Seminar „Pleiten, Pech und Pannen der Informatik“

Simulation und Testbenchmethodik

Juli 2002

1 Das Quebec Bridge Disaster, 29. August 1907

1887 wurde das Quebec Bridge Project gegründet. Geplant war eine Auslegerbrücke über den St.Lorenz-Strom in der Nähe von Quebec City, Kanada. Es wurden Pläne ausgearbeitet und gezeichnet, doch weiter geschah die nächsten 13 Jahre nichts.

Im Jahr 1900 übernahm dann ein Amerikaner namens Theodore Cooper die Leitung des Projekts. Er genoss einen sehr guten Ruf als Ingenieur, war er doch schon am Bau mehrerer grosser und prestigeträchtiger Brücken in den USA beteiligt. Seine erste Massnahme war die Modifizierung der vorhandenen Baupläne. So vergrösserte er die Spannweite der Brücke von 1600 auf 1800 Fuss. Damit hoffte er, Kosten zu sparen, da die Stützpfeiler der Brücke nun nicht mehr im Wasser des Stroms, sondern auf Land gebaut werden konnten. Ausserdem machte diese Verlängerung die Quebec Bridge zur grössten Auslegerbrücke der Welt. Aufgrund des guten Rufes, den Cooper genoss, und aufgrund der Tatsache, dass das Geld sehr knapp war, wurden diese Veränderungen übernommen, ohne dass man erneute Tests und Berechnungen durchführte. Cooper sagte, dass die Brücke das Mehr an Gewicht ver-tragen würde, und so schob man sämtliche Zweifel erst einmal beiseite.

Als das Geld für den Brückenbau endlich beisammen war, verlangte die kanadische Regierung eine neue, unabhängige Überprüfung der Baupläne. Doch Cooper, ausser sich vor Wut über die Zweifel seitens der Regierung, sagte, man hätte bereits genug Zeit verloren. 1903 wurde mit

Bereits im Jahr 1906 wurde klar, dass die Brücke sehr viel schwerer werden würde als geplant. Man hatte bereits das für die ganze Brücke veranschlagte Gewicht erreicht, obwohl man noch lange nicht fertig war. Doch die einzige Alternative wäre zu diesem Zeitpunkt gewesen, die Brücke abzureissen und von vorne zu beginnen. Da jedoch schon sehr viel Zeit und Geld in den Bau geflossen waren, wurden die Bauarbeiten fortgesetzt.

1907 wurden dann erste Verbiegungen an den Stahlträgern gemeldet. Anfangs ignorierte Cooper diese Meldungen, aber am 27. August sendete er doch ein Telegramm, man solle die Arbeiten einstellen. Die ausführende Baufirma vermutete allerdings, dass die Träger schon mit diesen Verbiegungen aus der Fabrik kamen, oder dass diese von der Installation herrühren. Man machte sich weiter keine Gedanken. Der nächste Tag verging, ohne dass es Probleme gegeben hätte.

Am Morgen des 29. August beauftragte Cooper einen Assistenten, zur Baustelle zu fahren, und sich mit den Leitern der Baufirma zu treffen. Dort angekommen, traf er sich mit diesen zu einer Unterredung, und man beschloss, eine Entscheidung über den Fortgang der Arbeiten auf den nächsten Morgen zu vertagen, da in diesem Moment sowieso schon die Feierabendsirene ertönte. Noch vor dem zweiten Sirenenlaut, der den Feierabend für die Arbeiter auf der Brücke bedeutet hätte, brach die Brücke unter ihrem eigenen Gewicht zusammen. Von den 84 Arbeitern, die sich zu diesem Zeitpunkt noch auf der Brücke befanden, überlebten 11 das Unglück.

Die Parallelen zur heutigen Situation der Mikrochip-Fertigung sind unübersehbar. Jedes neue Design ist eine neue Herausforderung. Nur weil man vielleicht schon Erfahrung hat, darf man bei einem neuen Design nicht auf unabhängige Überprüfung verzichten. Fehler werden gemacht, auch von renommierten und bekannten Designern. Deswegen nimmt Verifikation im Hardware-Designprozess eine sehr grosse Rolle ein.

2 Hardware-Design

2.1 Hardware Description Languages

Früher wurde Hardware am Bildschirm grafisch editiert, Gatter für Gatter wurde auf einem Schalt-plan aneinandergereiht, entsprechend der Spezifikation. Anschliessend wurde dieser Entwurf in reale Hardware umgesetzt. Dann setzte man an die Eingänge verschiedene Ströme an und mass die Werte an den Ausgängen. Wurden Fehler entdeckt, begann das ganze wieder von vorne: verbesserter grafischer Schaltplan, Umsetzung, Messen. Das macht eine Fehlerkorrektur natürlich sehr aufwendig und dementsprechend hoch war natürlich auch die Entwicklungszeit.

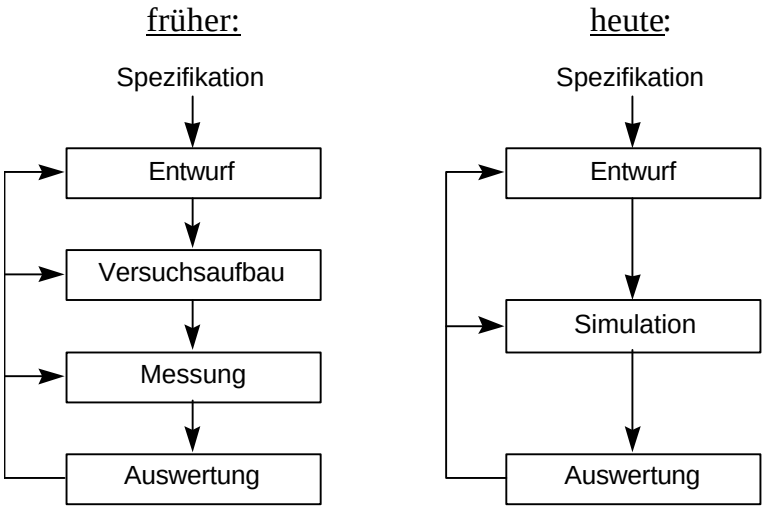

Abbildung 1: Hardware-Design, früher und heute

Bis heute sind die Transistorzahlen pro IC rasant angewachsen. Die Designs wurden grösser und komplexer und dementsprechend sind heutzutage auch grössere Arbeitsgruppen an einem Design beschäftigt. Aus diesem Grund gewannen auch Simulation und Dokumentation immer mehr an Bedeu-tung. Deshalb wird Hardware heute nicht mehr grafisch editiert, sondern mittels Hardwarebeschrei-bungssprachen (*Hardware Description Language*, *HDL*) zunächst textuell umgesetzt. HDLs sind abstrakte Programmiersprachen, die einige Vorteile mit sich bringen. Erstens arbeitet man auf einem sehr hohen, abstrakten Niveau, und braucht sich nicht um einzelne Gatter zu kümmen, wenn man es nicht wünscht. Das Abstraktionsniveau ist aber ausserdem variabel, man kann also durchaus die funktionelle Ebene verlassen und in die Details des Designs eingreifen. Desweiteren ist Code, der mittels einer HDL erstellt wurde, wiedervendbar, da man nicht maschinenorientiert arbeitet, sondern problemorientiert. Beispiele für solche HDLs wären VHDL, Verilog oder SystemC.

2.2 Verifikation

Wie wichtig die Verifikation eines Designs heutzutage ist, zeigen folgende Zahlen: Infineon veranschlagt ca. 70% des Hardware-Designprozesses für Verifikation, bei Synopsys spricht man von ca. 75% und bei Cisco Systems von ca. 80%. Dabei unterscheidet man die formale und die funktionale Verifikation. Während die formale Verifikation mit Hilfe von Kalkülen die korrekte Umsetzung einer Spezifikation mathematisch vollständig beweist, wird in der funktionalen die Funktionalität des Designs mittels Simulation überprüft. Dies ist aber kein Beweis für die korrekte Umsetzung der Spezifikation. Ob Spezifikation und Umsetzung übereinstimmen, hat der Designer hier selbst zu entscheiden.

3 Simulation

3.1 Simulatoren

Simulatoren sind die verbreitetsten und bekanntesten Verifikationswerkzeuge. Sie ermöglichen die Fehlererkennung und -beseitigung vor der eventuellen Fertigung von Prototypen und helfen so Zeit und Kosten zu sparen. Simulatoren sind der Versuch einer Annäherung an die Realität und wollen das Design, bzw. das Modell, d.h. die Umsetzung des Designs in ausführbaren Code einer HDL, in eine künstliche Umwelt einbetten. Dabei werden einige physische Gegebenheiten vernachlässigt bzw. vereinfacht. Während in der Realität beispielsweise ein ständiger Stromfluss herrscht, der unendlich viele Werte annehmen kann, wird diese Zahl in einem digitalen Simulator auf

die Werte ‘0’, ‘1’, ‘undefiniert’ und ‘high impedance’ beschränkt. Die Realitätsnähe variiert auch je nach benutztem Abstraktionslevel. Diese künstliche Umwelt, in die das Design vom Simulator eingebettet wird, nennt man *Testbench*. Eine Testbench ist ebenfalls ein Stück Code, das das Design mit Input-Werten versorgt und die von dem Design produzierten Outputs errechnet. Diese werden dann entweder gespeichert oder verarbeitet oder mit Referenzwerten verglichen. Somit sind Simulatoren keine statischen Tools. Wichtig ist, dass Abwesenheit von Fehlern mit Simulatoren nicht gezeigt werden kann, lediglich deren Anwesenheit.

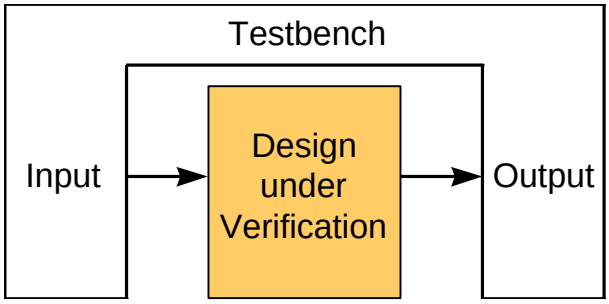

Abbildung 2: Testbench

Es gibt verschieden Arten von Simulatoren, die wichtigsten sind die analogen und die digitalen Simulatoren. Analoge Simulatoren arbeiten zeit- und wertekontinuierlich, und sind aber eher für kleinere Schaltkreise geeignet. Ein Beispiel für einen analogen Simulator wäre das Programm *spice*. Bei digitalen Simulatoren gehen hingegen Zeit- und Werteinformationen verloren. Die Simulation findet hier auf der Basis von logischen Werten und Funktionen statt.

3.2 Event-Driven Simulation

Ein grosses Problem ist, dass Simulatoren immer langsamer arbeiten als die reale Welt. Elektrizität bewegt sich in der Realität nahezu mit Lichtgeschwindigkeit, und Transistoren schalten über eine Milliarde Mal in der Sekunde. Ein Computer aber kann nur mehrer Millionen Instruktionen in der Sekunde abarbeiten.

Performanzverbesserungen sind aber möglich, wenn man sich einmal genauere Gedanken darüber macht, wann ein Modell überhaupt ausgeführt werden muss. Man betrachte beispielsweise einmal folgende Umsetzung eines XOR-Gatters in HDL-Code:

```
XOR_Gate: process (A, B)
begin
   if A = B then
      O <= ‘0’;
   else
      O <= ‘1’;
   end if;
end process XOR_GATE;
```

Obwohl in der Realität ständig Strom an diesem Gatter anliegt, ist dies in einem Simulator nicht notwendig. Die Outputs verändern sich lediglich für eine Veränderung der Inputs, und das auch

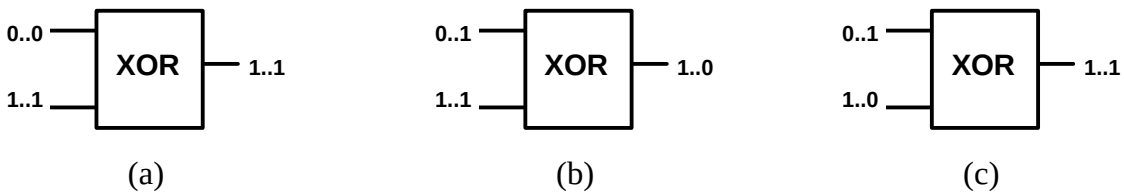

Abbildung 3: XOR-Gatter mit verschiedenen Input-Wechseln

Wenn man nun das Modell bei konstanten Eingängen nicht ausführt, sondern lediglich bei einer Veränderung eines Eingangswertes, so spricht man von *Event-Driven Simulation*.

Eine Ausführung des Modells geschieht auch im Fall (c), wo eine Veränderung der Inputs scheinbar keine Veränderung des Outputs bewirkt. Aber erstens weiss der Simulator nicht, was für ein Gatter er da ausführt, sondern nur, dass er ein Gatter ausführt. Im Fall dieses XOR-Gatters muss das Modell vielleicht nicht ausgeführt werden, aber wenn es sich um eine andere Schaltung handelt, dann eventuell schon. Und zweitens ist diese Veränderung des Outputs nur scheinbar nicht vorhanden. In der Realität aber kann z.B. ein Flackern auftreten, wenn die beiden Eingänge nicht perfekt synchron geschaltet sind. Dieses Flackern kann man bei Bedarf auch simulieren, indem man in das Modell Delay-Zeiten einbaut.

3.3 Cycle-Based Simulation

Weitere Verbesserungen der Event-Driven Simulation sind möglich. Man betrachte einmal folgendes Design und seine Timinganalyse:

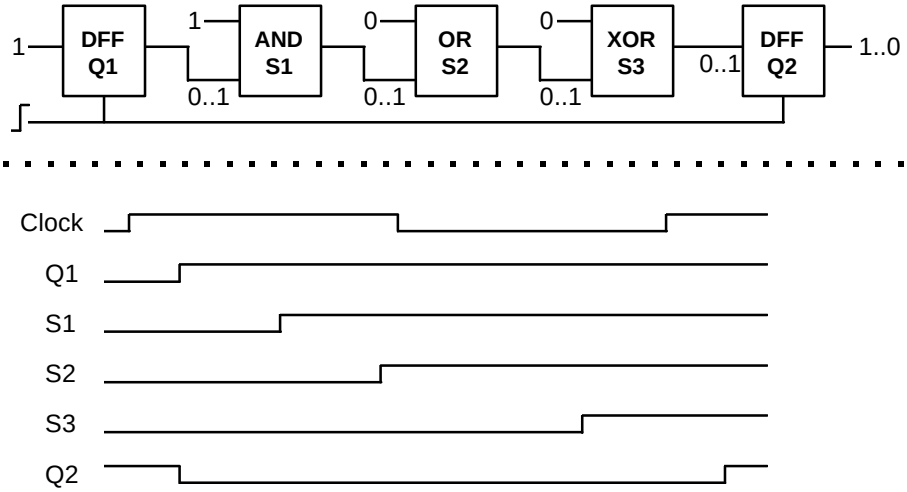

Abbildung 4: Flip-Flop-Schaltung mit Timinganalyse

In einer normalen Event-Driven Simulation hat man hier 7 Modelle (wenn man das Innenleben der Flip-Flops mitzählt) und 6 Events (zweimal steigende Clock-Flanke, Änderung von Signal Q1, Signal S1, Signal S2 und Signal S3). Änderungen des Outputs geschehen hier aber nur bei steigender Clock-Flanke, nicht zwischendrin. Wenn man jetzt also die Schaltung immer nur an einer steigenden Clock-Flanke simuliert, hat man eine *Cycle-Based Simulation*. In der Praxis sieht das so aus, das diese Schaltung beim Kompilieren in einen einzigen Ausdruck verwandelt wird, mit einem Eingang und einem Ausgang.

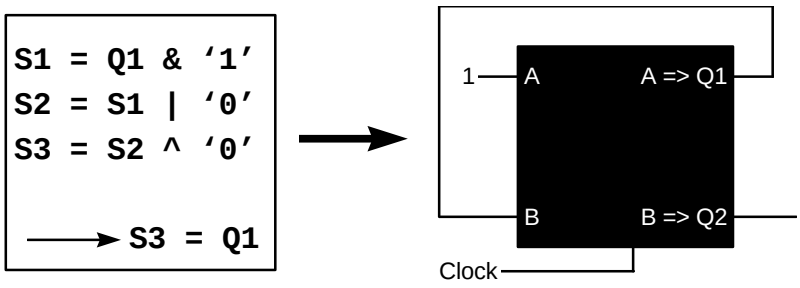

Abbildung 5: Umwandlung der Schaltung aus Abb. 4

Damit müssen dann nur noch ein Modell und zwei Events, nämlich die zwei steigenden Clock-Flanken berücksichtigt werden. Da nur Clock-Flanken signifikant sind, gehen natürlich sämtliche Timing-Informationen verloren, und es können auch nur pefekt synchrone Designs simuliert werden.

3.4 Co-Simulatoren

Um nun die Vorteile einer Cycle-Based Simulation auch in teilweise asynchronen Designs nutzen zu können, gibt es die Möglichkeit einer *Co-Simulation*. Dabei werden die synchronen Teile von einer Cycle-Based Simulation simuliert, und der Rest wie bisher mittels einer Event-Driven Simulation. Umgesetzt wird dies, indem beide Simulationen gleichzeitig arbeiten, und zwar in einem *lock-step* Verfahren. Das bedeutet, dass die schnellere der beiden Simulationen immer an bestimmten Punkten auf die andere Simulation wartet. Dadurch ist gewährleistet, dass die Ergebnisse abgeglichen und ausgetauscht werden können. Es ist offensichtlich, dass die Gesamtgeschwindigkeit durch den langsamsten Simulator bestimmt wird. Ausserdem wird die Leistung gemindert, durch zusätzliche Kommunikations- und Synchronisationskosten zwischen den Simulatoren.

Eine Verbesserungsmöglichkeit bietet eine sogenannte *time-warp synchronization*, in der der schnellere Simulator schon weit vorraus rechnet, unter Annahme bestimmter Ergebnisse des anderen Simulators. Mithilfe von Checkpoints merkt er sich aber vergangene Zustände, zu denen er zurückkehren kann, falls eine seiner getroffenen Annahmen nicht zutrifft.

Denkbar sind auch VHDL/Verilog Co-Simulatoren oder analog/digital Co-Simulatoren. Probleme können aber z.B. die Übersetzung von VHDL-Werten in Verilog-Werte - oder andersrum - bereiten. Während Verilog 128 logische Zustände (in Kombination mit verschiedenen Stärken) kennt, sind es bei VHDL nur 9.

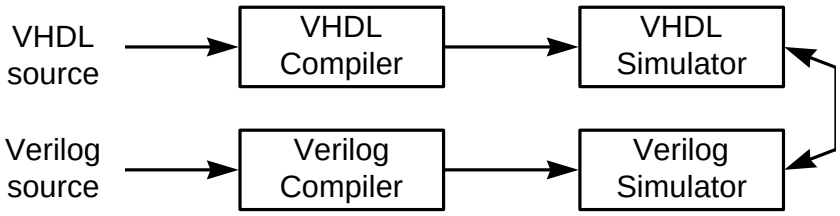

Abbildung 6: Co-Simulation

4 Testbenches

Testbenches sind üblicherweise in der gleichen Sprache geschrieben, wie das Modell, also typischerweise in VHDL oder Verilog. Es gibt auch spezielle Verifikationssprachen zum Erstellen von Testbenches, wie „VERA“ von Synopsys oder „e“ von Verisity. Eine mögliche Alternative wären auch Testbenches in herkömmlichen höheren Programmiersprachen wie C oder C++. Diese Sprachen haben allerdings keine explizite Kenntnis von Hardware, was die Erstellung von Testbenches hiermit etwas erschwert. So ist z.B. die Realisierung von Parallelität oder einer Clock ziemlich aufwendig.

Wann ist aber nun die Verifikation abgeschlossen? Und wie werden die Inputs für eine

4.1 Metriken

Um nun den Fortgang der Verifikation zu erkennen, werden *Metriken* eingesetzt. Das sind Abdeckungsmasse, die z.B. feststellen, ob der Code eines Modells schon komplett abgearbeitet wurde. Angestrebt sind natürlich 100% Abdeckung, doch ist diese in den seltensten Fällen zu erreichen. Stellvertretend für andere Metriken wird hier *Code Coverage* vorgestellt.

4.1.1 Code Coverage

Der Source Code des Modells wird hierfür mit Checkpoints versehen. Dieses neue Modell heisst nun *instrumented model* und wird mit allen zur Verfügung stehenden Testbenches simuliert. Eine angeschlossene Datenbank sammelt die Spuren aller Simulationen durch den Code. Dadurch lässt sich feststellen, wieviel Code schon abgearbeitet wurde, und welches Stück Code noch nicht erreicht wurde.

Verschieden Arten von Code Coverage Metriken existieren, unter anderem Statement Coverage, Path Coverage und Expression Coverage. *Statement Coverage* schaut nur, welche Zeilen Code abgearbeitet wurden. Würde man mehrere Statements, wie beispielsweise den Inhalt einer WHILE-Schleife, zusammenfassend als Statement-Block betrachten, so spricht man von *Block Coverage*. Allerdings können Codezeilen ja auf verschiedene Weisen erreicht werden.

```
if (parity == ODD || parity == EVEN) begin
   tx <= compute_parity(data, parity);
   #(tx_time);
end
tx <= 1'b0;
#(tx_time);
if (stop_bits == 2) begin
   tx <= 1'b0;
   #(tx_time);
end
```

In obigem Beispiel wäre es zum Beispiel denkbar, dass einmal die obere if-Anweisung ausgeführt wurde, und die untere nicht, und in einer zweiten Simulation genau andersrum. Damit hat man zwar 100% Statement Coverage, denn alle Zeilen wurden ja abgearbeitet, aber alle Pfade durch den Code hat man noch nicht berücksichtigt. Es könnten ja auch beide if-Anweisungen zusammen ausgeführt, bzw, nicht ausgeführt werden. Diese Pfade durch den Code misst *Path Coverage*.

Die obere if-Anweisung kann aufgrund von zwei verschiedenen Bedingungen ausgeführt werden, was die ganze Sache noch weiter verkompliziert. Mit *Expression Coverage* wird diese Möglichkeit auch noch berücksichtigt.

Es ist sehr schwer, 100% Path Coverage zu erreichen, und 100% Expression Coverage sind extrem schwer. In beiden Fällen ist eigentlich für grosse Designs 100% Abdeckung unmöglich, will man nicht zu viel Zeit mit der Verifikation verbringen.

Eine Möglichkeit ist auch, bestimmte falsche Teile absichtlich einzubauen, also Code, der eigentlich gar nicht erreicht werden kann. Wird er dann doch erreicht, so weiss man, dass ein Fehler vorliegt. Dieser Code darf aber nicht in die Metrik mit einfliessen und muss besonders gekennzeichnet werden. Diese Technik des Programmierens nennt man *defensive coding*.

Code Coverage Tools können auch zur Performanz-Optimierung herangezogen werden. Man kann z.B. messen lassen, welche Zeilen, bzw. Blöcke, besonders oft abgearbeitet wurden. An diesen Stellen sollte dann die Optimierung ansetzen. Diese Technik heisst *profiling*.

4.1.2 Was bedeuten 100% Abdeckung?

Code Coverage zeigt lediglich an, wie gründlich der Code abgearbeitet wurde. Aussagen über die Korrektheit können nicht getroffen werden. Er sollte auch nur ein _____ Fortschritt der Verifikation sein. Zwar bedeutet eine niedrige Abdeckung, dass die Verifikation längst nicht abgeschlossen ist, aber der Umkehrschluss, dass 100% Abdeckung das Ende der Verifikationsarbeit bedeuten, ist falsch! Zusätzlich sollte man noch weitere Metriken zu Rate ziehen. Man kann dann irgendwann sagen, man habe ausreichend getestet, aber abgeschlossen im eigentlichen Sinn ist die Verifikation trotzdem nicht.

4.1.3 Weitere Metriken

Weitere Metriken existieren, die in Verbindung mit Code Coverage benutzt werden können.

- *Source Code Changes*: um so weniger, um so stabiler der Code.
- *Number of outstanding issues*: muss noch viel am Code verändert werden?

usw.

4.2 Generieren von Stimuli

Die Inputs, die eine Testbench an ein Modell anlegt, werden auch *Stimuli* genannt. Sie stimulieren das Design, Outputs zu produzieren. Diese Stimuli können von der Testbench fest vorgegeben sein, zufällig erstellt oder abhängig von den Outputs errechnet werden. Normalerweise wird auch nicht nur eine einzige Belegung an das Modell angelegt, sondern ganze *Sequenzen* von Belegungen.

4.2.1 Feste Testmuster (*directed testpatterns*)

Die einfachste Lösung wäre, die Inputs fest in die Testbench einzubauen, entweder in Form von Code oder als externes File, in dem mögliche Inputsequenzen gespeichert sind. Diese werden dann bei Bedarf von der Testbench eingelesen. Diese Möglichkeit der festen Testmuster bietet sich besonders an, wenn man gezielt nach einem Fehler sucht. Die Erstellung dieser Testmuster wird allerdings immer aufwendiger, um so grösser das Design ist. Andererseits ist aber die Überprüfung der Outputs relativ einfach, da man ja bestimmte Ergebnisse erwartet, die man bereits im vorraus errechnen kann.

4.2.2 Zufällige Testmuster (*random testpatterns*)

Eine andere Möglichkeit ist die Belegung der Eingänge mit Wahrscheinlichkeitswerten für ‘0’ und ‘1’, aufgrund derer dann zufällige Eingangswerte oder -sequenzen erstellt werden. Dies ist näher an der Realität und man kann auch mehr Tests durchführen, da der Zeitaufwand nicht so gross ist wie bei festen Testmustern. Aber man kann keine zu erwartenden Werte im vorraus berechnen, sondern hier kann z.B. ein Referenzmodell - falls vorhanden - implementiert werden, dass dann die korrekten Werte gleichzeitig berechnet. Hinterher können dann die Werte verglichen werden.

4.2.3 Automatic/Reactive Testbench Generation

Mit Hilfe von Testbench Generation Tools wie „VERA“ können automatisch Testmuster erstellt werden. Dazu benutzen solche Programme Code Coverage Metriken und schauen, welcher Teil des Codes noch nicht abgearbeitet wurde. Dementsprechend werden dann neue Inputsequenzen erstellt. Die alleinige Konzentration auf Code Coverage ist allerdings in Frage zu stellen, wie bereits weiter oben erwähnt wurde.

Ein interessanter Fall ist, wenn selbst ein teures professionelles Tool bestimmten Code nicht erreichen kann. Dann kann man normalerweise davon ausgehen, dass dieser Code auch nicht erreichbar ist (*toter Code*), und diesen löschen.

4.2.4 Beispiel: Genetic Algorithm

Das automatische Erstellen solcher Inputsequenzen geschieht unter Verwendung eines Algorithmus wie z.B. diesem hier kurz vorgestellten Genetic Algorithms. Die Anbieter der kommerziellen Tools machen allerdings keine expliziten Angaben darüber, wie ihr Algorithmus genau aussieht.

Die Basis des Algorithmus ist die Kodierung von Inputsequenzen in Bitmatrizen von unterschiedlicher Länge. Am Anfang wird wird eine Anzahl solcher Matrizen zufällig generiert, die sogenannte *Anfangs-Population*. Das Ziel des Algorithmus ist die Steigerung des *Fitness-Werts*

dieser Population. Die Fitness-Funktion misst dabei die Nähe einer Sequenz zum angestrebten Zielblock des Codes, den man noch nicht erreicht hat.

In der Anfangsphase versucht man, aus der Population eine Menge von Sequenzen S herauszufiltern, die möglichst viele Statement-Blöcke erreicht (*aktiviert*). Damit ist die Fitnessfunktion:

$$\text{fitness(S)} = \frac{\text{activated\_blocks}(S)}{tot\_blocks}$$

Damit hat man auch gleich die Blöcke erkannt, die sehr einfach zu erreichen sind, und die nicht mehr erreicht werden müssen. In der nächsten Phase visiert der Algorithmus einen Zielblock T an, und versucht, eine Sequenz zu generieren, die diesen Block erreichen kann. Dafür wird die Fitnessfunktion für jede der übriggebliebenen Sequenzen S wie folgt definiert:

Die Korrelation gibt hierbei die Nähe eines Basisblocks zu dem anvisierten Zielblock an, und als Basisblöcke werden solche bezeichnet, die die Sequenz bereits erreicht hat.

Nebenbei werden auch Sequenzen erkannt, die für spätere Zielblöcke wichtig sein können, diese werden dann gespeichert und später in die entsprechende Anfangs-Population eingefügt (aber nicht mehr als 5% der Anfangs-Population dürfen so erzeugt sein).

Sequenzen mit besonders hohem Fitness-Wert werden nun ausgewählt und zur Züchtung einer neuen Generation herangezogen. Dies kann geschehen durch *Mutation*, d.h. diese Matrix wird erweitert oder gekürzt, oder Bits werden vertauscht. Oder durch *Mating*. In diesem Fall werden zwei Matrizen verbunden, um Nachkommen zu erzeugen. Man kann dann von der einen Sequenz den Anfang, von einer anderen das Ende nehmen, oder man kann die Matrizen spaltenweise mischen, oder ähnliches.

Diese Schritte werden solange wiederholt, bis der Zielblock erreicht ist, dann wird der nächste Block anvisiert.

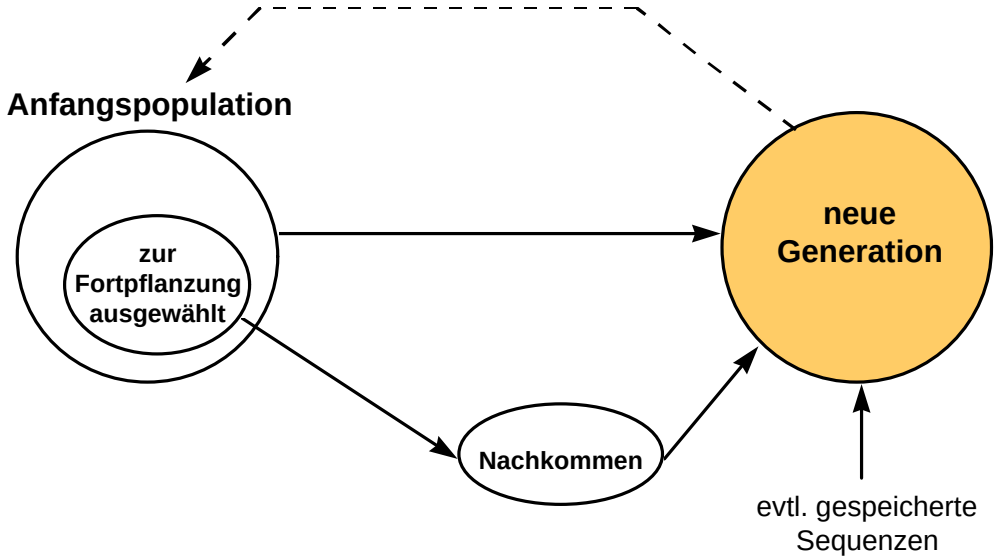

Abbildung 7: der Genetic Algorithm im Überblick

5 Epilog

Die Brücke über den St.Lorenz-Strom wurde später erneut gebaut. Dieses mal mit den entsprechenden Verstärkungen. Ironischerweise kam es 1916, während der Bauarbeiten, zu einem neuerlichen Unglück. Das bereits vorgefertigte Mittelteil sollte gerade an seinen Platz gehievt werden, als es krachend ins Wasser fiel. Dabei wurden 11 Menschen getötet. Dieses Ereignis ist heute als das zweite Quebec Bridge Disaster bekannt.

Endgültig fertiggestellt wurde die Brücke dann 1918. Sie wird heute noch genutzt und ist immer noch die grösste Auslegerbrücke der Welt.

6 Quellen und Literaturhinweise

- „Writing Testbenches“, Janick Bergeron. Kluwer Academic Publishers. 2000.
  [http://www.janick.bergeron.com]
- Folien zur Vorlesung „Hardwareverifikation“, Jürgen Ruf und Thomas Kropf. 2001.
  [http://www-ti.informatik.uni-tuebingen.de/~ruf/vorlesung]
- „Hardwaremodellierung“, Christian Siemers. Hauser. 1999.
- „Automatic Validation of Protocol Interfaces Described in VHDL“, F. Corno, M.S. Reorda, G. Squillero. 2000.
  [http://www.cad.polito.it/pap/db/evotel2000a.pdf]
- „The First Quebec Bridge Disaster - A Case Study“
  [http://www.civeng.carleton.ca/ECL/reports/ECL270/Introduction.html]
- [http://www.synopsys.com/products/vera]
- [http://www.verisity.com]