sein zu können, dass man vollständige Ergebnisse erhält. Wir folgen auch in diesem Abschnitt den Ausführungen in [BCC+03a].

Für jedes Model M und jede Eigenschaft p beziehungsweise Formel f existiert eine maximale Konstante K für die die beschränke Semantik gleich der normalen Model Checking-Semantik ist. Diese Konstante wird *Vollständigkeitsschranke (Completeness Threshold)* genannt.

Im praktischen Einsatz des Bounded Model Checking werden meist Eigenschaften untersucht, die die Form Gf haben, wobei f selbst keine temporalen Operatoren mehr enthält. Für diese eingeschränkten Fälle ist K schlimmstenfalls gleich der Anzahl der Zustände die M umfasst. Ein solches Beispiel ist die Kodierung eines Zählers der deterministisch von 0 bis n zählt. Der minimale Wert von K, für den alle Zustände eines Modells erreicht werden können, wird als *Erreichbarkeitsdurchmesser rd (Reachability Diameter)* bezeichnet [BCC+03a]. Häufig wird rd durch eine Approximation nach oben beschränkt: der *Rekurrenzdurchmesser (Recurrence Diameter)* ist definiert als die Länge des längsten zyklenfreien Pfades ausgehend von einem Startzustand.

Beide Maße erlauben es, Bounded Model Checking vollständig zu machen, beziehungsweise die Vollständigkeit zu testen. Ein offensichtlicher Nachteil ist, dass die Berechnung der obigen Durchmesser im Allgemeinen genauso schwierig ist, wie die Berechnung der Gültigkeit der Formel selbst. In der Praxis kann K durch den Nutzer erraten, iterativ bestimmt oder geeignet approximiert werden. Des Weiteren kann die Vollständigkeitsprüfung gleichzeitig mit dem BMC vollzogen werden, was zu einem geringeren Aufwand führt. Letztere Technik wird beim Software Model Checking (Abschnitt 2.2.7) noch erläutert werden.

Als Ergänzung zu den obigen *Sicherheitseigenschaften*, die sich in der Form Gp ausdrücken lassen, haben in der Praxis *Lebendigkeitseigenschaften* der Form AFp eine große Bedeutung. Diese lassen sich wie folgend kodieren

Definition 15 *Die Übersetzung des Gültigkeitsproblems einer Lebendigkeitseigenschaft AFp für ein Modell M ist definiert als*

$$[\![M, AFp]\!]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \rightarrow \bigvee_{i=0}^{k} p(s_i)$$

Das folgende Theorem überbrückt die Differenz zwischen dem beschränkten und unbeschränkten Fall

Theorem 3

$$M \models AFp \text{ gdw. } \exists k \text{ für das } [\![M, AFp]\!]_k \text{ gültig ist.}$$

Gemäß diesem Theorem wird ein k gesucht welches $[\![M, AFp]\!]_k$ erfüllbar macht. Durch Anwendung von Definition 15 erhält man direkt eine Entscheidungsprozedur, falls die Eigenschaft gilt. Falls die Eigenschaft nicht gilt, kann man zusätzlich die Negation der Eigenschaft prüfen: $M \not\models AFp$ ist

gleich $M \models EG\neg p$ und somit durch das Verfahren zur Prüfung von Sicherheitseigenschaften prüfbar. Durch beide Prüfungen zusammen erhält man eine vollständige Entscheidungsprozedur für Lebendigkeitseigenschaften. Auch hier muss k so groß gewählt werden, dass Rekurrenz- oder Erreichbarkeitsdurchmesser abgedeckt werden.

Induktion Die Anwendung von Bounded Model Checking wird in der Praxis dadurch erschwert, dass die Durchmesser realer Systeme sehr groß sein können. Gründe hierfür sind – wie bereits angesprochen – Zähler oder Berechnungschleifen. Aus der Mathematik kennt man jedoch das Verfahren der Induktion, durch das Beweise auf rekursiv definierten Strukturen (zum Beispiel die natürlichen Zahlen nach Peano Axiomatisierung) in zwei (endliche) Fälle geteilt werden können: Induktionsschritt und Basisfall (Induktionsanker). Ein ähnliches Verfahren haben Sheeran et al. für das BMC von Sicherheitseigenschaften vorgeschlagen [SSS00].

Der Basisfall ist trivialerweise gegeben durch die Überprüfung, dass im initialen Zustand die Eigenschaft gilt. Zusätzlich muss eine induktive Invariante gefunden werden, die die Sicherheitseigenschaft impliziert. Induktivität der Invariante heißt, dass aus der Gültigkeit bus zur Schranke k, auch die Gültigkeit der Invariante bis zur Schranke $k + 1$ folgen muss.

Weitere Verfahren Verfahren, die BMC effizient für den unbeschränkten Fall erweitern, werden von Prasat et al. [PBG05] aufgeführt. Da diese nicht weiter in dieser Arbeit verwendet werden beschränken wir uns auf eine kurze Erwähnung.

Interpolation McMillan [McM03] präsentiert eine Methode um mittels Craig-Interpolanten BMC für den unbeschränkten Fall effizient berechnen zu können. Ein Interpolant ist eine Approximation erreichbarer Zustände die 1. eine Obermenge der tatsächlich erreichbaren Zustände sein muss und 2. die Korrektheit der Eigenschaft implizieren muss. Da die Obermenge der erreichbaren Zustände keinen Fehlerzustand enthält, kann für diesen Fall auch geschlossen werden, dass die tatsächliche Menge von erreichbaren Zuständen keinen Fehler zulässt.

QBF Die Überprüfung, ob Sicherheitseigenschaften in einem (endlichen) Modell gelten, kann effizient ohne die in modernen Model Checking Algorithmen verwendete Fixpunktberechnung gelöst werden [PBG05]. Hierzu wird das Problem der Erreichbarkeit in eine *Quantifizierte Boolesche Formel (QBF)* kodiert. Die Formel enthält nur eine Kopie der Übergangsrelation und nicht eine Kopie für jeden der k-Schritte im Bounded Model

Checking. Es werden jedoch dreimal so viele Variablen wie die in der ursprünglichen Zustandsbeschreibung benötigt. Die Erfüllbarkeitsprüfung für QBF ist PSPACE-vollständig.

2.2.7 Software Model Checking

Die bisher gezeigten Model Checking und Bounded Model Checking Techniken nehmen als Eingabe ein LTS oder eine Kripke Struktur M und eine LTL-Formel f und berechnen ob $M \models f$ gilt. Beim *Software Model Checking (SMC)* ist die Eingabe ein Programm P und eine Eigenschaft p. Ausgegeben wird, ob das Programm die Eigenschaft p erfüllt, also ob es ein Modell für die Eigenschaft ist. Software Model Checking und Software Bounded Model Checking verwenden eine Reduktion des Problems auf Model Checking, respektive Bounded Model Checking: Das Eingabeprogamm wird in ein LTS oder eine Kripke Struktur übersetzt. Im Unterschied zu Model Checking wird die Eigenschaft p nicht in LTL formuliert, sondern direkt in die Struktur kodiert (zum Beispiel in [BR02, CKL04, CKSY05]). Dies ist möglich da meist nur Sicherheitseigenschaften Gf geprüft werden. Für jeden Zustand wird dann geprüft, ob f gilt und eine neue Variable eingeführt, die kodiert, ob es sich um einen Fehlerzustand handelt. Anschließend wird nur noch geprüft, ob ein Fehlerzustand erreichbar ist, ohne dass eine LTL Formel explizit übersetzt werden müsste. Die Prüfung auf Sicherheitseigenschaften wird reduziert auf das *Erreichbarkeitsproblem*. Letzteres setzt eine Lösung des Halteproblems voraus.

Es gibt bisher noch keinen einheitlichen Formalismus für die Kodierung von Eigenschaften, die im Software Model Checking geprüft werden. Häufig werden Sicherheitseigenschaften in Paaren von `assert` / `assume` Anweisungen formuliert. `assert` Befehle drücken aus, dass eine Eigenschaft an der Stelle im Programm gelten solll, in der die Anweisung steht. `assume` Anweisungen kodieren Annahmen, die in einem Korrektheitsbeweis verwendet werden dürfen. Ein Beispiel:

```
assert(b!=0);
i = a / b;
```

Zu beweisen ist also, dass `b` nicht null ist, wenn die zweite Zeile erreicht wird. Andere Spezifikationssprachen sind SLIC [BR01], SLICx (Abschnitt 4.1) und JML [LPC+07]. Das verwendete Werkzeug CBMC [CKL04] implementiert Software Bounded Model Checking mittels `assert` / `assume` Spezifikationen.

Häufig wird die Umwandlung des Programms und der Eigenschaft nicht erst explizit berechnet, sondern implizit dargestellt. Die erneute Umwandlung

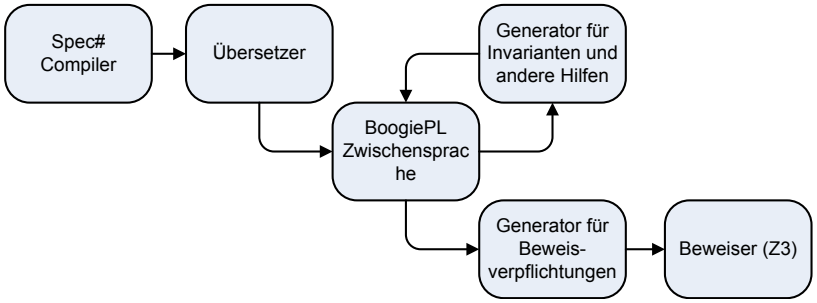


Abbildung 2.3: Das Spec# Programmiersystem enthält zum Zweck der Verifikation viele Schichten in denen unterschiedliche Front- und Backends eingesetzt werden können. Allen gemein ist die Umwandlung eines Programms in Spec# (oder auch C,C# und C++) nach BoogiePL. Letztere ist eine (imperative) Zwischensprache die die letzte (programmähnliche) Stufe vor der Verifikation darstellt. Innerhalb des Microsoft Research Projektes SLAM [BR02] ist BoogiePL zentrales Austauschformat zwischen verschiedenen Werkzeugen.

beispielsweise beim Software Bounded Model Checking ermöglicht die direkte Transformation eines Programms und einer Eigenschaft in eine aussagenlogische Formel, die erfüllbar ist genau dann, wenn ein Fehlerzustand erreichbar ist.

Im SLAM-Projekt [BR02] tritt der entgegengesetzte Fall auf. Statt ein Verifikationsproblem direkt in eine Formel zu kodieren wird eine Kaskade von Transformationen durchlaufen, die eine hohe Wiederverwendbarkeit der einzelnen Werkzeugkomponenten gewährleistet. Abbildung 2.3 stellt den mehrschrittigen Transformationsprozess im SLAM-Projekt dar. Ein Werkzeug, das diesen Prozess implementiert, ist der *Static Driver Verifier* [BBC+06].

Programm-Programm Transformation

Programm-Programm Transformationen übersetzen ein Programm in ein anderes. Ziel ist meist die Normalisierung des Programms, also beispielsweise das Entfernen von im Standard nicht definierten syntaktischen Ausdrücken. Beispiele für verletzte Wohldefiniertheit sind auf Seite 7 gegeben.

Generell können folgende Transformationen unterschieden werden

- Programm-Programm Transformationen, bei denen Quell- und Zielsprache gleich sind oder die Zielsprache eine Teilmenge der Quellsprache ist (vgl. das CIL Werkzeug [NMRW02]).
- Programm-Übersetzungen in eine andere Programmiersprache wie etwa die Erzeugung von Assemblerdateien in einem Compiler.

- Anreicherungen eines Programms mit Spezifikationen mittels `assert / assume` Anweisungen.

Programm-zu-Programm Transformationen dienen im Bereich Programmverifikation meist dazu, Eingaben für Verifikationwerkzeuge zu vereinfachen. Beispielsweise können aus einem imperativen Programm Seiteneffekte entfernt werden. Dies ändert nichts an der Programmsemantik, erleichtert aber die Verarbeitung des Programms in den folgenden Schritten. Die Ersetzung von `i++;` durch `i = i + 1;` wäre ein Beispiel für eine solche Transformation[4].

Programm-Übersetzungen in eine andere Sprache sind eng verwandt mit obiger Programmvereinfachung. Ein Beispiel findet sich in dem SLAM Projekt: Programme der Sprachen C,C++,C# und Spec# werden in die Zwischensprache BoogiePL [DL05] übersetzt. BoogiePL kann alle Aspekte obiger Sprachen in einer einheitlichen Form wiedergeben.

Wie zuvor erwähnt werden Sicherheitseigenschaften beim Model Checking und BMC geprüft, indem ein Produktautomat – bestehend aus dem Automat, der die Programmsemantik kodiert, sowie einem Automat der die Sicherheitseigenschaft kodiert – gebildet und anschließend das Erreichbarkeitsproblem gelöst wird. Es ist im Bereich der Programmverifikation möglich und zweckmäßig die Bildung dieses Produktautomaten auf der Ebene der Programme selbst zu vollziehen.

SLICx [PK07] ist eine Spezifikationssprache für Schnittstellenregeln für C-Programme die Sicherheitseigenschaften entsprechen. SLICx Regeln, die in Kapitel 4 genauer eingeführt werden, können in die Sprache C übersetzt werden. Nach Übersetzung kann man statt einen Produktautomaten zu berechnen direkt die Regeln in das zu verifizierende Programm mittels `assert / assume` einfügen.

Bisher wurde die Methode des Bounded Model Checking eingeführt. Nun wird genau beschrieben wie C Programme beim Software Bounded Model Checking – wie in CBMC [CKL04] implementiert – direkt in aussagenlogische Formeln übersetzt werden können. Hierbei finden eine Reihe von Programm-Programm Transformationen Anwendung.

2.2.8 SAT-basiertes Software Bounded Model Checking

Ähnlich wie beim BMC wird im Software Bounded Model Checking (SBMC) das Gültigkeitsproblem mittels einer beschränkten Semantik in ein aussagenlogisches Erfüllbarkeitsproblem umgewandelt. Die Schranke im Bounded

[4] Außerhalb des Verifikationsbereiches werden Transformationen auch zum *Refactoring* im Software Engineering eingesetzt.

Model Checking betrifft die Länge der Sequenzen, die betrachtet werden. Im Software Bounded Model Checking ist mit der Schranke die Anzahl von rekursiven Aufrufen und Schleifenausführungen gemeint: Eine Schranke $k = 5$ bedingt also, dass nur Programmausführungen betrachtet werden, in denen eine Schleife wie `while(true) P1;` maximal fünf mal ausgeführt wird. Reicht dieses *Abrollen* oder *Entfalten* von Schleifen nicht aus, wird die Gültigkeit konservativ approximiert.

Im Folgenden stellen wir diese Umwandlung im Detail vor, wobei wir den Ausführungen von Clarke et al. folgen [CKY03]. Wir beschränken uns auf die Transformation von ANSI-C Programmen wobei die vorgestellten Techniken prinzipiell auch für andere imperative Programmiersprachen anwendbar sind. Nennenswerte Unterschiede zwischen den Semantiken von Java und C Programmen sind beispielsweise in [Gla07] zusammengefasst.

In diesem Abschnitt definieren wir, wie die beschränkte Semantik eines C Programms in eine aussagenlogische Formel übersetzt werden kann. Hierzu definieren wir, mit welchen Zuständen ein C Programm modelliert und welche Zustandsübergänge angenommen werden müssen. Grundsätzlich ist das Problem zu lösen, dass eine Sprache wie C Übergänge nicht durch eine Relation von momentanen auf folgende Zustandsmengen beschreibt.

Im Besonderen muss geklärt werden, wie Schleifen, Referenzen, Typen und Zeiger übersetzt werden können. In den folgenden Paragraphen wird die Übersetzung stückweise wie durch Clarke et al. beschrieben [CKY03] eingeführt. Programm-Programm Transformationen werden zunächst eingesetzt um eine einfache Normalform zu konstruieren.

ANSI-C nach GOTO/WHILE Als ersten Schritt ihrer Reduktion des Software Bounded Model Checking Problems wandeln Clark et al. [CKY03] ein ANSI-C Programm in ein äquivalentes GOTO-Programm mit WHILE-Schleifen um. Hierzu werden folgende atomaren Schritte ausgeführt:

1. `break` und `continue` Anweisungen werden durch äquivalente `goto` Anweisungen ersetzt. Hierzu werden gegebenenfalls neue Labels in das Programm eingefügt.

2. `switch` Blöcke mit `case` Fällen werden durch kaskadierte `if` und `goto` Anweisungen ersetzt.

3. `for` Schleifen werden in `while` Schleifen umgewandelt. Aus `for (e1; e2; e3) P;` wird `e1; while(e2) {P; e3;}`

4. `do {P} while (e);` wird ersetzt durch `P; while (e) P;`.

Die obigen Ersetzungen werden ausgeführt bis keine Ersetzung mehr möglich ist. Im Folgenden beschränken wir das Programm auf Ausführungen in denen Schleifen und Rekursion nur endlich oft vorkommen können. Dies entspricht dem Übergang von einem Model Checking zu einem Bounded Model Checking Problem.

GOTO/WHILE nach Bounded GOTO Anschließend gibt es drei Möglichkeiten, Zyklen in einem Programm vorliegen zu haben. Diese werden zunächst entfernt um anschließend eine Ordnung der Programmbefehle erhalten zu können. Diese Ordnung kann später genutzt werden, um durch geeignete Variablenersetzungen den zeitlichen Ablauf des Programms kodieren zu können.

1. `while` Schleifen werden in Abhängigkeit von der Schranke k des SBMC entfaltet. Eine Entfaltung transformiert den Ausdruck `while(e) P;` in `if (e) P; while(e) P;`. Termination der Entfaltung wird erreicht, indem die Entfaltung genau k mal ausgeführt wird. Nach der k-ten Ausführung wird die restliche Schleife wie folgend ersetzt:
   `while(e) P;` durch `if (e) assert(false);`
   Dies hat zur Folge, dass falls die Entfaltung der Schleife nicht ausreichend war um die möglichen Ausführungen vollständig zu umfassen, das `assert` zu einer Meldung führt. Indem jedes `assert` auf mögliche Verletzungen geprüft wird, erhält man so eine Warnung (siehe die Begriffe des Rekurrenzdurchmessers im BMC).

2. Aufrufe von Funktionen werden entfaltet, indem der Aufruf durch eine Kopie der Funktionsdefinition ersetzt wird (Inlining). Hierbei wird die Bedeutung von Parameterübergaben durch explizite Zuweisungen simuliert. Rekursive Funktionen werden genau wie Schleifen k-mal entfaltet. Dass die Entfaltung ausreichend (tief genug) ist, kann wiederum mittels eines `asserts` geprüft werden. Diese Art von Entfaltung von Funktionen wird in der Literatur als *Inlining* bezeichnet. Da Inlining zu einer potentiellen Vergrößerung des Programms führen kann, die exponentiell in der Tiefe der Rekursion ist, ist diese Technik besonders für systemnahe Programme geeignet, in denen nur vereinzelt rekursive Aufrufe vorkommen.

3. `goto` Anweisungen stellen Schleifen dar, falls das Ziel des Sprungs im Programmfluss vor der `goto`-Anweisung selbst liegt (*backward-jump*). In solchen Fällen wird verfahren wie in dem Fall einer Schleife ohne Abbruchbedingung.

Die speziellen `assert` Anweisungen, die im obigen Schritt eingeführt wurden, werden in der Literatur *unwinding assertions* genannt [CKY03]. Falls

nur nach Fehlern in einem Programm gesucht werden soll, das heißt die Abdeckung aller möglichen Schleifenausführen nicht sichergestellt werden muss, können diese in CBMC [CKL04] vor der Prüfung deaktiviert werden.

Eine Deaktivierung führt dazu, dass eine Unterapproximation statt einer Überapproximation vorgenommen wird: Enthält ein Programm einen Fehler nur nach einem $k+1$-ten Schleifendurchlauf, liefert der Algorithmus das Ergebnis *Das Programm ist korrekt* obwohl dies nicht der Fall ist. In dem Fall, dass unwinding assertions aktiv sind und kein Fehler, aber eine $k+1$ Schleifenausführung vorliegt, meldet der Algorithmus konservativ *Das Programm kann nicht als korrekt bewiesen werden.* Als Grund wird zusätzlich die Verletzung einer Schranke angegeben.

Als nächstes wird das Programm in die *Single Static Assignment* Form überführt: das heißt für jede Variable existiert im Programmfluss nur noch eine Zuweisung.

Bounded GOTO nach Single Assignment Bounded GOTO Das bisher erstellte Programm ist eine beschränkte Entfaltung des ursprünglichen Programms. Als nächstes wird die Umbenennung der Variablen vorgenommen, um die zeitliche Abfolge implizit in einer Formel kodieren zu können.

Hierzu wird für jede Variable v im Programm eine Menge von versionierten Variablenversionen $v_1, ...$ erstellt. Es gilt, dass v_0 die initiale Belegung der Variablen ist. Bei jeder im Programmfluss nachfolgenden Zuweisung dieser Variablen wird eine neue Version angelegt. Zuweisungen können jeweils den Zusammenhang zwischen einer Programmvariable v_{i+1} und v_i beschreiben.

Die Anweisungen `x = x + 1; x = x + 3;` würden folgendermaßen Übersetzt: $x_1 = x_0 + 1$ und $x_2 = x_1 + 3$. Die zeitliche Abfolge der Zuweisungen spielt also keine Rolle mehr.

Das Programm liegt nun in einer so genannten *Single Static Assignment (SSA)* Form vor. Eine Dokumentation effizienter Berechnungsalgorithmen wird von Cytron et al. [CFR+89] gegeben. Jede Zuweisung einer Variable kann sich *nur* auf die vorhergehende Version ihrer selbst, beziehungsweise auf die aktuellen Versionen aller anderen Variablen beziehen.

Entfernung von Seiteneffekten Seiteneffekte umfassen, nach Entfernung von Funktionen durch Inlining, pre- und post- Inkrement- und Dekrementierung sowie die Seiteneffektzuweisungen wie beispielsweise `+=`. Erstere können durch zusätzliche Zuweisungen vor dem zu transformierenden Befehl entfernt werden. Zusätzliche Versionen von Variablen können gegebenenfalls neu erzeugt werden. Letztere Operatoren sind vom Standard wie vorgesehen durch äquivalente Ausdrücke ohne Seiteneffekte ersetztbar. So wird aus `x +=`

`1;` die Anweisung `x = x + 1;`.

Der ANSI-C Standard erlaubt compiler- und architekturspezifisches Verhalten bei der Auswertungsreihenfolge von Seiteneffekten. In [CKL04] wird jede mögliche Auswertungsreihenfolge berücksichtigt. Zwar sind dies exponentiell viele in der Anzahl der Seiteneffekte, jedoch ist es unüblich oder sogar durch Kodierrichtlinien (vgl. den MISRA-C [Aut04] Standard) verboten Seiteneffekte in dieser Weise zu kombinieren.

CIL [NMRW02] ist ein Tool zur Programm-Programm Transformation das Seiteneffekte aus C Programmen entfernt. Hierbei wird ein spezieller Compiler oder eine Architektur vorgegeben und dann die korrekte Semantik bezogen auf eine konkrete Implementierung umgesetzt. CIL betrachtet neben dieser Undefiniertheit des Standards auch andere problematische, das heißt undefinierte, Semantikbestandteile und liefert implementierungskonforme Übersetzungen. Für die Verifikation von Software, die in Dialekten von ANSI-C geschrieben ist, kann dies eine enorme Erleichterung sein, das Implementierungsverhalten von der eigentlichen Analyse zu trennen. Wir werden in Kapitel 4 noch näher auf diese Problematik eingehen.

Entfernen von GOTO Die Auswirkung eines `goto` Befehls in einem Programm ist derart, dass der Code am Sprungziel ausgeführt wird. Wird ein `goto` Befehl ausgeführt unter einer bestimmten Bedingung, zum Beispiel `if (e) goto END;`, dann ist `e` hinreichende Bedingung für die Ausführung des Quellcodes nach `END`. Die Konjunktion aller Bedingungen aller `goto END;` Anweisungen – zusammen mit der Bedingung für ein normales Erreichen der Zeile mit `END` – ist die notwendige Bedingung für die Ausführung. Ein Beispiel:

```
if (e1) {
  if (e2)
    goto END;

  x = 4;
} else {
  if (e3)
    goto END;
}


P;
END: ;
```

```
if (e1) {


  if (!e2)
     x = 4;
} else {


}
if ( ( e1 && !e2) ||
     (!e1 && !e3) )
   P;
END: ;
```

Der rechte Teil beschreibt den Quellcode nach Ersetzung der `goto` Befehle. Jeder Programmteil wird durch die notwendige und hinreichende Bedingung für die Ausführung geschützt. Diese Bedingung wird `Guard` genannt.

Nach Erhalt des Programms ähnelt das Programm bis auf das Vorkommen von `if`, `'='`, `assume`, `assert` sowie von Datentypen einem Gleichungssystem.

Umwandlung in ein Bit-Vektor Gleichungssystem Das Programm wird nun in eine Menge von Gleichungen übersetzt. Innerhalb der Gleichungen werden getypte C Symbole und arithmetische Operationen verwendet. Zusätzlich enthalten sind aussagenlogische Operatoren wie $\vee$ und $\wedge$. Ein Beispiel für eine solche Menge ist

$$\{x_1 == x_0 + 1, y_1 == ((x1 >> 1) == y0)\}$$

Die Transformation in ein Gleichungssystem wird über zwei Funktionen realisiert. Hierbei ist p das Eingabeprogramm und g die Bedingung unter der das Eingabeprogramm ausgeführt wird (*guard*). Die Funktion $C(p, g)$ kodiert mögliche Übergänge, also die Transitionsrelation. C umfasst also die Semantik des Programms und die künstlich über `assume` eingefügten Bedingungen. Die Funktion $P(p, g)$ kodiert die Eigenschaften, die gelten sollen. P umfasst `assert` Anweisungen sowie implizite Eigenschaften, die in jedem Programm gelten sollen. Ein Beispiel für letztere ist die Forderung, dass niemals durch Null dividiert werden soll.

Im Folgenden sei $p(e)$ die Funktion, die Variablen in e versioniert. Beispielsweise ist $p(\{\texttt{x = x + 1;}\}) = \{\texttt{x1 = x0 + 1;}\}$

Die Anfrage, die schließlich im Rahmen der Verifikation beantwortet werden soll, ist $C \implies P$ – beziehungsweise die Frage, ob $C \wedge \neg P$ unerfüllbar ist.

Die genaue Definition der Funktionen ist rekursiv:

`skip` Leere Anweisungen: $C(\texttt{skip}, g) := true$ und $P(\texttt{skip}, g) := true$

`if` Bedingungen: Sei p eine `if` Anweisung mit Bedingung c und den Blöcken I_c beziehungsweise $I_{\neg c}$. Dann ist

$$C(p, g) := C(I_c, g \wedge p(c)) \wedge C(I_{\neg C}, g \wedge \neg p(c))$$

$$P(p, g) := P(I_c, g \wedge p(c)) \wedge P(I_{\neg C}, g \wedge \neg p(c))$$

“`;`” Sequentielle Komposition: Besteht ein Programm p aus der Hintereinanderausführung von I und I' dann ist

$$C(p, g) := C(I, g) \wedge C(I', g)$$

$$P(p, g) := P(I, g) \wedge P(I', g)$$

Beide rekursiven Aufrufe haben die gleiche Vorbedingung g da sie unter den gleichen Bedingungen ausgeführt werden.

`assert` Beweisverpflichtungen: Sei p ein `assert` Befehl mit Argument a, dann wird a umbenannt und als Eigenschaft betrachtet, die nach der Ausführung des Befehls gilt:

$$C(p, g) := \top$$

$$P(p, g) := g \Rightarrow p(a)$$

`x = y` Zuweisungen: Die Zuweisung kann dank der vorherigen Umwandlung in SSA Form als direkte Gleichheit der linken und rechten Seite kodiert werden. Aus dem Zuweisungszeichen `=` wird also Gleichheit `==`. Falls x und y keine Arrays oder `structs` sind, ist C wie folgend definiert:

$$C(p, g) := (x_\alpha = (g?p(y) : x_{\alpha-1}))$$

α kodiert hierbei die neue Versionsnummer im Sinne der SSA Versionierung. ? ist der ternäre Operator aus der Sprache C. $p(y)$ ist die Umbenennung des rechten Teils der Zuweisung. g ist die notwendige und hinreichende Bedingung, unter der die Zuweisung ausgeführt wird (Guard).

Betrachten wir nun den Fall, dass x ein Array ist. Sei a der Index so dass das Programm die Form `x[a] = e` hat. Da a einen zur Analysezeit unbekannten Wert haben kann, wird der Effekt der Zuweisung durch eine funktionale Kodierung definiert. Für jede Zelle in dem Array x gilt, dass ihr Wert unverändert (gleich der vorherigen Version) ist, falls a nicht auf diese Position verweist, andernfalls wird der durch y beschriebene Wert zugewiesen

$$C(p, g) := x_\alpha = \lambda i : ((g \wedge i = p(a))?(p(e)) : (x_{a-1}[i]))$$

Zu beachten ist, dass λ ausschließlich durch mögliche Werte für gültige Indextypen instantiiert werden kann (`int, short,...`). Zusätzlich zur obigen Überprüfung kann eine Bedingung hinzugefügt werden, dass a größer oder gleich 0 ist und kleiner ist als die Anzahl der Elemente in x.

Strukturen (`structs`) werden analog zu Arrays kodiert. Einzig die Adressierung der Felder verwendet keine Zahlentypen sondern eine explizite endliche Abbildung von Feldernamen auf Speicherzellen.

Die Funktionen C und P können rekursiv berechnet werden. Initial werden die Funktionen mit einem dem ganzen Programmen und dem Guard $TRUE$ aufgerufen. Zum Beweis, dass das Programm die gewünschten Eigenschaften besitzt, muss die Formel $C \Rightarrow P$ als tautologisch bewiesen werden. Dies impliziert, dass für alle möglichen Eingabewerte und alle möglichen Programmabläufe (im Sinne von Pfaden) die Eigenschaften erfüllt sind.

Das folgende Programm gibt ein kurzes Beispiel [CKY03] für die Kodierung eines kleinen Programms:

Programm	p **angewendet**	C **und** P
`x = x + y;`	`x1 = x0 + y0;`	$C := x_1 = x_0 + y_0 \wedge$
`if (x != 1) {`	`if (x1 != 1) {`	$x_2 = ((x_1 \neq 1)?2 : x_1) \wedge$
`  x = 2;`	`  x2 = 2;`	$x_3 = ((x_1 \neq 1 \wedge z_0)?x_2 + 1 : x_2)$
`  if (z)`	`  if (z0)`	$P := x_3 \leq 3$
`    x++;`	`    x3 = x2+1;`	
`}`	`}`	
`assert(x<= 3);`	`assert(x3<= 3);`	

Im Folgenden muss noch die Übersetzung von Ausdrücken mit Zeigern behandelt werden.

Umwandlung von Zeigern Clarke et al. schlagen in [CKL04] die Modellierung von Zeigern durch ein stark strukturiertes Speichermodell vor. Jedes Objekt, so beispielsweise ein Array, eine Struktur `struct` oder auch nur eine Variable von einfachem Datentyp wie `int`, besitzen ein Speicherfeld dessen Größe durch den Typ eindeutig bestimmt ist. Ein Array, bestehend aus 10 `int` Variablen, hat auf einer 64-Bit Architektur eine Größe von 640 Bit. Ein Zeiger, der auf dieses Array zeigt kann vom Anfang des Arrays aus bis zu 9 mal inkrementiert werden bevor die Grenze des gültigen Speicherbereichs verlassen wird.

In CBMC wird diese Eigenschaft zur Abstraktion des Speichers verwendet. Der Speicher besteht aus n nicht überlappenden Regionen, die einem Objekt (das heißt einer im Speicher liegenden Variablen) entsprechen. Zeiger können *nur* innerhalb dieses Objektes bewegt werden. Es ist also nicht möglich einen Zeiger auf eine Struktur zu setzen und durch Inkrementierung des Zeigers in eine *zufällig* benachbarte Speicherregion zu schreiben. Die Iterierung durch große Speicherfelder (Arrays) ist jedoch möglich, da hier immer das Array selbst als Bezugsrahmen gegeben ist.

Abbildung 2.4 zeigt ein Beispiel, in dem die abstrakte Modellierung zu einer ungewollten Fehlermeldung führt. Ein häufiges Beispiel, in dem die Abstraktion nicht ausreicht, ist das byteweise Kopieren oder Füllen von größeren Datentypen. In C ist dies in den Operationen `strcpy` oder auch `memcpy` möglich. Intern wird ein `char*` verwendet um alle Bytes z.B. einer `struct` zu kopieren. CBMC meldet dieses Verhalten als Fehler, da ein `char *` inkompatibel zu dem `struct` Objekt ist, auf das der Zeiger zeigt.

Der grundlegende Algorithmus zur Behandlung von Referenzen und Derefe-

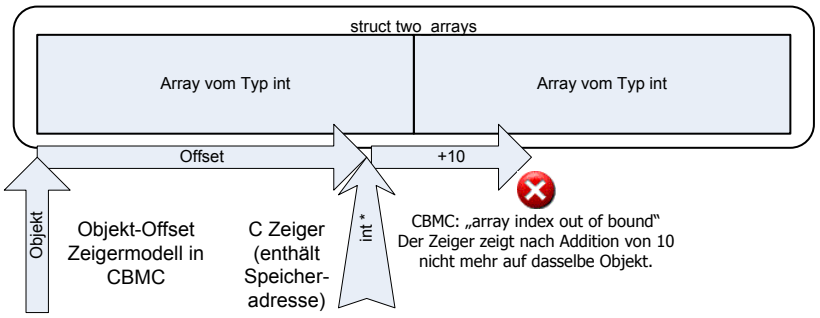


Abbildung 2.4: Das Speichermodell von CBMC: Zeiger werden in eine Objektreferenz und einen Offset geteilt. Wird durch eine Zeigeroperation der Bereich eines Objektes verlassen, meldet CBMC einen Fehler. In manchen praktischen Anwendungen ist jedoch sichergestellt, dass selbst nach Verlassen von Objektgrenzen ein Zeiger ein definiertes Ziel hat. Im Beispiel der Linuxverifikation (Kapitel 4) führte diese Einschränkung zu einigen Falschmeldungen.

renzen beruht auf der Tatsache, dass für jede Verwendung eines Zeigers in einem SSA-Programm die vorherige Definition des Zeigerziels bekannt ist. Im einfachsten Falle kann man einen Ausdruck $*p$ ersetzen durch das Objekt auf das der Zeiger p zeigt. Eine ausführlichere Behandlung der Zeigerproblematik findet sich in [CKY03].

In Fällen mit verschachtelten Operationen und Zeiger-Arithmetik kann wie oben zumindest das Speicherfeld bestimmt werden, auf das ein Zeiger zeigt. Nachfolgend zeigen wir, wie Clarke et al. die genaue Adresse, respektive das Ziel des Zeigers, berechnen.

Eine rekursive Funktion $\phi(e, g, o)$ bildet einen Zeigerausdruck e in einen Ausdruck mit *Guard* g und einen *Offset* o ab. Die Berechnung von ϕ entspricht der Bestimmung der genauen Speicherstelle, die durch e referenziert wird. o bezieht sich hierbei immer auf die genaue Adresse in dem Speicherfeld.

Es folgt ein Beispiel, um den Vorgang zu motivieren:

```
int *ip;
int a[10];
 ip = a[5];
*ip = 4;
```

Der Ausdruck `*ip` soll entfernt werden. Hierzu wird die SSA Definition ip_1 verwendet, in der `ip` auf das Array `a` mit einem Offset fünf zeigt. Da in jedem Fall Definition und Zugriff stattfinden, ist der Guard wahr.

In Fällen, wo der Dereferenzoperator $*$ oder ein einfacher Indexoperator verwendet wird, ist die Semantik wie folgend:

$$\begin{aligned} *e &\rightarrow \phi(e, g, 0) \\ e[o] &\rightarrow \phi(e, g, o) \end{aligned}$$

Der Typ von e wird als T bezeichnet. ϕ wird nun über eine Fallunterscheidung über e definiert.

Im Beispiel ist also $e = \mathtt{a}, g = \top, o = 5$.

1. Ist e ein Zeiger, so ist er im bisherigen Programmfluss definiert worden. Wurde noch kein Ziel angegeben, gilt Fall 8. e' bezeichne die vorhergehende Definition von e. Dann ist
$$\phi(e, g, o) := \phi(e', g, o)$$
2. Ist e ein Array dann ist der Ausdruck äquivalent zu $e = \&a[0]$ wobei a das Array bezeichnet, auf das e verweist (siehe Fall 4).
3. Sei e ein Ausdruck, dessen oberstes Element ein Adressoperator von einem Symbol s ist, das heißt $e = \&s$. Dann ist
$$\phi(\&s, g, 0) := s$$
4. Für Ausdrücke derart, dass e die Adresse von einem Array Element enthält, wird der Index des Array-Elementes zum momentanen Offset addiert
$$\phi(\&a[i], g, o) := a[i + o]$$
5. Ist e eine Anwendung des ternären Konditionaloperators ?, so werden beide Fälle rekursiv behandelt
$$\phi(c?e' : e'', g, o) := c?\phi(e', g \wedge c, o) : \phi(e'', g \wedge \neg c, o)$$
6. Ist e ein Ausdruck mit Zeiger-Arithmetik, so wird ϕ rekursiv auf dem Zeiger angewendet und der zusätzliche Offset addiert
$$\phi(e' + i, g, o) := \phi(e', g, o + i)$$
7. Typumwandlungen durch *casts* sind transparent
$$\phi(Q*)e', g, o := \phi(e', g, o)$$
8. In anderen Fällen ist das Verhalten undefiniert. Mittels einer neuen `assert` Anweisung wird geprüft, dass dieser Fall nicht eintritt.

Umwandlung in ein aussagenlogisches Erfüllbarkeitsproblem In den vorangegangenen Schritten wurde ein C Programm in eine Logik mit Bitvektoren, Gleichheit, Ungleichheit und arithmetischen Operationen umgewandelt. Es existieren bereits Entscheidungsverfahren für eine Logik dieser Bauart, jedoch wandelt das Werkzeug CBMC das obig erzeugte Gleichungssystem direkt in eine aussagenlogische Formel um [CKL04].

Wie in der Berechnung erreichbarer Zustände im Bounded Model Checking wird auch obige Gleichung durch die Beschreibung von Übergängen zwischen aktuellen und zukünftigen Variablenversionen kodiert (vergleiche mit den Versionen einer Variable in einer SSA-Form). Jede Variable repräsentiert einen Bitvektor, dessen Breite vom Typ der Variable abhängt. Durch Verwendung von Booleschen Kodierungen von kombinatorischen Schaltnetzen kann man die genaue Semantik aller in C verwendeten arithmetischen Operationen genau abbilden.

Als Beispiel sei eine Operation auf `int` Variablen aufgeführt. Für den Ausdruck `x == y`, wobei `x` und `y` Integervariablen mit jeweils 8 Bit sind, ist die Boolesche Gleichung gegeben durch

$$e_{=} \leftrightarrow \left(\bigwedge_{i=0}^{7} (x_i \leftrightarrow y_i) \right)$$

Ungleichheit, `x != y`, lässt sich kodieren:

$$e_{\neq} \leftrightarrow \left(\bigvee_{i=0}^{7} (x_i \oplus y_i) \right)$$

$\oplus$ steht für das exklusive Oder (XOR). Will man nun einen zusammengesetzten Ausdruck `(x == y) && (x != y)` kodieren, kann man direkt die Variablen $e_{=}$ und $e_{\neq}$ referenzieren:

$$e \leftrightarrow e_{=} \wedge e_{\neq}$$

Abbildung 2.5 zeigt auf, wie der obige Ausdruck in einer Programmanalyse als Baum kodiert vorliegt. Die letzte Formel ist natürlich unerfüllbar. Das Vorgehen zur Kodierung ganzer Formeln ist somit evident: Starte in den Blättern des Baumes, der die Ausdrücke kodiert: Führe für jede Variable und jede Version neue Boolesche Variablen ein. Führe für jeden Teilausdruck, der eine Operation kodiert, neue Hilfsvariablen ein, die das Ergebnis der Operation kodieren. Verbinde alle Variablen über eine Boolesche Kodierung der Operationen.

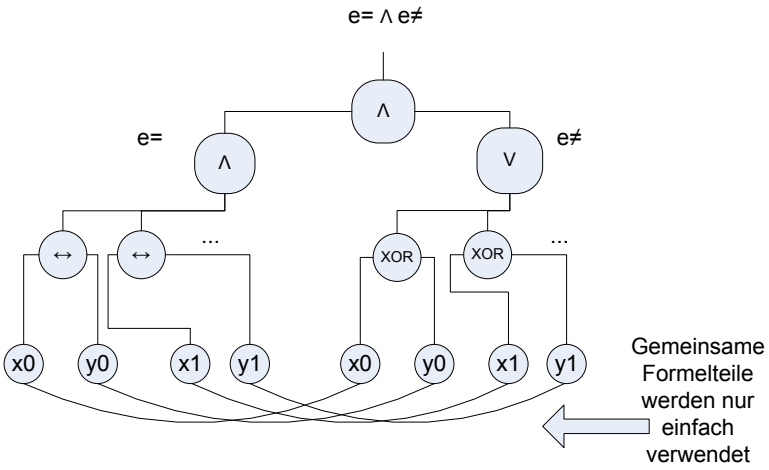


Abbildung 2.5: Beispiel für einen kodierten C Ausdruck.

2.2.9 Aussagenlogische Erfüllbarkeitsprüfung (SAT)

Software Bounded Model Checking, wie in CBMC [CKL04], wandelt das Model Checking Problem in ein *aussagenlogisches Erfüllbarkeitsproblem* (SAT) um:

Definition 16 *Die Frage, ob für eine aussagenlogische Formel ϕ eine Belegung der in ϕ vorkommenden Variablen existiert, so dass ϕ zu wahr auswertet, bezeichnen wir als* aussagenlogisches Erfüllbarkeitsproblem *(SAT).*

Der beste bekannte Algorithmus für das obige Problem benötigt eine Laufzeit, die schlechtestenfalls exponentiell in der Anzahl der Variablen wächst. In praktischer Anwendung ist die Laufzeit jedoch überraschend gering. Dieser Effekt ist auch in den Fallstudien in Kapitel 3 bis 6 zu beobachten. In fast allen modernen Werkzeugen, beispielsweise Minisat [ES03], wird zur Lösung des SAT-Problems eine Variante des Davis-Putnam-Logemann-Loveland Algorithmus (DPLL) verwendet.

Aussagenlogische Formeln werden meist in der *Konjunktiven Normalform* (CNF) kodiert. Hierbei sind nur Formeln, die eine nicht verschachtelte Konjunktion von Disjunktionen umfassen, zulässig. Negationen dürfen nur direkt vor Variablen vorkommen. Die folgende Formel ϕ ist ein Beispiel für eine CNF-Formel:

$$\phi := (a \vee b \vee c) \wedge (c \vee \neg a)$$

a, b und c sind die *Variablen*, die in ϕ vorkommen. Eine Belegung, die ϕ erfüllt, ist folgende:

$$a = \top, b = \bot, c = \top$$

In einer CNF-Formel heißen die Sequenzen von Disjunktionen, wie $(a \vee b)$, zwischen den Konjunktionen *Klauseln*. In den folgenden Kapiteln werden Formelgrößen in der Anzahl von Variablen und Klauseln angegeben.

Aus einer konkreten Belegung einer Formel kann CBMC einen Programmablauf rekonstruieren, in dem die Verletzung einer Eigenschaft stattfindet.

2.3 Zusammenfassung des Kapitels

Nach der theoretischen Einführung in die Methode des Software Bounded Model Checking (SBMC) folgt nun der Hauptteil der Arbeit. In der Literatur war die Anwendung von SBMC stets nur auf kleinen Beispielen erfolgt, die hauptsächlich aus einem theoretischen Rahmen stammen oder bestimmte aus theoretischer Sicht interessante Fälle behandeln. Wir wollen nun die Anwendbarkeit auf industrielle Fälle wagen und beginnen mit dem einfachsten Fall, zwei Implementierungen eines Algorithmus auf Äquivalenz zu prüfen. Einfach ist der Fall aufgrund der Tatsache, dass der Durchmesser des Softwaresystems, also die maximale Anzahl von Schleifendurchläufen, bereits in der theoretischen Beschreibung grob festgelegt ist.

Ebenso ist die Zahl der Speicherzellen, die während der Programmausführung beschrieben werden können endlich und leicht durch Heuristiken zu erkennen. Trotz dieser Erleichterungen stellt die folgende Fallstudie dennoch einen Probierstein der Methode dar. Dies ist damit zu begründen, dass der Algorithmus aus der Kryptologie eine Vielzahl von komplexen Bitoperationen vornimmt. Die genaue Abbildung derselben ist zwingend, da sonst keine Äquivalenz zweier Implementierungen nachweisbar wäre.

Software Bounded Model Checking basiert auf dem Lösen des aussagenlogischen Erfüllbarkeitsproblems. Dieses benötigt im schlechtesten Fall exponentiell lange Laufzeit in der Anzahl von Variablen. Da bei dem Erzeugen der Software Bounded Model Checking Gleichung jedes Bit des Programmzustands repräsentiert wird, besteht die Gefahr, dass der Einsatz bei einer realen Applikation an der *State Space Explosion* scheitert.

AES-Fallstudie 3

Nach der Einführung unterschiedlicher Methoden zur Analyse eines Software-Systems, wird nun inkrementell der Einsatz von Software Bounded Model Checking vorgenommen. Anhand vorgefundener Probleme wird die Methode fortlaufend erweitert und verfeinert.

Zu Beginn steht eine Fallstudie, in der SBMC auf ein Programm mit hoher kombinatorischer Komplexität angewendet wird.

AES [DR98] ist ein Standard zur Ver- und Entschlüsselung von Daten, der 1996 im Rahmen einer öffentlichen Ausschreibung aus dem Rijndael Algorithmus hervorging. Der Kern dieser Chiffre besteht aus zwei Funktionen $\mathcal{V}(T, K)$ und $\mathcal{E}(T, K)$, die aus einem Datenstrom T und einem geheimen Schlüssel K einen verschlüsselten Text $\mathcal{V}$ berechnen, beziehungsweise diesen wieder zu entschlüsseln ($\mathcal{E}$).

AES ist eine Block-Chiffre, das heißt der Basisalgorithmus verarbeitet jeweils Textblöcke einer bestimmten Größe unabhängig voneinander. Der Textstrom T wird hierbei unterteilt und sequentiell verarbeitet. Die definierten Größen der Blöcke sind $128, 192$ oder 256 Bit. Das gleiche Größenschema findet Verwendung bei der Schlüsselgröße, die je nach gewünschtem Sicherheitsgrad gewählt werden kann.

Als weiteres Charakteristikum von AES ist zu erwähnen, dass Verschlüsselung und Entschlüsselung symmetrisch ablaufen. So wird beispielsweise der gleiche Schlüssel als Eingabe für $\mathcal{V}$ und $\mathcal{E}$ verwendet. AES hat eine rundenbasierte Struktur. Jede einzelne Runde enthält 4 Phasen. Der Aufbau der Entschlüsselungsschritte ist ebenso in umgekehrter Reihenfolge vorgenommen worden. In Abbildung 3.1 wird ein graphischer Überblick über den üblichen Ablauf verschiedener Phasen gegeben.

Aufgrund des Algorithmendesigns sind Implementierungen des AES relativ kompakt in dem Sinne, dass die Größe der Implementierung wenig Codezeilen umfasst. Vorliegende Programme umfassen weniger als 1000 Zeilen C-Code. Des Weiteren ist zu erwähnen, dass der Kontrollfluss innerhalb des Programms sehr wenig Variationen unterworfen ist. Es gibt also wenig mögliche Pfade in dem Programm, wenn man von den Eingabedaten abstrahiert.

Dies ist sogar durch das Design begründet, da eine Abhängigkeit der Laufzeit des Algorithmus von den Eingaben zu so genannten *Timing Attacks*

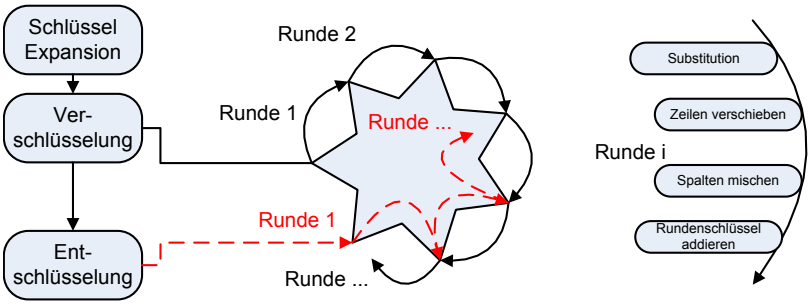


Abbildung 3.1: AES ist eine rundenbasierte Chiffre. Verschlüsselung und Entschlüsselung werden je nach Parameter 10 bis 14 mal ausgeführt. Hierbei besteht jede Runde, bis auf die letzte und erste, aus genau 4 Phasen.

führen kann. Hierbei wird aufgrund der Laufzeit einer Chiffre Information über mögliche Eingaben extrapoliert.

Der Code implementiert eine reine Berechnungsvorschrift. Dies spiegelt sich im Code durch den hohen Anteil arithmetischer Operationen wider. Zu nennen sind insbesondere XOR- und Shift-Operationen welche durch ihre Nichtlinearität eine Herausforderung für viele Verifikationsverfahren darstellen beziehungsweise erst gar nicht unterstützt werden.

Normalerweise wird beim Software Bounded Model Checking die notwendige Schranke, also die Anzahl von Schleifenausführungen im Code, geraten und gegebenenfalls erhöht. Im Werkzeug CBMC [CKL04] sind Heuristiken implementiert, die vor der Übersetzung in Aussagenlogik notwendige Schranken für häufige Fälle richtig erkennen:

```
for (int i=0; i!=10; i++) {...}
```

Für dieses Beispiel erkennt CBMC automatisch, dass die Schleife zehn mal ausgeführt wird und setzt die Schranke (Entfaltung) speziell für diese Schleife auf zehn. Für beide Implementierungen erkennt CBMC automatisch *alle* notwendigen individuellen Schranken.

In AES hängt die Zahl der meisten Schleifenausführungen nur von der gewählten Schlüssel- und Textgröße ab: Je größer der Schlüssel oder der Klartext, desto höher die Anzahl der Runden. Die beiden Parameter bestimmen eindeutig die Anzahl von Schleifenausführungen.

Der Eingabeschüssel wird verteilt auf *Rundenschlüssel*, so dass in jeder Runde ein Teil der Gesamtinformation verwendet wird. Die Verteilung der Information ist eine eigene Phase, genannt *Schlüsselexpansion* (siehe Abbildung 3.1). Für die Berechnung der Rundenschlüssel jeweils für Ver- und Entschlüsselung finden sich ebenso Schleifen im Programm.

Laufzeitfehler stellen ein wenig interessantes Untersuchungsziel bei den vorliegenden Implementierungen von AES dar. Es gibt keine Divisionen und sehr wenige Zeiger mit definierten und typsicheren Zielen. Da immer alle Zellen der verwendeten Datenstrukturen adressiert werden und sehr wenig Varianz im Kontrollfluss möglich ist, ist es unwahrscheinlich, dass man durch konventionelles Testen mögliche Zugriffsverletzungen nicht gefunden hat.

Funktionale Verifikation in dem Sinne, dass der implementierte Algorithmus den Standard erfüllt, ist nicht ohne weiteres zu erreichen. Hierzu müssten beide Ebenen aufeinander abgebildet werden und das resultierende Verifikationsproblem enthielte nicht-triviale mathematische Theoreme, die sich zumindest der Methode des Software Bounded Model Checking entzögen.

Eine aus unserer Sicht interessante Fragestellung ist mit Bezug auf optimierte Implementierungen gegeben: *Sind Reimplementierungen äquivalent zur Referenzimplementierung?* Mit Äquivalenz ist gemeint, ob unterschiedliche Implementierungen bei gleichen Eingaben auch gleiche Ausgaben berechnen (*funktionale Äquivalenz*).

Die Gesamtarbeit evaluiert die Einsatzfähigkeit von Software Bounded Model Checking. In diesem Kapitel lautet die Frage: *Kann Software Bounded Model Checking trotz State space explosion die Äquivalenz von berechnungsintensiven Programmen berechnen?*

Die Beantwortung der ersten Frage impliziert hierbei eine positive Antwort der zweiten. Die Erwartung ist jedoch, dass SBMC nicht in der Lage sein wird, die Komplexität der Funktionen zu bewältigen. Selbst bei der einfachsten Konfiguration mit Block- und Schlüsselgröße von 128 Bit entsteht eine Zahl möglicher Eingaben von 2^{256} - die Existenz vieler nicht trivialer Zwischenwerte mit einer komplexen Verflechtung ist hierbei noch unberücksichtigt. Für die Prüfung auf Äquivalenz müssen zwei Implementierungen gleichzeitig in einer Gleichung kodiert werden. Hierdurch verdoppelt sich die Anzahl der Zustandsbits. Ebenso unberücksichtigt ist das Faktum, dass keine direkte Eingabe-Ausgabe Logik vorliegt (wie in einem kombinatorischen Schaltkreis), sondern dass Ergebnisse schrittweise über temporäre Zustände (SSA-Versionen) berechnet werden. Die tatsächliche Anzahl von Variablen ist erwartungsgemäß also sehr viel größer.

Im nächsten Abschnitt wird das Vorgehen zur Verifikation erläutert. Abschnitt 3.3 enthält die produzierten Ergebnisse.

3.1 Vorgehen

Die Äquivalenzprüfung ist eine weit verbreitete Aufgabe im Bereich der Hardware Verifikation. Im Beispiel findet die funktionale Äquivalenzprüfung An-

```
function int A(int input) { ... }
function int B(int input) { ... }

void wrapper() { // 'miter' Funktion
      int input_A  = nondet_int(); int input_B  = nondet_int();
      // Eingaben sind gleich
Pre:  assume(input_A == input_B);
      // Sequentielle Ausführung
      int result_A  = A (input_A); int result_B  = B (input_B);
      // Sind Ausgaben gleich?
Post: assert(result_A == result_B);
}
```

Abbildung 3.2: Die Überprüfung funktionaler Gleichheit ist mit dem Werkzeug CBMC leicht zu erreichen, sobald die Funktionen so transformiert sind, dass sie sich nicht gegenseitig beeinflussen können. Allerdings entspricht der Zustandsraum in solch einem Programm dem Zustandsraum eines Produktautomaten.

wendung. Mit Ausnahme der expliziten Parameter der C Funktionen, die den AES implementieren, gibt es keine relevanten Größen, die auf Gleichheit geprüft werden. Insbesondere ist keine Überprüfung eines Zeit- beziehungsweise Taktbegriffes verlangt. Ebenso sind die Werte von Zwischenzuständen irrelevant.

In Abbildung 3.2 wird ein kurzes Beispiel gegeben, wie eine Äquivalenzbedingung für das Werkzeug CBMC kodiert werden kann: Zwei Funktionen `A` und `B` aus zwei unterschiedlichen Implementierungen erhalten Eingaben `input_A` beziehungsweise `input_B`. CBMC interpretiert den `assume` Befehl als gültige Vorbedingung für den Beweis. Beide Funktionen werden nacheinander aufgerufen.

Hierzu muss sichergestellt werden, dass sie keine Seiteneffekte enthalten, die die gegenseitige Ausführung beeinflussen. Dies kann beispielsweise durch die Einrichtung zweier disjunkter Namensräume erreicht werden. Nach Ausführung werden alle (relevanten) Ausgabeparameter mittels eines `assert` Befehls auf Gleichheit geprüft. Die Funktion, die die Äquivalenzbedingung kodiert, wird im Hardwarebereich als so genannte *Miter-Schaltung* bezeichnet.

Zu zeigen ist, dass beide Funktionen dann gleiche Ausgaben berechnen, falls sie gleiche Eingaben bekommen. Diese Bedingung kann durch `assert / assume` Befehle im Quellcode ausgedrückt werden (siehe Abschnitt 2.2.8). Im Folgenden werden die behandelten Implementierungen unserer Fallstudie kurz vorgestellt.

3.1.1 Die Referenz Implementierung (RI)

Barreto und Rijmen erstellten eine Referenzimplementierung, die man öffentlich herunterladen kann[1]. Die folgenden Funktionen umfassen die Schnittstelle beziehungsweise implementieren die notwendigen Phasen.

- `int rijndaelKeySched (word8 k[][], int keyBits, int blockBits, word8 rk[][][])`
- `int rijndaelEncrypt (word8 a[][], int keyBits, int blockBits, word8 rk[][][])`
- `int rijndaelDecrypt (word8 a[][], int keyBits, int blockBits, word8 rk[][][])`

Eine Zuordnung der Phasen zu den Funktionen ist in Tabelle 3.1 gegeben. `k` ist der Parameter der auf den Eingabeschlüssel verweist, während `rk` eine expandierte Version enthält. Die expandierte Version enthält die Schlüssel, die in den einzelnen Runden der Verifikation verwendet werden (Rundenschlüssel). Die Parameter mit `Bits` Postfix bezeichnen die Anzahl der Bits für die Kodierung des Schlüssels beziehungsweise für die Größe eines Textblocks. Innerhalb eines vollständigen Verschlüsselungs-Entschlüsselungs-Zyklus, das heißt Schlüsselerzeugung, Verschlüsselung und Entschlüsselung, müssen die Größenparameter sowie der Quellschlüssel `k` übereinstimmen.

3.1.2 Mike Scotts Implementierung (MSI)

Mike Scott erstellte eine weitere Implementierung des AES Standards, die Optimierungen gegenüber der Referenzimplementierung beinhaltet[2]. Auch diese Implementierung ist in ANSI-C geschrieben. Es sind auch keine Funktionen in Assemblersprachen enthalten, weshalb diese Implementierung als Vergleichsobjekt gewählt wurde. Darüber hinaus ist MSI in ihrer Struktur sehr stark an der RI angelehnt. Dies erleichtert die Verifikationsarbeit, da keine weiteren Dokumente zur Identifizierung gleicher Subkomponenten benötigt werden.

Die Schnittstellen von MSI sind durch folgende Funktionen gegeben:

- `void gentables(void)`
- `void gkey(int nb,int nk,char *key)`
- `void encrypt(char *buff)`
- `void decrypt(char *buff)`

`gentables` erzeugt Tabellen, die vorberechnete Funktionsergebnisse enthalten (z.B. der diskrete Logarithmus). Die Funktion muss vor den anderen Funktionen ausgerufen werden. `gkey` erzeugt die Rundenschlüssel durch die

[1] www.iaik.tugraz.at/Research/krypto/AES/old/~rijmen/rijndael/rijndaelref.zip
[2] Erhältlich unter: ftp://ftp.compapp.dcu.ie/pub/crypto/rijndael.c

Tabelle 3.1: AES Phasen werden durch die folgenden Funktionen und Datenstrukturen implementiert.

AES	Referenz Implementierung	Mike Scotts Impl.
S-Box	`S`	`fbsub`
Inverse S-Box	`Si`	`rbsub`
Schlüssel Expansion	`rijndaelKeySched`	`gkey`
Verschlüsselung	`rijndaelEncrypt`	`encrypt`
Entschlüsselung	`rijndaelDecrypt`	`decrypt`

Expansion des Eingabeschlüssels, der durch `key` übergeben wird. Tabelle 3.1 fasst zusammen, welche Phasen welchen Funktionen in beiden Implementierungen entsprechen. Aufgrund der klaren Unterteilung kann nun modular bewiesen werden, dass für jede einzelne Phase die Funktionen äquivalent sind.

Beide Implementierungen sind in ANSI-C geschrieben. Beide Programme nutzen die Möglichkeit, Parameter von Funktionen per Referenz (das heißt über eine Adresse) zu übergeben. Während des Einsatzes von CBMC traten Fehler bei der Behandlung dieser Referenzparameter auf. Aus diesem Grund wurden Referenzparameter vor den folgenden Experimenten in globale Variablen umgewandelt. Die Umwandlung ist technisch effizient und nicht fehleranfällig, da oft die gleichen Objekte als Parameter der Funktionen übergeben werden.

3.2 Synchronisierung der Eingaben

RI und MSI benutzen ähnliche Tabellen und Matrizen mit vorberechneten Ergebnissen und Konstanten. Ein Beispiel hierfür die so genannte S-Box, die vorgibt, welche nicht-linearen Ersetzungen vorgenommen werden. Die S-Box ist durch den Standard eindeutig definiert und folglich in beiden Implementierungen als konstantes Array implementiert. Die sequentielle Ausführung beider Implementierungen kann also dadurch vereinfacht werden, dass die konstanten Tabellen nur einmalig verwendet werden. In dem Teil des Quellcodes, der zu MSI gehört, ersetzen wir die Referenzen durch Referenzen auf Tabellen der RI. Im Programm kann dies effizient und wenig fehleranfällig durch den Einsatz von C Präprozessor Makros vorgenommen werden, z.B.: `#define fbsub S`. Als Folge dieser Vereinfachung kann die Funktion `gentables` aus dem Programmfluss von MSI entfernt werden.

Verschlüsselung, Entschlüsselung und Schlüsselgenerierung basieren auf den Parametern Schlüssel und einem Textpuffer. In Abbildung 3.4 a) wird das `assert` kodiert, dass beide Implementierungen auf gleichen Angaben operieren. Die Kodierung dieser Annahme setzt voraus, dass der Verifizierer

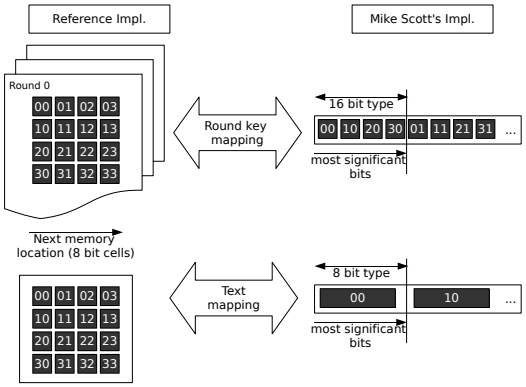


Abbildung 3.3: RI und MSI nutzen eine unterschiedliche Strukturierung des Speichers um den Schlüssel und die Texte zu kodieren. Die Information, wie genau die einzelnen Eingabebits beider Implementierungen korrespondieren ist eine Information, die vor der eigentlichen Verifikation aus dem Quellcode extrahiert werden muss. Ist die Abbildung definiert, kann mittels `assume` Befehlen ausgedrückt werden, dass beide Implementierungen beim Äquivalenztest auf gleichen Eingaben laufen (siehe Abschnitt 3.2).

weiß, wie sich die Kodierung der Eingaben genau unterscheidet. Eine reine Gleichheit zwischen Bitblöcken führt nicht zum Ziel. Statt dessen muss eine Abbildung der Eingaben definiert werden. Diese ist in Abbildung 3.3 graphisch verdeutlicht.

3.3 Ergebnis

Die zu prüfende Eigenschaft ist, ob die generierten Ausgaben der Ver- und Entschlüsselungsfunktionen gleich sind. Dekomposition kann erreicht werden, indem gezeigt wird, dass gleiche Schlüsseleingaben gleiche Rundenschlüssel erzeugen. Nachfolgend kann dann für die Behandlung der Hauptphasen von der Schlüsselgenerierung abstrahiert werden. Nun folgt die Beschreibung wie die Implementierung der Rundenschlüsselgenerierung äquivalent bewiesen werden.

3.3.1 Schlüsselgenerierung

Die Prüfung auf Äquivalenz benötigt eine Abbildung zwischen den Bits der unterschiedlich kodierten Eingabeformate. Diese wird in den Abbildungen 3.3 und 3.4 vorgestellt. Nach deren Bereitstellung stellt sich die Verifikation dieses Teils des Algorithmus als einfach heraus: Die einzige Eingabe ist der 128 Bit Schlüssel, welcher mittels der von CBMC bereitgestellten Funktion

a)

```
for (i = 0; i != 4; i++) {
  for (j = 0; j != 4; j++) {
    key[i][j] = nondet_char();
// initialisiere den RI Text
    text[i][j] = nondet_char();
// initialisiere den MSI Text
    text_ms[j*4+i] = text[i][j];
  }
}
```

b)

```
BYTE tmp[4]; WORD res;
for (r = 0; r != 11; r++) {
  for (j = 0; j != 4; j++) {
     for (i = 0; i != 4; i++) {
        tmp[i] = rk[r][i][j];
     }
     res = pack(tmp);
// initialisiere die
//   MSI Rundenschlüssel
// (Verschlüsselung)
(*)  fkey[r*4+j]=res;
  }
}
```

Abbildung 3.4: a) Die Eingaben, Text und Schlüssel, müssen vor der sequentiellen Ausführung synchronisiert werden. b) Bei Ent- und Verschlüsselung müssen zusätzlich die Rundenschlüssel synchronisiert werden. Die in den Beispielen verwendete Funktion `pack` kodiert 4 Byte in ein 32 Bit `WORD`. `rk` (RI) und `fkey` (MSI) sind die für die Rundenschlüssel verwendeten Symbole.

`nondet_int()` nicht-deterministisch gewählt wird.

Die Eigenschaft, die wir in diesem Teil der Verifikation beweisen wollen ist die Gleichheit der erzeugten Rundenschlüssel (in Bezug auf die gefundene Abbildung). Die Bedingung kann mit CBMC über `assume` Befehle ausgedrückt werden wie Abbildung 3.4 b) aufzeigt: Als kleine Änderung muss jedoch die mit `(*)` markierte Zeile geändert werden. Statt einer synchronisierenden Zuweisung wird nachfolgend eine Beweisverpflichtung ausgedrückt:

```
(*)     assert(fkey[r*4+j]==res);
```

Wie aus Tabelle 3.2 hervorgeht (Zeile KEY), war der Äquivalenzbeweis der Schlüsselgenerierung erfolgreich. Im Folgenden können damit die Rundenschlüssel einmalig von der RI generiert werden. Sie werden dann in die Eingabearrays von MSI kopiert. Es wird sich zeigen, dass diese Vereinfachung notwendig ist um die restlichen Teilprobleme lösen zu können.

Im Folgenden werden alle Experimente zur Verschlüsselung mit EN und alle Experimente zur Entschlüsselung mit DE bezeichnet. Die nachfolgende Zahl gibt die Anzahl von ausgeführten Runden an. Zusätzlich gibt die Folge 'zz' an, dass ein Experiment mit Null gefüllten Schlüssel- und Textbits ausgeführt wurde. 'loop'-Experimente betreffen den Nachweis, dass eine beliebige Runde mit beliebigen Text- und Schlüsseleingaben gleiche Ergebnisse produziert.

Tabelle 3.2: Gemessene Größen aller Verifikationsläufe. Zur Entscheidung des SAT-Problems wurde Minisat2 [ES03] benutzt. Die Zeit in Spalte CBMC wird nur für die Kodierung des Problems benötigt. (S) kennzeichnet eine erfüllbare Instanz (das heißt die Implementierungen waren nicht gleich oder die Dekomposition war schlecht gewählt). (TO) kennzeichnet einen Timeout (mehr als zwölf Stunden). Vereinfachte Instanzen in denen Text und Schlüssel auf mit null initialisierte Arrays gesetzt wurden sind mit "zz" abgesetzt.

	CBMC [s]	SAT [s]	Zuweisungen	Entfernt	Variablen	Klauseln
KEY	25	76	13.958	5.641	79.145	475.049
EN1r	21	2	15.577	9.489	34.665	221.485
EN2r	37	1.313	16.471	7.472	267.541	1.573.693
EN3r	51	3.274	17.365	7.756	500.385	2.925.901
EN4r	69	(TO)	18.259	8.040	733.229	4.278.109
ENloop	344	21	10.502	4.654	276.374	1.598.822
ENzz2r	37	5	16.488	7.521	253.125	1.486.261
ENzz10r	166	954	23.740	9.831	2.116.005	12.303.925
DE1r	20	1	15.645	9.515	34.658	220.870
DE2r	69	(TO)	10.659	4.035	886.030	4.821.654
DE3r	172	(TO)	20.131	8.152	1.737.370	9.422.438
DEloop(S)	1.446	2.868	26.632	8.953	4.244.263	22.604.459
DEzz2r	85	15	17.905	7.730	883.094	4.805.802
DEzz10r	1.679	(TO)	35.849	11.498	7.693.814	41.612.074

3.3.2 Verschlüsselung

Um nicht direkt auf Laufzeit- und Komplexitätsprobleme zu treffen wurde die Schwierigkeit graduell gesteigert (siehe jeweils mit Tabelle 3.2). Zunächst (EN1r) wurde die Verschlüsselung nur für eine Runde von zehn vom Standard vorgesehenen Runden ausgeführt. Dies schließt die Rundenschlüsselgenerierung zunächst mit ein. Die Ausgaben beider Algorithmen konnten als gleich für alle 2^{256} möglichen Eingaben gezeigt werden. Die Zahl der Runden wurde nachfolgend erhöht (EN2r, EN3r und EN4r).

Für die gewählte Klartext und Schlüsselgröße von 128 Bit verlangt der AES Standard eine Mindestzahl von 10 Runden. Die Erweiterung der Verifikation obiger Experimente auf 10 Runden schlug jedoch fehl, da sie in einer SAT-Instanz resultierte, die nicht binnen von 12 Stunden gelöst werden konnte: schon bei Experiment EN4r kam es zu der Überschreitung der Laufzeitgrenze (*Timeout*).

Die Lösung dieses Problems kann durch erneute Dekomposition erreicht werden: Ist jede einzelne Runde R_i äquivalent, dann ist auch die Sequenz der Runden $\langle R_0, R_1, ..., R_{10} \rangle$ äquivalent. Die genaue Umsetzung ist der einer Induktion ähnlich: Beide Implementierungen erzeugen den gleichen Startzustand (mit Bezug auf ihre unterschiedliche Kodierung) vor oder nach ei-

nigen Ausführungen der Verschlüsselung (siehe EN1r-EN3r). Beweist man nun noch den Induktionsschritt, der besagt, dass falls der Zustand beider Programme nach i Runden gleich ist, dann muss er auch nach $i+1$ Runden äquivalent sein, dann ist die Induktion vollständig und der Nachweis für zehn (oder mehr) Rundenverschlüsselungen ist vollbracht.

In den Experimenten ist auffällig, dass der relative Anstieg der Komplexität von EN1r zu EN2r viel größer als zwischen EN2r und EN3r ist. Dies ist darauf zurückzuführen, dass die erste und die letzte Runde der Ver- und Entschlüsselung weniger Phasen enthalten.

In Experiment ENzz2r wird eine zwei-Runden-Verschlüsselung mit *deterministischen* Text und Schlüsselarrays vorgenommen. Beide Arrays sind mit null gefüllt. Dies erlaubt es den Einfluss freier Variablen auf die Berechnungskomplexität zu messen. Sind Text und Schlüssel fest, dann enthält das Verifikationsproblem keine freien Variablen mehr. Die Laufzeit bestätigt die Vermutung, dass das Setzen der freien Variablen signifikante Laufzeitunterschiede bringt, obwohl der Größenunterschied des Programms und der Formel gering ausfällt. Bei zwei Runden ist die Laufzeit um den Faktor 20 geringer. Bei festen Eingaben ist sogar das zehn Rundenproblem ohne induktives Vorgehen lösbar (ENzz10r).

3.3.3 Entschlüsselung

Die Struktur des AES Algorithmus ist symmetrisch bezogen darauf, dass die Rundenschlüssel in umgekehrter Reihenfolge verwendet und somit die letzte Runde zuerst bei Entschlüsselung ausgeführt wird. Interessanterweise wird die daraus erwachsende Erwartung, dass auch die Laufzeiten für die Verifikation ähnlich sein sollten, nicht bestätigt (siehe Abbildung 3.2).

Alle Entschlüsslungsversuche (DE*) führen zu größeren Formeln und benötigen beim Lösen durch den SAT-Solver Minisat2 erheblich mehr Zeit. Die Gründe hierfür sind nicht bekannt. Wir vermuten folgende Einflüsse

- MSI benutzt eine andere Reihenfolge in der die Entschlüsselungsrunden ausgeführt werden. Folglich sind möglicherweise Teilausdrücke in der Programmformel nicht wiederverwendbar.
- MSI nutzt optimierte inverse Rundenschlüssel `rkey` die sich von denen zur Verschlüsselung unterscheiden. Bei der RI werden die gleichen Schlüssel verwendet. Zwar wird die Abbildung im Quellcode definiert, aber es ist nicht klar wie eine Modularisierung unter diesen Bedingungen vorgenommen werden kann.
- Die `rkey` Rundenschlüssel hängen von der vorherigen Generierung der

normalen Rundenschlüssel ab. Die vorherige Generierung der anderen Schlüssel führt zu einer Asymmetrie. Die zusätzlichen Schleifen und Speicherfelder können im SBMC eine exponentiellen Vergrößerung des Problems bewirken.

Experiment DE1r beweist, dass zumindest die erste Runde der Entschlüsselung äquivalent ist. DE2r und DE3r SAT-Probleme konnten generiert, aber nicht gelöst werden. Die Verwendung von konkretem Klartext und Schlüssel ermöglichte die Prüfung einer zwei-Runden Entschlüsselung.

Die Implementierungen zeigen bemerkenswerte Unterschiede in der Art und Reihenfolge der Entschlüsselungsrunden. Dies hat zur Folge, dass die Dekomposition – respektive das induktive Vorgehen – eine Instanz mit Gegenbeispiel (DEloop) hervorbrachte. Die hohen Laufzeiten sind erwartungsgemäß kleiner als die Laufzeiten, die für eine korrekte Dekomposition benötigt würden[3].

3.4 Erfahrungen

In dem letzten Abschnitt wurden die Laufzeiten und Beweisergebnisse aufgelistet. Im Folgenden diskutieren wir die Erfahrungen bezüglich des tatsächlichen Verifikationszyklus (Abschnitt 3.4.1) sowie der Aussagekraft der durch einen SAT-Solver generierten Beweise (Abschnitt 3.4.2).

3.4.1 Verifikationszyklen

Der erste Schritt um ein Programm zu verifizieren ist die genaue Vorbereitung des Programms und das Ausformulieren der Spezifikation. Dieser Schritt kann syntaktische Transformationen, semantische Aufteilung (*Slicing*), Abstraktion oder die Vorbereitung eines induktiven Vorgehens sein. Jeder einzelne dieser Schritte kann neue Fehler einführen. Gründe hierfür sind, neben der bloßen Möglichkeit Fehler zu machen, auch das häufig fehlende Verständnis von der zu verifizieren Software. Der Verifizierer muss am besten *vorher* verstehen, dass und warum eine Eigenschaft erfüllt oder falsch ist um effizient vorgehen zu können. Mühlberg und Lüttgen [ML07] weisen auch auf dieses Problem hin. Die Kodierung der Spezifikation setzt voraus, dass die in CBMC notwendigen `assert`-Befehle an den richtigen Stellen im Programm eingefügt werden. Im vorliegenden Falle war es wichtig eine Abbildung der Eingabekodierungen herzustellen.

[3] Diese Erwartung basiert auf der allgemeinen Tendenz, dass unerfüllbare SAT-Probleme im Allgemeinen eine längere Laufzeit verlangen als erfüllbare.

In der Praxis ist der Prozess der Softwareverifikation ein zyklischer Prozess in dem eine Anpassung des Programms oder der Spezifikation einer Ausführung des eigentlichen Verifikationswerkzeugs folgt. Das Ergebnis desselben muss nun genau analysiert werden. Schließlich soll eine Konvergenz erreicht werden – überlicherweise ist dies mit dem erfolgreichen Beweis, dass eine Eigenschaft gilt assoziiert. Aber auch dieser Nachweis kann hinterfragt werden da noch immer Fehler in der Programmabstraktion oder der Formulierung der Spezifikation vorliegen können. Ohne eine zu skeptische Position annehmen zu wollen sei auf das zugrunde liegende Problem verwiesen, dass nicht geklärt ist, wann ein Verifikationsprozess erfolgreich beendet oder mit Misserfolg abzubrechen ist.

Im nächsten Abschnitt gehen wir auf einen Teilaspekt dieses Problems genauer ein.

3.4.2 Beweiserklärungen

CBMC [CKL04] generiert SAT-Probleme die von einem SAT-Solver gelöst werden. Ist die Instanz erfüllbar kann CBMC einen Pfad im Programm generieren der diesem aussagenlogischen Gegenbeispiel entspricht. Dieses Gegenbeispiel kann man anhand der Spezifikation und des konkreten Programms leicht nachprüfen.

Im umgekehrten Fall wird eine Eigenschaft, das Ausbleiben eines Fehlerzustands, bewiesen. Moderne Beweiser liefern zusätzlich einen Beweis, dass eine Eigenschaft gilt. Der Beweis – meist in einer deduktiven Form angegeben – beinhaltet jedoch für die vorliegenden Problemgrößen eine zu große Anzahl von Schritten, als dass man hoffen könnte etwas über das Programm zu lernen. Es wäre wünschenswert einen abstrakteren Beweis generieren zu können, der einem erklärt *warum* eine Eigenschaft gelten soll. Ist noch ein Fehler in der Abstraktion des Programms oder der Spezifikation der Eigenschaft enthalten, entzieht dieser sich zur Zeit nur durch stichprobenartige Kontrolle der Erkennung.

Dieses Problem ist auch in der Wissenschaftstheorie bekannt. Formal kann man schreiben

$$P \wedge S \wedge T \models \phi$$

wobei P für ein Programm, S für eine Spezifikation und T für eine Transformation stehe. Ist das Ergebnis nicht wie erwartet, vermutet man einen Fehler auf der linken Seite – ob dieser in P, S oder T vorliegt ist nicht entscheidbar.

3.4.3 Verwandte Arbeiten

Matsumoto et al. [MSF05] präsentieren eine Methode zur Prüfung von C Beschreibungen. Einer der betrachteten Fälle ist eine Implementierung von AES. Die anderen Fälle variieren sehr von der hier vorgenommenen Studie: Statt Äquivalenz zweier unterschiedlicher Implementierungen zu prüfen, fügen die Autoren eine kleine Anzahl von lokalen Änderungen ein. Für AES sind die Änderungen beschränkt auf die Ersetzung von äquivalenten XOR Operationen. Die Äquivalenzprüfung basiert auf symbolischer Ausführung mit Äquivalenzklassenabstraktion. Eine Einschränkung der Methode ist, dass isomorphe Kontrollflussgraphen vorliegen müssen was bei zwei verschiedenen Implementierungen nicht angenommen werden kann.

Die Verwendung von aussagenlogischen Beweisern für kryptographische Probleme ist ein aktives Forschungsgebiet. Massacci und Marraro [MM00] analysieren den *Data Encryption Standard (DES)* durch Kodierung in Aussagenlogik. Durch diese Kodierung konnte eine Berechnung des Schlüssels für DES mit bis zu drei Runden erreicht werden. Im Gegensatz zu unserer Arbeit benötigen Massacci und Marraro eine manuelle, und damit fehleranfälligere, Kodierung des Algorithmus. Andere Arbeiten in dem Gebiet betreffen die Analyse von Hash-Funktionen [MZ06, DKV07].

Software Model Checking Fallstudien anderer Gruppen behandeln typischerweise weniger arithmetikintensive Programme. Oft ist die Einhaltung von Schnittstellenspezifikationen das Prüfungsziel wie es auch im folgenden Kapitel behandelt werden wird. Ball et al. [BBC+06] wenden das Verfahren auf Windows Gerätetreiber an und suchen nach Fehlern in Schnittstellenprotokollen mit endlichen Zustandsräumen. Die Eigenschaften basieren zumeist auf Zustandsübergängen eines Sicherheitsautomaten die beim Aufruf von Funktionen ausgelöst werden. Post und Küchlin [PK07] präsentieren eine ähnliche Fallstudie, in der Bounded Model Checking zur Prüfung von Schnittstelleneigenschaften in Linux Gerätetreibern verwendet wird. MOPS ist ein weiteres Werkzeug das angewendet wurde, um abstrakte Spezifikationen in Linux Anwendungen zu prüfen [CW02].

Die obigen Software Model Checking Fallstudien betrachten Programme, die wenig arithmetische Komplexität beinhalten, wie es typisch ist für Gerätetreiber, die meist endliche Zustandsabfolgen als Implementierungsbasis haben. Die Äquivalenzprüfung zweier Kryptoalgorithmen zeigt eine inverse Charakteristik: Die Programme sind kleiner und beinhalten einen großen Anteil komplexer arithmetischer Operationen. Der Kontrollfluss ist bereits zur Kompilezeit festgelegt. Ein weiterer Unterschied ist, dass fast alle Variablen und Bits des Programms für die Korrektheitsanalyse relevant sind – eine Abstraktion oder Reduktion des Programms auf relevante Anteile ist daher nicht effektiv.

Genau aus letzterem Grund scheitert auch die Anwendung von Abstraktionstechniken: abstrakte Interpretation [CC77] und statische Analysewerkzeuge ermöglichen es, variable Abstraktionen und abstrakte Wertebereiche zu verwenden um die Komplexität zu verringern. Eine aktuelle Fallstudie ist von den Autoren Delmas und Souyris [DS07] beschrieben. Andere Gruppen verwenden festgelegte Abstraktionen wie etwa Engler und Ashcraft [EA03]. Die Technik der Verfeinerung von Abstraktionen durch Gegenbeispiele (*CEGAR*) wird in Werkzeugen wie MAGIC [CCG+03] eingesetzt. Bryant et al. modifizieren die Technik für Bitvektor Arithmetik [BKO+07] jedoch ist gerade das Design des AES Algorithmus derart, dass alle Abstraktionstechniken zwangsläufig scheitern müssen wenn Sie nicht genau auf die Aufgabe angepasst werden. Da jedoch beide Implementierungen in unserem Fall Operationen durch verschiedene arithmetische Operationssequenzen umsetzen, ist auch diese Verbesserung nicht anwendbar.

3.5 Zusammenfassung des Kapitels

In der obigen Fallstudie konnte gezeigt werden, dass SBMC in der Lage ist eine kompakte aber komplexe Software funktional zu verifizieren. Die hohe Anzahl relevanter Zustandsbits konnte durch Induktion über die Anzahl der fast identischen Runden erreicht werden.

Neben dem Einsatz des BMC Werkzeugs CBMC wurde auch das Werkzeug SATABS [CKSY05] eingesetzt. SATABS implementiert Software Model Checking mit variabler Abstraktion von Programmzuständen. Wir vermuten, dass Abstraktion auf einem Programm, in dem es auf bitweise Äquivalenz ankommt, nicht anwendbar ist: Mit SATABS wurde kein einziges Teilresultat erzielt. Folglich erweitert Software Bounded Model Checking die Domäne der Software die praktisch verifiziert werden kann.

Implementierungen von kryptographischen Algorithmen ist eine Art systemnaher und praxisnaher Software. Jedoch handelt es sich bezogen auf die Kenngrößen der Software keineswegs um eine Standardsoftware wie sie üblicherweise vorkommt.

Systemnahe Software beschreibt in anderen Fällen, als der Implementierung von Chiffren, größere und weniger stark strukturierte Zustandsräume.

Das zahlenmäßig häufigste Beispiel für die Anwendung von statischen Analysetechniken und Software Model Checking in der Literatur sind Gerätetreiber. Diese übernehmen im Rahmen eines Betriebssystems die Steuerung einzelner Hardwarekomponenten. Im folgenden Abschnitt soll nun untersucht werden ob und wie sich SBMC erfolgreich zur Verifikation oder zum Finden von Fehlern in Gerätetreibern einsetzen lassen kann.

4 Fehlersuche in Linux Treibern

Linux ist ein Open-Source Betriebssystem welches mehrere Millionen Zeilen in C geschriebenen Quellcode umfasst. Vorherige AES Fallstudie weist signifikante Unterschiede zu Linux Treibern auf:

- Linux-Treiber sind um bis zum Faktor 100 größer.
- Seiteneffekte durch Bibliotheksaufrufe sind im gesamten Code vorhanden: es wird ein Teil der Funktionalität in Treibern durch generische Bibliotheksfunktionen implementiert.
- Die Architektur des Gesamtsystems ist stark modularisiert, jedoch sind die Interaktionen der einzelnen Subsysteme nicht gut dokumentiert.
- Funktionale Eigenschaften sind meist weniger relevant als Protokolle und korrekte Abfolgen von Schnittstelleninteraktionen.

Da die Verifikation der Gesamtapplikation Linux außerhalb der Möglichkeiten präziser Techniken wie SBMC steht, werden im Folgenden Gerätetreiber einzeln untersucht. Es stellt sich trotzdem die Frage nach der Skalierbarkeit selbst bei einzelnen Treiberdateien. Des Weiteren stellt sich die Frage ob und inwieweit die Modularität der Analyse die Genauigkeit der Verifikation beeinflusst: Funktionsaufrufe in andere Module, Subsysteme oder Bibliotheken, die vom Treiber zum Zweck eines kleineren Verifikationsmoduls abgetrennt werden, können dazu führen, dass fehlerhaftes Programmverhalten fälschlicherweise angenommen (*false positive*) oder fälschlicherweise nicht berücksichtigt wird (*false negative*).

Es gibt mehrere tausend Treiber, das heißt es muss eine Auswahl getroffen werden ob im Detail einzelne Treiber oder - mit weniger Detail - zunächst alle Treiber grob nach Fehlern untersucht werden sollen. Im letzten Fall kommt die Nebenbedingung hinzu, dass die Analyse vollständig automatisch vollzogen werden muss, da ein manueller tausendfacher Eingriff für einzelne Dateien nicht leistbar ist.

Es zeigt sich schnell, dass SBMC die notwendigen Rekurrenz-Durchmesser für reale Treiber nicht erreichen kann. Es finden sich beispielsweise Schleifen über unbeschränkte Listen:

```
// Suche ein Gerät mit 'id' == 5
while (device_iterator->next) {
     if (device_iterator->id == 5) break;
     device_iterator = device_iterator->next;
}
...
```

In diesem Fall hängt die maximale Anzahl von Ausführungen der Schleife von vorhandener Hardware ab. Es kann zur Compilezeit keine Schranke für die Ausführung der Schleife gefunden werden. Eine Schleife, die ein Element einer Liste sucht, kann aber abstrahiert werden: Die Schleife terminiert immer und findet entweder kein Element oder liefert ein Gerät von dem nur bekannt ist, dass das `id` Feld fünf ist.

Solche Schleifen alleine stellen schon ein Problem dar welches einer gesonderten Behandlung bedarf. Eine gesonderte Behandlung bedingt aber unumgänglich den Einsatz eines manuellen Reviews da zur Zeit keine automatischen Methoden bekannt sind, einen nennenswerte Anzahl von realen Schleifen abstrahieren zu können.

In den folgenden Abschnitten wird sukzessive eine Methode zur Verifikation modularer Software entwickelt und erweitert. Aufgrund der Größe des Gesamtsystems und der Forderung nach automatischer Verifikation beschränkt sich das Ziel zunächst auf die Suche nach Fehlern. Eine Erweiterung der Methode zur schlüssigen Verifikation wird schließlich in Kapitel 5 gegeben werden.

4.1 Transfer der AES Resultate auf Gerätetreiber

Die Verifikation von Gerätetreibern, die in dem spärlich dokumentierten GNU-C Dialekt geschrieben sind, ist mit einigen technischen Problemen verbunden. Im Laufe der Arbeit an Linux Quellcode wurde eine Werkzeugkette erstellt, die im Folgenden als *Avinux* bezeichnet wird. Avinux umfasst neben der automatischen Transformation realen Quellcodes in CBMC-kompatiblen ANSI-C Code auch die Konstruktion heuristischer Umgebungsmodelle, sowie eine Infrastruktur zu Spezifikation komplexer Schnittstelleneigenschaften. Zunächst soll die Transformations- und Vorverarbeitungskomponente vorgestellt werden. Die Vorverarbeitung von Quellcode ist notwendig um überhaupt Resultate auf realem Quellcode zu erzielen.

4.1.1 Zwei Komponenten Vorverarbeitung

Linux Gerätetreiber sind in einem Dialekt von ANSI-C [ISO99] geschrieben. Der Dialekt, gewöhnlich mit GNU-C bezeichnet, ist stark von der Entwicklung des GNU Compilers beeinflusst. Die Auswirkung auf die Verifikation besteht in der Notwendigkeit, den Code von GNU-C nach ANSI-C zu portieren. Andere Projekte wie CIL [NMRW02] haben diese Aufgabe bereits adressiert, jedoch zeigt sich, dass die von CIL umgesetzten Transformationen nicht ausreichend sind, wie Sauter erläutert [Sau07].

Für die Verifikation mittels CBMC müssen die unterstützten C Elemente berücksichtigt werden. Um einen Gerätetreiber so weit zu normalisieren, dass er als Eingabe für das ANSI-C Werkzeug CBMC geeignet ist, wurde das Quellcode Analyse- und Transformationswerkzeug CIL eingesetzt und erweitert.

Im Übersetzerbau bezeichnet der Begriff *Lvalue* alle Ausdrücke, die auf der linken Seite einer Zuweisung vorkommen können. Variablennamen gehören hierzu, Funktionsnamen sind nicht erlaubt.

CIL behandelt hierbei folgende Konstrukte, die von CBMC als Syntaxfehler klassifiziert werden[1]:

1. Verschachtelte Funktionsdefinitionen
2. Konstruierte Funktionsaufrufe:

   ```
   void * __builtin_apply (void (*function)(),
                           void *arguments, size_t size)
   ```

   Die obige Funktion ruft die Funktion `function` auf und liefert einen Zeiger auf das Ergebnis zurück.
3. Benennung eines Ausdruckstyps wie in `typedef name = exp;`
4. Komplexe Zahlen
5. `float` Konstanten, die in hexadezimaler Notation angegeben werden
6. Indexoperationen auf Arrays die keine Lvalues sind, das heißt also keine Modifikationen erlauben:

   ```
   struct foo {int a[4];};
   struct foo f();
   ```

[1] Die Spracherweiterungen des GCC Compilers sind etwa unter *http://gcc.gnu.org/onlinedocs/gcc-3.0.2/gcc_ 5.html* dokumentiert.

```
bar (int index)
{
      return f().a[index];
}
```

7. Forward-Deklarationen von Funktionsparametern

Weiterhin werden die folgenden Merkmale durch CIL [NMRW02] entfernt

1. Attribute für Funktionen, Variablen und Typen
2. Funktionsdefinitionen und Prototypen nach dem Kernighan und Ritchie Stil [KR78]
3. Lokal deklarierte Labels: Im Zuge der Übersetzung werden neue Labels generiert
4. Labelvariablen und berechnete Sprungziele eines `goto`-Befehls. GCC erlaubt es, die Adresse eines Labels zu berechnen und zu modifizieren. CIL simuliert dieses Verhalten durch Zuweisung neuer Konstanten die als Labeladressen dienen. Berechnete Sprünge werden über eine neue `switch`-Anweisung simuliert.
5. Generalisierte Zuweisungsziele. GCC erlaubt Zuweisungen wie etwa `(a, b) += 5`.
6. Der `?` Operator mit weniger als 3 Operanden `x ? : y` wird vervollständigt zu `x ? x : y`.
7. Unterstützung des Typs `long long` und des korrespondierenden `LL` Suffix für Konstanten. CIL verwendet zur Angleichung 64 Bit integer.
8. Lokale Arrays variabler Länge werden von CIL mittels der Funktion `alloca` implementiert. Nachfolgend werden alle Vorkommen der Arrayvariable durch den Rückgabezeiger ersetzt. Zusätzlich werden Ausdrücke mit `sizeof` angepasst.
9. Lokale Initialisierungen, die nicht konstant sind, werden in Zuweisungen umgewandelt.
10. Zusammengesetzte Literale werden in Zuweisungen übersetzt.
11. Designierte Initialisierungsanweisungen werden standardisiert:

```
int a1[6] = { [4] = 29, [2] = 15 };
int a2[6] = { 0, 0, 15, 0, 29, 0 };
```

CIL wandelt die Formulierung `a1` nach `a2` um.

12. `case`-Ausdrücke mit Intervallen werden in reguläre `case`-Anweisungen zerlegt.

13. Transparente `unions` sind eine Erweiterung, die durch das Attribut `transparent_union` eingeführt werden. Sie dienen dazu Bibliotheksfunktionen mit unterschiedlichen Schnittstellen konfliktfrei nebeneinander halten zu können. Hierzu wird einer der Funktionsparameter als transparente Union deklariert. Innerhalb der Union können dann die Untertypen die verschiedenen möglichen Schnittstellen kodieren. CIL ändert den formalen Funktionsparameter, so dass er den Typ des ersten Feldes in der `union` erhält.

14. Die GCC-Makros `__FUNCTION__` und `__PRETTY_FUNCTION__` werden durch entsprechende Konstanten ersetzt.

In Phase 1 der Vorverarbeitung wird CIL auf Gerätetreiber angewendet. Zusätzlich zu den von CIL vorgenommenen Vereinfachungen sind weitere Anpassungen notwendig, die im Rahmen der Avinux Werkzeugkette vorgenommen werden. Beispielhaft gibt die folgende Liste an, welche Änderung am Quellcode nötig sind, um CIL und CBMC auf den Quellcode anwenden zu können:

- `enum` Typen werden mittels `int` Konstanten kodiert.
- `__attribute__((...))` Dekoratoren werden entfernt.
- Aufgrund der Limitierung von CBMC auf die Kernsprache C können Assembleranteile nicht interpretiert werden. Die wenigen Codestücke in Assemblersprache werden auskommentiert - dies kann sowohl zu zusätzlichem wie auch zu unbetrachtetem Verhalten führen.
- `inline` und `register` Spezifizierer werden entfernt.
- Leere `struct` Datentypen erhalten eine zusätzliche Variable:

  ```
  struct struct_name { };
  ```

  wird ersetzt durch

  ```
  struct struct_name { int tmp; };
  ```

- Das `__alignof__(expr)` Makro wird durch eine Konstante '1' ersetzt. CBMC ignoriert diese Ausdrücke aufgrund des abstrakten Speichermodells, welches nicht von der tatsächlichen Anordnung im Speicher abhängig ist.

- Redundante `(char*)` Typumwandlungen vor Stringkonstanten werden entfernt.
- Arrays der Größe null werden durch einelementige Arrays ersetzt: `int x[0];` wird ersetzt durch `int x[1];`.
- Konstanten vom Typ `unsigned long long` werden in `unsigned long` Konstanten umgewandelt: aus `0x0ULL` wird `0x0UL`.
- Unvollständige Typen werden in vollständige Typen umgewandelt:
  - `extern void x;` wird zu `extern int x;`.
  - `extern int x[];` ist gleich zu `extern int * x;`.

Eine vollständige Auflistung der notwendigen Änderungen und deren Implementierung wird von Sauter [Sau07] gegeben. Erwähnenswert ist, dass der zeitliche Aufwand zur Erstellung der Komponenten zur syntaktischen Anpassung des Quellcodes das Gesamtprojekt Avinux dominiert hat. Die unzureichende Dokumentation von Standardabweichungen in Linux, sowie knappe Fehlermeldungen der Frontends von CIL und CBMC führten zu einer *Blackbox* Analyse der syntaktischen und semantischen Probleme.

In der zweiten Phase wird die Aufgabe adressiert, relevante Spezifikation für Gerätetreiber zu extrahieren und zu formalisieren. Projekte wie der Static Driver Verifier (SDV) [BBC+06] liefern einen festen Regelsatz von über 70 Regeln, die jeder Treiber erfüllen muss. Die Regeln gelten allerdings speziell für Treiber des Windows Driver Frameworks und sind somit nicht auf Linux Treiber anzuwenden.

Eine weitere notwendige Voraussetzung für die Verifikation von Linuxtreibern ist eine Spezifikationssprache zur Formulierung von Schnittstelleneigenschaften. Die im SDV verwendete Spezifikationssprache SLIC [BR01] ist nicht verfügbar.

Unter diesen Voraussetzungen ist es notwendig, den im Static Driver Verifier [BBC+06] geleisteten Spezifikationsansatz komplett neu zu implementieren. Ergebnis dieser Neuimplementierung ist die Erweiterung SLICx. Zunächst wird beschrieben, welche Eigenschaften mit Hilfe von SLIC und SLICx kodiert werden.

4.1.2 Schnittstelleneigenschaften für Linux

SLIC ist die Spezifikationssprache für Schnittstelleneigenschaften, die im Static Driver Verifier (SDV) [BBC+06] verwendet wird. Im Folgenden wird zunächst SLIC beschrieben. Hiernach wird die Sprache SLICx definiert, welche SLIC reimplementiert und um einige Aspekte erweitert (Abschnitt 4.1.2).

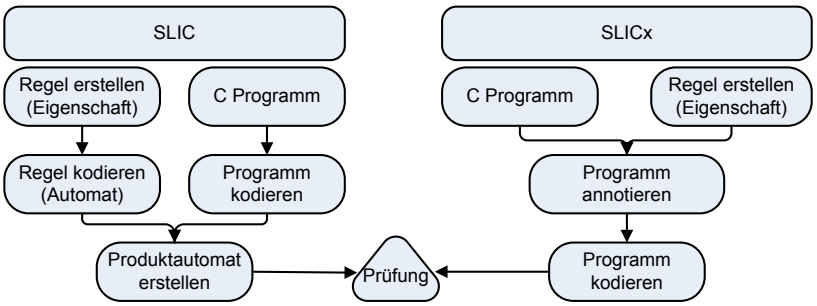


Abbildung 4.1: SLIC und SLICx erlauben es, Prüfeigenschaften in der Form von Regeln zu erstellen.

Abbildung 4.1 enthält eine Übersicht über den Prozess der Spezifikation von Prüfeigenschaften mit SLIC [BR01] im Rahmen des Static Driver Verifier. In gleicher Abbildung wird der Prozess der Spezifikation von Eigenschaften mit SLICx und CBMC entgegengestellt: Die Verwendung von SLIC im Rahmen des SDV [BBC+06] führt dazu, dass Regel und Programm separat in eine Automatendarstellung überführt wird. Anschließend wird das Produkt beider Automaten gebildet und geprüft. SLICx verfolgt den Ansatz, Eigenschaften direkt in das Programm einzufügen (annotieren). Diese Methode war ursprünglich auch für den Einsatz von SLIC [BR01] vorgeschlagen, aber so nicht umgesetzt worden.

Ein Beispiel einer Schnittstelleneigenschaft, die mit Hilfe von SLIC formuliert wird, ist die eines paarweisen Aufrufs eines Funktionspaares `spin_lock(...)` und `spin_unlock(...)`.

Locks (Schlösser) sind im Kontext paralleler Systeme ein Mittel zur Synchronisation nebenläufiger Programme. Eine häufige Anwendung ist die Sequentialisierung von mehreren parallelen Zugriffen auf eine Ressource, zum Beispiel eine Datei. Sobald ein Prozess oder ein Thread die Ressource nutzt, werden alle anderen Prozesse, die ebenso Zugriff erlangen wollen, blockiert. Umgesetzt wird dieses Modell mit Hilfe eines Locks welches vor jedem Schreib- oder Lesezugriff von einem Prozess angefordert werden muss, aber nur von einem Prozess gleichzeitig besessen werden kann.

Zur Anforderung eines Locks stehen festgelegte Schnittellen bereit. Im Linux Kontext sind dies unter anderem die Funktion `spin_lock(&lock)` sowie die inverse Funktion `spin_unlock(&lock)`. Ein Problemfall besteht nun darin, dass ein Prozess, der ein dediziertes Lock bereits erhalten hat, erneut dasselbe Lock anfordern kann. Falls dieser Fall nicht von der Implementierung abgefangen wird[2], wartet nun der Prozess auf eine Ressource, die von ihm selbst nicht zurückgegeben wird. Es tritt der Fall eines zyklischen Wartens

[2]Linux wie auch Windows Implementierungen der Locks überprüfen diesen Fall nicht.

ein – dieser Effekt wurde bereits von [CES71] beschrieben und ist noch immer eine der häufigsten Ursachen für den Stillstand (*Deadlock*) eines Teilsystems.

Eine Spezifikation soll nun eine hinreichend starke Bedingung definieren, die eine solche Situation verhindert. Der folgende Satz formuliert die Bedingung[3]:

> Ein Prozess darf niemals die Funktion `spin_lock` aufrufen solange er das angeforderte Lock bereits hält.

Die Bedingung ist offensichtlich eine Sicherheitseigenschaft. Es ist zur Prüfung dieser Eigenschaft ausreichend, Programmzustände nach Aufruf von Funktionen, in diesem Fall `spin_lock`, zu betrachten. Hieraus erwächst das Design von SLIC wie auch unserer Erweiterung SLICx: Ein endlicher Sicherheitsautomat wechselt den Zustand immer dann, wenn eine spezifizierte Schnittstellenfunktionen aufgerufen werden. Endliche Automaten können zur Definition einer regulären Sprache verwendet werden. Versteht man die Menge der möglichen Programmausführungen als Wörter und die Korrektheitseigenschaft als reguläre Sprache, dann ist die Frage, ob ein Pfad eine Eigenschaft erfüllt, äquivalent zu der Frage, ob ein Wort zu einer Sprache gehört. Stellt man den Automaten, der eine Eigenschaft kodiert als LTS oder Kripke Struktur dar, so kann ein Model Checking Problem auf dem Produktautomaten aus Modell und Spezifikation berechnet werden.

Das Design von Sicherheitsautomaten, die beim Aufruf von Schnittstellen ihren Zustand wechseln, beinhaltet eine Beschränkung die aus Effizienzgründen vorgenommen wurde: Eigenschaften, die Fehlerzustände abseits von den Aufrufen von Schnittstellenfunktionen betreffen, können nicht direkt mit SLIC spezifiziert werden. Beispiele für nicht direkt prüfbare Eigenschaften sind Zugriffsverletzung bei Arrays und Zeigern, sowie arithmetische Ausnahmen und Probleme beim parallelen Zugriff auf gemeinsame Ressourcen durch Zeiger.

SLIC wird zusammen mit dem Static Driver Verifier eingesetzt, der nicht alle der obigen Ausnahmen selbst unterstützt. CBMC prüft einige der obigen Eigenschaften von sich aus, das heißt, ohne dass eine manuelle Spezifikation durch `assert/assume` formalisiert werden müsste. Die Abdeckung von Ausnahmen, die weder von CBMC noch von SLIC direkt unterstützt werden, wird weiter unten erläutert.

Ohne genau auf die Definition von SLIC einzugehen, führen wir nun direkt SLICx ein. SLIC wird am Ende des nächsten Abschnitts durch die Unterschiede zu SLICx differenziert.

[3] Die Bedingung ist nicht stark genug, jede Art von Problemen mit der *Lock*-Schnittstelle zu umfassen.

Eigenschaften in SLICx

SLICx ist eine direkte Erweiterung und Neuimplementierung der Spezifikationssprache SLIC [BR01]. SLICx wird verwendet um endliche nichtdeterministische Automaten zu definieren. Diese Automaten beschreiben Sicherheitseigenschaften beziehungsweise Protokolle deren Einhaltung sich als eine Sicherheitseigenschaft formulieren lässt. In Tabelle 4.1 wird die Syntax von SLICx mit den Unterschieden zu SLIC aufgelistet.

Wie jeder endliche Automat besteht auch der durch SLICx beschriebene aus einer Zustandsmenge und einer Definition von möglichen Übergängen zwischen Zuständen. Ein einfaches Beispiel einer Zustandsraumdefinition in SLICx besteht aus folgender Zeile:

```
// SLICx Zustand
1: state {
2:  int zaehler;
3: }
```

Das obige Beispiel beschreibt einen Zustandsraum, der aus einer wie in ANSI-C definierten `int` Variable besteht. Durch folgende SLICx Beschreibung wird eine Menge von Übergängen $\{0 \mapsto 1, 1 \mapsto 2, ..., \texttt{MAX\_INT} \mapsto \texttt{ERROR}\}$ definiert. `MAX_INT` ist hierbei der größte darstellbare Wert des C `int` Typs. `ERROR` ist ein speziell markierter Fehlerzustand auf dessen Erreichbarkeit geprüft wird.

Zur Spezifizierung von Übergängen des Zustands definiert SLIC sogenannte *Patterns*. Diese bestehen aus dem Namen einer Funktion und einem von vier möglichen *Events*: *call, return, entry* und *exit*. Ein Übergang kann so aussehen:

```
// SLICx Übergang
1: zaehle.entry {
2:  if(zaehler == MAX_INT -1) abort "Fehler: Überlauf";
3:  zaehler = zaehler + 1;
4: }
```

Dieser Übergang wird immer dann ausgeführt wenn der Kontrollfluss in die Funktion `zaehle` eintritt (*entry*). Der genaue Übergang des Zustands wird in SLICx ähnlich einer C-Funktion kodiert, die die SLICx Zustandsvariablen, wie `zaehler`, modifizieren kann.

Eine Angabe der Menge der Folgezustände findet sich in Zeile 3. Die Addition wird jedoch nur ausgeführt, wenn in Zeile 2 kein möglicher Überlauf detektiert wird. Sollte es zu einem Überlauf kommen, spezifiziert das `abort`

Statement einen Übergang zum Fehlerzustand `ERROR`. Überlicherweise wird die Verifikation beim Erreichen eines solchen abgebrochen.

Innerhalb der Übergänge sind einige weitere reservierte Wörter mit spezieller Semantik erlaubt:

halt Beendet das Programm ohne einen Fehlerzustand.

reset Setzt den Zustand des Sicherheitsautomaten auf den initialen Zustand.

***** Steht für einen nichtdeterministisch gewählten Wert für den der Modellprüfer beide Möglichkeiten prüfen soll. In CBMC entspricht dies dem Ausdruck `nondet_bool()`.

$i Das i-te Argument des letzten Aufrufs der Funktion wird mit $i bezeichnet.

$return $return verweist auf den Rückgabewert einer Funktion.

Mit SLIC und SLICx wird kein Produktautomat explizit konstruiert. Statt dessen wird das zu prüfende Programm direkt mit den Spezifikationen *annotiert*:

Eine Transition des Prüfautomaten wird genau dann ausgeführt, falls eine Funktion `zaehle` aufgerufen wird. Durch die Verbindung zwischen der Transition des Automaten und dem Aufruf einer Funktion wird der Zustandsautomat an mögliche Programmausführungen gekoppelt. Ist in dem zu verifizieren Programm ein Aufruf der Funktion `zaehle` vorhanden, wird der Quellcode in folgender Weise annotiert:

```
// Originaler Quellcode
1: zaehle();

// Annotierter Quellcode
// Im Falle eines Überlaufs wird
// das Programm mit einem Fehler beendet.
  if (slicx.zaehler == MAX_INT -1) assert(0);
// Zustandstransition im Automaten
  slicx.zaehler = slicx.zaehler + 1;
1: zaehle();
```

Um ungewollte Kollisionen zwischen den Zustandsvariablen einer Regel und dem Namensraum eines Programms zu vermeiden, werden Namen durch einen Namenspräfix `slicx.` getrennt.

Tabelle 4.1: SLICx Syntax in Backus-Naur Notation. In der rechten Spalte werden Unterschiede zu SLIC erläutert.

Syntax			**Unterschied zu SLIC**
S	::=	$(extDecl)^*$ $[state]$ $transFun^*$	Zu Beginn einer SLIC Regel dürfen beliebige C Deklarationen und Funktionsdefinitionen stehen.
state	::=	**state** { $fieldDecl^+$ }	Die Zustandsstruktur selbst ist unverändert.
fieldDecl	::=	*fieldType id* = *expr* ;	Felder sind bestimmte C Typen
	|	**enum** { *id* (, *id*)$^+$ } *id* = *id* ;	oder enum
fieldType	::=	**int *** | **int** | **void ***	Typen eingeschränkt
transFun	::=	*pattern stmt*	Die Übergangsfunktionen selbst sind unverändert.
pattern	::=	*id* . *event*	
event	::=	**call** | **return** | **entry** | **exit**	
stmt	::=	*id* = *expr* ;	Parallele Zuweisungen durch normale ersetzt.
	|	**if** (*choose*) *stmt* [**else** *stmt*]	
	|	**abort** *string*;	
	|	**reset;**	
	|	**halt;**	
	|	*cStmt*	Alle sonstigen C Anweisungen
choose	::=	*	unverändert
	|	*expr*	
expr	::=	*cExpr*	Alle C Ausdrücke
id	::=	*C_identifier*	unverändert
	|	**$** *int*	max. 10 Parameter
	|	**$ return**	unverändert
	|	**$** *C_identifier*	unverändert

Bemerkenswert ist der Zusammenhang mit *Aspekt-orientierter Softwareentwicklung*. Hierbei werden verschiedene funktionale Aspekte eines Programms aufgetrennt und separat entwickelt. Erst kurz vor der eigentlichen Kompilation findet ein Zusammenfügen der Programmaspekte statt. Im Falle von SLIC und SLICx ist die Spezifikation von Eigenschaften ein Aspekt der automatisch und nachträglich in ein Programm eingefügt werden kann.

4.2 Anwendung auf Linux

Für Linux lassen sich eine Reihe von relevanten Eigenschaften finden, die durch eine Spezifikation in SLICx geprüft werden können. In den folgenden Abschnitten stellen wir die Eigenschaften für bestimmte Teile des Linux Betriebssystems vor und geben vereinzelt Beispiele wie sie in SLICx zu for-

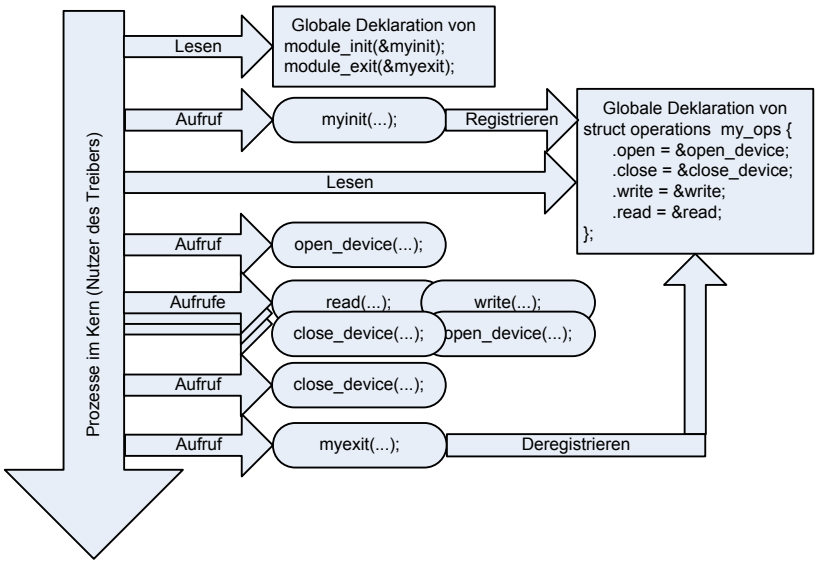


Abbildung 4.2: Linux Treiber stellen eine Schnittstelle zur eigenen Initialisierung (`module_init`, `module_exit`) bereit. Zusätzlich werden Operationen, die andere Kernprozesse ausführen dürfen, im Rahmen der Initialisierung registriert (`struct operations`).

mulieren sind. Die eigentliche Verifikation der Regeln wird in Abschnitt 4.3 beschrieben.

Zunächst führen wir kurz das Verifikationsumfeld ein.

4.2.1 Linux Treiber

Im Gegensatz zur Verifikation der AES Implementierungen aus Kapitel 3 sind Linux Treiber offene Software. Dies wirft das Problem auf, dass Daten in Funktionen berechnet werden, die nicht Bestandteil eines Verifikationslaufs sind (zum Beispiel Bibliotheksfunktionen). Noch problematischer ist die Unkenntnis des genauen Ablaufs innerhalb eines Treibers.

Abbildung 4.2 stellt die typische Interaktion eines Treibers mit dem restlichen Betriebssystem dar (dem *Kern*). Ein Treiber stellt über Makros die Namen von Funktionen bereit, die zur Initialisierung und zum Entladen des Treibers aufgerufen werden müssen. Der Aufruf wird vom Betriebssystem vorgenommen.

Ist ein Treiber initialisiert und zuvor in den Speicher geladen worden, kann ein beliebiger Prozess im Kern Servicefunktionen des Treibers aufrufen. Hierzu zählen das Anmelden einer neuen Hardware, beispielsweise ein USB-Speichergerät das im laufenden Betrieb angeschlossen wurde. Für Speicher-

hardware stellt der Treiber in der Form einer Schnittstelle noch Operationen wie Lesen (*read*) und Schreiben (*write*) bereit.

Nachdem alle Hardwaregeräte vom Treiber abgemeldet sind – in Abbildung 4.2 durch die Funktion `close_device` – kann der Treiber auf das Entladen vorbereitet werden. Spätestens hier müssen alle vom Treiber dynamisch allozierten Daten freigegeben werden.

Häufige Szenarien, unter denen ein Treiber nicht korrekt arbeitet, sind Fälle, in denen bereits bei der Initialisierung ein Fehler auftritt: Das Gerät wird physikalisch abgetrennt während der Treiber gestartet wird [CRKH05].

Die Sprache SLICx, SLICx-Regeln gemäß obiger Spezifikation und das Verifikationswerkzeug CBMC bilden zusammen eine Werkzeugkette ähnlich dem Static Driver Verifier (SDV) [BBC+06]. Detaillierte Angaben über den Aufbau und die Implementierung der Werkzeugkette in der SLICx eingebettet ist, geben Post et al. in [PSK08]. Beispiele für Regeln sind unter der Webseite http://www-sr.uni-tuebingen.de/~post/avinux zu finden.

Avinux kann im Gegensatz zu CBMC alleine und dem Static Driver Verifier folgende wichtige Eigenschaften analysieren:

1. Nachweisen der Abwesenheit von Speicherlecks
2. Sequentielle Simulation von preemptiver Parallelität
3. Nachweis der Abwesenheit von Deadlocks
4. Nachweis der Abwesenheit von Race-Conditions

Es existieren zahlreiche Werkzeuge im Bereich der statischen Analyse und des Quellcode Model Checking die obige Probleme heuristisch untersuchen. In Avinux gelingt es alle Fehlerquellen in einer SDV ähnlichen Weise direkt zu analysieren. Im Folgenden werden einzelne Spezifikationen beziehungsweise die korrespondierenden Regeln erläutert. Falls die Tatsache, dass ein Programm eine Regel erfüllt, impliziert, dass das Programm den betreffenden Fehler wirklich nicht enthält, nennen wir eine Regel *schlüssig*. Im umgekehrten Fall, dass die festgestellte Verletzung einer Regel die Existenz eines wirklich im Programm vorliegenden Fehlers erzwingt, nennen wir die Regel *vollständig*.

Die Spezifikationen, die den Regeln zugrunde liegen, sind, so sie nicht direkt vom ANSI-C Standard abgeleitet sind, der Dokumentation des Linux Kerns [Varb] und aus dem Linux Device Driver Buch von Corbet, Rubini und Kroah-Hartman [CRKH05] entnommen. Schnittstellenregeln werden häufig mit neuen Versionen des Linux Kerns geändert: die vorliegende Referenzversion ist der Linux Kern 2.6.18.

Auf den folgenden Seiten stellen wir dar wie sich mit SLICx und CBMC typische Probleme aus der Betriebssystemverifikation formulieren und untersuchen lassen. Die Probleme werden kurz erläutert ohne zu sehr auf die Hintergünde von Betriebssystemarchitekturen einzugehen. Der interessierte Leser sei auf Standardwerke wie *Operating System Concepts* von Silberschatz et al. [SGG04] verwiesen.

4.2.2 Referenzzählung

Bei einem automatischen Speichermanagement wird häufig der Mechanismus der Referenzzählung (*Reference counting*) angewandt. Objekte, die dynamisch alloziert wurden, können freigegeben werden, sobald keine Referenzen mehr auf sie existieren. Im Linux Betriebssystem wird das Referenzzählen zusammen mit der Verwaltung hierarchischer Strukturen umgesetzt.

Mit der Integration von Stromsparmechanismen in den meisten Bus- und Hardwareschichten wurde die Repräsentation von Abhängigkeiten der Geräte und Busse notwendig: *Ein Gerät kann nur abgeschaltet werden, falls alle abhängigen Geräte auch abgeschaltet werden können.* In Linux wird diese Information durch Einbettung von `kobject`s in andere Strukturen mitgeführt. Neben der hierarchischen Struktur kodieren diese das Entwurfsmuster der Referenzzählung, die sichere Deallokation ermöglicht: *Ein Objekt darf genau dann dealloziert werden, falls keine Referenzen auf dieses Objekt existieren.*

Vor jedem Zugriff auf eine Instanz von einem `kobject` muss die Funktion `kobject_get(object)` aufgerufen werden. Nachdem alle Schreib- und Lesezugriffe erfolgt sind, kann mittels `kobject_put(object)` das Objekt freigegeben werden. Wir isolieren hieraus notwendige Regeln, die für die korrekte Benutzung von `kobjekt`s eingehalten werden müssen.

1. Falls der Referenzzähler null ist darf die Funktion `kobject_put` nicht für dieses Objekt aufgerufen werden.
2. Ein Zugriff auf ein Objekt mit Referenzzähler null ist verboten.
3. Ein Objekt welches nicht in anderen Modulen verwendet wird, muss am Ende des Lebenszyklus eines Moduls einen Referenzzähler von null aufweisen.

Wir betrachten nur diesen Teil der Schnittstelle und nennen die Regeln zusammen *Kobjekt Referenzzählung* (KRC). Die obigen Teilaspekte nennen wir KRC1, KRC2 und KRC3.

Die in den Regeln annotierten Funktionen sind `kobject_init`, `kobject_put` und `kobject_get`. Die Kodierung des Sicherheitsautomaten in SLICx ist

```
// Nach Aufruf von kobject_put
kobject_put.exit{
// Zeigt der erste Parameter NULL?
   if ($1){
// Wird das richtige Objekt
// behandelt?
     if (which_kobject == $1){
// Erniedrige Zahl der Referenzen
       reference_count--;
// Zu viele Freigaben ausgeführt?
       if (reference_count == -1){
// KRC1 verletzt
// Abbruch mit Fehler
          abort "ERROR: KRC1";
       }
     }
   }
}
```

```
// Nach Aufruf der letzten
// Funktion
main.exit{
// Sind nicht alle Referenzen
// freigegeben?
    if (reference_count>0){
// Abbruch mit Fehler KRC3
          abort "ERROR: KRC3";
    }
}
```

Abbildung 4.3: Der Auszug aus einer SLICx Regel für KRC zeigt wie KRC1 und KRC3 umgesetzt werden. Die Übergangsfunktionen für `kobject_init` und `kobject_get` wurden ausgelassen.

direkt möglich:

- `kobject_init` setzt den Referenzzähler auf eins. Dies wird durch einen internen Aufruf zu `kobject_get` realisiert was im Folgenden berücksichtigt werden muss.
- `kobject_get` erhöht den Zähler um eins.
- `kobject_put` erniedrigt den Zähler um eins.

In der zugehörigen SLICx Regel wird der Status des Automaten durch zwei Felder kodiert: `void * which_kobject` ist ein Zeiger auf die Instanz eines `kobject`, die gerade kontrolliert wird; `int reference_count` implementiert den Referenzzähler. Die Implementierung der Zustandsübergangsfunktion ist in Abbildung 4.3 dargestellt.

Die Implementierung der Regel KRC2 ist der Implementierung der Prüfung von Wettlaufsituationen (Race-Condition) sehr ähnlich und wird deshalb erst im Zusammenhang mit diesen in Abschnitt 4.2.7 dargestellt. Um KRC3 zu überprüfen nutzen wir aus, dass Linux bei Modulen ein Standardgerüst vorgibt. Der Lebenszyklus eines Moduls endet immer genau dann wenn die Funktion, die durch das Makro `module_exit` gekennzeichnet wurde, beendet ist. Falls keine Funktion gekennzeichnet wurde, kann das Modul nicht entladen werden und KRC3 ist trivialerweise erfüllt. Es bleibt also nur die Überprüfung, dass am Ende der Entladefunktion der Referenzzähler null ist.

Anstatt direkt die letzte Funktion eines jeden Moduls zu annotieren, weichen wir auf eine für den Treiber von uns generierte `main` Funktion aus. Selbige ruft als letzte Handlung die Entladefunktion des aktuellen Moduls auf (Abbildung 4.3). Gilt die Bedingung nach `main`, so gilt sie automatisch nach der letzten Funktion des Lebenszyklus.

4.2.3 Speichermanagement

Zugriffsprüfung durch CBMC

Die Validität von Speicherzugriffen betrifft das Vorhandensein von ungültigen, zum Beispiel `* (void *) 0`, und undefinierten Zugriffen auf geschützte oder deallozierte Speicherbereiche. Die geprüften Eigenschaften werden durch den ANSI-C Standard vorgeschrieben und gelten deshalb für alle Programme (generische Spezifikation):

- Dereferenzierung von `NULL` Zeigern
- Verwendung von Referenzen, die bereits dealloziert wurden
- Zugriff auf nicht initialisierte Objekte
- Dereferenzierung von Adressen außerhalb der Grenzen eines Objektes
- Aufruf von Funktionen über Funktionszeiger mit einem Offset

CBMC [CKL04] hat Prüfungen der obigen Eigenschaften eingebaut, das heißt diese müssen nicht extra durch `assert/assume` spezifiziert werden.

Neben diesen sprachspezifischen Fehlerquellen existiert das bereits angesprochene Problem, dass Speicherlecks auftreten können. Gerade in lange laufenden Programmen, wie der Implementierung eines Serverbetriebssystems, stellen diese Speicherlecks ein Problem dar.

Speicherlecks (SL)

Das Linux Betriebssystem stellt zur Allokation von Speicherbereichen auf dem Heap die Funktion `kmalloc()` zur Verfügung. Speicher, der nicht mehr benötigt wird, muss über einen Aufruf von `kfree()` freigegeben werden. Dies lässt sich direkt in eine Spezifikation einer Lebendigkeitseigenschaft umschreiben: *Nach* `kmalloc` *muss irgendwann* `kfree` *aufgerufen werden.*

Diese Spezifikation kann für Linux Module angepasst werden. Da der Lebenszyklus eines Moduls an einem definierten Punkt endet, muss `kfree` spätestens an diesem Punkt aufgerufen worden sein. Dieser Punkt wird in der

Praxis beispielsweise bei Plug-and-Play Geräten erreicht, deren Treiber entladen werden kann sobald das Gerät entfernt wurde.

Die Lebendigkeitseigenschaft kann also in eine Sicherheitseigenschaft umformuliert werden falls sichergestellt ist, dass ein Modul immer das Ende des Lebenszyklus erreicht. Die modifizierte Spezifikation lautet:

Niemals darf das Ende von `main` *erreicht werden falls ein Objekt nicht dealloziert wurde.* (SL)

Dies ist ein genereller Reduktionsmechanismus von Lebendigkeitseigenschaften auf Sicherheitseigenschaften unter der zusätzlichen Beweisverpflichtung, dass ein Programm terminiert. Diese Reduktion wird von Werkzeugen wie Terminator [CPR06] ausgenutzt um durch eine Terminationsanalyse sämtliche Lebendigkeitseigenschaften abdecken zu können.

Wir nennen diese Regel SL. Abbildung 4.2 zeigt den Lebenszyklus eines Linux Moduls auf. Die Umsetzung desselben muss mittels eines Modells des Betriebssystems erreicht werden. Andernfalls kann CBMC keine Informationen über die korrekte Reihenfolge möglicher Aufrufe von externen Schnittstellen nutzen, die das Modul zur Verfügung stellt. Modelle wie dieses finden sich auch im SDV [BBC+06], jedoch wurde dieses Verfahren bisher nicht eingesetzt um Speicherlecks zu finden.

Die Prüfung dieser Regel betrifft eine potentiell unbegrenzte Zahl von Objekten. Durch die endliche Entfaltung kann man jedoch die Menge betroffener Ressourcen auf eine endliche Menge einschränken. Aufgrund der Formulierung der Regel kann man zudem die Einhaltung der Eigenschaft für alle Objekte unabhängig voneinander prüfen.

Anstatt für jedes Objekt einen eigenen Sicherheitsautomaten einzuführen, bedienen wir uns des *Universal Quantification Trick (UQT)*: Aus der Menge aller Objekte wird eines nichtdeterministisch ausgewählt und separat überprüft. Durch den Nichtdeterminismus betrachtet der Modellprüfer das korrekte Verhalten bezogen auf jede Instanz des Objektes separat. Die erste Nennung dieses Vorgehens findet sich in dem Software Modellprüfer Bandera [CDH+00].

Eine Erweiterung des Tricks wird bei Implementierung der Regel 'Lock Ordnung' (LO) vorgenommen: Anstatt ein Objekt auszuwählen wird ein Paar von Objekten gewählt.

Zur Umsetzung des obigen Mechanismus bei der SL Regel wird die Allokationsfunktion annotiert. Zur Prüfung der Lebendigkeitseigenschaft annotieren wir die Funktion, die dem Makro `module_exit` als Parameter übergeben wird.

4.2.4 Simulation von Preemption (PS)

Die bisherigen Regeln können mit CBMC und SLICx nur für sequentielle Programme analysiert werden. Parallele Ausführungen, das heißt die Verwendung mehrerer Prozesse oder *Threads of Control*, bedingen eine exponentielle Vergrößerung der Zahl möglicher Programmausführungen: Nach jeder Anweisung kann jeder der unabhängigen Prozesse als nächstes ausgeführt werden. Synchronisation, etwas durch Locks, schränkt diese Verzahnung ein – eine exponentielle Vergrößerung bleibt in Programmen der Praxis jedoch bestehen, da nur sehr kleine Programmblöcke Einschränkungen der Parallelität vornehmen sollen. Anderenfalls, bei einer grobgranularen zeitlichen Verzahnung, besteht die Gefahr, dass der Geschwindigkeitsvorteil, den die parallele über einer sequentiellen Architektur bietet, nicht mehr genutzt werden kann [CRKH05].

Eine Möglichkeit Parallelität zu modellieren besteht darin eine sequentielle Simulation paralleler Abläufe zu entwickeln. Die Werkzeuge KISS [QW04] und TCBMC [RG05] implementieren solche Verfahren und dienen im Folgenden als Vergleich für die Methode, die wir mittels SLICx direkt umsetzen können. Der Vergleich basiert auf der Ausdrucksstärke der Verfahren, das heißt der Frage welche möglichen Programmverzahnungen überhaupt analysierbar sind. Im Gegensatz zu Ansätzen, die vollständige Modellierung von Parallelität erlauben, beschränken wir uns auf die Modellierung von Unterbrechungen (Preemption) durch *Interrupts* oder abstrakte Umgebungsmodelle. Nach einer solchen Unterbrechung (aber nicht vorher!) wird das Programm oder der Prozess fortgeführt.

Abbildung 4.4 zeigt ein typisches Szenario auf einem Multiprozessorsystem: Eine Servicefunktion nimmt eine Anfrage entgegen, wird aber durch einen *Interrupt* unterbrochen. Die Unterbrechung bedingt eine Ausführung der mit dem Interrupt assoziierten Serviceroutine. Letztere kann wiederum durch Interrupts höherer Priorität und Fehlerbehandlungsroutinen (beispielsweise die Behandlung von *Page fault* Ausnahmen) unterbrochen werden. Die genaue Verschachtelung ist Betriebssystem-spezifisch. In Linux kann die Fehlerbehandlung selbst nicht wieder unterbrochen werden. Die Ausführung von Interrupt Service Routinen (ISR) kann in Linux durch reservierte Befehle zeitweise ausgestellt werden. Die Modellierung des Interruptzustands ist somit auch von Belang um keine falschen Programmflüsse zu modellieren.

Die Motivation für die gewählte Modellierung basiert auf zwei Beobachtungen die wir durch Studien an den Linux Treibern gewonnen haben:

- Die Möglichkeit einer Unterbrechung hängt vom Zustand der ISR ab.
- Die häufigste Art der parallelen Ausführung in Linux ist die Interaktion zwischen ISR und Nicht-ISR Treiberquellcode.

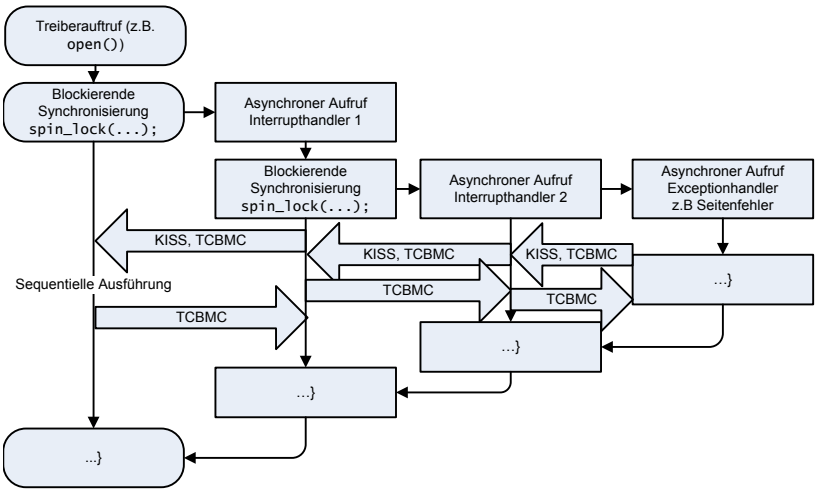


Abbildung 4.4: Parallele Programmausführungen können durch die zusätzliche Ausführungspfade im sequentiellen Programm simuliert werden. KISS, TCBMC und unser Ansatz zeigen eine unterschiedliche Abdeckung der simulierten Pfade. Obwohl die Simulation mit SLICx die kleinste Abdeckung hat, zeigt sie für systemnahe Programme eine gute Balance zwischen der Abdeckung praktisch bedeutsamer Unterbrechungen und dem exponentiellen Anstieg der Laufzeit.

Zunächst stellen wir kurz Abdeckungen von parallelen Ausführungen in den Werkzeugen KISS [QW04] und TCBMC [RG05] vor:

1. In KISS wird jeder Befehl mit zusätzlichem Quellcode instrumentiert. Dieser simuliert einen Aufruf von nebenläufigem Code und einen weiteren Kontextwechsel. In dem obigen Beispiel betrachtet KISS auch den Fall, dass die ISR terminiert, bevor sie vollständig ausgeführt wird. Im Allgemeinen ist dieses Verhalten möglich wenn zwei parallele Prozesse sich abwechseln. Im vorliegenden Fall jedoch wird von Linux gewährleistet, dass die ISR immer beendet wird, bevor der Treiberkontrollfluss wieder hergestellt wird. In einem System mit nur einem Prozessor wäre dies die Einführung von Verhalten, das ein Treiber nicht zeigen wird. Generell simuliert KISS strikt mehr Verhalten, als in unserer Methode – möglicherweise aber auch nicht im echten System vorliegendes.

2. TCBMC ist eine Erweiterung von CBMC, die eine beschränkte Anzahl von Kontextwechseln simuliert. Im Beispiel betrachtet TCBMC zusätzlich zu KISS die Situation in der ein Interrupthandler auftritt, selbst wieder unterbrochen wird, und nachfolgend weiter abgearbeitet wird. TCBMC kann bei genügend hoher Anzahl von Kontextwechseln mehr Verhalten als KISS prüfen. Bei geringen Anzahlen von Kontextwechseln prüft KISS mehr Verhalten.

Preemption mit SLICx In SLICx kann die Unterbrechung von Linuxtreibern durch Interrupts durch explizite, nichtdeterministische Aufrufe der ISR realisiert werden. Dieser Simulationsaspekt wird Preemption Simulation (PS) genannt. Hierbei wird die ISR nur aufgerufen wenn Interrupts auch eingeschaltet sind. Bemerkenswert ist, dass auch Interrupthandler instrumentiert werden können, so dass eine verschachtelte Unterbrechung modelliert werden kann.

Der generische Teil der Modellierung besteht in der Erfassung von Zuständen der Interrupts: die globale Deaktivierung, Aktivierung und das Registrieren einer neuen ISR werden annotiert. Gespeichert wird eine Boolesche Variable, die die Aktivität von Interrupts kodiert, sowie ein Funktionszeiger auf die registrierte ISR.

Die Instrumentierung durch den Aufruf der ISR kann auf relevante Funktionen beschränkt werden. Relevant sind in diesem Sinn alle Funktionen, die im Rahmen der PS analysiert werden sollen. Will man die Referenzzählung unter einem Szenario mit PS überprüfen, so ist es ausreichend jedes Teilprogramm *zwischen* `kobject_init`, `kobject_get` und `kobject_put` Anweisungen nur einmalig mit einem Aufruf der ISR zu versehen. Der Vollständigkeit halber muss vor dem Ende des Lebenszyklus ebenfalls noch einmal die ISR aufgerufen werden[4]. Die Reduktion führt zu einer begrenzten Zunahme der Komplexität. Eben diese Beschränkung ermöglicht es Parallelität zu untersuchen ohne sich auf zu kleine Programme beschränken zu müssen. Die Beschränkung der Parallelität ermöglicht also eine höhere Performanz sowie eine Verminderung unrealistischer Verzahnungen durch die generische Modellierung. PS bietet keine eigenständige Spezifikation sondern nur eine Erweiterung der anderen Regeln.

Ein Beispiel für die Anwendung von PS findet sich in Abbildung 4.5.

4.2.5 Schlüssiges Locking (AL)

Eines der häufigsten Beispiele für eine Sicherheitseigenschaft ist die Regel 'Alternierendes Locking'. Sie besagt, dass für ein Lock die Funktionen `lock` und `unlock` immer abwechselnd aufgerufen werden müssen. Die doppelte Freigabe einer Ressource durch `unlock;unlock` ist hierbei weniger kritisch als die doppelte Anforderung derselben. In letzterem Fall wartet ein Prozess (oder ein Thread) auf eine Ressource die er selbst hält. Dies ist das kürzeste Beispiel für ein zirkuläres Warten welches einen Deadlock (Stillstand) auslöst. Die Regel ist schlüssig in dem Sinne, dass eine Verletzung immer auf ein echtes Problem hindeutet, während die Einhaltung der Regel keine Freiheit von Deadlocks garantiert. Die Regel kann fast identisch zu ihrer For-

[4] Eine zusätzliche Annahme ist, dass ein zweimaliger Aufruf der ISR irrelevant ist.

mulierung in SLIC [BR01] übernommen werden. Im Gegensatz zu [BBC+06] erweitern wir die Regel insofern als dass alle Linux Funktionen abgedeckt werden.

4.2.6 Vollständiges Locking (LO)

Im Folgenden präsentieren wir eine vollständige, aber nicht schlüssige, Regel bezogen auf das Problem der Deadlocks die durch die Lock-Schnittstellen in Linux entstehen können. Coffman, Elphick und Shoshani [CES71] präsentieren vier notwendige Bedingungen, die erfüllt sein müssen damit ein Deadlock vorliegen kann. Drei dieser Regeln sind bereits durch die Implementierung der Locks in Linux zwangsweise erfüllt. Folglich kann man die Entstehung von Deadlocks durch eine zwangsweise Verletzung der vierten notwendigen Bedingung vermeiden: Deadlocks aufgrund der Locking-Schnittstellen können nicht entstehen, wenn Prozesse nicht zirkulär aufeinander warten. Dies schließt keine Deadlocks, die aus anderen Gründen entstehen sowie andere Verletzungen gewünschter Eigenschaften ein.

Eine Lösung zur Vermeidung zirkulärer Warteschlangen besteht darin eine totale Ordnung einzuführen. Ein Prozess darf Locks nur in einer bestimmten Reihenfolge anfordern. Dies führt indirekt dazu, dass keine Zyklen entstehen können. Die Reihenfolge kann entweder statisch definiert werden, so dass man sie für jeden Thread einzeln prüfen kann. Alternativ kann ein Sicherheitsautomat sich für ein nichtdeterministisch gewähltes Paar von Locks merken, in welcher Reihenfolge sie verwendet wurden. Beide Methoden haben unterschiedliche Stärken und Schwächen. Abbildung 4.5 gibt ein Beispiel für die Prüfung der Ordnung in Zusammenhang mit PS.

Statische Lock Ordnung

+ Jeder Thread kann unabhängig voneinander geprüft werden.

- Eine Software muss gegebenenfalls umstrukturiert werden um die Ordnung nicht zu verletzen (obwohl kein Fehler vorlag).

- Jede Laufzeit-Instanz eines Locks muss bekannt sein sobald die Ordnung festgelegt wird.

Dynamische Lock Ordnung

- Alle Threads die auf den Lockobjekten arbeiten müssen zusammen analysiert werden.

+ Nur benutzte Paare von Locks werden überprüft.
+ Die Lockordnung muss nicht bekannt sein.

Da die statische Lock Ordnung sich nicht für automatische Analyse tausender Treiber eignet, wird im Folgenden nur die zweite Regel umgesetzt. Die hier implementierte Regel wird 'Locking Order (LO)' genannt und ist wie folgend umgesetzt:

Bei der Initialisierung wird ein Lock nicht-deterministisch als Lock A oder Lock B gewählt. Bei jedem Aufruf `lock` wird gespeichert ob Lock A oder Lock B zuerst angefordert wurde. Falls niemals der Fall eintritt, dass die beiden Locks sowohl in der Ordnung A-B, als auch in der Reihenfolge B-A angefordert werden, ist sichergestellt, dass die Coffman-Bedingungen nicht erfüllt sind. Folglich kann keine zirkuläre Wartesituation – und somit ein Deadlock – auftreten.

4.2.7 Wettlaufsituation (Race-Conditions) (UA)

Eine Wettlaufsituation (*Race Condition*) tritt auf wenn zwei parallel laufende Threads auf eine gemeinsame Speicherlokation zugreifen. Zusätzlich muss zumindest einer der Threads schreibend zugreifen. Anstatt ein Analysewerkzeug speziell für das Aufspüren dieser Fehler zu schreiben, geben wir eine konservative Regel an, mit der indirekt Probleme dieser Art gefunden werden können. Der Grund für dieses Vorgehen ist die gewünschte Wiederverwendbarkeit bestehender Werkzeuge, die selbst keine Wettläufe aufspüren können. Die Regel wird 'ungeschützter Zugriff (UA)' genannt und deckt das folgende Szenario ab: *Auf eine dynamisch allozierte* `struct` *soll nicht zugegriffen werden ohne dass das, mit dieser Ressource assoziierte, Lock angefordert wurde.* Diese Anforderung wird weiter abgewandelt: *Nach einem Aufruf von* `unlock` *sind Zugriffe auf die Struktur nicht erlaubt solange nicht eine erneute Anforderung des Locks stattfindet.* Der Grund für diese umständliche Umwandlung ist die Einschränkung, dass SLIC und SLICx nicht die Möglichkeit bieten Zugriffe auf Speicherbereiche zu instrumentieren. Anstatt Zugriffe zu annotieren greifen wir auf die eingebauten Prüfungen zur Speichersicherheit zurück.

In Abbildung 4.6 ist ein kurzes Beispiel für eine Wettlaufsituation gegeben. Zugriff auf die Struktur `driver` wird durch das Lock `lock` exklusiv gehalten. Treiberquellcode ist auf der linken Seite abgebildet. In Zeile vier findet ein ungeschützter Zugriff statt der zu einem Wettlauf führen kann falls mehrere Threads gleichzeitig die Funktion aufrufen können. Die Anwendung der Regel auf der rechten Seite fügt die unnummerierten Regeln in den Treiber ein. Hierbei ist

```
// SLICx Regel um die Einhaltung
// der Lockordnung zu prüfen

spin_lock.exit{
  // Fall 1: Parameter ist das erste
  // Lock des Paars
  if(which_first_lock!=NULL &&
     // $1 ist der erste Parameter
     ($1 == which_first_lock)){
     if (first_lock_locked==0){
      if (second_lock_locked==1) {
        if (!order_set) {
          first_before_second = 0;
          order_set = 1;
        } else
          if (first_before_second) {
          abort "Lock order viol.";
          }
      }
     first_lock_locked = 1;
  } else {
      abort "Double acquire!";
  }
  // Fall 2: Parameter ist das zweite
  // Lock des Paars
  ...
}
```

```
// PS Szenario für die Prüfung
// der Lock Ordnung

// Interrupt Code
irq_return_t interrupt_handler()
{
       spin_lock(lock2);
       spin_lock(lock1);
       ...
       spin_unlock(lock1);
       spin_unlock(lock2);
}

// Treiberfunktion
void device_read() {
       // ...
       spin_lock(lock1);
       if (nondet_bool())
           interrupt_handler();
       spin_lock(lock2);
       if (nondet_bool())
           interrupt_handler();
       // ...
       spin_unlock(lock2);
       if (nondet_bool())
           interrupt_handler();
       spin_unlock(lock1);
       if (nondet_bool())
           interrupt_handler();
}
```

Abbildung 4.5: (l) Ein Auszug aus der Implementierung der Spezifikation der LO Regel. Die `which_[first/second]_lock` Zeiger speichern das nichtdeterministisch ausgewählte Paar von Locks. `order_set` gibt an, ob eine Reihenfolge bereits festgestellt werden konnte. (r) Dieses Beispiel enthält ein typisches Szenario in dem eine ISR und eine Operation in einem Treiber gemeinsam zwei Locks verwenden. Die ISR kehrt die Reihenfolge der Anforderung der Locks um. Im vorliegenden Fall kann dies zu einem Deadlock führen was durch Analyse der LO Regel mit CBMC aufgedeckt wird. Da die Regel nicht vollständig ist, muss die Warnung genauer analysiert werden um festzustellen, ob ein Deadlock oder nur eine Verletzung der Ordnung vorliegt.

```
// Vorherige Annotationen entfernt
1: spin_lock(&lock);
2: driver->request_nr++;
3: spin_unlock(&lock);
// nondet_bool() implementiert '*'
  if (which_lock==&lock) {
    if (nondet_bool()) {
         object_destroyed = 1;
         free(driver);
    }
  }
// Ungültiger Zugriff:
4: driver->request_nr++;
5: spin_lock(&lock);
  if (which_lock==&lock) {
     if (object_destroyed) assume(0);
  }
6: driver->request_nr++;
7: spin_unlock(&lock);
// Weitere Annotationen entfernt
```

```
spin_lock.entry {
  if (which_lock==$1) {
    if (object_destroyed==1)
       // Beende Ausführungen
       assume(0);
  }
}

spin_unlock.exit {
  if(which_lock==$1) {
    if(*) {
      object_destroyed = 1;
      kfree(which_object);
    }
  }
}
```

Abbildung 4.6: (l) Ein Beispiel für das Aufspüren von Wettlaufsituationen. Der Quellcode ohne Nummerierung auf der linken Seite entspricht den durch die auf der rechten Seite eingeführten Annotationen.

- `which_lock` das betrachtete Lock.
- `$1` der Wert des ersten Parameters der Funktionen `lock` beziehungsweise `unlock`. Als Abkürzung der Darstellung setzen wir direkt `&lock` auf der rechten Seite ein.
- `assume(0);` beendet alle Ausführungen.

Durch Anwendung der Instrumentierung kann CBMC die Zugriffsverletzung in Zeile 4 detektieren. Hierzu wird nichtdeterministisch die Funktion `free` auf dem Objekt `driver` aufgerufen. Letztere Funktion ist in CBMC so interpretiert dass nachfolgende Zugriffe (in der Literatur als *use-after-free* Fehler bekannt) gemeldet werden. Die obige Implementierung bedingt aber ebenso Falschmeldungen die entstehen da `driver` nach erneutem Aufruf von `spin_lock` (Zeile 5) wieder ein gültiges Objekt sein soll. Es ist leider nicht möglich die Auswirkungen von `free` bei erneutem `spin_lock` wieder zurückzunehmen. Statt der Wiederherstellung findet eine Filterung möglicher Ausführungspfade durch das Bit `object_destroyed` statt. Nach Zeile 5 wird geprüft ob das Objekt dealloziert wurde. Falls dies der Fall ist, wird der aktuelle Pfad durch `assume(0);` terminiert. So kann eine Programmausführung Zeile 6 erreichen in der zuvor `free` aufgerufen wurde. Eine zusätzliche Modellierungsverpflichtung besteht darin, dass das `driver` Objekt im Programm realloziert werden kann. Die Reallokation kann ungültige Zugriffe

Tabelle 4.2: Eine Klassifikation der Regeln. *S* steht für Schlüssigkeit während *V* Vollständigkeit abkürzt. Beide Begriffe betreffen jeweils nur einen bestimmten Teil der Schnittstellen.

Regel	Problem	Spezifikation	Impl. der Spezifikation	Sicherheit(Si) / Lebendigkeit (Le)
AL	Deadlock	S	S,V	Si
LO	Deadlock	V	S,V	Si
ML	Speicherlecks	S,V	S,V	Le
UA	Wettlaufs.	(V)	S,V	Si

maskieren. Dieses Problem lässt sich mittels zusätzlichen Annotationen der Allokationsfunktionen lösen.

Die Implementierung der Spezifikation ist vollständig und schlüssig. Die Spezifikation selbst ist allerdings nur vollständig bezüglich der Wettlaufsituationen.

Um die Regel umzusetzen wird zusätzlich die Information benötigt, welches Lock den Zugriff auf welche Objekte beschränkt. In Linux kann diese Information leicht inferiert werden: Locks beschützen meist die `struct` in der sie enthalten sind.

4.3 Ergebnisse

Wir geben zunächst ein Beispiel in der Avinux Fehler in einem Linux Treiber gefunden hat. Nachfolgend geben wir eine Übersicht über Laufzeiten und den Arbeitsaufwand für die Prüfung von Linux mit verschiedenen Regeln. Abschließend, in Unterabschnitt 4.3.3, diskutieren wir noch kurz Fragen der Codeabdeckung in der Verifikation.

4.3.1 Fehlerbeispiel

Wenn in Linux ein neues IDE Gerät erkannt wird, folgt der Aufruf der Funktion `do_probe` in der Datei `ide-probe.c`. Innerhalb der Funktion wird auf einigen Pfaden der Speicher auf den das Symbol `drive->id` verweist deallokiert. Die Ausrufsequenz ist hierbei:

```
do_probe(...) ->
try_to_identify(...) ->
```

```
actual_try_to_identify(...) ->
do_identify(...) ->
kfree(...)
```

Obwohl der Speicher möglicherweise deallociert wurde, wird er später in `do_probe` ohne Prüfung verwendet (Zeile 579):

```
strstr(drive->id->model, "E X A B Y T E N E S T")
```

Dieser Fehler wurde bei der Suche nach Fehlern der Verwendung von `kfree` entdeckt, das heißt CBMC meldete die Möglichkeit einer doppelten Deallokation. In echten Programmabläufen würde diese aber nicht eintreten aufgrund des obigen vorher auftretenden Fehlers.

4.3.2 Empirische Resultate

BLASTing Linux code [ML07] ist eine technische Fallstudie in der beschrieben wird welche Transformationen und Quellcodevereinfachungen notwendig sind um den Modellprüfer BLAST [HJMS03] anzuwenden. Die Testfälle, die in der Studie verwendet wurden, stellen eine minimale Testmenge dar an der wir unsere Werkzeugkette Avinux prüfen können. Hierbei ist unsere Vorgabe, dass die Prüfung der Fehler automatisch vorgenommen wird. Einzig die Modellierung einer minimalen Umgebung ist manuell erlaubt. Zwar müssen die SLICx Regeln ebenfalls manuell erstellt werden, jedoch muss dies nur einmal pro Version von Linux global erfolgen. Dieser Aufwand ist vernachlässigbar.

Die Laufzeit der Analyse ist durch die Kompilierung des kompletten Linux Betriebssystems dominiert (siehe Tabelle 4.4). Hierbei können noch Optimierungen vorgenommen werden um nur die in der Testmenge relevanten Dateien kompilieren zu müssen. Die Laufzeit von CBMC um einen einzelnen Treiber zu prüfen beträgt weniger als eine Minute. Im Gegensatz hierzu benötigte das Herausarbeiten der Fehlerbeispiele aus dem Linux Kern etliche Tage. Dies ist darauf zurückzuführen, dass die Subsysteme und deren Zusammenwirken wenig dokumentiert ist.

Von den acht Fehlern der BLAST Fallstudie im Bereich Speichersicherheit konnten direkt sechs gefunden werden. Von den Beispielen zu Deadlocks und Wettläufen konnten nur drei von acht gefunden werden. Dies ist darauf zurückzuführen, dass die globalen Zusammenhänge, die zu einem Fehler führen, nicht genau genug in einem Umgebungsmodell erfasst sind.

Neben dem Nachvollziehen alter Fallstudien untersuchten wir alle Linuxtreiber mit einigen neuen Regeln. Hierbei konnten weitere Fehler gefunden werden. So tritt in der Datei `drivers/char/tpm/tpm.c` ein möglicherweise ungültiger Zugriff auf:

Tabelle 4.3: Ergebnisse der ausgeführten Experimente.

Beschreibung	Linux Version	Ergebnis
Analyse der Speicherverletzungsfehler (aus [ML07])	2.6.11-14	6 von 8 Fehlern wurden mit minimalem Aufwand erkannt.
Analyse der Deadlock- und Wettlauf-bezogenen Fehler (aus [ML07])	2.6.13-15	3 von 8 Fehlern wurden gefunden.
Analyse aller Dateien auf das Vorkommen eines doppelten Aufrufs von `free`	2.6.19	1 Fehler wurde gefunden in `ide-probe.c`.
Effektivitätstest der DEC Komponente bei Analyse von `kmem_cache_free`	2.6.20.4	Falschmeldungen um 50% reduziert.

```
line 1130: devname = (char *)__SLIC_kmalloc(7U, 208U);
line 1131: scnprintf(devname, 7UL, "%s%d", "tpm", chip->dev_num);
```

Unter Berücksichtigung der Tatsache, dass die Speicherallokation mittels der Funktion `kmalloc` fehlschlagen kann, findet in Zeile 1131 ein ungültiger Zugriff statt. Wäre eine Überprüfung auf ungültige Eingaben in der Funktion `scnprintf` integriert, hätte der Fehler vermieden werden können (Quelle: Linux Version 2.6.19).

Zusätzlich zu den obigen Beispielen konnten folgende Fehler gefunden werden (Quelle: Linux Version 2.6.20.4):

- `drivers/char/rtc.c`: Die Benutzung eines Locks ohne vorherige Initialisierung.
- `/drivers/net/wan/sbni.c`: Ein Deadlock, der aufgrund eines Wettlaufs zwischen einer ISR und der Initialisierungsfunktion eines Moduls stattfinden kann.
- `fs/xfs/xfs_da_btree.c`: Eine mögliche Benutzung eines ungültigen Zeigers.
- `drivers/infiniband/mad.c`: Eine Verletzung der Schnittstellenprotokolle der Funktion `kmem_cache_free` .

Zusätzliche Details zu diesen Fehlern werden auf der Avinux Webseite zur Verfügung gestellt [5].

Tabelle 4.4 enthält Laufzeiten für die Phasen der Verifikation. Tabelle 4.3 fasst die Resultate der Experimente zusammen. Im Folgenden erweitern wir die Analyse der Ergebnisse durch eine Betrachtung des größten Problems der Treiberverifikation: die manuelle Erstellung von Umgebungsmodellen. Abbildung 4.7 gibt einen Überblick über die schließlich eingesetzte Toolchain. Die Komponente DEC wird im Folgenden Abschnitt 4.4 genauer diskutiert.

[5] www-sr.uni-tuebingen.de/~post/avinux

Tabelle 4.4: Liste der Laufzeiten aller Phasen der Verifikation (Linux Version 2.6.20.4, `allyesconfig` Konfiguration; auf einem Pentium 4 mit 3 Ghz, 1 GB RAM). Für eine gegebene Regel und ein Model des Betriebssystems ist die Laufzeit durch CBMC dominiert.

Phase	Laufzeit	Kommentar
Konfiguration	1 min	Aktivierung alle möglichen Subsysteme.
Erstellung der SLICx Regel	1 m / mehrere Tage	Die Modellierung von Subsystemen und Parallelität dominiert dies Phase.
Instrumentierung	1 St.	Annotation aller Treiber.
Instrumentierung	+2 St.	Annotation aller Quellen (Netzwerk, Dateisysteme,...).
Kompilierung mittels CIL	1 St. - 5 St.	Hohe Abhängigkeit von Konfiguration und Architektur
Zusammenfügen von abhängigen Dateien zu Subsystemen.	1 m pro Operation	Die Angabe welche Dateien zusammengehören muss manuell gegeben werden.
CLEANC	30 min	Entfernen von syntaktischen Problemen.
Modellierung der Umgebung	1 m -/ mehrere Tage pro Model	Abhängig von der Komplexität des Subsystems und der Dokumentation.
DEC	2 St.	Für alle Dateien
Laufzeit von CBMC	6 St. / mehrere Tage	Prüfung einer Regel in allen Dateien: z.B. 36 Stunden für die `kmem_cache_free` Regel.

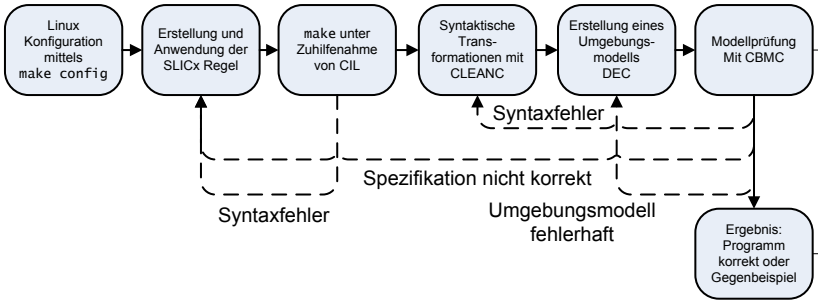


Abbildung 4.7: Die Erstellung eines verifizierbaren Programms erfordert eine Kette von Transformationen. Die SLICx Regel und ein Model der Umgebung in der der Treiber läuft müssen manuell erstellt werden. Alle anderen Schritte werden automatisch ausgeführt.

4.3.3 Abdeckung von Eintrittspunkten

Bei einem offenen System wie einem Gerätetreiber in Linux fehlt es an Information wie der Treiber vom Rest des Betriebssystems genutzt wird. CBMC akzeptiert nur die Eingabe welches die Startfunktion eines Programms ist. In einem Gerätetreiber kann jede Funktion, die von außen ausgerufen werden kann, als potentielle Startfunktion betrachtet werden. Um eine Garantie zu geben, dass ein Treiber korrekt ist, reicht es jedoch jede Funktion die direkt, oder indirekt, eine annotierte Funktion aufrufen kann als mögliche Startfunktion zu betrachten. Diese Effizienzsteigerung reduziert die Anzahl der notwendigen Aufrufe von CBMC erheblich.

Um die obige Menge zu berechnen, erstellen wir einen statischen Aufrufsgraphen über alle Funktionen im Modul. Ausgehend von allen annotierten Funktionen erweitern wir die Menge um alle Aufrufer der aktuell in der Menge enthaltenen Funktionen. Der transitive Abschluss auf dem Aufrufsgraphen liefert dann die gewünschte Menge, der Funktionen die zu einem Fehler führen können.

Als heuristische Einschränkung kann man zum Zweck der Fehlersuche nur die Wurzeln im Ausrufsgraphen, die auch in der transitiven Hülle enthalten sind, als Startfunktionen verwenden. Dies ist mit der Linux Konvention zu begründen, dass Eintrittspunkte in ein Modul sich nicht gegenseitig aufrufen. Somit erhält man über die Wurzeln des statischen Aufrufsbaums eine gute Approximation der Benutzung des Treibers durch das Betriebssystem.

Zu beachten ist, dass zwar im obigen Schritt die Einstiegspunkte berechnet werden, jedoch fehlt die Information wie die Einstiegspunkte zeitlich zu verknüpfen sind. In einem typischen Lebenszyklus wie in Abbildung 4.2 können Aufrufe einer Funktion `close` nur nach Aufrufen von korrespondierenden `open` Funktionen stattfinden. Solche Beschränkungen der Reihenfolge von Funktionen, die Einstiegspunkte sind, werden im nächsten Abschnitt diskutiert.

4.4 Umgebungsmodelle für Gerätetreiber

Modulare Verifikation ohne Umgebungsmodelle kann zu unerwünschten Warnungen (*false positives*) und zu fehlenden Warnungen (*false negatives*) führen. Ein einfaches Beispiel für eine inkorrekte Fehlermeldung wird durch folgenden Code induziert. CBMC benötigt als Eingabe den Startpunkt aller Programmausführungen, im Beispiel ist dies die Funktion `teile`:

```
// Notwendig: divisor != 0
int teile(int dividend , int divisor) {
```

```
   return dividend / divisor;
}

// Einzige Stelle an der "teile" aufgerufen wird
void main() {
   ...
   teile(20,5);
   ...
}
```

Wird `teile` ohne Angabe von Umgebungsinformation analysiert, so liefert CBMC eine Warnung über eine mögliche Division durch null. Dieser Fall kann aber nicht eintreten, da `teile` an nur einer Stelle und mit einem Divisior, der ungleich null ist, aufgerufen wird. Folgende Möglichkeiten bestehen um die Einschränkung in der Analyse explizit zu machen:

- Die Funktion `main` wird in die Analyse einbezogen.
- Eine abstrakte Version der Funktion `main`, welche alle Aufrufe von `teile` enthält, wird zugefügt.
- Die Anforderung, dass der Divisor ungleich null sein muss, wird durch Einfügen eines `assume` Befehls explizit gemacht.

Das übliche Vorgehen des letzten Vorschlags hat die Form einer *Assume-Guarantee*[Pnu85] Analyse. Zunächst beweist man, dass die obigen Annahmen wirklich durch die Umgebung – im Beispiel die Funktion `main` – erfüllt sind. Ist dies der Fall, bleibt als zweite Beweisverpflichtung noch der Nachweis, dass `teile` unter der Assume Annahme, also dem Umgebungsmodell, korrekt ist. Im Allgemeinen ist es schwieriger nachzuweisen, dass das Umgebungsmodell korrekt ist. Dies gilt insbesondere für Linux, da die Umgebung eines jeden Treibers Millionen Zeilen von Quellcode betrifft.

Neben der Möglichkeit von false positives kann fehlende Information über den Kontext auch zu einer unvollständigen Abdeckung, das heißt zu false negatives, führen:

```
// wird nicht direkt aufgerufen
int extern_teile(int dividend , int divisor) {
   return dividend / divisor;
}

void main() {
   ...
```

```
    export_function(&extern_teile)
    ...
}
```

Die Funktion `main` exportiert einen Zeiger auf die Funktion `extern_teile` in eine unbekannte Umgebung: Im Rahmen der Analyse wird die Funktion `extern_teile` nicht direkt aufgerufen, kann also auch nicht zu einer Division durch null führen. Wird die Funktion `extern_teile` aus der Umgebung heraus aufgerufen, kann ein Fehler auftreten, obwohl die modulare Analyse ihn nicht entdeckt.

In diesem Abschnitt werden nun einige Methoden vorgestellt wie die Zahl der durch fehlende Umgebungsinformation hervorgerufenen false negatives und false positives reduziert werden kann. Bemerkenswert ist, dass dieselbe Problematik auch bei herkömmlichem Testen vorliegt.

Im Folgenden werden drei Ansätze zur Modellierung behandelt, die speziell auf Linux Module zugeschnitten, und experimentell auf Effizienz und Machbarkeit getestet worden sind.

Minimales Model

Ein minimales Betriebssystem-Modell (minBSM) für Linux Module geht von dem typischen Lebenszyklus eines Moduls aus: In Linux sind fast alle Module im laufenden Betrieb ladbar und wieder entladbar. Im Besonderen trifft dies für Treiber einer Hardware zu, von der angenommen wird, dass sie im Betrieb ausgewechselt wird. Im Bereich der Großrechner sind auch Prozessoren und Festplatten im laufenden Betrieb auswechselbar.

Das minBSM umfasst zwei Stufen:

- Die Initialisierung des Treibers.
- Die Vorbereitung des Entladens.

Auch wenn dieses Modell sehr einfach erscheint können bereits Fehler bei dieser Modellierung gemacht werden. So ist eine Entladung des Treibers nur dann gestattet, falls die Initialisierung erfolgreich war. Bezüglich der Fehlerausbeute in den Experimenten ist das minBSM hervorragend geeignet um schnell viele Fehler aufzuspüren. Der Grund hierfür ist in der komplexen Struktur der Initialisierung zu sehen. Eine Kaskade von kleinen Schritten muss nach und nach ausgeführt werden und falls einer dieser Schritte fehlschlägt müssen alle Schritte – zum Beispiel die Allokation von Speicher – korrekt rückgängig gemacht werden. Gängige Fehler die durch Avinux aufgespürt werden, sind Wettlaufsituationen (*Race-Conditions)* zwischen dem

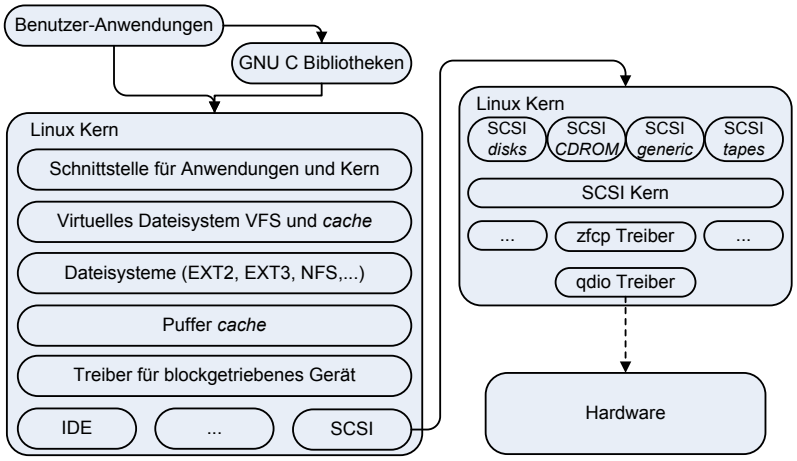


Abbildung 4.8: Der zfcp Treiber kommuniziert vor allem mit den Kern-Komponenten des SCSI Subsystems und dem QDIO Treiber, der Befehle direkt an die Hardware weitergibt. Das Umgebungsmodell muss zwar das komplette Betriebssystem berücksichtigen, jedoch reicht in diesem Fall die Modellierung von Aufrufen des SCSI Kerns sowie von Rückrufen aus dem QDIO Treiber.

Treiberthread und der installierten ISR. Alle Schnittstellen, die korrekte Abfolgen von Operationen verlangen (beispielsweise `lock / unlock`) schlagen häufig fehl, da ein Fehler bei der Initialisierung zu nicht alloziertem Speicher führen kann, der beim Entladen ungültigerweise freigegeben wird.

4.4.1 Manuelles Modell

Ein manuelles Betriebssystemmodell (manBSM) wird auf Basis einer genauen Analyse des Quellcodes des Moduls und aller aufrufenden Module erstellt. Zusätzlich können natürlichsprachliche Informationen aus der Dokumentation hinzugezogen werden. Der Vorteil einer solchen Detailanalyse ist die höhere Präzision die das Model erreichen kann. Der Nachteil liegt im erhöhten Aufwand. Für das Betriebssystem Linux ist es zur Zeit nicht plausibel Modelle für *alle* Subsysteme manuell zu erstellen.

Getestet wird die Erstellung anhand des zfcp Subsystems welches eine Protokollschicht zur Ansteuerung von – über Fibre-Channel angesteuerten Speichersystemen – im Bereich Großrechner implementiert. Abbildung 4.8 illustriert die Einbettung in die Protokollschichten.

Der Arbeitsaufwand zur Erstellung des Modells betrug mehrere Monate. Es zeigt sich, dass mit zunehmender Komplexität des Umgebungsmodells die Laufzeit von CBMC schnell ansteigt. Dies ist damit zu begründen, dass CBMC Programmanteile, die nicht im Rahmen eines manBSMs ausgeführt

werden können, aus Optimierungsgründen nicht kodiert. Ein manBSM welches mehr Anteile des zfcp antreibt, verursacht somit auch einen größeren Anteil an Zeilen, die in das BMC Problem kodiert werden müssen.

Abbildung 4.9 zeigt die normale zeitliche Abfolge von Interaktionen zwischen verschiedenen Komponenten in Linux. Zusätzlich ist eine Darstellung der Abfolge von Interaktionen im Umgebungsmodell gegenübergestellt.

Das manBSM zeigt eine erhöhte Abdeckung gegenüber dem minBSM. Zur Verifikation im Sinne einer vollständigen Abdeckung möglicher Programmabläufe ist die Verwendung eines manBSM zweckmäßig. Zur automatischen Verifikation einer kompletten Linuxversion ist die Methode jedoch ungeeignet da der Aufwand eine Beschränkung auf wenige Module verlangt.

4.4.2 Schablonenmodelle

Witkowski et al. [WBWK07] präsentieren eine weitere Art von Umgebungsmodell welche einen Kompromiss zwischen den obigen Extremen darstellt. Ihr Werkzeug, DDVerify, stellt Schablonen für gängige *Klassen* von Treibern zur Verfügung. Ein Treiber, der eine Schnittstelle implementiert für die eine Schablone existiert, kann somit über eine ausgefüllte Schablone angetrieben werden.

In Linux gibt es verschiedene Klassen und Kategorien von Treibern. Die gängigste Unterscheidung besteht zwischen block- und zeichengetriebenen Treibern: Festplatten sind ein Beispiel für blockgetriebene Geräte – Daten werden nicht byteweise, sondern in großen Sammlungen an das Gerät übertragen. Ein zeichengetriebenes Gerät ist beispielsweise der Tastaturtreiber. Hier findet die Kommunikation auf der Ebene einzelner Zeichen statt. Beide Klassen haben eine Schnittstelle, die von allen Treibern der Klasse implementiert wird.

Schablonen bieten hohe Automatisierung bei größerer Abdeckung als es bei minBSMs der Fall ist. Dennoch gibt es Begrenzungen die aus der großen kombinatorischen Anzahl von Geräteklassen resultiert. Zum Einen decken die von Witkowski et al. bereitgestellten Schablonen [WBWK07] nur gängige Geräteklassen ab und zum Anderen ist es möglich, dass ein Treiber mehrere Klassen implementiert. Hierbei kann ein Gerät gleiche Funktionalität durch verschiedene Schnittstellen bereitstellen. Eine vollständige Abdeckung verlangt also alle Schablonen, die für einen Treiber zutreffen zu kombinieren. Für eine möglichst breite automatische Fehlersuche stellt jedoch das DDVerify Werkzeug die bisher aussichtsreichste Möglichkeit zur Erstellung von Umgebungsmodellen dar.

Die obigen drei Ansätze betreffen Umgebungsmodelle, die sowohl Aufrufe

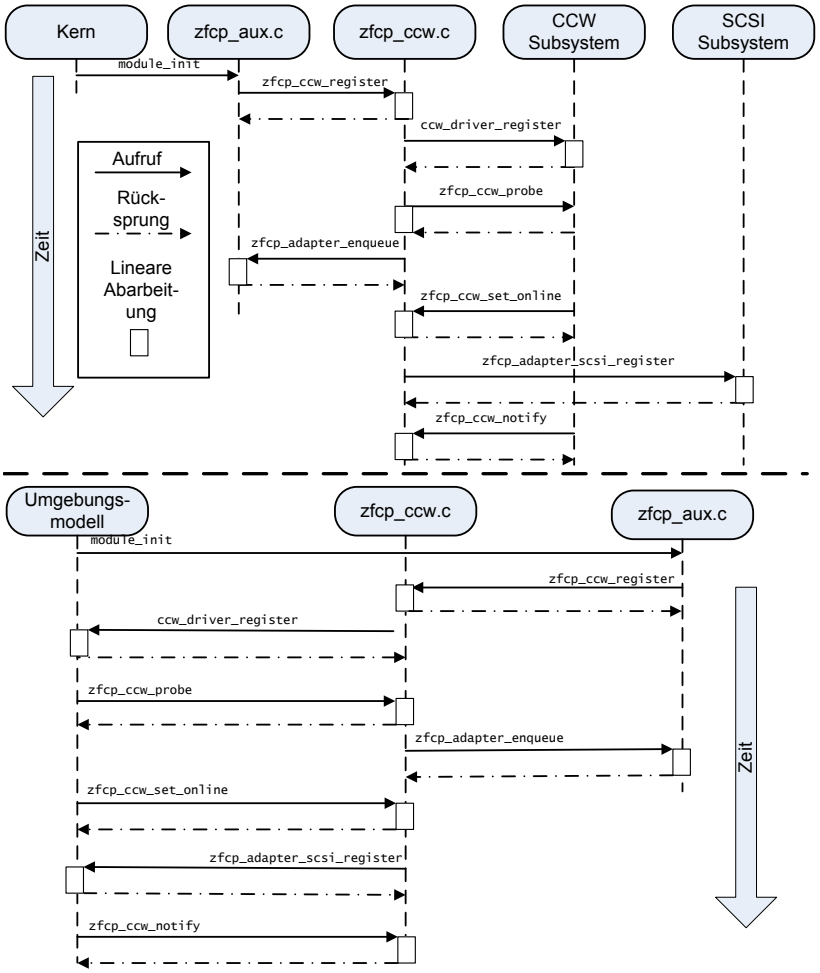


Abbildung 4.9: Im oberen Teil dieser Abbildung ist die Interaktion zwischen verschiedenen Linux Subsystemen dargestellt. Bei modularer Analyse – wie im unteren Teil dargestellt – sind Kern, das CCW Subsystem und das SCSI Subsystem ausgeblendet. Dies führt dazu, dass alle Interaktionen mit diesen durch das Umgebungsmodell simuliert werden müssen. Die Darstellung beinhaltet nur die Initialisierung des Treibers und eines Gerätes, nicht aber Operationen auf einem aktiven Gerät oder der Entladevorgang. Nebenläufige Initialisierung mehrerer Geräte ist ebenso nicht berücksichtigt.

als auch Daten für eine Exploration möglicher Abläufe in einem Treiber bereitstellen. Die reine Modellierung von Eingabedaten wird durch die Avinux Komponente DEC behandelt.

4.4.3 Modellierung komplexer Eingabedaten

Offene Systeme wie Treiber stellen den Anwender von CBMC vor das Problem, dass jede Funktion, die Startpunkt einer Ausführung ist und einen Parameter per Referenz übergibt, automatisch als fehlerhaft bestimmt wird:

```
   void open(struct device * device) {
1:   device->name = ...
     ...
   }
```

In Zeile 1 wird möglicherweise ein ungültiger Zeiger verwendet. CBMC wird diesen Fehler immer zuerst finden und dann die Analyse abbrechen. In fast allen Fällen ist aber sichergestellt, dass der übergebene Zeiger in tatsächlichen Aufrufen von `open` gültig ist. In solchen Fällen interessiert das Ergebnis unter der Annahme, dass die Referenz wirklich korrekt initialisiert wurde. CBMC selbst stellt keine Möglichkeit bereit die Verifikation unter dieser Annahme auszuführen. Als beste Annäherung ist die Option `-no-pointer-check` zu nennen, welche global alle Prüfungen ungültiger Zugriffe ausschaltet.

Dies ist aber nicht zielführend: Zum Einen sollen alle anderen Zugriffe in dem Modul nicht deaktiviert werden. Zum Anderen besteht das Problem, dass CBMC Zugriffe auf nicht-initialisierte Objekte nicht so abbildet, dass ein gültiges Objekt vorliegt: Wird über einen Zeiger ein ungültiges Objekt beschrieben, kann in der Modellierung von CBMC diese Information in späteren Zeitpunkten im Programmfluss nicht immer abgerufen werden. Dieses Verhalten ist eine Auswirkung der Tatsache, dass Zugriffe auf möglicherweise nicht-initialisierte Objekte undefiniertes Verhalten herbeiführen. Zur Lösung dieses Problems bleibt nur die Sicherstellung, dass in der Tat alle Schnittstellenzeiger *immer* auf gültige Objekte zeigen.

Dieses Ziel wird erreicht indem eine zusätzliche Komponente (DEC) gültige Ziele für Schnittstellenzeiger erstellt. Im vorliegenden Fall wird eine Funktion erstellt, die `open` aufruft und eine Referenz auf ein gültiges `struct device` Objekt übergibt:

```
   void main() {
      struct device tmp;
      // tmp ist ein beliebiges Objekt, das ein gültiges Ziel
```

```
    // für den Schnittstellenzeiger bietet.
    open(&tmp);
    ...
}
```

Das genaue Verfahren ist in [PK06] beschrieben. Wir prüfen im Folgenden die Effektivität dieser Komponente.

4.4.4 Datenumgebungen für Schnittstellen: DEC

Datenumgebungen in unserem Sinne kodieren die Annahme, dass alle Zeiger, die Bestandteil einer Schnittstelle sind, auf gültige (aber ansonsten beliebige) Objekte zeigen. Zur Fehlersuche ist diese Annahme sehr effektiv, da sie eine im Linux Betriebssystem fast immer gültige Vereinbarung zwischen Aufrufern und Aufgerufenden umfasst.

Um die Effektivität dieser Umgebungsmodelle zu evaluieren prüfen wir folgende Eigenschaft für alle Treiber: *Für jeden Aufruf von* `kmem_cache_free` *ist der erste Parameter nicht* `NULL`. `kmem_cache_free` ist eine Funktion zur Deallokation eines Objekts welches aus einer Sammlung (*cache*) von häufig verwendeten Speicherbereichen alloziert wurde.

CBMC wurde auf 589 Einsprungspunkten aufgerufen, die direkt, meist aber indirekt, `kmem_cache_free` aufrufen. Die Prüfung umfasst nicht die generischen Eigenschaften wie beispielsweise die Speichersicherheit bei der Verwendung von Zeigern. Ohne DEC meldet CBMC, dass für 194 Funktionen ein Fehler vorliegen kann. Unter der obigen Annahme, die durch die DEC Komponente in C kodiert wurde, wurden nur noch 81 Warnungen erzeugt. Zwar benötigt die manuelle Analyse der 81 Warnungen einige Tage, dennoch ist der Aufwand signifikant reduziert worden.

Durch DEC findet die Erstellung des Datenmodells nur bis zu einer bestimmten Tiefe statt. Ähnlich wie bei der Einschränkung des BMC definieren wir eine Abrolltiefe für verschachtelte, oder rekursive, Datenstrukturen. Für das obige Experiment war diese Tiefe zwei, das heißt beispielsweise Listen wurden nur bis zur Länge zwei initialisiert. Das Datenmodell wurde nur für Schnittstellenparameter, nicht aber für globale Zeiger erstellt.

4.5 Zusammenfassung des Kapitels

In Kapitel 4 wurde ein großes System mit mehreren Million Zeilen Quellcode analysiert. Die Überprüfung fand hierbei modular auf der Basis einzelner Treiber statt. Die Laufzeit von CBMC [CKL04] ist aufgrund des hohen

restlichen Aufwands (durch die Modularität und syntakische Probleme) zu vernachlässigen.

Modularität führt dazu, dass Informationen über die Benutzung eines Treibers durch das restliche System nicht vorliegen. Unbekannte Eingabeparameter können zu unerwünschten Warnungen führen. Dieses Problem muss durch Modellierung der Umgebung, in der ein Treiber arbeitet, gelöst werden.

Drei mögliche Ansätze für Umgebungsmodelle wurden vorgestellt. Zur Analyse eines kompletten Betriebssystems eignet sich jedoch nur ein minimales Modell, dass für alle Treiber einsetztbar ist, nicht aber alle Nutzungsweisen jedes Treibers abdeckt. Als Folge können nicht alle möglichen Ausführungen berücksichtigt werden. Umgebungsmodelle führen jedoch zu einer sehr stark verminderten Anzahl von Warnungen, die wiederum echte Fehler maskieren können.

SLICx ist eine neue Spezifikationssprache, die speziell auf die Formulierung von Linux Schnittstelleneigenschaften zugeschnitten ist. Mittels SLICx können eine Reihe wichtiger Probleme im Kontext von Betriebssystemen formuliert und somit auch mittels CBMC geprüft werden. SLICx erweitert hierbei die Sprache SLIC [BR01] um die Facette der Modifikation des zu analysierenden Treibers: Eine SLICx Regel kann beispielsweise Aufrufe von Funktionen enthalten um Unterbrechungen von Interrupt Funktionen zu simulieren.

Nach umfassender Behandlung semantischer und syntaktischer Abweichung des Linux C Dialekts, konnten eine Reihe nicht trivialer Fehler gefunden werden. Wir folgern, dass der Einsatz von CBMC zur Qualität von Linux beigetragen hat. Die Fehler beinhalten Probleme bei sequentieller und paralleler Ausführung in Modulen, die seit Jahren in jeder Version von Linux in viel genutzten Komponenten bestehen.

Aus der Linux-Fallstudie lernen wir, dass SBMC zwar anwendbar für große offene Systeme ist, jedoch keine vollständige Abdeckung von allen Ausführungen erreicht werden kann. Um das Problem zu lösen, stellen wir im nächsten Kapitel ein neues Verfahren vor, dass SBMC mit abstrakter Interpretation kombiniert. Die Kombination beider Verfahren ermöglicht die vollständige Abdeckung aller Ausführungen mit geringer Rate an false positives. Getestet wird das neue Verfahren anhand eines Programms aus dem Bereich der Software des Automobilbereichs. Die Software ist mit einigen hunderttausend Zeilen und Schnittstellen zu anderen Komponenten ähnlich zu Linux, jedoch um einige Größenordnungen kleiner.

5 Kombination mit abstrakter Interpretation

In den vorangegangenen Abschnitten wurde evaluiert inwieweit Software Bounded Model Checking durch die endliche Schranke in der Anwendbarkeit limitiert wird. Für große Systeme wie Linux Subsysteme oder große Treiber ist es ersichtlich, dass der Durchmesser nicht ausreichend groß sein kann um Verifikation auf Modulebene zu realisieren. Im kleinen Umfang, das heißt bei kleinen Applikationen oder auch bei der modularen Verifikation von Funktionen oder kleinen zusammenwirkenden Funktionsmengen ist SBMC eine aussagekräftige Methode. Zusammengefasst ist die Schwäche von SBMC die Unschlüssigkeit bei Korrektheitsaussagen.

Andere Ansätze zur Verifikation haben ein inverses Profil. Die Technik der abstrakten Interpretation wird häufig [Pol08, CCF+05] in Form von Werkzeugen zur Verifikation umgesetzt, die schlüssige Aussagen liefern. Die Aussagen betreffen zumeist Sicherheitseigenschaften und aufgrund der theoretischen Grenzen kann im Allgemeinen die Vollständigkeit der Analyseergebnisse nicht gewährleistet werden. Es liegt also nahe die Stärken beider Methoden zu kombinieren um die Menge von inkorrekten Ergebnissen soweit wie möglich zu reduzieren.

Im Folgenden wird erläutert wie sich schlüssige abstrakte Interpretation mit SBMC kombinieren lässt um ein schlüssiges Analyseergebnis mit der Vollständigkeit von SBMC zu erzielen. Zugelassen werden hierbei Falschaussagen, die aufgrund von Laufzeitproblemen des SBMC nicht geprüft werden können.

5.1 Einführung

Verifikationstechnologie, die in industriellen Software Projekten eingesetzt werden soll, wird aufgrund einer Balance zwischen Laufzeit, Grad der Automatisierung und erforderlicher Präzision der Analyse gewählt. Die Robert Bosch GmbH, der weltweit größte Zulieferer von Automobilkomponenten, setzt unter anderem abstrakte Interpretation in Form des Produktes Polyspace [Pol08] bei der Entwicklung von Software für Fahrerassistenzsysteme ein. Dies ist zur Zeit nur eine zusätzliche Qualitätsmaßnahme, die das traditionelle Testen komplementiert.

Obwohl Polyspace das Gesamtsystem mit 300.000 Zeilen Quellcode analysieren kann, findet sich doch eine zu hohe Rate von Falschmeldungen. Falschmeldungen sind sie in dem Sinne, dass Polyspace nicht beweisen kann, dass eine bestimmte Zeile im Quellcode unter allen Umständen sicher ist, obwohl dies der Fall ist. Die Laufzeit für das Gesamtprojekt beträgt zwei Wochen unter diesen Präzisionsparametern, das heißt die Präzision kann nicht weiter erhöht werden ohne das Verfahren zu verbessern. Für die Zukunft ist zu erwarten, dass die Komplexität der untersuchten Software weiter ansteigt und dies wird entweder auf Kosten der Laufzeit oder der Präzision geschehen.

Andere Projekte wie etwa Orion [DN05] haben einen Ausweg aufgezeigt. Anstatt eine Technik weiter zu verbessern, schlagen Dams und Namjoshihave eine Kombination von einer globalen Datenflussanalyse und einer anderen Techniken höherer Präzision vor. In einem ersten Schritt generiert Orion eine Menge möglicher Fehlerkandidaten. In einem zweiten Schritt werden die Fehlerkandidaten genauer analysiert. Der erste Schritt hilft hierbei unnötige Bestandteile des Systems zu ignorieren (*slicing*).

In diesem Kapitel variieren wir die Orion Idee, indem wir die globale Datenflussanalyse des ersten Schritts durch eine mächtigere Technik, nämlich eine Implementierung abstrakter Interpretation, ersetzen. Diese wird von dem Werkzeug Polyspace [Pol08] bereitgestellt. Anstatt nun nur einen zweiten Analyselauf vorzunehmen, wenden wir BMC an um die verbleibenden Warnungen einzeln und so lokal wie möglich zu verfeinern. Dams und Namjoshihave verwenden für die zweite Phase die Beweiser CVC und Simplify [DNS05]. Wir verwenden den Software Bounded Model Checker CBMC [CKL04]. Eine der grundlegenden Hypothesen dieses Kapitels ist, dass viele der Warnungen, die durch Polyspace nicht widerlegt werden können, durch die vorgenommene Abstraktion zustande kommen. Wir prüfen diese Annahme zusätzlich indem wir den abstraktionsfähigen Modellprüfer SATABS [CKSY05] zusätzlich einsetzen. SATABS ist ein Software Model Checker, der die CEGAR Technik [CL00] verwendet um die größt mögliche Abstraktion zu ermitteln.

Die Anwendung von CBMC im zweiten Schritt findet auf zwei verschiedene Weisen statt: Zunächst starten wir CBMC direkt auf den Teilen des Quellcodes auf denen Polyspace mögliche Fehler meldet. Diese automatisierbare Anwendung nennen wir *Phase A*. Das Ergebnis dieser Phase ist eine Angabe zu jeder Warnung, ob diese als Falschmeldung identifiziert werden konnte. Für alle Fälle, die nicht definitiv geklärt werden können, fahren wir in einer zweiten Phase B mit der Analyse fort. Im Gegensatz zu dem automatischen Vorgehen, erlauben wir zusätzlich die Anwendung manueller Hilfen wie die Spezifikation von lokalen Vorbedingungen und Invarianten.

Die Studie, die in diesem Kapitel vorgestellt wird, hat als Ziel nicht den Vergleich zwischen den Techniken BMC und AI. Statt dessen wird nur ge-

prüft, ob BMC die Ergebnisse der abstrakten Interpretation verfeinern kann. Die Fallstudie basiert auf einer Menge von 77 Warnungen die von Polyspace ausgegeben wurden. Als Vorgriff auf die Ergebnisse sei gesagt, dass wir in Phase A (Abschnitt 5.3.1) die Warnungen um 23% reduzieren können. Mittels manueller Unterstützung kann schließlich die Gesamtzahl verbleibender Warnungen um mehr als 70% reduziert werden.

Da das Problem nicht-triviale Eigenschaften über ein Programm zu beweisen als unentscheidbar bekannt ist, sind Approximationen notwendig. Diese können ausreichen bestimmte Eigenschaften für eine eingeschränkte Menge von Programmen zu beweisen. Approximation kann zu falschen Fehlermeldungen (*false positives)* oder zu falschen Korrektheitsaussagen (*false negatives*) führen. Es ist unvermeidbar eines der Fehlergebnisse zuzulassen ohne die Menge der Programme unrealistisch einzuschränken. Als Folge wurden viele verschiedene Techniken entwickelt die angepasst und erfolgreich angewendet wurden. Die Bandbreite reicht von einfachen Datenflussanalysen, wie sie in Compilern eingesetzt werden, bis hin zu der semi-automatischen Anwendung von Beweisern der Theorie der Dynamischen Logiken erster Ordnung [HJL+06]. Es muss immer eine Balance gefunden werden zwischen Anzahl von Fehlmeldungen, Automatisierbarkeit und verfügbaren Ressourcen.

Für die Analyse von sicherheitskritischen Systemen ist es zweckmäßiger false negatives zu vermeiden und false positives zuzulassen. Trotzdem verursachen die resultierenden falschen Warnungen einen beträchtlichen Aufwand weil sie einer manuellen Inspektion bedürfen um Fehler auszuschließen. Ziel muss es deshalb sein die Zahl der false positives gleichzeitig so weit wie möglich zu reduzieren. Um die Kombination von BMC und AI detailliert zu motivieren, stellen wir noch einmal kurz die Techniken vor. Für eine ausführlichere Vorstellung sei auf die Literatur oder die Abschnitte 2.2.4, 2.2.6 und 2.2.7 verwiesen.

Abstrakte Interpretation (AI) Abstrakte Interpretation ist eine Technik, die unter anderem eingesetzt wird, um die Abwesenheit von Laufzeitfehlern zu beweisen. Hierzu gehören Pufferüberläufe, Zugriffe über ungültige Zeiger, Verletzung von Arraygrenzen sowie arithmetische Überläufe. Abstrakte Interpretation wurde erfolgreich auf viele verschiedene imperative Sprachen angewendet. Überlicherweise wird eine Überapproximation des Systems verwendet, das heißt es werden mehr Programmabläufe (*Traces*) in Erwägung gezogen, als tatsächlich im Programm vorkommen können. Die Überapproximation wird über die Verwendung abstrakter Domänen und die Verwendung von *widening* und *narrowing* Operatoren erreicht. Falls ein Programm Schleifen, Rekursion oder anderweitig wiedereintretenden Quellcode verwendet, wird eine Lösung für die Menge der Programmabläufe über eine Fixpunktiteration erreicht. Es ist zwar bekannt, dass Abstraktion auch in

```
1: void foo(unsigned int var) {
2:      var = var & 0x02;
3:      assert(var==0x02 || var==0);
4: }
```

Abbildung 5.1: Polyspace kann nicht beweisen, dass die Forderung aus Zeile drei immer erfüllt ist. CBMC modelliert alle Operation bitgenau und beweist die Fehlerfreiheit des Programms automatisch.

der Praxis falsche Warnungen verursacht, aber das Problem ist derzeit nicht gelöst (siehe [Riv05]).

Bounded Model Checking (BMC) Bounded Model Checking wurde von Biere et al. [BCCZ99] vorgeschlagen um Sicherheitseigenschaften von Hardwareschaltungen zu prüfen. Darauf aufbauend beschreiben Clarke et al. die Anwendung der Technik auf C Programme [CKL04].

Software Bounded Model Checking kann, wie bereits in Abschnitt 2.2.7 erläutert, Rekursion und Schleifen nur durch ein endliches Abrollen approximieren. Datentypen und arithmetische Operationen werden jedoch im Gegensatz zu abstrakte Interpretation genau modelliert. Dies bedeutet, dass SBMC die Semantik eines C Programms genau kodiert falls die verwendeten Schranken für das Abrollen groß genug sind. CBMC ist die Implementierung für SBMC, die für die Experimente dieses Abschnitts verwendet wird. CBMC kodiert alle möglichen Ausführungspfade eines Programms als Boolesche Formel. Letztere ist durch einen Erfüllbarkeitsprüfer für das SAT-Problem analysierbar.

Das endliche Abrollen führt dazu, dass keine Fixpunktiteration berechnet werden muss. Allerdings kann, falls die Schranke nicht ausreichen sollte, immer noch eine Iteration in der Schrankengröße nötig sein. Für die gemachten Experimente war jedoch der Durchmesser durch die in CBMC eingebauten Heuristiken berechenbar. SBMC wurde bereits erfolgreich auf mittelgroße bis große Softwaresysteme angewendet wie in den vorangegangen Kapiteln erläutert wird. Neben den Anwendungen den vorherigen Kapiteln wurde auch abstraktionsbasierte Modellprüfung wie in SATABS [CKSY05] erfolgreich auf großen Systemen eingesetzt [BBC$^+$06, CW02].

Als Alternative zum Software Bounded Model Checking wird der Software Model Checker SATABS [CKSY05] hinzugenommen und mit CBMC verglichen. SATABS basiert auf der iterativen Anwendung von Modellprüfung und Abstraktion CEGAR [CL00].

Ein minimales Beispielprogramm in Abbildung 5.1 illustriert die typischen

Unterschiede von Polyspace und CBMC. Polyspace kann nicht beweisen, dass die Forderung in Zeile 3 immer erfüllt ist. Aus diesem Grund trifft Polyspace eine konservative Approximation und meldet, dass eine mögliche Verletzung vorliegt. CBMC kann diese Warnung evaluieren um zu beweisen, dass die Verletzung tatsächlich niemals auftreten kann.

5.2 Abstrakte Interpretation mit Polyspace

Typische Laufzeitfehler in eingebetteten C Programmen umfassen Divisionen durch Null, arithmetische Überläufe sowie ungültige Speicherzugriffe. Polyspace analysiert den Quellcode zeilenweise und führt für jede Operation, die einen dieser Fehler auslösen kann eine *Beweisverpflichtung* an: Eine Zeile `a[5] = 6;` führt zu der Verpflichtung, dass die Größe des Arrays fünf übersteigt. Andernfalls würde auf eine ungültige Speicherstelle zugegriffen.

Als Initialzustand nimmt Polyspace an, dass alle Beweisverpflichtungen zu Fehlern führen können. Nur falls Polyspace beweisen kann, dass die Beweisverpflichtung immer erfüllt sein muss, wird diese nicht mehr als Warnung ausgegeben. Falls die Abstraktion zu grob ist, die Laufzeit zu lang ist oder aus anderen Gründen der Beweis nicht geführt werden kann, sprechen wir von einem *false positive*. Aufgrund des konservativen Startzustands ist einsichtig, dass der umgekehrte Fall, ein *false negative*, niemals eintreten kann.

Jede Beweisverpflichtung kann genau im Quellcode lokalisiert werden. Zur Kodierung des Wissens über den Status einer Beweisverpflichtung, wird in Polyspace [Pol08] eine Farbkodierung eingeführt:

R)ot: Die Beweisverpflichtung ist für jede Programmausführung, die die Zeile erreicht, nicht erfüllt.

O)range: Die generierte Überapproximation erlaubt mindestens eine Programmausführung in der die Beweisverpflichtung nicht erfüllt ist. Es ist jedoch unklar ob es sich um ein false positive handelt.

G)rün: Die Beweisverpflichtung ist für alle Programmausführungen erfüllt.

B) Schwarz: Die mit der Beweisverpflichtung assoziierte Zeile ist niemals erreichbar. Entweder ist die Zeile gar nicht erreichbar oder die Ausführung wird durch einen vorher auftretenden Fehler maskiert.

Aus den Zuständen G und R gibt es jeweils einen Übergang in den Zustand B, z.B. falls eine im Kontrollfluss vorgelagerte Beweisverpflichtung als Fehler erkannt werden. Die möglichen Übergänge sind in Abbildung 5.2 aufgeführt.

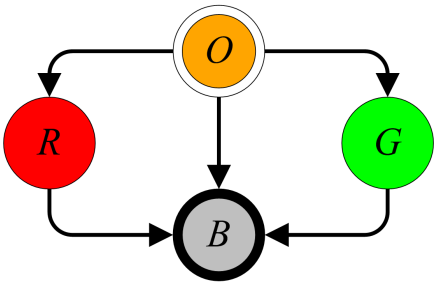


Abbildung 5.2: Polyspace definiert vier Zustände für Beweisverpflichtungen. Wenn man die Analyse als iterative Verfeinerung betrachtet, dann ist der initiale Zustand, dass alle Beweisverpflichtungen im Zustand O starten. Weitere Iterationen führen zu Übergängen in die Zustände R, G oder B. Im Zustand G und R ist noch immer ein Übergang in Zustand B möglich.

Tabelle 5.1 enthält eine quantitative Zusammenfassung der Beweisverpflichtungs-Zustände die Polyspace auf dem der Fallstudie zugrundeliegenden Softwaresystem meldet. Letzteres besteht aus neun Komponenten die unabhängig von Polyspace analysiert werden. Als Ergebnis werden 468 Warnungen ausgegeben von denen drei den Zustand R erhalten. 465 Warnungen sind also mögliche Falschmeldungen für Probleme wie Division durch null, Array Index Operationen und arithmetische Überläufe.

Auch wenn die Anzahl definitiver Fehler gering ist, muss dennoch ein enormer Aufwand getrieben werden um die 465 Warnungen zu sichten und zu beurteilen. In einigen Fällen ist die Warnung von Polyspace nicht nachvollziehbar, da der Quellcode arithmetische Komplexität beinhaltet und keine weiteren Informationen wie beispielsweise Programmabläufe, die den Fehler verursachen, von Polyspace gegeben werden. Ein Entwickler, der eine Warnung nachvollziehen will benötigt also einige Zeit um eine ungenaue Warnung nachzuvollziehen.

75 Warnungen und 2 Fehler wurden als Eingabe für die Verfeinerung durch BMC ausgewählt. Im Wesentlichen beschränkt sich die Auswahl auf Fehler der Kategorie Überläufe aus Modulen die nicht in C++ oder mit Assembleranteilen entwickelt wurden. Getrieben ist die Wahl durch die Auswahl von Modulen, die letztere Kriterien erfüllen.

In den nachfolgenden Abschnitten ist es das Ziel die 75 Warnungen auf Echtheit zu prüfen. Jede als Falschmeldung identifizierte Warnung vermindert den Inspektionsaufwand erheblich. Ein weiteres Ziel ist es, für die zwei Fehler wie auch für Warnungen, die nicht als Fehler oder false positive klassifiziert werden könnnen, Gegenbeispiele in Form von Programmabläufen zu erstellen. Dies wird direkt von CBMC unterstützt, da CBMC aus den Belegungen der erfüllbaren SAT-Formeln stets konkrete Programmpfade konstruiert.

Tabelle 5.1: Polyspace meldet 465 mögliche und 3 definitive Fehler in der analysierten Software. Die drei Beweisverpflichtungen mit Zustand R werden in jedem Fall bei Ausführung einen Laufzeitfehler verursachen. Beweisverpflichtungen im Zustand O weisen auf mögliche Probleme hin. Beweisverpflichtungen mit Zustand G sind immer sicher während Beweisverpflichtungen im Zustand B immer nicht erreichbar sind. 77 der 468 Warnungen wurden ausgewählt um sie mittels BMC weiter zu verfeinern.

Software	Array Index Operation				Division durch Null				Überlauf				Andere			
Modul	R	O	G	B	R	O	G	B	R	O	G	B	R	O	G	B
1	0	6	192	81	0	0	8	24	2	10	875	1005	0	27	3214	2551
2	0	5	577	3	0	0	17	1	0	31	1012	28	0	99	5148	157
3	0	0	14	0	0	0	3	0	0	13	151	30	0	21	1522	236
4	0	0	13	0	0	0	0	0	0	4	96	0	0	0	360	3
5	0	3	467	16	0	1	31	3	0	38	1097	94	0	58	4511	490
6	0	17	135	26	0	0	6	2	0	12	686	184	0	19	3385	1054
7	0	1	32	0	0	0	2	0	0	0	54	0	0	11	611	5
8	0	0	65	0	0	2	12	0	0	0	130	14	1	10	352	91
9	0	6	40	6	0	0	4	0	0	11	419	18	0	60	3595	177
Summe	0	38	1535	132	0	3	83	30	2	119	4520	1373	1	305	22698	4764
Ausgewählt	-	38	-	-	-	3	-	-	2	34	-	-	0	0	-	-
%Ausgewählt	-	100%	-	-	-	100%	-	-	100%	29%	-	-	0%	0%	-	-
Total	77/468 (16.5%)															

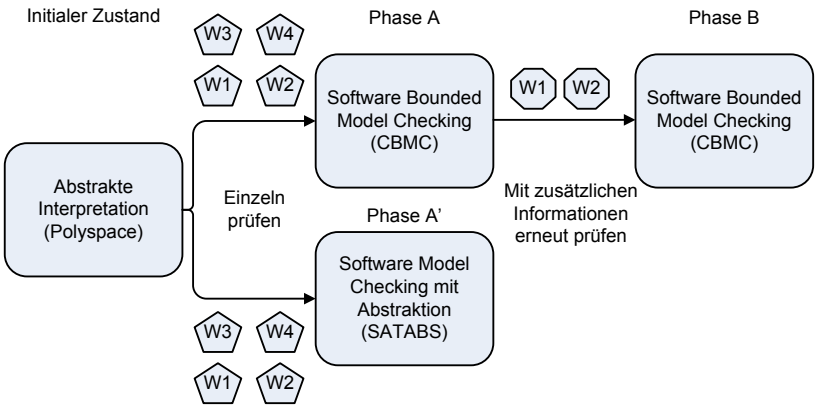


Abbildung 5.3: Die Warnungsausgaben von Polyspace werden in zwei Phasen A und B durch CBMC einzeln gefiltert. Alternativ wird in Phase A' das Werkzeug SATABS [CKSY05] eingesetzt.

5.3 Integration von Bounded Model Checking

Der Ansatz abstrakte Interpretation und Bounded Model Checking zu kombinieren kann als iterative Filterung bezeichnet werden: Zunächst wird abstrakte Interpretation auf den Quellcode angewendet. Jede erhaltene Warnung für eine konkrete Zeile im Quellcode wird anschließend einzeln mit CBMC geprüft. Die Prüfung durch CBMC kann den Zustand der Beweisverpflichtung ändern. Alle Änderungen sind hierbei als positiv anzusehen, da sie mehr Informationen über die entsprechende Korrektheit der Codezeile kodieren. Als Beispiel sei der Fall genannt, dass CBMC eine von Polyspace als mögliche Division durch Null gekennzeichnete Beweisverpflichtung als sicher beweist. Der Zustand wird also von O zu G transformiert. Die Filterung findet in zwei Phasen, A und B, statt. Abbildung 5.3 verdeutlicht die Verzahnung der Phasen.

Die initiale Eingabe für die Filterung durch CBMC ist die Polyspace Ausgabe. In Phase A wird CBMC automatisch und lokal auf Funktionen angewendet. Phase A' beschreibt die alternative Anwendung von SATABS statt CBMC. Falls in Phase A keine Verfeinerung erreicht wurde, kann meist zumindest ein Gegenbeispiel ausgegeben werden. Dieses kann zusätzlich mit einer manuellen Inspektion die Ursache einer Warnung klären.

In Phase B wird CBMC mit zusätzlichen Spezifikationen, die aufgrund der in Phase A gewonnenen Informationen gewonnen werden, als interaktives Werkzeug erneut als Filter eingesetzt. Zwar ist auch jeder einzelne Lauf von CBMC in Phase B automatisch, jedoch geht jedem Lauf eine manuelle Analyse und Spezifikation voraus.

5.3.1 Phase A: Automatische Filterung durch CBMC

In Phase A wird über alle Warnungen iteriert. Da sowohl CBMC wie auch SATABS für die Analyse ganzer Programme verwendet werden, muss in beiden Läufen ein Start- oder Einsprungspunkt für die Analyse definiert werden. Vollständig beschrieben werden kann die Menge der nötigen Funktionen durch alle, deren Aufruf direkt oder indirekt einen Aufruf der mit der Beweisverpflichtung assoziierten Codezeile führt. Wir nennen die Kette von dem Einsprungpunkt zu der Funktion, die die Kodezeile enthält, einen (möglichen) *Kontext* einer Warnung. Im Folgenden starten wir mit dem minimalen Kontext, also dem, der nur die Funktion enthält, in der das Problem auftritt. Nachfolgend wird der Kontext um Aufrufer der Funktion erweitert bis schließlich bewiesen werden kann, dass das Problem für alle aktuellen Kontexte nicht auftreten kann. Das Vorgehen wird von uns *schrittweise Kontexterweiterung* (SKE) genannt. Abbildung 5.4 stellt den Prozess anhand eines Beispiels graphisch dar.

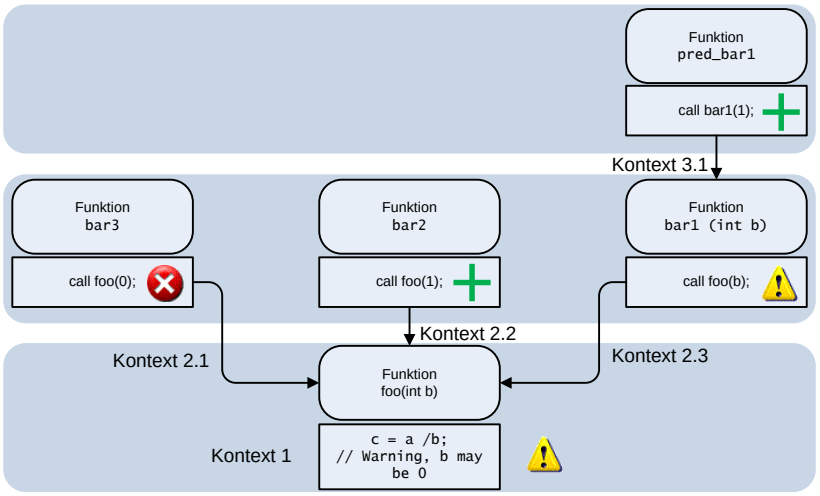


Abbildung 5.4: Ein Beispiel für schrittweise Kontexterweiterung mit CBMC.

Im Beispiel enthält die Funktion `foo` eine Zeile, die von Polyspace als mögliche Division durch null gekennzeichnet wurde. Um die Warnung zu widerlegen starten wir CBMC auf der Funktion `foo` (minimaler Kontext). Das Ergebnis des Aufrufs von CBMC ist, dass mit minimalem Kontext der Laufzeitfehler auftreten kann, da `b` null sein kann. Daraufhin erweitern wir den Kontext in drei verschiedene Weisen um die Funktionen `bar1`, `bar2` und `bar3`. Die drei Funktionen umfassen alle Aufrufer der Funktion `foo`. In zweiter Ebene wird CBMC auf allen drei Kontexten gestartet. Der Kontext `bar3` führt zu dem Ergebnis, dass die Beweisverpflichtung mit R gekennzeichnet werden muss. Hiermit wird der Erweiterungsprozess terminiert, da keine Sammlung

von Kontexten mehr gefunden werden kann, die das Ergebnis noch ändert.

Nimmt man an, die Funktion `bar3` sei im Beispiel nicht enthalten, würde CBMC beweisen, dass in den zwei anderen Kontexten `bar1` und `bar2` die Beweisverpflichtung erfüllt ist. In diesem Fall kann der Prozess mit dem Ergebnis G beendet werden.

Es ist möglich, dass CBMC aufgrund von Laufzeitproblemen keine weitere Kontexterweiterung mehr vornehmen kann. In einem solchen Fall bleibt das Ergebnis O. In fast allen Fällen kann CBMC jedoch für Kontexte Gegenbeispiele generieren. Diese tragen ebenso Information über den möglichen Fehler bei. Warnungen, für die ein Gegenbeispiel generiert werden kann, werden deshalb mit Zustand Oc abgegrenzt. Diese und andere Erweiterungen des Zustandsmodells für Beweisverpflichtungen sind in Abbildung 5.5 dargestellt.

Auch wenn die Experimente in Bezug auf die Kontexterweiterung manuell durchgeführt wurden, kann jedoch leicht eine Automatisierung erreicht werden, da die Berechnung (einer Überapproximation) des Aufrufgraphs ein einfaches Problem der Programmanalyse darstellt.

Phase A': Automatische Filterung durch SATABS CBMC wird im Gegensatz zu Polyspace lokal auf Funktionen und auf einzelnen Warnungen eingesetzt. Als Alternative zu CBMC betrachten wir den Software Model Checker SATABS [CKSY05], der mittels einer dynamischen Abstraktion (CEGAR [CL00]) arbeitet. Der Zweck der parallelen Evaluation zweier Werkzeuge ist die Prüfung der These, dass ein Einsatz eines Werkzeugs mit geringerem Abstraktionsgrad, wie CBMC, notwendig zur Verbesserung der Resultate ist. In anderen Worten: Wir prüfen, ob die falschen Warnungen wirklich durch die Abstraktion verursacht werden.

5.3.2 Phase B: Manuelle Analyse

In den Phasen A und A' konnte die Anzahl der false positives automatisiert verringert werden. Die Klasse der restlichen Warnungen wird in dieser Phase manuell genauer untersucht. Neben der reinen Verbesserung steht auch die Analyse der Gründe für die Entstehung der Warnungen im Vordergrund. Letztere kann helfen die Werkzeuge zu verbessern um zukünftig bessere Resultate automatisch wie in Phase A erreichen zu können. Im einzelnen werden drei Möglichkeiten für Verbesserungen in Phase B untersucht:

1. *Bibliotheksfunktionen* Die Gesamtsoftware beinhaltet einige Bibliotheken. Für Funktionen, die einer solchen Bibliothek entstammen, gilt die

Anforderung, dass sie auf dem gesamten Eingaberaum fehlerfrei arbeiten müssen. Normalerweise wird angenommen, dass eine Funktion nur unter den Argumenten korrekt arbeiten muss, die auch wirklich von anderen Programmteilen verwendet werden. Der Vorteil der strengen Annahme ist, dass keine Erweiterung des Kontextes über die Grenzen der Bibliothek hinaus vorgenommen werden muss. Kann die Sicherheit für eine Bibliotheksfunktion nicht bewiesen werden, so liegt in diesem Fall immer ein echter Fehler vor.

2. *Beschränkungen für Kommunikationskanäle* Kommunikation zwischen Modulen findet in der Form gemeinsam verwendeten Speichers statt. Zugriffe auf diesen müssen im untersuchten Produkt mittels definierten `get` und `set` Funktionen erfolgen. Sind die Werte, die ein Modul einem anderen als Eingabe senden kann im Wertebereich beschränkt, und ist diese Beschränkung nicht formalisiert, kann dies zu false positives führen, falls das sendende Modul nicht im aktuellen Kontext vorhanden ist. Im schlimmsten Fall müsste also die Gesamtsoftware in jede Analyse eingebunden werden – somit ginge jede Modularität der Analyse und damit die Hoffnung auf plausible Rechenzeit verloren. Die manuelle Analyse der mittels `set` gesendeten Werte kann zu einer Einschränkungen der Rückgabewerte der `get` Funktion führen. Diese Einschränkungen werden mittels des `assume` Befehls für CBMC formalisiert und können somit dazu führen, dass Warnungen widerlegt werden. In Fällen, in denen die Einschränkung des Wertebereichs nicht sicher feststeht, erhält man nur einen Korrektheitsbeweis unter der Annahme, dass die `assume` Anweisungen immer erfüllt sind.

3. *Problem Dekomposition* Das Problem des exponentiellen Zustandsraumes sowie das Vorhandensein unendliche oft auszuführender Schleifen stellen zwei natürliche Anwendungsprobleme für CBMC dar. Unter den analysierten Fällen finden sich in der Tat einige Warnungen, für die die Korrektheit von einer Initialisierung abhängt, die vor einer unendlichen Schleife und an entfernter Stelle im System stattfindet. Dies führt zu zwei Problemen: Erstens enthält der relevante Kontext eine unendliche Schleife und zweitens müssen große Teile von Quellcode zum Kontext hinzugefügt werden.

   Die Probleme lassen sich in der Praxis durch Dekomposition lösen: Besteht ein Programm aus zwei Teilen `P1; P2;` dann kann die Beweisverpflichtung aufgeteilt werden in die Annahme, dass eine Eigenschaft p_1 nach `P1` gilt, und den Beweis, dass unter diesen Umständen eine Eigenschaft p_2 in `P2` gilt. Eigenschaften, die in jedem Schleifendurchlauf gelten sollen, können ebenso durch Dekomposition bewiesen werden:

   - Die Eigenschaft gilt vor der ersten Ausführung.

- Falls die Eigenschaft vor einer beliebigen Ausführung gilt, dann ist sie auch nach einer weiteren Ausführung wahr.
- Die Beweisverpflichtung ist vor jeder von ihr abhängigigen Zeile innerhalb der Schleife wahr.

Sind alle drei Aussagen beweisbar, dann gilt die Verpflichtung in allen relevanten Fällen. Genauer muss noch eben die notwendige und hinreichende Invariante ermittelt und spezifiziert werden, damit die Dekomposition vorgenommen werden kann. Das Finden von Invarianten ist kein entscheidbares Problem, aber im Rahmen der Fallstudie war dies effizient machbar.

Wir geben nun jeweils ein Beispiel für die obigen Methoden. Die Beispiele sind fast identisch aus echten Funktionen übernommen. Es wurden jedoch Vereinfachungen und Umbenennungen vorgenommen.

Bibliotheksfunktionen In der folgenden Funktion `add()` kann es zu einem Überlauf in Zeile L1 kommen, falls `o2` nicht im Wertebereich von `short` liegt, aber die Summe von `o1` und `o2` wieder in dem Bereich von `short` ist.

```
    short add(short o1,long o2){
      if(o2>=(32767-(long)o1))
        return 32767;
      else{
        if(o2<=(-32768-(long)o1))
          return -32768;
        else
L1:       return o1+(short)o2;
      }
    }
```

Falls die Funktion nur mit bestimmten, kleinen Werten aufgerufen würde, fände kein Überlauf statt. Da die Funktion in der Fallstudie eine Bibliotheksfunktion ist, muss sie für alle Eingaben korrekt arbeiten. Folglich ist der Kontext, der nur aus der Funktion selbst besteht, ausreichend.

Eingabebeschränkungen Die Funktion `c_getval()` liest einen `short` Wert von einem geteilten Speicherbereich (das `ext[]` Array). In Zeile L2 kann ein Überlauf stattfinden aufgrund der impliziten Typumwandlung zum Rückgabetyp `short`. Polyspace, SATABS und CBMC geben korrekt aus, dass es in Zeile L2 zu einem Überlauf kommen kann (`o1 = -10000` und `o2 = 40000`).

```
extern unsigned int ext[10];

_Bool __precondition(){
  return (ext[0] & 0xfff << 3)+(ext[4] & 0xf)
     < 32768-5;
}

short c_getval(){
L2: return (ext[0] & 0xfff << 3)
      +(ext[4] & 0xf);
}
```

Entwickler wiesen darauf hin, dass der Überlauf nicht vorkommen kann, da die Werte in `ext[]` auf einen kleinen Wertebereich beschränkt seien. Eine hinreichende Einschränkung ist in der Funktion `__precondition()` kodiert. Die Beweisverpflichtung, dass `__precondition()` immer zu `true` evaluiert, kann durch Inspektion aller Schreibzugriffe auf `ext[]` bestätigt werden[1]. Nach Annahme, dass die Invariante gilt, konnte CBMC beweisen, dass kein Problem in Zeile L2 vorliegt.

Dekomposition Im Folgenden präsentieren wir die Behandlung unendlicher Schleifen durch Dekomposition. `m_init` wird nur einmal am Anfang des Lebenszyklus des Moduls aufgerufen. `m_main` wird periodisch, das heißt unendlich oft, aufgerufen.

```
// L3: Array Index ist im Intervall [0..9]
short ready,i,x; long v[10];

_Bool m_invariant(){
  return ready? (x<10 && i<=x && i>=0) : 1;
}

void m_init(){ ready = 0; }

void m_main(){
  if(!ready) x=10,i=0;
  for(;i!=x; i++)
    if(m_getval()+5)
L3:     v[i]=add(-i,c_getval()*2L);
```

[1] Weder CBMC, SATABS noch Polyspace konnte diese globale Invariante beweisen. Polyspace liefert allerdings die Information in welchen Zeilen Schreibzugriffe stattfinden, so dass die Invariante nach Inspektion bestätigt werden konnte.

```
        else{
          ready=0; break;
        }
    }
```

Das false positive stammt aus Zeile L3. Die Beweisverpflichtung ist, dass `v[i]` die Arraygrenzen einhält. Falls `m_invariant()` global erfüllt ist, dann ist auch Zeile L3 sicher: Die Schlüsselidee hierzu ist, dass `x` und `i` immer initialisiert sind, sobald die Programmausführung Zeile L3 erreicht. Die Initialisierung wird beim ersten Aufruf von `m_main()` vorgenommen. Die Invariante drückt aus, dass bei jedem Eintritt von `m_main()` die Initialisierung bereits stattgefunden haben muss. Die Invariante ist wahr nach der Ausführung von `m_init()`. Falls sie vor einer beliebigen Ausführung von `m_main()` gilt, dann gilt sie auch nach der Ausführung. Alle diese Beweisverpflichtungen werden von CBMC bewiesen.

Zusätzliche Beweisverpflichtungs-Zustände. Die vormalige Klassifikation definiert G, O, R, und B Zustände (Abschnitt 5.2). Um die neuen Ergebnisse zu erfassen benötigen wir eine erweiterte Klassifikation:

C)yan: Warnungen, die mittels Dekomposition und Eingabebeschränkungen als false positive erkannt werden.

O_c/O) range: Warnungen mit / ohne ein Gegenbeispiel, die in beiden Phasen weder widerlegt noch bestätigt werden [2].

M)agenta: Warnungen in Bibliotheksfunktionen, die aufgrund eines Gegenbeispiels direkt als Fehler erkenntlich sind.

Die zusätzlichen Zustände werden samt möglicher Übergänge in der Abbildung 5.5 aufgelistet. Als O markierte Beweisverpflichtungen können also in die Klassen R, M, B, C oder G übergehen.

5.4 Zusammenfassung der Ergebnisse

Tabelle 5.2 gibt die Resultate der Phasen A, A' und B wieder. Es ist bemerkenswert, dass SATABS nicht in der Lage war mehr Information als CBMC zu generieren. Es gibt nur einen Fall in dem SATABS ein Ergebnis liefern konnte und CBMC kein Ergebnis lieferte. Umgekehrt ist CBMC in der Lage 18 Warnungen zu widerlegen – SATABS ist auf drei Warnungen beschränkt.

[2] Ein großer Anteil konnte aufgrund fehlender Unterstützung von einigen Typkonversionen durch CBMC nicht untersucht werden.

Tabelle 5.2: Ergebnisse nach den Verfeinerungen der Phasen A,A' und B. Die initialen Zustände sind den Übergängen wie in Abbildung 5.5 gezeigt unterworfen.

		Start	Phase A/A'			Phase B
		Polyspace	SATABS	CBMC	Kombiniert	CBMC
Korrekt	B	0	1	1	1 (≈1%)	1 (≈1%)
	G	0	2	17	17 (22%)	27 (35%)
	C	-	-	-	-	13 (17%)
Unbekannt	O	75 (97%)	43	22	21 (27%)	15 (19%)
	O_c	-	30	36	37 (47%)	8 (10%)
Fehler	M	-	-	-	-	12 (16%)
	R	2 (≈3%)	0	0	0	0
	R_c	-	1	1	1 (≈1%)	1 (≈1%)

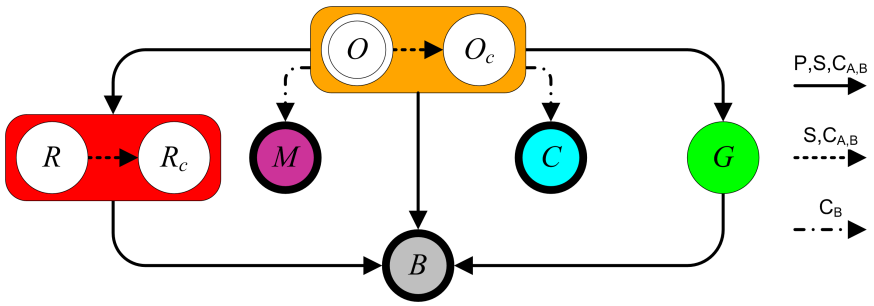


Abbildung 5.5: Erweiterung des Zustandsübergangsgraphen. Zusätzlich zu den ursprünglichen Zuständen (siehe Abbildung 5.2), wurden die Zustände R_c, O_c, C, und M hinzugefügt. Pfeile geben an welche Übergänge von Polyspace, SATABS oder CBMC in welchen Phasen ausgelöst werden. Punktierte Pfeile weisen darauf hin, dass SATABS oder CBMC Gegenbeispiele generieren. Übergänge zwischen O/O_c und M/C stammen aus Phase B.

Phase B verbessert die Ergebnisse gegenüber Phase A substantiell. Weniger als ein Drittel der Warnungen bleiben im initialen Status. 12 der nicht verbesserten Zustände sind einer unzureichenden Unterstützung von *Type punning*, also der Umwandlung von Typen, zuzuschreiben. Acht Warnungen sind mit Gegenbeispielen versehen. In nur drei Fällen kann aus Laufzeitgründen kein Ergebnis erlangt werden.

Die in Phase A erreichte Reduzierung um 23% konnte fast automatisch berechnet werden. Phase B beinhaltet manuelle Hilfestellungen für BMC. In drei Wochen konnte aber ein nicht mit dem geprüften System vertrauter Student 12 neue Fehler finden, die zuvor auch mittels intensivem Unittest nicht gefunden werden konnten. 13 Warnungen konnten in Klasse C transferiert werden. Sie bedürfen einer Änderung im Quellcode. 53% der Warnungen sind als false positives entlarvt.

Nur 23 (30%) der Warnungen bleiben für die manuelle Analyse durch Entwickler erhalten. Die Hälfte dieser kann mittels einer Erweiterung von CBMC um Typumwandlungen über Zeiger nochmals geprüft werden. Die Ergebnisse zeigen klar auf, dass CBMC sehr gut geeignet ist die Ergebnisse eines abstraktionsbasierten Werkzeugs zu verfeinern. Die Gegenprüfung mit SATABS lieferte weniger Ergebnisse.

Es muss beachtet werden, dass die Resultate möglicherweise sehr stark von der untersuchten Domäne, das heißt systemnahe Programme aus dem Automobilbereich, abhängig sein könnten. Zusätzlich sind die ausgewählten Beweisverpflichtungen vor allem aus dem Bereich arithmetischer Überläufe entnommen. Dies ist vielleicht eine besonders wenig ergiebige Domäne für Abstraktionen.

Laufzeiten Alle Experimente wurden auf einem System mit Intel Pentium 4 Prozessor (3 Ghz.) und 4 GB Speicher ausgeführt. Die maximal erlaubte Laufzeit wurde mit sieben Stunden festgesetzt. Die drei Tabellen, 5.3, 5.4 und 5.5), fassen die Laufzeit und Speicherverbrauchsdaten zusammen.

Tabelle 5.4 enthält die Laufzeiten aus Phase A beschränkt auf die Beweisverpflichtungen bei denen CBMC und SATABS die gleichen Ergebnisse erzielt haben. In Fällen, in denen ein Gegenbeispiel generiert werden konnte, zeigen CBMC und SATABS ähnliches Verhalten. Hierbei wird die Tatsache ignoriert, dass CBMC in sieben Fällen Ergebnisse liefert, in denen SATABS das Laufzeitlimit erreicht. Umgekehrt findet sich nur eine Beweisverpflichtung in der SATABS alleine ein Ergebnis liefert. Gerade die letzteren Beweisverpflichtungen tragen zu der insgesamt höheren Laufzeit von CBMC bei (Tabelle 5.3). Die von CBMC generierten Booleschen Formeln enthalten bis zu sieben Millionen Variablen und 23 Millionen Klauseln. Wie zu erwarten wurden erfüllbare Instanzen schneller als unerfüllbare gelöst.

Tabelle 5.5 zeigt, dass in Phase B, wohl aufgrund der zusätzlichen Information durch Invarianten und durch Dekomposition, Instanzen schneller gelöst werden konnten.

5.5 Verwandte Arbeiten

Abstrakte Interpretation wurde bereits zuvor erfolgreich auf systemnahe Quellen angewendet: Delmas und Souyris berichten von der Anwendung auf Programme aus der Flugzeugindustrie [DS07]: Die Autoren betonen, dass eine enge Verzahnung zwischen den Entwicklern und den Anwendern der Verifikationswerkzeuge das Ergebnis sehr beeinflusst. Im Besonderen ist das manuelle Entfernen und Untersuchen von Warnungen davon betroffen. In der speziellen Domäne, die die Autoren untersuchten, konnten alle Warnungen eliminiert werden. Im Gegensatz zu ihrer Arbeit sind in der vorliegenden Arbeit modulare Programme mit datenintensiven Eigenschaften untersucht worden. Das Verifikationswerkzeug wurde nicht zusammen mit den Entwicklern auf diese spezielle Domäne angepasst und es ist folglich fraglich, ob alle Warnungen mit dem abstrakten Interpretationwerkzeug Astrée [CCF+05] hätten vermieden werden können.

Rival [Riv05] untersucht den Ursprung von false positives in Astrée. Als Lösung für das Problem stellt er einen Prozess vor zur semi-automatischen Untersuchung von Warnungen. Wie in unserem Ansatz schlägt Rival schlüssige Analysetechniken vor um die Warnungen zu filtern. Anstatt auf Model Checking, verweist der Autor auf verschiedene spezialisierte Techniken der statischen Analyse: Rückwärtige Analyse, die Partitionierung von Abläufen und das *Slicing* von Programmen. In drei kleinen Fallstudien kann der

Tabelle 5.3: Laufzeiten von Phase A und Polyspace (gruppiert nach Ergebnis, max. sieben Stunden). VF steht für Beweisverpflichtungen, die nicht wahr sind, während VS für wahre und bewiesene Beweisverpflichtungen steht. Die Spalten vier bis sechs stehen für verschiedene Fehler bei der Verifikation, die ein Ergebnis verhindert haben.

	Information gewonnen		Fehler / Keine neue Information				
	VF	VS	C-Semantik	Verfeinerungsfehler	Speicherfehler	Laufzeitende	Total
SATABS							
$\sum t$ [h:m]	0:36	4:12	0:12	16:22	28:18	56:27	106:09
Timeout	0	0	0	0	0	8	8
#claims	31	3	12	14	9	8	77
CBMC							
$\sum t$ [h:m]	9:15	19:57	0:01	-	6:23	42:06	77:43
Timeout	0	0	0	-	6	6	6
#claims	37	18	12	-	4	6	77
Polyspace							
$\sum t$ [h:m]	-	-	-	-	-	-	28:08
#runs	-	-	-	-	-	-	9

Tabelle 5.4: Vergleich von Laufzeiten für Experimente mit identischem Ergebnis (Phase A, Probleme durch nicht-unterstützte C Syntax nicht enthalten).

	VF		VS		Segm. fault		Total	
	CBMC	SATABS	CBMC	SATABS	CBMC	SATABS	CBMC	SATABS
$\sum t$ [h:m]	0:39	0:36	0:02	4:12	1:58	0:23	2:39	5:12
$\#claims$	30		3		1		34	

Tabelle 5.5: Laufzeiten nach Phase und Ergebnis akkumuliert.

Phase → Result	A → VF	B → VF	A → VF	B → VS	A → Segm. f.	B → VS	∑A	∑B
$\sum t$ [h:m]	0:16	0:06	7:45	6:17	56:07	16:02	64:08	22:26
$\#claims$	1		16		7		24	

Autor Warnungen semi-automatisch widerlegen. Es ist unklar wie sich sein Ansatz beim ACC System und der großen Anzahl von Warnungen verhalten würde. Im Gegensatz zu Phase A verwendet Rival manuell ausgesuchte Ausführungsmuster und Beschränkungen von Eingaben, das heißt sein Ansatz ähnelt eher Phase B. Die Laufzeiten die Rival angibt, sind jedoch niedriger als beim Einsatz von CBMC.

Andere Beispiele für die Anwendung von Model Checking auf große systemnahe Programme sind in [BBC+06] und [CW02] gegeben. Da keine Anwendung von beschränkter Modellprüfung vorliegt, sind die Ergebnisse am besten mit der Anwendung von SATABS zu vergleichen. Da die Studien keinen Vergleich anstellen, bescheinigen sie abstraktionsbasiertem Model Checking gute Analysefähigkeit für die obigen Systeme. CBMC konnte die Erfolge von SATABS in unserer Studie klar übertreffen. Die gewonnene zusätzliche Information hängt jedoch auch von der Domäne der Software und der Spezifikationen ab die untersucht werden. In [BBC+06] und [CW02] werden generische Eigenschaften von Schnittstellen geprüft. Unsere Beobachtung bezieht sich auf die Analyse bitsensitiver Eigenschaften wie arithmetische Überläufe was die Vergleichbarkeit der Studien mit unserer Arbeit einschränkt.

Andere Forschergruppen haben ebenso die Kombination verschiedener Verifikationstechniken untersucht. Das Orion Projekt [DN05] beinhaltet eine Kombination von Datenflussanalyse mit dem *Satisfiability Modulo Theory* (SMT)-Beweiser Simplify [DNS05]. In ihrer Publikation [DN05] schlagen die Autoren eine fast beliebige Kombination von Techniken vor ohne diese jedoch zu konkretisieren oder anzuwenden.

Beyer et al. [BHT07] beschreiben die Konvergenz zwischen (klassischer) Programmanalyse und Modellprüfern. Sie präsentieren einen Algorithmus der verbandbasierte und baumbasierte Verfahren engmaschig kombiniert. Im Gegensatz zu ihrer Arbeit haben wir ein industrielles System untersucht. Statt einer engmaschigen Verzahnung schlagen wir eine Iteration in zwei Phasen vor.

Schmidt [Sch98] ist unserer Einschätzung nach einer der ersten Autoren, der die Kombination von Modellprüfung, Flussanalysen und abstrakter Interpretation in Erwägung zieht. In seiner Arbeit beschreibt Schmidt jedoch nur die Ähnlichkeiten der Verfahren.

5.6 Diskussion und Erfahrungsbericht

Die Idee verschiedene Techniken zur Verifikation zu kombinieren ist ansprechend und ohne großen Aufwand realisierbar. In unserer Studie konnten wir zeigen, dass CBMC mehr als 20% der Warnungen von Polyspace automa-

tisch widerlegen konnte. Durch manuelle Inspektion in Phase B konnte die Zahl verbleibender Warnungen weiter reduziert werden. Aufgrund des sicherheitskritischen Anwendungsbereichs und der hohen Kosten einer manuellen Quellcodeanalyse ist die vorgeschlagene Methode für die Praxis sehr gut geeignet.

Die Autoren anderer Fallstudien, beispielsweise Delmas und Souyris [DS07], geben an, 100% der falschen Warnungen identifizieren zu können. Diese Aussage ist mit drei Vorbedingungen verknüpft:

- Das Programm ist nicht-modular und der Quellcode muss formal spezifiziert sein, so dass Wissen über Schnittstellen abgeleitet werden kann.
- Die Anwender und Entwickler des Verifikationswerkzeugs müssen eng zusammenarbeiten.
- Das Programm muss aus der Domäne der eingebetteten Systeme der Flugzeugindustrie stammen und darf nur wenig datenintensive Berechnungen enthalten.

Es gibt Fälle wo alle Forderungen erfüllt sind wie Delmas und Souyris gezeigt haben. In unserem Fall und allgemein in industrieller Praxis ist es jedoch fraglich ob auch nur eine der Anforderungen erfüllt werden kann. Dann kann manuelle Anwendung von Werkzeugen nicht vermieden werden wie es unsere Ergebnisse aufzeigen. Phase B unterstützt die Aussage, dass Software Bounded Model Checking in solchen Fällen noch immer eine beträchtliche Hilfe darstellen kann. Die durch SBMC gelieferte Präzision scheint notwendig um die Reviewkosten zu minimieren und Akzeptanz für formale Softwaremethoden zu schaffen. Nur 31% der Warnungen konnten nicht widerlegt werden. Von diesen konnten 50% der fehlenden Unterstützung von CBMC eines speziellen C Mechanismus zugeschrieben werden. Diese kann unmittelbar in CBMC nachimplementiert werden. Für 37% der verbleibenden Warnungen konnten zudem Gegenbeispiele als Analysehilfe generiert werden.

5.7 Zusammenfassung des Kapitels

Nachdem in Kapitel 4 das Problem der Verifikation offener und großer Systeme herausgearbeitet wurde, konnte in diesem Kapitel eine Lösung vorgestellt werden.

Die Kombination von abstrakter Interpretation als initiale Methode zum schlüssigen Auffinden problematischer Zeilen und der Methode des Software Bounded Model Checking zur weiteren Filterung, hat sich in der Bosch-Fallstudie als sehr erfolgreich erwiesen. Im Vergleich zum alleinigen Einsatz

von abstrakter Interpretation konnte die Zahl der false positives stark reduziert werden. Der alleinige Einsatz von SBMC hätte zu einer stärkeren Modularisierung und somit, wie in der Linux-Fallstudie, zu nicht betrachtetem Verhalten geführt.

In der Literatur setzte nach Veröffentlichung der *counter example guided abstraction refinement* (CEGAR) Methodik [CL00] ein Trend zur Abstraktion von Programmen ein. Die gewonnenen Verfahren, zu denen man auch abstrakte Interpretation rechnen kann, sollten unserer Ansicht nach sinnvollerweise mit nicht abstraktionsbasierten Verfahren kombiniert werden: Durch den Vergleich von SATABS [CKSY05] mit CBMC [CKL04], konnte die These gestützt werden, dass Abstraktion für viele Falschmeldungen verantwortlich ist. CBMC komplementiert abstrakte Interpretation.

Im folgenden Kapitel wenden wir uns einer Unzulänglichkeit bei der *Einsetzbarkeit* von Verifikationtechniken zu. Software im industriellen Kontext oder auch beim Linux Betriebssystem liegt nicht alleine als Sammlung von C-Dateien vor. In dem Feld der *Software Produkt Familien* oder *Software Produkt Linien* werden Mengen von konkreten Programmen zusammengefasst. Linux kennt unzählige *Varianten* für viele Architekturen und Hardware-Konfigurationen, die zusammen eine Produktlinie bilden.

Linux ist also nicht ein Programm, sondern eine Menge von spezialisierten Programmen, *Varianten* genannt. Varianten werden häufig mittels bedingter Kompilation, also der Modifikation von Quellcode außerhalb der Programmiersprache, beispielsweise C, kodiert. Da wir, und die Literatur allgemein, nur die Semantik von C ohne Präprozessor betrachtet haben, kann die Betrachtung von Produktfamilien bisher nicht vorgenommen werden.

In Kapitel 6 wird die Einsetzbarkeit von Verifikationstechniken, die auf C Dateien arbeiten erweitert, so dass Software Produkt Familien analysierbar werden.

Verifikation auf Software Produkt Linien

6

Kapitel 5 enthält einen Rahmen zur Kombination von abstrakter Interpretation und Software Bounded Model Checking. Dieser stellt bereits eine ausgereifte Technik dar mit der man systemnahe Programme industrieller Komplexität analysieren kann. Es gibt jedoch eine weitere Komplexitätsdimension die bisher im Gebiet der Softwareverifikation nicht behandelt wurde: Moderne Softwareprodukte, wie auch das Linux Betriebssystem, müssen sich speziellen Nutzeranforderungen anpassen können: beispielsweise das Vorhandensein von spezieller Hardware. Zu diesem Zweck wird in vielen industriellen Programmen eine Möglichkeit geboten, mehrere Varianten eines Produkts nebeneinander zu entwickeln und ausliefern zu können. Diese Varianten können in der Form von *Software Produkt Linien* (SPL) organisiert sein.

Bisherige Ansätze der Softwareverifikation verwenden implizit die Einschränkung, dass nur jeweils eine Variante eines Produkts separat verifiziert werden kann. Zur vollständigen Korrektheitsaussage muss also jede Variante einzeln betrachtet werden. Im Falle von Linux sind dies mehr als 2^{4000} Varianten wie wir im Rahmen der Erstellung der folgenden Fallstudie herausfanden. Im Folgenden wird beschrieben wie die Verifikation einer Menge von Varianten effizient vollzogen werden kann. Zunächst beschreiben wir jedoch noch einmal die Motivation und Ausprägung von Software Produkt Linien in der Praxis.

Bisher wurde ein Softwaresystem immer als eine Sammlung von Quellcodedateien angesehen. Für viele moderne Systeme vernachlässigt dies jedoch die Dimension der (statischen) Konfigurierbarkeit. Eine Software kann beispielsweise verschiedene Module enthalten, die jeweils an- und abgeschaltet werden können (zum Beispiel ein Treiber für eine Hardware). Bei vielen systemnahen Applikationen wird eine Software speziell für eine (Hardware) Zielplattform konfiguriert: die Größe von `int` Datentypen in C ist ein prominentes Beispiel einer systemnahen Diversifizierung. Je nach Prozessortyp und Compiler kann eine `int` Variable in C 16, 31, 32 oder 64 Bit umfassen.

Softwareverifikation demonstriert in Fallstudien die Anwendbarkeit, indem nur eine Variante oder eine Konfiguration ausgewählt und behandelt wird. Offensichtlich kann dies jedoch zu maskierten Fehlern (*false negative*) führen. Sind die Annahmen, die die Auswahl einer Konfiguration begleiten, nicht for-

malisiert, kann es ebenso zu Falschmeldungen kommen: Wird eine C Anwendung für eine Plattform konfiguriert, die 16 Bit breite `int` Typen verwendet, dann darf das Verifikationswerkzeug nicht annehmen, dass auch 32 Bit `int` zugelassen sind.

Die genaue Unterscheidung zwischen einer konfigurierbaren Software und einer Softwareproduktlinie ist anhand vieler Faktoren des Gesamtentwicklungsprozesses zu treffen. Wir beschränken das Verifikationsproblem auf die Herausforderung, eine große Anzahl möglicher, und ähnlicher, Varianten eines Programms zu untersuchen. Bei Produktlinien existiert ein Modell, das die Generierung und Unterschiede der Varianten beschreibt. Dieses Modell muss in Betracht gezogen werden, soll eine schlüssige und vollständige Verifikation von großen Mengen von Softwarevarianten ermöglicht werden.

Im folgenden Kapitel beschreiben wir das generelle Problem der unzureichenden Abdeckung von Softwarevarianten in allen modernen Veriifkationsmethoden. Anschließend beschreiben wir abstrakt einen Ausweg aus dem Problem. Zur Demonstration des neuen Verfahrens wenden wir es konkret auf ein hochkonfigurierbares System industrieller Komplexität an: das Linux Betriebssystem.

6.1 Einleitung

Das Hauptziel der Einführung von Konfigurationen in ein Softwaresystem ist Anpassungsfähigkeit. Letztere kann zu höherer Leistung durch bessere Adaptierbarkeit, eine breitere Anwendbarkeit (zum Beispiel für unterschiedliche Hardwareplattformen) und einer besseren Wartbarkeit führen. Oft wird auch der Begriff der *Wiederverwendbarkeit* genannt.

Als Gegenpol zu den Vorteilen findet zwangsläufig eine Erhöhung der Komplexität des System statt. Ein Merkmal eines Systems, *Feature* genannt, ist die atomare Einheit in der sich Varianten unterscheiden können. Sind n an- und abschaltbare *Features* vorhanden ergeben sich bis zu 2^n mögliche Varianten. Da konventionelles Testen häufig bereits ohne diese Komplexität an die Grenzen stößt, tritt die Frage auf, ob diese zusätzliche Dimension den Testansatz vollständig unplausibel macht.

Formale Techniken haben die Fähigkeit mit sehr großer Komplexität umzugehen wie in der AES-Fallstudie gezeigt. Es ist also wünschenswert diesen Vorteil für konfigurierbare Systeme zu nutzen. Bisher findet sich in der Literatur keine formale Verifikation einer kompletten Familie von Varianten. Die Prüfung von Konfigurationsbeschränkungen, wie etwa Abhängigkeiten unterschiedlicher Teilbereiche, wurde bereits von Sinz et al. anhand des Konfigurationsmodells des Apache Webservers beschrieben [SKKM03]. Jedoch

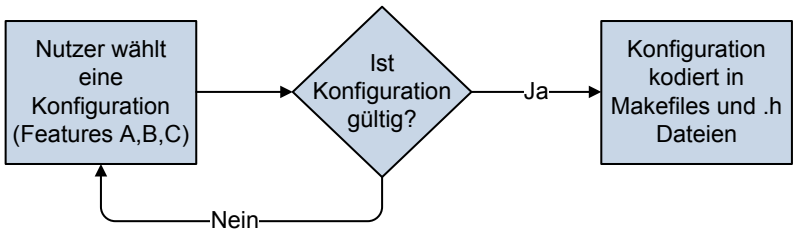


Abbildung 6.1: Traditioneller Konfigurationsprozess mittels Make und bedingter Compilation: Der Nutzer wählt zunächst eine Konfiguration aus. Ist die Konfiguration gültig, wird anhand zusätzlich generierter Makefiles und C-Header Dateien die aktuelle Konfiguration an den Compiler weitergegeben. Dieser erzeugt dann die gewählte Variante.

betrifft die Studie nur die Ebene der Konfiguration, nicht aber die Auswirkungen der Konfiguration auf die Software.

Im Folgenden stellen wir eine neue Technik vor, die eben dieses Problem löst. Die Technik – *Lifting* genannt – erweitert den Verifikationsprozess um eine Konfigurationssemantik und deren Auswirkung auf mögliche Programmabläufe. Wir konzentrieren uns bei der Beschreibung der Technik auf Softwaresysteme, die zur Compilezeit mittels Präprozessor modifiziert werden können. Der Ansatz ist jedoch breiter einsetzbar, beispielsweise können auch die von Estublier und Casallas zusätzlich eingeführten historischen und kooperativen Dimensionen [EC95] der Variantenbildung behandelt werden. Die letzteren finden sich in der Praxis kodiert in Versionen einer Software (*relẽase*) und parallel vorgenommen Entwicklungen (*branches*) in Versionsverwaltungssystemen wie SVN oder CVS. Lifting kann als Prozess verstanden werden, der eine konfigurierbare Software umwandelt, so dass alle Konfiguration zur Laufzeit stattfindet. Traditionelle Konfiguration wird in Abbildung 6.1 illustriert.

Durch Lifting können folgende Defekte eines Software Entwicklungsprozesses mit generierbaren Varianten [CE00] behandelt werden:

D1. Das Modell, das die Abhängigkeiten von Features beschreibt (*Feature Modell*), kann inkonsistent sein.

D2. Die Beschränkungen und Abhängigkeiten des Feature-Modells können

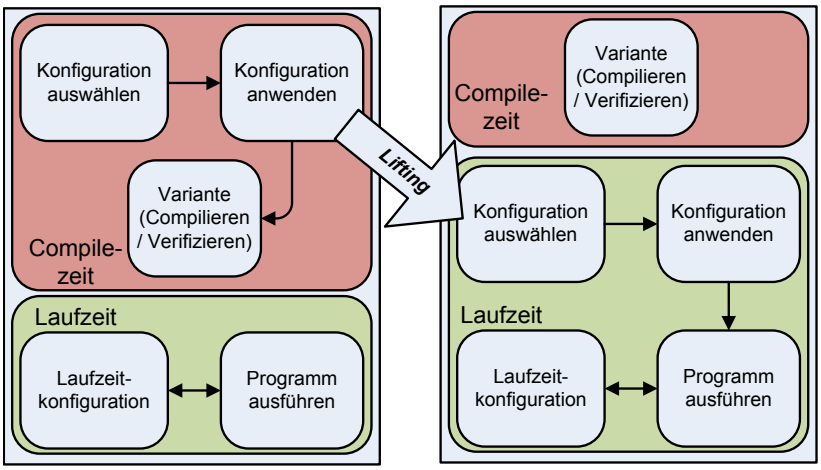


Abbildung 6.2: Um bisher Verifikation auf Software Produktlinien zu betreiben muss eine konkrete Variante hergestellt werden. Diese kann dann mit gängigen Softwareanalyse-Werkzeugen untersucht werden (vgl. a). Mittels Lifting können alle Varianten gleichzeitig untersucht werden, indem sie in ein Metaprogramm kompiliert werden. Dies ist dadurch ermöglicht, dass das Metaprogramm in der gleichen Sprache geschrieben ist wie die Varianten. Wir setzen voraus, dass die Basis aus der die Varianten generiert werden in einer ähnlichen Sprache wie die Varianten geschrieben ist. Die Variantenerstellung kann durch z.B. bedingte Kompilation erstellt werden.

von den tatsächlich implementierten Varianten abweichen.

D3. Es kann einzelne Varianten geben die Laufzeitfehler enthalten.

Das dritte Problem ist hierbei das grundlegenste. Als Hinführung werden wir zunächst beschreiben wie Probleme der ersten beiden Arten behandelt werden können. Aufbauend wenden wir uns dann der dritten Art zu. Die Linux Fallstudie dieses Kapitels (Abschnitt 6.3) und die Ergebnisse (Abschnitt 6.4) werden nachweisen, dass Fehler der ersten beiden Arten gefunden werden können. Zusätzlich präsentieren wir Laufzeiten, die belegen, dass die Abdeckung der Konfigurationsdimensionen in Beispielen keine zusätzliche Laufzeit in einer Software Bounded Model Checking Verifikation verursacht. Somit kann das dritte Problem als gelöst betrachtet werden.

6.2 Einführung von Lifting

Zubnächst führen wir einige Begriffe ein um dann die Technik *Lifting* einzuführen.

6.2.1 Begriffseinführung

Der Begriff *Verifikation* wird, wie bereits in Abschnitt 1.2 erläutert, für viele verschiedene Verfahren zur systematischen Prüfung verwendet. Lifting ist prinzipiell mit jeder Art von (Quellcode-)Verifikationstechnik kombinierbar, in der folgenden Fallstudie wird jedoch nur Software Bounded Model Checking verwendet.

Eine der bedeutendsten Bereiche des *Software Configuration Management* (SCM) behandelt das Management und die Kontrolle verschiedener *Versionen* eines Software Systems. Die Versionen können in den historischen, logischen und kooperativen Dimensionen entstehen [EC95]. *Software Varianten* fallen in die logische Dimension der Versionierung und spiegeln die Notwendigkeit wider, Software für verschiedene Kunden oder allgemein Anforderungen entwickeln zu müssen. In dieser Arbeit behandeln wir nur logische Versionierung obwohl das Verfahren auch auf andere Dimensionen anwendbar ist.

Historische und kooperative Dimensionen werden überlicherweise über ein Versionskontrollsystem (wie SVN oder CVS) verwaltet. Die Verwaltung logischer Varianten scheint bisher auf domänenspezifischen Insellösungen zu beruhen. Laut Estublier und Casallas [EC95] sind in der Praxis zwei Vorgehensweisen vorherrschend:

1. Varianten werden als Zweige oder Versionen verwaltet.

2. Varianten werden durch bedingte Kompilation implementiert.

Die atomare Einheit für Software Configuration Management wird *Konfigurationselement* genannt. Im Zusammenhang mit Variantenmanagement wird der Begriff *Feature* verwendet um Einheiten eines Systems zu bezeichnen, die sich in unterschiedlichen Varianten unterscheiden können. Repräsentation, Verwaltung und Konfiguration von Varianten ist ein Teil des Feldes der *Generativen Software Entwicklung* (GSE [CE00]). In der GSE werden Einschränkungen und Abhängigkeiten zwischen verschiedenen Features in einem *Feature Modell* beschrieben. Die Form eines konkreten Modells ist meist eine Sammlung von Booleschen Wenn-Dann Regeln mit kleineren Erweiterungen wie Kardinalitätskonstanten: *Wenn Feature A ausgewählt wird, dann müssen auch* **zwei** *Elemente der Menge {B,C,D} ausgewählt werden.*

Im Folgenden geben wir eine Beschreibung der Probleme, die mittels Lifting gelöst werden können. Zusätzlich geben wir eine abstrakte Beschreibung von Lifting, die im weiteren Verlauf durch eine genaue Implementierung in der Fallstudie konkretisiert wird.

6.2.2 Lifting

Definition 17 (Konfigurations) Lifting *ist ein Prozess, der ein Software Generierungs System und eine Software Produktfamilie als Eingabe nimmt und als Ausgabe ein Metaprogramm*[1] *erstellt, das jede mögliche Variante kodiert. Das Metaprogramm muss in der Sprache der Varianten geschrieben sein.*

Lifting muss sicherstellen, dass das Metaprogramm genau dann verifizierbar ist, falls alle Varianten verifizierbar sind. Im Detail kann dies realisiert werden, indem alle Programmabläufe die in einer Variante möglich sind, auch im Metaprogramm stattfinden können.

Abbildung 6.2 erläutert den Unterschied zwischen dem konventionellen generativen Erstellungsprozess einer Software und dem Prozess der mittels Lifting durchlaufen wird. Für klassische Verifikation muss der Verifikationsprozess für jede Variante durchlaufen werden und es ist nicht möglich Information aus verschiedenen Läufen wiederzuverwenden (zum Beispiel gewonnene Kandidaten für Invarianten). Mit Lifting wird ein Metaprogramm einmalig erstellt und in diesem werden alle Varianten auf einmal kodiert und verifiziert. Der Zusammenhang zwischen einzelnen Varianten oder das Faktum, dass das Ergebnis der Verifikation nicht von der Wahl einer konkreten Variante abhängt, können also berücksichtigt werden. Da das Metaprogramm in einer normalen Programmiersprache geschrieben ist, kann zur Verifikation auf Standardwerkzeuge wie CBMC [CKL04] zurückgegriffen werden.

Wie bereits erwähnt soll das Metaprogramm das Verhalten aller Varianten umfassen. Um dies zu erreichen müssen zwei Teilprobleme gelöst werden:

Zunächst muss das Feature-Modell in die Zielsprache, also die des Metaprogramms, übersetzt werden. Dies kann durch Einführung neuer Variablen für jedes Feature erreicht werden. Im Falle von Features die nur aktiviert und deaktiviert werden können reicht eine Boolesche Variable. Jede Beschränkung kann nun in einer Beschränkung der neuen Variablen kodiert werden.

Beispiel 2 *Ein Feature-Modell enthalte zwei Features A und B die sich gegenseitig ausschließen. Lifting führt zwei neue Variablen* `bool A,B` *ein. Der Ausschluss kann durch eine Bedingung* `!(A && B)` *ausgedrückt werden.*

Die zweite Anforderung an ein Metaprogramm ist die Repräsentation von Effekten, die durch *bedingte Kompilation* erreicht werden. Bedingte Kompilation ist mächtig genug zwei beliebige Varianten in einem Programm zu kodieren:

[1] Eine andere Bezeichnung wäre *Multivarianten-Programm.*

```
#ifdef VARIANTE_A
   PROGRAM_A
#elseif
   PROGRAM_B
#endif
```

In ähnlicher Weise ist es auch immer möglich ein solches Programm in ein (Präprozessor-freies) Metaprogramm umzuwandeln:

```
bool A;
PROGRAM_A'  // Namen mit Präfix '__A_' versehen
PROGRAM_B'  // Namen mit Präfix '__B_' versehen

void main() {
    if(A)
      __A_main();
    else
      __B_main();
}
```

Wobei im obigen Beispiel die Programme A und B umgewandelt werden müssen, so dass sie getrennte Namensräume verwenden (`__A_main` statt `main`). Zwar ist das Beispiel C spezifisch, ähnliche Beispiele lassen sich jedoch leicht für andere Sprachen konstruieren. Es ist meist sinnvoll die Granularität der Verzweigung nicht auf Ebene der Hauptfunktion `main` zu machen. In der späteren Fallstudie beschreiben wir detailliert wie eine optimale Kodierung für Linux Module gefunden werden kann (Abschnitt 6.3).

Abbildung 6.3 zeigt die Struktur eines Metaprogramms auf.

Die Komplexität der Analyse des Metaprogramms ist schlimmstenfalls gleich der Summe der Komplexität der sequentiellen Analyse aller Varianten. Dies ist ersichtlich, da der Verifikationsalgorithmus durch Fallunterscheidung den Suchraum direkt wieder in die Varianten aufspalten kann. Wir nehmen aber an, dass Varianten einen hohen Anteil gleicher Codestücke enthalten. Unter dieser Annahme entstehen Vorteile bei der Analyse des Metaprogramms:

1. Konfigurationsunabhängige Ergebnisse werden nur einmal berechnet.
2. Das Analysewerkzeug kann den Suchraum effizient aufspalten: Wenn die Korrektheit nur von einigen Features abhängt, reicht es, nur diese zu untersuchen. Die Durchsuchung des Suchraums ist für ein Metaprogramm frei wählbar während bei naiven Vorgehen zuerst nach Varianten aufgegliedert wird.

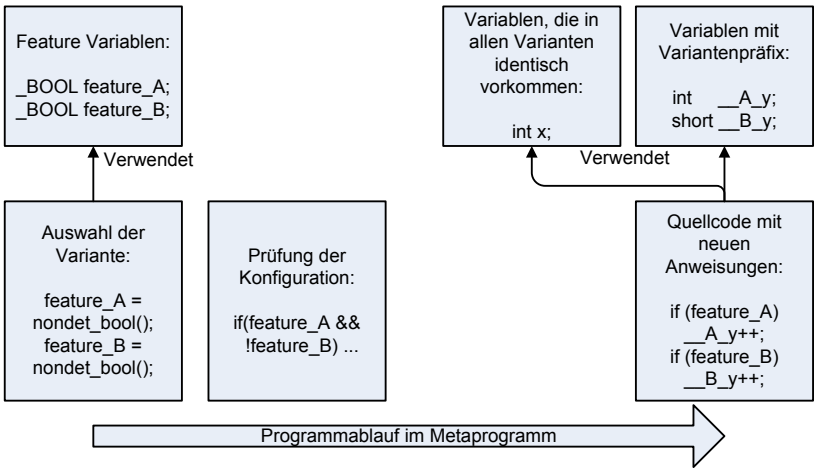


Abbildung 6.3: Ein Metaprogramm, das durch Lifting erstellt wurde, enthält zwei neue Variablenkategorien. Zum Einen werden zur Kodierung der Konfiguration *Feature*-Variablen erstellt. Zum Zweiten werden Variablen, die nicht in allen Varianten gleich definiert sind kopiert und mit einem Präfix versehen. Im Bereich der Anweisungen kommt die Auswahl und die Prüfung der Konfiguration hinzu. Zusätzlich werden im originalen Quellcode Anweisung eingefügt falls eine Anweisung nur in einer Variante ausgeführt würde.

Bisher ist die Beschreibung der Lifting Technik abstrakt und es ist fraglich ob sie für die große Anzahl von Implementierungstechniken für Software Produktfamilien generell konkretisiert werden kann. Im Folgenden zeigen wir die Anwendung von Lifting auf das Betriebssystem Linux. Das Vorhandensein einer konkreten Implementierung ermöglicht es uns, den Prozess genau zu definieren.

6.3 Eine Fallstudie: Linux als eine Software Produktlinie

Sincero et al. beschreiben inwiefern sich Linux als Software Produktlinie auffassen lässt [SSSPS07]. Die Generierung von Varianten in Linux ist durch bedingte Kompilation realisiert. Der Quellcode ist in C – und einem sehr kleinen Anteil von systemspezifischem Assemblercode – geschrieben. Letzterer ist nicht in dieser Studie inbegriffen.

Der Linux C Quellcode, der in Tausenden von Modulen und Subsystemen organisiert ist, kann über die folgenden Mittel konfiguriert werden:

- Die *Architektur Definition*, die bestimmt welche Verzeichnisse, Dateien und Parameter kompiliert werden.

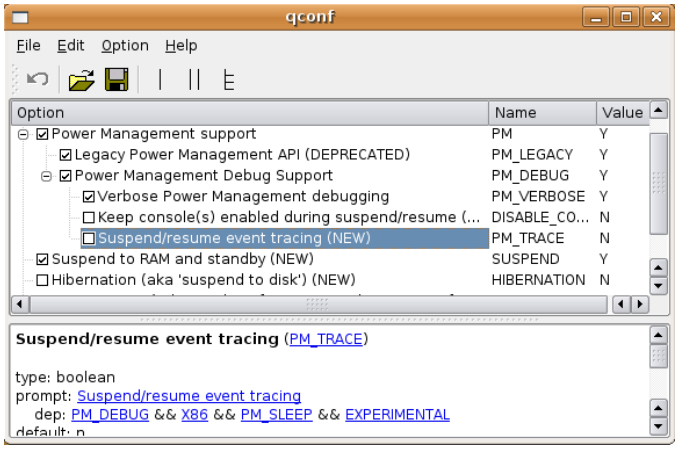
- Ein Regel-basiertes Konfigurationssystem *Kbuild* welches das Feature Modell kodiert und umsetzt.
- Eine Menge von *Makefiles*, über die die Kompilation kontrolliert wird. Diese Dateien hängen von dem Ergebnis der Konfiguration mittels Kbuild ab.
- Eine Menge von *Präprozessor Anweisungen*, die Quellcodezeilen entfernen, hinzufügen oder modifizieren können.
- Eine Menge von *Laufzeitparametern* für Module und den Kern des Betriebssystems.

In einem typischen Konfigurationsprozess wählt ein Entwickler zunächst die Hardwarearchitektur aus (beispielsweise über `make ARCH=i386 config`). Dieser Parameter wird dem Kbuild Programm mitgeteilt welches daraufhin das Feature Modell einschränkt. Aus allen noch möglichen Konfigurationen kann der Nutzer Features auswählen oder abwählen. Hierbei werden die Regeln, die mögliche korrekte Konfigurationen beschreiben, auf Einhaltung geprüft. Die Regeln sind meist aussagenlogischer Natur und in der Sprache *Kconfig* geschrieben. Häufig drücken die Regeln Abhängigkeiten zwischen Features aus: *Feature A hängt von Feature B ab.* Zusätzlich können vorgegebene Werte, Wertebereiche und Typinformationen angegeben werden. Problematisch ist die Verwendung von rückwärtigen Abhängigkeiten: *Wird Feature A gewählt, dann soll automatisch Feature B deaktiviert werden.* Das Problem besteht in der Tatsache, dass rückwärtige Abhängigkeiten, falls sie unabhängig von vorwärtsgerichteten Abhängigkeiten geprüft werden, eine Inkonsistenz zwischen beiden Regelbasen einführen können. Beide obigen Regeln zusammen lassen keine gültige Konfiguration zu. Wird die rückwärtige Abhängigkeit zuletzt und unabhängig ausgewertet, würde Feature B inaktiv sein und Feature A aktiviert sein. Dies widerspräche der ersten Regel.

Eine detaillierte Beschreibung der Sprache und des obigen Problems findet in den Abschnitten 6.3.2 und 6.4.1 statt.

Als nächster Schritt wird Linux mittels `make` kompiliert. `make` verwendet hierzu die Kbuild Ausgaben und den Architekturparameter[2] um festzustellen, welche Quelldateien und Verzeichnisse bearbeitet werden sollen. Der GNU C Präprozessor modifiziert während der Vorverarbeitung die Quelldateien auf der Basis der konfigurierten Features. Die Information wird von Kbuild über eine generierte C Datei mit Präprozessordefinitionen zur Verfügung gestellt. Abbildung 6.4 enthält ein Beispiel wie Kbuild die Kompilation beeinflusst.

[2]Der *Architekturparameter*, also etwa *Intel Prozessor mit 32 Bit (i386)* oder *Solaris* ist die einzige Festlegung, die schon vor Anwendung der eigentlichen Konfigurationssoftware stattfindet. Sie hat also Einfluss auf das gewählte Featuremodell.




```
// Kernel: 2.6.22
// Datei:   net/wireless/bcm43xx/bcm43xx_main.c
// Zeile:  4247-4256
static struct pci_driver bcm43xx_pci_driver = {
        .name = KBUILD_MODNAME,
        .id_table = bcm43xx_pci_tbl,
        .probe = bcm43xx_init_one,
        .remove = __devexit_p(bcm43xx_remove_one),
#ifdef CONFIG_PM
        .suspend = bcm43xx_suspend,
        .resume = bcm43xx_resume,
#endif
};
```

Abbildung 6.4: Kbuild bietet ein graphische Anwendung mit der Features ausgewählt werden können. Im Bild kann das Feature `PM` "*power management support*" an- und abgeschaltet werden. Falls `PM` aktiv ist wird eine Präprozessordefinition `CONFIG_PM` erzeugt. Die Schnittstelle `bcm43xx_pci_driver` im unteren Teil implementiert die Operationen `suspend` und `resume` nur falls `PM` aktiviert wurde.

6.3.1 Ein Beispiel zur Motivation

Zur Veranschaulichung der Problematik der Verifikation von konfigurierbaren Systemen erweitern wir das Beispiel aus Abbildung 6.4. Das Szenario enthält ein Modul welches von dem Feature `PM` abhängt. Bei der Initialisierung der Schnittstelle `bcm43xx_pci_driver` werden die `suspend` und `resume` Operationen nur benötigt, falls das Energiesparsystem vorhanden ist, das heißt falls die Präprozessordefinition `CONFIG_PM` vorhanden ist. Ein zweites Modul (siehe Abbildung 6.5) nutzt diese Schnittstelle und ist folglich auch von `PM` abhängig. Um Korrektheit zu gewährleisten müssen folgende Bedingungen gelten:

1. `suspend` und `resume` dürfen nicht aufgerufen werden, falls sie nicht initialisiert wurden.

2. Zwischenlösungen, in denen nur eine der beiden Operationen initialisiert sind, dürfen nicht auftreten.

Darüberhinaus müssen `suspend` und `resume` genau dann initialisiert werden, falls `PM` aktiv ist. In dem zweiten Modul in Abbildung 6.5 gelten alle diese Bedingungen. Folglich ist die – durch Präprozessoranweisungen geschützte – Verwendung der Operationen sicher.

```
extern struct pci_driver
           bcm43xx_pci_driver;
void restart_device() {
  #ifdef CONFIG_PM
    bcm43xx_pci_driver.suspend();
  #endif
  #ifdef CONFIG_PM
    bcm43xx_pci_driver.resume()
  #endif
}
```

Abbildung 6.5: Die Verwendung von Operationen, die in einigen Konfigurationen nicht definiert sind, muss durch geeignete Präprozessor Direktiven eingeschränkt werden.

6.3.2 Konfigurations Lifting für Linux

In diesem Abschnitt wenden wir Lifting auf die einzelnen Konfigurationsschritte von Linux an.

Generelle Einführung

Zuerst wird das Feature Modell, welches durch die Kbuild Regeln definiert wird, als Programmteil in C kodiert. Abbildung 6.3 enthält diesen Teil des Metaprogramms. Nach diesem Schritt können Probleme der Kategorie D1 (siehe Seite 141) bereits analysiert werden.

Nachfolgend beschreiben wir, wie man den Effekt der Makefiles in C ausdrücken kann. So können auch Probleme der Klasse D2 gefunden werden: *Gibt es Konfigurationen in denen Funktionen aufgerufen werden, die nicht definiert sind.*

Zuletzt werden die Präprozessoranweisungen eliminiert. Auf diese Weise entsteht ein präprozessorfreies ANSI-C Programm, das von Werkzeugen wie dem Software Bounded Model Checker CBMC [CKL04] untersucht werden kann. CBMC, unter Eingabe des Metaprogramms, prüft somit Probleme der Klasse D3, das heißt, ob es überhaupt Varianten gibt in denen Laufzeitfehler vorkommen können (D3).

Lifting von Kbuild

Wie bereits erwähnt definiert Kbuild ein Feature Modell in einer Sammlung von Kconfig Dateien. Enthalten sind eine Menge von Features und regelbasierte Einschränkungen, also Abhängigkeiten. Alle diese Dateien werden wie folgt nach C übersetzt:

Features. Konfigurationseinheiten, also Features, werden über die `config <identifier> <type> <attribute>` Direktive in Kconfig deklariert. Jedes Feature kann einen der Typen `bool, tristate` oder `hex` annehmen. Im Folgenden beschränken wir die Möglichkeiten auf Boolesche Features, da sie fast alleinig Verwendung finden. Falls ein Feature nicht aktiv ist, erhält die Variable den Wert 'n', `false` beziehungsweise null zugewiesen. Sei `A` ein Feature, welches nicht aktiv ist, dann ist die Abfrage `#ifdef CONFIG_A` in allen C Dateien in Linux immer falsch. Der Präfix `CONFIG_` dient in Kbuild zur Vermeidung von Namensraumproblemen mit bestehendem Programmcode. Durch Abfragen an den Präprozessor wird bedingte Kompilation realisiert.

Um diesen Effekt zu modellieren, führen wir für jedes Feature eine neue C Variable ein. Folglich kodiert die Menge der neu eingeführten C Variablen den Raum aller möglichen und unmöglichen Konfigurationen (siehe Abbildung 6.3).

Konfigurationsregeln. *Feature `A` hängt von einem Feature `B` ab.* Solche und andere Abhängigkeiten werden in Kconfig durch die Schlüsselwörter `depends [on]` und `requires` formalisiert. Zusätzlich kann über das Wort `range` der gültige Bereich für eine Featurevariable angegeben werden, falls es sich nicht um eine Boolesche Variable handelt. Über das Schlüsselwort `choice` kann spezifiziert werden, dass genau eine von einer Menge von Features ausgewählt werden muss.

Beispiel 3 *Eine einfache Kconfig Datei, die zwei sich gegenseitig ausschließende Features beschreibt:*

```
config A depends on !B
config B depends on !A
```

Eine konkrete Konfiguration, in der `A` und `B` aktiv sind, wird durch obige Regeln ausgeschlossen. Statt der obigen Formulierung kann auch das Schlüsselwort `choice` verwendet werden.

Die genaue Semantik der anderen Schlüsselwörter ist selbsterklärend, deshalb sei auf die offizielle Dokumentation verwiesen [Vara]. Abbildung 6.6 enthält eine Rekonstruktion der Syntax der Sprache.

Neben den Deklarationen und den bisher genannten Restriktionen können in Kconfig auch vorgegebene (*default*) Werte definiert werden. Diese werden bei unseren Experimenten im Folgenden berücksichtigt.

Lifting von Makefiles

Makefiles steuern den Kompilationsprozess, indem festgelegt wird, welche Quelldateien in welcher Ordnung kompiliert und gelinkt werden. Makefiles selbst hängen wiederum von Featurevariablen ab wie folgendes Beispiel zeigt:

Beispiel 4 *Das Verzeichnis `feature_A/` soll nur dann kompiliert werden, falls eines der Features `A1` oder `A2` aktiv ist. Ein Makefile, das genau dies kodiert, sieht in Linux folgendermaßen aus:*

```
obj-$(CONFIG_A1) += feature_A/
obj-$(CONFIG_A2) += feature_A/
```

Ist eines der Features `A1` und `A2` aktiv, evaluiert `$(CONFIG_A1)` zu wahr 'y'.

Da die Konfiguration nach Lifting erst zur Laufzeit bestimmt wird, muss potentiell jede mögliche Quelldatei im Lifting inbegriffen werden. Wir berechnen für jede Quelldatei die Bedingung unter der sie kompiliert würde.

```
<start> ::=  <start> <start> |
  <config> | <choice> | <menu> | <if>

<identifier> ::= 'a name'
<constant> ::= 0 | 1 | 2 | n | m | y
<symbol> ::= <constant> | <identifier>

<expr> ::=
  <symbol>              |
  <symbol> = <symbol>   |
  <symbol> != <symbol>  |
  ( <expr> )            |
  !<expr>               |
  <expr> && <expr>      |
  <expr> || <expr>

<type> ::= bool | tristate | hex | string

<attribute> ::=           |
 <attribute> <attribute> |
 <attribute> if <expr>   |
 <dependency> | <requirement> | <selection> |
 <default> | <range>

<dependency> ::=  |
  depends <expr> |
  depends on <expr>
<requirement> ::= requires <expr>
<selection> ::= selects <identifier>
<default> ::= default <constant>
<range> ::= <symbol> <symbol>

<config> ::=
  config <identifier> <type> <attribute>

<choice> ::=
 choice <attribute> <config> endchoice

<menu> ::=
  menu <symbol> <dependency>
       <start> endmenu |
  menu <symbol> <dependency> if <expr>
       <start> endmenu

<if> ::= if <expr> <start> endif
```

Abbildung 6.6: Die Syntax und Semantik der Kconfig Sprache sind nur informell definiert. Der obige Ausschnitt ist eine (unvollständige) Rekonstruktion der Grammatik.

Im obigen Beispiel ist dies `A1 || A2`. Solche Bedingungen werden benötigt um Problemklasse D2 zu untersuchen (siehe Abschnitt 6.4.2).

Einige Implementierungseinschränkungen Die Sprache die `Make` zugrundeliegt erlaubt es Kompiler-Parameter zu definieren oder zu modifizieren. Wir bestimmen die Parameter für jede Architektur manuell und fügen sie später nach der Extraktion der Makefileinformationen hinzu.

Selten enthalten Makefiles auch Funktionen. Diese werden zur Zeit nicht berücksichtigt.

Lifting von Präprozessoranweisungen

Der C Präprozessor ändert den Quellcode eines Programms vor der eigentlichen Übersetzung in Maschinencode. Kbuild nutzt dies um mittels generierter Präprozessordateien die Ausgabe der Präprozessorphase im Compiler zu ändern.

Beispiel 5 *Durch den Präprozessor kann jedes C Programm `A` in jedes andere Programm `B` übersetzt werden:*

```
#ifdef CONFIG_A
   variant_A
#elseif
   variant_B
#endif
```

In der Praxis werden jedoch nur kleine Modifikationen – wie Änderungen von Typen, bedingte Ausführung einzelner Befehle oder Initialisierungen – verwendet.

Präprozessor Regionen Um den Zusammenhang zwischen Kbuild Regeln, Makefiles und Präprozessor zu verstehen führen wir das Konzept der *(Präprozessor) Region* ein.

Definition 18 *Eine* Präprozessor Region *ist eine größte zusammenhängende Region von Quellcodezeilen, die keine Präprozessoranweisung enthält.*

Eine '.c' und '.h' Datei in Linux enthält also mindestens eine meist aber dutzende Präprozessorregionen. Jede Region enhält nur reinen C Code. Präprozessoranweisungen sind beispielsweise die Anweisungen `#if`, `#ifdef`, `#else` und `#endif`. Für jede Region berechnen wir die genaue Bedingung (*Guard*)

unter der sie aktiv ist. In dem Guard sind neben den verschachtelte Präprozessorbedingungen auch die Bedingungen von Makefiles enthalten. Dies ist am besten an einem Beispiel nachzuvollziehen:

Beispiel 6 *Eine Datei in dem Verzeichnis* `feature_A/` *enthält folgenden C Code:*

```
#ifdef CONFIG_I
  int j;
  int i;
#endif
```

Der Auszug enthält eine (nicht-leere) Region. Diese enthält zwei Deklarationen, die genau dann aktiv sind, falls auch die Region aktiv ist. Aus dem Beispiel 4 wissen wir, dass das Verzeichnis von `make` *nur inkludiert wird, falls* `CONFIG_A1` *oder* `CONFIG_A2` *aktiv ist. Hinzu kommt die Bedingung die im Präprozessorbefehl* `#ifdef` *genannt wird. Der Guard für die Region ist insgesamt* `(CONFIG_I` && `(CONFIG_A || CONFIG_A2))`.

Regionen und ihre Bedingungen können effizient berechnet werden durch Traversierung des abstrakten Syntaxbaums der C Datei. Nachdem alle Bedingungen aller Regionen berechnet worden sind, kann nun mit der Entfernung von Präprozessoranweisungen begonnen werden.

Übersetzen von Regionen Betrachten wir Anweisungen und Deklarationen als die kleinste konfigurierbare Einheit: Immer wenn ein Ausdruck innerhalb einer Deklaration oder einer Anweisung konfigurierbar ist, betrachten wir die ganze Anweisung oder die Deklaration als konfigurierbar. Die häufigsten Konfigurationen ermöglichen es, die Ausführung von Anweisungen, Definitionen, Deklarationen, Typen oder Initialisierungen zur Kompilezeit zu ändern. Wir zeigen nun exemplarisch für einige Fälle die Umwandlung für ein Metaprogramm, so dass der Effekt der Konfiguration in C kodierbar ist.

Anweisungen Jede Anweisung mit einem nicht-leeren Guard wird in eine neue `if` Anweisung eingebettet. Der Guard wird hierzu nach C übersetzt und in die `if` Bedingung eingefügt.

Beispiel 7 *Anweisungen werden nur ausgeführt, falls die C-Übersetzungen der Guards zu* `TRUE` *auswerten:*

```
// Originales Programm
#ifdef A
i++;
#endif
i = #ifdef B
       1;
    #elseif
       0;
    #endif
```

```
// Metaprogramm
if (A) {
   i++;
}
if (B) {
   i = 1;
} else {
   i = 0;
}
```

Deklarationen. Eine Deklaration ist konfigurationsabhängig falls sie in einer Region steht, die einen Guard ungleich `TRUE` besitzt. Im Lifting versehen wir den Namen des Symbols mit einem neuen Postfix. Zusätzlich wird jede direkte Referenz auf diesen Namen durch eine indirekte Referenz (in der Regel ein Zeiger) ersetzt. Die Initialisierung des Zeigers ist abhängig von den aktiven Features.

Beispiel 8 *Konfigurierbare Deklarationen benötigen eine globale Umbenennung des deklarierten Namens:*

```
// Originales Programm
#ifdef A
void open_A() {
  ...
}
#endif
int main() {
  open_A();
}
```

```
// Metaprogramm
void open_A__1() {
  ...
}

void (*open) ();

int main() {
  if (A) {
    open = &open_A__1;
  }
  open();
}
```

Deklarierte Objekte können nun im Metaprogram referenziert werden.

Anwendung auf historische und kooperative Versionen Der Guard hat bisher die logischen Dimensionen sowie die Feature Modell und Makefile Bedingungen umfasst. Zur Erweiterung auf historische und kooperative Dimensionen kann der Guard einfach um neue Variablen und Abhängigkeiten erweitert werden. Ist eine Deklaration nur in einer Konfiguration enthalten, hängt sie davon ab, dass die Konfiguration ausgewählt wird. Ist eine Deklaration nur in einer Version einer Software enthalten, kann dies durch das gleiche Prinzip kodiert werden. Hierzu müssen neue Variablen eingeführt werden die die Versionsnummer als Feature kodieren. Im Prinzip können also die obigen Dimensionen als reine Erweiterung des Feature Modells gesehen werden.

6.4 Verifikations-Resultate

In den vergangenen Abschnitten wurde ausgeführt wie Lifting konkret für Linux umgesetzt werden kann. Als Zwischenresultate stehen immer C Programme die sich eignen um Probleme der drei Problemklassen (aus Abschnitt 6.1) zu prüfen. Da zumindest die Regeln des Feature Modells eine fast ausschließlich Boolesche Struktur besitzen, scheint die Anwendung eines SAT-basierten Verifikationswerkzeugs wie CBMC [CKL04] sinnvoll. Zu erwähnen ist aber, das das Metaprogramm von allen Werkzeugen, die die Analyse von C Quellcode unterstützen, behandelt werden kann.

Für die folgende Diskussion der Problemklassen ist jedes Feature durch eine C Variable kodiert (siehe Abschnitt 6.3.2).

Beispiel 9 *Das folgende Programm repräsentiert das Feature Modell für zwei sich gegenseitig ausschließende Features* `A` *und* `B` *(wie in Beispiel 3):*

```
// Feature Variablen
int A; // A und B können jeden Wert annehmen
int B;
// Feature Abhängigkeiten
if (!A && !B)) assume(0); // A depends on !B
if (!B && !A)) assume(0); // B depends on !A
// Beweisverpflichtung: A und B schließen sich aus
assert(!(A && B));
```

Der obige Auszug wird ausgeführt bevor das Programm ausgeführt wird. Falls eine der Abhängigkeiten verletzt ist, wird das Metaprogramm beendet ohne dass ein Fehler ausgelöst wird. Die `assert` *Anweisung prüft, dass die Features sich in der Tat gegenseitig ausschließen. Die* `assume` *Anweisungen bedingen, dass bei ungültigen Konfigurationen das Programm ohne Fehler beendet wird. Ohne die Abhängigkeiten würde CBMC eine mögliche Verletzung melden.*

6.4.1 Problemklasse D1

Nach der Übersetzung des Feature Modells können beliebige Eigenschaften auf diesem geprüft werden. Folgende Anfragen sind sinnvoll:

- Existiert mindestens eine gültige Konfiguration?
- Ist jedes Feature in mindestens einer Konfiguration aktivierbar und in mindestens einer Konfiguration deaktivierbar?
- Existieren äquivalente, das heißt redundante, Feature Variablen?

- Kann ein Feature aktiviert werden obwohl seine Abhängigkeiten nicht erfüllt sind? Probleme dieser Art sind möglich aufgrund der Implementierung der umgekehrten Abhängigkeiten in Kbuild.

Die ersten drei Anfragen scheinen entweder trivialerweise testbar oder nicht sicherheitskritisch. Im Folgenden wird deshalb nur die vierte Eigenschaft geprüft. Kbuild erlaubt, dass ein Feature `A` bei Aktivierung eines Features `B` aktiviert wird falls bei `B` die Regel `selects A` Verwendung findet. Diese Aktivierung findet ohne erneute Prüfung der Abhängigkeiten von `A` statt, das heißt `select` kann die gewählte Konfiguration in eine ungültige umwandeln ohne dass dies erkannt wird.

Ein `awk` Skript übersetzt die Regelbasis für jede Hardwarearchitektur einzeln nach C. Das resultierende Programm, dass nur die Wahl- und Abhängigkeitsprüfung enthält, kann mittels CBMC [CKL04] auf Erreichbarkeit eines ungültigen Zustands und damit einer ungültigen Konfiguration geprüft werden.

Die Modellierung der beiden Typen von Abhängigkeiten geht leicht über die aussagenlogischen Regeln hinaus, da durch das Anwählen und Abwählen von Features Zustände eingeführt werden. Uns interessiert jedoch nur die Frage, ob eine Ausführung einer Aktivierung mittels `select` ein Problem verursachen kann. Die Modellierung ist wie folgt (siehe Abbildung 6.3):

1. Feature Variablen werden mit beliebigen Werten gesetzt. Dies kann erreicht werden durch Verwendung lokaler Variablen ohne Initialisierung oder auch durch den Einsatz der reservierten Funktionen `nondet_` in CBMC [CKL04].

2. Falls mindestens eine Abhängigkeit durch die obige Wahl verletzt wurde, terminiert das Programm ohne Fehlermeldung um Ausführungen des Metaprogramms unter ungültigen Konfiguration auszuschließen. Nach diesem Punkt ist die noch immer unbekannt Konfiguration, aber gültig (bezogen auf die `depends` und `requires` Direktiven).

3. Ausgehend von der unbekannten, aber festen, Konfiguration werden alle `select` Anweisungen ausgeführt, falls das Feature, welches das `select` enthält, aktiv ist.

4. Nachfolgend wird erneut geprüft ob die neue Konfiguration alle Abhängigkeiten erfüllt.

CBMC prüft, ob alle Abhängigkeiten erfüllt sind und falls eine dieser verletzt ist, gibt das Werkzeug ein Gegenbeispiel aus. Aus diesem kann abgelesen werden wie die initale Konfiguration aussieht und welches `select` zu

Tabelle 6.1: Analyse von Problemen durch rückwärtige Abhängigkeiten (Problemklasse D1) für die i386 Architektur (Linux Version 2.6.23-rc3).

Features	4675
Abhängigkeiten	3640
Rückwärtige Abhängigkeiten	1830
Variablen in SAT Formel	142004
Klausen in SAT Formel	280629
Anzahl von Zuweisungen im Programm	15233
Laufzeit	108s

einer Verletzung der Abhängigkeit geführt hat. Die Analyse führte zu der Entdeckung eines Fehlers:

Fehler 1 *(Linux 2.6.23-rc3) Das Feature* `HOTPLUG_CPU` *kann aktiviert werden, obwohl seine Abhängigkeiten nicht erfüllt sind. Auf x86_64 Architekturen führt die Deaktivierung des Features* `EXPERIMENTAL` *und einer anschließenden Aktivierung von* `SUSPEND_SMP` *zu einer Aktivierung mittels* `select` *von* `HOTPLUG_CPU`. `HOTPLUG_CPU` *hängt aber von* `EXPERIMENTAL` *ab, so dass die erlangte Konfiguration nicht gültig, aber trotzdem erreichbar ist.*

Statistiken über Laufzeiten und Problemgröße sind in Tabelle 6.1 aufgelistet. Obwohl das Featuremodell 4675 Boolesche Variablen umfasst konnte die Analyse von Abhängigkeiten und rückwärtigen Abhängigkeiten in weniger als 2 Minuten abgeschlossen werden.

6.4.2 Problemklasse D2

Probleme der Klasse D2 stehen in Zusammenhang mit Kompilationsproblemen einiger Varianten. Genau wie bei der ersten Klasse wählen wir repräsentativ ein Beispielproblem der Klasse, anhand dessen wir das Vorgehen erläutern. Folgendes Problem wurde auf einem der offiziellen Emailverteiler für Linux Entwickler beschrieben: *Einige Funktionen sind nur in einigen Varianten verfügbar. Es ist jedoch nicht sichergestellt, dass sie nur in diesen aufgerufen werden.* Ein Beispiel sind Funktionen die von Energiesparschnittstellen zur Verfügung gestellt werden. Unterstützt die Hardware keinen Energiesparmodus, sind auch die Funktionen nicht definiert.

In solchen Fällen schlägt bestenfalls der Kompilationsprozess fehl. In schlimmeren Fällen entsteht ein Fehler im laufenden Betrieb was zu Datenverlust und Nicht-Verfügbarkeit führen kann.

Im Folgenden bezeichne G_D den Guard für die Definition und G_A bezeichne den Guard für einen Aufruf der Funktion. Für jeden Aufruf muss gewährleistet sein, dass $G_A \Rightarrow G_D$.

Tabelle 6.2: Experiment zur Untersuchung der Guards von Funktionsaufrufen (Problemklasse D2) für die Power PC Architektur (Linux Version 2.6.23-rc3). Die Analyse beeinhaltet die Untersuchung von Makefiles und der Quelldateien in den Unterverzeichnissen `drivers/base` und `drivers/macintosh`.

Quelldateien	68
LOC	≈31000
Nicht-tautologische Bedingungen	50
Features	4495
Makefiles	952
Bedingungen aus Makefiles	10489
Gefundene Fehler	5
Neue Fehler	1
Laufzeit der Makefile Analyse	25s
Laufzeit der Quellkodeanalyse (Regionen)	30s
Erstellung der Verifikationsbedingungen und Laufzeit von *cogent*	10s
Laufzeit der Analyse	≤1s

Für das Experiment kodieren wir wie für D1 das Feature Modell in einem C Programm. Zusätzlich werden alle Makefiles analysiert. Die Bedingungen unter denen Verzeichnisse und Dateien kompiliert werden sind ebenso nach C zu übersetzen. Für Version 2.6.23-rc3 konnten 10489 Bedingungen aus 952 Makefiles in weniger als 25 Sekunden extrahiert werden. Zusätzlich wird für jede Funktionsdefinition und jeden Aufruf der Guard berechnet:

Beispiel 10 *Eine Beispieldatei aus Verzeichnis* `feature_A/`*:*

```
#ifdef CONFIG_A_OUTPUT
1: void A_output(...) {...}
#endif
#ifdef CONFIG_A_INPUT
2:  void A_input(...) {
3:      A_output(...);
4:  }
#endif
```

Die Funktion `A_output()` *wird definiert falls die Datei kompiliert wird und das Feature* `A_OUTPUT` *aktiv ist. Aus Beispiel 4 wissen wir, dass die Bedingung aus dem Makefile* `(A1 || A2)` *lautet. Zusammen ergibt sich* $G_D =$ `(A1 || A2)` && `A_OUTPUT`*. Für* G_A *ergibt sich die Bedingung:* $G_A =$ `(A1 || A2)` && `A_INPUT`*. Da aber* $G_A \Rightarrow G_D$ *nicht ohne weiteres gilt, muss durch Konfigurationsregeln sichergestellt sein, dass* `A_INPUT` *von* `A_OUTPUT` *abhängt. Nur dann ist der Aufruf sicher.*

Die Prädikate G_A und G_D können wie oben direkt nach C übersetzt werden. Falls $G_A \Rightarrow G_D$ für alle Aufrufe gilt, ist das System bezogen auf diese Eigenschaft sicher. In der Analyse wurden 5 Fehler gefunden. 4 dieser Fehler

wurden unabhängig von unserer Arbeit auch von den Entwicklern gefunden. Ein weiterer Fehler wurde gefunden, jedoch ohne Fehlerbeschreibung - möglicherweise zufällig - in späteren Versionen des Kerns entfernt:

Fehler 2 *In Linux 2.6.20.4 und den Verzeichnissen* `drivers/base/power` *findet sich ein Fall in dem die Makefiles unzureichende Guards verwenden.*

```
// Datei power.h
#ifdef CONFIG_PM
...
static inline struct device *
     to_device(struct list_head * entry)
{ ... }
...
#endif

// Datei trace.c
struct device * dev = to_device(entry);

// Makefile Guard
obj-$(CONFIG_PM_TRACE)  += trace.o
```

Die zu beweisende Bedingung für obiges Programm ist $G_A \Rightarrow G_D$*, also* `!PM_TRACE || PM`*. Da für keine Architektur* `PM_TRACE` *von* `PM` *abhängt, ist eine Konfiguration* `PM_TRACE = 0, PM = 0` *möglich. In neueren Versionen des Linux Kerns (2.6.23-rc3) findet sich die notwendige Abhängigkeit. CBMC findet diesen Fehler in der alten Version und bestätigt die Tatsache, dass das Problem in neueren Version behoben wurde.*

Um Paare von Aufrufen und Definitionen, die überprüft werden müssen, zu reduzieren, wird der Beweiser cogent [CKS05] eingesetzt. cogent identifiziert Fälle in denen die Bedingung $G_A \Rightarrow G_D$ schon ohne die Konfigurationsinformation tautologisch ist. Dies ist ein häufiger Fall da meist Definitionen und Aufrufe exakt den identischen Guard verwenden. In mehr als 99% der Fälle konnte cogent die konfigurationsunabhängige Korrektheit direkt beweisen. Jeder Aufruf war dabei in Millisekunden erfolgreich. In Tabelle 6.2 finden sich weitere Daten über das Experiment.

6.4.3 Problemklasse D3

Problemklasse D3 ist das eigentliche Ziel dieses Kapitels. Nachdem die Vorarbeiten in den letzten Abschnitten gemacht wurde, kann nun die Verifikation von C Code unter Berücksichtigung aller Konfigurationseffekte realisiert werden. Hierzu demonstrieren wir die Effektivität indem wir ein Linux Modul, `sound/oss/ad1848.c`, in ein Metaprogramm umwandeln und anschließend

Tabelle 6.3: Laufzeiten und Problemgrößen der Verifikation von Metaprogrammen mit CBMC. Inkrementelles Lifting von Features führt zu keinem signifikanten Einfluß auf die Laufzeit für das Modul `ad1848.c`.

Feature	Varianten	Laufzeit	Zuweisungen	Variablen	Klauseln
Minimale Variante	1	71,0s	54532	8.079.480	13.319.552
+`MODULES`	2	74,6s	54535	+1	+3
+`DEBUGXL`	4	75,6s	54725	+25657	+81983
+`EXCLUDE_TIMERS`	8	75,4s	54728	+3	-295
+`CONFIG_SMP`	16	72,9s	54822	+117.704	+185.367
+`CONFIG_PNP`	32	73,0s	55327	+4	+13975

mit CBMC auf Laufzeitfehler untersuchen. Das Metaprogramm bleibt durch einen normalen Compiler kompilierbar, die Ausführung hängt jedoch von den Variablen ab, die die gewählte Konfiguration kodieren.

Der Treiber enthält fünf Boolesche Features die voneinander unabhängig sind. Dies führt zu 32 möglichen Varianten. Um den Effekt des Liftings auf die Laufzeit und die Problemgrößen zu messen starten wir mit der Variante, die das kleinste Programm kodiert. Anschließend werden weitere Features umgewandelt, so dass mit jedem neuen Feature die Zahl der im Metaprogramm kodierten Varianten verdoppelt wird. Insgesamt ist der Effekt des Liftings in 56 neuen `if` Anweisungen kodiert. Typen und Deklarationen in dem Programm sind nicht konfigurationsabhängig.

Jedes der Zwischenprodukte des Liftings wird mit einem kleinen Umgebungsmodell versehen, das einen typischen Lebenszyklus dieses Moduls implementiert. CBMC [CKL04] überprüft anschließend ob es Varianten gibt, die Laufzeitfehler enthalten können. Typische Fehler sind falsche Indizes bei Arrayzugriffen, Division durch null und ungültige Zeigerzugriffe. Für das Experiment wurde eine Rekursions- und Scheifenbeschränkung (*Bound*) von eins verwendet. Die Architektur wurde als 64 Bit angenommen. Tabelle 6.3 gibt Laufzeiten und Größen der von CBMC generierten Probleminstanzen wieder.

Das Ergebnis des Experiments ist die Bestätigung der Vermutung, dass für Programme bei denen die Varianten sich nur marginal unterscheiden, die Verifikation des Metaprogramms fast mit gleicher Laufzeit wie bei einer einzelnen Variante möglich ist. Zu beachten ist, dass sich durch inkrementelles Hinzufügen von Feature Variablen die Laufzeit scheinbar nur zufällig ändert. Diese Varianz der Laufzeit kann schon allein mit der Varianz der Laufzeit des im CBMC verwendeten Erfüllbarkeitsprüfers Minisat [ES03] erklärt werden.

6.5 Literaturhinweise und Diskussion

6.5.1 Verwandte Arbeiten

In diesem Kapitel werden die Bereiche Software Konfiguration und Software Verifikation miteinander verschmolzen. Nach bestem Wissen ist dies der erste Ansatz der die Verifikation von mehreren Varianten einer Software behandelt und systematisiert. Andere Studien im Bereich Software Model Checking, beispielsweise von Chen et al. [CDW04], erwähnen explizit, dass nur eine Variante der Software untersucht wird.

Im Bereich der Software Konfiguration gibt es vielfältige Literatur [SKKM03, NEF01] jedoch ist die Kombination mit Quellcodeverifikation bisher unbehandelt.

Schirmeier und Spinczyk [SS07] behandeln die Nutzung von Methoden der statischen Analyse zur Generierung und Komplementierung von Feature Modellen. Die ist in dreifacher Hinsicht von unserem Ansatz zu unterscheiden. Zum Ersten ist der Prozess umgekehrt, das heißt die Analyse generiert Feature Modelle, anstatt Feature Modelle bei der Verifikation zu verwenden. Des Weiteren ist die verwendete Methode weder vollständig noch mit Sicherheit schlüssig da sie zu einem großen Teil auf Heuristiken zur Erkennung möglicher Features im Quellcode beruht. Zum Dritten ist die Art der behandelten Probleme mit Problemklasse D2 zu vergleichen.

Das von uns vorgestellte Lifting kann mit jeder Art von Verifikationstechnik kombiniert werden.

6.5.2 Einschränkungen

Die bisherige Umsetzung von Lifting ist noch zweifach eingeschränkt: Zum Einen ist nur ein Modul auf das Laufzeitverhalten bei der Verifikation von Metaprogrammen untersucht worden. Zum Anderen ist die generelle Darstellung von Lifting allgemein gehalten. Da aber bisher eine gemeinsame Basis für die Implementierung von Software Produktlinien fehlt konnte nur eine konkrete Umsetzung für die momentan gängigste Technik, die bedingte Kompilation, gegeben werden. Eine Kombination mit anderen Verifikationsmethoden wie abstrakter Interpretation, Model Checking oder deduktiver Verifkation ist zwar angesprochen [DN05], aber nicht experimentell evaluiert worden.

Wie bereits erwähnt können für Linux Funktionen in Makefiles nicht behandelt werden.

6.6 Zusammenfassung des Kapitels

Das Lifting von Software Konfigurationen ist eine neue Technik die notwendig zur Verifikation industrieller Software ist, sobald es sich nicht um klassische, nur zur Compilezeit konfigurierbare Software handelt. Es wurde gezeigt, dass Lifting erfolgreich auf das Linux Betriebssystem angewendet werden kann. Das Vorhandensein verschiedener Ebenen von Konfiguration (Kbuild, Makefiles und Präprozessor) ist kein Hindernis, sondern im Gegenteil eine Möglichkeit Lifting auf jeder Ebene zu betreiben. In den Experimenten konnten Inkonsistenzen zwischen Feature Model Abhängigkeiten und Inkonsistenzen zwischen Feature Model und Implementierung gefunden werden. Als bedeutendster Erfolg ist die Realisierung des Software Bounded Model Checkings eines Metaprogramms zu nennen, welches 32 Varianten in einem Programm kodiert. Die Laufzeit für dieses Beispiel ist in der Tat 32 mal kleiner als die Summer der Laufzeiten für eine naive Einzelprüfung jeder einzelnen Variante.

Die Linux Software Familie enthält mehr als 4600 Features. Trotzdem konnten alle Experimente ohne besondere Optimierungen innerhalb von Minuten ausgeführt werden. Die Methode ist folglich für den Einsatz in industriellen Bereichen als geeignet anzusehen.

Zusammenfassung 7

Im Rahmen der Kapitel 3 bis 6 wurde die Technik des Software Bounded Model Checking zur Verifikation von drei verschiedenen Softwaresystemen eingesetzt. Die Fallstudien umfassen einige tausend Zeilen lange, aber hoch komplexe Programme, wie eine AES Implementierung, und große offene Systeme, wie das Linux Betriebssystem. Zusätzlich wurde der Bereich der eingebetteten Software untersucht. Die Fallstudien liefern somit umfassend Daten zur *Anwendbarkeit* der Methode des Software Bounded Model Checking.

Bei der Ausführung der Fallstudien konnten die Probleme der *Einsetzbarkeit* isoliert werden: Große Softwaresysteme sind aufgrund der Schranke des Software Bounded Model Checking nur eingeschränkt verifizierbar. Zudem ist die Dimension der Compilezeitkonfiguration in der Literatur bisher nicht betrachtet worden. Existieren mehrere Varianten einer Software müssen sie bisher alle einzeln geprüft werden. Beide Probleme werden in der Arbeit behandelt und in den Kapiteln 5 und 6 gelöst.

Die Frage ob Software Bounded Model Checking einen Beitrag zur Qualität, beziehungsweise zum Vertrauen in die korrekte Implementierung, eines Programms leisten kann, wird also durch die drei Fallstudien und die zwei Problembehandlungen beantwortet:

Fallstudien mit SBMC. Es ist schwierig eine automatische Technik zur Verifikation zur Anwendung zur bringen. Bisher wurde nicht versucht Software Bounded Model Checking auf systemnahe, nicht-akademische Software anzuwenden.

Im Rahmen dieser Arbeit wurden Implementierungen des AES Standards, mehrere Versionen des Linux Betriebssystems sowie einige Softwareprodukte aus dem Automobilbereich untersucht. Neben dieser hohen Bandbreite an unterschiedlichsten systemnahen Programmen erstrecken sich die Fallstudien auf eine Vielfalt von untersuchten Spezifikationen von Sicherheitseigenschaften. Die in CBMC [CKL04] eingebaute und gängige Untersuchung von undefiniertem Laufzeitverhalten und möglichen Laufzeitfehlern wurde in allen Fallstudien betrachtet. Darüberhinaus wurde funktionale Äquivalenz für zwei AES Implementierungen gezeigt.

Zusätzlich wurde die Sprache SLICx zur Spezifikation von Schnittstelleneigenschaften aus den Vorgänger SLIC entwickelt. Mit SLICx konn-

ten erstmals Spezifikation und Umgebungsmodell durch die Einführung von Programmmodifikationen durch Spezifikationen verzahnt werden. Durch manuellen Einsatz und die Entwicklung von Transformations- und Automatisierungstechniken konnten die Probleme der Anwendung des SBMC auf große reale Anwendungen realisiert werden. Die Hauptfrage dieser Arbeit, ob Software Bounded Model Checking aus der Domäne der Hardwareverifikation in die Verifikation systemnaher Software erfolgreich zu übertragen ist konnte positiv beantwortet werden.

Kombination von SBMC mit abstrakter Interpretation. Die größte Schwäche des Verfahrens Software Bounded Model Checking ist die Tatsache, dass in Abhängigkeit von der gewählten Schranke (*Bound*) eventuell kein Beweis erbracht werden dann, dass ein System sicher ist.

Wir schlagen einen neuen Prozess vor, der abstrakte Interpretation mit Software Bounded Model Checking kombiniert: Durch abstrakte Interpretation wird ein Großteil des Programms als korrekt bewiesen. Hiernach noch unsichere Zeilen werden durch Software Bounded Model Checking gefiltert. Die Zahl von Warnungen, die weder durch abstrakte Interpretation noch durch Software Bounded Model Checking als korrekt bewiesen werden, ist in Folge wesentlich kleiner als beim Einsatz nur einer Technik. Das Gesamtergebnis bleibt schlüssig und im Vergleich zu reiner abstrakter Interpretation bleiben weniger Restwarnungen, die einer Inspektion bedürfen.

Die Kombination dieser zwei Techniken ist neu und deren Effektivität wurde in Kapitel 5 anhand der Fallstudie zu Bosch-Steuergeräten nachgewiesen.

Verifikation von konfigurierbarer Software. Eine große Beschränkung aller bestehender Verifikationsverfahren ist die eingeschränkte Behandlung von Softwarefamilien, also einer Menge von Varianten, die über bedingte Kompilation abgeleitet werden. Vor der Einführung der *Lifting*-Technik aus Kapitel 6, war Verifikation effektiv auf einzelne Varianten beschränkt. Lifting löst dieses Problem und erlaubt so erstmals schlüssige Verifikation ganzer Software Produkt Familien. Als Nebenprodukte des Verfahrens konnten Fehler in dem Feature Modell des Linux Betriebssystems gefunden werden. Neben anderen gefunden Fehlern konnte beispielhaft der Nachweis erbracht werden, dass für ein Linux Modul die Prüfung von 32 Varianten in gleicher Laufzeit vollzogen werden kann wie die klassische Analyse nur einer Variante.

Durch die vorliegende Arbeit wird Software Bounded Model Checking für den Einsatz auf konfigurierbare, große Programme erweitert. Die Tatsache, dass in allen Fällen Fehler in Implementierungen gefunden werden konnte, die in langen Jahren rigorosen Testens nicht entdeckt wurden, zeigt, dass

eine durch Software Bounded Model Checking untersuchte Software einen höheren Sicherheitsgrad besitzt.

Abkürzungsverzeichnis

AES Advanced Encryption Standard

AI Abstrakte Interpretation

BMC Bounded Model Checking

CTL Computation Tree Logic

DLTS Deterministic Labelled Transition System

LTL Linear Temporal Logik

LTS Labelled Transition System

SBMC Software Bounded Model Checking

SMC Software Model Checking

SMT Satisfiability Modulo Theory

SPL Software Produkt Line

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ABD+02] Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, pages 230–246, London, UK, 2002. Springer.

[Abr96] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

[Aut04] Diverse Autoren. *MISRA-C:2004 - Guidelines for the use of the C language in critical systems.* The Motor Industry Software Reliability Association, 2004.

[Bal97] Helmut Balzert. *Lehrbuch der Software-Technik, Bd. 2: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung.* Spektrum Akademischer Verlag, November 1997.

[BB07] Robert Brummayer and Armin Biere. C32SAT: Checking C expressions. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007*, pages 294–297, Berlin, Germany, July 2007. Springer.

[BBC+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the the 2006 EuroSys Conference*, pages 73–85, New York, NY, USA, 2006. ACM Press.

[BCC+03a] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[BCC+03b] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for

large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer.

[BCM+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90)*, pages 428–439, Philadelphia, Pennsylvania, USA, June 1990. IEEE Computer Society.

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV)*, pages 504–518. Springer, Berlin, 2007.

[BKO+07] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007*, pages 358–372, Braga, Portugal, April 2007. Springer.

[BR01] Thomas Ball and Sririam K. Rajamani. SLIC: A specification language for interface checking. Technical report, Microsoft Research, MSR-TR-2001-21, [21 August 2007], 2001.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symp. on Principles of Programming Languages (POPL), Los Angeles, California, January, 1977*, pages 238–252, Los Angeles, California, January 1977. ACM Press.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In Shmuel Sagiv, editor, *European Symposium On Programming 2005 (ESOP'05)*, pages 21–30, Edinburgh, Scotland, April 2005. Springer.

[CCG+03] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *25th International Conference on Software Engineering (ICSE), Proceedings*, pages 385–395, Washington, DC, USA, 2003. IEEE Computer Society.

[CCK+06] Sagar Chaki, Edmund M. Clarke, Nicholas Kidd, Thomas W. Reps, and Tayssir Touili. Verifying concurrent message-passing C programs with recursive calls. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 12th International Conference, Proceedings*, pages 334–349, 2006.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *Software Engineering (ICSE), 22nd International Conference , Proceedings*, pages 439–448, New York, NY, USA, 2000. ACM Press.

[CDW04] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004*, pages 171–185, San Diego, California, USA, 2004. The Internet Society.

[CE00] K. Czarnecki and U.W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* Addison Wesley, May 2000.

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 25–35, Austin, Texas, January 1989. ACM PRESS.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, Cambridge, MA, USA, 1999.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 168–176, 2004.

[CKS05] Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving for program verification. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005*, pages 296–300, Edinburgh, Scotland, UK, July 2005. Springer.

[CKSY05] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 11th International Conference, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 570–574, 2005.

[CKY03] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. Technical Report CMU-CS-03-126, School of Computer Science, Carnegie Mellon University, 2003.

[CL00] Edmund Clarke and Yuan Lu. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169, London, UK, 2000. Springer.

[Cou07a] P. Cousot. Proving the absence of run-time errors in safety-critical avionics code. In C. Kirsch and R. Wilhelm, editors, *Proceedings of the Seventh ACM & IEEE International Conference on Embedded Sofware, Embedded Systems Week, (EMSOFT 2007)*, pages 7–9, Salzburg, Austria, September–October 2007. ACM press.

[Cou07b] Patrick Cousot. The rôle of abstract interpretation in formal methods. In M. Hinchey and T. Margaria, editors, *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)*, pages 135–140, London, England, UK, September 2007. IEEE Computer Society.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV, Proceedings*, pages 415–418, Seattle, WA, USA, August 2006. Springer.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3 edition, 2005.

[CW02] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, New York, NY, USA, 2002. ACM Press.

[DKV07] Debapratim De, Abishek Kumarasubramanian, and Ramarathnam Venkatesan. Inversion attacks on secure hash functions using satsolvers. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference*, pages 377–382, Lisbon, Portugal, May 2007. Springer.

[DL05] Rob DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[DN05] Dennis Dams and Kedar S. Namjoshi. Orion: High-precision methods for static error analysis of C and C++ programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, Third International Symposium, FMCO 2005*, pages 138–160, Amsterdam, The Netherlands, November 2005. Springer.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[DR98] Joan Daemen and Vincent Rijmen. The block cipher Rijndael. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Proceedings of the The International Conference on Smart Card Research and Applications*, pages 277–284, Louvain-la-Neuve, Belgium, September 1998. Springer.

[DS07] David Delmas and Jean Souyris. Astrée: From research to industry. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007*, pages 437–451, Kongens Lyngby, Denmark, August 2007.

[EA03] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In Michael L. Scott

and Larry L. Peterson, editors, *19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–252, Bolton Landing, NY, USA, October 2003. ACM Press.

[EC95] Jacky Estublier and Rubby Casallas. Three dimensional versioning. In Jacky Estublier, editor, *Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Selected Papers*, pages 118–135, Seattle, Washington, USA, 1995. Springer.

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, pages 502–518, Santa Margherita Ligure, Italy, May 2003. Springer.

[Gla07] Christoph Gladisch. How C differs from Java for symbolic program execution. In Hendrik Tews, editor, *Proceedings, C/C++ Verification Workshop, Oxford, United Kingdom*, Oxford, United Kingdom, July 2007. Technical report ICIS-R07015 of the Radboud University Nijmegen.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[HJL+06] James J. Hunt, Eric Jenn, Stéphane Leriche, Peter Schmitt, Isabel Tonin, and Claus Wonnemann. A case study of specification and verification using JML in an avionics application. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems, JTRES 2006*, pages 107–116, Paris, France, October 2006. ACM Press.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with Blast. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop*, number 2648 in LNCS, pages 235–239, Portland, OR, USA, May 2003. Springer.

[ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[JN94] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.

[JR00] Daniel Jackson and Martin Rinard. Software analysis: a roadmap. In *ICSE 2000, Proceedings of the 22nd International Conference on on Software Engineering*, pages 133–145, Limerick Ireland, June 2000. ACM.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice-Hall, Englewood Cliffs, N.J. :, 1978.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, Louisiana, January 1985. ACM Press.

[LPC+07] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. JML Reference Manual. Available from http://www.jmlspecs.org, October 2007.

[McM03] Kenneth L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003*, pages 1–13, Boulder, CO, USA, July 2003.

[ML07] Jan Tobias Mühlberg and Gerald Lüttgen. BLASTing Linux Code. In Lubos Brim, Boudewijn R. Haverkort, Martin Leucker, and Jaco van de Pol, editors, *Formal Methods: Applications and Technology, 11th International Workshop, FMICS 2006*, LNCS, pages 211 – 226, Bonn, Germany, August 2007. Springer.

[MM00] Fabio Massacci and Laura Marraro. Logical cryptanalysis as a SAT problem. *Journal for Automated Reasoning*, 24(1/2):165–203, 2000.

[MOSS99] Markus Müller-Olm, David A. Schmidt, and Bernhard Steffen. Model-checking: A tutorial introduction. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis, 6th International Symposium, SAS '99*, pages 330–354, Venice,Italy, September 1999. Springer.

[MSF05] Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita. An equivalence checking method for C descriptions based on symbolic simulation with textual differences. *IEICE A: Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E88-A(12):3315–3323, 2005.

[MZ06] Ilya Mironov and Lintao Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In Carla P. Gomes Armin Biere, editor, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference*, pages 102–115, Seattle, WA, USA, August 2006. Springer.

[NEF01] C. Nentwich, W. Emmerich, and A. Finkelstein. Static consistency checking for distributed specifications. In *16th IEEE International Conference on Automated Software Engineering*, pages 115–125, Coronado Island, San Diego, CA, USA, November 2001. IEEE Computer Society.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. Nigel Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002*, pages 213–228, Grenoble, France, April 2002. Springer.

[NN92] H. Riis Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer New York, Inc., Secaucus, NJ, USA, 1999.

[PBG05] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):156–173, 2005.

[PK06] Hendrik Post and Wolfgang Küchlin. Automatic data environment construction for static device drivers analysis (poster abstract). In Michal Young and Premkumar T. Devanbu, editors, *SAVCBS'06 Specification and Verification of Component-Based Systems, Workshop at ACM SIGSOFT 2006/FSE-14*, pages 89–92, Portland, Oregon, USA, November 2006. ACM Press.

[PK07] Hendrik Post and Wolfgang Küchlin. Integration of static analysis for Linux device driver verification. In Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods, 6th International Conference, IFM 2007*, pages 518–537, Oxford, UK, July 2007. Springer.

[Pnu85] A. Pnueli. *In transition from global to modular temporal reasoning about programs*, volume 13 of *NATO Asi Series, Series F, Computer and Systems Sciences*, chapter Logics and Models of Concurrent Systems, pages 123–144. Springer New York, Inc., New York, NY, USA, 1985.

[Pol08] PolySpace Technologies. Polyspace Client / Server for C/C++, Version 4.1.1.6 , 2008, http://www.polyspace.com. online, August 2008.

[PSK08] Hendrik Post, Carsten Sinz, and Wolfgang Küchlin. Towards automatic software model checking of thousands of Linux modules - a case study with Avinux. *Software Testing, Verification and Reliability (to appear)*, 2008.

[QW04] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. In William Pugh and Craig Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004*, pages 14–24, Washington, DC, USA, June 2004. ACM Press.

[RG05] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005*, pages 82–97, Edinburgh, Scotland, UK, July 2005. Springer.

[Riv05] X. Rival. Understanding the origin of alarms in ASTRÉE. In Igor Siveroni Chris Hankin, editor, *Static Analysis, 12th International Symposium, SAS 2005*, pages 303–319, London, UK, September 2005. Springer.

[Sau07] Matthias Sauter. Automatisierung und Integration regelbasierter Verifikation für Linux Gerätetreiber. Diplomarbeit, Universität Tübingen, Deutschland., 2007.

[Sch98] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 38–48, San Diego, CA, USA, January 1998. ACM Press.

[SGG04] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, December 2004.

[SKKM03] Carsten Sinz, Amir Khosravizadeh, Wolfgang Küchlin, and Viktor Mihajlovski. Verifying CIM models of Apache web server configurations. In *3rd International Conference on Quality Software (QSIC 2003)*, pages 290–297, Dallas, TX, USA, November 2003. IEEE Computer Society.

[SS06] Werner (Red. Bearb.) Scholze-Stubenrecht, editor. *Duden, die deutsche Rechtschreibung : auf der Grundlage der neuenamtlichen Rechtschreibregeln*. Dudenverl., Mannheim [u.a.], 24., völlig neu bearb. und erw. aufl. edition, 2006.

[SS07] Horst Schirmeier and Olaf Spinczyk. Tailoring infrastructure software product lines by static application analysis. In *Software Product Lines, 11th International Conference, SPLC 2007*, pages 255–260, Kyoto, Japan, September 2007. IEEE Computer Society.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000*, pages 108–125, Austin, Texas, USA, November 2000. Springer-Verlag.

[SSSPS07] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is the Linux kernel a software product line? In Frank van der Linden and Björn Lundell, editors, *Processdings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, pages 9–12, Kyoto, Japan, September 2007.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[Tur49] Alan M. Turing. Checking a large routine. In *Report on a Conference on High Speed Automatic Computation*, pages 67–69, Cambridge, UK, June 1949. University Mathematical Laboratory, Cambridge University. Inaugural conference of the EDSAC computer at the Mathematical Laboratory, Cambridge, UK.

[Vara] Various authors. Documentation of the "kconfig-language". Distributed with Linux kernel sources under `Documentation/kbuild`. (kernel version 2.6.23-rc3 from `http://www.kernel.org`).

[Varb] Various authors. Linux kernel releases. Available online under http://www.kernel.org.

[WBWK07] Thomas Witkowski, Nicolas Blanc, Georg Weissenbacher, and Daniel Kroening. Model checking concurrent Linux device drivers. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *22nd IEEE International Conference on Automated Software Engineering (ASE)*, pages 501–504, Atlanta, Georgia, USA, November 2007. ACM Press.

[YTEM06] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors.

ACM Transactions on Computer Systems (TOCS), 24(4):393–423, 2006.

Veröffentlichungen

Zeitschriftenpublikationen

[1] Katharina A. Lehmann, Hendrik D. Post, and Michael Kaufmann, Hybrid graphs as a framework for the small-world effect. *Physical Review E*, 73, 056108 (2006).

Abstract. In this paper we formalize the small-world effect which describes the surprising fact that a hybrid graph composed of a local graph component and a very sparse random graph has a diameter of O(ln n) whereby the diameter of both components alone is much higher. We show that a large family of these hybrid graphs shows this effect and that this generalized family also includes classic small-world models proposed by various authors although not all of them are captured by the small-world definition given by Watts and Strogatz. Furthermore, we give a detailed upper bound of the hybrid's graph diameter for different choices of the expected number of random edges by applying a new kind of proof pattern that is applicable to a large number of hybrid graphs. The focus in this paper is on presenting a flexible family of hybrid graphs showing the small-world effect that can be tuned closely to real-world systems.

[2] Hendrik Post, Carsten Sinz and Wolfgang Küchlin, Towards automatic software model checking of thousands of Linux modules - a case study with Avinux *Journal for Software Testing, Verification and Reliability*, 10.1002/stvr.399 (2008).

Abstract. Modular software model checking of large real-world systems is known to require extensive manual effort in environment modelling and preparing source code for model checking. Avinux is a tool chain that facilitates the automatic analysis of Linux and especially of Linux device drivers. The tool chain is implemented as a plugin for the Eclipse IDE, using the source code bounded model checker CBMC as its backend. Avinux supports a verification process for Linux that is built upon specification annotations with SLICx (an extension of the SLIC language), automatic data environment creation, source code transformation and simplification, and the invocation of the verification backend. In this paper technical details of

the verification process are presented: Using Avinux on thousands of drivers from various Linux versions led to the discovery of six new errors. In these experiments, Avinux also reduced the immense overhead of manual code preprocessing that other projects incurred.

Konferenzbeiträge

[1] Hendrik Post and Wolfgang Küchlin, Integration of static analysis for Linux device driver verification. In Davies, J., Gibbons, J., eds.: Integrated Formal Methods (IFM), 6th International Conference, Proceedings, Oxford, UK, Juli 2007, pp. 518–537. (2007).

Abstract. We port verification techniques for device drivers from the Windows domain to Linux, combining several tools and techniques into one integrated toolchain. Building on ideas from Microsoft's Static Driver Verifier (SDV) project, we extend their specification language and combine its implementation with the public domain bounded model checker CBMC as a new verification back-end. We extract several API conformance rules from Linux documentation and formulate them in the extended language SLICx. Thus SDV-style verification of temporal safety specifications is brought into the public domain. In addition, we show that SLICx, together with CBMC, can be used to simulate preemption in multi-threaded code, and to find race conditions and to prove the absence of deadlocks and memory leaks.

[2] Hendrik Post, Carsten Sinz, Alexander Kaiser and Thomas Gorges: Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking, In: 23rd IEEE/ACM International Conference on Automated Software Engineering, Proceedings, L'Aquila, Italien, Sept. 2008, pp. 188–197.

Abstract. Fully automatic source code analysis tools based on abstract interpretation have become an integral part of the embedded software development process in many companies. And although these tools are of great help in identifying residual errors, they still possess a major drawback: analyzing industrial code comes at the cost of many spurious errors that must be investigated manually. The need for efficient development cycles prohibits extensive manual reviews, however. To overcome this problem, the combination of different software verification techniques has been suggested in the literature. Following this direction, we present a novel approach combining abstract interpretation and source code bounded model checking, where the model checker is used to reduce the number of false error reports. We apply our methodology to source code from the automotive industry written in C, and show that the number of spurious errors emitted by an abstract interpretation product can be reduced considerably.

[3] Hendrik Post, and Carsten Sinz: Configuration Lifting: Verification meets Software Configuration. In: 23rd IEEE/ACM International Conference on Automated Software Engineering, Proceedings, L'Aquila, Italien, Sept. 2008, pp. 347–350.

Abstract. Configurable software is ubiquitous, and the term software product line (SPL) has been coined for it lately. It remains a challenge, however, how such software can be verified over all variants. Enumerating all variants and analyzing them individually is inefficient, as knowledge cannot be shared between analysis runs. Instead of enumeration we present a new technique called lifting that converts all variants into a meta-program, and thus facilitates the configuration-aware application of verification techniques like static analysis, model checking and deduction-based approaches. As a side-effect, lifting provides a technique for checking software feature models, which describe software variants, for consistency. We demonstrate the feasibility of our approach by checking configuration dependent hazards for the highly configurable Linux kernel which possesses several thousand of configurable features. Using our techniques, two novel bugs in the kernel configuration system were found.

[4] Hendrik Post and Carsten Sinz: Proving Functional Equivalence of two AES Implementations using Bounded Model Checking. In: 2nd IEEE/ACM International Conference on Software Testing, Verification, and Validation, Proceedings, Denver, Colorado (USA), April 2009, (to appear).

Abstract. Symbolic equivalence checking is a highly successful technique in the hardware domain. The same holds for bounded model checking. Recently, bit-vector bounded model checkers like CBMC have been developed that are able to check properties of (mostly low-level) software written in C. However, using these tools to check equivalence of software implementations has rarely been pursued. In this case study we tackle the problem of proving the functional equivalence of two implementations of the AES crypto-algorithm using automatic bounded model checking techniques. Cryptographic algorithms heavily rely on bit-level operations, which makes them particularly suitable for bit-precise tools like CBMC. Other software verification tools based on abstraction refinement or static analysis seem to be less appropriate for such software. We could semi-automatically prove equivalence of the first three rounds of the AES encryption routines. Moreover, by conducting a manually assisted inductive proof, we could show equivalence of the full AES encryption process.

Begutachtete Workshop-Beiträge

[1] Hendrik Post and Wolfgang Küchlin: Automatic data environment construction for static device drivers analysis. In: SAVCBS'06 Specification and Verification of Component-Based Systems, Workshop at ACM SIGSOFT 2006/FSE-14, New York, NY, USA, November 2006, ACM Press (2006), pp. 89–92

Abstract. Linux contains thousands of device drivers that are developed independently by many developers. Though each individual driver source code is relatively small— 10k lines of code—the whole operating system contains a few million lines of code. Therefore Linux device drivers offer a useful application area for modular analysis.Our finding is that despite the precise modeling of most features of the standard systems programming language C, model checking software verification tools for C fail to provide means for modular analysis of device drivers. We inspected CBMC, SLAM-SDV, MAGIC, BLAST and others and found that a rich additional environment model for every device driver is needed. This model must provide information on out-of-scope initialized pointers and complex data structures. We present strategies to automatically create feasible, bounded data environments for Linux device drivers instead of creating them manually. Our solution differs from general interface generation mechanisms (e.g. CUTE), because is it specialised on bounded model checking of Linux device drivers written in C. Our contribution is a preprocessing step that extends the usability of CBMC for modular Linux device driver analysis.

[2] Hendrik Post, Carsten Sinz and Wolfgang Küchlin: Towards Automatic Verification of Linux Device Drivers. In ProVeCS Workshop Proceedings, TOOLS Europe 2007: Object, Models, Components and Patterns, Zurich, Switzerland, June 2007.

Abstract. Avinux is a tool that facilitates the automatic analysis of Linux and especially of Linux device drivers. The tool is implemented as a plugin for the Eclipse IDE, using the source code bounded model checker CBMC as its backend. Avinux supports a verification process for Linux that includes specification annotation in SLICx (an extension of the SLIC language), automatic data environment creation, source code transformation and simplifiation, and the invocation of the verification backend. We have successfully used Avinux for the automatic analysis of Linux device drivers reducing the immense overhead of manual code preprocessing that other projects incurred.

Lebenslauf

Name	Hendrik Post
Geburtsdatum	26.03.1980
Geburtsort	Münster
Familienstand	ledig
Staatsangehörigkeit	deutsch
Konfession	katholisch

Schulbildung

07/86-07/90	Dechant Wessing Grundschule Hoetmar
08/90-05/99	Gymnasium Laurentianum Warendorf, Abschluss Abitur

Zivildienst

06/90-09/91	Biologische Station der Rieselfelder Münster

Studium

10/00-03/03	Studium der Bio-Informatik an der Universität Tübingen (Vordiplom 09/02)
03/03-10/03	Auslandssemester an der Queensland University of Technology, Brisbane, Australien
11/03-06/05	Studium der Informatik mit Nebenfach Philosophie an der Universität Tübingen (Diplom)

Berufliche Tätigkeit und Stipendien

10/05-02/08	Promotion als Stipendiat der Stiftung der Deutschen Wirtschaft (sdw) an der Universität Tübingen, Lehrstuhl Prof. Küchlin
07/05-09/05	Praktikum bei der IBM Deutschland Research & Development GmbH
02/08-	Wissenschaftlicher Mitarbeiter in der Forschungsgruppe Verifikation trifft Algorithmik der Universität Karlsruhe (TH)