SLAM
Model Checking von C Programmen mithilfe boolescher und kartesischer Abstraktion

Werner A. König

Seminar
Modellierung und Analyse von Softwaresystemen
Universität Konstanz

Dezember 2005

Inhaltsverzeichnis

1 Motivation **3**
1.1 Problemstellung . . . 4
1.2 Anwendungsdomäne Gerätetreiber . . . 5
1.3 Projektablauf . . . 6

2 SLAM **8**
2.1 Verifikationsprozess und Komponenten . . . 8
2.2 SLIC: Spezifikationssprache . . . 10
2.3 C2bp: Boolesche und kartesische Abstraktion . . . 12
2.4 Bebop: Model Checker . . . 14
2.5 Newton: Pfad-Analyse und Prädikatgewinnung . . . 15
2.6 Laufzeit . . . 16

3 Schlussfolgerung **17**

1 Motivation

„Software Fehler verursachen jährlich 59,5 Milliarden US-Dollar Kosten für die US-amerikanische Wirtschaft."
(US National Institute for Standards and Technology, 2002)

„85 Prozent der Abstürze von Microsoft Windows XP werden durch fehlerhafte Gerätetreiber verursacht."
(Robert T. Short, Vice President of Windows Core Technology, 2003)

Diese Feststellungen verdeutlichen den immensen materiellen und immateriellen Schaden, welcher durch fehlerhafte Software bedingt wird. Beispielsweise erleidet Microsoft einen nicht zu vernachlässigenden Imageverlust durch Systemabstürze, obwohl diese mehrheitlich auf Fehler in Gerätetreibern von Drittanbietern zurückzuführen sind.

Heuristische Methoden, wie die Software Inspection, bei welcher projektexterne Entwickler entlang eines definierten Prozesses den bestehenden Quellcode einer Software Schritt für Schritt begutachten, helfen einen gewissen Anteil bestehender Fehler zu identifizieren, sind aber sehr zeit- und kostenintensiv.

Thomas Ball und Sriram K. Rajamani von Microsoft Research initiierten Anfang 2002 das Projekt SLAM[1], im Rahmen dessen ein Tool entwickelt werden sollte, mit welchem automatisch ein in der Programmiersprache C geschriebenes Programm auf Schnittstellenverletzungen hinsichtlich einer externen Bibliothek mittels Model Checking[2] überprüft werden kann [BCLR04].

Die Größe der Programme, von hunderten bis zu mehreren Millionen Zeilen Code, und die inhärente Komplexität der Programmstrukturen erschweren manuelle und automatische Verfahren gleichermaßen. Im nächsten Kapitel werden die vorherrschenden Problemstellungen für automatische Verfahren und die Anwendungsdomäne des Projektes SLAM näher betrachtet.

[1] Der anfänglich als Akronym gedachte Projektname SLAM war laut den Projektverantwortlichen zu schwerfällig zu erklären, weshalb nun die Verwandtschaft zu „slamming the bugs" betont wird.

[2] Model Checking ist ein Verfahren zur automatischen Verifikation von Systembeschreibungen (Modell M) gegen eine Spezifikation S. Für formale Korrektheit muss gelten: $M| = S$.

1.1 Problemstellung

Mithilfe von SLAM sollen temporale Eigenschaften von Systemsoftware, wie Gerätetreiber und Betriebssystemkomponenten überprüft werden. Hierfür wird auf die Kontrolle der Invarianten[3] fokussiert, auf welches Model Checking von Sicherheitseigenschaften reduziert werden kann [BPR01].

Trotz der Fokussierung auf Invarianten müsste theoretisch jeder mögliche Zustand entlang aller möglichen Ausführungspfade des Programms durchlaufen werden, um sicherzustellen, dass keine Verletzung der Spezifikation auftritt. Werden Rekursion oder nicht begrenzte Datentypen verwendet, können auch schon sehr kleine Programme bei einer Überprüfung zu einer so genannten Zustandsexplosion führen, d.h. die zu kontrollierenden Zustände wachsen in ihrer Anzahl exponentiell an und können nicht mehr in annehmbarer Zeit getestet werden.

Ball et al. versuchen dieser Komplexität durch eine Kombination von boolescher und kartesischer Abstraktion entgegen zu wirken. Hierbei wird ein Programm P in ein boolesches Programm BP(P,E) unter Berücksichtigung einer Prädikatmenge E transformiert. Iterativ wird die Abstraktion durch Anpassung bzw. Erweiterung der Prädikatmenge verfeinert.

Eine weitere Problematik stellt die Erstellung der Spezifikation dar, also die Definition von Sicherheitseigenschaften, gegen welche die zu testenden Programme verifiziert werden sollen. Obwohl die Überprüfung weitestgehend automatisch verläuft, müssen die Sicherheitseigenschaften manuell für die jeweilige Anwendungsdomäne bzw. Schnittstelle definiert werden.

Das finale Verifikationsergebnis ist offensichtlich direkt von der Qualität und dem Detaillierungsgrad der Spezifikation abhängig. Daher lohnt sich das von Ball et al. vorgestellte Verfahren vor allem bei Schnittstellen, welche beispielsweise als API[4] von vielen verschiedenen Programmen unter gleichen Sicherheitsbedingungen angesteuert werden. Die Anwendungsdomäne API für Gerätetreiber bei Microsoft Windows scheint dementsprechend ein ideales Anwendungsgebiet zu sein und wurde im Rahmen des Projektes SLAM auch als erstes Testgebiet gewählt.

[3] Invarianten sind Eigenschaften oder Ausdrücke, welche über die komplette Laufzeit einer Anwendung hinweg erfüllt sein müssen.

[4] Application Programming Interface (dt. Schnittstelle zur Anwendungsprogrammierung): APIs werden von Betriebssystemen oder Frameworks als Schnittstelle zur Erweiterung, Verwaltung oder Anbindung bereitgestellt. Dienen der Modularisierung und Standardisierung von Komponenten.

1.2 Anwendungsdomäne Gerätetreiber

Gerätetreiber sind in einer sehr kritischen Umgebung, bei Microsoft Windows innerhalb des Systemkerns, integriert. Bei einer fehlerhaften Implementation der Systemschnittstelle von Seiten einer Applikation stürzt normalerweise auch nur diese ab. Jedoch bei fehlerhaften Gerätetreibern kann durch die Nähe zum Systemkern das ganze System stillstehen oder abstürzen. Des Weiteren ist der Systemkern höchst komplex und durch die Einbindung von beliebigen Treibern das Laufzeitverhalten nicht gänzlich vorhersehbar.

Die Mehrzahl der Gerätetreiber wird von den Herstellern der Hardware selbst und nicht durch die Windows Organisation entwickelt und ausgeliefert. Microsoft hat daher keinen Einfluss auf die Qualität der Treiber. Darüber hinaus ist anzunehmen, dass die Kompetenzen hinsichtlich der Windows API bei Drittherstellern nicht in der Ausprägung wie bei Microsoft selbst zu finden sind.

Seit Windows 2000 vergibt Microsoft ein Zertifikat für Treiber von Drittherstellern, welche gewisse Mindestbedingungen erfüllen und die Testroutinen durch Microsoft erfolgreich absolvieren. Der Treiber wird dann entsprechend digital signiert und mit dem Logo „Designed for Windows“ ausgewiesen. Hiermit entsteht ein gewisser Anreiz für die Hardwarehersteller, da dies eine eventuell günstigere Marktposition ermöglicht.

Die Stabilität eines Betriebssystems ist für viele Benutzer ein entscheidender Qualitätsfaktor. Der durchschnittliche Anwender kann bei einem Absturz aber nicht unterscheiden, ob die Ursache hierfür bei Microsoft oder bei einem Dritthersteller liegt. Fehlerhafte Treiber können also durchaus direkte Auswirkungen auf das Qualitätsempfinden gegenüber Windows und damit auf das Image von Microsoft haben.

Es ist nicht weiter verwunderlich, dass die Bedeutung der Treiberdomäne und das Potential von SLAM bei Verantwortlichen von Microsoft Windows relativ schnell erkannt wurden. SLAM ermöglicht eine automatische Verifikation der Treiber und gegebenenfalls eine Identifizierung und Lokalisierung der Fehler.

Ein wesentlicher Vorteil der Treiberdomäne ist, dass die Windows API für alle Treiber identisch ist und somit keine Anpassung der Sicherheitsregeln benötigt wird. SLAM kommt zugute, dass die Komplexität durchschnittlicher Treiber relativ zu Anwendungen gering ist und deren Quellcode zumeist weniger als hunderttausend Zeilen Code aufweist.

1.3 Projektablauf

Im Folgenden werden die verschiedenen Meilensteine für das Projekt SLAM und diesbezüglich relevante Ereignisse kurz aufgeführt.

Die Idee zu SLAM resultierte aus einem Gespräch zwischen Thomas Ball und Sriram K. Rajamani von Microsoft Research über Symbolic Execution, Model Checking und Program Analysis und wie diese kombiniert werden könnten, um Schnittstellenprobleme vor allem im Hinblick auf Gerätetreiber zu vermeiden. Anfang 2000 veröffentlichten sie zusammen einen technischen Bericht [BR00], in welchem sie das grundlegende Verfahren, den Prozess und die Architektur des nun initiierten Projektes SLAM konkretisierten.

Bis Mitte 2000 konnten schon die ersten Versionen der Komponenten Bebop (Model Checking) und C2bp (Code Abstraktion) an simplen Windows NT Treiber getestet werden. Im Herbst kam dann die dritte Komponente von SLAM Newton (Pfadanalyse und Prädikatgewinnung) hinzu.

Bis zum Frühling 2001 wurde SLIC, eine C-ähnliche Spezifikationssprache entwickelt, womit nun die ersten vollautomatischen Überprüfungen an realem Treiber-Code möglich waren, wobei die Spezifikationen natürlicherweise vorher manuell definiert werden mussten.

Mit den ersten erfolgreich gefundenen Fehlern in Gerätetreibern wurde auch 2001 das Paper Boolean and Cartesian Abstractions for Model Checking C Programs [BPR01] in Genua im Rahmen der TACAS[5] veröffentlicht.

Anfang 2002 wurde SLAM beim jährlichen TechFest von Microsoft Research, einem Event mit mehreren tausend Teilnehmern aus den Reihen der gesamten Microsoft Organisation, dem Publikum vorgestellt, wobei Bill Gates und andere führende Persönlichkeiten zum ersten Mal auf SLAM aufmerksam wurden. Zwei Wochen nach dem TechFest wurde SLAM nochmals Bill Gates präsentiert und der Transfer von Microsoft Research zu Microsoft Windows wurde beschlossen. Das Forschungsprojekt SLAM sollte nun zu einem Produktionstool namens Static Driver Verifier (SDV) weiter entwickelt werden.

SDV Version 1.0 wurde Ende 2002 an einen ausgewählten Kreis von Entwicklern bei Microsoft Windows freigegeben. Bis Anfang 2003 wurde das Projekt nochmals bezüglich der neuen Herausforderungen in einer Produktionsumgebung komplett überarbeitet und schließlich in Version 1.2 an alle internen Entwickler von Windows-Treibern ausgegeben.

Im Rahmen der Windows Driver Development Conference (DDC) wurde im Novem-

[5]TACAS: International Conference on Tools and Algorithms for the Construction and Analysis of Systems

ber 2003 die Version SDV 1.3 vorgestellt. Die Version zeichnete eine erweiterte SLIC-Spezifikation, 60 neue Sicherheitsregeln, ein verbessertes Modell des Windows-Kernels und eine bessere Integration in die Entwicklungsumgebung aus. Vor allem Microsoft-externe Entwickler, welche bisher noch kein Zugang zu dem Tool hatten, bekundeten während der DDC ihr Interesse an SDV.

Seit 2004 ist der Transfer des Projektes von Microsoft Research zu Windows vollendet und wird dort von einer eigenen Gruppe weiterentwickelt. Inzwischen ist SDV als Teil des Driver Development Kit (DDK) für die Öffentlichkeit zugänglich und fungiert auch Microsoft-intern als integraler Bestandteil des Entwicklungsprozesses.

2 SLAM

Das gesetzte Ziel von SLAM ist es, herauszufinden, ob ein Programm gegen Verwendungsregeln einer Schnittstelle verstößt oder nicht. Im Falle eines Verstoßes soll dieser im Code identifiziert, lokalisiert und anhand eines Ausführungspfades dem Entwickler kommuniziert werden. Hierfür werden statische Analysemethoden unter Verwendung von boolescher und kartesischer Abstraktion eingesetzt.

SLAM zeichnet sich vor allem durch einen weitestgehend automatischen Ansatz aus, d.h. der Entwickler muss keine manuellen Annotationen im Quellcode vornehmen, sondern bei gegebenen Spezifikationen und Systemmodellen nur die Analyse initiieren. Des Weiteren werden vermeintliche Fehler im Code durch einen iterativen Verfeinerungsprozess mittels Gegenbeispielen revalidiert und damit die Präzision des Resultates erhöht.

2.1 Verifikationsprozess und Komponenten

Mittels SLAM sollen sequenzielle Programme in der Programmiersprache C gegen temporale Sicherheitseigenschaften verifiziert werden. Eine Verletzung dieser kann durch einen endlichen Ausführungspfad [Lam79] bezeugt werden. Sicherheitseigenschaften fordern beispielsweise einen gegenseitigen Ausschluss (Mutual Exclusion) von Zugriffen auf gemeinsame Datenbereiche durch nebenläufige Prozesse, um inkonsistente Zustände zu verhindern.

Die Sicherheitseigenschaften werden in der eigens entworfenen Sprache SLIC[1] abgebildet, mit welcher endliche Automaten modelliert werden können, welche das Ausführungsverhalten des Programms auf der Ebene von Funktionsaufrufen und Rückgabeevents überwacht [BR02b].

Mit Hilfe der Automaten werden relevante Zustände bezüglich der Sicherheitseigenschaften protokolliert und beim Übergang in einen nicht erlaubten Zustand der Verstoß signalisiert. Praktisch wird dies durch eine Erweiterung des ursprünglichen Quellcodes eines Programms P mit den SLIC Spezifikationen S zu einem adaptierten Programm P'

[1]SLIC: Specification Language for Interface Checking

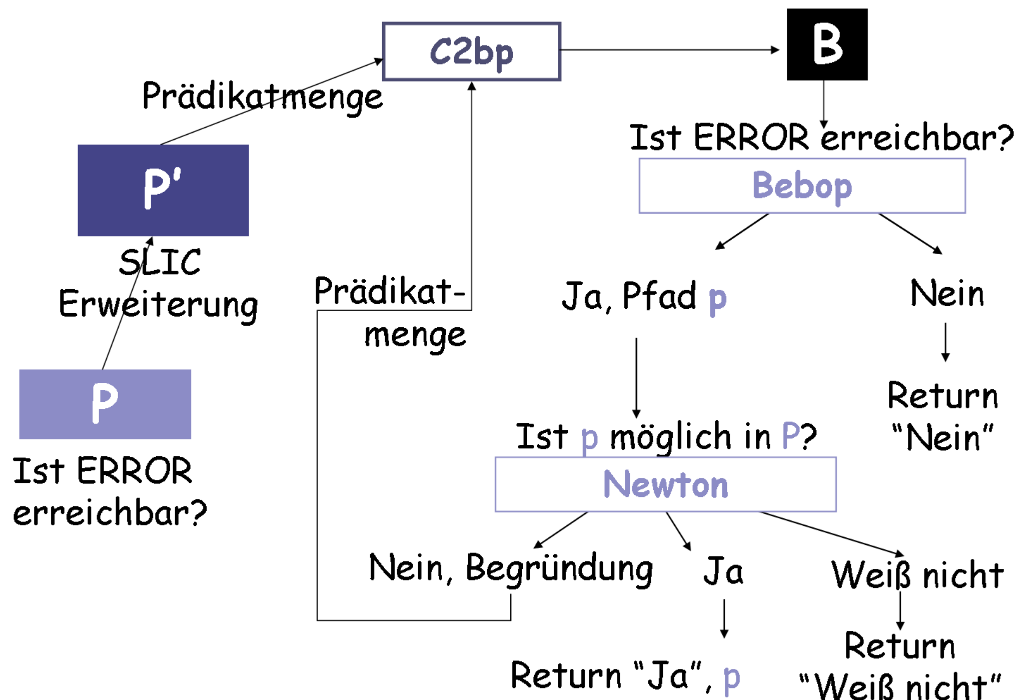

Abbildung 2.1: Iterativer Verifikationsprozess von SLAM mit der Spezifikationssprache SLIC und den Komponenten C2bp, Bebop und Newton.

umgesetzt. Somit ist in P' genau dann ein Zustand mit einer eindeutigen Fehlerkennung erreichbar, wenn P gegen S verstößt.

Um den Zustandsraum und damit die Komplexität zu minimieren, wird wie schon in Kapitel 1.1 angesprochen, das adaptierte Programm P' in ein boolesches Programm BP(P',E) durch die SLAM Komponente C2bp unter Berücksichtigung einer Prädikatmenge E transformiert. Ein boolesches Programm ist ein in C-Syntax gehaltenes Programm, in welchem alle Variablen vom Typ Boolean sind. Demnach können die Variablen nur die Werte „true", „false" und „*" (beliebig) annehmen. Boolesche Ausdrücke des Programms P' werden jeweils in BP(P',E) auf boolesche Variablen abgebildet. Die automatische Generierung der booleschen Programme wird mittels Prädikatabstraktion [GS97] durchgeführt.

Anhand von BP(P',E) wird nun durch eine weitere SLAM-Komponente namens Bebop überprüft, ob in dieser abstrahierten Programmstruktur ein durch die Sicherheitseigenschaften verbotener Zustand erreicht werden kann. Falls dies nicht möglich ist, kann daraus geschlossen werden, dass auch das originale Programm P nicht gegen die Sicherheitseigenschaften verstößt.

Im anderen Falle, wenn es einen Ausführungspfad in BP(P',E) gibt, welcher in einen

verbotenen Zustand führt, muss überprüft werden, ob dieser Pfad auch in Programm P möglich ist. Falls dem so ist, wurde eine fehlerhafte Implementierung der Schnittstelle entdeckt. Im anderen Falle wurde der Pfad erst durch die Abstraktion ermöglicht und das boolesche Programm BP(P',E) muss nochmals in Bezug auf eine erweiterte Prädikatmenge E verbessert werden und sich einer erneuten Pfadanalyse unterziehen.

Das Revalidieren von vermeintlichen Ausführungspfaden in P und die Gewinnung von weiteren Prädikaten für eine erneute Abstraktion wird im Projekt SLAM durch die Komponente Newton abgedeckt.

Grundsätzlich findet also im SLAM Prozess ein iterativer Ansatz Verwendung, d.h. es wird solange abstrahiert, analysiert und verfeinert bis ein Fehler gefunden, ausgeschlossen oder nicht ermittelt werden kann. Falls die Komplexität trotz Abstraktion ein gewisses Maß übersteigt, kann innerhalb einer gesetzten Zeit möglicherweise kein eindeutiges Ergebnis gefunden werden und die Verifikation bleibt offen.

2.2 SLIC: Spezifikationssprache

Mittels der Spezifikationssprache SLIC (Specification Language for Interface Checking) können Sicherheitseigenschaften definiert werden, gegen welche im SLAM Prozess die zu überprüfenden Programme verifiziert werden. Die SLIC Spezifikation ist in C-Syntax gehalten und enthält einerseits eine statische Menge von Zustandsvariablen und andererseits Events und Zustandstransitionen bezüglich der Events. Die Zustandsvariablen können von einem beliebigen C-Typ sein und jeglichen Wert annehmen, welcher die Programmiersprache C erlaubt. Mit SLIC kann folglich ein endlicher Automat modelliert werden, welcher das Ausführungsverhalten eines zu überprüfenden Programms überwacht.

Abbildung 2.2 zeigt die SLIC Spezifikation und den entsprechenden endlichen Automaten für eine Verwendungsregel, welche nur ein alternierendes Sperren und Entriegeln zulässt. Mit der Zustandsvariablen „locked“ wird der aktuelle Status gespeichert, welcher initial auf 0, also auf entriegelt bzw. unlocked gesetzt ist.

Des Weiteren sind zwei Funktionen AcquireLock und ReleaseLock definiert, welche als Zustandsübergänge fungieren und deren Rückgabewerte den aktuellen Zustand setzen. Im Falle eines Verstoßes gegen die SLIC Spezifikation, z.B. bei einem zweimaligen Sperren hintereinander, nimmt der endliche Automat den Error Zustand an und SLIC signalisiert eben diesen. Das zu überprüfende Programm P wird mittels Produktautomatenkonstruktion mit den Methodenaufrufen der SLIC Spezifikation S zu P' erweitert. In Abbildung 2.3 wird P' eines exemplarischen Treibers P dargestellt. Bei jedem Zugriff

des Treibers auf die API wird in P' zugleich auch die entsprechende SLIC Methode aufgerufen und eine Zustandstransition ausgelöst.

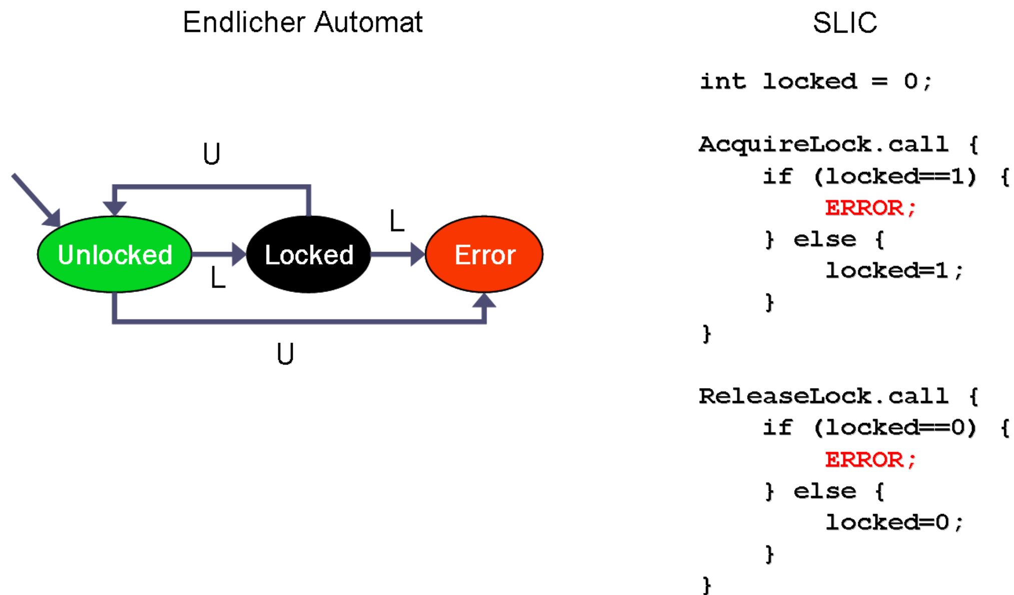

Abbildung 2.2: Links: Endlicher Automat einer Sicherheitseigenschaft für alternierendes Sperren und Entriegeln. Rechts: Entsprechende Modellierung der Sicherheitseigenschaften in SLIC.

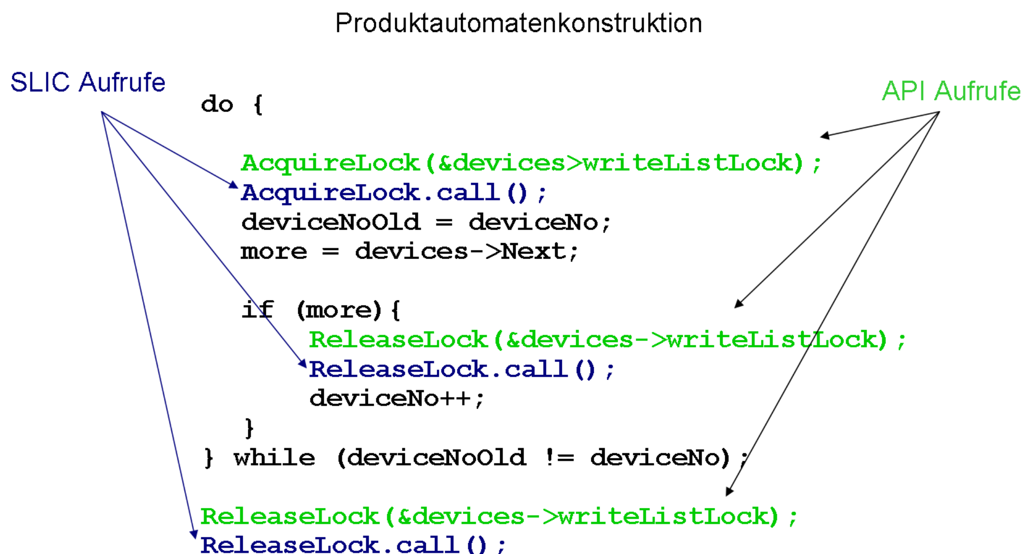

Abbildung 2.3: Kombination von Programm-Quellcode und SLIC Methodenaufrufen mittels Produktautomatenkonstruktion.

2.3 C2bp: Boolesche und kartesische Abstraktion

Abstraktion ist für das Model Checking von komplexeren Programmen unvermeidlich. Rekursive Ansätze, nicht begrenzte Datentypen und Strukturen können bei einer Verifikation ohne vorherige Abstraktion zu einer Zustandsexplosion führen, wodurch zumindest in angemessener Zeit keine Aussagen bezüglich Terminierung und Erreichbarkeit getroffen werden können.

Im Projekt SLAM transformiert die Komponente C2bp das mit den Spezifikationen erweiterte Programm P' in ein boolesches Programm BP(P',E) unter Berücksichtigung einer Prädikatmenge E. Hierbei werden konkrete Zustände auf abstrakte Zustände in Bezug auf eine endliche Menge von Prädikaten abgebildet.

Boolesche Programme können die gleichen Kontrollkonstrukte und Ablaufstrukturen wie Programme in C aufweisen, heben sich aber durch eine ausschließliche Verwendung von Variablen des Typs Boolean ab. Boolesche Variablen nehmen nur die Werte „true", „false" und „*" (beliebig) an. Wertaufrufe (call-by-value), lokale Variablen, Rekursion und nicht-deterministische Kontrollstrukturen sind in booleschen Programmen möglich. Somit weisen auch boolesche Programme unbegrenzte Programmstrukturen auf, bieten aber den Vorteil der Einschränkung auf Variablen mit einer per Definition endlichen Menge an möglichen Ausprägungen (true, false, *).

Eine weitere Reduzierung der Komplexität bietet die Kombination von boolescher und kartesischer Abstraktion. Letzteres wird durch die Vernachlässigung von Abhängigkeiten zwischen einzelnen Komponenten erreicht. Hierbei wird die Menge von booleschen Tupeln durch das kleinste kartesische Produkte, welches die Menge beinhaltet approximiert. Beispielsweise würde die Menge (true, false), (true, true) durch das kartesische Produkt (*,*), welches die Menge (true, true), (true, false), (false, true), (false, false) repräsentiert, angenähert.

Boolesche Programme entsprechen in ihrer Mächtigkeit der von Pushdown Automata, welche entscheidbar in Bezug auf Erreichbarkeit und Terminierung sind [BEM97] und für welche effiziente Model Checking Algorithmen existieren [EHRS00].

C2bp erstellt dementsprechend von einem Programm C ein abstrahiertes Modell in Form eines booleschen Programms, welches stark vereinfacht ist und alle Ausführungspfade des ursprünglichen Programms und mehr beinhaltet. Durch die Reduktion auf das kleinste kartesische Produkt einer Menge können im booleschen Programm auch Ausführungspfade möglich sein, welche das originale Programm nicht erlaubt. Daher ist beispielsweise bei der Analyse der Erreichbarkeit unbedingt notwendig, gefundene Ausführungspfade im booleschen Programm auch auf die Existenz im ursprünglichen C Programm zu überprüfen.

```
do {
   AcquireLock(&devices>writeListLock);
   AcquireLock.call();
   deviceNoOld = deviceNo;
   more = devices->Next;
   if (more){
        ReleaseLock(&devices->writeListLock);
        ReleaseLock.call();
        deviceNo++;
   }
} while (deviceNoOld != deviceNo);
ReleaseLock(&devices->writeListLock);
ReleaseLock.call();
```

Abbildung 2.4: Ausschnitt aus beispielhaften C Programm P' mit eingebundenen SLIC Methodenaufrufen bezüglich alternierender Zugriffe auf API Ressource.

```
do {

   AcquireLock.call();


   if ( * ){

        ReleaseLock.call();

   }
} while ( * );

ReleaseLock.call();
```

Abbildung 2.5: Boolesches Programm BP(P',E0) unter Berücksichtigung der Prädikatmenge $E0 = \{\}$.

```
do {

   AcquireLock.call();
   b = true;

   if (*){

        ReleaseLock.call();
        b = b ? false : *;
   }
} while (!b);

ReleaseLock.call();
```

Abbildung 2.6: Boolesches Programm BP(P',E1) unter Berücksichtigung der Prädikatmenge $E1 = \{(\text{deviceNoOld} = \text{deviceNo})\}$.

2.4 Bebop: Model Checker

Das von C2bp erzeugte boolesche Programm wird im SLAM Prozess als Input dem Bebop Model Checker übergeben. Bebop berechnet für jede Anweisung des booleschen Programms die Menge der erreichbaren Zustände mithilfe von Algorithmen der interprozeduralen Datenflußanalyse [RHS95].

Demzufolge beinhaltet ein Zustand eines booleschen Programms an einer Anweisung w einen Bit-Vektor, welcher für alle boolesche Variablen im Geltungsbereich der Anweisung w den entsprechenden Wert abbildet [BMMR01]. Die Menge der erreichbaren Zustände bzw. Invarianten bei w ist dementsprechend eine Menge von Bit-Vektoren.

```
        decl g;

        main()
        begin
          decl h;
[6]       h := !g;
[7]       A(g,h);
[8]       skip;
[9]       A(g,h);
[10]      skip;
[11]      if (g) then
[12]    R:  skip;
          else
[14]        skip;
          fi
        end

        A(a1,a2)
        begin
[20]      if (a1) then
[21]        A(a2,a1);
[22]        skip;
          else
[24]        g := a2;
          fi
        end
```

```
bebop v1.0: (c) Microsoft Corporation.
Done creating bdd variables
Done building transition relations

Label R reachable by following path:

Line 12        State g=1 h=0
Line 11        State g=1 h=0
Line 10        State g=1 h=0
    Line 22    State g=1 a1=1 a2=0
        Line 24 State g=1 a1=0 a2=1
        Line 20 State g=1 a1=0 a2=1
    Line 21    State g=1 a1=1 a2=0
    Line 20    State g=1 a1=1 a2=0
Line 9         State g=1 h=0
Line 8         State g=1 h=0
    Line 22    State g=1 a1=1 a2=0
        Line 24 State g=1 a1=0 a2=1
        Line 20 State g=1 a1=0 a2=1
    Line 21    State g=1 a1=1 a2=0
    Line 20    State g=1 a1=1 a2=0
Line 7         State g=1 h=0
Line 6         State g=1
```

Abbildung 2.7: Die Anweisung „skip“ in Zeile 12 ist durch den von Bebop ausgewiesenen Ausführungspfad im aufgeführten booleschen Programm erreichbar [BMMR01].

Die Zustandsmengen und Übergangsfunktionen werden implizit in binären Zustandsdiagrammen[2] repräsentiert [Bry86]. Zur expliziten Kommunikation werden aber nicht Zustandsdiagramme, sondern Kontrollfluss-Graphen verwendet. Hierbei werden die Lokalitätseigenschaften ausgenutzt, indem nur diejenigen Variablen im Kontrollfluss-Graphen berücksichtigt werden, welche im Gültigkeitsbereich des entsprechenden Punktes im Graphen liegen.

[2]Binäres Zustandsdiagramm: engl. Binary Decision Diagram (BDD)

2.5 Newton: Pfad-Analyse und Prädikatgewinnung

Jeder Ausführungspfad eines C Programms P' ist in dem zugehörigen durch C2bp abstrahierten booleschen Programm BP(P',E) abgebildet, aber nicht jeder in BP(P',E) mögliche Pfad ist auch in dem ursprünglichen Programm P' zulässig. Daraus folgt, falls der Model Checker Bebop im booleschen Programm einen Pfad zu einem durch die SLIC Spezifikation nicht erlaubten Fehlerzustand findet, muss dieser Pfad auch auf die Zulässigkeit im originalen C Programm validiert werden.

Die Revalidierung des Fehlerpfads übernimmt die SLAM Komponente Newton, indem gegengepüft wird, ob entlang des gegebenen Fehlerpfads $p = s_1, s_2, ...s_n$ die Prädikate des C Programms zu Widersprüchen führen.

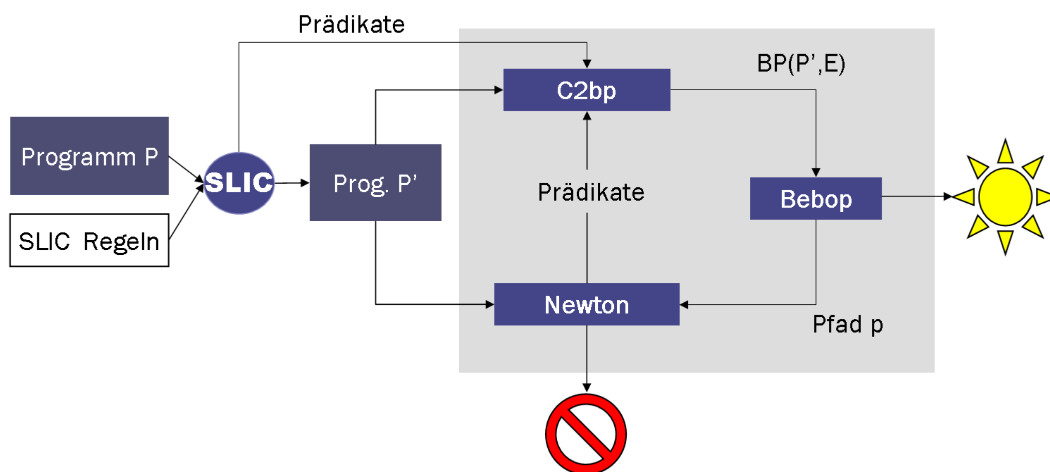

Abbildung 2.8: SLAM Gesamtprozess mit Iterationszirkel entlang der Komponenten C2bp, Bebop und Newton.

Lassen sich keine Widersprüche finden, so ist ein Fehlerpfad und damit ein Verstoß gegen die Sicherheitsregeln identifiziert und validiert. Im anderen Fall werden die Prädikate, welche zu Widersprüchen führten, der Prädikatmenge E hinzugefügt und eine weitere Iteration des SLAM Prozesses wird angestoßen (siehe Abbildung 2.8). C2bp erstellt dementsprechend aufgrund der erweiterten Menge an Prädikaten eine verfeinerte Abstraktion, welche den vermeintlichen Fehlerpfad nicht mehr zulässt.

Die Iteration aus Abstraktion, Model Checking, Pfadanalyse und Prädikatgewinnung wird so lang durchgeführt, bis ein Fehlerpfad validiert wurde oder keiner mehr gefunden werden kann.

2.6 Laufzeit

Die Laufzeit von SLAM setzt sich aus der Anzahl der notwendigen Iterationen und der Laufzeit der einzelnen SLAM Komponenten zusammen.

Die Komponente C2bp weist eine Laufzeit von $O(|P|*|E|^k)$ auf. C2bp verhält sich daher linear in der Größe des zu überprüfenden Programms und exponentiell in der Anzahl der übergebenen Prädikate. Empirisch scheint ein $k = 3$ als für den praktischen Einsatz genügend präzise. So ergibt sich eine Laufzeit von $O(|P| * |E|^3)$.

Für ein Programm mit globalen Variablen g und einem Maximum von lokalen Variablen l über alle Prozeduren beträgt die Laufzeit des Model Checkers Bebops $O(E * 2^{g+l})$. E steht hier für die Anzahl der Kanten des interprozeduralen Kontrollfluss-Graphen des booleschen Programms und ist linear zu der Anzahl der Anweisungen des Programms.

Für die Pfadanalyse bzw. Prädikatgewinnung benötigt Newton eine Laufzeit von $O(|p|)$ und verhält sich damit linear zu der Länge des vermeintlichen Fehlerpfads.

Insgesamt kann nicht gewährleistet werden, dass der SLAM Prozess terminiert. Im bisherigen praktischen Einsatz bei Microsoft beendete SLAM spätestens nach 20 Iterationen den Prozess. Mittels SLAM können Programme mit über 10.000 Zeilen Code und mehreren hundert booleschen Variablen in einer Zeitspanne von einer Minute bis zu einer halben Stunde überprüft werden [BR02a].

3 Schlussfolgerung

Ball et al. von Microsoft Research stellen mit SLAM einen automatisierten Prozess zur Validierung von temporalen Sicherheitseigenschaften von Softwareschnittstellen vor. Hierbei sind keine manuellen Annotationen oder Umformungen des Quellcodes von Seiten der Entwickler notwendig.

Die Verwendung von boolescher und kartesischer Abstraktion ermöglicht auch komplexere Programme zu validieren. Trotz der Abstraktion kann ein Präzisionsverlust mittels Revalidierung durch Gegenbeispiele und iterativer Verfeinerung der Abstraktion vermieden werden.

SLAM bzw. SDV beweist als Teil des Driver Development Kit seine Praxistauglichkeit bei Dritttherstellern von Windows Treibern und ist bei Microsoft integraler Bestandteil des Entwicklungsprozesses.

Die Erkennungsrate bzw. -qualität von Fehlern hängt bei SLAM direkt von der SLIC Spezifikation ab. Es können nur Fehler gefunden werden, welche explizit durch Sicherheitseigenschaften modelliert wurden. Daher muss viel Sorgfalt und hoher Aufwand in die Regeldefinition investiert werden.

Diese Investitionskosten zahlen sich aber meistens nur aus, wenn mit einer Spezifikation eine Vielzahl von Programmen validiert werden können. Die Treiberdomäne ist prädestiniert für den Einsatz von SLAM – fraglich ist aber, ob noch ähnlich profitable Domänen gefunden werden können.

Abbildungsverzeichnis

2.1 Iterativer Verifikationsprozess von SLAM mit der Spezifikationssprache SLIC und den Komponenten C2bp, Bebop und Newton. 9
2.2 Links: Endlicher Automat einer Sicherheitseigenschaft für alternierendes Sperren und Entriegeln. Rechts: Entsprechende Modellierung der Sicherheitseigenschaften in SLIC. 11
2.3 Kombination von Programm-Quellcode und SLIC Methodenaufrufen mittels Produktautomatenkonstruktion. 11
2.4 Ausschnitt aus beispielhaften C Programm P' mit eingebundenen SLIC Methodenaufrufen bezüglich alternierender Zugriffe auf API Ressource. . 13
2.5 Boolesches Programm BP(P',E0) unter Berücksichtigung der Prädikatmenge E0 = {}. 13
2.6 Boolesches Programm BP(P',E1) unter Berücksichtigung der Prädikatmenge E1 = {(deviceNoOld = deviceNo)}. 13
2.7 Die Anweisung „skip“ in Zeile 12 ist durch den von Bebop ausgewiesenen Ausführungspfad im aufgeführten booleschen Programm erreichbar [BMMR01]. 14
2.8 SLAM Gesamtprozess mit Iterationszirkel entlang der Komponenten C2bp, Bebop und Newton. 15

Literaturverzeichnis

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, pages 1–20, 2004.

[BEM97] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory*, pages 135–150, London, UK, 1997. Springer-Verlag.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, New York, NY, USA, 2001. ACM Press.

[BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283, London, UK, 2001. Springer-Verlag.

[BR00] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, January 2000.

[BR02a] Thomas Ball and Sriram K. Rajamani. The slam project: Debugging system software via static analysis. In *Symposium on Principles of Programming Languages*, Portland, USA, 2002.

[BR02b] Thomas Ball and Sriram K. Rajamani. Slic: A specification language for interface checking. Technical Report MSR-TR-2001-21, Microsoft Research, January 2002.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[EHRS00] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 232–247, London, UK, 2000. Springer-Verlag.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83, London, UK, 1997. Springer-Verlag.

[Lam79] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.*, 1(1):84–97, 1979.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM Press.