Hochschule München

Fakultät für Informatik und Mathematik.

Bachelorarbeit

Analyse von bit-Level Eigenschaften zur Hardware-Verifikation

Abgabetermin: 18. März 2022

Verfasser:
Klaus Riedl
Bruckmaier 1
82541 Münsing
Matrikelnummer: 22076817

Prüfer: Prof. Dr. Matthias Güdemann

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die Bachelorarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Münsing, 18.03.2022	Riedl.
Ort, Datum	Unterschrift

Abstract

Im Laufe der Entwicklungsphase der Industrie 4.0 wurde Hardware zunehmend komplexer und ließ dadurch den Anspruch an die Hardwarequalität steigen. Folgen dessen erhöhte sich der Stellenwert der Hardware-Verifikation.

Das Ziel in der vorliegenden Arbeit ist es den Einfluss von Bit-Ebenen Informationen auf die Hardwareverifikation auf der Wort-Ebene zu untersuchen. Im Detail soll die Frage beantwortet werden, ob die Modellüberprüfung auf der Wort-Ebene durch zusätzliche Informationen von der Bit-Ebene verstärkt werden kann. Um die Forschungsfrage zu beantworten, wurde ein Programm entwickelt, welches eine BTOR2 Datei mit zusätzlichen Bit-Informationen aus einer Bit-Ebenen Analyse ergänzt. Hierzu wird die BTOR2 Datei in das Bit-Ebenen Format Aiger umgewandelt, welche anschließend analysiert wird. Hier steht das BTOR2 Format stellvertretend für die Wort-Ebene und das Aiger Format für die Bit-Ebene. Mithilfe dieses Programmes soll ein Benchmark ausgeführt werden, die Aufschluss geben soll, ob die Bit-Informationen hilfreich sind.

Durch die Ergebnisse des Benchmarks ist klar erkennbar, dass zusätzliche Bit-Informationen aus einer Bit-Ebenen Analyse zu einer Verbesserung der Modellüberprüfung führen kann. Der Hauptpunkt der Verbesserung stellt das Aufwand-Nutzungs-Verhältnis im Zusammenhang mit der zeitlichen Gesamtausführungsdauer da.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Zum Einstieg in diese Arbeit soll durch eine Motivation die Relevanz von korrekter Hardware dargestellt werden und die Intention dieser Arbeit aufzeigen. Anschließend wird auf die Forschungsfrage eingegangen. Zum Abschluss der Einleitung wird der weitere Ablauf dieser Bachelorarbeit beschrieben.

1.1 Motivation

"Bisher hieß es immer: Computer machen keine Fehler. Dank modernster Hardware wurde jetzt auch dieses Manko beseitigt". - Holger Lamm [1]

Dieses Zitat entstand, nachdem im November 1994 der Pentium FDIV Fehler entdeckt wurde. Dieser Fehler war ein Hardwarefehler, der bei Pentium Prozessoren bei Divisionen bestimmter Gleitkommazahlen zu falschen Ergebnissen geführt hat.[2] Um solche Hardwarefehler zu vermeiden, gibt es den Prozess der Hardwareverifikation im Bereich der Qualitätssicherung bei einem Hardware-Entwurfsprozesses.

Durch die stets wachsende Komplexität von Hardware, steigt auch der Aufwand zur Qualitätssicherung. Jedoch sind die korrekte Funktionalität und Sicherheit der Hardware heute wichtiger als je zuvor. Da heutzutage die meiste Hardware nicht nur für digitale Fertigungsprozesse oder normalen Endverbraucher PCs verwendet wird, begegnen wir in unserem Alltag so gut wie überall Hardware, angefangen vom Motorsteuergerät des Autos bis hin zum Herzschrittmacher. Deshalb ist es heute wichtiger als je zuvor Hardwarefehler bei der Entwicklung von Hardware zu finden und auszubessern.

1.2 Fragestellung

Im Bereich der Hardwareverifikation gibt es verschiedene Formate für Beschreibungen. Die Formate basieren auf verschiedenen Abstraktionsebenen. Diese Arbeit beschäftigt sich mit der Frage, ob die Modellüberprüfung auf der Wort-Ebene mit zusätzlichen Informationen der Bit-Ebene verbessert werden kann. Um die Fragestellung zu beantworten, wird ein Programm entwickelt, welche dem Wort-Ebenen Format BTOR2 Informationen aus der Bit-Ebenen Analyse aus dem Bit-Ebenen Format Aiger hinzufügt.

1.3 Übersicht

Im nächsten Kapitel werden die Grundlagen der Hardwareverifikation erläutert. Im darauffolgenden Kapitel werden die benötigten Beschreibungsformate und andere Grundlagen beschrieben, welche notwendig sind, um der Bachelorarbeit folgen zu können. Anschließend wird im Kapitel 3 der theoretische Ansatz erläutert, sowie die benötigten Implementierungsschritte abgearbeitet. Des Weiteren folgen die Auswertung und Analyse einiger Testdaten. Mithilfe dieser Auswertung soll die Ursprungsfrage im Kapitel Abschluss beantwortet werden.

2 Grundlagen

2.1 Hardwareverifikation

Dieses Kapitel ist der Beginn des Grundlagenkapitels. In diesem wird zum einen eine allgemeine Einleitung zum Thema Hardwareverifikation gegeben. In dieser wird die Grundidee der Hardwareverifikation und die funktionsweise der einzelnen Verifikationsverfahren kurz erläutert. Anschließend wird auf die formale Verifikation genauer eingegangen.

2.1.1 Einleitung

Unter dem Begriff Hardwareverifikation versteht man den Überprüfungsprozess eines Hardware-Entwurfs. Hierbei wird die Hardware mithilfe einer Hardwarebeschreibungssprache (englisch Hardware Description Language, HDL) beschrieben. Der Entwurf basiert hierbei auf der zuvor festgelegten Spezifikation. In dieser werden die Anforderungen an die zu entwerfende Hardware festgehalten. Diese Anforderungen basieren auf den grundsätzlichen Bedarf an Zuverlässigkeit, Verfügbarkeit und Sicherheit. Die Verifikation spielt in der Hardwareentwurfsphase (siehe Abbildung 1 Gelbe Markierung) eine zentrale Rolle. Durch die immer komplexer werdenden Hardwaresysteme ist die Minimierung der Entwurfsfehler mit stetig steigendem Aufwand verbunden.

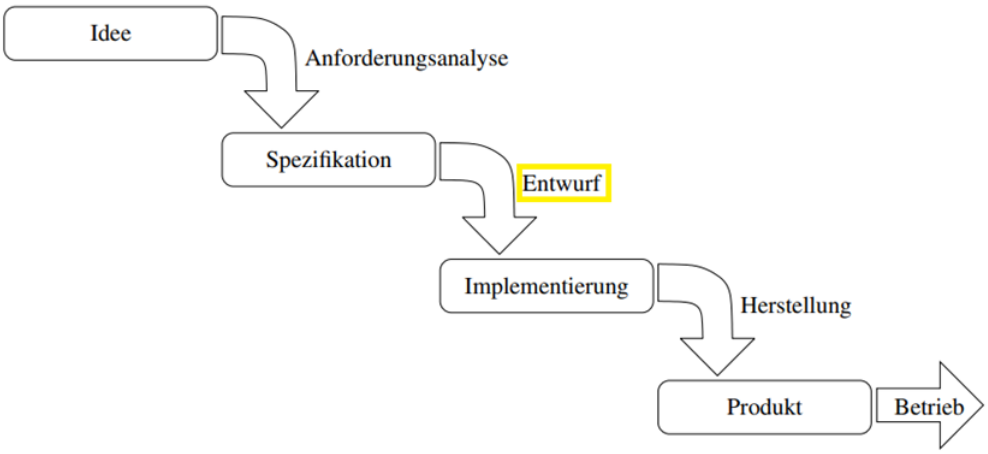


Abbildung 1: *Stark vereinfachter Entwicklung Prozess [9]*[2]

2.1.2 Verifikationsverfahren

Allgemein lassen sich die Verifikationsverfahren in die Bereiche Simulation, Emulation und formale Verifikation unterteilen.

- Simulation:
  Bei der Simulation wird überprüft, ob das System, welches aus dem zuvor definierten Entwurf entstanden ist, sich korrekt verhält. Dafür werden Testfälle mit verschieden Testeingaben erstellt. Die Testeingaben werden als Eingabe verwendet und für die Überprüfung der korrekten Funktion wird die Ausgabe verwendet.[14]

- Emulation:
  Bei der Emulation werden z.B. FPGA´s verwendet um Hardware Schaltungen zu realisieren. Durch die Realisierungen mit programmierbarer Hardware, kann diese vorab als Prototyp dargestellt werden, ohne das sie produziert werden muss. Durch eine solche prototypische Umsetzung, kann man diese auch auf nicht funktionale Eigenschaften wie Latenz, Durchsatz und Leistungsaufnahme testen. Dies ist mit formalen oder simulativen Methoden teilweise nicht oder nur sehr ungenau möglich.[14]

- Formale-Verifikation:
  Bei der formalen Verifikation erfolgt diese auf der Basis von mathematischer Logik und Beweisverfahren. Da sie eine wichtige Rolle in dieser Arbeit darstellt, wird sie im Folgendem genauer erläutert.[14]

2.1.3 Formale Verifikation Model Checking

Unter dem Begriff Model Checking (deutsch Modellüberprüfung) versteht man ein Verifikationsverfahren zur Verifikation eines Modells gegen eine Spezifikation. Hierzu wird ein Modell so lange untersucht, bis die Korrektheit der Spezifikation bewiesen oder widerlegt ist. Im Falle der Widerlegung wird ein entsprechendes Gegenbeispiel ausgegeben.

Es gibt zwei verschiedene Arten von Eigenschaften, welche durch die Spezifikationen festgelegt werden. Diese sind die Safety- und Liveness Eigenschaften. Die Safety Eigenschaft beschreibt den “schlimmsten Fall”, welcher unter gar keinen Umständen erfüllt werden darf und definiert somit einen verbotenen Zustand. Bei der Liveness Eigenschaft hingegen, muss es einen Fall geben wo diese Eigenschaft erfüllt wird. Die klassischen Liveness Eigenschaften sollen Deadlocks sowie Livelocks verhindern.

Bei der Modellprüfung gibt es mehrere Verifikationsarten, welche wiederum mehrere Verifikationsverfahren nutzen. Die in dieser Arbeit verwendeten Modellüber-

prüfungsprogramme verwenden den Ansatz der symbolischen Modellprüfung und nutzen hier das SAT-basierte Verifikationsverfahren.
Die Idee hinter der symbolischen Modellprüfung ist es, den Zustandsraum symbolisch darzustellen, womit die Zustandsmengen implizit durch boolesche Formeln repräsentiert werden. Eine solche Formel kann eine Menge von Zuständen beschreiben. Somit lassen sich große Zustandsmengen einfacher darstellen und manipulieren. Bei dem SAT-basierten Ansatz wird aus dem Modell sowie der Spezifikation eine aussagenlogische Formel erstellt, welche mithilfe eines SAT Solver auf Erfüllbarkeit geprüft wird. Neben dem SAT-basierten Ansatz gibt es noch den BDD-basierten Ansatz für symbolische Modellprüfung. Dieser Ansatz benutzt ein binäres Entscheidungsdiagramm (englisch binary decision diagram, BDD), welches dazu verwendet wird, um boolesche Funktionen darzustellen. Der BDD Ansatz hat den Nachteil, das Zustandsmengen explosiv ansteigen.

Beim SAT-basierten Ansatz spricht man auch von einer beschränkten Modellprüfung (englisch bounded model checking), bei dieser wird die maximale Länge eines Gegenbeispiels festgelegt. Die Idee dahinter ist es von k=0 bis k= [festgelegte Obergrenze] zu überprüfen, ob ein Gegenbeispiel der Länge k vorhanden ist. Der Buchstabe k gibt die Anzahl der Zyklen Durchläufe an.

Durch die Begrenzung von k lassen sich Probleme auf ein propositionales Erfüllbarkeitsproblem reduzieren, welches mit einen SAT-Solver gelöst werden kann. Da die SAT Methoden nicht unter einer zustandsraumen Explosion leiden, fällt der einschränkende Faktor des Speicherbedarfs, welcher der Flaschenhals beim binären Entscheidungsdiagramm ist, weg. Ein wesentlicher Nachteil des beschränkten Modellprüfungsansatzes ist, dass eine Oberschranke manuell festgelegt werden muss. Falls diese zu klein gewählt wird, kann dieser Ansatz unvollständig sein und wenn sie zu groß festgelegt wird, kann die Ausführungsdauer nicht mehr zielführend sein. [9][3]

Anhand des zuvor erwähnten Ansatzes wird deutlich, dass diese Art von Modellprüfung das Verifikationsziel die Falsifikation ist. Bei der Falsifikation geht es darum eine Annahme mithilfe eines Gegenbeispiels zu widerlegen. Somit wird bei erfolgreicher Falsifikation durch ein Gegenbeispiel gezeigt, dass eine geforderte Eigenschaft nicht erfüllt ist. [9][1]

Es gibt die Möglichkeit eine Verifikationsvollständigkeit mit Hilfe der Induktion zu erzielen. Diese Unterart der Induktion ist auch unter dem Namen K-Induktion bekannt. Somit kann man die K-Induktion als eine mögliche Erweiterung der beschränkten Modellüberprüfung betrachten, welche dazu verwendet wird, um Eigenschaften zu beweisen.

Die Grundidee ist, dass man neben dem Basisfall, welcher überprüft, ob eine Eigenschaft zum Zeitpunkt k verletzt ist, noch einen Induktionsfall hat. Dieser Fall beginnt nicht bei einem initialen Zustand, sondern am zu überprüfenden Endzustand. Als Ansatz ist es zu beweisen, dass bei einer Sequenz von gültigen Zuständen auch nur gültige Zustände folgen können. Die beiden Fälle werden in Abbildung 2 dargestellt.

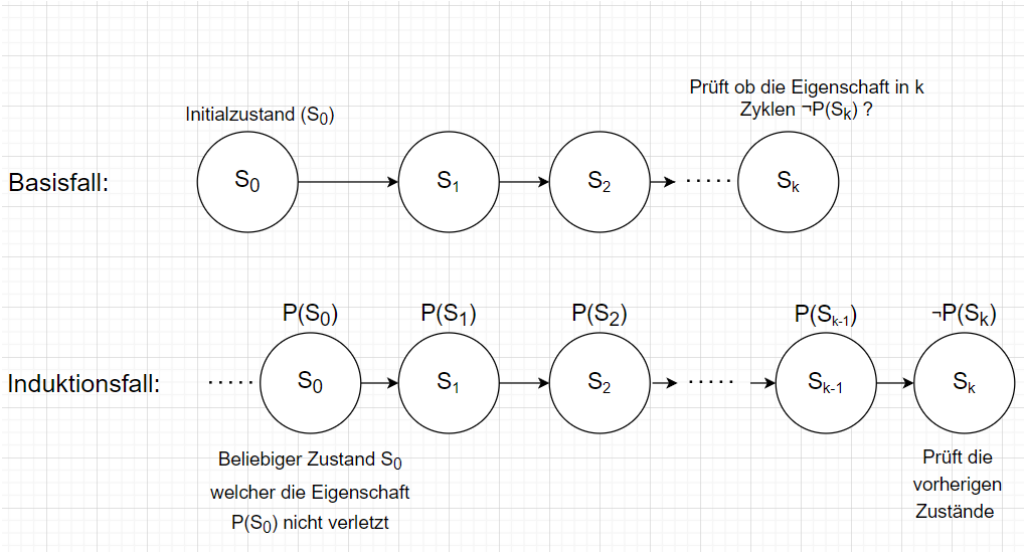


Abbildung 2: *Grafische Darstellung des Ansatzes der K-Induktion [15]*

2.2 Formate - Aiger/BTOR2

In dieser Bachelorarbeit soll festgestellt werden, ob man die Hardwareverifikation auf der Wort-Ebene durch das Verstärken mit Hilfe von Informationen aus der Bit-Ebene verbessern kann.
Hierzu wird für die Wort-Ebene das BTOR2 Format verwendet und für die Bit-Ebene das Aiger Format. Da es schwierig ist eine Hardware direkt in BTOR2 zu Beschreiben, wird die Hardwarebeschreibungssprache Verilog verwendet. Diese kann mithilfe des Open source Tool Yosys in eine BTOR2 Datei umgewandelt werden. In diesem Abschnitt der Arbeit sollen die Formate und deren zusammenhängende Entstehung, sowie Yosys erläutert werden.
Die Informationen aus den nachfolgenden zwei Kapitel, stammen,wenn nicht explizit eine Referenzierung angegeben ist, aus den technischen Berichten, welche auf der Webseite des "Institute for Formal Models and Verification" veröffentlicht wurden.[5] [6] [7] [8]

2.2.1 Bit-Ebene Aiger

Der Name Aiger besteht zum einen Teil aus dem Akronym AIG von And-Inverter Graphs und klingt, wenn er auf Deutsch ausgesprochen wird, wie der Name des "Eiger", eines Berges in den Schweizer Alpen. Die Namensgebung soll den Ursprung des Formats verdeutlichen. Der Grund hierfür ist, dass das Format zum ersten Mal öffentlich auf dem Alpine Verifikationstreffen 2006 in Ascona als Möglichkeit diskutiert wurde. Hierbei war es das Ziel, ein einfaches und kompaktes Dateiformat für einen an die CAV(Computer Aided Verification) Konferenz 2007 angeschlossenen Modellprüfungswettbewerb (Hardware Model Checking Competition Abk. HWMCC) zu schaffen.

Ein And-Inverter Graph (Abk. AIG) ist ein gerichteter, azyklischer Graph, der eine strukturelle Implementierung der logischen Funktionalität einer Schaltung oder eines Netzwerks darstellt. Wie der Name bereits erahnen lässt, besteht ein AIG aus Knoten mit zwei Eingängen, die ein logisches UND-Gatter darstellt, Endknoten, welche die Variablennamen beinhalten und Kanten mit optionaler Invertierung.

Durch die einfache Struktur von AIG lassen sich Boolesche Netzwerke durch simple Gatter Transformationsregeln in AIG umwandeln und auch verändern. Somit kann man Knotenpunkte einfach hinzufügen oder diese auch miteinander zusammenfügen. Zwar ist die Darstellung für große Schaltungen oder Netzwerke nicht sehr effizient, jedoch eignet sich dieser gut für die Darstellung von Manipulationen von booleschen Funktionen.[16]

Das Ziel des Aiger-Formates ist es, ein einfaches Format zu schaffen, um verschiedene Modellüberprüfungsprogramme miteinander zu vergleichen. Dies wäre ohne das Aiger Format nur sehr eingeschränkt möglich, da jedes Modellüberprüfungsprogramm sein eigenes Format hat. Zusätzlich gibt es auch Unterschiede in der Syntax und in der Semantik, wodurch ein Vergleich erschwert wird. Deshalb hat man sich beim Entwurf des Aiger Formats dazu entschieden die Semantik auf ein Minimum zu reduzieren, um das Format so einfach wie möglich zu halten. Dies hat dementsprechend zur Folge das man die Unterstützung dieses Formates leicht implementieren kann.

Weitere Anforderungen, welche die Designentscheidungen beeinflusst haben, waren:

- Modellierung von kombinierbaren Schaltungen
- Beschreibung von SAT Problemen
- Beschreibung von Modellprüfungsproblemen
- Modellierung von sequenziellen Schaltungen
- Beschränkung von Operationen auf die Bit-Ebene und müssen so einfach wie möglich sein
- Zuverfügungsstellung von einem standarddisiertem und kompakten binären Formats
- Erweiterung soll leicht sein

Im weiteren Verlauf der Aiger Entwicklung war der Punkt der Erweiterbarkeit besonders wichtig. Denn im Jahre 2011 wurde die Aiger Version 1.9 vorgestellt. Diese Version wurde syntaktisch so gestaltet, damit die Abwärtskompatibilität gewährleistet wird. Durch die Abwärtskompatibilität konnte im HWMCC 11 bei dem das Aiger Format verwendet wurde, noch Tools benutzen, die für das alte Format entwickelt wurden.

Die einzige Ausnahme war es, wenn neue Funktionen der Version 1.9 verwendet wurden. In der Version 1.9 sind fünf neue semantische Merkmale hinzugekommen:

- reset logic
- multiple properties
- invariant constraints
- justice properties
- fairness constraints

2.2.2 BTOR/BTOR2

Ein Vorschlag für die Ursprungsversion von BTOR (BTOR 1.0) wurde im Jahre 2008 ausgearbeitet. Die Intention war dieselbe wie bei der Aiger Entwicklung. Es sollte ein möglichst einfaches Format entstehen, welches dann für standardisierte Benchmarks verwendet wird, um verschiedene Modellüberprüfungsprogramme auf der Wort-Ebene miteinander zu vergleichen. Die Überprüfung auf der Wort-Ebene hat den Vorteil, dass bestimmte Ansätze von den zusätzlich zu Verfügung stehenden Informationen, auf dieser Ebene profitieren. Dies ist besonders bei SMT-Techniken der Fall.

Das BTOR Format ist ein quantifikatorfreies Format auf der Wort-Ebene für Formeln über Bit-Vektoren in Kombination mit eindimensionalen Arrays. Dies ermöglicht das Modellieren von SMT-Problemen über die BIT-Vektor Theorie und ist im Prinzip eine Verallgemeinerung des Aiger Formats auf der Wort-Ebene.

Die Weiterentwicklung zum BTOR2 Format wurde 2018 veröffentlicht. Das BTOR2 Format verallgemeinert und erweitert die ursprüngliche BTOR Version, welche wiederum als eine Verallgemeinerung des Aiger Formates angesehen wird. Somit folgt das Format den Designprinzipien des Aiger Formates, welche im vorherigen Kapitel erläutert wurden und folgt der SMT-Lib Semantik. Jedoch ist BTOR2 aufgrund von einigen Designveränderungen nicht abwärtskompatibel. Bei BTOR2 kann die Dateiendung ".btor" oder auch ".btor2" verwendet werden. Um welche Version es sich handelt entscheidet das jeweilige Modellüberprüfungsprogramm.

Wenn im Laufe dieser Arbeit BTOR erwähnt wird, so ist das neue BTOR Format gemeint.

Um der später folgenden Implementierung gut folgen zu können, werden anschließend die wichtigsten Punkte der BTOR2 Syntax beschrieben. Dies hat den Grund, da das zu entwickelte Programm eine BTOR2 Datei modifiziert.
Abbildung 3 zeigt die Btor2 Syntax und bildet mit den unterstützen Operatoren (Abbildung 4) das BTOR2-Format. Das BTOR Format ist wie das Aiger Format zeilenbasiert und kann dank der überschaubaren und einer sehr intuitiven Bezeichnung, auch leicht vom Menschen gelesen werden. Die in Abbildung 3 stehende Abkürzung <sid> steht für den Sort-Bezeichner (englisch sort identifiers) und repräsentiert die Zeilennummer, in der ein beliebiger Bit Vektor oder ein sortiertes Array definiert wird. Die Abkürzung <nid> ist die Abkürzung für Knotenpunkt-Bezeichner (englisch node identifiers) welche die Zeilennummer repräsentieren, in der ein entsprechender Knotenpunkt (englisch nodes) angelegt wird. Die Syntax, für das Programm,

welches ich im Laufe der Arbeit erläutern werde und folglich notwendig ist, ist recht überschaubar und soll hier kurz aufgelistet werden.

<const> : Legt eine Konstante an. Die Angabe erfolgt in der Binärdarstellung.

<state> : Legt eine Zustandsvariable an, die das Spezifizieren von Registern und Speicher ermöglicht. Vor der BTOR2 Erweiterung waren Register immer mit null initialisiert und der Speicher war standardmäßig gar nicht initialisiert.

<bitvec> : Legt einen Bit-Vektor an.

<constraint> : Legt eine unveränderbare und global geltende Beschränkung fest.

<bad> : Legt eine "Bad State" Eigenschaft fest, welche an sich die Negation einer "Safety" Eigenschaft darstellt.

Sowie die Logischen Operationen Gleichheit (eq), Ungleichheit (neq), Konjunktion (and) sowie Disjunktion (or). Somit werden in dieser Arbeit werden nur grundlegende Funktionen des BTOR2 Formats benötigt.

```
⟨num⟩       ::= positive unsigned integer (greater than zero)
⟨uint⟩      ::= unsigned integer (including zero)
⟨string⟩    ::= sequence of whitespace and printable characters without '\n'
⟨symbol⟩    ::= sequence of printable characters without '\n'
⟨comment⟩   ::= ';' ⟨string⟩
⟨nid⟩       ::= ⟨num⟩
⟨sid⟩       ::= ⟨num⟩
⟨const⟩     ::= 'const' ⟨sid⟩ [0-1]+
⟨constd⟩    ::= 'constd' ⟨sid⟩ ['-']⟨uint⟩
⟨consth⟩    ::= 'consth' ⟨sid⟩ [0-9a-fA-F]+
⟨input⟩     ::= ( 'input' | 'one' | 'ones' | 'zero' ) ⟨sid⟩ | ⟨const⟩ | ⟨constd⟩ | ⟨consth⟩
⟨state⟩     ::= 'state' ⟨sid⟩
⟨bitvec⟩    ::= 'bitvec' ⟨num⟩
⟨array⟩     ::= 'array' ⟨sid⟩ ⟨sid⟩
⟨node⟩      ::= ⟨sid⟩ 'sort' ( ⟨array⟩ | ⟨bitvec⟩ )
              | ⟨nid⟩ ( ⟨input⟩ | ⟨state⟩ )
              | ⟨nid⟩ ⟨opidx⟩ ⟨sid⟩ ⟨nid⟩ ⟨uint⟩ [⟨uint⟩]
              | ⟨nid⟩ ⟨op⟩ ⟨sid⟩ ⟨nid⟩ [⟨nid⟩ [⟨nid⟩]]
              | ⟨nid⟩ ( 'init' | 'next' ) ⟨sid⟩ ⟨nid⟩ ⟨nid⟩
              | ⟨nid⟩ ( 'bad' | 'constraint' | 'fair' | 'output' ) ⟨nid⟩
              | ⟨nid⟩ 'justice' ⟨num⟩ ( ⟨nid⟩ )+
⟨line⟩      ::= ⟨comment⟩ | ⟨node⟩ [⟨symbol⟩] [⟨comment⟩]
⟨btor⟩      ::= ( ⟨line⟩'\n' )+
```

Abbildung 3: *BTOR2 Syntax [8]*

indexed		
[su]ext w	(un)signed extension	$\mathcal{B}^n \rightarrow \mathcal{B}^{n+w}$
slice u l	extraction, $n > u \geq l$	$\mathcal{B}^n \rightarrow \mathcal{B}^{u-l+1}$
unary		
not	bit-wise	$\mathcal{B}^n \rightarrow \mathcal{B}^n$
inc, dec, neg	arithmetic	$\mathcal{B}^n \rightarrow \mathcal{B}^n$
redand, redor, redxor	reduction	$\mathcal{B}^n \rightarrow \mathcal{B}^1$
binary		
iff, implies	Boolean	$\mathcal{B}^1 \times \mathcal{B}^1 \rightarrow \mathcal{B}^1$
eq, neq	(dis)equality	$\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{B}^1$
[su]gt, [su]gte, [su]lt, [su]lte	(un)signed inequality	$\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^1$
and, nand, nor, or, xnor, xor	bit-wise	$\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^n$
rol, ror, sll, sra, srl	rotate, shift	$\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^n$
add, mul, [su]div, smod, [su]rem, sub	arithmetic	$\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^n$
[su]addo, [su]divo, [su]mulo, [su]subo	overflow	$\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^1$
concat	concatenation	$\mathcal{B}^n \times \mathcal{B}^m \rightarrow \mathcal{B}^{n+m}$
read	array read	$\mathcal{A}^{\mathcal{I}\rightarrow\mathcal{E}} \times \mathcal{I} \rightarrow \mathcal{E}$
ternary		
ite	conditional	$\mathcal{B}^1 \times \mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^n$
write	array write	$\mathcal{A}^{\mathcal{I}\rightarrow\mathcal{E}} \times \mathcal{I} \times \mathcal{E} \rightarrow \mathcal{A}^{\mathcal{I}\rightarrow\mathcal{E}}$

Abbildung 4: *BTOR2 Operationen [8]*

2.3 Yosys und Verilog

Verilog zählt zusammen mit VHDL (englisch Very High Speed Integrated Circuit Hardware Description Language) zu den meistgenutzten Hardware-Beschreibungssprachen und ist standardisiert als IEEE 1364-2005. Mit Verilog ist es möglich Hardware auf einer höheren Abstraktionsebene zu beschreiben. Durch die abstrakte Darstellung werden Informationen wie Details zur Implementierung oder der Technologie nicht beachtet. Verilog interessiert sich somit hauptsächlich für den Ablauf und den Verhaltenseigenschaften [3].

Verilog war zu Beginn eine reine Simulationssprache und wurde deshalb nur zur Verifikation durch Simulation verwendet. Die Funktionalität zur Synthese wurde in Verilog erst später hinzugefügt. Diese wurde dann im weiteren Verlauf zu einer Hauptfunktion von Verilog. Somit ist Verilog als eine Hardwarebeschreibungssprache bekannt, die Hardware beschreibt, simuliert, synthetisiert und verifiziert. Mit Verilog lassen sich zwar so gut wie fast alle Entwürfe simulieren, jedoch nicht immer synthetisieren [4].

Um Verilog Dateien in das BTOR2 Format umzuwandeln kann man Yosys verwenden. Yosys ist ein Open Source-Framework für RTL-Synthesewerkzeuge und verfügt über eine umfangreiche Verilog-2005 Unterstützung. Darüber hinaus bietet Yosys einen Basissatz von Synthesealgorithmen für verschiedene Anwendungsbereiche an.

Dank Yosys kann man Hardware in Verilog beschreiben und im BTOR Format verifizieren. Durch Yosys entstanden so gut wie alle Test Dateien, welche in dieser Arbeit verwendet werden. Dies erkennt man daran, dass Yosys einen Kommentar am beginn einer erzeugten BTOR Datei anlegt.

2.3.1 Formate im Vergleich

Da nun alle Informationen über die in dieser Arbeit vorkommenden Formate wiedergegeben wurden, soll die Abbildung 5 und 6 einen Vergleich der einzelnen Formate darstellen. Hierbei wird in Abbildung 5 ein einfacher Zähler in der Beschreibungssprache Verilog beschrieben. Dieser zählt von null bis drei hoch, setzt sich anschließend auf null zurück und zählt wieder von vorne. Eine Bad State wird erreicht wenn der Zähler größer als drei wird. Wandelt man diese Beschreibung mithilfe von Yosys in eine BTOR Datei um, so entsteht die Beschreibung auf der rechten Seite von Abbildung 6. Die entsprechende Aiger Beschreibung, welche auf der linken Seite dargestellt wird, erhält man in dem man die BTOR Datei mithilfe des btor2Aiger Tools umwandelt.

simpleCounter.sv

```
module count_up_to_three (
  input clk,
        output [3:0] counter
);

        reg [3:0] current_state = 4'b0;
        reg [3:0] next_state;

  always @ (posedge clk) begin
    current_state <= next_state;
  end

  always @* begin
         if(current_state == 3 )
              next_state <= 4'd0;
        else
              next_state <= current_state + 4'd1;
  end

   assert property (~(current_state > 3));

  assign counter = current_state;
endmodule
```

Abbildung 5: *Simple Counter: Verilog*

simpleCounterAig.aig

```
aag 21 1 4 0 15 1
4
12 38
10 40
8 42
6 7
33
32 11 13
34 8 32
36 6 34
14 6 8
20 10 14
28 13 21
26 12 20
30 27 29
38 30 37
22 11 15
24 21 23
40 24 37
16 7 9
18 15 17
42 18 37
i0 clk
l0 current_state[3]
l1 current_state[2]
l2 current_state[1]
l3 current_state[0]
```

simpleCounter.btor

```
; BTOR description generated by Yosys 0.10+60 (git sha1 5f00b
1 sort bitvec 1
2 input 1 clk ; simpleCounter.sv:2.9-2.12
3 sort bitvec 4
4 const 3 0000
5 state 3 current_state
6 init 3 5 4
7 output 5 counter ; simpleCounter.sv:3.22-3.29
8 sort bitvec 2
9 const 8 11
10 uext 3 9 2
11 ugt 1 5 10
12 not 1 11
13 const 1 1
14 not 1 12
15 and 1 13 14
16 bad 15 simpleCounter.sv:18.6-20.42
17 uext 3 13 3
18 add 3 5 17
19 uext 3 9 2
20 eq 1 5 19
21 ite 3 20 4 18
22 uext 3 21 0 next_state ; simpleCounter.sv:7.19-7.29
23 next 3 5 21
; end of yosys output
```

Abbildung 6: *Simple Counter: Aiger und BTOR*

2.4 HWMCC

Der bereits erwähnte BDD-Ansatz wurde bereits Anfang der 90er Jahre in die Qualitätssicherungsprozesse in Hardwarefirmen eingeführt. Der Ansatz der begrenzten Modellprüfung wurde dann im Jahre 1999 eingeführt.[11] Dies zeigt wie einige alte Ansätze heute noch verwendet werden. Im Laufe des industriellen Wandels wurde die Hardware komplexer und die Qualitätsansprüche immer höher.

Die Computer-Aided Verification (CAV) ist eine jährlich stattfindende Konferenz, welche sich mit theoretischen und praktischen Themen beschäftigen, welche die formale Analyse von Software und Hardwaresystemen in den Fokus setzten. Zu den wichtigsten Ergebnissen, die ursprünglich auf der CAV veröffentlicht wurden, gehören bahnbrechende Techniken der Modellüberprüfung wie Counterexample-Guided Abstraction Refinement (CEGAR) und partielle Ordnungsreduktion.[12]

Aus der CAV 2007 entstand ein Hardware-Modellüberprüfungswettbewerb (englisch Hardware Model Checking Competition abkz. HWMCC). Das Ziel dieses Wettbewerbes sowie der darauffolgenden zehn HWMCC´s war es die Weiterentwicklung der Modellüberprüfung. Hierbei stehen die Kernalgorithmen im Fokus.

3 Projekt: Btor2Tool

Das Ziel dieser Arbeit ist es herauszufinden, ob die Modellüberprüfung auf der Wort-Ebene durch zusätzliche Informationen von der Bit-Ebene verstärkt werden kann. Um dies herauszufinden, wird in dieser Arbeit ein Programm entwickelt, mit welchem man eine BTOR2 Datei einlesen kann und diese mit Informationen über Bit Zustände erweitert. Die Bit Informationen entstehen aus einer Bit-Ebenen Analyse, basierend auf dem Aiger Format. Hierbei steht BTOR2 stellvertretend für die Modellüberprüfung auf der Wort-Ebene, sowie die Informationen aus der Aiger Bit-Ebenen Analyse für die Bit-Ebene. Somit basiert das Ergebnis auf diesen beiden Formaten.

Diese beiden Formate eigenen sich gut, um eine allgemeine Aussage treffen zu können, da wie im Grundlagenkapitel bereits ausführlich beschrieben, sie als Formate ausgedacht wurden, um möglichst viele Modellüberprüfungsprogramme miteinander zu vergleichen.

Der Grundgedanke ist es, die Modellüberprüfung auf einer hohen Abstraktionsebene zu verbessern. Deshalb konzentriert sich diese Arbeit nicht auf konkreten Verbesserungen von Algorithmen, was die ursprüngliche Intension der HMWWC ist, sondern es soll versucht werden, die Beschreibung mit zusätzlichen Informationen zu ergänzen. So soll eine Möglichkeit geschaffen werden, um die Beschreibung im BTOR2 Format unabhängig vom verwendeten Modellüberprüfungsprogramm zu optimieren.

Somit wird dann der eigentliche Überprüfungsprozess in dieser Arbeit als eine “Black Box” betrachtet. Das Programm, welches im Laufe dieser Arbeit entwickelt wurde, ist in der Programmiersprache Haskell geschrieben. Dieses bekommt als Eingabe eine BTOR-Datei und Bit Informationen und fügt dann die Informationen über die einzelnen Bits mithilfe eines internen Mappings der BTOR-Datei hinzu.

Das Programm lässt sich in zwei Arbeitsschritte unterteilen: zuerst werden BTOR-Datei und die Bit-Informationen eingelesen und mithilfe einer internen Darstellung repräsentiert. Anschließend werden aus dieser Darstellung die Bit-Informationen in eine äquivalente Darstellung ins BTOR-Format umgewandelt und hinzugefügt.

Der gesamte Programmcode, sowie die Skripte für die Auswertung, befinden sich im Projekt Repository [23]. Hier kann die genaue Implementierung nachgesehen werden, was begründet, warum die Implementierungsschritte in der Arbeit selbst, nicht bis ins letzte Detail aufgeführt werden.

Die Grafiken 7 und 8 zeigen den Arbeitsablauf der Modellüberprüfungsprogramme BTORMC[18] und des AVR-Model Checker[17]. Diese sollen verdeutlichen, wo das entstehende Programm integriert werden kann(rote Kreise) und soll die Unabhängigkeit mit dem eigentlichen Prüfungsverfahren darstellen.

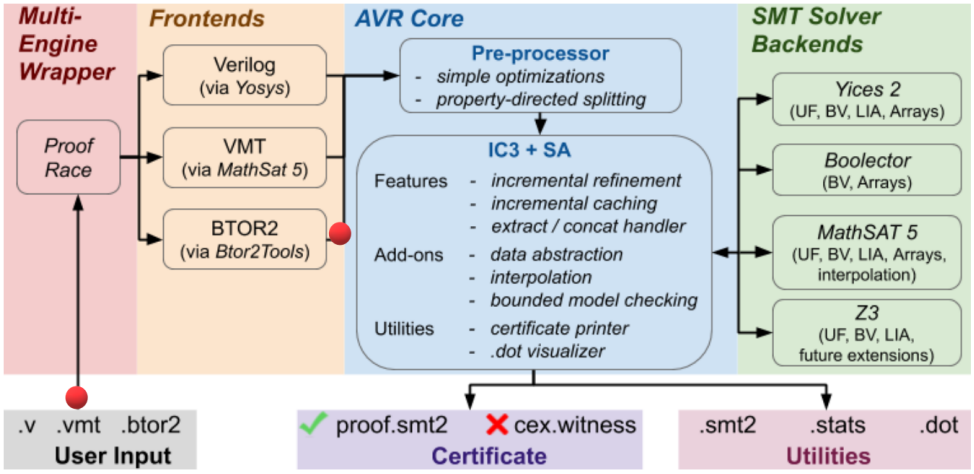


Abbildung 7: *AVR - Ablauf [18]*

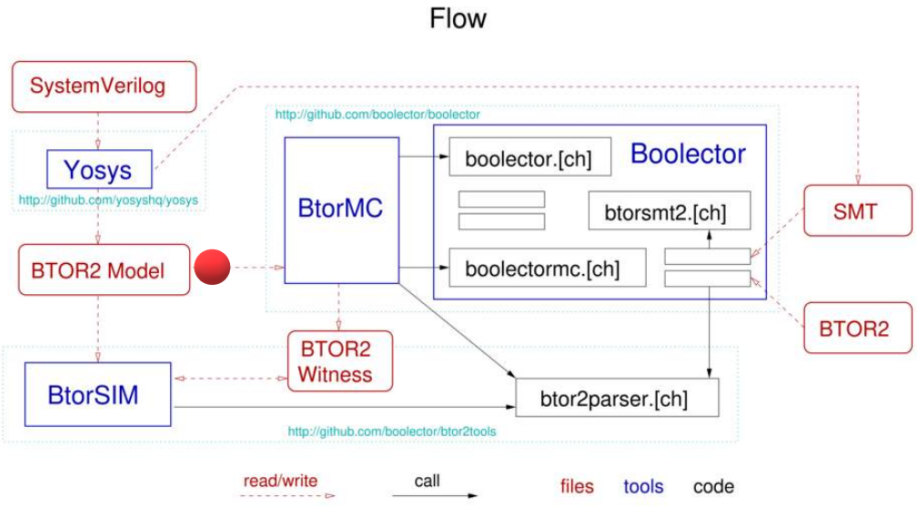


Abbildung 8: *Boolector-Ablauf (BTORMC ist ein Teil des Boolector)[10]*

3.1 Programmablauf und Implementierung

In dem folgenden Unterkapitel soll der gesamte Programmablauf dargestellt werden und es wird auf die wichtigsten Implementierungsschritte genauer eingegangen. Eine gesamte Übersicht wird in der Abbildung 9 grafisch dargestellt.

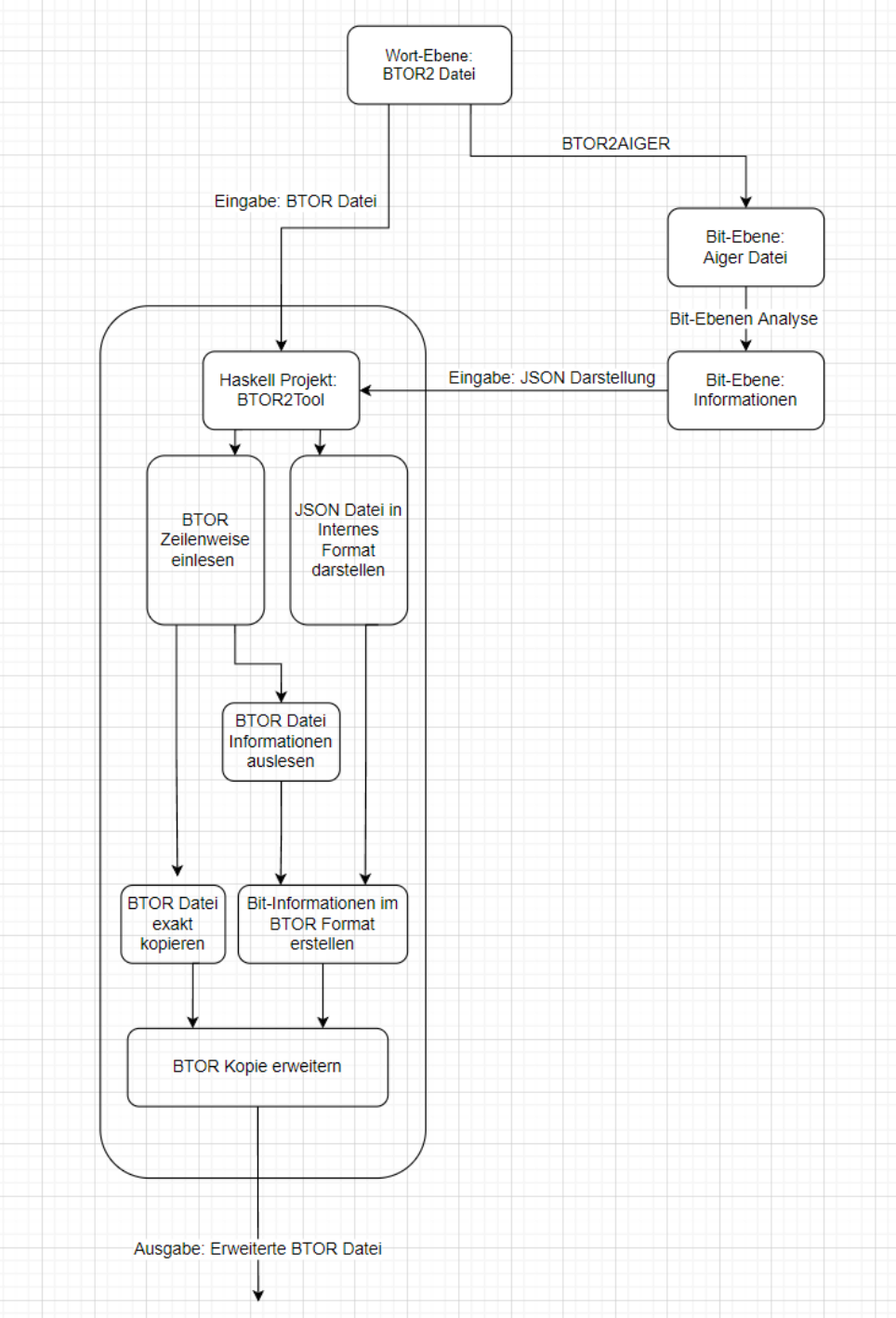


Abbildung 9: *Programmübersicht*

3.2 Eingabe Format

Um umfangreiche Bit-Ebenen Informationen möglichst einfach und effizient darzustellen, wird ein neues Eingabeformat benötigt.

Durch ein solches Format ist man somit nicht an ein bestimmtes Analysetool gebunden. Denn dank des einfachen Formataufbaus, lassen sich Analyseergebnisse mit wenig Aufwand in das benötigte Format abbilden. Als Datenformat wird JSON (JavaScript Object Notation) verwendet, da es hierfür in vielen Programmiersprachen bereits Parser für dieses Format gibt.
Hierzu wurde das JSON Schema aus Abbildung 10 verwendet.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "listOfClause": {
      "type": "array",
      "items": [
        {
          "type": "object",
          "properties": {
            "listOfExpression": {
              "type": "array",
              "items": [
                {
                  "type": "object",
                  "properties": {
                    "bit": {
                      "type": "integer"
                    },
                    "negation": {
                      "type": "boolean"
                    },
                    "name": {
                      "type": "string"
                    }
                  },
                  "required": [
                    "bit",
                    "negation",
                    "name"
                  ]
                } ] } },
            "required": [
              "listOfExpression"
            ]
          } ] } },
  "required": [
    "listOfClause"
  ]
}
```

Abbildung 10: *JSON Schema für die Darstellung von Bit-Informationen*

Hierbei enthält eine JSON Eingabedatei ein “listOfClauses” Objekt, welches wiederum ein Array aus “listOfClause” Objekten enthält. Dieses Objekt hat wiederum ein Array aus “listOfExpression”. In diesen werden die einzelnen Bitinformationen gesammelt. Beinhaltet das Array zwei oder mehr Einträge, so werden diese als eine Oder Verknüpfung interpretiert. Durch dieses Format lassen Sich also Informationen über einzelne Bits oder eine Disjunktion möglicher Bitzustände darstellen.

Implementierung:
Die Datenstrukturdarstellung wurde genau so aufgebaut wie das JSON Schema. Dadurch konnte man die Data.Aeson Bibliothek [13] zum parsen verwenden und musste für die interne Darstellung keine Funktionen selbst umsetzen, sondern es konnten die generischen Funktionen genutzt werden.

```
import Data.Aeson ( FromJSON, ToJSON )
import GHC.Generics ( Generic )

data Clauses
    = Clauses {listOfClause :: [Clause]}
      deriving (Show, Eq, Generic, ToJSON, FromJSON)

data Clause
  = Clause {listOfExpression :: [Expression]}
    deriving (Show, Eq, Generic, ToJSON, FromJSON)

data Expression    =
    Expression {
        negation :: Bool
        ,name :: String
        ,bit :: Int
    }   deriving (Show, Eq, Generic, ToJSON, FromJSON)
```

Abbildung 11: *Interne Darstellung*

3.3 Vorarbeit: Bit Ebenen Analyse

Erläuterung:
Die Vorarbeit umfasst die Umwandlung der BTOR2 Datei in eine Aiger Datei und dessen Analyse. Die Analyse ist nicht Teil dieser Arbeit, da es hierfür bereits umfangreiche Analyseprogrammlösungen gibt. Die Ergebnisse müssen dann in ein spezielles JSON Format umgewandelt werden. Dies bildet dann neben der ursprünglichen BTOR Datei die Eingabeparameter, welche für das Haskell Programm notwendig sind.

Zur Entwicklung dieses Programmes wurden die Analyseprogramme m2c und reduce_aiger verwendet. Jedoch ist das Programm m2c nicht uneingeschränkt zuverlässig. Es kann aufgrund unbekannter Randbedingungen zu falschen Analyseergebnissen kommen. Dies ist beim explorativen Testen im Laufe des Entwicklungsprozesses nicht aufgetreten. Jedoch um die Korrektheit der Ergebnisse zu gewährleisten, wird zur Analyse nur das Programm reduce_aiger verwendet.

Da das gesamte Programm und auch das Formatdesign auf den oben erwähnten Analyseprogrammen basieren, besitzt das Programm, die Funktion die Ergebnisse dieser beiden Programme direkt einzulesen. Für das Umwandeln in eine Aiger Datei gibt es das Tool btor2aiger [19] welches ein Teil des Boolector Projektes ist.

Zwischeninformation:
Die Analyseprogramme reduce_aiger und m2c wurden von Prof. Dr. Matthias Güdemann [20] entwickelt und für diese Arbeit zur Verfügung gestellt. Hierbei basiert m2c auf einen IC3 Algorithmus. Dieses Programm verstärkt Eigenschaften so weit, dass diese sich mit der 1-Induktion beweisen lassen und die verstärkten Klauseln ausgibt. Reduce_aiger nutzt hingegen die Saturationsmethode von Gunnar Stålmarck um Äquivalenzen von Bitzuständen auszugeben.

Implementierung: Einlesen von reduce_aiger und m2c Lösungen
Der Einstiegspunkt für die Einlesefunktion von reduce_aiger oder m2c Ausgaben ist in der Main Methode. Hierbei werden in den Funktionen “readReduceAigerSolution” und “readM2cSolution” relevanten Informationen Zeilenweise ausgelesen und in der internen Darstellung dargestellt.
Da die beiden erwähnten Programme die Ausgabe jeweils zeilenweise erstellt, kann man die Dateien zeilenweise einlesen, Zeilen mit relevanten Informationen in die interne Darstellung umwandeln und anschließend mit der “endcode” Funktion, auch bekannt unter toJson, in ein ByteString umwandeln. Diesen kann man dann mithilfe der System.IO.hPutStr Funktion in die JSON Datei schreiben.

```
main = do
    args <- getArgs
    if length args == 1 then do
        let arg = head args
        if arg == "h" || arg == "-h" || arg == "help" || arg=="-help" then do printHelp
        else printError
    else do
        let opt = readOption (args) -- Überprüft die übergebenen Optionen
        if getError opt == True then printError
            else do
            if getSolutionType opt == "m" then do
                --JSON Datei wird aus der übergebenen m2c Ausgabe erstellt.
                readM2cSolution (getSolutionIn opt) (getJsonOut opt)
                --Die BTOR Datei wird eingelesen und mit den Informationen aus der zuvor erstellten JSON Datei ergänzt.
                readBtor (getBtorIn opt) (getJsonOut opt) (getBtorOut opt) (getPropertyFlag opt) (getPropertyOptionFlag opt)
                print "generate json from m2c-solution" --Status Ausage
            else if getSolutionType opt == "r" then do
                --JSON Datei wird aus der übergebenen reduce_aiger Ausgabe erstellt.
                readReduceAigerSolution (getSolutionIn opt) (getJsonOut opt)
                --Die BTOR Datei wird eingelesen und mit den Informationen aus der zuvor erstellten JSON Datei ergänzt.
                readBtor (getBtorIn opt) (getJsonOut opt) (getBtorOut opt) (getPropertyFlag opt) (getPropertyOptionFlag opt)
                print "generate json from aiger_reduce-solution" --Status Ausage
            else do
                --Die BTOR Datei wird eingelesen und mit den Informationen aus der übergebenen Json Datei ergänzt.
                readBtor (getBtorIn opt) (getJsonIn opt) (getBtorOut opt) (getPropertyFlag opt) (getPropertyOptionFlag opt)
```

Abbildung 12: *Main Funktion*

Hierbei ist es aufgrund des Haskell Lazy Prinzips wichtig, das große Dateien zeilenweise in die JSON Datei geschrieben werden, da es ansonsten vorkommen kann, dass das Schreiben der JSON Datei einen nicht mehr rentablen Zeitraum in Anspruch nimmt.

Der mögliche Größenanstieg einer BTOR Datei soll mithilfe des Beispiels ridecore.btor gezeigt werden. Diese Datei stammt aus dem HWMCC2020 Archiv[21]
Die Ursprungs Datei ist insgesamt 62.459 Zeilen lang. Die Ausgabe von reduce_aiger umfasst insgesamt 709.688 Zeilen und die JSON Datei, die man nach dem Einlesen in Haskell erhält, beinhaltet Informationen über 65.642 Bits. Wenn diese Informationen in die Ursprungsdatei ergänzt werden, wächst diese auf 259.565 Zeilen an. Zu sehen ist dies im Auswertungsverzeichnis auf GitLab [23]

3.4 Eingaben einlesen

3.4.1 JSON einlesen

Da die JSON Datei unabhängig von Zeilenumbrüchen formatiert werden kann, wird diese Datei vollständig eingelesen und in das interne Dateiformat dargestellt. Bei der Implementierung gibt es im Vergleich zu der JSON Datei, welche aus den Lösungen generiert werden, keine wesentlichen Unterschiede außer, dass zu Beginn die JSON Datei einmal mit der Funktion decode (FromJson) in das interne Format dargestellt wird. Dies soll jedoch nur die korrekte Form der JSON Datei überprüfen.

3.4.2 BTOR einlesen

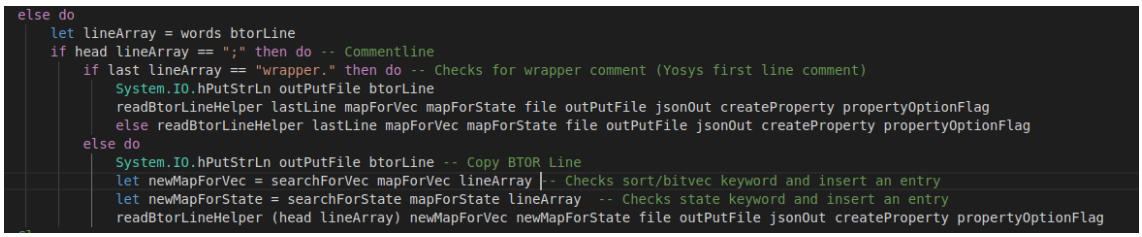
Die BTOR Datei wird zeilenweise eingelesen, die wichtige Informationen werden herausgefiltert und als Mapping zu Verfügung gestellt. Somit kommt man an Informationen, welche später eventuell benötigt werden, um die Bit-Informationen als Constraints abzubilden.

Gleichzeitig wird die jeweilige Zeile exakt kopiert und in eine neue Datei hinzugefügt. Diese wird dann später verwendet, um die Constraints hinzuzufügen. Die Syntaxbasis wurde bereits im Kapitel 2.2.2 umfangreich erläutert. Nun geht es darum welche Informationen in einer BTOR Datei benötigt werden, um später alle benötigten Schritte zum Erweitern mithilfe von Constraints umzusetzen. Die Bit Ebenen Analyse enthält optimalerweise Informationen über ein bzw. mehrere Bits von einer oder mehreren Zustandsvariablen (keyword states) und deshalb werden alle Zustandsvariableninitialisierungen beim Einlesen gespeichert. Zusätzlich sind die Informationen der einzelnen Sort-Bezeichner und deren Größe notwendig.

Somit erhalten wir dann die folgenden zwei benötigten Mappings:
Zustandsvariable **:** (Zustandsvariable Zeilennummer, Zeilen der Typendeklarierung)
Zeilen der Typendeklarierung **:** Größe in Bits

Eine ausführliche Darstellung erfolgt im Abschnitt 3.5.2

Implementierung Mapping:
Das Mapping wird während des einlesen der BTOR Datei erstellt. Hierbei wird zuerst überprüft ob es sich um eine Kommentarzeilen handelt, solche Kommentarzeilen werden meist von Yosys in der ersten und letzten Zeile angelegt. Anschließend wird in den Methoden searchForVec und searchForState die Zeile auf bestimmte Schlüsselwörter überprüft und gegebenenfalls ein neuer Eintrag in die Map hinzugefügt. Wird kein Schlüsselwort gefunden so wird die alte map unverändert zurückgegeben. Der entsprechenden Code Ausschnitt ist in Abbildung 13 dargestellt.



```
else do
    let lineArray = words btorLine
    if head lineArray == ";" then do -- Commentline
        if last lineArray == "wrapper." then do -- Checks for wrapper comment (Yosys first line comment)
            System.IO.hPutStrLn outPutFile btorLine
            readBtorLineHelper lastLine mapForVec mapForState file outPutFile jsonOut createProperty propertyOptionFlag
            else readBtorLineHelper lastLine mapForVec mapForState file outPutFile jsonOut createProperty propertyOptionFlag
        else do
            System.IO.hPutStrLn outPutFile btorLine -- Copy BTOR Line
            let newMapForVec = searchForVec mapForVec lineArray -- Checks sort/bitvec keyword and insert an entry
            let newMapForState = searchForState mapForState lineArray  -- Checks state keyword and insert an entry
            readBtorLineHelper (head lineArray) newMapForVec newMapForState file outPutFile jsonOut createProperty propertyOptionFlag
```

Abbildung 13: *Mapping: Erstellung während des einlesen und kopieren der BTOR Datei*

3.5 Constraints

In diesem Kapitel soll der theoretische Ansatz erläutert werden, wie man Informationen über Bit Zustände im BTOR2 Format darstellt und in einer bereits vorhandenen BTOR Datei hinzufügen kann. Anschließend soll das Vorgehen an einem Beispiel praktisch gezeigt werden.

3.5.1 Allgemeine Vorgehen: (Schema)

Um das Umwandlungsschema darzustellen, wird angenommen, dass man die Information hat, das Bit Nummer “x” von der Zustandsvariable “P” immer gesetzt ist. Für den Umwandlungsschritt muss zuerst eine Konstante “K” und ein Nullvektor “N” mit derselben Größe wie “P” angelegt werden. Bei der Konstante “K” sind alle Bits bis auf Bit “x” null. Anschließend bildet man die Konjunktion aus Zustandsvariable “P” und Konstante “K”.

Nun kann die Aussage getroffen werden, dass die soeben gebildete Konjunktion nicht mit dem Nullvektor “N” identisch ist, da Bit “x” gesetzt wurde.

Daraus folgt, dass man eine Aussage zur Äquivalenz treffen kann. Diese kann dann zur Erstellung von Constraints verwendet werden.
Dieser Vorrang wird wie folgt dargestellt:
$\backsim$ ((“P” && “K”) $\leftrightarrow$ “N”)

Hätten wir stattdessen die Information, dass Bit “x” nicht gesetzt ist, dann würde demzufolge nur die Negation wegfallen.
(“P” && “K”) $\leftrightarrow$ “N” gilt dann folglich, dass “N” gleich der Konjunktion aus “P” und “K” ist.

3.5.2 Konkretes Beispiel

In diesem Abschnitt wird die Umsetzung des allgemeinen Ansatzes vorherigen Abschnitt in das BTOR Format beschrieben.

Hierbei werden die Informationen aus zwei verschiedenen Quellen benötigt. Die Informationen über das gesetzte Bit bzw. nicht gesetzte Bit. Diese allein reichen nicht zum Anlegen von Constraints aus. Denn hierzu werden noch die Informationen aus der BTOR Datei benötigt, welche wie im Abschnitt 3.4.2 ausgelesen wurden und per Mapping zu Verfügung stehen. Dieses Mapping wird benötigt, um die Konstante und den Nullvektor anzulegen, da man die Größe der jeweiligen Zustandsvariablen benötigt, sowie die Zeile, in der Diese selbst angelegt wurde.

Um die einzelnen Schritte zu visualisieren, wird ein konkretes Beispiel verwendet. Hierfür wird als Beispiel die analog estimation_convergence [24] BTOR Datei verwendet. Mithilfe welcher nun gezeigt wird wie einen Bit-Ebenen Information im BTOR Datei darstellen kann.
Gegeben:
Informationen aus der Bit-Analyse:

Abbildung 14: *JSON Darstellung der Bit-Information*

Diese beinhaltet die Information, welche besagt, das Bitnummer 1 der Zustandsvariable "state_counter", immer gesetzt ist.
Informationen aus der BTOR Datei:

Abbildung 15: *Mapping: Zustandsvariable auf die Zeilennummer der Deklarierung und die Zeilennummer in die der Typen angelegt wurde.*

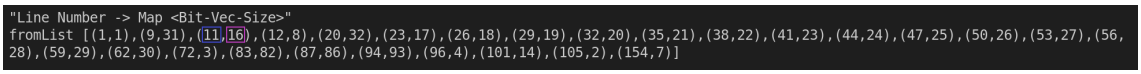


Abbildung 16: *Mapping: Zeilennummer der Typen Deklarierung auf die Größe des Bit-Vektors*

Das Mapping zeigt an welche Zustandsvariable in welcher Zeile angelegt wurde und in welcher Zeile der Sort-Bezeichner(englisch sort identifiers <sid>) mit der entsprechenden Bit Vektor Größe angelegt wird.
Diese Informationen wurden aus den folgenden Zeilen ausgelesen:

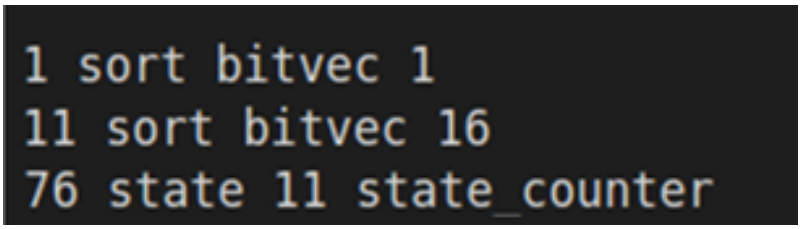


Abbildung 17: *Zeilen aus denen die BTOR Information ausgelesen wurden*

Nun wird das Schema von 3.5.1 umgesetzt. Hierfür wird die Syntax benötigt, mit der man Konstanten und Constraints anlegen kann. Sowie die Operationen eq, neq, and und or. Diese wurden im Kapitel 2.2.2 bereits erläutert.

Konkrete Schemaumsetzung:
Dazu wird am Ende der Datei eine neue Zeile angelegt. Das ist in diesem Beispiel die Zeile 177, in welcher eine 16 Bit Konstante angelegt wird, bei der das erste Bit gesetzt wird. Anschließend wird in Zeile 178 ein 16 Bit großer Nullvektor ebenfalls als Konstante angelegt. Nun muss die Konjunktion aus der Zustandsvariablen, welche in Zeile 76 initialisiert wurde und der angelegten Konstante aus Zeile 177 bilden. Nun muss diese Konjunktion mit dem Nullvektor auf Ungleichheit geprüft werden. Diese Ungleichheit wird anschließend als Constraint angelegt. In der Abbildung 18 kann gesehen werden, wie das Beispiel im BTOR2 Format dargestellt ist.

```
177 const 11 0000000000000001
178 const 11 0000000000000000
179 and 11 76 177
180 neq 1 179 178
181 constraint 180
```

Abbildung 18: *Neue Zeilen, welche in die BTOR Datei hinzugefügt wurde*

3.6 Constraints verifizieren

Das Vorgehen, welches im Punkt 3.5 beschrieben wurde um Constraints hinzuzufügen, funktioniert selbstverständlich nur wenn die dazu benötigten Informationen auch korrekt sind.

Dafür wurde eine Option eingebaut, mit welcher man die hinzugefügten Constrains auf der Wort-Ebene überprüfen kann. Der Ansatz hierzu ist dem eigentlichen Constrains Ansatzes sehr ähnlich, da die Darstellung der Bit Informationen dieselbe ist. Die Idee ist es die Negation der Information, anstatt einen Constrains, als Bad State festzulegen. Anschließend kann mithilfe eines Modellüberprüfungsprogramm gezeigt werden, dass der Bad State nicht erreichbar ist. Folglich lässt sich dann die Aussage treffen, dass die Informationen aus der Bit-Ebenen Analyse korrekt sind. Zur Vereinfachung werden zur Verifikation alle angelegten Bit Informationen als Disjunktion miteinander verknüpft und können somit als einen einzelnen Bad State dargestellt werden. In Abbildung 19 wird dargestellt, wie die Überprüfung der Beispiels-Bit-Information aus Abschnitt 3.5.2 im BTOR Format angelegt werden kann.

```
177 const 11 0000000000000001
178 const 11 0000000000000000
179 and 11 76 177
180 eq 1 179 178
181 bad 180
```

Abbildung 19: *Constraints verifizieren anhand der Beispieldatei aus Abschnitt 3.5.2*

4 Auswertung

Das Ziel dieses Kapitels ist das Generieren von Testdaten und wie man diese anschließend visualisiert darstellt, um im Anschluss eine Aussage über die ursprüngliche Ausgangsfrage "ob man die Verifikation durch das Verstärken der Wort-Ebene mithilfe von Informationen aus einer Bit-Ebenen Analyse verbessern kann" machen zu können.

4.1 Testdaten-Generierung

Voraussetzungen:
Um verwendbare Testdaten zu erhalten, braucht man zum einen valide BTOR Dateien und ein Modellüberprüfungsprogramm, mit welchen man die BTOR Dateien überprüft.

Für die Testdaten-Generierung werden die BTOR Dateien aus dem HWMCC 2020 verwendet. Dadurch gelangt man an eine große Anzahl von BTOR Dateien, welche explizit ausgesucht wurden für einen Model-Checker-Benchmark.
Darüber hinaus sind die BTOR Dateien schon in Arrays und BV (Bitvektoren) unterteilt. Dies ist deshalb wichtig, um sicher zu stellen, dass man nur BTOR Dateien mit Bitvektoren benutzt, da das verwendete Umwandlungstool btor2aiger aktuell noch keine Array Unterstützung besitzt und wir somit ungültige Auswertungen erhalten.

Als Model Checker wurde der AVR: Abstractly Verifying Reachability von Aman Goel und Karem Sakallah. [18] verwendet. Dieser errang im Jahre 2020 den Gesamtsieg beim HWMCC. Der Grund für die Auswahl war es das der AVR ein einfaches Ausgabenformat besitzt und darüber hinaus sehr gut optimiert ist. Durch die Verwendung eines sehr guten Modellüberprüfungsprogramms, soll erreicht werden, dass die Ergebnisse auch einen realistischen Bezug erhalten.

	gold	silver	bronze
avr	7	1	1
abc	2	1	1
nuxmv		5	2
pono		2	4
pdtrav			1

Abbildung 20: *HWMCC 2020 Ranking [22]*

Aufbau:
Die Testdaten werden mittels Bashskript generiert, anschließend mit einen Python Skript ausgewertet und visualisiert.

1. Die BTOR Datei wird mithilfe des btor2aiger Tools in eine Aiger Datei umgewandelt.

2. Die Aiger Datei wird mit dem Tool reduce_aiger analysiert und die Ergebnisse werden abspeichert.

3. Wenn es eine Lösung gibt, dann wird diese zusammen mit der BTOR Datei als Eingabeparameter, für das in Haskell geschriebene Erweiterungsprogramm verwendet.

4. Nun wird überprüft, ob die BTOR Datei erweitert wurde. Anschließend beginnt der eigentliche Überprüfungsprozess mittels AVR Aufruf. Da die Testdaten-Generierung eine Eingabe von 323 BTOR Dateien hat ist es wichtig den AVR direkt hintereinander auszuführen, um eine zeitabhängige Leistungsbeeinträchtigung des Systems zu verhindern und somit die Werte nicht zu verfälschen. Die Überprüfung ob die BTOR Datei erweitert wurde, ist wichtig da eine Überprüfung ansonsten nur Zeit kostet und keinen Mehrwert für diese Arbeit bietet.

5. Nun werden die Ergebnisse der beiden Ausführungen abgespeichert. Dazu wird ein Counter verwendet, um bei Fehlern die Verbindung zwischen der originalen BTOR Datei und der bearbeiteten BTOR Datei herstellen zu können.

4.2 Testdaten-Auswertung

In diesem Kapitel sollen die Testdaten ausgewertet und grafisch dargestellt werden. Anschließend sollen die erkennbaren Zusammenhänge erläutert werden. Das Ziel dieser Auswertung ist es, zu untersuchen, ob man durch zusätzliche Bit-Level Eigenschaften, die Verifikation auf der Wort-Ebene beeinflussen kann.

4.2.1 Testdaten-Auswertung: Bit-Ebenen Analyse

Daten Grundlage:
Insgesamt iteriert der Benchmark über 323 BTOR Dateien. Bei diesen generiert das Analyse Tool reduce_aiger bei insgesamt 251 eine Ausgabe. Von diesen enthalten 96 hilfreiche Informationen, mit denen man eine BTOR Datei erweitern kann.

Für die nachfolgende Rechenzeitanalyse für den Umwandlungs- und Erweiterungsprozess, werden nur diese verwendet, wo das Tool eine Ausgabe erzeugt hat. Dies hat den Grund, dass auch Rechenzeit entsteht, wenn eine Ausgabe eingelesen wird, obwohl es keine Informationen über die Zustandsvariablen gibt.

Datenauswertung:
In diesem Unterkapitel geht es darum den zusätzlichen Rechenzeitaufwand, welcher benötigt wurde, um eine BTOR Datei zu eine Aiger Datei umzuwandeln, zu analysieren und anschließend diese Ergebnisse in eine Kopie der Ursprungs BTOR Datei hinzufügt.

Hierfür wurden folgenden vier Zeiträume festgelegt:

- Zeitraum A: Ausführungsdauer bis 1 Sekunde
- Zeitraum B: Ausführungsdauer über 1 Sekunde und bis 2 Sekunden
- Zeitraum C: Ausführungsdauer über 2 Sekunden und bis 10 Sekunden
- Zeitraum D: Ausführungsdauer über 10 Sekunden

Die Ergebnisse werden in der Tabelle 1 dargestellt. Diese erhält man beim Einsortieren der Ergebnisse in die jeweilige Aufwandskategorie.

Tabelle 1: *Zeitaufwandsanalyse*

Kategorie	Anzahl	min	max	durchschnitt
Zeitraum A	179	0.04 s	1.0 s	0.35 s
Zeitraum B	38	1.01 s	1.74 s	1.43 s
Zeitraum C	31	2.03 s	9.91 s	6.61 s
Zeitraum D	3	10.43 s	151.84 s	58.89 s

Hierbei ist besonders die Verteilung der jeweiligen Kategorien interessant, da man bei einem Wettbewerb mit einem festgelegten Zeitlimit den zusätzlichen Zeitbedarf berücksichtigen muss. Deshalb wäre es schlecht, wenn allein der Vorbereitungsschritt im Sinne einer Bit-Ebenen Analyse mit anschließender BTOR Erweiterung, bereits einen großen Teil der verfügbaren Zeit in Anspruch nehmen würde. Das Verhältnis ist in der Abbildung 21 sehr gut dargestellt und man erkennt, dass die wirklich zeitkritischen Zeiträume der Kategorie D nur sehr selten vorkommen.

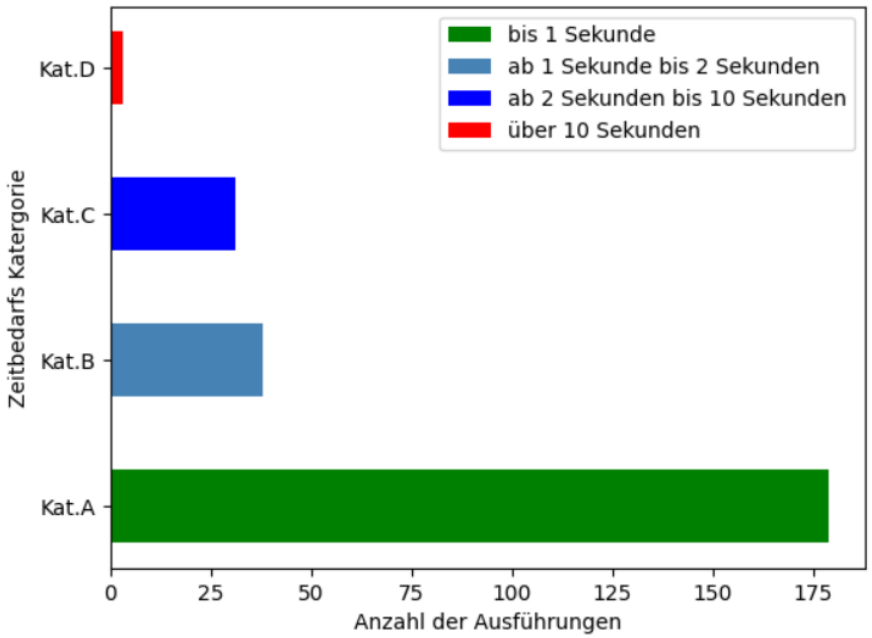


Abbildung 21: *Grafische Darstellung der Zeitaufwandsanalyse*

Erkenntnis:
Die Ergebnisse zeigen deutlich, dass sich der Analyse- und Erweiterungsprozess im Regelfall sehr performant ausführen lassen. Nur in sehr seltenen Fällen dauert es unter bestimmten Umständen sehr lange. Aufgrund der geringen Datenlage kann man jedoch keine Aussage über die Dauer des Sonderfalls treffen.

4.2.2 Auswertung: Gleichbleibende Ergebnisse

In dieser Teilauswertung soll der Zeit- und Speicherbedarf mit Ergebnissen des AVR-Modellüberprüfungsprogrammes verglichen werden, bei denen das Ergebnis gleich geblieben ist. Somit soll untersucht werden, ob die zusätzlichen Informationen aus der Bit-Ebenen Analyse Auswirkungen auf den Überprüfungsprozess haben.

Zeitbedarf:
Um den Zeitbedarf zu analysieren, wurden die Ergebnisse nach der Ausführungszeit aufsteigend sortiert und als Grafik dargestellt. In dieser wird dargestellt, wann ein Ergebnis gefunden wird. In diesem Fall werden zwei verschiedene Szenarien betrachtet.

Zum einen werden die Ergebnisse des Modellüberprüfungsprozesses der Ursprungs Dateien mit der erweiterten BTOR Dateien miteinander verglichen.

Zusätzlich soll die Ausführungsdauer, welche zum Erstellen der entsprechenden, erweiterten BTOR Datei benötigt wird, mit in den Vergleich aufgenommen werden. Dieser Vergleich soll die zeitliche Auswirkung der zusätzlichen Bit-Information auf das Modellüberprüfungsverfahren zeigen. Die unten dargestellten Abbildungen 22 und 23 zeigen die zeitliche Konsequenz, welche durch die zusätzlichen Bit Informationen entstanden sind. Hierbei ist bei beiden Untersuchungsszenarien ein eindeutiger, zeitlicher Vorsprung bei den erweiterten BTOR Dateien zu sehen.

Durch den Vergleich beider Abbildungen fällt direkt auf, dass sich der Überprüfungsprozess bei sehr kurzen Beispielen nicht besonders unterscheidet. Dies erkennt man in der Abbildung 22 an der starken Überlappung bis ca. 500 Sekunden.

Folglich lassen sich die Auswirkungen auf dem Graphen, bei dem der Erweiterungsprozess mit eingerechnet ist, schon erahnen. Kleine Beispiele profitieren von den zusätzlichen Informationen nicht. Wenn man die Zeit, welche für die Erweiterung benötigt wurde, in den Vergleich mit aufnimmt, haben kleinere Beispiele eine höhere Gesamtausführungsdauer. Diese Zeit wird jedoch bei Dateien mit längerer Überprüfungsdauer wieder ausgeglichen.

Zeitbedarf:
BTOR unverändert: 1394.34 Sekunden
BTOR erweitert (ohne Erweiterungsprozess): 1089.06 Sekunden
BTOR erweitert (mit Erweiterungsprozess): 1120.06 Sekunden

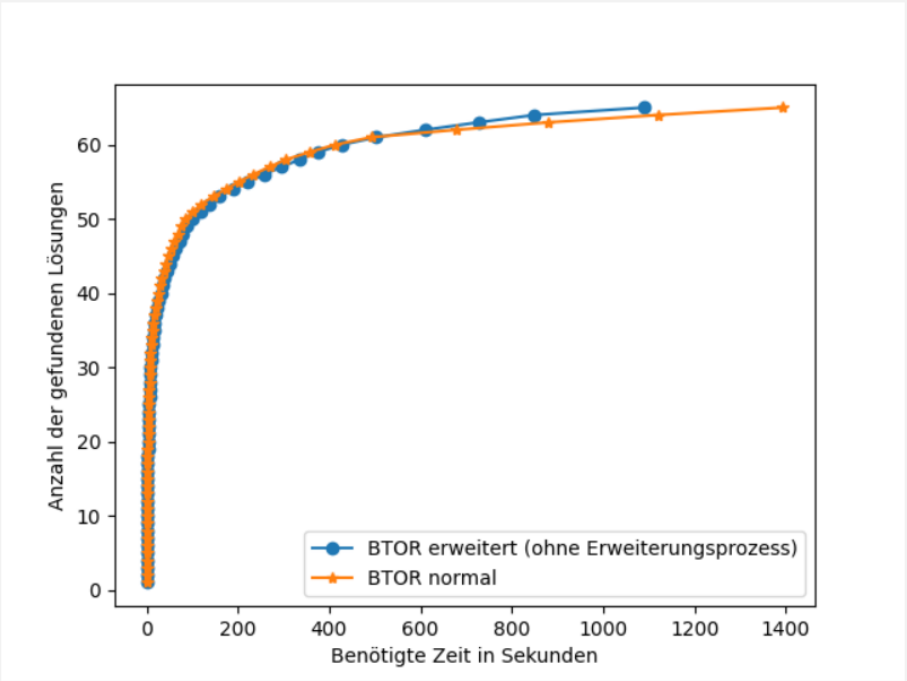


Abbildung 22: *Zeitliche Darstellung ohne Erweiterungsprozess*

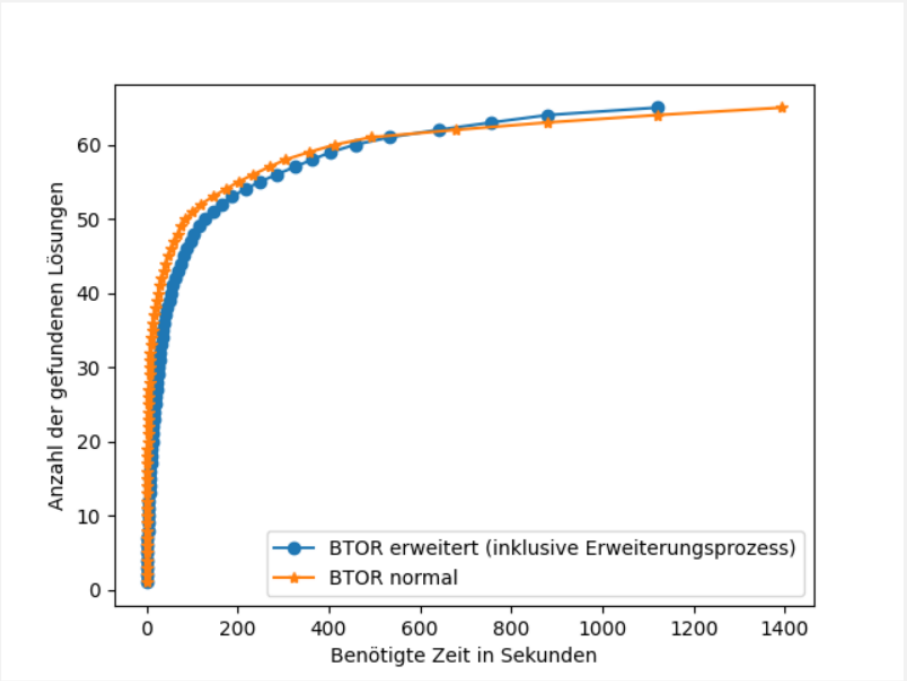


Abbildung 23: *Zeitliche Darstellung inklusive Erweiterungsprozess*

Erkenntnis:
Die Abbildungen 22 und 23, sowie die summierten Zeitwerte zeigen, dass wenn man nur den Überprüfungsprozess betrachtet, man zum größten Teil keine Veränderungen bei sehr kurzer Ausführungsdauer erkennt. Jedoch wird bei steigender Ausführungsdauer der positive Effekt der zusätzlichen Informationen sichtbar und man erkennt bei der absoluten Ausführungsdauer den Zeitvorsprung in diesem Benchmark.

Speicherbedarf:
In der heutigen Zeit spielt in der Regel der Zeitbedarf eine größere Rolle als der Speicherbedarf. Deshalb soll hier bloß das Ergebnis der Speicherbedarfsauswertung knapp dargestellt werden.
Bei dieser Auswertung wird nur der Speicherbedarf beim Überprüfungsprozess selbst verglichen. Der Speicherbedarf für den Erweiterungsprozess wird nicht explizit berücksichtigt.

Der Speicherbedarf wird in der Abbildung 24 nach der Ausführungsreihenfolge der einzelnen Dateien dargestellt.

Speicherbedarf:
BTOR unverändert: 9799.00 MB
BTOR erweitert (ohne Erweiterungsprozess): 8389.00 MB

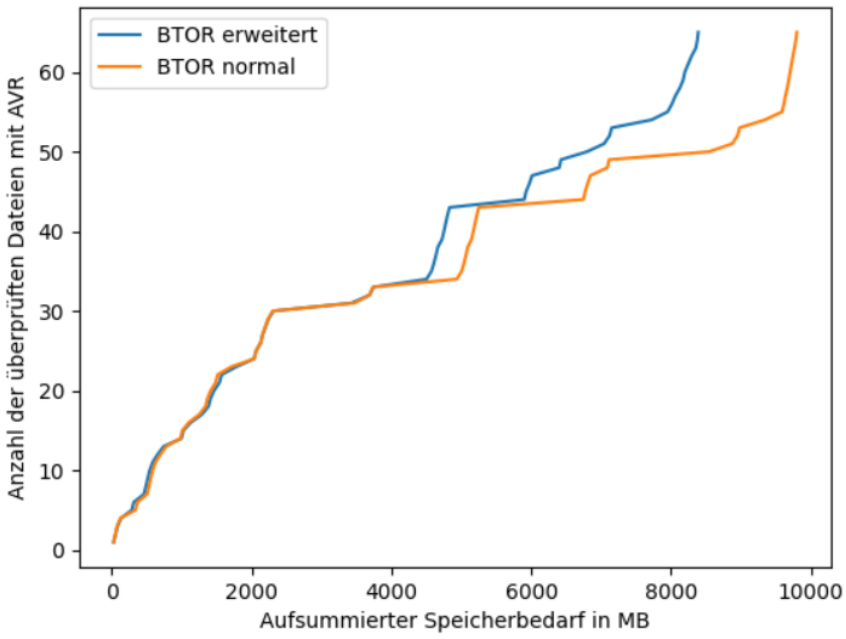


Abbildung 24: *Speicher Darstellung ohne Erweiterungsprozess*

Erkenntnis:
Trotz des am Anfang größeren Speicherbedarfs, welcher sich aus der vergrößerten BTOR Datei ergibt, wird zum Finden einer Lösung, insgesamt weniger Speicher benötigt.

4.2.3 Auswertung: Veränderte Ergebnisse

In diesem Unterkapitel sollen die Fälle dargestellt werden, in denen es zu einer Veränderung des Ergebnisses gekommen ist. Unter einem Ergebnis versteht man das Resultat, welches der AVR Modellprüfer ausgibt.

AVR stellt die Ergebnisse mithilfe der folgenden Abkürzungen dar:

- h: Eigenschaft gilt (safe)
- v: Eigenschaft verletzt (unsafe)
- f_to: Zeitlimit wurde erreicht
- f_mem: Speichergrenze wurde erreicht
- f_err: Fehler einer anderen Art ist aufgetreten

Tabelle 2: *AVR - Veränderte Ergebnisse*

Ergebnis vorher	Ergebnis nachher	Anzahl
f_to	h oder v	1
f_err	h oder v	1
h oder v	f_to	0
h oder v	f_err	0
h	v	0
v	h	0

Zusätzlich kam es bei der Ausführung von AVR zu internen Fehlern(Exceptions). Dies passierte während des gesamten Benchmarks 5-mal.

Da es für diese Fehler keine Muster gab, bzw. die Fehler bei dem erweiterten- und dem nicht erweiterten BTOR Format, jeweils unabhängig aufgetreten sind, liegt es nah, dass es sich um einen Fehler in der internen Optimierung von AVR handelt.

Auf diese werden wir in dieser Arbeit nicht weiter eingehen, haben diese jedoch der Vollständigkeitshalber erwähnt.

Erkenntnis:
Dieser Teil der Auswertung stellt eher einen informativen Teil dar, da man nicht direkt eine Aussage treffen kann. Die einzige Ausnahme wäre es, dass die Erweiterung keinen negativen Einfluss auf das Resultats des AVR hat.

4.3 Auswertung: Cherry Pick

In diesem Abschnitt soll auf ein bestimmtes Ergebnis des AVR´s genauer eingegangen werden. Das Ergebnis, um welches es geht, ist jenes aus Tabelle 2 bei dem sich das AVR Resultat geändert hat. Hier konnte innerhalb der Zeitbeschränkung eine Lösung gefunden werden, dies war bei dem Ursprungsformat nicht möglich.

Bei dieser Datei handelt es sich um die "qspiflash_dualflexpress_divthree-p054.btor" Datei. Diese Verbesserung kam durch die Hinzugabe der Informationen über 7 Bits einer einzelnen Zustandsvariablen zu Stande.

Um zu untersuchen, wie lange es dauern würde, eine Lösung für die Ursprungs BTOR Datei zu finden, wurde ein AVR Ausführung mit der ursprünglichen BTOR Datei mit einem Timeout durchgeführt, welcher auf 3.000 Sekunden angehoben wurde. Die Ergebnisse werden in den nachfolgenden Abbildungen dargestellt.

Result	Time	Mem.	#Refs
	sec	MB	
f_to	299.99	1267	223

Abbildung 25: *AVR-Ergebnis mit der Ursprungs BTOR Datei*

Result	Time	Mem.	#Refs
	sec	MB	
h	26.20	172	227

Abbildung 26: *AVR-Ergebnis mit der erweiterten BTOR Datei*

Result	Time	Mem.	#Refs
	sec	MB	
h	724.48	2996	264

Abbildung 27: *AVR-Ergebnis mit der Ursprungs BTOR Datei mit erhöhten Zeitlimit*

Erkenntnis:
Man benötigt bei der "qspiflash_dualflexpress_divthree-p054.btor" Datei ca. 698,28 Sekunden weniger Zeit und 2.824 MB weniger Speicher. Dies zeigt deutlich, welche enorme Auswirkung die Bit-Informationen auf die Modellüberprüfung auf der Wort-Ebene haben können.

4.4 Interpretation der Ergebnisse

Die einzelnen Auswertungen aus den vorherigen Kapiteln zeigen, dass die Hardwareverifikation auf der Wort-Ebene durch zusätzliche Informationen der Bit-Ebene profitieren können. Einzelfälle wie z.B. im Kapitel Cherry Pick, zeigen auf, welchen enormen Einfluss diese Information auf den gesamten Überprüfungsprozess haben kann. Aber auch die Auswertung des Zeit- und Speicherbedarfs zeigen, dass die Effizienz stark variiert. In Summe konnte jedoch ein klarer Vorteil in den beiden Bereichen erzielt werden.

Schlussfolgernd bedeutet dies, dass zusätzliche Bit-Ebenen Informationen die Hardware-Verifikation auf der Wort-Ebene verbessern können.

Dies geht daraus hervor, dass ein konkreter Fall gefunden wurde, bei dem eine deutliche Verbesserung festgestellt werden konnte. Zusätzlich lässt die gesamte Auswertung erkennen, dass dies keinen Einzelfall darstellt.

5 Fazit

Das Ziel war es, mit einem erstellten Programm zu untersuchen, ob Informationen aus der Bit-Ebene einen Einfluss auf die Modellüberprüfung der Wort-Ebene haben. Im Folgenden wird erläutert, weshalb diese Bachelorarbeit geschrieben wurde und ob die gesteckten Ziele erreicht wurden. Anschließend wird ein kurzer Ausblick gegeben, welche möglichen Erweiterungen auf diese Arbeit aufbauen könnten.

5.1 Zusammenfassung

In dieser Bachelorarbeit wurde der mögliche Einfluss von Bit-Ebenen Informationen auf die Modellüberprüfung auf der Wort-Ebene untersucht. Für die Untersuchung wurde ein Programm entwickelt, welches Bit-Informationen aus einer Bit-Ebenen Analyse in das Wort-Ebenen Format BTOR2 hinzufügt.

Durch diese Arbeit soll eine Möglichkeit aufgezeigt werden, die Modellüberprüfung auf der Wort-Ebene mithilfe von Daten aus der Bit-Ebene zu verbessern.

Hierzu wird als Stellvertreter der Wort-Ebene das noch junge BTOR2 Format gewählt, da dieses bei dem Entwurf als ein Format erstellt wurde, um verschiedene Modellüberprüfungsprogramme der Wort-Ebene miteinander zu vergleichen.

Die Ergebnisse haben gezeigt, dass ein Wort-Ebenen Format wie BTOR2 von zusätzlichen Bit-Informationen profitieren kann. Insbesondere wurde gezeigt, dass man einen zeitlichen Vorteil erhalten kann. Der Grad des Vorteils ist jedoch abhängig vom Hardwareentwurf und den spezifizierten Eigenschaften, sowie dessen Komplexität. Jedoch kann man durchaus sagen, dass der zeitliche Aufwand für die Bit-Ebenen Analyse und der Formats-Erweiterung, relativ gering im Verhältnis zum potenziellen Vorteil ist.

Nur bei sehr kleinen und einfachen Beispielen hat man einen erhöhten Speicherbedarf, da mehr Informationen zu Beginn vorhanden sind.

Durch die Auswertung des Benchmarks wurde gezeigt, dass Informationen aus einer Bit-Ebenen Analyse, einen Mehrwert für die Modellüberprüfung auf der Wort-Ebene bieten. Insbesondere im Kosten- und Nutzungsverhältnis, auf Bezug zur Gesamtlaufzeit, ist dies klar erkennbar.

5.2 Ausblick

Diese Arbeit zeigte, dass man Informationen zwischen zwei Abstraktionsebenen austauschen kann und dadurch die Modellüberprüfung unter Umständen verbessert wird. Jedoch wurde auch gezeigt, dass nicht jede analysierte Datei Informationen beinhaltet, die einen Mehrwert für die Modellüberprüfung haben.

Um herauszufinden unter welchen Bedingungen die zusätzlichen Informationen eine signifikante Verbesserung nach sich ziehen, könnte man einen Benchmark mit verschiedenen Modellüberprüfungsprogrammen erstellen.

Hierbei wäre es interessant zu erfahren, welche Kernalgorithmen bei den Modellüberprüfungsprogrammen am meisten von den zusätzlichen Informationen profitieren.

6 Literaturverzeichnis

Literatur

[1] Sammlung von Zitaten. Zugriff am 17.03.2022 auf https://www.quotez.net/german/computer.htm

[2] Christof Windeck. 2019. Vor 25 Jahren: Intel Pentium mit FDIV-Bug. Zugriff am 17.03.2022 auf https://www.heise.de/ct/artikel/Vor-25-Jahren-Intel-Pentium-mit-FDIV-Bug-4571751.html

[3] Wecker Dieter. 2021. Prozessorentwurf mit Verilog HDL. de Gruyter Oldenbourg. S.2, ISBN: 9783110717846

[4] Yosys. Zugriff am 17.03.2022 auf https://yosyshq.net/yosys/

[5] Armin Biere. (2007). The AIGER And-Inverter Graph (AIG) Format Version 20071012. Zugriff am 03.12.2021 auf http://fmv.jku.at/papers/Biere-FMV-TR-07-1.pdf

[6] Armin Biere; Keijo Heljanko; Siert Wieringa. (2011). AIGER 1.9 And Beyond. Zugriff am 04.12.2021 auf http://fmv.jku.at/papers/BiereHeljankoWieringa-FMV-TR-11-2.pdf

[7] Robert Brummayer; Armin Biere; Florian Lonsing (2008). BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking. Zugriff am 04.12.2021 auf http://fmv.jku.at/papers/BrummayerBiereLonsing-BPR08.pdf

[8] Aina Niemetz; Mathias Preiner; Clifford Wolf; Armin Bier. (2018). BTOR2 , BtorMC and Boolector 3.0. Zugriff am 05.12.2021 auf http://fmv.jku.at/papers/NiemetzPreinerWolfBiere-CAV18.pdf

[9] Christian Haubelt; Jürgen Teich. (2010). Digitale Hardware/Software-Systeme. Springer Heidelberg Dordrecht London New York, S.97/98[1],S.4[2], S.200[3]. DOI 10.1007/978-3-642-05356-6

[10] Aina Niemetz; Mathias Preiner; Clifford Wolf; Armin Bier. (2018). BTOR2 , BtorMC and Boolector 3.0 Slides. Zugriff am 05.12.2021 auf https://easychair.org/smart-slide/slide/gm3D#

[11] Armin Biere; Alessandro Cimatti; Edmund M. Clarke; Ofer Strichman; Yunshan Zhu. Bounded Model Checking. Zugriff am 05.12.2021 auf https://www.cs.cmu.edu/ emc/papers/Books%20and%20Edited%20Volumes/Bounded%20Model

[12] Clarke Edmund. 2000. Counterexample-Guided Abstraction Refinement. CAV. Lecture Notes in Computer Science. Vol. 1855. S.154-169. DOI 10.1007/10722167_15

[13] Haskell Data-Aeson Bibliothek. Zugriff am 15.03.2022 auf https://hackage.haskell.org/package/aeson-2.0.2.0/docs/Data-Aeson.html

[14] Uni Bremen.(SoSe 2003). Vorlesungsunterlagen. Zugriff am 03.02.2022 auf http:www.informatik.uni-bremen.de/agra/doc/lehrmat/sose03/qhe/verif_1.pdf

[15] Aleksandar Zeljic (stanford university). Zugriff am 03.02.2022 auf https://web.stanford.edu/class/cs357/lecture12.pdf

[16] Sunil P. Khatri, Kanupriya Gulati Advanced Techniques in Logic Synthesis, Optimizations and Applications. Springer New York Dordrecht Heidelberg London, S.187. DOI 10.1007/978-1-4419-7518-8

[17] Boolector. Zugriff am 01.03.2022 auf https://github.com/Boolector/boolector

[18] AVR - Model Checker. Zugriff am 01.03.2022 auf https://github.com/amangoel/avr

[19] btor2tools Repository. Zugriff am 01.03.2022 auf https://github.com/boolector/btor2tools

[20] https://www.cs.hm.edu/die_fakultaet/ansprechpartner/professoren/guedemann/index.de.ht Zugriff am 15.03.2022

[21] HWMCC2020 Benchmark Archiv Zugriff am 09.12.2021 auf http://fmv.jku.at/hwmcc20/hwmcc20benchmarks.tar.xz.

[22] http://fmv.jku.at/hwmcc20/. Zugriff am 09.12.2021

[23] Projekt Repository. Zugriff am 15.03.2022 auf https://gitlab.lrz.de/hm-kriedl/btor2project

[24] BTOR Benchmark. Zugriff am 09.12.2021 auf https://github.com/makaimann/btor-benchmarks