

Hauptseminar Automotive Software Engineering Testen, Rapid Prototyping und X - in the loop

Christoph Niedermayr

20.01.2005

Inhaltsverzeichnis

1	Motivation	3
2	Allgemeine Verfahren des Softwaretests	4
2.1	Der statische Test	4
2.1.1	Das Review	4
2.1.2	Statische Analyse	6
2.2	Der dynamische Test	7
2.2.1	Blackbox Verfahren	8
2.2.2	Whitebox Verfahren	11
3	Spezielle Testverfahren im Automotive	13
3.1	Model-in-the-loop	13
3.2	Software-in-the-loop	14
3.3	Processor-in-the-loop	15
3.4	Hardware-in-the-loop	15
3.5	Rapid Prototyping	16
A	Regeln für eine Reviewsitzung	17
B	Bildung eines Übergangsbaums	18

1 Motivation

Im Rahmen der Qualitätssicherung nimmt das Testen von Software heutzutage eine bestimmende Stellung ein. Testen beschränkt sich nicht mehr nur auf eine Phase die auf die Implementierung folgt, sondern stellt eine eigenständige Aufgabe dar, der in verschiedensten Stadien des Entwicklungsprozesses nachgegangen werden muss.

Zu welchen Konsequenzen unzureichendes, unsystematisches oder – im schlimmsten Falle – nicht erfolgtes Testen führen kann illustrieren folgende aktuelle Beispiele:

- Im Juni 2004 ruft Mercedes europaweit rund 10.000 Fahrzeuge der Modelle Vito und Viano in die Werkstätten zurück. In den Dieselvarianten könnte ein Softwarefehler dazu führen, dass das Dieselsteuergerät die Spritzzufuhr abstellt und der Motor ausgeht.[1]
- Im August 2002 werden in den USA 20.500 3er BMWs aus dem Jahr 1999 softwaremäßig upgedatet: Die Airbags lösten mitunter aus, wenn die Autos mit hoher Geschwindigkeit auf einen Bordstein fuhren oder in ein Schlagloch gerieten. BMW musste schon einmal knapp 35.000 3er-Fahrzeuge aus den Jahren 1998 und 1999 wegen eines derartigen Problems zurückrufen.[2]
- Im Mai 2002 werden 7er BMWs zurückgerufen weil ein Softwarefehler die Benzineinspritzpumpe dazu veranlassen kann, zu wenig oder gar keinen Kraftstoff mehr zu befördern.[3]
- Der fehlgezündete Beifahrer-Airbag eines VW Golf tötete 1999 ein Kind. Die Ursache: Möglicherweise ein Softwarefehler![4]

2 Allgemeine Verfahren des Softwaretests

In diesem Abschnitt werden allgemeine Verfahren des Testens von Software vorgestellt. Dabei werden die Verfahren nicht in Bezug auf das zu testende Objekt oder die Entwicklungsphase, in der der Test durchgeführt wird gegliedert, sondern danach ob der Test unter Ausführung des Testobjekts durchgeführt wird, oder nicht.

Grundlegende Informationen zum Thema Softwaretest wurden [6, Kapitel 3] entnommen.

2.1 Der statische Test

Bei den *statischen Testverfahren* handelt es sich um Tests bei denen das Testobjekt nicht zur Ausführung gebracht wird. Geeignete Testobjekte für statische Tests sind alle Dokumente die im Entwicklungsprozess anfallen, einschließlich des Sourcecodes. Falls Dokumente einer formalen Struktur unterliegen ist es auch möglich werkzeuggestützte Tests einzusetzen.

2.1.1 Das Review

Beim *Review* handelt es sich um eine manuelle Untersuchung (Lesen und Nachvollziehen) von Dokumenten durch eine Gruppe von Personen. Typischerweise werden Reviews dazu eingesetzt, Fehler in Verträgen, Spezifikationen, Handbüchern oder auch im Sourcecode festzustellen. Zeitlich sollten Reviews möglichst bald nach Fertigstellung des zu untersuchten Dokuments durchgeführt werden, wobei eine werkzeuggestützte Analyse des Testobjekts noch vor dem Review erfolgen sollte um die im Review diskutierten Punkte zu minimieren.

Zeitlicher Aufbau

Ein Review kann in folgende fünf Phasen unterteilt werden:

1. *Planung*: Auswahl der zu untersuchenden Dokumente und der teilnehmenden Personen, sowie Festlegung von Ort und Zeit des Reviews.
2. *Einführung*: Information der am Review beteiligten Personen über Zweck des Reviews sowie Bereitstellung der für das Review benötigten Referenzdokumente (Pflichtenheft, Standards, Richtlinien).
3. *Vorbereitung*: Vorbereitung der Teilnehmer durch intensive Auseinandersetzung mit dem zu prüfenden Dokument sowie den Referenzdokumenten.
4. *Reviewsitzung*: Durchführung des Reviews unter Leitung eines Moderators.

Das Review sollte nach festen Regeln¹ durchgeführt werden. Ein Protokoll muss erstellt werden.

5. *Nachbereitung*: Beseitigung der festgestellten Mängel sowie Überprüfung der Beseitigung. Bei nicht akzeptablem Ergebnis sollte ein weiteres Review durchgeführt werden.

Rollenverteilung

Die an einem Review beteiligten Personen können nach ihrer Funktion in folgende Gruppen gegliedert werden:

- *Manager*: Durchführung der Planung des Reviews sowie Festlegen des weiteren Vorgehens nach dem Review. Sollte wegen möglicherweise geringerem Fachwissens nicht am Review selbst teilnehmen. Ausserdem könnte durch die Teilnahme des Managers die freie Diskussion unter den anderen Teilnehmern eingeschränkt werden.
- *Moderator*: Unvoreingenommen und unbeteiligter Leiter der Diskussion.
- *Autor*: Der Ersteller (oder ein benannter Hauptverantwortlicher, falls es mehrere Ersteller gibt) des zu prüfenden Dokuments.
- *Gutachter*: Bis zu fünf Fachexperten die sowohl die für gut befunden Teile des Dokuments als auch mangelbehaftete Teile entsprechend kennzeichnen. Um die Effizienz des Reviews zu erhöhen sollten sich einzelne Gutachter auf spezielle Aspekte konzentrieren.
- *Protokollant*: Führt das Protokoll über das Review, in dem die gefundenen Mängel aufgelistet werden. Aus pragmatischen Gründen kann der Protokollant mit dem Autor identisch sein.

Review-Arten

Es gibt verschiedene Ausprägungen des Reviews, wobei in der Praxis eine klare Abgrenzung, so wie hier dargestellt, nicht möglich ist. Vielmehr ist die tatsächliche Form eines Reviews firmen- und sogar projektabhängig, was zu einer Verbesserung des Wirkungsgrades führt, da das Review an die herrschenden Bedürfnisse angepasst wird.

- *Walkthrough*: Der Walkthrough ist nur für kleine Entwicklungsteams und unkritische Dokumente geeignet und zielt vor allem darauf ab, Wissen über das Testobjekt zu verbreiten, Lösungsalternativen zu diskutieren, Einhaltung von Spezifikationen und Standards zu prüfen und Mängel durch spontane Fragen aufzudecken. Es finden kaum Nachbereitung und Vorbereitung statt.

¹siehe Anhang A

- *Inspektion*: Die Inspektion hat einen formalen Ablauf mit fest verteilten Rollen. Die Gutachter haben sich auf das Review vorzubereiten und müssen Checklisten zu den einzelnen Prüfkriterien, die in der Planung festgelegt wurden, erstellen. Ein Gutachter stellt den Inhalt des Testobjekts vor, wobei die Gutachter Fragen stellen, die vom Autor beantwortet werden. Der Moderator sorgt dafür, dass sich die Diskussion auf die Prüfkriterien beschränkt. Der Protokollant hält die festgestellten Mängel fest, wobei Abweichungen von Spezifikationen oder Standards separat behandelt werden. Die Inspektion wird vor allem zur Überprüfung von Sourcecode eingesetzt, weshalb man auch von *Design-* oder *Codereview* spricht.
- *Technisches Review*: Das Ziel des technischen Reviews besteht darin, die Eignung des Testobjekts für den beabsichtigten Einsatz, die Übereinstimmung mit der Spezifikation sowie die Einhaltung von Standards zu überprüfen. Dazu werden unbeteiligte Fachexperten als Gutachter herangezogen, die im Vorfeld der Sitzung das Testobjekt mittels der offiziellen Spezifikation prüfen. Der Moderator priorisiert die resultierenden Bewertungen und stellt die wichtigsten in der Sitzung zur Diskussion. Das Urteil der Sitzung muss einstimmig von allen Beteiligten akzeptiert werden.
- *Informelles Review*: Das informelle Review wird in der Regel durch den Autor geplant und beinhaltet keine Sitzung sondern beschränkt sich auf das Gegenlesen eines Dokuments durch Kollegen. Diese teilen dem Autor ihre Bemerkungen bis zu einem festgelegten Termin schriftlich mit. Aufgrund des geringen Aufwands ist diese Form des Reviews in der Praxis stark verbreitet.

2.1.2 Statische Analyse

Die *statische Analyse* ist die werkzeuggestützte Überprüfung eines Dokuments, das einer formalen Struktur unterliegt. In der Praxis ist der Programmcode das einzige Prüfobjekt für eine statische Analyse. Der Compiler ist ein Beispiel für ein Analysewerkzeug: Er überprüft mindestens die Einhaltung der Syntax und möglicherweise auch Kriterien wie Typsicherheit, nicht erreichbarer Code, fehlende Variablendeklaration oder die Einhaltung von Arraygrenzen.

Es ist auch möglich die Einhaltung von Standards mittels Analysatoren zu überprüfen. Dies kann zu einer höheren Bereitschaft zur Umsetzung von Programmierkonventionen durch die Programmierer führen, wenn diese mit einer automatisierten Prüfung ihres Codes rechnen müssen.

Datenflussanalyse

Die *Datenflussanalyse* ist eine Art des statischen Tests, bei der die Verwendung von Daten auf Pfaden durch den Programmcode untersucht wird. Dazu werden den verwendeten Variablen folgende Zustände zugeordnet:

- *definiert*: Zuweisung eines Wertes

- *referenziert*: Verwendung der Variable
- *undefiniert*: Variable hat keinen definierten Wert

Man kann nun folgende Datenflussanomalien identifizieren, die jedoch nicht immer zu einem fehlerhaftem Verhalten des Programms führen:

- *ur*: Verwendung einer Variable mit undefiniertem Wert.
- *du*: Zuweisung eines Wertes, der nicht verwendet wird bevor die Variable ungültig wird.
- *dd*: Überschreibung einer Variablen, ohne dass der vorherige Wert verwendet wurde.

Kontrollflussanalyse

Zur Durchführung einer *Kontrollflussanalyse* benötigt man ein Werkzeug, das den Programmcode in einen gerichteten Graphen transformiert. Hierbei werden Verzweigungsstellen im Code als Kanten und Sequenzen von Anweisungen als Knoten dargestellt. Da dieser Kontrollflussgraph wesentlich anschaulicher als der Programmcode selbst ist, lassen sich Fehler im Kontrollfluss des Programms leichter erkennen. Ist der Graph unübersichtlich sollten die betreffenden Teile des Programms überarbeitet werden, da komplexe Ablaufstrukturen ohnehin fehlerträchtig sind.

Metriken

Aus den Daten, die bei der automatisierten Prüfung von Programmcode anfallen, können Maßzahlen für die Güte des Testobjekts berechnet werden. Jede dieser sogenannten *Metriken* liefert einen Anhaltspunkt für die Qualität eines bestimmten Aspektes des untersuchten Objekts. Ein Beispiel ist die *zyklomatische Zahl* v , die die Komplexität des Kontrollflusses, und damit die Wartbarkeit und Testbarkeit, beurteilt (nach T.J. McCabe). Ein Programmstück sollte überarbeitet werden, wenn die zugehörige zyklomatische Zahl größer zehn ist. Sie berechnet sich für einen Kontrollflußgraph G aus der Anzahl der Knoten n , der Anzahl der Kanten e und der Anzahl der verbundenen Komponenten (Funktionen) p wie folgt:

$$v(G) = e - n + 2p$$

2.2 Der dynamische Test

Beim *dynamischen Test* sollte das Testobjekt als ablauffähiges Programm vorliegen. Falls das nicht der Fall ist, kann das Testobjekt in ein *test bed* eingebettet werden. Dieses übernimmt dann die Funktionalität der fehlenden Programmteile und versorgt das Testobjekt mit Eingabedaten. Ruft das Testobjekt weitere Programmteile auf, die zum Zeitpunkt des Tests noch nicht fertig implementiert sind, werden diese

durch Platzhalter (*stubs*) ersetzt, die das Verhalten der fehlenden Programmteile simulieren.

Liegt nun ein ablauffähiges Testobjekt vor, so müssen auf systematische Weise Testfälle generiert werden, um eine möglichst große Anzahl an möglichen Situationen abzudecken. Die Vorgehensweisen zur Testfallgenerierung können in zwei Kategorien eingeteilt werden: Blackbox und Whitebox Verfahren.

2.2.1 Blackbox Verfahren

Bei *Blackbox Verfahren* wird von einer äußeren Sicht auf das Testobjekt ausgegangen. Es liegen keine Informationen über den inneren Ablauf des Testobjekts vor und somit ist auch keine Steuerung des inneren Ablaufs möglich. Die einzige Beeinflussungsmöglichkeit ist die Wahl der Eingabedaten. Daher finden diese Verfahren vor allem in höheren Phasen des Testens (zum Beispiel Systemtest) Anwendung. Die Testfälle werden anhand der Anforderungen an das Testobjekt und der Spezifikation des Testobjekts generiert. Da die Kombinationsmöglichkeiten der Eingabedaten in der Regel sehr groß sind, bedarf es Verfahren zur sinnvollen Auswahl von Testfällen.

Bei Blackbox Tests können keine Fehler in der Spezifikation oder den Anforderung aufgedeckt werden, das diese ja als Referenz für das Verhalten des Testobjekts dienen. Hierfür ist auf die →Reviews zu verweisen.

Es nicht ebenfalls nicht möglich das Testobjekt daraufhin zu Untersuchen, ob es über die Spezifikation hinausgehende Funktionalität – und damit möglicherweise Sicherheitslücken – bietet.

Äquivalenzklassenbildung

Die Menge aller möglicher Eingabedaten wird in *Äquivalenzklassen* unterteilt. Alle Elemente einer Äquivalenzklasse müssen zum gleichen Verhalten des Testobjekts führen, so dass der Test eines Repräsentanten einer Äquivalenzklasse den Test aller Elemente ersetzen kann. Es müssen sowohl Äquivalenzklassen für gültige, als auch für ungültige Eingabedaten identifiziert werden. Ein Testfall besteht nun aus einem Repräsentanten einer Äquivalenzklasse, dem erwarteten Ergebnis und, falls nötig, der Vorbedingung des Tests.

Bei der Wahl des Repräsentanten empfiehlt es sich, die Intervallgrenzen der Äquivalenzklassen zu wählen, da diese oftmals aus der Spezifikation generiert werden und es hierbei zu Unstimmigkeiten zwischen der umgangssprachlichen Formulierung und der mathematischen Darstellung kommen kann (ist bei der Formulierung „bis zu x Meter“ x noch gültig oder nicht?).

Es ist auch möglich, Äquivalenzklassen der Ausgabedaten zu identifizieren, jedoch fällt die Auswahl des Eingaberepräsentanten schwieriger, da zu jedem Ausgabewert die Menge der ihn erzeugenden Eingabewerte ermittelt werden muss.

Da ein Testobjekt in der Regel über mehrere Eingabeparameter verfügt, und jeder dieser Parameter mindestens zwei Äquivalenzklassen besitzt (gültige und ungültige Eingabewerte), stellt sich die Frage wie die gewählten Repräsentanten zu Testfällen kombiniert werden sollen. Würde man das kartesische Produkt der Mengen der Repräsentanten verwenden, so erhielte man bereits bei wenigen Parametern eine große Anzahl an Testfällen, weshalb man folgende Regeln zur Einschränkung der Testfälle verwendet:

- Ein Repräsentant einer ungültigen Äquivalenzklasse ist nur mit Repräsentanten gültiger Äquivalenzklassen zu kombinieren.
- Die Menge aller Testfälle wird nach Wahrscheinlichkeit des Eintretens sortiert und nur die benutzungsrelevanten Fälle werden übernommen.
- Testfälle die Grenzwerte von Äquivalenzklassen enthalten werden bevorzugt.
- Alle Paare von Repräsentanten müssen in einem Testfall zur Ausführung kommen.

Grenzwertanalyse

Die *Grenzwertanalyse* dient zur Ergänzung der mittels Äquivalenzklassenbildung generierten Testfälle. Hierbei werden besonders die Grenzen der vorher identifizierten Äquivalenzklassen betrachtet. An jedem Rand einer Äquivalenzklasse wird sowohl der Grenzwert, als auch der benachbarte noch in der Äquivalenzklasse liegende Wert getestet. Weiterhin können die so gefundenen Grenzen noch mit den Grenzen die der Rechner für den zugehörigen Datentyp aufweist, ergänzt werden. Wie bei der Äquivalenzklassenbildung können nun die gefundenen Grenzen zu Testfällen kombiniert werden.

Oftmals sind die Äquivalenzklassen keine Intervalle sondern Mengen. In diesem Fall ist mangels Grenzen die Grenzwertanalyse nicht anwendbar.

Zustandsbezogener Test

In keiner der bisher vorgestellten Methoden wurde berücksichtigt, dass der Zustand in dem sich das System befindet möglicherweise Einfluß auf das Ergebnis von Berechnungen hat. Diese Lücke schließt der *zustandsbezogene Test*. Hierzu wird das Verhalten des Testobjekts als Zustandsautomat modelliert. Bei dieser Art von Test nehmen die Vor- und Nachbedingungen des Test eine besonders wichtige Stellung ein, da die Vorbedingung ja gerade den Zustand des Systems darstellt in dem der Test durchgeführt werden soll, und die Nachbedingung den Endzustand nach Durchführung des Tests.

Dieses Vorgehen ist insbesondere für den Test graphischer Benutzeroberflächen geeignet, wobei zum Beispiel Dialoge oder Eingabemasken als Zustand und Benutzereingaben als Zustandsübergänge aufgefasst werden können. Auch objektorientierte Systeme, in denen Objekte verschiedene Zustände einnehmen können, werden bevorzugt auf diese Weise geprüft.

Der zustandsbezogene Test sollte mindestens jeden Zustand erreichen, besser aber alle für einen Zustand spezifizierten Funktionen ausführen. Die hierfür erforderlichen Testfälle gewinnt man aus einem sogenannten Übergangsbaum², der im Gegensatz zum Zustandsautomat sicher zyklensfrei ist. Er enthält bestimmte Zustandswechsel des Zustandsautomaten, es sollten aber mindestens alle Zustände und alle Übergänge des Zustandsautomaten vorkommen.

Weiterhin ist ein Robustheitstest nötig, der prüft wie sich das Testobjekt bei unspezifizierter Verwendung, also zum Beispiel beim Aufruf einer Funktion an einer nicht dafür vorgesehener Stelle, verhält. Dazu wird der Übergangsbaum entsprechend erweitert.

Zu jedem Testfall der dem Übergangsbaum entnommen wurde und den in ihm enthaltenen Zustandsübergängen müssen folgende Informationen dokumentiert werden:

- Der Zustand vor dem Test beziehungsweise Übergang
- Das auslösende Ereignis für den Übergang
- Das durch den Übergang ausgelöste Verhalten
- Der Zustand nach dem Test beziehungsweise Übergang

Ursache-Wirkungs-Graph-Analyse

Bei der *Ursache-Wirkungs-Graph-Analyse* werden Eingaben (Ursache) und deren Wirkungen im System in Beziehung gestellt. Dazu wird folgendermaßen ein Graph erstellt: Ursachen und Wirkungen werden als Knoten dargestellt und nach Zusammenhang verbunden. Die so entstanden Kanten werden mit booleschen Operatoren versehen um auszudrücken ob zum Beispiel eine Wirkung nur mit mehreren Ursachen, oder auch beim Gegenteil (**not**-Operator) eintritt. Anschließend wird der Graph in eine Entscheidungstabelle umgesetzt, aus welcher die Testfälle abgelesen werden können.

Syntaxtest

Unterliegen die Eingaben für das Testobjekt einer Syntax, so können aus deren Regeln Testfälle gewonnen werden. Es sind sowohl Tests möglich die das Verhalten bei Einhaltung als auch bei Verletzung dieser Regeln prüfen.

Zufallstest

Beim *Zufallstest* werden aus der Menge der möglichen Werte für jedes Eingabedatum per Zufall gewählte Repräsentanten getestet. Falls die Eingaben in der Realität einer statistischen Verteilung gehorchen, so ist diese auch für die Repräsentantenwahl zu verwenden.

²Ein Verfahren zur Bildung des Übergangsbaumes findet sich in Anhang B

Smoke-Test

Der *Smoke-Test* ist ein besonders einfaches Verfahren das angewandt wird um die Eignung des Testobjekts für intensiveres Testen festzustellen. Hierfür wird meist nur die Hauptfunktionalität geprüft, wobei auf die Ermittlung von Sollwerten für den Test verzichtet wird, und nur auf offensichtliches Fehlverhalten oder Systemabstürze geachtet wird.

2.2.2 Whitebox Verfahren

Bei *Whitebox* (oder *Glassbox*) *Verfahren* wird von einer inneren Sicht auf das Testobjekt ausgegangen. Der vorliegende Programmtext des Testobjekts dient als Grundlage zur Testfallgenerierung. Daher können beim Whitebox Test unimplementierte Teile der Spezifikation nicht gefunden werden. Anwendungsgebiet für diese Verfahren sind die unteren Teststufen. Wie beim Blackbox Test gibt es auch hier verschiedene Methoden um Testfälle zu erhalten, die im Folgenden vorgestellt werden.

Anweisungsüberdeckung

Ziel des Tests ist es, eine Mindestquote aller Anweisungen des Testobjekts zur Ausführung zu bringen. Dazu wird der Programmcode des Testobjekts in einen Kontrollflußgraphen³ transformiert. Aus diesem sind Testfälle als Kombinationen von Kanten zu generieren, für die das erwartete Verhalten des Testobjekts vorher festgelegt und dann mit dem tatsächlichen Verhalten verglichen werden muss. Die auszuführenden Anweisungen sind dann die durch die gewählten Kanten verbundenen Knoten.

Technisch realisiert wird diese Methode durch das *Instrumentieren* des Codes: Das Programm wird automatisiert um Zähler erweitert die die Häufigkeit des Besuchs einer Anweisung festzustellen.

Bei diesem Vorgehen kann unerreichbarer Code aufgedeckt werden, falls Anweisungen durch keinen Testfall zur Ausführung gebracht werden können. Dagegen ist es zum Beispiel nicht möglich, leere Teile einer `if-then-else` Abfrage – in der Regel wurde dort Code vergessen – festzustellen, da im Kontrollflussgraph kein Knoten für die leere Anweisung existiert.

Zweigüberdeckung

Während bei der Anweisungsüberdeckung die Knoten des Graphen im Mittelpunkt stehen, sind es hier die Kanten. Der Test zielt also darauf ab, eine Mindestquote an Programmverzweigungen (`if`, `case`) abzudecken. Testfälle werden analog zur Anweisungsüberdeckung generiert, wobei als Kriterium für die Kantenauswahl nicht der Besuch von Knoten, sondern das Gehen der Kanten dient.

Im Gegensatz zur Anweisungsüberdeckung werden bei diesem Verfahren auch

³siehe 2.1.2: Kontrollflussanalyse

Testfälle generiert die leere Teile von `if-then-else` Abfragen abdecken, da hier auch für die Kante der leeren Anweisung gegangen werden muss.

Test der Bedingungen

Oftmals handelt es sich bei dem Ausdruck der an einer Verzweigungstelle ausgewertet wird um ein komplexes Gebilde. Bisher wurde nur die Auswertung selbst, aber nicht deren Zustandekommen aus atomaren Teilbedingungen⁴ berücksichtigt. Dieses kann man in verschiedener Intensität prüfen.

- Bei der *einfachen Bedingungsüberdeckung* wird gefordert, dass jede atomare Teilbedingung im Test sowohl den Wert `true` als auch `false` angenommen hat.
- Bei der *Mehrfachbedingungsüberdeckung* wird gefordert, dass alle Kombinationen der Wahrheitswerte der atomaren Teilbedingungen im Test geprüft werden. Da die Zahl der Kombination mit der Anzahl der atomaren Teilbedingungen exponentiell wächst, ist dies sehr aufwändig.
- Bei der *minimalen Mehrfachbedingungsüberdeckung* müssen nur noch die Kombinationen betrachtet werden, bei denen die Änderung des Wahrheitswertes einer atomaren Bedingung den Wahrheitswert der Gesamtbedingung ändert.

Die Methode der (minimalen) Mehrfachbedingungsüberdeckung erzeugt auch alle Testfälle die durch Anweisungs- und Zweigüberdeckung generiert werden. Daher müssen diese Verfahren nicht mehr zusätzlich durchgeführt werden. Ein Nachteil dieser Methode besteht darin, dass nur Bedingungen innerhalb von `if` Anweisungen überprüft werden. Programmcode der Form

```
a = x || y;  
if (a) then ... else ...
```

würde den Test unterlaufen.

Pfadüberdeckung

Die *Pfadüberdeckung* fordert die Ausführung aller möglichen Pfade durch ein Testobjekt (beziehungsweise durch den zugehörigen Kontrollflussgraphen). Dies ist im Allgemeinen kaum möglich, da zum Beispiel auch jede mögliche Anzahl von Schleifenwiederholungen zu einem eigenem Pfad führt.

Linear Code Sequence and Jump

Diese Verfahren eignet sich für Programme die Sprünge enthalten. Der Code wird in Sequenzen von Anweisungen, die mit einem Sprung enden, unterteilt. Kombinationen dieser Abschnitte werden beim Test berücksichtigt.

⁴Bedingungen, die nur Relationssymbole (`>`, `≥`, `=`, `≤`, `<`, `≠`) enthält

3 Spezielle Testverfahren im Automotive

Die Entwicklung von Software für Steuergeräten im Automobilbereich durchläuft mehrere Phasen:

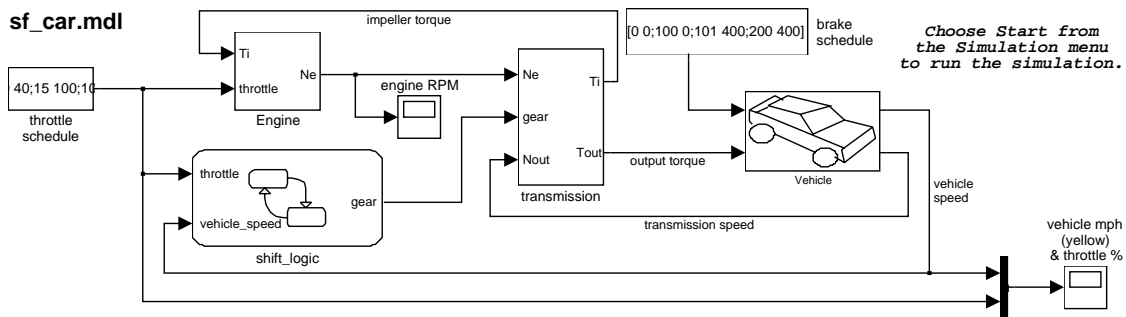
1. Umsetzung der Spezifikation in ein Modell mit Hilfe von graphischen Programmiersprachen wie zum Beispiel Matlab/Simulink (The MathWorks) oder ASCET-SD (ETAS). Man spricht dann von der sogenannten *ausführbaren Spezifikation*.
2. Erzeugung von Code
3. Einbringen des Codes auf ein Testboard. Hier findet der Übergang von der Entwicklungsplattform auf die Anwendungsplattform statt.
4. Einbringen des Codes auf die tatsächliche Steuergeräthardware.

Jedes dieser Stadien stellt unterschiedliche Anforderungen an das Testen: Während man in den ersten beiden Entwicklungsstufen noch Einblick auf den Code der Testobjekte hat (→ Whitebox-Test), ist das bei den letzten Beiden nicht mehr der Fall (→ Blackbox-Test). Weiterhin spielt es eine Rolle ob die Umwelt (also das in der Entwicklung befindliche Fahrzeug) bereits zum Test verwendet werden kann oder simuliert werden muss. Die im Weiteren vorgestellten Testverfahren sind in nachfolgender Abbildung dargestellt.

		Umwelt	
		Simuliert	Real
S y s t e m	Modell	Model in the loop	Rapid Prototyping
	Code	Software in the loop	
	Testboard	Processor in the loop	
	Steuergerät	Hardware in the loop	

3.1 Model-in-the-loop

Model-in-the-loop stellt den ersten Vertreter der in-the-loop-Tests. Für dieses Verfahren (sowie für alle folgenden in-the-loop Verfahren) wird ein Modell der Umgebung des zu Entwickelnden Steuergeräts benötigt. Das Umgebungsmodell darf einen hohen Abstraktionsgrad besitzen, da zum Beispiel Sensoren und Aktoren nicht aufwändig modelliert werden müssen – deren Ein- und Ausgabeverhalten kann direkt implementiert werden.



Simulink-Modell eines Fahrzeugs

Der Test findet ohne den Einsatz von spezieller Hardware auf einem oder verteilt auf mehreren Rechnern statt. Typischerweise werden hier Zustandstests und Pfadüberdeckungstests durchgeführt. In dieser Entwicklungsphase ist es ein leichtes verschiedene Lösungsmethoden gegeneinander abzuwägen. Auch lassen sich schnell Fehler in der Spezifikation erkennen, da diese oftmals bereits aus der Implementierung herausgelesen werden können (\rightarrow Reviews). Ist die Model-in-the-loop Phase abgeschlossen, so liegt mit dem verifizierten Modell ein elektronisch ausführbares Lastenheft vor.

3.2 Software-in-the-loop

Das aus der Model-in-the-loop Phase hervorgegangene Modell kann nun in Code für das Steuergerät umgewandelt werden. Dies geschieht oftmals automatisiert und wird häufig bereits durch die Entwicklungsumgebung der graphischen Programmiersprache ermöglicht. Meist ist die Zielsprache C. Das hier verwendete Umgebungsmodell kann mit dem vorhergehenden weitgehend identisch sein. Wurde der Code nicht automatisch aus dem Modell erzeugt kann das Modell zur Testfallgenerierung herangezogen werden.

Da noch keine Hardwarekomponenten am Test beteiligt sind, ist nicht nötig Echtzeittests durchzuführen. Weiterhin ist der Programmcode vollständig einsehbar was Whitebox-Tests ermöglicht. Aus diesen beiden Gründen eignet sich die Software-in-the-loop Phase sehr gut für ausführliche Testreihen, vor allem mit intensiven Überdeckungstests, da tiefgehende Tests bei geringem Zeitaufwand durchführbar sind. Daneben können in diesem Stadium auch Testmuster für die Überprüfung der weiteren Entwurfsschritte zusammengestellt werden.

Software-in-the-loop Tests können den Entwicklungsprozess erheblich beschleunigen, da die Software getestet werden kann ohne auf die zu diesem Zeitpunkt unter Umständen noch nicht verfügbare Zielhardware angewiesen zu sein.

3.3 Processor-in-the-loop

Die *Processor-in-the-loop* Phase stellt einen möglichen Zwischenschritt dar, bei dem die Steuersoftware zum ersten Mal auf der Zielhardware eingesetzt wird. Dazu wird jedoch nicht das komplette Steuergerät, sondern lediglich der Prozessor, auf dem die Software später ausgeführt werden soll benutzt. Dieser befindet sich dabei auf einem Testboard (auch *evaluation board* genannt), das mit dem Simulationsrechner verbunden ist. Ab dieser Phase sind nur noch → Blackbox-Tests möglich, da offensichtlich kein Einblick mehr in die inneren Abläufe des Systems vorhanden ist.

3.4 Hardware-in-the-loop

Im *Hardware-in-the-loop* Stadium wird nun die Software auf das tatsächliche Steuergerät übertragen und dort mit Hilfe eines Umweltmodells zur Ausführung gebracht. Das Steuergerät ist wie im realen Fahrzeug bedienbar, und die Umweltsimulation ermöglicht je nach Komplexität das Nachbilden von realen Situationen. Hier ist natürlich ein vergleichsweise komplexes Modell vonnöten, da die Software nicht beliebig mit Daten versorgt werden kann, sondern tatsächlich das Steuergerät selbst angesteuert werden muss. Eine weitere Forderung ist dass die Umweltsimulation um den Faktor zehn schneller als das zu testende Steuergerät ist (Echtzeitfähigkeit).

Hardware-in-the-loop ermöglicht über den bisherigen Horizont der Steuergeräte hinaus zum Beispiel „virtuelle Probefahrten“. Hier ist jedoch der Simulations- und Testsystemaufbau wesentlich komplexer, aber im Vergleich zum Aufwand realer Testfahrten immer noch geringer. Ein weiterer Vorteil ist die Möglichkeit unabhängig vom der tatsächlichen Umwelt (Wetter) und gefahrlos das Verhalten in jeglicher Situation nachzubilden. Vor allem bei der Entwicklung kostspieliger Systeme können so Tests durchgeführt werden, ohne das Risiko der Beschädigung oder gar des Verlusts des Prototypen. Darüber hinaus macht Hardware-in-the-loop auch Tests möglich, die in der Realität nur schwer oder sogar unmöglich durchführbar wären (Raumfahrt).

Die Durchführung von Tests kann bei Einsatz von Hardware-in-the-loop automatisiert werden, was bei normalen Testserien nicht oder nur schwer möglich wäre. Ausserdem sind die Ergebnisse der Tests reproduzierbar und Tests können schnell in abgewandelter Form, zum Beispiel nach Austausch des Getriebes, durchgeführt werden. Da kein Einblick auf das Innere des Steuergeräts gegeben ist werden vor allem Zustandstests durchgeführt, auch Smoke-Tests von neuen Versionen einer Komponente bieten sich an.

Grundlegende Informationen zum Thema x-in-the-Loop wurden [5, 2.4.3, 2.4.4] entnommen.

3.5 Rapid Prototyping

Unter *Rapid Prototyping* versteht man Verfahren, die es ermöglichen anhand der Spezifikation schnell einen realen Prototypen zu entwickeln. Dafür sind sowohl die Simulation des Prototypen, als auch die Möglichkeit Änderungen schnell vornehmen zu können wichtig.

Wie bei Model-in-the-loop liegt das zu entwickelnde System auch hier als Modell vor, Hardware wird nicht benötigt. Ein Unterschied ist jedoch, dass das Modell in Verbindung mit der realen Umwelt getestet wird – ist diese nicht verfügbar wird alternativ auf einen Hardware-in-the-loop-Simulator zurückgegriffen. Rapid Prototyping Systeme müssen daher in der Lage sein, alle in der Umgebung vorhandenen Sensordaten zu verarbeiten und Aktuatoren anzusteuern.

Man unterscheidet drei Ausprägungen des Rapid Prototyping.

- *Konzeptorientiertes* Rapid Prototyping: Das Ziel ist die Generierung eines Prototyps, der nicht von der späteren Realisierung abhängt. Dazu wird ein Modell des Systems mithilfe graphischer Programmiersprachen erzeugt. Aus dem Modell wird automatisiert Code generiert und auf einem Rechner, der mit der realen Umwelt verbunden ist, getestet. Um eine hohe Geschwindigkeit des Verfahrens zu garantieren wird ausschließlich automatisch generierter Code verwendet.
- *Architekturorientiertes* Rapid Prototyping: Nun wird eine Zielhardware ausgewählt, auf diese das System angepasst werden muss. Weiterhin wird das System in Funktionseinheiten unterteilt (Spannungsversorgung, Prozessor, E/A) die möglicherweise bereits vorhanden sind (da zum Beispiel Standardkomponenten eingesetzt werden) oder sich aber noch in der Entwicklung befinden. Mit dem resultierenden Prototyp kann ein System getestet werden, dessen Komponenten sich in unterschiedlichen Stufen der Entwicklung befinden. Er ist dem Endprodukt schon sehr ähnlich, da die Hardware im wesentlichen übereinstimmt und ermöglicht daher bereits eine Bewertung in Bezug auf die Realisierbarkeit mit der gewählten Zielhardware.
- *Realisierungsorientiertes* Rapid Prototyping: Hier werden spezialisierte Prototypen erzeugt, wobei oftmals sogar handgeschriebener Code einfließt. Sie werden in kleinen Vorserienproduktionen hergestellt oder bestehen aus existierenden Komponenten die mit einer neuen Komponente zu einem System integriert werden müssen.

Grundlegende Informationen zum Rapid Prototyping wurden [7, 2.5.4] entnommen.

A Regeln für eine Reviewsitzung

[6, S.76]

1. Die Sitzung ist auf zwei Stunden beschränkt. Falls nötig wird eine weitere Sitzung frühestens am nächsten Tag einberufen.
2. Der Moderator hat das Recht eine Sitzung abzusagen oder abubrechen, wenn
 - einer oder mehr Gutachter nicht erschienen oder ungenügend vorbereitet sind
 - er aus irgendeinem Grund nicht im Stande ist, die Sitzung erfolgreich und effizient zu leiten
3. Das Prüfobjekt und nicht der Autor stehen zur Diskussion
 - die Gutachter müssen auf ihre Ausdrucksweise achten
 - der Autor darf weder sich noch das Prüfobjekt verteidigen müssen
4. Der Moderator darf nicht als Gutachter fungieren
5. Allgemeine Stilfragen dürfen nicht diskutiert werden
6. Die Entwicklung und Diskussion von Lösungen ist nicht Aufgabe des Reviewteams
7. Jeder Gutachter muss Gelegenheit haben, seine Befunde angemessen zu präsentieren zu können
8. Der Konsens der Gutachter zu einem Befund ist laufend zu protokollieren
9. Befunde sind nicht Form von Anweisungen an den Autor zu protokollieren
10. Die einzelnen Befunde sind zu gewichten als
 - *kritischer Fehler*: Prüfobjekt ist für den vorgesehenen Zweck unbrauchbar
 - *Hauptfehler*: Nutzbarkeit des Prüfobjekts ist beeinträchtigt
 - *Nebenfehler*: Beeinträchtigt die Nutzung kaum
 - *gut*: fehlerfrei
11. Vom Reviewteam ist eine Empfehlung über die Annahme des Prüfobjekts abzugeben:
 - akzeptieren ohne Änderungen
 - akzeptieren mit Änderungen ohne weiteres Review
 - nicht akzeptieren
12. Am Schluß haben alle Teilnehmer das Protokoll zu unterschreiben

B Bildung eines Übergangsbaums

[6, S.118]

1. Der Anfangszustand ist die Wurzel des Baumes
2. Für jeden möglichen Übergang vom Anfangszustand in einen Folgezustand im Zustandsdiagramm erhält der Übergangsbaum von der Wurzel aus eine Verzweigung zu einem Knoten, der den Nachfolgezustand repräsentiert.
3. Der letzte Schritt wird für jedes Blatt des Übergangsbaumes wiederholt, bis eine dieser Bedingungen eintritt:
 - Der dem Blatt entsprechende Zustand ist bereits im Pfad von der Wurzel zum Blatt enthalten (Durchlauf eines Zykluses).
 - Der dem Blatt entsprechende Zustand ist ein Endzustand.

Literatur

- [1] <http://www.auto-motor-und-sport.de>
- [2] <http://futurezone.orf.at>
- [3] <http://www.heise.de>
- [4] Spiegel, Ausgabe 12/1999
- [5] Nico Hartmann: Automation des Tests eingebetteter Systeme am Beispiel der Kraftfahrzeugelektronik. Dissertation. Karlsruhe: Universität Fredericiana, 2001
- [6] Andreas Spillner, Tilo Linz: Basiswissen Softwaretest. 2.Auflage, Heidelberg: **dpunkt.verlag**, 2004
- [7] Bernhard Spitzer: Modellbasierter Hardware-in-the-Loop Test von eingebetteten elektronischen Systemen. Dissertation. Karlsruhe: Universität Fredericiana, 2001