Implementierung eines hierarchischen Verifikationsverfahrens mit ω-Automaten

Studienarbeit von Ralf-Ulrich Garbe

Betreuer: Klaus Schneider, Prof. Dr.-Ing. D. Schmid

Universität Karlsruhe
Fakultät für Informatik
Institut für Rechnerentwurf und Fehlertoleranz

Karlsruhe, den 20. Februar 1995

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Der Begriff Verifikation . . . 1
1.3 Stand der Technik . . . 2
1.4 Ziel der Studienarbeit . . . 2
1.5 Gliederung der Ausarbeitung . . . 3

2 Grundlagen **5**
2.1 ω-reguläre Sprachen . . . 5
2.2 ω-Automaten . . . 6
2.2.1 Büchi-Automat . . . 7
2.2.2 Rabin-Automat . . . 7
2.3 Automatenformeln . . . 8
2.4 Boolesche Operationen auf deterministischen Automatenformeln . . . 9

3 Hardwareformeln **11**

4 Hierarchische Verifikation **15**

5 Temporallogik und HWFn **19**

6 Resümee der Studienarbeit **23**
6.1 Aufgabenstellung . . . 23
6.2 Die Vermehrung der Zustandsvariablen . . . 24
6.2.1 Disjunktion von HWFn . . . 24
6.2.2 Konjunktion von HWFn . . . 24
6.2.3 Negation von HWFn . . . 24
6.2.4 Implikation und Äquivalenz von HWFn . . . 24
6.2.5 Temporallogische Operatoren . . . 25
6.2.6 Umwandlung von HWFn in Büchi-Formeln . . . 25
6.3 Vermeidung von zusätzlichen Zustandsvariablen . . . 25
6.3.1 SMV und HWFn . . . 26

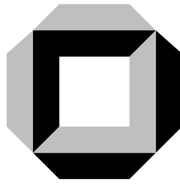
7 Experimentelle Ergebnisse **27**
7.1 Ablauf einer Verifikation 27
7.2 Bewiesene Schaltungen 32

Abbildungsverzeichnis

2.1 Nichtdeterministischer Büchiautomat: $\mathcal{L} = \{0,1\}^* \cdot 1^\infty$ 7
2.2 Deterministischer Rabinautomat 8

3.1 Implementierung des 110-Detektors und dessen formale Beschreibung. . . . 13

4.1 Hierarchische Verifikation 16



Universität Karlsruhe
Institut für Rechnerentwurf und Fehlertoleranz
Prof. Dr.-Ing. D. Schmid
Kaiserstr. 12, Geb. 20.20
76128 Karlsruhe

Erklärung

Hiermit versichere ich, daß ich die vorliegende Studienarbeit selbständig verfaßt und keine anderen Quellen und Hilfsmittel als die im Literaturverzeichnis aufgeführten verwendet habe.

Karlsruhe, den 20. Februar 1995

Kapitel 1

Einleitung

1.1 Motivation

Verifikation ist ein wichtiger Bestandteil beim Entwurf digitaler Schaltungen, aber auch allgemein ein wichtiger Prozeß, der zu der Entstehungsgeschichte eines neuen Produkts gehört. Schließlich soll ein neu entwickeltes Produkt auch das leisten, was es verspricht. Gerade in der Industrie ist es wichtig, daß man Produkte anbieten kann, die sicher sind. Digitale Schaltungen nehmen da eine besondere Rolle ein. Sie finden ihren Einsatz in immer mehr Bereichen des Lebens. Vor allem sicherheitskritische Anwendungen sind es, die dazu zwingen ein Produkt zu verifizieren. Man denke an die Steuerung von Atomkraftwerken oder von Flugzeugen. Hier muß man sich auf die eingesetzten Produkte 100-prozentig verlassen können. Nach neueren Gesetzen kann ein Hersteller sogar für Schäden, die durch ein fehlerhaftes Produkt entstehen, haftbar gemacht werden, wenn ihm nachgewiesen werden kann, daß der Fehler vermeidbar gewesen wäre.

Verifikation darf sich also nicht nur darauf beschränken zu testen, ob eine Schaltung das Gewünschte tut, sie muß es vielmehr beweisen und zwar unumstößlich. Digitale Schaltungen von heute sind so komplex, daß sie weder vom Menschen noch vom Computer als Ganzes behandelt werden können. Deshalb macht man es wie in der Systemtheorie üblich und bricht das komplexe System auf. Dabei setzt sich das Gesamtsystem aus Teilsystemen geringerer Komplexität zusammen. Diese Subsysteme können nun rekursiv weiter unterteilt werden, bis man Systemgrößen erreicht hat, die beherrschbar sind. Bei der Entwicklung von Schaltungen ist dies ein gängiges und sehr erfolgreiches Verfahren. Man denkt in Modulen, Teilmodulen und Grundmodulen. Das gleiche Verfahren soll nun auch in der Verifikation Anwendung finden. Man orientiert sich dann bei der Zerteilung der Schaltung an dem, was die Entwicklung schon geschaffen hat.

1.2 Der Begriff Verifikation

Verifizierung (lat.), der Erweis der Wahrheit von Aussagen. Verifikation beweist also die Richtigkeit von Aussagen, die bezüglich eines Sachverhaltes gemacht werden. Bei

der Hardware-Verifikation werden Aussagen bezüglich einer Schaltung gemacht; z.B. soll eine Ampelsteuerung gewährleisten, daß niemals zwei sich kreuzende Richtungen gleichzeitig grün bekommen. Die Aussage, die gemacht wird, nennt man die Spezifikation der Schaltung. Bei der Schaltung spricht man von der Implementierung und die Schaltungsbeschreibung nennt man die Implementierungsbeschreibung. Verifiziert wird eine Schaltung immer bezüglich einer Spezifikation. Als Ergebnis einer Verifikation erhält man eine der beiden Aussagen:

1. die Schaltung erfüllt die Spezifikation oder
2. die Schaltung ist bezüglich der Spezifikation nicht korrekt.

Damit wurde dann jedoch nur gezeigt, daß die Schaltung eine konkret spezifizierte Anforderung erfüllt. Der Beweis, daß auch andere Anforderungen eingehalten werden, kann dann Schritt für Schritt durchgeführt werden.

1.3 Stand der Technik

Im Großen und Ganzen gibt es zwei unterschiedliche Vorgehensweisen, um Aussagen zu beweisen. Bei der ersten Vorgehensweise werden alle möglichen Zustände, die die Schaltung annehmen kann überprüft, ob in ihnen die Schaltung das gewünschte Verhalten hat. Bei dieser Art der Beweisführung werden in der Regel Model-Checker benutzt, die technisch schon sehr weit entwickelt sind. Vorteil dieser Beweisführung[1] ist die Entscheidbarkeit und damit die Automatisierbarkeit. Der Nachteil der Model-Checker liegt in der Begrenzung der Komplexität, die die Schaltungen haben dürfen, denn die Anzahl der Zustände[2] wächst exponentiell mit der Schaltungsgröße.

Die andere Vorgehensweise stützt sich auf mathematische Systeme, in denen ein Beweis geführt wird. Es werden Axiome und Ableitungsregeln aufgestellt, auf deren Basis der Beweis geführt wird. Dieser elegante Weg hat jedoch den Nachteil, daß er nicht entscheidbar ist und damit auch nicht automatisiert werden kann [Goed31]. Bei dieser Vorgehensweise muß der Mensch den Beweisablauf unterstüzen indem er Vorgaben zum Beweisablauf gibt. Z.B. gibt er an, daß es günstig wäre, als nächstes eine bestimmte Ableitungsregel zu benutzen. Das verlangt von dem Betreuer natürlich eine große Erfahrung.

1.4 Ziel der Studienarbeit

Die Studienarbeit bildet nun einen Mittelweg der beiden Ansätze. Auf der einen Seite sollen die weit entwickelten Model-Checker für den eigentlichen Beweis benutzt werden, auf der anderen Seite sollen auf einer mathematisch fundierten Grundlage aufbauend, Vorleistungen genutzt werden, um auch komplexere Schaltungen verifizieren zu können.

[1] In dieser Arbeit wird ein Model-Checker auch als Beweiser angesehen.
[2] und damit die Laufzeit

1.5 Gliederung der Ausarbeitung

Die Hardware-Verifikation soll Aussagen, die bezüglich einer Schaltung gemacht werden, beweisen. Dafür werden im Rahmen dieser Arbeit Formeln in Logik höherer Ordnung hergeleitet, mit denen man abstrakt eine Spezifikation und gleichzeitig recht anschaulich die Implementierung einer Schaltung beschreiben kann. Bei diesen Formeln handelt es sich um Hardwareformeln [Schn95b].

In Kapitel 2 werden die formalen Grundlagen gegeben. Neben den ω-regulären Sprachen werden der Büchiautomat und der Rabinautomat vorgestellt und es werden Automatenformeln für diese beiden Automatentypen definiert. In Kapitel 3 wird die Klasse der Hardwareformeln (HWFn) und auf diesen boolesche Verknüpfungen definiert. Hardware-Formeln sind spezielle Automaten-Formeln, die auf die Bedürfnisse der Hardware-Verifikation zugeschnitten sind. Es wird dann anhand eines Beispiels gezeigt, wie aus einer Schaltungsbeschreibung eine Hardwareformel hergeleitet werden kann. In Kapitel 4 wird die Vorgehensweise der Hierarchischen Verifikation erläutert. In Kapitel 5 wird gezeigt, wie aus temporallogischen Aussagnen Hardwareformeln hergeleitet werden. Kapitel 6 stellt einige aus der praktischen Arbeit hervorgegangenen Ergebnisse vor. In Kapitel 7 wird Anhand des 110-Detektors aus Kapitel 3 vorgeführt, was die Implementierten Funktionen an Ausgaben liefern.

Kapitel 2

Grundlagen

Es werden zunächst einige grundlegende Definitionen gegeben. Auf diesen aufbauend werden dann die *ω-regulären Sprachen* und Automaten definiert.

Σ	$=$	$\{\sigma_1, ..., \sigma_m\}$: *endliche Menge von Symbolen (Alphabet)*
Σ^*	:	*Menge der endlichen Aneinanderreihungen von Elementen aus* Σ
Σ^∞	:	*Menge der unendlichen Aneinanderreihungen von Elementen aus* Σ
Wort	$\in$	$\Sigma^* \cup \Sigma^\infty$
Sprache	$=$	*Menge von Worten* $\mathcal{L} \subseteq \Sigma^* \cup \Sigma^\infty$
$\alpha \cdot \beta$	:	*Aneinanderreihung der Wörter* $\alpha \in \mathcal{L}_1$ *und* $\beta \in \mathcal{L}_2$ *(auch* $\alpha\beta$*)*
$\mathcal{L}_1 \cdot \mathcal{L}_2$	$:=$	$\{\alpha\beta \mid \alpha \in \mathcal{L}_1 \wedge \beta \in \mathcal{L}_2\}$
$\alpha^{(i)}$	:	*Elemente eines Wortes* $\alpha = \alpha^{(0)}\alpha^{(1)}\alpha^{(2)}... \in \Sigma^\infty$. $\alpha^{(i)} \in \Sigma$

2.1 ω-reguläre Sprachen

Digitale Schaltungen transformieren Eingangssignale in Ausgangssignale. Dabei kann man das Eingangssignal als Eingangswort und das Ausgangssignal aus Ausgangswort ansehen. Wenn nun also digitale Schaltungen formal beschrieben werden sollen, so müssen die Worte (Signale) mit denen die Schaltung arbeiten soll, formalisiert werden. Dazu werden die regulären Mengen eingeführt.

Definition 2.1.1 (reguläre Sprachen) *Sei* Σ *ein endliches Alphabet, dann ist die Menge M der regulären Sprachen induktiv wie folgt definiert.*

i) *jede Teilmenge von* Σ *ist regulär*
ii) M *regulär über* $\Sigma \Rightarrow M^*$ *regulär über* Σ
iii) $M_1,\ M_2$ *regulär über* $\Sigma_1,\ \Sigma_2\ \Rightarrow M_1 \cup M_2$ *und* $M_1 \cdot M_2$ *regulär über* $\Sigma_1 \cup \Sigma_2$

Reguläre Sprachen haben damit die Eigenschaft, daß sie nur endlich lange Worte enthalten. Bei der Verifikation soll aber die Korrektheit der Schaltung nicht nur für eine endliche Zeit, sondern für immer bewiesen werden. Deshalb werden nun die ω-reguläre Sprachen eingeführt.

Definition 2.1.2 (ω-reguläre Sprachen) *Sei Σ ein endliches Alphabet und M_1, M_2 reguläre Sprachen über Σ dann ist $M_1 \cdot M_2^\infty$ ω-regulär.*

Beispiel ω-reguläre Sprachen
sei $\Sigma = \{a, b, c, d, e, f\}$, $M_1 = \{aa, ab, ac\}$, $M_2 = \{cd, de\}$,
dann ist $\{aa \cdot cd^\infty,\ aacd \cdot (de)^\infty,\ aa \cdot (cdde)^\infty,\ abac \cdot (cd)^\infty\}$ eine ω-reguläre Sprache.

Ein Wort einer ω-regulären Sprache hat damit einen endlichen Anfang aus der Sprache M_1 und ein unendliches Ende mit Worten aus M_2. Damit kann man einen unendlich langen Eingabestrom $i = i^0 i^1 i^2 i^3 ...$ und einen unendlich langen Ausgabestrom $o = o^0 o^1 o^2 o^3 ...$ beschreiben. Bei der Verifikation muß nachgewiesen werden, daß die Schaltung die geforderte Funktion $f(t)$ implementiert.

2.2 ω-Automaten

In Kapitel 2.1 wurden die ω-regulären Sprachen eingeführt. Sie dienen zur Beschreibung von unendlich langen Folgen von Elementen eines Alphabets, d.h. sie beschreiben die Worte, die zu einer Sprache gehören. Die in diesem Kapitel beschriebenen Automaten dienen ebenfalls zur Beschreibung einer Sprache. Ein Automat trifft Aussagen darüber, ob ein Wort zu seiner Sprache gehört oder nicht. Gehört das Wort zu der vom Automaten spezifizierten Sprache, so sagt man der Automat akzeptiert das Wort. Dabei wird in der Regel von endlichen Automaten, d.h. von Automaten mit endlich vielen Zuständen, ausgegangen.

Definition 2.2.1 (Automat) *Ein Automat $\mathcal{A}$ besteht aus einem Zustandsübergangssystem $\mathcal{G} = (Q, \Sigma, \delta, q_0)$ und einem Akzeptanzkriterium.*
Dabei ist
Q: die Mengen der Zustände des Automaten
Σ: das Alphabet über dem der Automat arbeitet
δ: Zustandsübergangsrelation $\delta : Q \times \Sigma \times Q$
q_0: der Anfangszustand mit $q_0 \in Q$

Bemerkung: δ ist nur bei deterministischen Automaten eine Funktion. Bei indeterministischen Automaten ist der Folgezustand nicht eindeutig bestimmt und somit keine Funktion, sondern eine Relation.

Definition 2.2.2 (Durchlauf) *Sei $\alpha = \alpha^{(0)}\alpha^{(1)}\alpha^{(2)}... \in \Sigma^\infty$ dann nennt man jedes unendlich lange Wort $\beta = \beta^{(0)}\beta^{(1)}\beta^{(2)}... \in Q^\infty$ einen Durchlauf von α falls*
$\beta^{(0)} = q_0$ und $\forall k \in I\!N. \beta^{(k+1)} \in \delta(\beta^{(k)}, \alpha^{(k)})$

Definition 2.2.3 (RUN, INF) *Die Menge aller möglichen Durchläufe eines Wortes α durch einen Automaten $\mathcal{A}$ bezeichnet man mit RUN($\alpha, \mathcal{A}$).*
INF(β) bezeichnet die Menge der Zustände, welche von einem Durchlauf $\beta \in$ RUN($\alpha, \mathcal{A}$) unendlich oft durchlaufen werden, d.h. INF(β) := $\{q \in Q | \forall t_1. \exists t_2. \beta^{t_1+t_2)} = q\}$.

Ein Automat $\mathcal{A}$ akzeptiert ein Wort α genau dann, wenn der Durchlauf dem Akzeptanzkriterium des Automaten genügt. Was dieses Akzeptanzkriterium aussagt, wird durch den Automatentypen festgelegt. Es werden nun zwei Automatentypen vorgestellt, der Büchi-Automat und der Rabin-Automat.

2.2.1 Büchi-Automat

Der Büchi-Automat wird durch ein Zustandsübergangssystem $\mathcal{G} = (Q, \Sigma, \delta, q_0)$ und durch eine Finalmenge $Q_F \in Q$ beschrieben. Ein Wort wird genau dann akzeptiert, wenn mindestens ein Zustand aus der Finalmenge Q_F unendlich oft durchlaufen wird. Der Büchiautomat akzeptiert dann die folgende Sprache:

$$\mathcal{L}_B(\mathcal{A}) := \{\alpha \in \Sigma^\infty | \exists \beta. \in RUN(\alpha, G).INF(\beta) \cap Q_F \neq \{\}\}$$

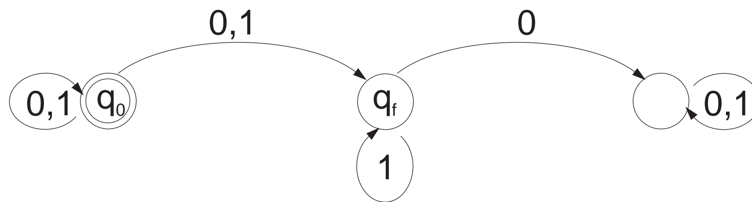


Abbildung 2.1: Nichtdeterministischer Büchiautomat: $\mathcal{L} = \{0, 1\}^* \cdot 1^\infty$

Zum Beispiel akzeptiert der in Abb. 2.1 dargestellte indeterministische Büchiautomat alle Worte mit unendlich vielen Einsen am Ende. Durch den Indeterminismus ist es möglich für Worte mit unendlich vielen Einsen am Ende Durchläufe zu konstruieren, die das Akzeptanzkriterium nicht erfüllen. Bei einem indeterministischen Automaten kommt es aber nur darauf an, daß ein akzeptierender Durchlauf existiert, denn dann kann der Automat das Wort konstruieren und es gehört demnach zu seinem Sprachumfang. Man kann zeigen, daß die Menge der ω-regulären Sprachen und die von indeterministischen Büchiautomaten akzeptierten Sprachen identisch sind [Thom90a] [Chou74]. Die deterministische Version des Büchiautomaten leistet dieses nicht, wie man an dem obigen Beispiel sieht. Ein deterministischer Automat kann nämlich nicht entscheiden, ab welchem Zeitpunkt nur noch Einsen folgen, weil der Durchlauf mit dem Eingabewort eindeutig bestimmt ist.

2.2.2 Rabin-Automat

Das Akzeptanzkriterium eines Rabin-Automaten wird durch Zustandstuppel (R_0, S_0), ..., $(R_f, S_f)\})$, $R_i, S_i \in Q$, charakterisiert. Der Automat akzeptiert ein Wort genau dann, wenn ein Durchlauf und ein Paar R_j, S_j existieren mit

1. mindestens ein Zustand aus R_j wird unendlich oft durchlaufen und

2. von einem gewissen Zeitpunkt an werden keine Zustände aus S_j mehr durchlaufen.

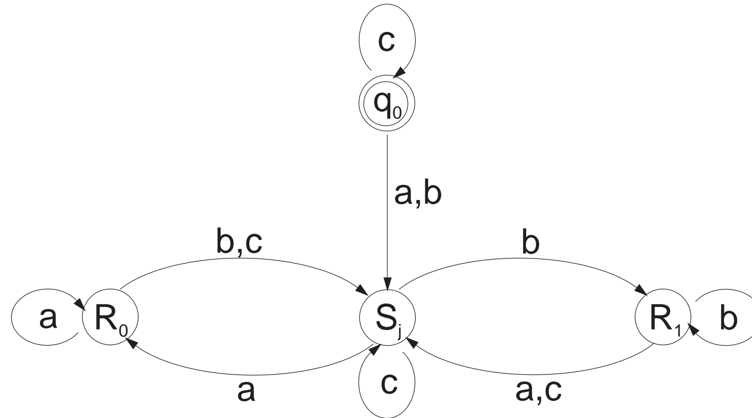


Abbildung 2.2: Deterministischer Rabinautomat

Rabin-Automaten sind im deterministischen wie im indeterministischen Fall gleich mächtig wie die ω-regulären Sprachen [Chou74]. Den Beweis, daß Rabin-Automaten mindestens so mächtig sind, wie Büchi-Automat, ist der, daß man den Büchi-Automaten einfach auf einen Rabin-Automaten zurückführen kann. Man muß nur $f = 0, R_0 = Q_F$ und $S_0 = \{\}$ setzen. Der deterministische Rabin-Automat aus Abb. 2.2 arbeitet über dem Alphabet $\Sigma = \{a, b, c\}$ und akzeptiert alle Worte, die unendlich viele a's oder b's am Ende haben.

2.3 Automatenformeln

Im letzten Abschnitt wurden Automaten eingeführt. Mit diesen Automaten kann man testen, ob ein Wort zu einer Sprache gehört.

Automatenformeln dienen der Beschreibung von Automaten in Logik höherer Ordnung. Dazu ist es erforderlich, daß die Zustände und das Alphabet mit booleschen Werten kodiert werden, damit boolesche Verknüpfungen möglich werden.
Damit läßt sich dann der Zustandsübergang in Logik beschreiben.

$$\mathcal{T}(\vec{i}, \vec{q}) := \forall t. \left(\vec{q}^{(0)} \leftrightarrow q_0\right) \wedge \left(\vec{q}^{(t+1)} \leftrightarrow \Omega(\vec{i}^{(t)}, \vec{q}^{(t)})\right)$$

Dabei ist zu beachten, daß $\vec{q}$ die Abfolge der Zustände über der Zeit beschreibt. Ebenso beschreibt $\vec{i}$ die Abfolge der Eingaben über der Zeit. $\mathcal{T}(\vec{i}, \vec{q})$ wird genau dann wahr, wenn $\vec{q}$ ein Durchlauf von $\vec{i}$ darstellt.

Das Akzeptanzkriterium macht Aussagen über die Finalmenge. Die Finalmenge kann in Aussagenlogik mit ihrer charakteristischen Funktion $\Phi(q)$ beschrieben werden. Das

Akzeptanzkriterium macht dann Aussagen über Φ. Eine Automatenformel hat damit folgendes allgemeines Aussehen.

$$AF^{\Phi}_{\omega,\Omega}(\vec{i}) := \Big(\exists \vec{q}.\mathcal{T}(\vec{i},\vec{q}) \wedge \Phi(q^t)\Big)$$

Die Formel ist folgendermaßen zu lesen. Die Eingabe-Sequenz $\vec{i}$ gehört genau dann zur Sprache des Automaten, wenn es einen Durchlauf $\vec{q}^t$ gibt, der sowohl TRANS$(\vec{i},\vec{q})$ als auch die Aussage Φ erfüllt.

Definition 2.3.1 (Deterministische Büchiformeln [Buec62] [Schn95b])
Beschreibe $\vec{\omega}$ den Anfangszustand des Automaten, $\Omega(\vec{i},\vec{q})$ den Zustandsübergang und $\Phi(\vec{q})$ die Akzeptanzaussage, dann bildet die folgende Formelmenge die Menge der deterministischen Büchiformeln:

$$\left(\begin{array}{l} \exists \vec{q}. \\ \quad \Big[\forall t.\ \big(\vec{q}^{(0)} \leftrightarrow \vec{\omega}\big) \wedge \Big(\vec{q}^{(t+1)} \leftrightarrow \vec{\Omega}(\vec{i}^{(t)},\vec{q}^{(t)})\Big)\Big] \wedge \\ \quad \big[\forall t_1.\exists t_2.\Phi(\vec{q}^{(t_1+t_2)})\big] \end{array}\right)$$

Definition 2.3.2 (Deterministische Rabinformeln [Schn95b])
Beschreibe $\vec{\omega}$ den Anfangszustand des Automaten, $\Omega(\vec{i},\vec{q})$ den Zustandsübergang und $\Phi(\vec{q})$ die Akzeptanzaussage, dann bildet die folgende Formelmenge die Menge der deterministischen Rabinformeln:

$$\left(\begin{array}{l} \exists \vec{q}. \\ \quad \Big[\forall t.\ \big(\vec{q}^{(0)} \leftrightarrow \vec{\omega}\big) \wedge \Big(\vec{q}^{(t+1)} \leftrightarrow \vec{\Omega}(\vec{i}^{(t)},\vec{q}^{(t)})\Big)\Big] \wedge \\ \quad \displaystyle\bigvee_{j=0}^{f} \big[\forall t_1.\exists t_2.\Phi_j(\vec{q}^{(t_1+t_2)})\big] \wedge \big[\exists t_1.\forall t_2.\Psi_j(\vec{q}^{(t_1+t_2)})\big] \end{array}\right)$$

2.4 Boolesche Operationen auf deterministischen Automatenformeln

Die Klasse der deterministischen Automatenformeln ist bezüglich der booleschen Operationen nicht immer abgeschlossen. So ist der Büchiautomat bezüglich der Negation nicht abgeschlossen, weil nach [Chou74] für die Formel $\exists t_1.\forall t_2.p^{(t_1+t_2)}$ kein deterministischer Büchiautomat existiert. [Thom90a]

Die Klasse der deterministischen Rabinautomaten ist nach [Chou74] bezüglich der booleschen Verknüpfungen abgeschlossen.

Allgemein gilt, daß die Klasse der deterministischen Automatenformeln genau dann abgeschlossen ist, wenn sich jede boolesche Verknüpfung der Akzeptanzbedingungen wieder in der selben Automatenklasse darstellen läßt.

Um den Beweis der booleschen Verknüpfungen von Automatenformeln einfach zu halten wird das folgende Theorem eingeführt.

Theorem 2.4.1 (Implikationsform deterministischer Automaten)
Sei $\vec{\omega} \in I\!B^n$ und $\vec{\Omega}(\vec{a}, \vec{b}, \vec{c})$, $\Phi(\vec{a}, \vec{b})$ aussagenlogische Formeln mit den Aussagenvariablen $\vec{a}, \vec{b}, \vec{c}$ dann gilt:

$$\begin{array}{l}\forall \omega\, \Omega\, \Phi\, \vec{i}\, t_0 . \\ \quad \left(\begin{array}{l} \exists \vec{q}. \\ \quad \left[\forall t.\, \left(\vec{q}^{(t_0)} \leftrightarrow \vec{\omega}\right) \wedge \left(\vec{q}^{(t+t_0+1)} \leftrightarrow \vec{\Omega}(\vec{i}^{(t+t_0)}, \vec{q}^{(t+t_0)})\right)\right] \wedge \Phi(\vec{i}, \lambda t. \vec{q}^{(t+t_0)}) \end{array} \right) \\ \leftrightarrow \\ \quad \left(\begin{array}{l} \forall \vec{q}. \\ \quad \left[\forall t.\, \left(\vec{q}^{(t_0)} \leftrightarrow \vec{\omega}\right) \wedge \left(\vec{q}^{(t+t_0+1)} \leftrightarrow \vec{\Omega}(\vec{i}^{(t+t_0)}, \vec{q}^{(t+t_0)})\right)\right] \rightarrow \Phi(\vec{i}, \lambda t. \vec{q}^{(t+t_0)}) \end{array} \right)\end{array}$$

<u>*Beweis:*</u> Siehe [Schn95b]. ■

Mit Hilfe des Theorems und einfachen prädikatenlogischen Gesetzen wie $(\neg \forall x. P(x)) = (\exists x. \neg P(x))$ folgt das folgende Korollar.

Korollar 2.4.1 (Boolesche Operationen auf det. Automatenformeln)
Seien $\mathcal{T}(\vec{\omega}_1, \vec{\Omega}_1, \vec{i})\vec{q}_1$ und $\mathcal{T}(\vec{\omega}_2, \vec{\Omega}_2, \vec{i})\vec{q}_2$ deterministische Zustandsübergangsformeln, dann gilt:

1. $\neg \left(\exists \vec{q}_1 . \mathcal{T}(\vec{\omega}_1, \vec{\Omega}_1, \vec{i})\vec{q}_1 \wedge \Phi(\vec{i}, \vec{q}_1)\right) \leftrightarrow \left(\exists \vec{q}_1 . \mathcal{T}(\vec{\omega}_1, \vec{\Omega}_1, \vec{i})\vec{q}_1 \wedge \neg \Phi(\vec{i}, \vec{q}_1)\right)$
2. *Für* $* \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$:

$$\begin{array}{l}\quad \left[\left(\exists \vec{q}_1 . \mathcal{T}(\vec{\omega}_1, \vec{\Omega}_1, \vec{i})\vec{q}_1 \wedge \Phi_1(\vec{i}, \vec{q}_1)\right) * \left(\exists \vec{q}_2 . \mathcal{T}(\vec{\omega}_2, \vec{\Omega}_2, \vec{i})\vec{q}_2 \wedge \Phi_2(\vec{i}, \vec{q}_2)\right)\right] \\ \leftrightarrow \\ \quad \left[\exists \vec{q}_1 \vec{q}_2 . \mathcal{T}(\vec{\omega}_1, \vec{\Omega}_1, \vec{i})\vec{q}_1 \wedge \mathcal{T}(\vec{\omega}_2, \vec{\Omega}_2, \vec{i})\vec{q}_2 \wedge \left(\Phi_1(\vec{i}, \vec{q}_1) * \Phi_2(\vec{i}, \vec{q}_2)\right)\right]\end{array}$$

Kapitel 3

Hardwareformeln

Automatenformeln eignen sich, um Automaten mit Logik zu beschreiben. Die in diesem Kapitel vorgestellten Hardwareformeln sind erweiterte Automatenformeln, die den Ansprüchen der Hardware-Verifikation angepaßt sind. Dazu gehört zum einen die Möglichkeit die Struktur einer Schaltung beschreiben zu können, und zum anderen wird die Akzeptanzbedingung den Erfordernissen angepaßt. Um die Struktur einer Schaltung zu beschreiben, werden interne Leitungsvariablen eingeführt. Die Erfahrung zeigt, daß bei der Verifikation hauptsächlich Sicherheits- und Lebendigkeits-Eigenschaften nachgewiesen werden müssen.

Sicherheits-Eigenschaften (SE) sind Anforderungen an die Schaltung, die immer gelten müssen. So kann z.B. eine SE einer Ampelsteuerung sein, daß niemals zwei sich kreuzende Richtungen gleichzeitig grün bekommen.

Lebendigkeits-Eigenschaften (LE) sind Anforderungen an die Schaltung, die irgendwann einmal zu mindestens einen Zeitpunkt gelten müssen. Eine LE ist z.B., daß nach dem Rücksetzen einer Schaltung die Initialisierung irgendwann einmal abgeschlossen ist. Damit hat eine Hardwareformel folgendes Aussehen.

$$\mathcal{H}(\vec{i}, t_0) := \left(\begin{array}{l} \exists\vec{\ell}.\exists\vec{q}. \\ \quad \left[\forall t.\vec{\ell}^{(t+t_0)} \leftrightarrow \vec{\Delta}(\vec{i}^{(t+t_0)}, \vec{\ell}^{(t+t_0)}, \vec{q}^{(t+t_0)})\right] \wedge \\ \quad \left[\forall t.\ \left(\vec{q}^{(t_0)} \leftrightarrow \vec{\omega}\right) \wedge \left(\vec{q}^{(t+t_0+1)} \leftrightarrow \vec{\Omega}(\vec{i}^{(t+t_0)}, \vec{\ell}^{(t+t_0)}, \vec{q}^{(t+t_0)})\right)\right] \wedge \\ \quad \bigvee_{j=0}^{f} \left[\forall t.\Phi(\vec{i}^{(t+t_0)}, \vec{\ell}^{(t+t_0)}, \vec{q}^{(t+t_0)})\right] \wedge \left[\exists t.\Psi(\vec{i}^{(t+t_0)}, \vec{\ell}^{(t+t_0)}, \vec{q}^{(t+t_0)})\right] \end{array} \right)$$

Dabei bedeutet:
$\vec{i}$: Eingabewort
t_0 : definierter Zeitpunkt
$\vec{l}$: interne Leitungsvariablen
$\vec{q}$: interne Zustandsvariablen
$\vec{\Delta}$: Leitungs-Übergangsfunktion
$\vec{\Omega}$: Zustandsvariablen-Übergangsfunktion

Φ_j: Sicherheitseigenschaft als aussagenlogische Formel
Ψ_j: Lebendigkeitseigenschaft als aussagenlogische Formel

Die HWF ist nun folgendermaßen zu lesen: $\mathcal{H}(\vec{i}, t_0)$ wird genau dann wahr[1], wenn zu einem Eingabewort $\vec{i}$ eine Folge von Zuständen $\vec{q}$ und eine Folge von Leitungszuständen $\vec{l}$ existiert, die sowohl den Übergangsfunktionen $\vec{\Delta}$ und $\vec{\Omega}$ genügen als auch eines der Paare der SE und LE erfüllt.

Das Vorhandensein von mehreren SE und LE ist für die boolesche Abgeschlossenheit der HWFn zwingend notwendig, wie man an Korollar 2.4.1 sehen kann.

Jede HWF läßt sich in eine äquivalente HWF ohne interne Leitungen umwandeln, indem man alle möglichen Einsetzungen der Leitungen ineinander durchführt (siehe [Schn95b]. Dabei bleiben nur die Leitungen übrig, die Ausgänge von speichernden Elementen sind. Die wichtigste Eigenschaft der HWFn ist ihre boolesche Abgeschlossenheit, denn dadurch wird es möglich HWFn zu veknüpfen [Schn95b].

Beispiel

Anhand eines Beispiels soll nun vorgeführt werden, wie mit HWFn die Struktur einer Schaltung direkt beschrieben werden kann und wie die internen Leitungen bis auf die Zustandsleitungen eliminiert werden können. Die Schaltung stellt einen 110-Detektor dar, der am Ausgang o genau dann eine 1 liefert, wenn am Eingang i die Folge 110 anlag. Im weiteren Verlauf der Arbeit wird anhand dieses Beispiels gezeigt, wie die Verifikation mit Hardwareformeln abläuft.

Zunächt werden für die Grundgatter HWFn definiert [Schn95b].

$$\begin{array}{ll}
\mathsf{EQUIV}(i_1, i_2, o) & :\leftrightarrow \forall t. o^{(t)} \leftrightarrow i_1^{(t)} \leftrightarrow i_2^{(t)} \\
\mathsf{AND}(i_1, i_2, o) & :\leftrightarrow \forall t. o^{(t)} \leftrightarrow i_1^{(t)} \wedge i_2^{(t)} \\
\mathsf{OR}(i_1, i_2, o) & :\leftrightarrow \forall t. o^{(t)} \leftrightarrow i_1^{(t)} \vee i_2^{(t)} \\
\mathsf{INV}(i, o) & :\leftrightarrow \forall t. o^{(t)} \leftrightarrow \neg i^{(t)} \\
\mathsf{NAND}(i_1, i_2, o) & :\leftrightarrow \forall t. o^{(t)} \leftrightarrow \neg \left(i_1^{(t)} \wedge i_2^{(t)}\right) \\
\mathsf{NOR}(i_1, i_2, o) & :\leftrightarrow \forall t. o^{(t)} \leftrightarrow \neg \left(i_1^{(t)} \vee i_2^{(t)}\right) \\
\mathsf{XOR}(i_1, i_2, o) & :\leftrightarrow \forall t. o^{(t)} \leftrightarrow i_1^{(t)} \oplus i_2^{(t)} \\
\mathsf{MUX}(s, i_1, i_2, o) & :\leftrightarrow \forall t. o^{(t)} \leftrightarrow \left(s^{(t)} \Rightarrow i_1^{(t)} \middle| \, i_2^{(t)}\right) \\
\mathsf{DFF}(i, o) & :\leftrightarrow \exists q. \left[\forall t. \left(q^{(0)} \leftrightarrow \mathsf{F}\right) \wedge \left(q^{(t+1)} \leftrightarrow i^{(t)}\right)\right] \wedge \left[\forall t. o^{(t)} \leftrightarrow q^{(t)}\right] \\
\mathsf{TFF}(i, o) & :\leftrightarrow \exists q. \left[\forall t. \left(q^{(0)} \leftrightarrow \mathsf{F}\right) \wedge \left(q^{(t+1)} \leftrightarrow i^{(t)} \oplus q^{(t)}\right)\right] \wedge \left[\forall t. o^{(t)} \leftrightarrow q^{(t)}\right] \\
\mathsf{JKFF}(j, k, o) & :\leftrightarrow \exists q. \left(\begin{array}{l} \forall t. \left[q^{(0)} \leftrightarrow \mathsf{F}\right] \wedge \\ \quad \left[q^{(t+1)} \leftrightarrow \left(j^{(t)} \wedge \neg q^{(t)}\right) \wedge \left(\neg k^{(t)} \wedge q^{(t)}\right)\right] \wedge \\ \quad \left[\forall t. o^{(t)} \leftrightarrow q^{(t)}\right] \end{array}\right)
\end{array}$$

Mit diesen HWFn, die bis auf die speichernden Elemente nur aus einer SE bestehen, kann nun durch geeignete Konjunktion für den 110-Detektor eine HWF entwickelt werden.

[1]wahr heißt, daß $\vec{i}$ zur spezifizierten Sprache gehört

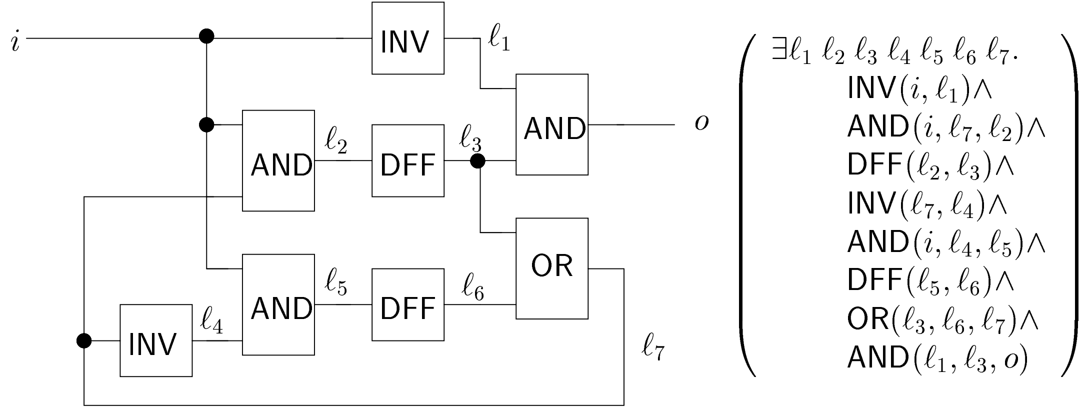


Abbildung 3.1: Implementierung des 110-Detektors und dessen formale Beschreibung.

Der 110-Detektor soll am Ausgang o genau dann eine 1 liefern, sobald am Eingang die Folge 110 anlag. Jeder Gatterausgang wird durch eine Leitungsvariable repräsentiert, die wiederum als Eingang eines anderen Gatters benutzt wird, ausnahme der Ausgang o.

Die formale Beschreibung der Schaltung ist wie folgt zu lesen. Es existieren Belegungen für die Leitungsvariablen, $l_1 ... l_7$, so daß die Aussagen $INV(i, l_1)$[2] $AND(i, l_7, l_2)$[3], ... $AND(l_1, l_3, o)$ wahr werden. Für diese Belegung gilt, daß der Ausgang o die gewünschte Funktion $f(i)$ erfüllt.

Nun müssen für die Gatter in der formalen Beschreibung die entsprechenden Definitionen eingesetzt werden, und danach soweit wie möglich ineinander eingesetzt werden.

$$
\left(\begin{array}{l}
\exists \ell_1\ \ell_2\ \ell_3\ \ell_4\ \ell_5\ \ell_6\ \ell_7. \\
\qquad \mathsf{INV}(i, \ell_1) \wedge \\
\qquad \mathsf{AND}(i, \ell_7, \ell_2) \wedge \\
\qquad \mathsf{DFF}(\ell_2, \ell_3) \wedge \\
\qquad \mathsf{INV}(\ell_7, \ell_4) \wedge \\
\qquad \mathsf{AND}(i, \ell_4, \ell_5) \wedge \\
\qquad \mathsf{DFF}(\ell_5, \ell_6) \wedge \\
\qquad \mathsf{OR}(\ell_3, \ell_6, \ell_7) \wedge \\
\qquad \mathsf{AND}(\ell_1, \ell_3, o)
\end{array}\right)
\leftrightarrow
\left(\begin{array}{l}
\exists \ell_1\ \ell_2\ \ell_3\ \ell_4\ \ell_5\ \ell_6\ \ell_7. \\
\quad \left(\forall t. \ell_1^{(t)} \leftrightarrow \neg i^{(t)}\right) \wedge \\
\quad \left(\forall t. \ell_2^{(t)} \leftrightarrow i^{(t)} \wedge \ell_7^{(t)}\right) \wedge \\
\quad \left(\forall t. \left[\ell_3^{(0)} \leftrightarrow \mathsf{F}\right] \wedge \left[\ell_3^{(t+1)} \leftrightarrow \ell_2^{(t)}\right]\right) \wedge \\
\quad \left(\forall t. \ell_4^{(t)} \leftrightarrow \neg \ell_7^{(t)}\right) \wedge \\
\quad \left(\forall t. \ell_5^{(t)} \leftrightarrow i^{(t)} \wedge \ell_4^{(t)}\right) \wedge \\
\quad \left(\forall t. \left[\ell_6^{(0)} \leftrightarrow \mathsf{F}\right] \wedge \left[\ell_6^{(t+1)} \leftrightarrow \ell_5^{(t)}\right]\right) \wedge \\
\quad \left(\forall t. \ell_7^{(t)} \leftrightarrow \ell_3^{(t)} \vee \ell_6^{(t)}\right) \wedge \\
\quad \left(\forall t. o^{(t)} \leftrightarrow \ell_1^{(t)} \wedge \ell_3^{(t)}\right)
\end{array}\right)
$$

Durch Einsetzen und Streichen der kombinatorischen Leitungsvariablen[4] in die Flip-Flop Gleichungen erhält man folgende HWF, die die Schaltung des 110-Detoktors beschreibt.

[2] $l_1 \leftrightarrow \neg i$

[3] $l_2 \leftrightarrow i \wedge l_7$

[4] Ausgänge der kombinatorischen Gatter

$$\mathcal{H}_{IMP_{110}} = \left(\begin{array}{l} \exists \ell_3\ \ell_6. \\ \quad \left(\forall t.\ \left[\ell_3^{(0)} \leftrightarrow \mathsf{F}\right] \wedge \left[\ell_3^{(t+1)} \leftrightarrow i^{(t)} \wedge \left\{\ell_3^{(t)} \vee \ell_6^{(t)}\right\}\right]\right) \wedge \\ \quad \left(\forall t.\ \left[\ell_6^{(0)} \leftrightarrow \mathsf{F}\right] \wedge \left[\ell_6^{(t+1)} \leftrightarrow i^{(t)} \wedge \neg \left\{\ell_3^{(t)} \vee \ell_6^{(t)}\right\}\right]\right) \wedge \\ \quad \left(\forall t. o^{(t)} \leftrightarrow \neg i^{(t)} \wedge \ell_3^{(t)}\right) \end{array} \right)$$

IMP gibt dabei an, daß die Hardwareformel die Implementierung der Schaltung beschreibt.

Kapitel 4

Hierarchische Verifikation

Das Hauptproblem bei der Verifikation liegt darin, daß die Schaltungen zu komplex werden können. Man kann die Schaltung zwar noch mit HWFn beschreiben, doch ein Beweis solch hochkomplexer Formeln ist aus Komplexitätsgründen nicht mehr möglich. Deshalb wird hier versucht die Komplexität der zu beweisenden Schaltung zu verringern.
Komplexe Schaltungen sind modular aufgebaut. Die Gesamtfunktion setzt sich aus der Kombination von Teilfunktionen zusammen. Die Hierarchische Verifikation führt eine schrittweise Verifikation der Gesamtschaltung durch. Dazu werden zunächst die Teilschaltungen verifiziert. Dann wird die IMP der Gesamtschaltung nicht durch die Konjunktion der IMP_j der Teilschaltungen beschrieben, sondern durch die Konjunktion der SPEC_j der Teilschaltungen. Die SPEC_j sind im allgemeinen weniger komplex als die IMP_j, weil i.a. nicht das gesamte Verhalten der Schaltung interessiert, sondern nur Teilaspekte für die Verifikation von Bedeutung sind. Damit verringert sich die Komplexität für den Beweisaufwand. Denn mit jeder Zustandsvariablen, die eingespart werden kann, kann sich der Beweisaufwand halbieren.
Abbildung 4.1 zeigt das Vorgehen der Hierarchischen Verifikation[1].

Um eine Verifikation einer Schaltung durchführen zu können, müssen die Implementierungsbeschreibung IMP und die Spezifikation SPEC vorliegen. Dabei spielt es keine Rolle, ob Bottom-up oder Top-down verifiziert wird. Bei dem Bottom-up Verfahren wird die Schaltung zuerst bis in ihre Grundmodule rekursiv zerlegt. Für die Grundmodule, die als korrekt angenommen werden, gibt es nur noch eine SPEC, die das interessierende Verhalten des Moduls beschreibt. Aus den SPEC_j der Teilmodule, die SPEC_j wurden als korrekt bewiesen, wird nun durch Konjunktion die IMP des Moduls gebildet. Dieses Modul kann dann bezüglich seiner SPEC mit dieser IMP verifiziert werden. Diese Schritte setzen sich rekursiv fort, bis die eigentliche Schaltung bewiesen wurde. Die Top-down Version des Beweises läuft gerade anders herum. Zuerst wird die Gesamtschaltung unter der Voraussetzung, daß die SPEC_j korrekt sind, verifiziert. Ist dies geschehen, so müssen im nächsten Schritt die SPEC_j der Teilmodule verifiziert werden. Dies wird rekursiv bis zu den Grundmodulen der Schaltung fortgesetzt. Diese müssen als korrekt angesehen werden.

[1] IMP seht für Implementierungsbeschreibung und SPEC für Spezifikation.

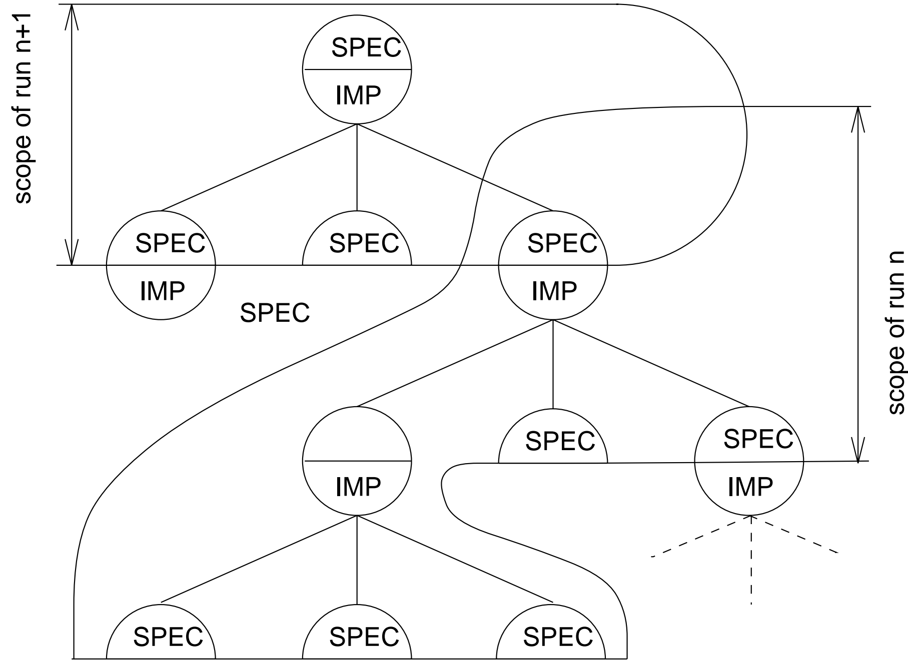


Abbildung 4.1: Hierarchische Verifikation

In Abbildung 4.1 kommt ein Fall vor, wo es zu einem Modul nur eine IMP aber keine SPEC gibt. Dieser Fall kann vorkommen, wenn die SPEC komplexer sein sollte als die IMP. In einem solchen Fall wird für die SPEC einfach die IMP eingesetzt. Dies ist problemlos möglich, weil beide IMP und SPEC mit der selben Formelklasse beschrieben wurden.

Das beschriebene Vorgehen der hierarchischen Verifikation basiert auf der folgenden Ableitungsregel:

$$\frac{\vdash \overbrace{\left(\exists \vec{x}.\ \bigwedge_{j=0}^{n} \mathrm{IMP}_j(\vec{i}_j, \vec{o}_j)\right)}^{\mathrm{IMP}(\vec{i},\vec{o})} \rightarrow \mathrm{SPEC}(\vec{i}, \vec{o})}{\vdash \bigwedge_{j=0}^{n} \mathrm{IMP}_j(\vec{i}_j, \vec{o}_j) \rightarrow \mathrm{SPEC}_j(\vec{i}_j, \vec{o}_j) \;\Big|\; \vdash \left(\bigwedge_{j=0}^{n} \mathrm{SPEC}_j(\vec{i}_j, \vec{o}_j)\right) \rightarrow \mathrm{SPEC}(\vec{i}, \vec{o})}$$

Um zu zeigen, daß aus der IMP die SPEC folgt, muß man zuerst beweisen, daß aus den IMP_j die SPEC_j folgen, und daß dann aus der Konjunktion der SPEC_j die SPEC folgt. Alle Operationen sind auf der Stuktur der HWFn definiert. Somit läßt sich der Beweis der Korrektheit einer Schaltung bezüglich einer Spezifikation erbringen.

Bemerkung: Soll eine Schaltung bezüglich einer SPEC verifiziert werden, so müssen

dementsprechend für alle Teilmodule dazu passende SPEC_j gegeben werden. Soll die gleiche Schaltung auch bezüglich einer anderen SPEC_2 verifiziert werden, so müssen i.a. auch die SPEC_j angepaßt werden.

Kapitel 5

Temporallogik und HWFn

Hardwareformeln eignen sich, um einen wichtigen Teil der linearen temporalen Logik zu beschreiben, die bei der Hardwareverifikation nützlich ist. Mit Temporallogik lassen sich Spezifikationen oftmals relativ einfach beschreiben. HWFn sind bezüglich des *when* und des *next* Operators abgeschlossen, d.h. *HWF when ALT* (ALT = Aussagenlogischer Term) oder *next HWF* ergeben wieder eine HWF. Für die anderen Operatoren können universelle Automatenformeln angegeben werden, die jedoch nicht immer in eine HWF überführt werden können. Für den *next* Operator wird die übliche symbolische Abkürzung $\bigcirc$ verwendet.

Die folgenden beiden Theoreme zeigen, wie temporallogische Aussagen in HWFn überführt werden.

Theorem 5.0.2 (Temporale Operatoren als Hardwareformeln)
Alle temporale Operatoren können durch universelle Hardwareformeln charakterisiert werden.[1] *Die entsprechenden Theoreme lauten wie folgt:*

- $[x \ \mathsf{ATNEXT}\ b]^{(t_0)} := \left(\begin{array}{l} \exists p\, q. \\ \quad \left[\forall t.\ \left(p^{(t_0)} = \mathsf{F}\right) \wedge \left(p^{(t+t_0+1)} = \mathsf{T}\right)\right] \wedge \\ \quad \left[\forall t.\ \left(q^{(t_0)} = F\right) \wedge \right. \\ \qquad \left. \left(q^{(t+t_0+1)} = p^{(t+t_0)} \wedge \left(q^{(t+t_0)} \vee b^{(t+t_0)}\right)\right)\right] \wedge \\ \quad \left[\forall t. \neg p^{(t+t_0)} \vee q^{(t+t_0)} \vee \neg b^{(t+t_0)} \vee x^{(t+t_0)}\right] \end{array} \right)$
- $[x \ \mathsf{WHEN}\ b]^{(t_0)} := \left(\begin{array}{l} \exists q. \\ \quad \left[\forall t.\ \left(q^{(t_0)} = F\right) \wedge \right. \\ \qquad \left. \left(q^{(t+t_0+1)} = q^{(t+t_0)} \vee b^{(t+t_0)}\right)\right] \wedge \\ \quad \left[\forall t. q^{(t+t_0)} \vee \neg b^{(t+t_0)} \vee x^{(t+t_0)}\right] \end{array} \right)$

---

[1] Dabei können für die Platzhalter x und b beliebige logische Konstrukte eingesetzt werden. Aber nicht jede Belegung führt zu einer Hardwareformel. Wenn x und b aussagenlogische Formeln sind, dann ergibt sich bei allen Umformungen eine HWF.

- $[x\ \mathsf{UNTIL}\ b]^{(t_0)} := \left(\begin{array}{l} \exists q. \\ \quad \left[\forall t.\ \left(q^{(t_0)} = F\right) \wedge \right. \\ \qquad \left.\left(q^{(t+t_0+1)} = q^{(t+t_0)} \vee b^{(t+t_0)}\right)\right] \wedge \\ \quad \left[\forall t. q^{(t+t_0)} \vee b^{(t+t_0)} \vee x^{(t+t_0)}\right] \end{array} \right)$
- $[x\ \mathsf{WHILE}\ b]^{(t_0)} := \left(\begin{array}{l} \exists q. \\ \quad \left[\forall t.\ \left(q^{(t_0)} = F\right) \wedge \right. \\ \qquad \left.\left(q^{(t+t_0+1)} = q^{(t+t_0)} \vee \neg b^{(t+t_0)}\right)\right] \wedge \\ \quad \left[\forall t. q^{(t+t_0)} \vee \neg b^{(t+t_0)} \vee x^{(t+t_0)}\right] \end{array} \right)$
- $[x\ \mathsf{BEFORE}\ b]^{(t_0)} := \left(\begin{array}{l} \exists q. \\ \quad \left[\forall t.\ \left(q^{(t_0)} = F\right) \wedge \right. \\ \qquad \left.\left(q^{(t+t_0+1)} = q^{(t+t_0)} \vee x^{(t+t_0)}\right)\right] \wedge \\ \quad \left[\forall t. q^{(t+t_0)} \vee \neg b^{(t+t_0)} \vee x^{(t+t_0)}\right] \end{array} \right)$
- $(\Box x)^{(t_0)} := \forall t. x^{(t+t_0)}$
- $(\Diamond x)^{(t_0)} := \exists t. x^{(t+t_0)}$
- $(\bigcirc x)^{(t_0)} := \left(\begin{array}{l} \exists p\ q. \\ \quad \left[\forall t.\ \left(p^{(t_0)} = \mathsf{F}\right) \wedge \left(p^{(t+t_0+1)} = \mathsf{T}\right)\right] \wedge \\ \quad \left[\forall t.\ \left(q^{(t_0)} = \mathsf{F}\right) \wedge \left(q^{(t+t_0+1)} = p^{(t+t_0)}\right)\right] \wedge \\ \quad \left[\forall t. \neg p^{(t+t_0)} \vee q^{(t+t_0)} \vee x^{(t+t_0)}\right] \end{array} \right)$
- $x^{(t_0)} := \left(\begin{array}{l} \exists p. \\ \quad \left[\forall t.\ \left(p^{(t_0)} = \mathsf{F}\right) \wedge \left(p^{(t+t_0+1)} = \mathsf{T}\right)\right] \wedge \\ \quad \left[\forall t. x^{(t+t_0)} \vee p^{(t+t_0)}\right] \end{array} \right)$

Theorem 5.0.3 (Abgeschlossenheit der Hardwareformeln bzgl. $\bigcirc$ und WHEN)

Sei $\mathcal{H}(\vec{i}, t_0)$ die folgende Hardwareformel:

$$\mathcal{H}(\vec{i}, t_0) := \left(\begin{array}{l} \exists \vec{q}. \\ \quad \left[\forall t.\ \left(\vec{q}^{(t_0)} = \vec{\omega}\right) \wedge \left(\vec{q}^{(t+t_0+1)} = \vec{\Omega}\left(\vec{i}^{(t+t_0)}, \vec{q}^{(t+t_0)}\right)\right)\right] \wedge \\ \quad \bigvee_{j=0}^{f} \left[\forall t. \Phi_j\left(\vec{i}^{(t+t_0)}, \vec{q}^{(t+t_0)}\right)\right] \wedge \left[\exists t. \Psi_j\left(\vec{i}^{(t+t_0)}, \vec{q}^{(t+t_0)}\right)\right] \end{array} \right)$$

Dann gilt:

1. $\left[\left(\lambda t. \mathcal{H}(\vec{i}, t)\right)\ \mathsf{WHEN}\ b\right]^{(t_0)}$ *ist zu der folgenden Hardwareformel äquivalent:*

$$\left(\begin{array}{l} \exists p\ \vec{q}. \\ \quad \left[\forall t.\ \left(p^{(t_0)} = \mathsf{F}\right) \wedge \left(p^{(t+t_0+1)} = p^{(t+t_0)} \vee b^{(t+t_0)}\right)\right] \wedge \\ \quad \left[\forall t.\ \left(\vec{q}^{(t_0)} = \vec{\omega}\right) \wedge \left(\vec{q}^{(t+t_0+1)} = \left(\left(b^{(t+t_0)} \vee p^{(t+t_0)}\right) \Rightarrow \vec{\Omega}\left(\vec{i}^{(t+t_0)}, \vec{q}^{(t+t_0)}\right) \middle| \vec{\omega}\right)\right)\right] \wedge \\ \quad \left[\left[\forall t. \neg b^{(t+t_0)}\right] \vee \bigvee_{j=0}^{f} \left(\begin{array}{l} \left[\forall t.\ \left(b^{(t+t_0)} \vee p^{(t+t_0)}\right) \rightarrow \Phi_j\left(\vec{i}^{(t+t_0)}, \vec{q}^{(t+t_0)}\right)\right] \wedge \\ \left[\exists t.\ \left(b^{(t+t_0)} \vee p^{(t+t_0)}\right) \wedge \Psi_j\left(\vec{i}^{(t+t_0)}, \vec{q}^{(t+t_0)}\right)\right] \end{array} \right) \right] \end{array} \right)$$

2. $\bigcirc\left(\lambda t.\mathcal{H}(\vec{i},t)\right)^{(t_0)}$ *ist zu der folgenden Hardwareformel äquivalent:*

$$\left(\begin{array}{l}\exists p\,\vec{q}.\\ \quad\left[\forall t.\left(p^{(t_0)}=\mathsf{F}\right)\wedge\left(p^{(t+t_0+1)}=\mathsf{T}\right)\right]\wedge\\ \quad\left[\forall t.\left(\vec{q}^{(t_0)}=\vec{\omega}\right)\wedge\left(\vec{q}^{(t+t_0+1)}=\left(p^{(t+t_0)}\Rightarrow\vec{\Omega}(\vec{i}^{(t+t_0)},\vec{q}^{(t+t_0)})\middle|\,\vec{\omega}\right)\right)\right]\wedge\\ \quad\bigvee_{j=0}^{f}\left[\forall t.p^{(t+t_0)}\rightarrow\Phi_j(\vec{i}^{(t+t_0)},\vec{q}^{(t+t_0)})\right]\wedge\left[\exists t.p^{(t+t_0)}\wedge\Psi_j(\vec{i}^{(t+t_0)},\vec{q}^{(t+t_0)})\right]\end{array}\right)$$

3. *Gilt ferner* $\Psi_j(\vec{i}^{(t+t_0)},\vec{q}^{(t+t_0)})\equiv\mathsf{T}$, *so ist* $\left[\left(\lambda t.\mathcal{H}(\vec{i},t)\right)\ \mathsf{WHEN}\ b\right]^{(t_0)}$ *zu der folgenden Hardwareformel äquivalent:*

$$\left(\begin{array}{l}\exists p\,\vec{q}.\\ \quad\left[\forall t.\left(p^{(t_0)}=\mathsf{F}\right)\wedge\left(p^{(t+t_0+1)}=p^{(t+t_0)}\vee b^{(t+t_0)}\right)\right]\wedge\\ \quad\left[\forall t.\left(\vec{q}^{(t_0)}=\vec{\omega}\right)\wedge\left(\vec{q}^{(t+t_0+1)}=\left(\left(b^{(t+t_0)}\vee p^{(t+t_0)}\right)\Rightarrow\vec{\Omega}(\vec{i}^{(t+t_0)},\vec{q}^{(t+t_0)})\middle|\,\vec{\omega}\right)\right)\right]\wedge\\ \quad\bigvee_{j=0}^{f}\left[\forall t.\left(b^{(t+t_0)}\vee p^{(t+t_0)}\right)\rightarrow\Phi_j(\vec{i}^{(t+t_0)},\vec{q}^{(t+t_0)})\right]\end{array}\right)$$

Beispiel In Kapitel 3 wurde für einen 110-Detektor die Implementierungsbeschreibung IMP_110 in die Hardwareformel $\mathcal{H}_{IMP_{110}}$ umgewandelt. Nun soll für die Spezifikation $SPEC_{110}$ eine Hardwareformel $\mathcal{H}_SPEC_{110}$ konstruiert werden. Die Spezifikation des 110-Detektors mit Temporallogik lautet:
$\bigcirc\bigcirc out = i\wedge\bigcirc i\wedge\neg\bigcirc\bigcirc i$

D.h. der Ausgang *out* wird genau dann 1, wenn vor 2 Zeitpunkten i=1 galt und vor einem Zeitpunkt i=1 galt und zum jetzigen Zeitpunkt i=0 gilt. Die Gleichung kann nun in die Pränexe $\bigcirc$-Normalform umgewandelt werden [Schn95b]. Dabei werden die $\bigcirc$-Operatoren vor die Gleichung gezogen, und es werden Zustandsvariablen eingeführt, die die Werte $\bigcirc i$ und $\neg\bigcirc\bigcirc i$ speichern. Man fängt zuerst mit der Gleichung $\bigcirc\bigcirc(out = p\wedge q\wedge\neg i)$ an und multipliziert die $\bigcirc$ Operatoren die Gleichung. Man erhält
$\bigcirc\bigcirc out = \bigcirc\bigcirc p\wedge\bigcirc\bigcirc q\wedge\neg\bigcirc\bigcirc i$

Ein Vergleich mit der Spezifikation führt zu der Formel
$(\bigcirc\bigcirc p = i)\wedge(\bigcirc\bigcirc q = \bigcirc i)\wedge\bigcirc\bigcirc(out = p\wedge q\wedge\neg i)$
$\rightarrow(\bigcirc\bigcirc p = i)\wedge(\bigcirc q = i)\wedge\bigcirc\bigcirc(out = p\wedge q\wedge\neg i)$
$\rightarrow(\bigcirc\bigcirc p = \bigcirc q)\wedge(\bigcirc q = i)\wedge\bigcirc\bigcirc(out = p\wedge q\wedge\neg i)$
$\rightarrow(\bigcirc p = q)\wedge(\bigcirc q = i)\wedge\bigcirc\bigcirc(out = p\wedge q\wedge\neg i)$

Diese Formel kann nun in eine HWF ohne kombinatorische Leitungsvariabllen umgewandelt werden, indem die ersten beiden Gleichungen in die Transitionsform überführt werden und die letzte Gleichung als Akzeptanzbedingung angesehen wird. Damit erhält man dann:

$$\mathcal{H}_{SPEC_{110}} := \left(\begin{array}{l} \exists p\ q. \\ \quad \left[\forall t. \left(p^{(t_0)} \leftrightarrow F\right) \wedge \left(p^{(t+t_0+1)} \leftrightarrow q^{(t+t_0)}\right)\right] \wedge \\ \quad \left[\forall t. \left(q^{(t_0)} \leftrightarrow F\right) \wedge \left(q^{(t+t_0+1)} \leftrightarrow i^{(t+t_0)}\right)\right] \wedge \\ \quad \left[\forall t. out^{(t)} \leftrightarrow p^{(t)} \wedge q^{(t)} \wedge \neg i^{(t)}\right] \end{array} \right)$$

Damit wurden nun die Implementierungsbeschreibung und die Spezifikation mit HWFn beschrieben. Jetzt kann der Verifikationsschritt $\mathcal{H}_{IMP_{110}} \to \mathcal{H}_{SPEC_{110}}$ durchgeführt werden.

$$\left(\begin{array}{l} \exists \ell_3\ \ell_6. \\ \quad \left(\forall t. \left[\ell_3^{(0)} \leftrightarrow \mathsf{F}\right] \wedge \left[\ell_3^{(t+1)} \leftrightarrow i^{(t)} \wedge \left\{\ell_3^{(t)} \vee \ell_6^{(t)}\right\}\right]\right) \wedge \\ \quad \left(\forall t. \left[\ell_6^{(0)} \leftrightarrow \mathsf{F}\right] \wedge \left[\ell_6^{(t+1)} \leftrightarrow i^{(t)} \wedge \neg \left\{\ell_3^{(t)} \vee \ell_6^{(t)}\right\}\right]\right) \wedge \\ \quad \left(\forall t. o^{(t)} \leftrightarrow \neg i^{(t)} \wedge \ell_3^{(t)}\right) \end{array} \right)$$
$$\to \left(\begin{array}{l} \exists p\ q. \\ \quad \left(\forall t. \left[p^{(0)} = \mathsf{F}\right] \wedge \left[p^{(t+1)} = i^{(t)}\right]\right) \wedge \\ \quad \left(\forall t. \left[q^{(0)} = \mathsf{F}\right] \wedge \left[q^{(t+1)} = p^{(t)}\right]\right) \wedge \\ \quad \left(\forall t. o^{(t)} = q^{(t)} \wedge p^{(t)} \wedge \neg i^{(t)}\right) \end{array} \right)$$

Das obige Beweisziel kann man nun durch Anwendung der booleschen Operationen für die Implikation auf die folgende Form bringen.

$$\left(\begin{array}{l} \exists p\ q\ \ell_3\ \ell_6. \\ \quad \left(\forall t. \left[\ell_3^{(0)} \leftrightarrow \mathsf{F}\right] \wedge \left[\ell_3^{(t+1)} \leftrightarrow i^{(t)} \wedge \left\{\ell_3^{(t)} \vee \ell_6^{(t)}\right\}\right]\right) \wedge \\ \quad \left(\forall t. \left[\ell_6^{(0)} \leftrightarrow \mathsf{F}\right] \wedge \left[\ell_6^{(t+1)} \leftrightarrow i^{(t)} \wedge \neg \left\{\ell_3^{(t)} \vee \ell_6^{(t)}\right\}\right]\right) \wedge \\ \quad \left(\forall t. \left[p^{(0)} = \mathsf{F}\right] \wedge \left[p^{(t+1)} = i^{(t)}\right]\right) \wedge \\ \quad \left(\forall t. \left[q^{(0)} = \mathsf{F}\right] \wedge \left[q^{(t+1)} = p^{(t)}\right]\right) \wedge \\ \quad \left\{\left(\exists t. o^{(t)} \oplus \neg i^{(t)} \wedge \ell_3^{(t)}\right) \vee \left(\forall t. o^{(t)} = q^{(t)} \wedge p^{(t)} \wedge \neg i^{(t)}\right)\right\} \end{array} \right)$$

Die berechnete Formel muß jetzt noch bewiesen werden. Wenn die Formel für alle möglichen Eingaben in allen möglichen Zuständen zu wahr evaluiert wird, dann genügt Spezifikation der Implementierungsbeschreibung. Als Beweissystem bietet sich der Model-Checker SMV an,dessen Eingabesprache CTL ist. Mit CTL lassen sich u.a. Zustandsübergänge und Bedingungen an den Automaten stellen, die in jedem Zustand gelten sollen. Damit lassen sich HWFn sehr einfach beweisen.

SMV allokiert zur Repräsentation der Übergangsgleichungen in einem BDD 115 Knoten. Nach der Reduktion bleiben davon 20 Knoten übrig. Insgesamt dauert der Beweisvorgang 33 Millisekunden.

Kapitel 6

Resümee der Studienarbeit

Der Beweis, daß eine HWF allgemeingültig ist sollte von SMV erbracht werden. SMV ist ein Model-Checker, der als Eingabesprache CTL benutzt. Dem Beweiser ist ein Programm vorgeschaltet, das das CTL-Programm in eine effiziente und gekürzte Datenstruktur umwandelt. Der Ablauf einer Verifikation sollte folgendermaßen aussehen:
1. Umwandlung der IMP in eine HWF HW_{imp}
2. Umwandlung der SPEC in eine HWF HW_{spec}
3. Berechnen der Implikation $HW = (HW_{imp} \rightarrow HW_{spec})$
4. Umwandlung von HW in ein CTL-Programm CP
5. Aufruf von SMV mit CP

Leider hat sich herausgestellt, daß SMV zum direkten Beweis von HWFn mit mehr als einer Sicherheits- und Lebendigkeits-Eigenschaft nicht in der Lage ist. Deshalb mußten die HWFn bevor sie bewiesen werden konnten in einen Büchiautomaten übersetzt werden 6.3.1. Doch hat die Büchiformel durch die Übersetzung bedingt immer mehr Zustandsvariablen als die zugehörige HWF [Schn95b].

6.1 Aufgabenstellung

In den bisherigen Kapitel wurde die für diese Arbeit erforderliche Theorie aus [Schn95b] vorgestellt. Die Aufgabe der Studienarbeit war es, den Beweisablauf für die Hardware-Verifikation mit HWF zu automatisieren und zu prüfen in wieweit sich HWFn in der Praxis einsetzen lassen. Als Programmiersprache wurde SML benutzt. SML ist eine Interpreter-Sprache. Geplant war am Ende der Arbeit eine komplette Schaltung hierarchisch zu Verifizieren was jedoch nicht gelang. Es konnte lediglich für die meisten Teilmodule gezeigt werden, daß sie der Spezifikation genügten. Das Problem der Verifikation lag in der Größe der zu beweisenden Automaten. So hatten diese teilweise mehr als 50 Zustandsvariablen, so daß der verwendete Model-Checker (SMV) überfordert war. Der Grund für diese Komplexität hat mehrere Ursachen die im folgenden erläutert werden.

6.2 Die Vermehrung der Zustandsvariablen

Bei allen Verknüpfungen und Umwandlungen von HWFn vergrößert sich die Anzahl von Zustandsvariablen. Wie, warum und in welchem Ausmaß wird im folgenden erläutert. Die Notation von HWFn sei wie folgt.

$\mathcal{H} = (Trans \wedge Akzeptanz)$
n_{Trans} = Anzahl der Transitionen
$n_{Akzeptanz}$ = Anzahl der Akzeptanzbedingungen

6.2.1 Disjunktion von HWFn

Die Disjunktion zweier HWFn ist die einfachste Verknüpfung von zwei HWFn.
Sei $\mathcal{H}_{disj}$ die zu $\mathcal{H}_1 \vee \mathcal{H}_2 =$ äquivalente HWF, welche nach dem Verfahren aus [Schn95b] entsteht. $\mathcal{H}_{disj}$ hat $n_{Trans1} + n_{Trans2}$ Transitionen (also auch Zustandsvariablen) und $n_{Akzeptanz1} + n_{Akzeptanz2}$ Akzeptanzpaare.

6.2.2 Konjunktion von HWFn

Die Konjunktion zweier HWFn ist aufwendiger als die Disjunktion. Zunächst einmal werden beide Transitionsteile $TRANS_1$ und $TRANS_2$ vereinigt, so daß sich auch hier die Anzahl der Zustandsvariablen addiert. Nun ist aber die Zusammenfassung der Akzeptanzbedingungen nicht so einfach, weil die beiden Akzeptanzbedingungen $ACCEPT_1$ und $ACCEPT_2$ (beides Disjunktionen) konjunktiv verknüpft werden. Diese Konjunktion muß nun wieder in eine Disjunktion umgewandelt werden. Beim Ausmultiplizieren entstehen dabei $n_{Akzeptanz1} * n_{Akzeptanz2}$ Terme. Für jede neue Lebendigkeits-Eigenschaft müssen neue Wächtervariablen eingeführt werden, so daß die Konjunktion $n_{Trans1} + n_{Trans2} + n_{Akzeptanz1} * n_{Akzeptanz2}$ neue Transitionen hat und $n_{Akzeptanz1} * n_{Akzeptanz2}$ Akzeptanzbedingungen besitzt.

6.2.3 Negation von HWFn

Die Negation ist die kritischste Operation auf den HWFn. Beim Ausmultiplizieren der negierten Akzeptanzbedingungen entstehen $2^{n_{Akzeptanz}}$ Terme. Das Problem sind wie schon bei der Konjunktion die Lebendigkeits-Eigenschaften, weil sich diese nicht so einfach zusammenfassen lassen. Die Negation erzeugt also $n_Trans + 2^{n_{Akzeptanz}}$ Transitionen und $2^{n_{Akzeptanz}}$ Akzeptanz-Bedingungen.

6.2.4 Implikation und Äquivalenz von HWFn

Diese beiden Operationen lassen sich einfach auf die drei Grundoperationen wie allgemein üblich zurückführen. Da die Disjunktion von zwei HWFn für die Verifikation von essentieller Bedeutung ist soll das Verhalten bezüglich Vergrößerung beschrieben werden. Um $HW_1 \rightarrow HW_2$ zu berechnen wird zunächst HW_1 negiert. Dabei kommen $2^{n_{Akzeptanz1}}$

neue Transitionen zu der Formel hinzu. Danach wird HW_2 disjunktiv damit verknüpft, so daß die resultierende Formel $n_{Trans1} + n_{Trans} + 2hochn_{Akzeptanz1}$ Zustandsvariablen und $n_{Akzeptanz1} + n_{Akzeptanz2}$ Akzeptanzbedingungen besitzt.

6.2.5 Temporallogische Operatoren

Die HWFn sind bezüglich des *next* und des *when* Operators abgeschlossen. D.h. *next HW* und *HW WHEN* t_0 sind wieder HWFn. Werden *next* und *when* auf HWFn angewandt, so werden jeweils zwei neue Zustandsvariablen gebraucht. Beim *when* Operator muß evtl. zusätzlich noch eine Sicherheits- Eigenschaft eingeführt werden, während sich beim *next* Operator die Anzahl der Akzeptanzbedingungen nicht ändert.
Werden die beiden Operatoren nicht auf HWFn sondern auf aussagenlogische Terme angewandt, so entstehen dabei HWFn mit einer Transitionsvariabeln und einer Akzeptanzbedingung.

Andere temporallogische Operatoren wie *until, while* und *before* sind bezüglich der HWFn nicht abgeschlossen [Schn95b]. Aber angewandt auf einen aussagenlogischen Term läßt sich immer eine äquivalente HWF konstruieren, die jeweils eine Sicherheits-Eigenschaft und eine Transition besitzt.

6.2.6 Umwandlung von HWFn in Büchi-Formeln

Bei der Umwandlung von HWFn in Büchi-Formeln [Schn95b] werden pro Sicherheits-Lebendigkeits-Eigenschaft der HWF zwei zusätzliche Zustandsvariablen benötigt. Die Sicherheits-Lebendigkeits-Eigenschaften werden in jeweils eine Unendlichkeits-Eigenschaft transformiert.

6.3 Vermeidung von zusätzlichen Zustandsvariablen

Die vorherigen Abschnitte haben gezeigt, daß jede Operation auf HWFn oder deren Umwandlung in andere Automaten-Formeln immer auch mit einem Zuwachs an Komplexität einhergehen. Schon bei kleinen Schaltungen, wie sie im Rahmen der Studienarbeit behandelt wurden, war SMV nicht mehr in der Lage in annehmbarer Zeit ein Ergebnis zu liefern. Dieses Problem liegt allerdings nur in der enormen Anzahl von Zustandsvariablen begründet, die den Suchraum von SMV aufspannen.

Bei den Operationen auf HWFn lassen sich direkt keine Einsparungen an Zustandsvariablen erzielen. Allerdings ist es sicherlich gut nach jeder Operation ein Verfahren zur Zustandsreduktion ablaufen zu lassen. z.B. Verfahren nach Paull/Unger. Die bisherige Implementierung eliminiert lediglich doppelte Vorkommen der Bauart $q^0 = F \wedge q^{t+1} = T$.

Die Umwandlung von HWFn in Büchi-Formeln bringt unnötiger Weise den Faktor 2 (bzgl. der Anzahl der Akzeptanzbedingungen) ins Spiel. Diese Umwandlung ist jedoch nur wegen SMV von Nöten. Eine große Einsparung von Zustandsvariablen wäre also möglich, wenn ein anderer auf HWFn zugeschnittener Beweiser verwendet würde.

Die größte Einsparung, die auch am einfachsten zu realisieren ist, ist das Beweisschema abzuändern. Das bisherige Beweisschema kombiniert IMP und SPEC zu einer großen $HWF = IMP \rightarrow SPEC$. Die Implikation besagt nun, daß in allen denjenigen Fällen, wo IMP richtig ist auch SPEC richtig sein muß. Richtig heißt in diesem Fall, daß das Ausgabewort der Schaltung zu einem Eingabewort gehört. Man kann nun also vom Beweiser alle die Fälle konstruieren lassen, in denen IMP zu wahr evaluiert wird, und nur für diese Fälle SPEC überprüfen. Diese Art der Verifikation lässt sich in SMV mit FAIRNESS-Bedingungen realisieren.

Die Hierarchische Verifikation hatte den Grundgedanken, daß die Spezifikation allgemeiner und somit weniger komplex sein sollte, als die Implementierung. Nun hat sich herausgestellt, daß bei Spezifikationen, die in temporallogischen Form angegeben werden, die Anzahl der Zustandsvariablen um ein Vielfaches größer sein kann, als die bei der eigentlichen Implementierung. Das liegt daran, daß oftmals spezifiziert wird, daß ein Ereignis nach n-Zeiteinheiten eintreten soll. Dabei entstehen aber direkt n neue Zustandsvariablen, obwohl ja eingentlich $logn$ Zustandsvariablen genügen müßten. Hier liegt also noch Potential zu einer entscheidenden Zustandsvariablen-Reduktion.

6.3.1 SMV und HWFn

Im Laufe der Arbeit hat sich herausgestellt, daß es nicht möglich war HWFn in ein für SMV korrektes CTL Programm zu übersetzen. Das Problem taucht immer dann auf, wenn die HWF mehr als eine Akzeptanzbedingung besitzt.
Die Akzeptanz der HWF habe zwei Sicherheits-Lebendigkeits-Eigenschaften
$\forall t.\Phi_1(t) \wedge \exists t.\Psi_1(t) \vee$
$\forall t.\Phi_2(t) \wedge \exists t.\Psi_2(t)$
Sei zum einfacheren Verständnis $\Psi_1(t) = \Phi_2(t) = T$, dann ergibt sich
$\forall t.\Phi_1(t) \vee$
$\exists t.\Psi_2(t)$
Diese Aussage wurde in die folgende CTL-Form überführt
$AG\Phi_1(t) \vee AF\Psi_2(t)$
Diese Form sagt aber nur aus, daß in jedem erreichbaren Zustand $\Phi_1(t)$ gilt, oder aber daß es zu jedem erreichbaren Zustand auf allen Pfaden einen Folgezustand gibt, der $\Psi_2(t)$ erfüllt. Es muß aber gezeigt werden, daß in jedem Zustand für alle folgenden Zustände Φ_1 gilt oder, daß es einen Folgezustand gibt, in dem Ψ_2 gilt.

Kapitel 7

Experimentelle Ergebnisse

7.1 Ablauf einer Verifikation

Der Ablauf einer Verifikation wird anhand des 110-Detektors, der in Kapitel 3 vorgestellt wurde, vorgeführt.

Da noch keine Funktion zur Leitungselimination implementiert wurde, wird direkt die gekürzte Formel 3 verwendet. Diese Formel kann nun als sml-Term angegeben werden.

Bevor mit dem Beispiel begonnen wird, soll die Notation von sml kurz erläutert werden.

```
val var1 = 1               weisst der Variablen var1 den Integerwert 1 zu
(--' term1:bool '--)       liefert einen Term vom Typ bool
[1,2,3]                    liefert eine Liste mit den Eintraegen 1,2,3
{1, (--' term1:bool '--)} liefert einen Rekord mit den Eintraegen 1 und term1
```

Die HWFn wurden in SML als Rekord implementiert. Dieser besteht aus aus einem Initialisierungszeitpunkt, der in InitPoit steht, einer Liste der ungebundenen Variablen FreeVars, einer Liste von Transitionen Transitions und einer Liste von Akzeptanzbedingungen Acceptance.
Eine Transition ist als Rekord implementiert und besteht aus der Zustandsvariablen StateVar, dem booleschen Wert dieser zum Initialisierungszeitpunkt und der Übergangsfunktion.
Eine Akzeptanzbedingung ist als Rekord implementiert und besteht aus einer Sicherheitseigenschaft Safety und einer Lebendigkeitseigenschaft Liveness.

```
val imp110_hw = {InitPoint = (--`t0:num`--),
          FreeVars = [(--`i:num->bool`--),(--`out:num->bool`--)],
          Transitions = [{StateVar = (--`l3:num->bool`--),
                          Init = (--`l3(t0:num) = F`--),
                          Next = (--` !t. l3(t+t0+1) = i(t+t0) /\
                                      (l3(t+t0) \/ l6(t+t0))`--)},
                         {StateVar = (--`l6:num->bool`--),
                          Init = (--`l6(t0:num) = F`--),
                          Next = (--` !t. l6(t+t0+1) = i(t+t0) /\
                                      ~(l3(t+t0) \/ l6(t+t0))`--)}],
          Acceptance = [{Safety = (--`!t. out(t:num):bool = ~i(t) /\ l3(t)`--),
                         Liveness = (--`T`--)}]};
```

Ebenso wird die Spezifikation ?? des Detektors als sml-Term eingegeben.

```
val spec110_temp = (--`(NEXT(NEXT out) t) =
                     (i t) /\  (NEXT i t) /\  ~(NEXT(NEXT i) t)  `--);
```

Diese Formel kann nun mit bereitgestellten Funktionen in eine HWF transformiert werden. Dazu wird zunächst $spec110_{temp}$ in die pränexe Normalform gebracht und dann im zweiten Schritt in eine HWF übersetzt.

```
val np = mk_next_prenex 0 spec110_temp;
val np =
  (--`!t.
        ?p0 p1.
          (ALWAYS (\t. NEXT p0 t = i t) t /\
           ALWAYS (\t. NEXT p1 t = p0 t) t) /\
          NEXT (NEXT (\t. out t = p1 t /\ p0 t /\ ~(i t))) t`--) : term

val spec110_hw = next_prenext2hwrec np;
val spec110_hw =
  {Acceptance=[{Liveness=(--`?t. next1 (t + t0) /\ next0 (t + t0) /\ T`--),
                Safety=(--`!t.
                             next1 (t + t0) ==>
                             next0 (t + t0) ==>
                             (out (t + t0) =
                              p1 (t + t0) /\ p0 (t + t0) /\ ~(i (t + t0))) \/
                             prop0 (t + t0)`--)}],
   FreeVars=[(--`out`--),(--`i`--)],InitPoint=(--`t0`--),
   Transitions=[{Init=(--`prop0 t0 = F`--),
                 Next=(--`!t.
                            prop0 (t + t0 + 1) =
                            next1 (t + t0) /\ next0 (t + t0) /\ T`--),
                 StateVar=(--`prop0`--)},
                {Init=(--`next0 t0 = F`--),
                 Next=(--`!t. next0 (t + t0 + 1) = next1 (t + t0) /\ T`--),
                 StateVar=(--`next0`--)},
                {Init=(--`next1 t0 = F`--),
                 Next=(--`!t. next1 (t + t0 + 1) = T`--),
                 StateVar=(--`next1`--)},
                {Init=(--`p0 t0 = F`--),
                 Next=(--`!t. p0 (t + t0 + 1) = i (t + t0)`--),
                 StateVar=(--`p0`--)},
                {Init=(--`p1 t0 = F`--),
                 Next=(--`!t. p1 (t + t0 + 1) = p0 (t + t0)`--),
                 StateVar=(--`p1`--)}]} : hw_record
```

Nun kann die Implikation angewandt werden, weil beide Beschreibungen als HWF vorliegen.

```
val hw = hw_imp imp110_hw spec110_hw;
val hw =
  {Acceptance=[{Liveness=(--`?t. out t = i t \/ ~(l3 t)`--),Safety=(--`T`--)},
               {Liveness=(--`?t. next1 (t + t0) /\ next0 (t + t0) /\ T`--),
```

```
               Safety=(--`!t.
                            next1 (t + t0) ==>
                            next0 (t + t0) ==>
                            (out (t + t0) =
                             p1 (t + t0) /\ p0 (t + t0) /\ ~(i (t + t0))) \/
                            prop0 (t + t0)`--)}],
   FreeVars=[(--`out`--),(--`i`--)],InitPoint=(--`t0`--),
   Transitions=[{Init=(--`l3 t0 = F`--),
                 Next=(--`!t.
                           l3 (t + t0 + 1) =
                           i (t + t0) /\ (l3 (t + t0) \/ l6 (t + t0))`--),
                 StateVar=(--`l3`--)},
                {Init=(--`l6 t0 = F`--),
                 Next=(--`!t.
                           l6 (t + t0 + 1) =
                           i (t + t0) /\ ~(l3 (t + t0) \/ l6 (t + t0))`--),
                 StateVar=(--`l6`--)},
                {Init=(--`prop0 t0 = F`--),
                 Next=(--`!t.
                           prop0 (t + t0 + 1) =
                           next1 (t + t0) /\ next0 (t + t0) /\ T`--),
                 StateVar=(--`prop0`--)},
                {Init=(--`next0 t0 = F`--),
                 Next=(--`!t. next0 (t + t0 + 1) = next1 (t + t0) /\ T`--),
                 StateVar=(--`next0`--)},
                {Init=(--`next1 t0 = F`--),
                 Next=(--`!t. next1 (t + t0 + 1) = T`--),
                 StateVar=(--`next1`--)},
                {Init=(--`p0 t0 = F`--),
                 Next=(--`!t. p0 (t + t0 + 1) = i (t + t0)`--),
                 StateVar=(--`p0`--)},
                {Init=(--`p1 t0 = F`--),
                 Next=(--`!t. p1 (t + t0 + 1) = p0 (t + t0)`--),
                 StateVar=(--`p1`--)}]} : hw_record
```

Da es mit SMV nicht möglich ist HWFn direkt zu beweisen muß die HWF *hw* zunächst in eine Büchiformel übersetzt werden 6.3.1.

```
val buechi = hwrec2buechi hw;
val buechi =
  {Acceptance=(--`!t1.
                    ?t2.
                      buechi2 (t1 + t2) /\ ~(buechi0 (t1 + t2)) \/
                      buechi5 (t1 + t2) /\ ~(buechi3 (t1 + t2))`--),
   FreeVars=[(--`out`--),(--`i`--)],InitPoint=(--`t0`--),
   Transitions=[{Init=(--`l3 t0 = F`--),
                 Next=(--`!t.
                           l3 (t + t0 + 1) =
                           i (t + t0) /\ (l3 (t + t0) \/ l6 (t + t0))`--),
                 StateVar=(--`l3`--)},
                {Init=(--`l6 t0 = F`--),
                 Next=(--`!t.
```

```
           l6 (t + t0 + 1) =
           i (t + t0) /\ ~(l3 (t + t0) \/ l6 (t + t0))‘--),
 StateVar=(--‘l6‘--)},
{Init=(--‘prop0 t0 = F‘--),
 Next=(--‘!t.
           prop0 (t + t0 + 1) =
           next1 (t + t0) /\ next0 (t + t0) /\ T‘--),
 StateVar=(--‘prop0‘--)},
{Init=(--‘next0 t0 = F‘--),
 Next=(--‘!t. next0 (t + t0 + 1) = next1 (t + t0) /\ T‘--),
 StateVar=(--‘next0‘--)},
{Init=(--‘next1 t0 = F‘--),
 Next=(--‘!t. next1 (t + t0 + 1) = T‘--),
 StateVar=(--‘next1‘--)},
{Init=(--‘p0 t0 = F‘--),
 Next=(--‘!t. p0 (t + t0 + 1) = i (t + t0)‘--),
 StateVar=(--‘p0‘--)},
{Init=(--‘p1 t0 = F‘--),
 Next=(--‘!t. p1 (t + t0 + 1) = p0 (t + t0)‘--),
 StateVar=(--‘p1‘--)},
{Init=(--‘buechi0 t0 = F‘--),
 Next=(--‘!t. buechi0 (t + t0 + 1) = buechi0 (t + t0) \/ F‘--),
 StateVar=(--‘buechi0‘--)},
{Init=(--‘buechi1 t0 = F‘--),
 Next=(--‘!t.
           buechi1 (t + t0 + 1) =
           buechi1 (t + t0) \/
           (out (t + t0) = i (t + t0) \/ ~(l3 (t + t0)))‘--),
 StateVar=(--‘buechi1‘--)},
{Init=(--‘buechi2 t0 = F‘--),
 Next=(--‘!t.
           buechi2 (t + t0 + 1) =
           buechi2 (t + t0) /\ buechi0 (t + t0) \/
           ~(buechi2 (t + t0)) /\ buechi1 (t + t0)‘--),
 StateVar=(--‘buechi2‘--)},
{Init=(--‘buechi3 t0 = F‘--),
 Next=(--‘!t.
           buechi3 (t + t0 + 1) =
           buechi3 (t + t0) \/
           next1 (t + t0) /\
           next0 (t + t0) /\
           (out (t + t0) =
            ~(p1 (t + t0)) \/ ~(p0 (t + t0)) \/ i (t + t0)) /\
           ~(prop0 (t + t0))‘--),StateVar=(--‘buechi3‘--)},
{Init=(--‘buechi4 t0 = F‘--),
 Next=(--‘!t.
           buechi4 (t + t0 + 1) =
           buechi4 (t + t0) \/
           next1 (t + t0) /\ next0 (t + t0) /\ T‘--),
 StateVar=(--‘buechi4‘--)},
{Init=(--‘buechi5 t0 = F‘--),
 Next=(--‘!t.
```

```
                         buechi5 (t + t0 + 1) =
                         buechi5 (t + t0) /\ buechi3 (t + t0) \/
                         ~(buechi5 (t + t0)) /\ buechi4 (t + t0)`--),
              StateVar=(--`buechi5`--)}]} : buechi_record
```

Nun wird büchi in ein CTL Programm übersetzt:

```
val ctl_prog = buechi2smvprog buechi;
MODULE main

VAR
l3 :boolean;
l6 :boolean;
prop0 :boolean;
next0 :boolean;
next1 :boolean;
p0 :boolean;
p1 :boolean;
buechi0 :boolean;
buechi1 :boolean;
buechi2 :boolean;
buechi3 :boolean;
buechi4 :boolean;
buechi5 :boolean;
out :boolean;
i :boolean;

ASSIGN
init(l3) := 0;
init(l6) := 0;
init(prop0) := 0;
init(next0) := 0;
init(next1) := 0;
init(p0) := 0;
init(p1) := 0;
init(buechi0) := 0;
init(buechi1) := 0;
init(buechi2) := 0;
init(buechi3) := 0;
init(buechi4) := 0;
init(buechi5) := 0;
next(l3) := (i) & ( (l3) | (l6));
next(l6) := (i) & ( !((l3) | (l6)));
next(prop0) := (next1) & ( (next0) & (  1 ));
next(next0) := (next1) & (  1 );
next(next1) :=  1 ;
next(p0) := i;
next(p1) := p0;
next(buechi0) := (buechi0) | ( 0 );
next(buechi1) := (buechi1) | ((out) <-> ((i) | (!(l3))));
next(buechi2) := ((buechi2) & ( buechi0)) | ((!(buechi2)) & ( buechi1));
next(buechi3) := (buechi3) | ((next1) & ( (next0) &
```

```
                  ( ((out) <-> ((!(p1)) | ((!(p0)) | (i)))) & ( !(prop0)))));
next(buechi4) := (buechi4) | ((next1) & ( (next0) & (  1 )));
next(buechi5) := ((buechi5) & ( buechi3)) | ((!(buechi5)) & ( buechi4));

SPEC
(AG AF (((buechi2) & ( !(buechi0))) | ((buechi5) & ( !(buechi3)))))
val it = () : unit
```

Dieses CTL Programm wird nun als File abgelegt und kann dann an SMV übergeben werden

```
string2file (smv_examples^"detect110.smv") ctl_prog;

smv_call "detect110.smv";
-- specification AG AF (buechi2 & !buechi0 | buechi5 & !b... is true

resources used:
user time: 0.166667 s, system time: 0.0666667 s
BDD nodes allocated: 2635
Bytes allocated: 917504
BDD nodes representing transition relation: 394 + 1
reachable states: 228 (2^7.83289) out of 32768 (2^15)
val it = 0 : int
```

7.2 Bewiesene Schaltungen

Die folgenden Schaltungen wurden der Diplomarbeit [SSSK94] entnommen.

Schaltung	Zeit	BDD-Knoten	erreichbare Zustände	Zustände insgesamt
110-Detektor	0,07 s	2635	228	32768
DFF2	0,4 s	7655	21504	1.04858e+06
DFF4	53.9667 s	241253	6.83213e+06	2.68435e+08
RSFF	0.116667 s	2164	1088	16384
RESET	4.86667 s	49309	4.20454e+06	1.07374e+09

Literaturverzeichnis

[Buec62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. of International Congress of Logic, Methodology and Philosophy of Science*, pages 1–12. Stanford University Press, 1960.

[Chou74] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[Goed31] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme. *Monatshefte der math. Physik*, 38:173–198, 1931.

[Schn95b] K. Schneider. *Ein einheitlicher Ansatz zur Unterstützung von Abstraktionsmechanismen der Hardwareverifikation.* PhD thesis, University of Karlsruhe, P.O. Box 6980, 76128 Karlsruhe, 1995.

[SSSK94] A. Schneider, B. Straube, K. Schneider, and T. Kropf. Verifikation eines digitalen Netzwerkes mit Hilfe des Beweissystems HOL. Technical Report SFB358-C-1/94, FHG Dresden/Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, 1994. http://goethe.ira.uka.de/hvg/techreports/SFB358-C-1-94.ps.gz.

[Thom90a] W. Thomas. *Handbook of Theoretical Computer Science*, chapter Automata on Infinite Objects. North-Holland, 1990.