



# Erfahrungsbericht Verifikationstechniken



## Zusammenfassung:

Zunehmend werden in der Softwareentwicklung Verifikationstechniken in ihren unterschiedlichen Ausprägungen eingesetzt. Mit Hilfe von Verifikation soll eine höhere Qualität eines Softwaresystems dadurch gewährleistet werden, dass Eigenschaften des Systems nachgewiesen werden.

Der Bericht beinhaltet basierend auf praktischen Erfahrungen eine Klassifikation verschiedener Verifikationstechniken. Besonders berücksichtigt wird dabei der Automatisierungsgrad und die Skalierbarkeit für große und komplexe Anwendungen im Umfeld der eingebetteten Systeme.

Der Schwerpunkt liegt dabei insbesondere auf formalen Verifikationstechniken, da diese besonders sorgfältig ausgewählt werden müssen, um in bestehende Software-Entwicklungsprozesse erfolgreich integriert werden zu können.

## Inhalt:

<b>1</b>	<b>Einleitung</b> .....	<b>3</b>
<b>2</b>	<b>Klassifikation von Verifikationsverfahren</b> .....	<b>3</b>
2.1	Statische Eigenschaften von Programmen .....	4
2.1.1	Syntaxprüfung.....	4
2.1.2	Typprüfung .....	4
2.1.3	Codierungsrichtlinienprüfung .....	5
2.2	Dynamische Eigenschaften von Programmen.....	5
2.2.1	Laufzeitsysteme für Programmiersprachen .....	5
2.2.2	Symbolische Analyse / Abstrakte Interpretation .....	6
2.2.3	Tests.....	6
2.3	Statische Eigenschaften von Modellen .....	7
2.3.1	Konsistenzprüfer für Modelle .....	7
2.3.2	Modellierungsrichtlinienchecker.....	7
2.4	Dynamische Eigenschaften von Modellen .....	7
2.4.1	Modellierungswerkzeuge, Simulatoren .....	8
2.4.2	Model Checking .....	8
2.4.3	Bounded Model Checker.....	8
2.4.4	Formale Beweissysteme.....	9
2.4.5	Werkzeugkombinationen.....	10
<b>3</b>	<b>Beispiele für Verifikationsverfahren</b> .....	<b>10</b>
3.1	Überprüfung von Codierrichtlinien.....	10
3.2	Abstrakte Interpretation mit dem Polyspace Verifier .....	12
3.3	Model Checking mit AutoFOCUS.....	14
3.4	Modellbasierte Testfallgenerierung mit AutoFOCUS.....	16
<b>4</b>	<b>Erfahrungen und Success-Stories</b> .....	<b>19</b>
4.1	Modellbasierte Testfallgenerierung.....	19
4.2	Model-Checking.....	19
4.3	Symbolische Analyse .....	20
<b>5</b>	<b>Einsatzempfehlung</b> .....	<b>20</b>
<b>6</b>	<b>Zusammenfassung</b> .....	<b>21</b>
<b>7</b>	<b>Literatur</b> .....	<b>21</b>

## 1 Einleitung

Die Verifikation (lat. verum + ficere = wahr machen) bedeutet das Überprüfen von Thesen anhand von vorliegenden Fakten auf ihren Wahrheitsgehalt. In der Software-Entwicklung wird bei Verifikation die Korrektheit eines Programms (oder Modells) überprüft. Dabei wird geprüft, ob das Programm (oder Modell) einer vorgegebenen Spezifikation entspricht. In der Praxis ist neben der Verifikation auch die Falsifikation, also das Aufzeigen von Fehlern, von großer Bedeutung. Der Korrektheitsbegriff ist dabei sehr vielfältig und reicht von Syntax bis zu beliebigen Prädikaten und Anforderungstests (siehe Abschnitt 2). Unterschiedliche Methoden werden für unterschiedliche Eigenschaften eingesetzt. Manche Verifikationsaufgaben sind so komplex, dass die vorhandenen Lösungsmethoden nur auf kleine oder mittlere Programme anwendbar sind.

Die Verifikation ist eine analytische Methode zur Sicherung der Softwarequalität, die allerdings bereits in frühen Phasen eingesetzt werden kann (z.B. bevor die Software implementiert wird).

Die Validierung beantwortet (laut [Boehm81]) die Frage: „Tun wir das Richtige?“, Verifikation beantwortet die Frage „Tun wir es richtig?“. Damit wird die Validierung am Anfang und am Ende des Software-Entwicklungsprozesses eingesetzt (siehe Abbildung 1), innerhalb der SW-Entwicklung kann dann die Verifikation in unterschiedlichen Ausprägungen eingesetzt werden.

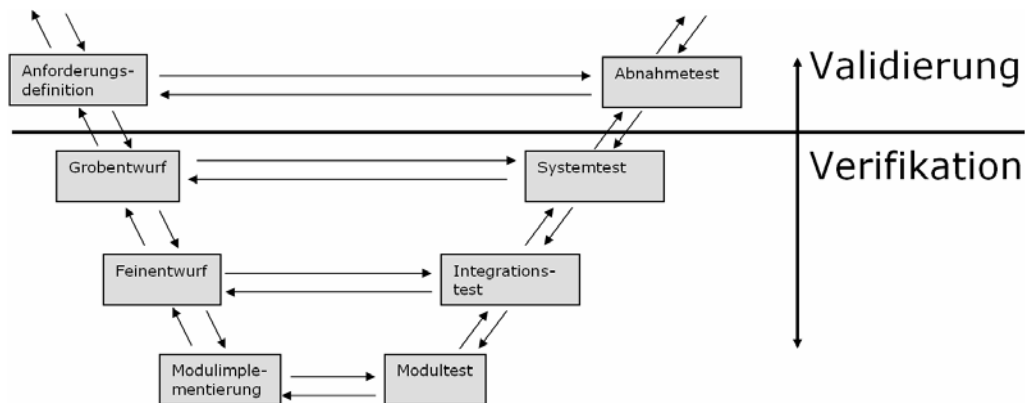


Abbildung 1: Verifikation im SW-Entwicklungsprozeß

In diesem Bericht wird ein Schwerpunkt auf die formalen Verifikationsverfahren gelegt.

## 2 Klassifikation von Verifikationsverfahren

Zur einer Verifikationsaufgabe gehört immer eine Menge von Fakten („Axiomen“) und eine zu beweisende Aussage („Verifikationsziel“), z.B. ein Programm und die Frage ob es korrekt typisiert ist. Eine Verifikationsmethode beschreibt ein Nachweisverfahren, mit dessen Hilfe ein Verifikationsziel anhand der gegebenen Axiome überprüft werden

kann. Eine für ein Verifikationsziel geeignete Verifikationsmethode liefert jedoch nicht notwendigerweise für jede beliebige Menge von Axiomen einen Beweis oder eine Widerlegung. Verifikationsmethoden werden in der Regel durch Werkzeuge unterstützt, so dass der Nachweis automatisch oder halbautomatisch geführt werden kann.

Im Folgenden werden Verifikationsziele und deren Nachweisverfahren anhand ihrer Charakteristika klassifiziert und kurz beschrieben. Es wird unterschieden zwischen den Verifikationsobjekten Modell und Programm. Dabei können Verifikationsziele für Programme in der Regel auch für Modelle sinnvoll angewendet werden können. Zum einen können aus Modellen oft Programme generiert werden, zum anderen enthalten Modelle oft auch kleine Programmfragmente. Daneben werden die Verifikationsziele in statische und dynamische Eigenschaften unterteilt. Statische Eigenschaften können überprüft werden, bevor ein lauffähiges System vorhanden ist und in der Regel müssen sie aus technischen oder organisatorischen Gründen auch zuvor erfüllt werden. Dynamische Eigenschaften betreffen Eigenschaften des laufenden Systems. In der Regel ist deren Nachweis komplexer.

### **2.1 Statische Eigenschaften von Programmen**

Statische Eigenschaften von Programmen (und Modellen) können (und müssen in der Regel auch) überprüft werden, bevor ein lauffähiges System erstellt werden kann.

#### **2.1.1 Syntaxprüfung**

Die Syntaxprüfung verifiziert (oder falsifiziert) ob ein Programm den syntaktischen Regeln der Programmiersprache entspricht.

Die Syntaxprüfung kann folgende Ausprägungen haben:

- Compiler zur Übersetzung der Programme
- Entwicklungsumgebungen (IDEs), die Syntaxfehler anzeigen (z.B. eclipse)

Ein Beispiel für die Syntaxprüfung ist, ob  $x=x+1$ ; eine korrekte Anweisung der Programmiersprache C ist.

Die Verifikationsmethode besteht aus einer automatischen Umsetzung der Syntax der Sprache (Grammatik) in ein Analyseprogramm (Parser) zur Überprüfung. Parser sind in der Regel so effizient, dass sie die Syntaxprüfung in beliebigen Programmen hinreichend schnell durchführen können, so dass Programme beispielsweise bereits zur Eingabezeit einer Syntaxprüfung unterzogen werden können.

#### **2.1.2 Typprüfung**

Die Typprüfung verifiziert (oder falsifiziert) ob ein Programm den Typisierungsregeln der Programmiersprache entspricht. Es gibt unterschiedliche Typ-Systeme mit sehr unterschiedlichen

Typisierungsregeln und Features wie Typinferenz und polymorphe Typen (siehe auch [Pie02] und [Pie05]).

Ein Beispiel für die Typüberprüfung ist, ob `boolean bVar = 4;` eine typkorrekte Zuweisung der Programmiersprache Java ist.

Die Typüberprüfung wird beim Entwurf von Programmiersprachen berücksichtigt, so dass die Überprüfung der Typkorrektheit in beliebigen Programmen hinreichend schnell durchgeführt werden kann. Mit modernen Werkzeugen können Programme bereits zur Eingabezeit einer Typprüfung unterzogen werden können.

### 2.1.3 Codierungsrichtlinienprüfung

Codierungsrichtlinien sind Richtlinien, die bei der Programmierung eingehalten werden müssen. Sie dienen zur Verbesserung der Qualität von Programmen. Die Überprüfung, ob ein Programm Codierungsrichtlinien entspricht, ist eine Verifikationsaufgabe. Für unterschiedliche Programmiersprachen gibt es unterschiedliche Richtlinien, beispielsweise die MISRA-Richtlinien [MISRA] oder die Java Konventionen [JavaConv].

Codierungsrichtlinien sind unterschiedlich schwer zu verifizieren. Beispiele reichen von der Formatierung, über syntaktische Einschränkungen bis hin zu dem Verbot von totem Code oder der Forderung, aussagekräftige Namen für Variablen zu verwenden.

Die meisten Codierungsrichtlinien lassen sich mit statischen Methoden (ähnlich zur Syntaxüberprüfung in Abschnitt 2.1.1) durchführen, z.B. mit Hilfe der Werkzeuge: lint [SPLint], QA-C [QA-C] und Checkstyle [CkStyle]. Die nicht mechanisch prüfbareren Codierungsrichtlinien müssen manuell überprüft werden. Das Werkzeug QA-C prüft die Regeln in [HIS], einem Subset der MISRA C Regeln, weitgehend automatisch ab. Ausnahmen sind die Regeln, die eine semantische Überprüfung erfordern wie beispielsweise die Regel, dass kein Code auskommentiert werden soll.

## 2.2 Dynamische Eigenschaften von Programmen

Dynamische Eigenschaften eines Programms (oder Modells) sind Eigenschaften, die für den Ablauf des entsprechenden Systems gelten.

### 2.2.1 Laufzeitsysteme für Programmiersprachen

Laufzeitsysteme für Programmiersprachen fangen Laufzeitfehler in Programmen ab, die während der Entwicklung nicht berücksichtigt wurden.

Beispiele für Laufzeitfehler sind: Division durch Null, Über- und Unterläufe, Index-Fehler in Feldern, Unverfügbarkeit von angeforderten Ressourcen (z.B. Speicher), Zugriff auf nicht definierte Variablen. Auch das Überprüfen von Zusicherungen (Assertions), die von einigen Programmiersprachen angeboten werden, kann vom Laufzeitsystem erfolgen.

Laufzeitsysteme können Laufzeitfehler nur bei ihrem Auftreten erkennen, eine situationsabhängige Fehlerkorrektur ist in der Regel nicht möglich. Prinzipbedingt können selten auftretende Laufzeitfehler lange unerkannt bleiben. In Abschnitt 2.2.2 wird eine Technik vorgestellt, die potentielle Fehlerquellen für Laufzeitfehler zur Entwicklungszeit lokalisiert.

### **2.2.2 Symbolische Analyse / Abstrakte Interpretation**

Mit symbolischer Analyse können Laufzeitfehler in Programmen entdeckt werden ohne die Programme auszuführen. Die symbolische Analyse ist partiell, d.h. nicht alle Ausführungsmöglichkeiten werden berücksichtigt. Aus Effizienzgründen wird abstrahiert („Abstrakte Interpretation“). Daher liefert die abstrakte Interpretation von Software drei mögliche Ergebnisse für jede erreichbare Anweisung in der Software: sicher laufzeitfehlerfrei, fehlerhaft und möglicherweise fehlerhaft. Möglicherweise fehlerhafte Stellen müssen manuell untersucht werden.

Symbolische Analyse basiert auf einer abstrahierenden Repräsentation der möglichen Variablenbelegungen für jede Anweisung im Programm. Je kompakter die Darstellung, desto effizienter aber auch ungenauer ist die abstrakte Interpretation. Für praktische Anwendungen bedeutet das, dass es mehrere Genauigkeiten der Analyse gibt. Die praktische Erfahrung mit dem Polyspace Verifier [PSPC] zeigt allerdings, dass auch Programme mit 100.000 Lines of Code mit hoher Genauigkeit hinreichend schnell (Laufzeit geringer als ein Tag) analysiert werden können.

Ähnliche Verfahren erlaube es, den Speicherbedarf von Software zu ermitteln und können so Speicherüberläufe und Laufzeitüberschreitungen aufdecken [AbsInt].

### **2.2.3 Tests**

Tests sind Ausführungen von Programmen oder Modellen. Tests legen Eingaben und erwartete Ausgaben fest. Mit Tests werden funktionale Anforderungen exemplarisch überprüft.

Ein Beispiel für einen einfachen Test ist die Überprüfung der Addition anhand der beiden Zahlen 3 und 4, mit dem erwarteten Ergebnis 7.

Tests werden in der Regel in der Programmiersprache des Softwaresystems geschrieben, oder mit dem verwendeten Modellierungswerkzeug erstellt. Der Aufwand beim Testen ergibt sich zum einen aus der Ausführungszeit der Tests (insbesondere bei sich häufig wiederholenden Tests) und aus dem Erstellungsaufwand der Tests (insbesondere bei sich häufig ändernden Anforderungen). Um den Erstellungsaufwand zu reduzieren, gibt es beispielsweise unterschiedliche Varianten der modellbasierten Testfallgenerierung in unterschiedlichen Modellierungswerkzeugen, z.B. RhapsodyATG [USC-ES], MTest für Matlab-Simulink [MTest], oder Test Input Generierung für AutoFOCUS [AF] (siehe auch Abschnitt 3.4).

### **2.3 Statische Eigenschaften von Modellen**

Statische Eigenschaften von Modellen können (und müssen in der Regel auch) überprüft werden, bevor ein lauffähiges System erstellt werden kann.

#### **2.3.1 Konsistenzprüfer für Modelle**

Konsistenzprüfer für Modelle überprüfen statische Konsistenzbedingungen auf Modellen. Beispiele hierfür sind Bedingungen zwischen unterschiedlichen Sichten eines Modells oder Modellierungsrichtlinien für Modelle. Häufig sind Konsistenzprüfer auf Modellen auch um benutzerspezifische Bedingungen erweiterbar.

Ein Beispiel für eine Sprache mit der man Konsistenzbedingungen formulieren kann, ist die Object Constraint Language (OCL). Die OCL ist in der UML integriert und wird dort verwendet um Konsistenzbedingungen an UML-Modelle zu formulieren.

Die Verifikation von statischen Konsistenzbedingungen erfolgt durch die Interpretation der Konsistenzbedingungen über dem erstellten Modell. Die Komplexität der Berechnung hängt von der Größe und der Mächtigkeit der Sprache der Bedingungen ab. Die typischen Konsistenzbedingungen, die der Typprüfung und der Richtlinienprüfung auf Programmen entsprechen, lassen sich in der Regel hinreichend schnell, also beispielsweise simultan zur Eingabe der Modelle, überprüfen.

#### **2.3.2 Modellierungsrichtlinienchecker**

Modellierungsrichtlinien sind Regeln zum Aufbau von Modellen. Sie schränken die zulässigen Modelle ein. Insbesondere bei sehr umfangreichen Modellierungssprachen und -werkzeugen wie der UML, MATLAB-Simulink/Stateflow oder ASCET ist dies notwendig, um die Modelle einfacher, leichter verständlich und effizienter implementierbar zu machen. Zusätzlich gibt es auch Richtlinien, um bekannte Fehler in Modellierungswerkzeugen oder deren Code-Generatoren zu umgehen.

Modellierungsrichtlinien sind unterschiedlich schwer zu verifizieren. Beispiele reichen von der graphischen Anordnung, über syntaktische Einschränkungen bis hin zu dem Verbot von „toten“ Modellteilen/Code oder der Forderung aussagekräftige Namen für Modellelemente zu verwenden.

Während sich viele Modellierungsrichtlinien mit Konsistenzprüfern für Modelle (siehe Abschnitt 2.3.1) durchführen lassen, können manche prinzipbedingt, wie zum Beispiel die Forderung nach aussagekräftigen Namen, nicht automatisiert überprüft werden.

### **2.4 Dynamische Eigenschaften von Modellen**

Dynamische Eigenschaften eines Modells sind Eigenschaften, die für den Ablauf des durch das Modell spezifizierten Systems gelten.

### 2.4.1 Modellierungswerkzeuge, Simulatoren

In den letzten Jahren werden mehr und mehr Modellierungswerkzeuge zur abstrakteren, graphischen Darstellung von Software angewendet. Ein Vorteil dieser Werkzeuge stellt meist die Unabhängigkeit von einer konkreten Zielplattform dar, so dass die Modelle unabhängig von der Zielplattform ausgeführt („simuliert“) werden können. Anhand der Simulation können Eigenschaften der Modelle falsifiziert oder validiert werden. Zur Implementierung können dann oft Code-Generatoren verwendet werden.

Die meisten Modellierungswerkzeuge bieten zudem unterschiedliche Sichten auf unterschiedliche Aspekte der Software an.

Beispiele für Modellierungswerkzeuge sind Matlab-Simulink, Rational-Rose und AutoFOCUS). Ein ausführlicher Vergleich von unterschiedlichen Werkzeugen ist in [Tools] zu finden.

### 2.4.2 Model Checking

Model Checking ist eine vollautomatische, formale Beweismethode, die zustandsbasierte Beschreibungen endlicher Systeme auf temporallogische Eigenschaften prüft. Die temporale Logik basiert auf der formalen Logik und ermöglicht es basierend auf atomaren Ausdrücken und booleschen Operatoren die Eigenschaften bestimmter Zustände eines Systems zu beschreiben. Mit Hilfe von temporalen Operatoren können dann Situationen beschrieben werden, in denen bestimmte Zustände „irgendwann“, „immer“ oder „niemals“ erreicht werden. Beim Model Checking werden temporallogische Eigenschaften vollautomatisch überprüft, wobei für ungültige Eigenschaften Gegenbeispiele geliefert werden. Dabei werden alle erreichbaren Zustände geprüft.

Ein Beispiel ist die Überprüfung der Eigenschaft, dass der Wert einer Variable  $x$  in allen möglichen Programmausführungen stets größer als 1 ist „always  $x > 1$ “.

Die Dauer der Prüfung ist dabei abhängig von der Zahl der im System erreichbaren und für die Eigenschaft relevanten Zustände. Üblicherweise können einige Millionen Zustände schnell durchsucht werden, die Zustandskomplexität des Gesamtsystems kann dabei auch  $10^{100}$  übersteigen. Die bekanntesten Model Checking Tools sind SMV [CadSMV], [SMV], NuSMV [NuSMV], Spin [Spin] und Provers [Prover]. Viele Model Checker sind auch als Framework konzipiert, so dass sie einfach in Anwendungen integriert werden können, z.B. [VIS] und [Prover].

### 2.4.3 Bounded Model Checker

Bounded Model Checking, ist wie Model Checking, eine vollautomatische Verifikationstechnik für temporale Eigenschaften. Im Gegensatz zu Model Checking werden beim Bounded Model Checking aber nicht alle erreichbaren Zustände überprüft, sondern nur die innerhalb einer festen Anzahl („bound“) erreichbaren. Daher kann mit Bounded Model Checking eine Eigenschaft zwar widerlegt, aber nicht verifiziert werden.

## Erfahrungsbericht Verifikationstechniken

Ein Beispiel ist die Überprüfung der Eigenschaft dass der Wert einer Variable  $x$  in allen Programmausführungen mit maximal 10 Schritten, immer größer als 1 ist „always  $x > 1$ “.

Durch die Grenze verringert sich der Suchraum, und die Bounded Model Checking-Verfahren lassen sich auf größere Softwaresysteme anwenden als die herkömmlichen Model Checking-Verfahren. Je kleiner die festgelegte Grenze ist, desto größer kann das zu verifizierende System sein, aber desto weniger Gegenbeispiele lassen sich mit der geringen Schrittzahl finden.

Einige Model Checker erlauben es Schranken anzugeben (z.B. NuSMV [NuSMV], Provers [Prover]); andere wurden speziell für Bounded Model Checking entwickelt (z.B. SATO [SATO] und chaff [ZChaff]).

### 2.4.4 Formale Beweissysteme

Formale Methoden ermöglichen eine exakte und unmissverständliche Spezifikation von Systemen und Eigenschaften mit Hilfe mathematischer Formalismen (formale Modelle). Die Bedeutung formaler Modelle ist durch eine „Semantik“ eindeutig festgelegt und es gibt einen „Kalkül“ mit Regeln um Eigenschaften abzuleiten. Mit einer geeigneten Formalisierung lässt sich jedes Verifikationsproblem als formale Beweisaufgabe darstellen. Den Korrektheitsnachweis zu erbringen kann allerdings aufwändig sein und manuelle Interaktion erfordern. Im Gegensatz zu Model Checking (feste Sprache) erlauben es formale Beweissysteme, eigene Logiken, Kalküle und Aussagen zu erstellen. Dadurch ist eine größere Vielfalt von Verifikationsproblemen geeignet darstellbar.

Ein typisches Beispiel für die Programmverifikation ist z.B. der Nachweis, dass eine Funktion `int fak(n)` für alle  $n \geq 0$  die Fakultätsfunktion  $n!$  berechnet. Im Gegensatz zu den mit Model Checking führbaren Beweisen ist dies eine Aussage für beliebige Zahlen und könnte z.B. mit einer Induktionsregel bewiesen werden.

Bei den formalen Beweissystemen unterscheidet man automatische und interaktive Beweissysteme. Automatische Systeme suchen die Lösung selbständig, interaktive Systeme erfordern manuelle Interaktion und damit einhergehend in der Regel tiefgehendes Verständnis des Beweissystems und des verwendeten Kalkül. Automatische Beweiser (z.B. otter, E, SPASS, Vampire, Waldmeister) erscheinen daher leichter einsetzbar zu sein, sind aber von ihrer Anwendbarkeit für die Software Verifikation noch eingeschränkter als Model Checking. Einen Überblick über das automatische Theorembeweisen gibt [ATP]. Interaktive Beweiser (z.B. [Isabelle], [PVS], [STeP]) erlauben es mittlerweile auch automatisch Teilprobleme zu lösen.

Die formalen Beweissysteme werden meist im universitären Umfeld entwickelt und stellen daher oftmals Forschungsprototypen dar, die sich in der Regel wenig anwenderfreundlich präsentieren. Der Einsatz erfordert meist umfangreiche logische Vorkenntnisse. Daher werden formale Beweiser in der Software-Entwicklung kaum eingesetzt, auch wenn

manche Standards (IEC [IEC], CC [CC], [ITSEC]) formale Methoden zur Erreichung der höchsten Sicherheitsstufen fordern.

### 2.4.5 Werkzeugkombinationen

Werkzeugkombinationen verbinden Modellierungswerkzeuge (wie in Abschnitt 2.1 beschrieben) mit formalen Verifikationswerkzeugen, insbesondere Model Checkern (wie in Abschnitt 2.4.2 und 2.4.1 beschrieben). Diese Werkzeugkombinationen ermöglichen es, Anforderungen an die entwickelten Modelle zu formalisieren und zu verifizieren. Die Formalisierungsmöglichkeiten hängen von der gewählten formalen Methode ab (z.B. CTL und LTL bei Model Checkern).

Ein Beispiel für eine modellbezogene Eigenschaft ist: Im Zustand **working** können die Systemausgänge **A** und **B** nie den gleichen Wert haben: **always State=Working => Val(A) != Val(B)**

Die Laufzeit von derartigen Verifikationsaufgaben hängt von folgenden Parametern ab:

- Komplexität der Modellierungssprache: Einfache Semantiken reduzieren die Laufzeit der Verifikation, komplexe Konstrukte, z.B. parallele Zustände erschweren die Verifikation,
- Komplexität der zu verifizierenden Eigenschaften (Temporale Logik, Scope der Eigenschaften,...),
- Größe der Modelle und
- Möglichkeit der Beweisunterstützung durch den Entwickler, z.B. Reduktion des Suchraums durch die Formulierung geeigneter Constraints.

Beispiele für solche Werkzeugkombinationen sind SCADE [SCADE], Statemate Model Checker [USC-ES], Quest [Slo00].

## 3 Beispiele für Verifikationsverfahren

Im Folgenden werden einige Verifikationsverfahren anhand von Beispielen detaillierter dargestellt und erläutert. Dabei werden auch die Grenzen der Verfahren detaillierter als im vorherigen Abschnitt dargestellt.

### 3.1 Überprüfung von Codierrichtlinien

Die Überprüfung von C-Code auf die Konformität zu Programmierrichtlinien wird anhand des folgenden einfachen Beispiels gezeigt:

```
if ( a=2 ) {  
    r=1;  
}
```

In dem Beispiel wird gegen die MISRA-Regel 82, die Zuweisungen innerhalb boolescher Ausdrücke verbietet, verstoßen. Die regelkonforme Überprüfung auf Gleichheit erfolgt in der Sprache C mit einem `==`, so dass das Statement `if (a==2)` verwendet werden sollte. Häufig entstehen in

solchen Programmteilen ungewollte Effekte und schwer zu findende Fehler.

Das Verfahren zur Erkennung solcher Regelverstöße, basiert wie die Kompilierung von Programmen auf dem abstrakten Syntax-Baum. Dieser wird vom C-Parser erzeugt. Im Gegensatz zu Compilern, die aus dem abstrakten Syntaxbaum Objektcode generieren, wird der Syntax-Baum von Prüfern auf die Einhaltung von Regeln geprüft.

Zum Verständnis der Funktionsweise des Baumaufbaus und der Regelprüfung ist die Kenntnis der Grammatik der Programmiersprache C notwendig. Im Beispiel werden folgende Teile der C-Grammatik verwendet (Angabe in Backus-Naur-Form):

```
IF-Statement ::= 'if' '(' Expression ')' ...  
Expression  ::= Assignment | Condition  
              | Number | Identifier;  
Assignment  ::= Identifier '=' Expression;  
Condition   ::= Expression '==' Expression;
```

Bei der Analyse des Terms: `if (a=2)` wird also von dem C-Parser der linke Syntax-Baum in Abbildung 2 aufgebaut. Das Prüfen auf die Einhaltung der MISRA-Regel 82 ist ein Top-Down-Verfahren, das Syntax-Bäume analysiert, und prüft, ob in `if`-Statement-Teilbäumen Assignments als Expressions verwendet werden. Falls ein solcher Fall, wie in Abbildung 2 gefunden wird, so wird dies als MISRA-Verletzung ausgegeben. Der rechte Teil von Abbildung 2 zeigt einen zulässigen Syntax-Baum für den Term `if (a==2)`.

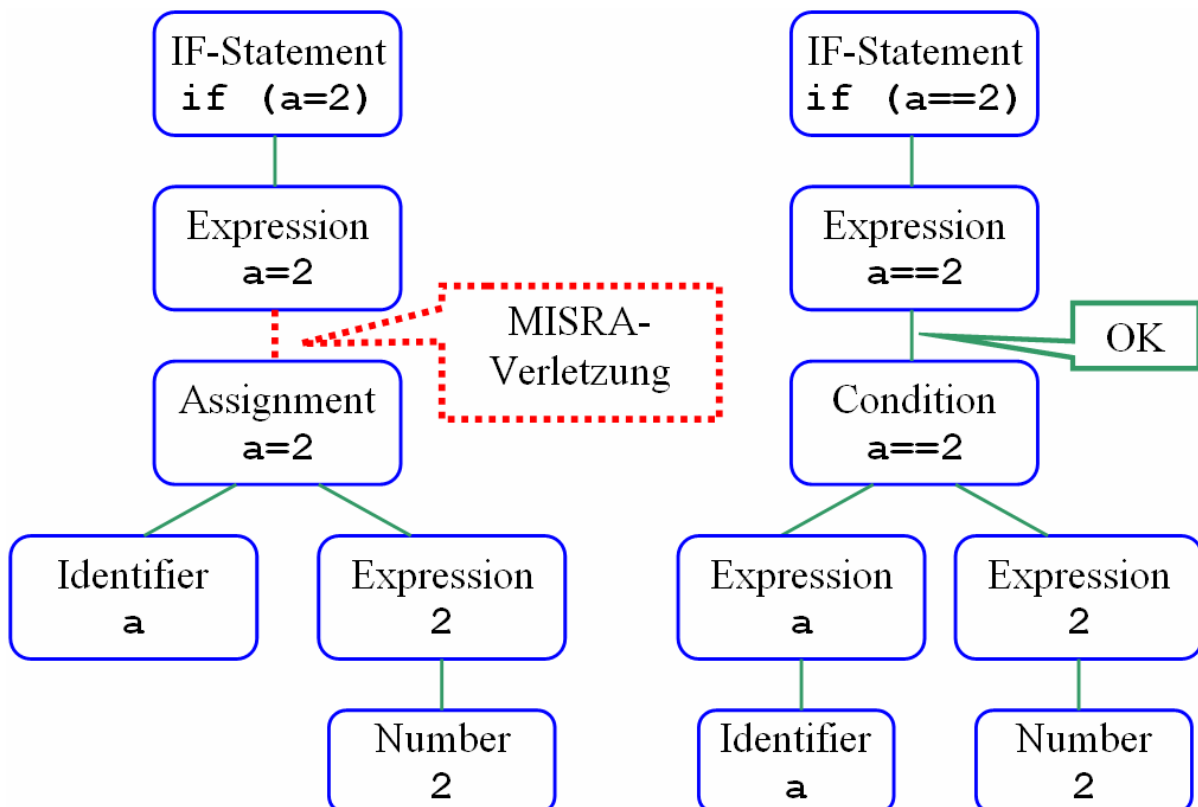


Abbildung 2: Syntax-Bäume für `if (a=2)` und `if (a==2)`

## Erfahrungsbericht Verifikationstechniken

Die meisten MISRA-Regeln werden auf diese Art geprüft. Varianten des Verfahrens werden zur Prüfung von Kommentaren und Präprozessor-Anweisungen eingesetzt.

Folgende MISRA-Regeln aus [HIS] können von [QA-C] nicht automatisch geprüft werden und müssen manuell, z.B. durch Reviews gesichert werden:

- Programme in anderen Sprachen außer C, dürfen nur verwendet werden, wenn ein definierter Standard für Object Code existiert, den beide Compiler/Assembler unterstützen.
- Es sollen keine Code-Stellen auskommentiert werden.
- Die Integer-Division des verwendeten Compilers muss festgelegt, dokumentiert und beachtet werden.
- Alle verwendeten Bibliotheken sollen nach diesen Regeln entwickelt und geprüft werden.
- Die Werte die an Bibliotheken übergeben werden, sollen geprüft werden.

### 3.2 Abstrakte Interpretation mit dem Polyspace Verifier

Am Beispiel in Abbildung 3 werden einfache Analysen und der Zusammenhang zwischen Programm und möglichen Werten gezeigt.

Programm / Anweisungen	Wertemengen der Variablen (i, j) vor Ausführung der Anweisung
<code>int i = 2;</code>	(-, -)
<code>int j=read(); // read input</code>	(2, -)
<code>j=j+5;</code>	(2, *)
<code>while (i&lt;10) {</code>	([2, 3, ..., 10], [*+5, *+8, ..., *+29])
<code>  i++;</code>	([2, 3, ..., 9], [*+5, *+8, ..., *+26])
<code>  j=j+3;</code>	([3, ..., 10], [*+5, *+8, ..., *+26])
<code>}</code>	
<code>i=1/(i-j);</code>	(10, *+29)
<code>pst_global_assert(1, i&lt;10);</code>	(10, *+29)

Abbildung 3: Beispiel zur Wertebelegung

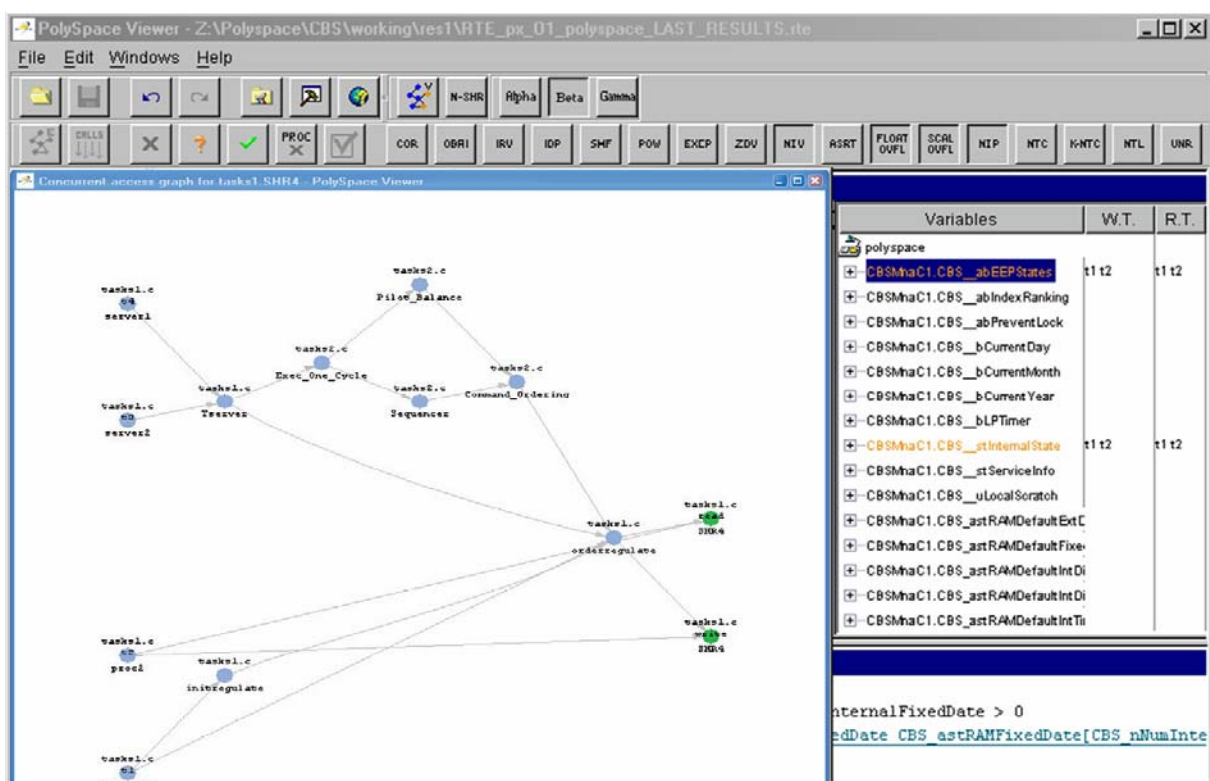
Das Programm in Abbildung 3 zeigt links ein einfaches Programm mit mehreren einfachen Berechnungen und einer Schleife. Auf der rechten Seite sind die möglichen Werte der beiden Variablen vor der Ausführung der Anweisungen des Programms dargestellt. Durch die Verwendung der Eingabe (read), kann z.B. in der zweiten Zeile ein beliebiger Wert \* der Variablen j zugewiesen werden. Durch die wiederholte Ausführung der while-Schleife mit unterschiedlichen Variablenwerten wird die Menge der möglichen Werte der Variablen größer. Vor der Division hat i den Wert 10 und j wurde insgesamt um 29 erhöht.

## Erfahrungsbericht Verifikationstechniken

Eine Analyse der möglichen Werte zeigt, dass bei jeder Addition zu der Variablen  $j$  ein Überlauf entstehen kann, falls die Variable  $j$  einen zu großen Wert hat. Diese Stellen sind im Programm **orange** markiert. Die sicher richtigen Programmteile sind **grün** markiert. Eine Division durch Null kann nur auftreten, wenn der Wert  $*$  der Variablen  $j$  gleich  $-19$  ist, denn dann hat der Divisor  $10 - (-19 + 29)$  den Wert  $0$ . Sicher auftretende Laufzeitfehler würden rot markiert und nicht erreichbarer Code grau. Die Zusicherung (mit der Nummer 1) am Ende, dass der Wert der Variablen  $i$  kleiner als  $10$  ist, ist natürlich sicher falsch und wird daher **rot** markiert.

Im Gegensatz zur zeilenorientierten Darstellung in Abbildung 3 verwendet die symbolische Analyse intern auch Werte für Teil-Ausdrücke, wie beispielsweise den Term  $(i-j)$  in der letzten Anweisung. Anhand des Beispiels sieht man auch, dass für eine Analyse des „Division durch Null“ Fehlers nicht alle möglichen Werte relevant sind. Im Beispiel hätte es auch ausgereicht, nur die Grenzwerte der Intervalle für die Variablen zu speichern. Daher liefert die symbolische Analyse häufig trotz Abstraktion genaue Ergebnisse.

Die abstrakte Interpretation im Polyspace Verifier basiert auf einer Datenflussanalyse. Diese kann auch parallele Tasks, wie in komplexen Anwendungen in eingebetteten Systemen üblich, enthalten. Auch die Ergebnisse der Datenflussanalyse (Variablen und deren lesender/schreibende Zugriff aus anderen Funktionen) sind für den Entwickler interessant und können mit dem Polyspace Viewer angezeigt werden. Abbildung 4 zeigt z.B. die Programmstellen (blauen Kreise) mit Funktions- und Dateinamen an, die auf die Variable `SHR4` lesend und schreiben zugreifen (grüne Kreise). Die Kanten in dem Datenflussgraph sind dabei die relevanten Teile aus dem Aufrufbaum des Programms.



**Abbildung 4: Zugriffsgraph für eine Variable**

Neben den Grenzen durch die Programmgröße und die gewünschte Genauigkeit der Analyse (siehe Abschnitt 2.2.2) gibt es noch eine werkzeugbedingte Grenze des Polyspace-Werkzeugs. Es kann derzeit keine 3-fach geschachtelten Pointer-Strukturen analysieren. Dies erschwert beispielsweise eine Analyse von Code, der mit ASCET generiert wurde erheblich.

### **3.3 Model Checking mit AutoFOCUS**

Das Werkzeug AutoFOCUS wird hier als ein Beispiel beschrieben, weil es eine frei verfügbare Model Checking Integration enthält und den Zusammenhang zwischen Modellen und Model Checking deutlich zeigt. Eine detaillierte Beschreibung des modellbasierten Model Checkings in AutoFOCUS ist in [AFMC].

Die Model Checking Integration von AutoFOCUS verbindet AutoFOCUS-Modelle mit den Model Checkern NuSMV [NuSMV] und Cadence SMV [CadSMV]. Dabei können Modelle automatisch in der Sprache der Model Checker codiert werden und gefundene Gegenbeispiele in die Ebene der Modelle rückübersetzt werden (siehe [Slo00], [PS99]).

Die mit dieser Werkzeugkombination verifizierbaren Eigenschaften werden durch die Möglichkeiten der verwendeten Model Checker bestimmt. Diese bieten die Computation Tree Logic (CTL) und die Linear Temporal Logic (LTL) an. Die unterstützten Modellelemente hängen von der Realisierung der Integration ab, d.h. welche Modelle von der Kopplung unterstützt werden und welche nicht. [AFMC] enthält eine detaillierte Beschreibung der unterstützten Modellelemente und der erlaubten Eigenschaften. Interessant ist dabei vor allem die Möglichkeit Eigenschaften über ein Modell zu spezifizieren, dazu wird die eingesetzte temporale Logik mit Operatoren wie **always**, **next**, **sometimes** und **until** um Elemente aus dem Modell erweitert. Dies sind die unterstützten Operationen (+,-,=,..) und die Elemente aus dem Modell (Zustände, Ports und Variablen von Komponenten). Der Property-Editor zur Eingabe der Eigenschaften (siehe Abbildung 5) unterstützt die Formulierung der Eigenschaften durch kontext- und modellabhängige Menüs.

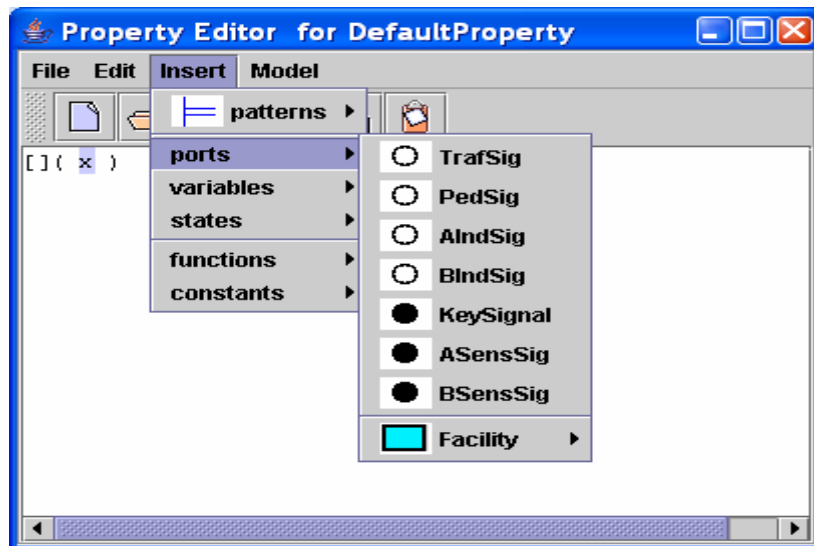


Abbildung 5: Property Editor von AutoFOCUS

Ein Beispiel für eine zu verifizierende Eigenschaft ist:  $[ ](\text{val}(\text{PL}) == \text{stop}) \Rightarrow \langle \rangle(\text{val}(\text{PL}) == \text{walk})$ . Diese bedeutet: immer wenn die Fußgängerampel  $\text{PL}$  den Wert  $\text{stop}$  hat, irgendwann die Ampel auch wieder auf  $\text{walk}$  schalten wird. Der Grund für Unkorrektheit dieser Aussage in dem Modell liegt darin, dass die Fußgängerampel nur auf Anforderung auf  $\text{walk}$  schaltet und daher die Aussage falsch ist.

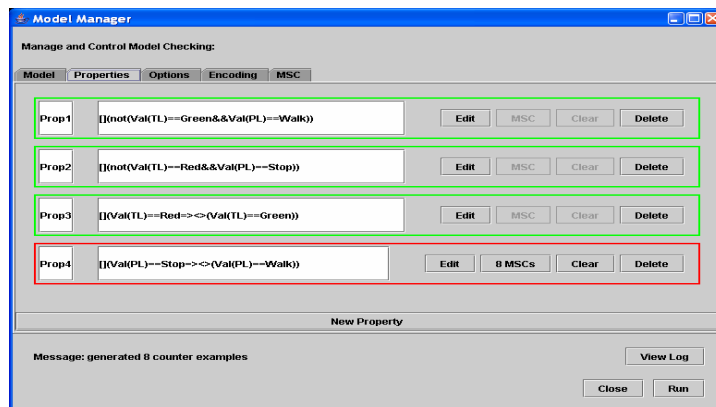


Abbildung 6: Verifikationszustände von Eigenschaften

Mehrere Zustände von Eigenschaften sind möglich und werden von der Integration durch Farben visualisiert: unverifiziert (grau), erfolgreich verifiziert (grün), falsifiziert (rot), verifiziert bis zu einer Schranke (orange). Der letzte Zustand kann nur bei einer Verifikation mit Bounded Model Checking erreicht werden. Die Falsifikation von Eigenschaften kann durch syntaktische oder Typisierungsfehler, oder durch die Angabe eines Gegenbeispiels erfolgen. Abbildung 6 zeigt ein Beispiel für Verifikationszustände der Model Checking Integration von AutoFOCUS.

Die Anwendbarkeit von Model Checking Integrationen auf Modelle hängt wesentlich von der Anzahl der erreichbaren Zustände ab (siehe Abschnitt 2.4.2). Als Abschätzung der Anzahl der möglichen Zustände kann folgendes Berechnungsschema als obere Grenze verwendet werden:  $\text{Eingaben} \wedge \text{Schritte}$ , dabei ist  $\text{Eingaben}$  die Anzahl aller möglichen

Eingaben in einem Berechnungsschritt des Modells, und *Schritte* die maximal zu betrachtende Länge der Berechnungen. Eingaben berechnet sich als Produkt  $\prod I_{n_i}$ , wobei  $I_{n_i}$  alle Möglichkeiten des  $i$ -ten Eingabeports enthält. Dies kann entweder kein Wert (leere Eingabe) sein, oder einer der möglichen Werte (je nach Datentyp). Ist beispielsweise der Typ des  $i$ -ten Ports `bool`, so hat  $I_{n_i}$  den Wert 3.

Im Beispiel einer Ampel mit zwei Fußgängerknöpfen, also zwei Ports, vom ein-elementigen Typ `signal` ( $I_n=2$ ) können also pro Schritt 4 unterschiedliche Eingaben anliegen. Bei einer maximalen Sequenzlänge von 10 Schritten bedeutet dies dass  $4^{10}$  (ca. 1 Million) mögliche Sequenzen betrachtet werden. Der erreichbare Zustandsraum ist allerdings wesentlich geringer, da die beiden Eingänge in einer Komponente zusammengefasst werden und mit lokalen Variablen und Zuständen zusammengefasst werden. Bei Kenntnis der Modellierung kann man der erreichbare Zustandsraum auch mit (*Zustandswerte*  $\times$  *Variablenwerte*)  $\times$  *Eingaben* abgeschätzt werden.

Ein weiterer Einflussfaktor ist die Komplexität der Zustandsübergänge, die in die Transitionsrelation umgesetzt wird, und in die Berechnungszeit der erreichbaren Zustände eingeht. Insbesondere umfangreiche Datentypen und Funktionen erhöhen die Komplexität der Transitionsrelationen. Die Ausführungszeit der Model Checker wird gemessen und protokolliert. In der Praxis zeigt sich allerdings der Erfahrungswert, dass Model Checking meist entweder mit sehr kurzer Ausführungszeit oder aber gar nicht funktioniert, also in der Regel für kleine und mittelgroße Modelle praktikabel ist. Der Anwendungsbereich des Model Checkings ist daher die Verifikation von zustandsintensiven aber kleinen Systemen. Regelungstechnische Systeme sind wegen der vielen kontinuierlichen Variablen und der großen Anzahl der zu betrachtenden Schritte schlecht mit Model Checking verifizierbar.

### 3.4 Modellbasierte Testfallgenerierung mit AutoFOCUS

Das Werkzeug AutoFOCUS wird hier als ein Beispiel beschrieben, weil es mehrere frei verfügbare Methoden zur modellbasierten Testfallgenerierung enthält (siehe [AF], [AFMC], [AFTIG]) und mit einem kleinen Subset an Modellierungstechniken auskommt, der von vielen Modellierungswerkzeugen unterstützt wird. Somit sind die hier vorgestellten Methoden oft einfach übertragbar auf andere Modellierungswerkzeuge.

Im Gegensatz zu den anderen vorgestellten Verifikationsmethoden, die ein Softwaresystem oder ein Modell auf Eigenschaften verifizieren, bietet die modellbasierte Testfallgenerierung nur Unterstützung zur eigentlichen Verifikation durch Test. Die generierten Testfälle werden zur Falsifikation verwendet. Daher können spezielle Testmodelle erstellt werden, um Testfälle zu spezifizieren. Gegenüber der Testfallgenerierung aus „Entwicklungsmodellen“ hat dies folgende Vorteile:

- Modelle können in mehrere Teilmodelle für unterschiedliche Aspekte aufgeteilt werden (kleinere, einfache, unabhängige Modelle,

## Erfahrungsbericht Verifikationstechniken

wesentlich weniger Aufwand als die Erstellung der Entwicklungsmodelle),

- Modelle sind abstrakter, da kein Produktionscode daraus generiert werden muss und
- Modelle sind näher an der Spezifikation (als an der Implementierung) und können so die Spezifikation einfacher validieren als Entwicklungsmodelle (das bedeutet zum Beispiel auch, dass eine Referenzimplementierung zur Falsifikation einer Spezifikation wesentlich ungeeigneter als ein Testgenerierungsmodell ist).

Eine Testfallgenerierung direkt aus den Entwicklungsmodellen ist zwar möglich, jedoch können so keine Entwicklungsfehler entdeckt werden.

Die modellbasierte Testfallgenerierung generiert aus dem Testmodell und Testspezifikationen Testsequenzen, z.B. eine Sequenz zu einem Modell, in der alle Eingänge des Modells nacheinander mit allen möglichen Kombinationen von Grenzwerten stimuliert werden. Es gibt verschiedene Methoden mit unterschiedlich guter Automatisierung für die Generierung der Sequenzen. Die wichtigsten Methoden sind:

- Simulation: Manuelle Eingabe von Werten zur Erreichung einfacher funktionaler Testziele (siehe [AF]).
- Constraint-basierte Suchverfahren: Verwendung einer Suchstrategie zur Erreichung funktionaler oder struktureller Testziele. Meist ist eine Formulierung neuer oder die Anpassung vorhandener Suchstrategien nötig, um Testziele in komplexen Testmodellen zu erreichen (siehe [LP01]).
- Vollständige Suche („Model Checking“): Automatische Suche nach funktionalen Testzielen in kleinen Testmodellen, insbesondere gut geeignet für einzelne Zustands-Diagramme (siehe Abschnitt 3.3).
- Bounded Model Checking: Suche nach Testfällen einer bestimmten Länge, basierend auf einer Umsetzung des Modells in Logik (siehe [WLPS00]).
- Szenario-basiertes Testen: Konkretisierung von lückenhaft vorgegebenen Testabläufen durch Auffüllen der Lücken durch passende Werte. Dies erfordert in der Regel auch eine Suchstrategie oder vollständige Suche (siehe [Wim00]).
- Zufall / Monte Carlo-Methode: Es werden beliebige Eingangswerte an das Testmodell angelegt, um strukturelle Testziele in zustandsarmen Systemen zu erreichen (siehe [AFTIG]).
- Kombinatorische Methoden: Wertebereiche für Eingänge und deren Kombinationen werden spezifiziert. Die Sollwerte werden durch Ausführung des Testmodells ermittelt (siehe [AFTIG]).

Die modellbasierte Testfallgenerierung verwendet Modelle für folgende Systemaspekte:

## Erfahrungsbericht Verifikationstechniken

- Systemstruktur,
- dynamisches Verhalten,
- Datenmodelle und
- Interaktionsszenarien.

Zusätzlich zu diesen Beschreibungstechniken können auch Klassifikationsbäume [CTE] verwendet werden um die Auswahl der Werte für Variablen zu klassifizieren und festzulegen (siehe [Slo00]).

Die kombinatorischen Methoden zur Testfallgenerierung sind für die Testprojekte meist die beste Lösung, da sie mit relativ einfachen Testspezifikationen, relativ schnell, relativ viele Testfälle generieren können. Folgende Varianten der kombinatorischen Testfallgenerierung sind in AutoFOCUS verfügbar und können direkt aus dem AutoFOCUS-Browser für beliebige Komponenten gestartet werden:

- **All Input Generator:** Generiert alle möglichen Kombinationen mit den Default-Werten für die Datentypen.
- **Configurable Input Generator:** Generiert beliebig konfigurierbare Werte und Kombinationen anhand von benutzerdefinierten Spezifikationen („Gruppen-Konfigurations Files“).
- **Connected Input Generator:** Generiert alle Kombinationen von Eingängen, die von derselben atomaren Komponente verarbeitet werden. Er verwendet ebenfalls die Default-Werte für die Datentypen.
- **Pair Input Generator:** Generiert alle Paare von Eingangswerten mit den Default-Werten für die Datentypen.
- **Simple Input Generator:** Generiert für jeden Eingang jeden Wert der Default-Datentypen genau einmal.

Jede Variante generiert eine Testdatei mit Eingaben für die selektierte Komponente und zusätzlich eine Spezifikationsdatei, die von Hand verändert und dann mit dem **Configurable Input Generator** zur weiteren Generierung verwendet werden kann. Dies reduziert den Aufwand zur Erstellung der Testspezifikation deutlich. Die Sollwerte der modellbasierten Tests können aus den Modellen berechnet werden (Simulation oder mit dem generierten Code).

Das modellbasierte Testen hat zwei Grenzen, die eine ist der Aufwand, die Anforderungen (Sollwerte) detailliert zu modellieren, die andere ist die Komplexität des Testziels. Daher ist beim Einsatz von Suchverfahren auf die mögliche Laufzeit der Suche zu achten. Die Grenzen der modellbasierten Testfallgenerierung sind also unterschiedlich, und hängen von den Methoden ab, die zur Erreichung der Testziele eingesetzt werden. Für Systeme mit komplexen Zuständen ist eine Testspezifikation zur Erreichung einer Zustandsabdeckung aufwändiger zu erstellen.

Kombinatorische Methoden haben sich in mehreren Anwendungen der bewährt (siehe z.B. Abschnitt 4.1), da sie gute Automatisierung mit

effizienten Generierungsverfahren kombinieren und einfach anzuwenden sind. Die Länge der generierbaren Testsequenzen kann, je nach Rechnerausstattung etwa 1 Millionen Zeilen lang werden. Damit ist die Länge der Testsequenzen ausreichend lang.

Für ein Prototyping, zum Beispiel zum Test der Testumgebung, hat sich die Simulation als einfachste Methode zur Testfallgenerierung bewährt. Simulation beruht auf manuellen Eingaben und ist daher auf sehr einfache oder wenige Testfälle beschränkt.

Die Einsatzmöglichkeit der vollständigen Suche hängt von der Komplexität der Testmodelle ab. Mehrere einfache Testmodelle sind für viele Testfallgenerierungsmethoden besser geeignet als ein komplexes Testmodell.

Szenario-basiertes Testen kann nur verwendet werden, wenn die Anforderungen in Form von Szenarien/Interaktionssequenzen gegeben sind. Wegen der benötigten Suchstrategie ist Szenario-basiertes Testen ebenso aufwändig wie die Suchverfahren zur Testfallgenerierung.

## 4 Erfahrungen und Success-Stories

Trotz der im vorherigen Abschnitt beschriebenen Grenzen der Verifikationstechniken wurden diese in vielen Entwicklungsprojekten erfolgreich eingesetzt. Einige Beispiele dafür werden in diesem Abschnitt kurz charakterisiert und referenziert.

### 4.1 Modelbasierte Testfallgenerierung

In einer Pilotanwendung wurde von der Validas AG ein Testmodell erstellt, das wesentliche Teile einer Smart-Card-Anwendung enthält. Mit Constraint-basierten Testfallgenerierungsmethoden wurden aus dem Testmodell Testfälle generiert. Diese Testfälle wurden in die Testumgebung von Giesecke & Devrient für Smart-Cards eingespeist und durchgeführt. Bemerkenswert bei dieser Fallstudie war die große Anzahl (60.000) der generierten Testfälle und deren hohe Abdeckung in dem Code der SmartCard (siehe [PPS03]).

### 4.2 Model-Checking

Anhand einer von BMW mit dem Modellierungswerkzeug ASCET entwickelten Komponente der Aktivlenkung wurde eine formale Verifikation durchgeführt. Die Integration des Model Checking-Werkzeugs erfolgte durch ein prototypisch realisiertes Werkzeug zum Einlesen und Umsetzen des von ASCET generierten C-Codes. Basierend auf dieser Integration konnten mit der Verifikationsumgebung einige sicherheitskritische Eigenschaften spezifiziert und verifiziert werden (siehe [DSWS03]). Besonders erwähnenswert bei der Fallstudie war die „Regressionsfähigkeit“ der Werkzeugumgebung, die eine Verifikation von neu entwickelten Versionen ohne weitere Interaktionen ermöglichte. In einer Vorversion des Systems konnte laut [DSWS03], ein Fehler mit einem Gegenbeispiel nachgewiesen werden.

### 4.3 Symbolische Analyse

Die Validas AG hat den Polyspace Verifier im Rahmen einer Fehlersuche eingesetzt. Gesucht wurde, wie bei der genaueren Analyse später festgestellt wurde, eine sporadische Nichtterminierung eines Programms. Der Fehler wurde mit dem Polyspace Verifier nicht gefunden. In einem parallel durchgeführten Code-Review wurde der Fehler in einem Timer entdeckt, der bei einem von 65536 möglichen Startwerten nicht korrekt terminierte. Der Fehler wurde mit Ersetzung eines  $\leq$  durch ein  $<$  in der Routine behoben. Fehler dieser Art, die in einer Nichtterminierung resultieren, können mit dem Polyspace Verifier nicht entdeckt werden.

Der Begriff des Laufzeitfehlers von Polyspace ist auf die Fehlerursache eingeschränkt, so dass Fehler mit Auswirkungen auf das Laufzeitverhalten auch nicht gefunden werden können. Auch Stack-Überläufe sind Laufzeitfehler, die mit Polyspace nicht erkannt werden können. Demzufolge kann Polyspace bei der Fehlersuche nach Laufzeitfehlern nur eine Ergänzung von anderen Verfahren sein.

Unter [http://www.polyspace.com/case\\_studies.htm](http://www.polyspace.com/case_studies.htm) sind zahlreiche Success-Stories des Einsatzes der symbolischen Analyse beschrieben, mit Code-Größen bis zu 200.000 Lines of Code. Schwerpunkt dieser Anwendungen ist eine Verbesserung der Code-Qualität im Allgemeinen.

## 5 Einsatzempfehlung

Verifikationstechniken sind generell geeignet um die Qualität zu verbessern und Fehler aufzuzeigen. Die meisten Software-Entwicklungsprozesse verwenden nicht alle Verifikationstechniken und können mit zusätzlichen Methoden erweitert werden. Allerdings müssen die neuen Methoden sinnvoll sein, d.h. der Mehraufwand zur Verifikation muss geringer sein, als ein Mehraufwand mit den bisherigen Methoden (z.B. Review) um die gleiche Qualitätsverbesserung zu erzielen.

Daher hat sich bei der Auswahl zusätzlicher Verifikationstechniken eine einfache Prozess-Integration bewährt. Wird beispielsweise C-Code entwickelt, so lassen sich in den Compilierungsprozess relativ einfach Werkzeuge zur Prüfung von MISRA-Regeln oder symbolischen Analyse integrieren. Ähnlich einfach und effektiv ist auch eine Überprüfung von Java-Programmierrichtlinien. Eine formale Verifikation mit einer zusätzlichen Erstellung der Modelle wird in solchen Szenarien üblicherweise abgelehnt.

Wird ein modellbasierter Entwicklungsprozess mit Code-Generierung angewendet, so ist eine Überprüfung von Modellierungsrichtlinien und eine Verifikation von Eigenschaften einfach ermöglicht (vorausgesetzt es gibt für das eingesetzte Modellierungswerkzeug Verifikationstechniken). Eine Überprüfung von Programmierrichtlinien auf dem generierten Code ist in der Regel nicht so ergiebig, wie eine Modell-Überprüfung. Ausnahme ist derzeit die symbolische Analyse, da diese auf Modellen noch nicht verfügbar ist, und somit auch in generiertem Code Fehler wie Überläufe von Werten in Modellen nachgewiesen werden können.

Für manche Anwendungsgebiete (z.B. sicherheitskritische Software) schreiben Standards wie die IEC 61508 den Einsatz von Verifikationstechniken vor, so dass die Frage nach dem Einsatz der Verifikationstechniken den Marktzugang regelt und daher keine Alternativen möglich sind. Gerade in diesem Fall ist allerdings eine sorgfältige Auswahl der Entwicklungs- und Verifikations-Werkzeuge wichtig um die Prozesse so einfach und effektiv wie möglich zu gestalten.

## 6 Zusammenfassung

Trotz der prinzipiellen Grenzen der formalen und modellbasierten Verifikationsmethoden sind diese für den praktischen Einsatz geeignet. Voraussetzung ist allerdings eine gezielte und für den Anwendungsbereich zugeschnittene Auswahl der Techniken. Viele Techniken ergänzen sich auch, beispielsweise liefert die symbolische Analyse nachweisbar bessere Ergebnisse aus MISRA konformen Code als auf beliebigen Programmen. Die modellbasierte Testfallgenerierung kann über Werkzeugkombinationen zur Verifikation von (anderen) Modellen bzw. Softwaresystemen verwendet werden, und die Testfallgenerierung aus endlichen Modellen kann Model Checking Verfahren verwenden.

In Zukunft werden modellbasierte Entwicklungswerkzeuge vermehrt eingesetzt werden, und auch die Validierung und Zertifizierung wird vielfach auf Modell-Ebene erfolgen. Vergleichbar mit dieser Entwicklung ist der Einzug von Hochsprachen in die Software-Entwicklung.

Allerdings sind auf diesem Weg auch noch einige Schritte zu gehen um den Entwicklern auf Modell-Ebene den gleichen Komfort wie auf Programm-Ebene zu bieten (z.B. Markierung von Fehlern, automatische Vervollständigung bei der Eingabe, Modellierungsmöglichkeiten von Varianten).

Es werden auch neue modellbasierte Verifikationstechniken entstehen, die die Vorteile der Modellbasierung ausnützen. So ist beispielsweise eine symbolische Analyse von Zustands- und anderen Invarianten, die auf Modellebene formulierbar sind, möglich.

## 7 Literatur

- [AbsInt] AbsInt, angewandte Informatik, [www.absint.de](http://www.absint.de)
- [AF] AutoFOCUS, <http://autofocus.in.tum.de>
- [AFMC] AutoFOCUS User Manual: Model Checking, Validas AG, 2005
- [AFTIG] AutoFOCUS User Manual: Test Input Generator, Validas AG, 2002
- [ATP] Überblick über automatische Beweisverfahren, Geoff Sutcliffe, <http://www.cs.miami.edu/~tptp/OverviewOfATP.html>
- [Balz01] H. Balzert: Software-Entwicklung. 2. Auflage. Spektrum Akademischer Verlag, Heidelberg 2001
- [Boehm81] Boehm B., Software Engineering conomics, Englewood Cliffs, N.J. Prentice Hall, 1981

## Erfahrungsbericht Verifikationstechniken

- [CadSMV] Cadence SMV, <http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/>
- [CC] Common Criteria, 1999, <http://www.commoncriteriaportal.org/>
- [CkSytle] Checkstyle from [checkstyle.sourceforge.net](http://checkstyle.sourceforge.net)
- [CTE] Grochtmann, Grimm: Tool-Supported Test Case Design for Black-Box Testing by Means of the Classification-Tree Editor, 1993, <http://www.systematic-testing.com/documents/eurostar1993.pdf>
- [DSWS03] W. Damm, C. Schulte, H. Wittke, M. Segelken (OFFIS); U. Higgen, M. Eckrich (BMW AG): Formale Verifikation von ASCET Modellen im Rahmen der Entwicklung der Aktivlenkung. GI Workshop Automotive Software Engineering, 2003
- [HIS] Programmierrichtlinien der Hersteller Initiative Software (HIS) [http://www.automotive-his.de/download/HIS\\_SubSet\\_MISRA\\_C\\_1.0.3.pdf](http://www.automotive-his.de/download/HIS_SubSet_MISRA_C_1.0.3.pdf).
- [IEC] IEC 61508: Functional safety of electrical / electronic / programmable electronic safety-related systems
- [Isabelle] Isabelle Beweis System, <http://isabelle.in.tum.de/>
- [ITSEC] Kriterien für die Bewertung der Sicherheit von Systemen in der Informationstechnik, 1991
- [JavaConv] Code Conventions for the Java Programming Language, <http://java.sun.com/docs/codeconv>
- [Lig02] Liggesmeyer, P., Software-Qualität: Testen Analysieren und Verifizieren von Software. Berlin: Spektrum Akademischer Verlag, 2002
- [LP01] Lötzbeyer, H., Pretschner, A.,: Model Based Testing with Constraint Logic Programming: First Results and Challenges Proc. 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification (WAPATV'01), Toronto, May 2001
- [MISRA] Guidelines For The Use Of The C Language in Vehicle Based Systems, The Motor Industry Software Reliability Association, 1998, <http://www.misra.org.uk/>
- [MTest] MTest von SPACE, <http://www.dspace.com>
- [NuSMV] NuSMV, <http://nusmv.irst.itc.it/>
- [Pie02] Benjamin C. Pierce, Types and Programming Languages, MIT Press, 2002
- [Pie05] Benjamin C. Pierce, ed., Advanced Topics in Types and Programming Languages , MIT Press, 2005
- [PPS03] Jan Philipps, Alexander Pretschner, Oscar Slotosch, Ernst Aiglstorfer, Stefan Kriebel, and Kai Scholl. *Model-based test case generation for smart cards*. In Formal Methods for Industrial Critical Systems, ENTCS, Vol. 80., pages 168–182, June 2003.
- [PS99] The Quest for Correct Systems: Model Checking of Diagramms and Datatypes, Proceedings of Asia Pacific Software Engineering Conference 1999, 449-458, Philipps, Slotosch
- [PSPC] PolySpace Technologies, <http://www.polyspace.com>

## Erfahrungsbericht Verifikationstechniken

- [Prover] Prover Technologies, <http://www.prover.com/>
- [PVS] Specification and Verification System, <http://pvs.csl.sri.com/>
- [QA-C] QA-C/MISRA <http://www.qa-systems.de>
- [SATO] <http://www.cs.uiowa.edu/~hzhang/sato.html>
- [Spin] Model Checker Spin, <http://spinroot.com/>
- [SPLint] Secure Programming Lint, <http://www.splint.org/>
- [SRS04] AFS verification results module FailSafe in diverse discrete version Preliminary version, Marc Segelken, Arun C. Rao, Christoph Schulte, 26th April 2004
- [STeP] The Stanford Theorem Prover, <http://www-step.stanford.edu/>
- [Wim00] Specification Based Determination of Test Sequences in Embedded Systems, Guido Wimmel, Diploma Thesis, Technische Universität München, 2000
- [SCADE] SCADE Suite, <http://www.esterel-technologies.com/>
- [Slo00] Modelling and Validation: AutoFocus and Quest, Oscar Slotosch, Formal Aspects of Computing, 2000 12:225-227
- [SMV] Symbolic Model Verifier, <http://www.cs.cmu.edu/~modelcheck/>
- [Tools] CASE Tools for Embedded Systems [pdf-Version] Bernhard Schätz, Tobias Hain, Wolfgang Prenninger, Martin Rappl, Jan Romberg, Oscar Slotosch, Martin Strecker, Alexander Wisspeintner, et.al. Technical Report TUMI-0309, Fakultät für Informatik, TU München, 2003 <http://www4.in.tum.de/~schaetz/papers/TUMI-0903.pdf>
- [USC-ES] Statemate ModelChecker, USC-ES, <http://www.osc-es.de>
- [VIS] Verification Interacting with Synthesis <http://www-cad.eecs.berkeley.edu/~vis/>
- [Wim00] Specification Based Determination of Test Sequences in Embedded Systems, Guido Wimmel, Diploma Thesis, Technische Universität München, 2000
- [WLPS00] Wimmel, G., Lötzbeier, H., Pretschner, A., Slotosch, O.: Specification Based Test Sequence Generation with Propositional Logic, Journal on Software Testing, Validation, and Reliability (STVR) 10(4):229-248, 2000
- [ZChaff] <http://www.princeton.edu/~chaff/zchaff.html>