UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik
Heinz Nixdorf Institut und Institut für Informatik
Fachgebiet Softwaretechnik
Warburger Straße 100
33098 Paderborn

Generierung von Eigenschaftsprüfern in einem Hardware/Software-Co-Verifikationsverfahren

Bachelor-Arbeit

im Rahmen des Studiengangs Informatik
zur Erlangung des Grades
Bachelor of Science

von
Felix Pauck
Warburger Str. 52
33098 Paderborn

vorgelegt bei
Prof. Dr. Heike Wehrheim
und
Prof. Dr. Reinhard Keil

Paderborn, September 2014

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

1.1 Überblick und Motivation

Die Verbesserung von IT-Lösungen spielt in der Informatik eine zentrale Rolle. Es existieren beispielsweise zahlreiche Methoden, um Programmabläufe zu beschleunigen. Dazu gehören sowohl manuelle als auch automatische Optimierungsmethoden für Software- und Hardwarekomponenten. Eine im Fokus aktueller Forschung stehende Methode beruht auf der Auslagerung von Programmteilen auf rekonfigurierbare Hardware, wodurch das Programm beschleunigt werden soll.

Im Rahmen dieser Methode wird ein Analysetool eingesetzt, das ermittelt, welche Programmteile lohnenswert ausgelagert werden können. In den meisten Fällen bestehen diese Programmteile aus einer Menge von Anweisungen, die wiederholt im Programm auftreten. Diese Programmteile werden auf rekonfigurierbare Hardware ausgelagert, indem eine passende Konfiguration synthetisiert wird.
Dieser Vorgang, bestehend aus der Analyse des Programms und Synthese der Konfiguration, kann automatisch durchgeführt werden. Das heißt, dass dieses Zusammenspiel von Software (Analysetools) und Hardware(-Konfigurationen) eine neue, automatisierbare Möglichkeit zur Beschleunigung von Programmen bietet.

Zu den Risiken dieser Methode gehört, dass unter Umständen die Software korrekt implementiert wurde, aber die Synthese der Konfiguration fehlerhaft ist und nicht dem auszulagernden Programmteil entspricht. Um dies auszuschließen, muss die Verifikation der Software bzw. des Programms die Verifikation der Programmteile, die in Hardware *gegossen* wurden, beinhalten.
Dazu werden mit Hilfe des Analysetools zu jedem Aufruf des auszulagernden Programmteils Vor- und Nachbedingungen ermittelt. Wenn diese Bedingungen unter allen Umständen gelten, respektive die Nachbedingung aus der jeweiligen Vorbedingung folgt, ist sichergestellt, dass die Konfiguration korrekt synthetisiert wurde.

Die daraus resultierende Herausforderung liegt darin, zu überprüfen, dass diese Bedingungen eingehalten werden. Ein Lösungsansatz verwendet Eigenschaftsprüfer, die anhand der Eingaben der Konfiguration und den zugehörigen Ausgaben der Konfiguration zeigen, dass die Bedingungen eingehalten werden.

Damit die gesamte Methode automatisierbar bleibt, müssen diese Eigenschaftsprüfer aus den vorliegenden Informationen (auszulagernder Programmcode + Vor- und Nachbedingungen) generiert werden. Daher steht die Generierung von Eigenschaftsprüfern im Mittelpunkt dieser Arbeit.

1.2 Lösungsansatz

Die Erstellung eines Konzepts zur Generierung von Eigenschaftsprüfern ist das Hauptziel dieser Bachelorarbeit. Zunächst wird dazu eine formale Beschreibungsmöglichkeit für die vorliegenden Informationen (auszulagernder Programmcode + Vor- und Nachbedingungen) ermittelt. Des Weiteren erfolgt die Festlegung eines Formats für die zu generierenden Eigenschaftsprüfer. Das Konzept wird zeigen, wie die Generierung bzw. Transformation der vorliegenden Informationen in einen Eigenschaftsprüfer diesen Formats abläuft.

Teilziele dieser Arbeit sind die Implementierung des Konzepts in Form eines Tools zur Generierung von Eigenschaftsprüfern und das Testen der generierten Eigenschaftsprüfer.

1.3 Struktur der Arbeit

Im ersten Kapitel werden alle benötigten Grundlagen eingeführt. Dazu gehört die genaue Beschreibung von möglichen Einsatzgebieten für generierte Eigenschaftsprüfer und die Vorstellung von zwei Sprachen, die im Konzept eine zentrale Rolle spielen.

Auf Grundlage einer dieser Sprachen (SMT-Lib - Abschnitt 2.3) wird im folgenden Konzeptkapitel (Kapitel 3) eine Teilsprache definiert, die als Eigenschaftsprüfer-Beschreibungssprache geeignet ist. Aufbauend auf dieser Definition in Form einer Grammatik werden Übersetzungsregeln passend zu den Produktionsregeln und Nicht-Terminalsymbolen der Grammatik erstellt. Weitere Lösungen zu zentralen Problemen, die im Laufe der Generierung von Eigenschaftsprüfern auftreten, werden ebenfalls im Rahmen des Konzeptkapitels entworfen.

Das Implementierungskapitel (Kapitel 4) dient der Dokumentation der Entwicklung des Tools. Hier wird das in Kapitel 3 entwickelte Konzept umgesetzt. Hinzu kommt der Entwurf eines Benutzerhandbuches.

Im folgenden Kapitel 5 werden Tests anhand des erstellten Tools und den dadurch generierten Eigenschaftsprüfern durchgeführt und die Ergebnisse vorgestellt.

Abschließend werden die Ergebnisse der Arbeit zusammengefasst und ein Ausblick auf die Verwendbarkeit des Konzepts gegeben.

2 Grundlagen

In diesem Kapitel werden zunächst Hardware/Software-Co-Verifikationsverfahren allgemein vorgestellt. Im Anschluss wird das Proof-Carrying Hardware Verfahren als weitere Verwendungsmöglichkeit für generierte Eigenschaftsprüfer näher betrachtet. Die letzten beiden Unterkapitel setzen sich auseinander mit zwei Sprachen, die eine zentrale Rolle im Rahmen des Konzeptes zur Generierung von Eigenschaftsprüfern spielen.

2.1 Hardware/Software-Co-Verifikationsverfahren

Hardware/Software-Co-Verifikationsverfahren werden in der Regel im Zusammenhang mit einem HW/SW-Co-Design eingesetzt. Die Art HW/SW-Co-Design, die im Rahmen dieser Arbeit betrachtet wird, besteht aus einem Programm, das zuteilen auf rekonfigurierbare Hardware ausgelagert wird, mit der Absicht eine Beschleunigung des Programms zu erreichen. Die ausgelagerten Programmteile werden im Folgenden als Custom Instructions bezeichnet, da sie auf Prozessoren mit einer sogenannten „Custom Instruction Set Extension“ [SF10] ausgeführt werden sollen. Anhand der Spezifikation einer Custom Instruction kann eine Konfiguration synthetisiert werden, die der Custom Instruction entspricht und auf rekonfigurierbarer Hardware eingesetzt werden kann.

> Die Umsetzung der Spezifikation in eine Implementierung wird als Entwurf bezeichnet. Erfolgt die Umsetzung einer Spezifikation in eine Implementierung automatisch, so spricht man von Synthese. [Hau10]

Die Co-Verifikation bezieht sich darauf, dass das gesamte Programm verifiziert werden soll. Das heißt, dass das gesamte Programm inklusive der ausgelagerten Custom Instructions auf Korrektheit überprüft werden soll, um zu zeigen, dass es fehlerfrei ist. Für den Softwareteil des Programms existieren zahlreiche Methoden zur Durchführung der Verifikation. Für den Hardwareteil werden im Folgenden drei Ansätze anhand des Papers von Jakobs et al. [JPWW14] vorgestellt, die die Verifikation einer Custom Instruction ermöglichen.

2.1.1 Ansatz 1: Funktionale Äquivalenz

Im ersten Ansatz wird die funktionale Äquivalenz betrachtet. Um zu überprüfen, ob die Implementierung der Custom Instruction ($I(\underline{x})$) bzw. die synthetisierte

Konfiguration für alle Eingaben ($\underline{x}$) das gleiche Ergebnis liefert wie die formale Spezifikation der Custom Instruction ($S(\underline{x})$), wird ein sogenannter „Miter“ [DKP10] generiert (siehe Abbildung 2.1).

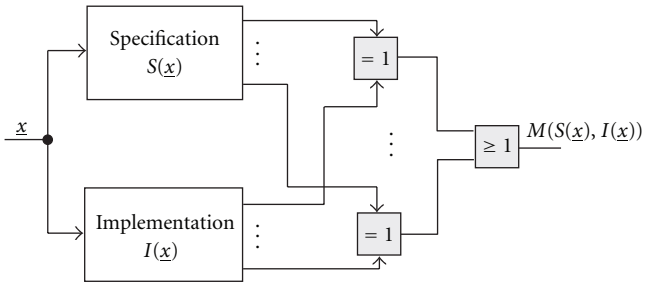

Abbildung 2.1: Miter [DKP10]

Dieser Miter überprüft die Formel: $\forall \underline{x} : S(\underline{x}) = I(\underline{x})$, indem er die Ergebnisse von Spezifikation und Implementierung zunächst miteinander per XOR verknüpft und dann alle XOR-Ergebnisse durch ein einzelnes OR zu einem Ergebnis führt. Dieses Endergebnis besteht aus einem einzelnen Bit. Entspricht dieses Bit dem Wert 1 bzw. *true* ist sicher, dass für mindestens eine Eingabe das Ergebnis der Konfiguration von dem Ergebnis, das anhand der Spezifikation berechnet wurde, abweicht. Demzufolge wäre die Verifikation fehlgeschlagen. Entspricht das Bit des Endergebnisses jedoch dem Wert 0 bzw. $false$ ist sichergestellt, dass die Konfiguration korrekt synthetisiert wurde und entsprechend der Spezifikation fehlerfrei funktioniert.

Für die Verifikation des gesamten Programms heißt das, dass die auf rekonfigurierbare Hardware ausgelagerten Teile fehlerfrei funktionieren und die Verifikation nur noch vom Softwareteil abhängt.

Dieser Ansatz weist einen Nachteil auf: Unter Umständen wird durch dieses aufwendige Verfahren viel mehr überprüft als unbedingt notwendig, um zu beweisen, dass die Implementierung der formalen Spezifikation der Custom Instruction entspricht. Dadurch kann ein unnötig hoher Aufwand entstehen.

2.1.2 Ansatz 2: Anforderungen an die Analyse-Domäne

Der zweite Ansatz basiert auf dem Einsatz einer Software-Analyse zur Festlegung einer Transfer-Relation.
Bevor die Software-Analyse mittels eines Analysetools durchgeführt wird, wird eine Analyse-Domäne festgelegt, wodurch die Software-Analyse an diese Domäne angepasst wird und demzufolge z.B. auf die Untersuchung von Vorzeichen beschränkt wird. Während der Software-Analyse wird eine Transfer-Relation ermittelt, die einerseits zur Analyse-Domäne und anderseits zur Custom Instruction passt. Die Eigenschaften dieser Transfer-Relation werden dann in der Verifikationsphase überprüft. Für alle möglichen Eingaben muss die Konfiguration eine

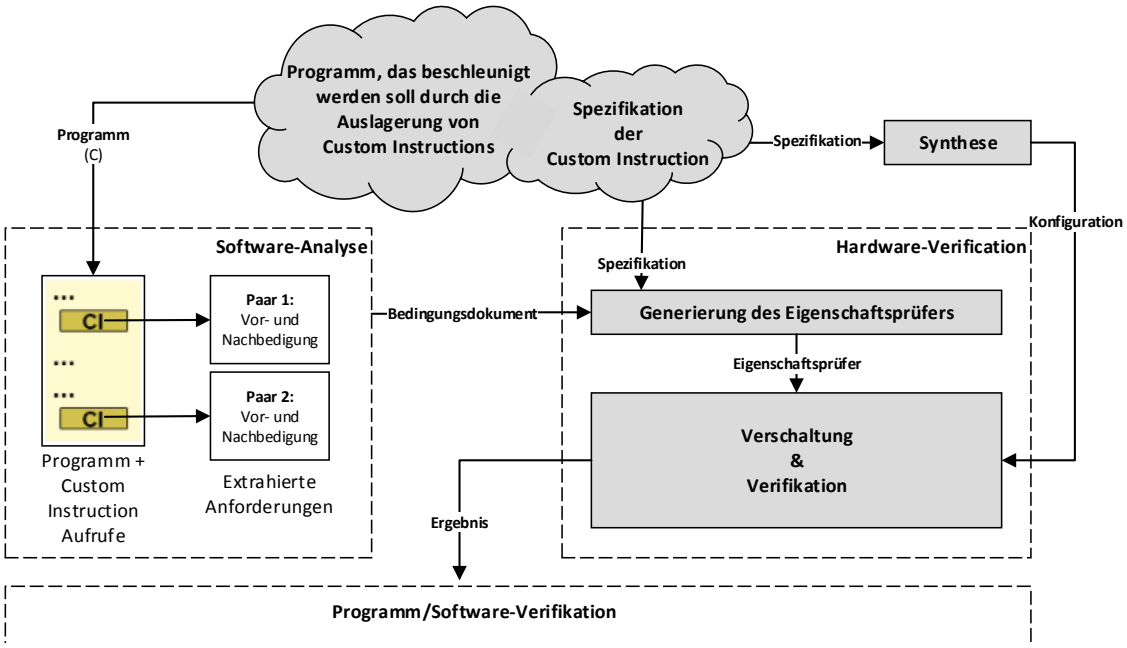

Abbildung 2.2: Workflow

Ausgabe entsprechend der Transfer-Relation berechnen, um als fehlerfrei angesehen zu werden.

Dieser Ansatz bietet zwei deutliche Vorteile: Erstens ist der Überprüfungsaufwand geringer, da nur die Eigenschaften der betrachteten Domäne überprüft werden. Im Falle einer Vorzeichen-Analyse ist das beispielsweise nur das für das Vorzeichen zuständige Bit. Zweitens enthalten Transfer-Relationen in vielen Fällen überlappende Eigenschaften und damit unterschiedliche Ausgangspunkte, die im Rahmen der betrachteten Domäne zum selben Ziel führen, wodurch die Anzahl der zu überprüfenden Fälle weiter schrumpft.

2.1.3 Ansatz 3: Anforderungen der jeweiligen Analyse

Der dritte Ansatz, dessen Workflow in Abbildung 2.2 dargestellt ist, verbindet die Software-Analyse enger mit der Hardware-Verifikation. Zur Umsetzung dieses Ansatzes werden Eigenschaftsprüfer benötigt, wie sie im Rahmen dieser Arbeit generiert werden. Daher wird dieser Ansatz ausführlich erläutert und als Grundlage für das Konzept genutzt.

Ausgangspunkt dieses Ansatzes ist wie zuvor ein Programm inklusive der ausgelagerten Custom Instructions sowie deren Spezifikation. Im ersten Schritt wird ähnlich wie in Ansatz 2 eine Software-Analyse durchgeführt. Die Anforderungen an die Hardware werden durch diese Analyse extrahiert. Das heißt, dass das Analysetool passend zu jedem Auftreten der Custom Instruction ein Paar bestehend aus einer Vor- und einer Nachbedingungen liefert. Dies wird erreicht, indem das Ana-

lysetool einen *abstract reachability graph* (ARG) generiert. Diesem ARG können direkt Anforderungen an die Hardware entnommen werden, die in vielen Fällen nur einer Teilmenge der Anforderungen einer Transfer-Relation, wie sie im zweiten Ansatz betrachtet wurde, entsprechen. Zumeist wird dadurch die Menge der zu überprüfenden Eigenschaften im Vergleich zu Ansatz 2 weiter eingegrenzt.

Die auszulagernde Custom Instruction taucht im ARG als Label einer oder mehrerer Kanten auf. Der Ausgangsknoten enthält jeweils die Vorbedingungen und der Zielknoten die Nachbedingungen. Bei nur einer Kante mit der Custom Instruction als Label entsteht somit ein Paar bestehend aus Vor- und Nachbedingungen. Im Falle mehrerer Kanten entstehen dementsprechend viele Paare. Als kleines Beispiel dient der in Abbildung 2.3 gezeigte Ausschnitt eines ARGs. Das aus diesem

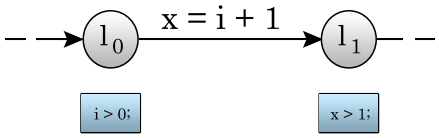

Abbildung 2.3: Ausschnitt eines ARGs

Ausschnitt zu entnehmende Paar passend zur Custom Instruction lautet (Custom Instruction: $a = b + c \Leftrightarrow$ Kantenlabel: $x = i + 1$):

$$\text{(Vorbedingung) } i > 0 \xrightarrow{x=i+1} \text{(Nachbedingung) } x > 0$$

Die ermittelten Vor- und Nachbedingungs-Paare werden durch das Analysetool auf eine standardisierte Form gebracht bzw. mittels einer fest definierten Sprache dargestellt. Die Darstellung der Paare in dieser Form bildet einen Teil des sogenannten **Bedingungsdokumentes**. Das Bedingungsdokument enthält neben den ermittelten Vor- und Nachbedingungen auch eine formale Beschreibung der Custom Instruction. Die standardisierte Form, die zur Darstellung der Bedingungspaare verwendet wird, wird auch für diese Beschreibung der Custom Instruction genutzt.

Aus dem Bedingungsdokument wird mittels des innerhalb dieser Arbeit entwickelten Konzepts ein Eigenschaftsprüfer generiert.
Dieser Eigenschaftsprüfer wird mit der Konfiguration verschaltet (siehe Abbildung 2.4), sodass der Eigenschaftsprüfer sowohl den Input als auch den Output der Konfiguration erhält.
Der Output des Eigenschaftsprüfers beschränkt sich auf ein einzelnes Fehlerbit, das anhand des In- und Outputs der Konfiguration ermittelt wird und anzeigt, ob die im Eigenschaftsprüfer kodierten Anforderungen erfüllt werden bzw. ob die Nachbedingungen aus den jeweiligen Vorbedingungen folgen.
Dieses Konstrukt, bestehend aus Konfiguration und Eigenschaftsprüfer, wird zur Verifikation als SAT-Formel kodiert. Falls die Konfiguration fehlerfrei synthetisiert

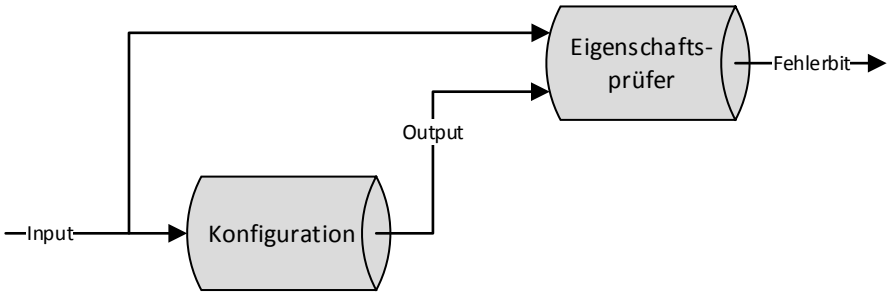

Abbildung 2.4: Verschaltung des Eigenschaftsprüfers

wurde, kann anhand dieser SAT-Formel mittels eines SAT-Solvers gezeigt werden, dass das Fehlerbit für keine Eingabe den Wert 1 bzw. *true* annimmt.

Für die Verifikation des gesamten Programms heißt das, dass die auf rekonfigurierbare Hardware ausgelagerten Teile vollständig verifiziert sind. Außerdem kann innerhalb der Verifikation des Softwareteils angenommen werden, dass die Custom Instruction-Aufrufe korrekte Ergebnisse liefern.

2.2 Proof-Carrying Hardware

Nach der Vorstellung des dritten Ansatzes, der eine Einsatzmöglichkeit für Eigenschaftsprüfer darstellt, wird in diesem Kapitel ein weiterer möglicher Einsatzbereich für generierte Eigenschaftsprüfer vorgestellt: Das Proof-Carrying Hardware (PCH) Verfahren.

PCH überträgt ein grundlegendes Prinzip der Softwaresicherheit, den Proof-Carrying Code [Nec97], in den Bereich der rekonfigurierbaren Hardware. Ein Beispielszenario zur Anwendung des PCH Verfahrens wird durch zwei Parteien verkörpert: Einen Konsumenten und einen Produzenten.
Der Konsument möchte Teile seines Programms auf rekonfigurierbare Hardware auslagern, um eine Beschleunigung seines Programms zu erreichen. Die Synthese einer Konfiguration übersteigt in den meisten Fällen jedoch die Möglichkeiten des Konsumenten. Infolgedessen beauftragt der Konsument einen Produzenten, eine Konfiguration zu synthetisieren. Zusätzlich zu dem auszulagernden Programmteil übergibt der Konsument dem Produzenten Vor- und Nachbedingungen, die unbedingt eingehalten werden müssen. Der Produzent kann seine Aufgabe nur unter Einsatz enormer Ressourcen bewältigen, weil er neben der Synthese der Konfiguration auch einen Beweis entwerfen muss, durch den garantiert wird, dass alle Bedingungen des Konsumenten eingehalten werden. Beweis und Konfiguration werden zusammen an den Konsumenten übergeben.

Durch Einsatz dieses Verfahrens ist der Konsument befreit von der Verantwortung, einen bekannten bzw. vertrauenswürdigen Produzenten zu wählen, weil er

die Bedingungen, die eingehalten werden müssen, selbst vorgibt. Zugleich kann er die Einhaltung der Bedingungen leicht anhand des Beweises überprüfen, ohne den Beweis selbst erstellen zu müssen, wodurch schlussendlich die Belastung des Konsumenten verringert wird.

Gleichermaßen wird dieses Verfahren im Einleitungskapitel des Papers von Drzevitzky et al. [DKP10] präsentiert. Dabei wird unter anderem ein spezieller Workflow vorgestellt. Dieser zeigt, wie eine logische Funktion mittels des PCH Verfahrens in eine Konfiguration umgewandelt wird, die garantiert die geforderte Funktion umsetzt. Dazu synthetisiert der Produzent zunächst eine Konfiguration. Dann generiert er einen Eigenschaftsprüfer, der folgendermaßen funktioniert.
Für alle Eingaben wird geprüft, ob das Ergebnis, das die Konfiguration liefert mit dem Ergebnis der logischen Formel für die jeweils selben Eingaben übereinstimmt. Ist dies für alle Eingaben der Fall wird ein Fehlerbit als Endergebnis auf 0 bzw. auf $false$ gesetzt. Konfiguration und Eigenschaftsprüfer werden als SAT-Formel kodiert, die genau in diesem Fall unerfüllbar ist. Der Beweis, der vom Produzenten erstellt und vom Konsumenten nur noch geprüft wird, besteht aus einer Auflistung von Resolutionsschritten, die die Unerfüllbarkeit der SAT-Formel beweisen. Konfiguration und Beweis werden dann an den Konsumenten versandt.

Ausgehend von seiner eigens vorgegebenen Funktion und der vom Produzenten versandten Konfiguration kann der Konsument nun seinen eigenen Eigenschaftsprüfer generieren. Die Kombination aus Konfiguration und Eigenschaftsprüfer wird auch auf Seiten des Konsumenten als SAT-Formel kodiert. Anhand dieser Formel kann der Konsument die im Beweis vorgegebenen Resolutionsschritte ausführen und erhält bei nicht manipulierter und korrekt erstellter Informationen das gleiche Ergebnis wie der Produzent. Das heißt, dass auch er zu dem Ergebnis kommt, dass die Formel nie erfüllt werden kann. Sollte er ein anderes Ergebnis erhalten kann geschlussfolgert werden, dass die Konfiguration und/oder der Beweis nicht korrekt ist oder bei der Übertragung manipuliert wurde. Wenn beides passend zueinander manipuliert wurde und die Überprüfung nicht fehlschlägt, wird die Manipulation nicht bemerkt, aber dennoch eine korrekte Konfiguration, die den Vorgaben des Konsumenten entspricht, verwendet.
Zum Abschluss kann der Konsument die rekonfigurierbare Hardware mit der neuen Konfiguration rekonfigurieren.

Diesem beschriebenen Workflow liegt als Ursprung eine logische Formel zugrunde. Angenommen die Grundlage wäre eine Custom Instruction mit dazu passenden Vor- und Nachbedingungen, müsste der Workflow nur geringfügig abgeändert werden, um auch auf dieser Grundlage zu funktionieren. Die in dieser Arbeit betrachteten Eigenschaftsprüfer können dementsprechend auch in diesem Workflow genutzt werden. Somit verkörpert auch PCH einen weiteren Einsatzbereich, für den die Generierung von Eigenschaftsprüfern unerlässlich ist.

2.3 SMT & SMT-Lib

Das Konzept zur Generierung von Eigenschaftsprüfern basiert auf der Transformation der erwähnten Bedingungsdokumente. Zur Darstellung dieser Bedingungsdokumente wird eine Teilsprache genutzt, deren Grundlage (SMT-Lib) in diesem Kapitel vorgestellt wird [BST12]. Zuvor werden jedoch die Satisfiability Modulo Theories (SMT) selbst erläutert [BSST08].

Innerhalb der **Satisfiability Modulo Theories** befasst man sich mir der Erfüllbarkeit von prädikatenlogischen Formeln, wobei die Darstellungsmöglichkeiten innerhalb dieser Formeln um eine Hintergrund-Theorie erweitert werden. Als Hintergrund-Theorie wird in diesem Zusammenhang eine Sammlung von Axiomen bezeichnet, wobei jedes Axiom neue Möglichkeiten für die Formulierung bietet und eine Interpretation vorschreibt.
Wird als Hintergrund-Theorie beispielsweise die quantorenfreie Integerarithmetik festgelegt, sind innerhalb dieser Theorie Axiome enthalten, die z.B. die Verwendung des arithmetischen Zeichens für Addition (+) oder die Verwendung der kleiner-Relation (<) ermöglichen und die zugehörigen Interpretationen festlegen.

Die Menge der Erfüllbarkeitsprobleme, die mittels dieser erweiterten prädikatenlogischen Formeln beschrieben werden können, werden als SMT-Probleme bezeichnet.

Mittels SMT-Solvern ist es in vielen Fällen möglich, die Entscheidbarkeit dieser Probleme zu ermitteln bzw. festzustellen, ob eine Lösung für das gegebene Problem existiert. Wenn eine Lösung existiert, wird durch den SMT-Solver das Ergebnis *sat* ausgegeben. Kann der SMT-Solver ermitteln, dass keine Lösung für das Problem existiert, lautet das Ergebnis *unsat*. Da SMT-Probleme jedoch zu den NP-vollständigen Problemen gehören, ist nicht garantiert, dass ein Ergebnis in polynomieller Zeit gefunden wird [CLRS09]. In diesem Fall kommt es oftmals zu einem Timeout durch den Solver oder die Ausgabe des Solvers lautet *unknown*.

SMT-Lib ist eine Sprache, die es erlaubt SMT-Probleme formal zu beschreiben. Sobald ein SMT-Problem mittels SMT-Lib vollständig beschrieben ist, können SMT-Solver anhand des Quellcodes Aussagen über die Erfüllbarkeit des Problems treffen.

Die Hauptgründe für die Entwicklung von SMT-Lib lagen in der Standardisierung der Ein- und Ausgabeformate für SMT-Solver und in der Standardisierung der Beschreibungen von Hintergrund-Theorien. Durch SMT-Lib wurde es möglich, SMT-Probleme und Hintergrund-Theorien auszutauschen, sodass unterschiedliche Solver die gleichen Grundlagen benutzen und die gleichen Probleme analysieren können. Dadurch wurde eine Vergleichbarkeit der Solver in Bezug auf ihre Leistung und ihre Lösungsgüte erreicht.

Jeder SMT-Lib-Quellcode, der ein SMT-Problem beschreibt, besteht aus einer Menge von Befehlen, die als Anweisungen für den Solver aufzufassen sind. Je-

der Befehl steht in Klammern und wird somit klar von einem anderen Befehl abgegrenzt. Der genaue Aufbau der einzelnen Befehle ist durch eine Grammatik definiert [BST12, Vgl. S. 35].

Beispiel

In folgendem Beispiel wird ein SMT-Problem beschrieben, in SMT-Lib-Quellcode ausformuliert und mittels eines SMT-Solvers auf Entscheidbarkeit untersucht. Betrachtet wird die Formel:

$$\alpha = x < y \wedge \neg(x < y + 0)$$

Als Hintergrund-Theorie soll die quantorenfreie Integerarithmetik gewählt werden. Dadurch können auch die Symbole $<, +$ und 0 innerhalb der logischen Formel genutzt und interpretiert werden. Codeausschnitt 2.1 zeigt die Formel α und den Aufruf die Erfüllbarkeit von α zu testen in Form von SMT-Lib-Quellcode.

```
1 (set-logic QF_LIA)
2 (declare-fun x() Int)
3 (declare-fun y() Int)
4 (define-fun alpha() Bool (and (< x y) (not (< x (+ y 0)))))
5 (assert alpha)
6 (check-sat)
```

Codeausschnitt 2.1: Beispiel SMT-Lib Code

In Zeile 1 wird die gewünschte Hintergrund-Theorie ($QF\_LIA \rightarrow$ Quantorenfreie Integerarithmetik) festgelegt. Zeile 2 und 3 definieren die zwei verwendeten Variablen x und y. Darauf wird die eigentliche Formel umgesetzt (siehe Zeile 4). Alle innerhalb der Formel bzw. innerhalb von SMT-Lib dargestellten Operationen werden in Präfix-Notation dargestellt. Die letzten beiden Zeilen legen fest, dass die Formel *alpha* bzw. α auf Erfüllbarkeit überprüft werden soll. Ein SMT-Solver kann anhand dieses Quellcodes entscheiden, dass α unerfüllbar (*unsat*) ist, was bedeutet, dass die Formel α im Rahmen der gewählten Hintergrund-Theorie durch keine Belegung der Variablen x und y erfüllt werden kann.

2.4 Verilog

Verilog wird im Konzept als Sprache zur Beschreibung der Eigenschaftsprüfer eingesetzt. Daher werden in diesem Kapitel die Grundlagen der Sprache erklärt und an einem Beispiel gezeigt.

Die Sprache Verilog gehört zur Klasse der Hardwarebeschreibungssprachen (HDLs – Hardware Description Languages). „HDLs werden beim Entwurf digitaler elektronischer Systeme eingesetzt, um die Schaltungsvorgaben zu erfassen, das Systemverhalten zu simulieren und das erstellte Design zu verifizieren“[Hop06]. Ve-

rilog ist neben VHDL[1] eine der meistverbreiteten HDLs. Die syntaktische und semantische Struktur von Verilog ähnelt der Struktur der Programmiersprache C.

Verilog wird benutzt, um Module zu beschreiben. Grundsätzlich besteht ein Verilog-Modul aus 4 Bestandteilen [Hop06]:

- dem Modulnamen
- den Eingangs- und Ausgangsports
- den Port- und Datentypdeklarationen
- und einer funktionalen Beschreibung auf Verhaltens- bzw. Strukturebene

Die Beschreibung eines Moduls beginnt mit dem Schlüsselwort **module** gefolgt von dem Namen des Moduls und einer Liste aller Ports, die das Modul nach außen hin verbinden.
Als Ports werden die Ein- und Ausgänge des Moduls bezeichnet. Da die Ein- und Ausgänge aus einer Reihe einzelner Bits bestehen, wird auch von Eingangs- bzw. Ausgangsvektoren gesprochen. Eingangsports werden durch das Schlüsselwort **input**, Ausgangsports durch **output** deklariert. Bei der Deklaration wird der Name und die Größe der Ports festgelegt. Die Größe gibt an, aus wie vielen Bits der jeweilige Port zusammengesetzt wird.
Variablen bzw. Datentypen werden z.B. durch das Schlüsselwort **wire** deklariert. wire ist einer der grundlegenden Datentypen in Verilog. „Analog zum elektrischen Draht können mit dem wire Verbindungen durchgeführt werden. Wie ein Draht kann aber ein wire keinen Signalzustand speichern.“[mik13] Bei der Deklaration einer Variablen wird wie bei der Deklaration eines Ports ein Name und eine Größe angegeben.
Die eigentliche Funktion des Moduls wird durch die Verschaltung der Ports festgelegt. Durch das Schlüsselwort **assign** können Ports und Variablen miteinander verschaltet werden. „Die assign-Anweisung kann als eine Art Verdrahtungsregel angesehen werden, mit der die Verbindung“ eines wires „beschrieben wird.“[mik13] Alle Anweisungen im Verilog-Code enden mit einem Semikolon; die Beschreibung eines Moduls bei dem Schlüsselwort **endmodule**.

Im folgenden Beispiel wird ein weiteres Schlüsselwort auftreten: **parameter**. Es dient der Festlegung und Initialisierung einer Konstanten.
Neben diesen Schlüsselwörtern werden direkt von Verilog unterstützte Operatoren auftreten, zu diesen gehören unter anderen:

[1] Very High Speed Integrated Circuit Hardware Description Language

Arithmetische Operatoren	**Logische Operatoren**	**Bitweise logische Operatoren**
• Addition: +	• Logisches Und: &&	• Logisches Und: &
• Subtraktion: −	• Logisches Oder: ||	• Logisches Oder: |
• Multiplikation: ∗	• Logisches Nicht: !	• Logisches Nicht: ∼
• Division: /		
• Modulo: %		

Nicht unterstützt wird beispielsweise die Implikation. Daher wird diese im folgenden Beispiel umgeformt: $A \rightarrow B \Leftrightarrow \neg A \vee B$

Beispiel

```
1  module Verilog_Bsp(input1, input2, error);
2          parameter N = 32;
3
4          // Definition der Ports
5          input[2*N-1:0] input1;
6          input[N-1:0] input2;
7          output error;
8
9          // Aufteilen von input1 auf die zwei
10         // Variablen x und y
11         wire[N-1:0] x;
12         wire[N-1:0] y;
13         assign x = input1[N-1:0];
14         assign y = input1[2*N-1:N];
15
16         // Definition von Bedingungen
17         wire con_1;
18         wire con_2;
19         wire con_3;
20         assign con_1 = (x == 0 );
21         assign con_2 = (y == 0 );
22         assign con_3 = (input2 == 0 );
23
24         // Verschaltung zu Vor- und Nachbedingung
25         wire pre;
26         wire post;
27         assign pre = (con_1 | con_2)
28         assign post = con_3
29
```

```
30          // Umsetzung einer Implikation zwischen Vor-
31          // und Nachbedingung
32          wire result;
33          assign result = ~pre | post;
34          assign error = ~result;
35  endmodule
```

Codeausschnitt 2.2: Verilog Beispiel

Das Beispiel zeigt ein Verilog Modul mit dem Namen $Verilog\_Bsp$ mit 2 Eingangsvektoren ($input1$ und $input2$) und einem Ausgang in Form eines einzelnen Bits ($error$). Das Modul stellt sicher, dass das Produkt ($input2$) der zwei in $input1$ enthaltenen Zahlen (x, y) nur 0 ist, wenn mindestens eine der Zahlen 0 ist. Sowohl das Produkt selbst als auch die Faktoren der Multiplikation werden als Input an das Modul übergeben und nicht im Modul berechnet. Das Modul überprüft somit die beiden Eingangsvektoren auf eine bestimmte Eigenschaften und gibt Auskunft über die Einhaltung dieser Eigenschaften in Form eines Fehlerbits.

3 Konzept

In diesem Kapitel wird das Konzept vorgestellt, das festlegt, wie die Generierung eines Eigenschaftsprüfers abläuft. Dieses Konzept wird im Verfahren eingesetzt, das im Grundlagenkapitel 2.1.3 vorgestellt wurde. Das Konzept knüpft an das Ende der Software-Analyse an. Das heißt, dass das Bedingungsdokument inklusive einer Beschreibung der Custom Instruction und der Anforderungen (Vor- & Nachbedingungen) bereits vorhanden ist. Das Konzept endet mit der vollständigen Generierung eines Eigenschaftsprüfers.

Generiert wird ein Eigenschaftsprüfer anhand der im Bedingungsdokument enthaltenen Bedingungen mit Hilfe von Transformationsregeln, die Teil dieses Konzeptes sind.

Die ebenfalls im Bedingungsdokument enthaltene Beschreibung der Custom Instruction dient einem anderen Zweck. Die innerhalb der Bedingungen genutzten Variablen wurden während der Analyse durch das Analyse-Tool benannt, wodurch die Variablennamen nicht unbedingt mit denen innerhalb Konfiguration bzw. der Spezifikation der Custom Instruction übereinstimmen. Um die Konfiguration dennoch korrekt mit dem Eigenschaftsprüfer zu verschalten (siehe Abbildung 2.4 – Kapitel 2.1.3) muss ein Matching erstellt werden, das die Variablen der Beschreibung mit den Variablen der Spezifikation verbindet. Die Erstellung dieses Matchings ist ebenfalls Gegenstand dieses Konzepts.

Als Analyse-Tool wird im Rahmen dieser Arbeit die konfigurierbare Software-Verifikations Plattform CPAchecker eingesetzt. Als Bedingungsdokument werden eine oder mehrere Dateien im SMT-Lib Format bezeichnet, da CPAchecker für jeden Aufruf der ausgelagerten Custom Instruction eine Datei mit den entsprechenden Anforderungen erzeugt. Demzufolge verwendet CPAchecker SMT-Lib als Standard zur Formulierung von Bedingungen und zur Beschreibung von Custom Instructions.

Das Tool zur Generierung von Eigenschaftsprüfern wiederum akzeptiert eine SMT-Lib Teilsprache, die in diesem Kapitel vorgestellt und anhand einer Grammatik definiert wird. Beschränkt ist die Teilsprache auf Worte aus SMT-Lib, die innerhalb der generierten Bedingungsdokumente auftreten können. Das heißt, dass die durch CPAchecker generierten Bedingungsdokumente als Eingabeinformationen für das Tool zur Generierung von Eigenschaftsprüfern genutzt werden können.

Die Eigenschaftsprüfer, die mit Hilfe dieses Konzeptes durch das Tool generiert werden, werden im Rahmen dieser Arbeit mittels der Hardwarebeschreibungsspra-

che Verilog spezifiziert.

Zusammenfassend lässt sich sagen, dass das hier vorgestellte Konzept die Umwandlung eines Bedingungsdokumentes (SMT-Lib Teilsprache) in einen Eigenschaftsprüfer (Verilog) ermöglicht, sodass ein Verilog-Modul entsteht, das die Überprüfung der Vor- und Nachbedingungen aus der Analyse automatisiert.

Abschließend wird in diesem Kapitel ein Beispiel angegeben, das den gesamten konzeptuellen Ablauf der Generierung widerspiegelt.

3.1 Verwendete SMT-Lib Teilsprache

Im Folgenden wird zunächst erläutert, welche Eigenschaften die SMT-Lib Teilsprache besitzt und warum die Teilsprache als Beschreibungssprache für Bedingungen und Custom Instructions ausreicht. Danach wird die Teilsprache anhand einer Grammatik eindeutig definiert.

SMT-Lib wird hauptsächlich im Zusammenhang mit SMT-Solvern verwendet und dient der Beschreibung von SMT-Problemen. Im Gegensatz dazu wird die hier vorgestellte Teilsprache ohne SMT-Solver verwendet und dient ausschließlich der Beschreibung von Bedingungen und Custom Instructions.

Dementsprechend muss die Teilsprache drei Aufgaben erfüllen:
Erstens muss es möglich sein, Vor- und Nachbedingungen darzustellen, weil diese während der Generierung in den eigentlichen Eigenschaftsprüfer transformiert werden. Zweitens muss die Teilsprache eine Möglichkeit zur formalen Beschreibung der Custom Instruction bieten, damit das Matching zur korrekten Verschaltung des generierten Eigenschaftsprüfers erstellt werden kann. Und drittens muss eine Möglichkeit zur Deklaration von Variablen existieren, damit innerhalb der Bedingungen die gleichen Variablen wie in der Beschreibung der Custom Instruction zur Verfügung stehen.

Um diese drei Aufgaben zu erfüllen werden die Befehle *declare-fun* und *define-fun* aus SMT-Lib genutzt.
Innerhalb von SMT-Lib weist *declare-fun* einem Symbol eine Funktion oder eine Konstante inklusive Typ zu. *define-fun* hingegen wird genutzt, um ein Symbol als Abkürzung für einen Ausdruck eines bestimmten Typs zu definieren.
Im Rahmen der Teilsprache werden den Befehlen andere Funktionen zugeordnet: *declare-fun* wird zur Deklaration und Typisierung von Variablen genutzt, wobei das Symbol als Variablenname und der Typ als Datentyp der Variablen verwendet wird. Stattdessen dient *define-fun* der Festlegung der Bedingungen und der Beschreibung der Custom Instruction, indem dem Symbol Teilausdrücke der Bedingungen bzw. der Beschreibung zugeordnet werden.
Anhand der Schlüsselwörter in Tabelle 3.1 wiederum werden die Teilausdrücke ihrer jeweiligen Funktion zugeordnet. Wird als Symbol beispielsweise das Schlüsselwort *ci* verwendet, gehört der zugehörige Ausdruck zur Beschreibung der Custom

Schlüsselwort	**Bedeutung**
pre (pre_x) post (post_x)	Vorbedingung(en) Nachbedingung(en) pre&post bilden ein Paar von Vor- und Nachbedingungen. Durch pre_x&post_x mit $x \in \mathbb{N}$ können weitere Paare definiert werden.
ci	Beschreibung der Custom Instruction

Tabelle 3.1: Schlüsselwörter der Teilsprache

Instruction.

Die mittels *define-fun* festgelegten Vor- und Nachbedingungen bestehen aus Kombinationen der folgenden Operatoren und enthalten weder All- ($\forall$) noch Existenzquantoren ($\exists$):

- Logische Operatoren: *and*, *or*, *not*
- Vergleichsoperatoren: $>$, $>=$, $<$, $<=$, $=$, *distinct*
- Arithmetische Operatoren: $+$, $-$, $*$, *div*, *mod*

Zusätzlich zu den standardmäßig in SMT-Lib nutzbaren Operatoren können somit auch *div* (Ganzzahldivision) und *mod* (Modulo) genutzt werden.
Innerhalb der Beschreibung der Custom Instruction wird der Vergleichsoperator ($=$) auch als mathematisches Gleichzeichen verwendet, um mathematische Funktionen mittels einer booleschen Formel darstellen zu können. Besteht die Beschreibung der Custom Instruction aus mehreren Ausdrücken, wird der *and*-Operator zur Verknüpfung verwendet.

Die Verwendung von Gleitkommazahlen ist innerhalb dieses Konzeptes nicht vorgesehen, weshalb die erlaubten Datentypen auf Int ($\mathbb{N}$) und $Bool$ ($false/true$) reduziert werden. Negative Zahlen und Variablen hingegen sind Teil des Konzeptes und werden durch eine unäre-Operation ermöglicht.

Die Teilsprache besteht somit ausschließlich aus den SMT-Lib-Befehlen *declare-fun* und *define-fun* und reicht aus, um alle benötigen Informationen zur Generierung eines Eigenschaftsprüfers darzustellen. Das Tool zur Generierung von Eigenschaftsprüfern akzeptiert daher nur Worte der Teilsprache.

3.1.1 Beispiel (Teil 1/3): Das Bedingungsdokument

Das hier abgebildete Bedingungsdokument gehört zu einem dreiteiligen Gesamtbeispiel, das in den folgenden Kapiteln vervollständigt wird (siehe Kapitel 3.2.3 und 3.3.2):

```
1  (declare-fun a() Int)
2  (declare-fun b() Int)
3  (declare-fun c() Int)
4  (declare-fun d() Int)
5  (declare-fun f() Int)
6
7  ;Custom Instruction
8  (define-fun ci_1() Bool (= d a))
9  (define-fun ci_2() Bool (and (> b f) (> c f)))
10 (define-fun ci() Bool (and ci_1 ci_2))
11
12 ;Vor- und Nachbedingungen
13 (define-fun pre() Bool (> b 0))
14 (define-fun pre_1() Bool (> c 0))
15 (define-fun post() Bool (> f b))
16 (define-fun post_1() Bool (> f c))
```

Codeausschnitt 3.1: Bedingungsdokument

Im Beispiel abgebildet sind die Definitionen der 5 enthaltenen Variablen a, b, c, d und f (Zeile 1-5). Die Beschreibung der Custom Instruction ist in den Zeilen 7 bis 10 zu sehen. Die beschriebene Custom Instruction besteht aus zwei Teilen, die mit einem *und* verknüpft werden. Der erste Teil beschreibt den Vergleich von d und a. Im Sinne einer Custom Instruction kann dieser boolesche Ausdruck auch wie eine Anweisung interpretiert werden: Setzte d gleich a. Der zweite Teil besteht aus zwei *größer-als*-Vergleichen, die durch ein weiteres *und* verknüpft werden. In den Zeilen 12 bis 16 werden zwei Paare von Vor- und Nachbedingungen festgelegt:

- **Paar 1**
  Vorbedingung (pre): $b > 0$
  Nachbedingung (post): $f > b$

- **Paar 2**
  Vorbedingung (pre_1): $c > 0$
  Nachbedingung (post_1): $f > c$

3.1.2 Die Grammatik der Teilsprache

Die zuvor textuell beschriebene Teilsprache wird durch die folgende Grammatik G präzise definiert ($L(G) = \textit{Teilsprache}$):

$G = (T, N, P, S)$

$T = \{\text{`␣'}, \text{`}+\text{'}, \text{`}-\text{'}, \text{`}/\text{'}, \text{`}*\text{'}, \text{`}=\text{'}, \text{`\%'}, \text{`?'}, \text{`!'}, \text{`.'}, \text{`\$'}, \text{`\_'}, \text{`\~{}'}, \text{`\&'}, \text{`\^{}'}, \text{`}<\text{'}, \text{`}>\text{'}, \text{`@'}, \text{`|'}, \text{`('}, \text{`)'}, \text{`\#'}, A-Z, a-z, 0-9\}$

$N = \{\text{`}\langle \textit{binary} \rangle\text{'}, \text{`}\langle \textit{command} \rangle\text{'}, \text{`}\langle \textit{hexadecimal} \rangle\text{'}, \text{`}\langle \textit{numeral} \rangle\text{'}, \text{`}\langle \textit{script} \rangle\text{'}, \text{`}\langle \textit{sorted\_var} \rangle\text{'}, \text{`}\langle \textit{spec\_constant} \rangle\text{'}, \text{`}\langle \textit{symbol} \rangle\text{'}, \text{`}\langle \textit{term} \rangle\text{'}\}$

$S = \{\text{`}\langle \textit{script} \rangle\text{'}\}$

$P = \{$

$p1 : \langle \textit{script} \rangle ::= \langle \textit{command} \rangle^*,$

$p2 : \langle \textit{command} \rangle ::= (\textit{declare-fun}\ \langle \textit{symbol} \rangle ()\ \ \langle \textit{sort} \rangle),$
$\quad | \ (\textit{define-fun}\ \langle \textit{symbol} \rangle (\langle \textit{sorted\_var} \rangle^*)\ \langle \textit{sort} \rangle\ \langle \textit{term} \rangle),$

$p3 : \langle \textit{term} \rangle ::= \langle \textit{spec\_constant} \rangle$
$\quad | \ \langle \textit{symbol} \rangle$
$\quad | \ (\langle \textit{symbol} \rangle\ \langle \textit{term} \rangle^+),$

$p4 : \langle \textit{sort} \rangle ::= \textit{Bool} \mid \textit{Int},$

$p5 : \langle \textit{sorted\_var} \rangle ::= (\langle \textit{symbol} \rangle\ \langle \textit{sort} \rangle),$

$p6 : \langle \textit{spec\_constant} \rangle ::= \langle \textit{numeral} \rangle | \langle \textit{hexadecimal} \rangle | \langle \textit{binary} \rangle | \langle \textit{bool} \rangle,$

$p7 : \langle \textit{symbol} \rangle ::=$ a non-empty sequence of letters, digits and the characters + − / * = % ? ! . $ _ ~ & ^ < > @ that does not start with a digit
$\quad |$ a sequence of printable ASCII characters other than \ that starts and ends with | and does not otherwise contain |

$p8 : \langle \textit{numeral} \rangle ::= 0 \mid$ a non-empty sequence of digits not starting with 0

$p9 : \langle \textit{hexadecimal} \rangle ::=$ #x followed by a non-empty sequence of digits and letters from A to F, capitalized or not

$p10 : \langle \textit{binary} \rangle ::=$ #b folowed by a non-empty sequence of 0s and 1s

$p11 : \langle \textit{bool} \rangle ::= \ \textit{true} \mid \textit{false}$

$\}$

Grammatik 3.2: G

Die Wörter der Grammatik G (p1)

G ist eine stark gekürzte Teilgrammatik, die auf der konkreten Grammatik von SMT-Lib basiert und im Folgenden als G_{SMT} bezeichnet wird [BST12, Appendix B]. Ein Teil von G_{SMT} gibt an, wie ein Skript konstruiert werden kann. Als Skripte werden alle Worte aus G_{SMT} bezeichnet die ausschließlich aus einer Reihe von Befehlen bestehen und keine anderen Elemente wie beispielsweise

Theorie-Definitionen enthalten. Da die Teilsprache nur aus zwei Befehlen besteht, können ausschließlich Skripte mittels G konstruiert werden. Die Konstruktion eines Skripts bzw. eines Bedingungsdokuments beginnt beim Startsymbol $\langle script \rangle$, das auf eine arbiträre Anzahl von Nicht-Terminalsymbolen vom Typ $\langle command \rangle$ abgebildet wird (p1).
Durch Produktionsregel p2 wird $\langle command \rangle$ auf spezifischere Befehle abgeleitet. Innerhalb von p2 werden jedoch weitere Nichtterminal-Symbole verwendet. Daher ist es sinnvoll, deren Bedeutung bzw. deren zugehörige Produktionsregeln zunächst zu erklären, bevor man p2 genauer betrachtet.

Variablennamen und Operatoren (p7)

Zuerst wird die Produktionsregel p7 betrachtet, die das Nicht-Terminalsymbol $\langle symbol \rangle$ direkt auf Terminalsymbole abbildet. Diese Produktionsregel wird verwendet, um Variablennamen zu deklarieren, zu verwenden oder Operatoren zu benutzen. Variablennamen dürfen im Rahmen der Teilsprache aus Buchstaben, Zahlen und einer Reihe von Sonderzeichen bestehen (siehe p7). Für Operatoren gelten dieselben Regeln, wobei im Rahmen der Teilsprache nur die zuvor aufgelisteten Operatoren unterstützt werden. Die Verwendung von anderen Symbolen als Operatoren wird durch die Produktionsregeln nicht ausgeschlossen, aber durch das Tool zur Generierung von Eigenschaftsprüfern überprüft und ggf. abgelehnt. Die Produktionsregel p7 wurde unverändert aus G_{SMT} übernommen.

Konstanten (p6, p8, p9, p10, p11)

Konstanten werden über das Nicht-Terminal $\langle spec\_constant \rangle$ abgeleitet. Die einzigen erlaubten Konstanten der Teilsprache sind natürliche Zahlen sowie die Wahrheitswerte $true$ und $false$. Negative, ganze Zahlen können mittels einer Kombinationen aus einer positiven Zahl und einer unären Operation dargestellt werden. Zahlen können in drei unterschiedlichen Formaten verwendet werden: Ganzzahlig-dezimal, hexadezimal und binär.

- Eine ganzzahlig-dezimale Darstellungsform wird erreicht durch die Abbildung auf das Nicht-Terminal $\langle numeral \rangle$. Dieses wiederum wird durch Produktionsregel p8 direkt auf Terminalsymbole abgebildet. In diesem Fall eine positive natürliche Zahl oder 0 ($\mathbb{N}_0$).

- Durch Abbildung auf $\langle hexadecimal \rangle$ wird die hexadezimale Darstellungsform gewählt. Dabei beginnt die Zahl mit dem Ausdruck „#x“ und wird gefolgt von einer Reihen von Zahlen bzw. den Buchstaben A bis F.

- Ähnlich definiert ist die binäre Darstellungsform. Der führende Ausdruck in diesem Fall ist „#b“ und wird gefolgt von beliebig vielen Nullen und Einsen. Durch Abbildung auf das Nicht-Terminalsymbol $\langle binary \rangle$ wird diese Darstellungsform ermöglicht.

Die Abbildung auf einen Wahrheitswert wird durch Regel p11 ermöglicht.

Im Vergleich zu der äquivalenten Produktionsregel in G_{SMT} wurden die Alternativen innerhalb von p6 reduziert. Die Möglichkeiten der Abbildung auf Strings und dezimale Zahlen wurden entfernt, da die Verwendung der Datentypen *String* und *Real* im Rahmen der Teilsprache nicht gestattet ist.

Datentypen (p4)

Produktionsregel p4 legt den Datentyp einer Variablen fest. p4 wurde erstellt auf Grundlage der Produktionsregel aus G_{SMT}, die das Nicht-Terminalsymbol $\langle sort \rangle$ auf eine von zwei Alternativen abbildet. Eine dieser Alternativen bildet $\langle sort \rangle$ auf einen beliebigen Datentyp ab. Durch die andere Alternative wird das Nichtterminal auf einen zusammengesetzten Datentyp abgebildet, z.B. eine Liste von Int-Werten.
Im Gegensatz dazu kann innerhalb von G $\langle sort \rangle$ direkt auf $Bool$ oder Int abgebildet werden, da dies die einzigen zulässigen Datentypen im Rahmen der Teilsprache sind und es nicht gestattet ist, neue Datentypen zu definieren. Daher lautet p4:

$$\langle sort \rangle ::= Bool \mid Int$$

Terme, Variablen und Konstanten (p3)

Die Produktionsregel p3 dient der Abbildung von mathematischen Termen und Teilbedingungen, die innerhalb der Bedingungen bzw. der Beschreibung der Custom Instruction genutzt werden. Dazu kann das Nicht-Terminalsymbol $\langle term \rangle$ auf Konstanten bzw. natürliche Zahlen, Variablen oder geklammerte mathematische/boolsche Ausdrücke abgeleitet werden:

$$\begin{aligned} \langle term \rangle ::= \quad & \langle spec\_constant \rangle \\ \mid \quad & \langle symbol \rangle \\ \mid \quad & (\langle symbol \rangle \quad \langle term \rangle^{+}) \end{aligned}$$

Durch die Ableitung auf $\langle spec\_constant \rangle$ wird $\langle term \rangle$ unter Nutzung von p6 auf eine Konstante abgebildet.
Die zweite Alternative dient der Abbildung auf eine Variable durch Ableitung über das Nicht-Terminal $\langle symbol \rangle$ (siehe p7). Dadurch, dass in der Teilsprache nur $Bool$ und Int als Datentypen verwendbar sind und indizierte Variablen nicht besonders behandelt werden, kann $\langle term \rangle$ innerhalb von G direkt auf $\langle symbol \rangle$ abgeleitet werden. In G_{SMT} dagegen ist dazu eine längere Ableitungskette nötig, die ausschließlich Alternativen beinhaltet, die hier nicht erlaubt sind.
Die dritte und letzte Alternative erlaubt die Abbildung auf einen geklammerten mathematischen oder logischen Term in Präfix-Notation.

Lokale Variablen (p5)

Durch p5 wird das bis hierher nicht aufgetretene Nicht-Terminalsymbol $\langle sorted\_var \rangle$ auf den Ausdruck ($\langle symbol \rangle$ $\langle sort \rangle$) abgebildet. Das Nicht-Terminal tritt im Zusammenhang mit dem Befehl „define-fun“ auf und dient der Festlegung von *lokalen* Variablen. Lokal heißt in diesem Zusammenhang, dass die definierten Variablen nur im darauffolgenden Term nutzbar sind. Das folgende Beispiel demonstriert die Verwendung einer lokal definierten Variablen ($localVar$):

$(declare\text{-}fun\ value()\ Int)$
$(define\text{-}fun\ greatherthan((localVar\ Int))\ Bool\ (>\ localVar\ value))$ ← **gültig**
$(define\text{-}fun\ greatherthanequalto()\ Bool\ (>=\ localVar\ value))$ ← **ungültig**

Die lokale Variable $localVar$ wird in der zweiten Zeile definiert und verwendet. In der dritten Zeile ist diese Variable bereits nicht mehr verfügbar. Im Gegensatz ist die in Zeile 1 definierte Variable *value* überall verfügbar.

Die Ableitung von $\langle sorted\_var \rangle$ entspricht einer Kurzform des Befehls *declare-fun* und dient der Abkürzung der Deklaration von einmalig genutzten Variablen.

Die Befehle „declare-fun“ und „define-fun“ (p2)

Produktionsregel p2, bildet $\langle command \rangle$ auf einzelne spezifischere Befehle ab:

$$\begin{aligned} \langle command \rangle ::= &\quad (declare\text{-}fun\ \langle symbol \rangle ()\ \ \langle sort \rangle) \\ \mid &\quad (define\text{-}fun\ \langle symbol \rangle (\langle sorted\_var \rangle^*)\ \langle sort \rangle\ \langle term \rangle) \end{aligned}$$

Alternative 1 dient der Abbildung auf den Befehl *declare-fun*. In G_{SMT} ist es möglich der Funktion, die durch $\langle symbol \rangle$ bezeichnet wird, Eingabeparameter zuzuweisen. Innerhalb der Teilsprache hat der Befehl *declare-fun* die Funktion, Variablen festzulegen. Die Informationen, die in diesem Fall benötigt werden, beschränken sich auf den Namen der Variablen ($\langle symbol \rangle$) und den Typ ($\langle sort \rangle$), wodurch die Angabe von Eingabeparametern überflüssig wird. Damit $L(G)$ weiterhin eine Teilsprache von SMT-Lib bleibt und um die SMT-Lib-ähnliche Struktur zu erhalten, bleiben die Klammern hinter dem Variablennamen dennoch erhalten.
Durch den folgenden Ausdruck wird im Kontext von $L(G)$ eine Variable mit dem Namen *bsp* vom Typ *Bool* definiert:

$$(declare\text{-}fun\ bsp()\ Bool)$$

Alternative 2 wurde unverändert aus G_{SMT} übernommen. Durch den Befehl *define-fun* werden sowohl die Vor- und Nachbedingungen definiert als auch die Beschreibung der Custom Instruction angegeben. Die in Tabelle 3.1 angegebenen Schlüsselwörter definieren, welche Funktion die jeweilige Instanz des Befehls erfüllt. Zum Beispiel stellen die folgenden zwei Zeilen die Vorbedingung dar, dass y kleiner als $(3\ *\ x)$ ist:

$$(define\text{-}fun\ term()\ Int\ (*\ 3\ x))$$
$$(define\text{-}fun\ pre()\ Bool\ (<\ y\ term))$$

Das folgende Beispiel zeigt die Beschreibung einer Custom Instruction, die eine Zahl auf das dreifache einer anderen Zahl abbildet ($y = 3 * x$):

$$(define\text{-}fun\ ci()\ Bool\ (=\ y\ (*\ 3\ x)))$$

In diesem Fall wird das Symbol = dazu verwendet, das Ergebnis einer Rechnung einer Variablen zuzuweisen. Da das =-Symbol innerhalb von SMT-Lib jedoch ausschließlich als Vergleichsoperator verwendet wird, wird als Datentyp im Zusammenhang mit der Beschreibung der Custom Instruction immer *Bool* genutzt.

Mittels dieser Darstellungsmöglichkeiten können alle Custom Instructions dargestellt werden, die aus einer beliebigen Anzahl boolescher Funktionen bestehen. Nicht darstellbar sind Custom Instructions, die Schleifen enthalten.

Ableitungsbeispiel

Folgende Ableitungskette spiegelt die Konstruktion des letzten Beispiels wieder; dabei werden alle Produktionsregeln außer p5, p9, p10 und p11 angewandt, da das Beispiel keine lokalen Variablen, hexadezimal bzw. binär dargestellte Konstanten oder konstante Wahrheitswerte enthält:

$$
\begin{aligned}
& \langle \textbf{script} \rangle \\
\xrightarrow{p1}\ & \langle \textbf{command} \rangle \\
\xrightarrow{p2}\ & (define\text{-}fun\ \langle \textbf{symbol} \rangle ()\ \langle sort \rangle\ \langle term \rangle) \\
\xrightarrow{p7}\ & (define\text{-}fun\ ci()\ \langle \textbf{sort} \rangle\ \langle term \rangle) \\
\xrightarrow{p4}\ & (define\text{-}fun\ ci()\ Bool\ \langle \textbf{term} \rangle) \\
\xrightarrow{p3}\ & (define\text{-}fun\ ci()\ Bool\ (\langle \textbf{symbol} \rangle\ \langle term \rangle^{+})) \\
\xrightarrow{p7}\ & (define\text{-}fun\ ci()\ Bool\ (=\ \langle \textbf{term} \rangle^{+})) \\
\xrightarrow{p3,p3}\ & (define\text{-}fun\ ci()\ Bool\ (=\ \langle \textbf{symbol} \rangle\ (\langle symbol \rangle\ \langle term \rangle^{+}))) \\
\xrightarrow{p7}\ & (define\text{-}fun\ ci()\ Bool\ (=\ y\ (\langle \textbf{symbol} \rangle\ \langle term \rangle^{+}))) \\
\xrightarrow{p7}\ & (define\text{-}fun\ ci()\ Bool\ (=\ y\ (*\ \langle \textbf{term} \rangle^{+}))) \\
\xrightarrow{p3,p3}\ & (define\text{-}fun\ ci()\ Bool\ (=\ y\ (*\ \langle \textbf{spec\_constant} \rangle\ \langle symbol \rangle))) \\
\xrightarrow{p6}\ & (define\text{-}fun\ ci()\ Bool\ (=\ y\ (*\ \langle \textbf{numeral} \rangle\ \langle symbol \rangle))) \\
\xrightarrow{p8}\ & (define\text{-}fun\ ci()\ Bool\ (=\ y\ (*\ 3\ \langle \textbf{symbol} \rangle))) \\
\xrightarrow{p7}\ & (define\text{-}fun\ ci()\ Bool\ (=\ y\ (*\ 3\ x)))
\end{aligned}
$$

Anmerkungen zur Grammatik G

Die Grammatik G mit allen beschriebenen Produktionsregeln reicht aus, um Bedingungen und Custom Instructions zur Erstellung von Eigenschaftsprüfern vollständig zu beschreiben. Warum das meiste aus G_{SMT} dazu nicht benötigt wird, wird im Folgenden anhand der restlichen Befehle neben *declare-fun* und *define-fun*

erläutert. Die im Folgenden mit 1 - 3 und 5 - 10 bezeichneten Befehle sind nur im Zusammenhang mit einem SMT-Solver einsetzbar und aus diesem Grund in der Teilsprache $L(G)$ nicht verfügbar.

1) Mit dem Befehl *set-logic* wird dem SMT-Solver mitgeteilt, welche Hintergrundtheorie bei der Betrachtung des Skripts berücksichtigt werden muss.

2) *set-option* und *get-option* bearbeiten Optionen, die festlegen, wie der SMT-Solver arbeiten soll. Eine Vielzahl von Optionen werden von SMT-Lib vorgegeben und können mit set-option gesetzt werden. Ob der verwendeter Solver die jeweilige Option unterstützt, kann durch den Aufruf get-option ermittelt werden.

3) *set-info* und *get-info* ähneln den Befehlen *set-option* und *get-option* mit dem Unterschied, dass hierbei Informationen über den Solver bearbeitet werden anstelle von Informationen über die Funktionsweise des Solvers. Die Informationen können variieren abhängig vom aktuellen Zustand des Solvers.

4) Mit *declare-sort* und *define-sort* können neue Datentypen festgelegt werden. „(declare-sort Pair 2)“ legt bespielsweise fest, dass der erstellte Datentyp Pair immer 2 Parameter benötigt. Innerhalb der Teilsprache sind jedoch nur *Int* und *Bool* als Datentypen zulässig, wodurch die Definition neuer Datentypen überflüssig wird.

5) „The push and pop commands enable some scoping of sort and function declarations and of assertions“ [Cok13]. Daraus geht hervor, dass durch *push* und *pop* beispielsweise Assertions oder Deklarationen auf einen Stapel gelegt oder aus diesem entnommen werden können. Der aktuelle Zustand des Stapels entscheidet dabei, welche Assertions bzw. Deklarationen im Falle einer Erfüllbarkeitsprüfung gelten. Festzuhalten bleibt, dass diese Befehle nur im Zusammenhang mit einem Solver sinnvoll einzusetzen sind, weil die Verwaltung des Stapels durch den Solver erfolgt.

6) Mit dem Befehl *assert* werden Assertions deklariert. Diese Assertions werden z.B. im Falle einer Erfüllbarkeitsprüfung betrachtet. Der Solver ermittelt, ob alle Assertions erfüllt werden können und gibt das entsprechende Ergebnis aus.

7) Durch *check-sat* wird eine Erfüllbarkeitsprüfung der bisher festgelegten Assertions veranlasst. Das Ergebnis dieser Prüfung wird vom Solver ausgegeben.

8) *get-assertions* veranlasst die Ausgabe aller Assertions, die aktuell aktiv sind. Das heißt, dass durch den Aufruf von *get-assertions* alle Assertions aufgelistet werden, die aktuell durch *check-sat* auf Erfüllbarkeit geprüft würden.

9) *get-proof* und *get-unsat-core* gehören zu den unsat-Operationen und können dazu genutzt werden, die verfügbaren Informationen nach Überprüfung der Erfüllbarkeit zu erweitern, falls das Ergebnis *unsat* ist. Wie die erweiterten Informationen aussehen und ob sie verfügbar sind, ist abhängig vom verwendeten Solver.

10) Das Gegenstück zu diesen unsat-Operationen stellen die zwei sat-Operationen *get-value* und *get-assignment* dar. In diesem Fall wird die Ausgabe erweitert, falls die Erfüllbarkeitsprüfung *sat* als Ergebnis liefert. Auch diese Befehle sind optional und eine Unterstützung ist abhängig vom verwendeten Solver.

Da diese Fülle an Befehlen nicht übernommen wurde und auch viele der Produktionsregeln aus G im Vergleich zum Gegenstück aus G_{SMT} stark vereinfacht wurden, wird erreicht, dass G sehr kompakt ist. Nichtsdestotrotz bietet G alle erforderlichen Möglichkeiten zur Beschreibung von Bedingungen und Custom Instructions, um Eigenschaftsprüfer zu generieren.

3.2 Matching der Variablen

Die vorgestellte Teilsprache ermöglicht es, sowohl die Vor- und Nachbedingungen als auch eine Beschreibung der Custom Instruction in einem Bedingungsdokument darzustellen. Dieses Bedingungsdokument wird durch ein Analyse-Tool wie beispielsweise CPAchecker erstellt. Wie die verwendeten Variablen bei der Erstellung benannt werden, ist nicht bekannt.
Sicher ist, dass mindestens zwei Eingangsvektoren für den Eigenschaftsprüfer existieren. Der erste entspricht dem Eingangsvektor, der zweite dem Ausgangsvektor der Konfiguration (siehe Abbildung 2.4 – Kapitel 2.1.3). Damit der Eigenschaftsprüfer generiert und korrekt mit der Konfiguration verschaltet werden kann, muss ermittelt werden, in welcher Reihenfolge die Werte der Variablen an den Eingangsvektoren anliegen. Um die Reihenfolge zu ermitteln wird ein Matching zwischen der Beschreibung der Custom Instruction aus dem Bedingungsdokument und der Spezifikation der Custom Instruction hergestellt. Die Struktur dieser Spezifikation wird im Folgenden definiert und anschließend zur Generierung des Matchings genutzt.

Werte, die zu den Variablen passen und nicht innerhalb einer Custom Instruction auftreten, aber dennoch in mindestens einer Vor- und/oder Nachbedingung genutzt werden, werden über einen weiteren Eingangsvektor an den Eigenschaftsprüfer übergeben. Die Reihenfolge der Werte auf diesem Eingangsvektor entspricht der Reihenfolge, in der die Variablen im Bedingungsdokument auftreten.

3.2.1 Spezifikation der Custom Instruction

Für die Spezifikation einer Custom Instruction wird das folgende, für diese Arbeit entworfene Schema benutzt.

In der ersten Zeile einer Spezifikation befindet sich die Signatur der Funktion bzw. der Custom Instruction. Dabei werden zunächst in runden Klammern und durch Kommata getrennt die eingehenden Variablen aufgezählt. Die Reihenfolge der Variablen entspricht dabei der Reihenfolge der zugehörigen Werte des ersten Eingangsvektors (siehe $in_1, in_2, ..., in_i$ mit $i \in \mathbb{N}$ in Abbildung 3.1).
Danach folgt ein Pfeil $(->)$, der aussagt, dass die zuvor genannten Variablen auf die folgenden Variablen abgebildet werden. Die folgenden Variablen entsprechen den ausgehenden Variablen, die durch die Konfiguration berechnet wurden. Die passenden Werte zu diesen Variablen befinden sich ebenfalls in entsprechender Reihenfolge auf dem zweiten Eingangsvektor. Auch diese Variablen werden in runden Klammern und durch Kommata getrennt angegeben (siehe $out_1, ..., out_j$ mit $j \in \mathbb{N}$ in Abbildung 3.1).

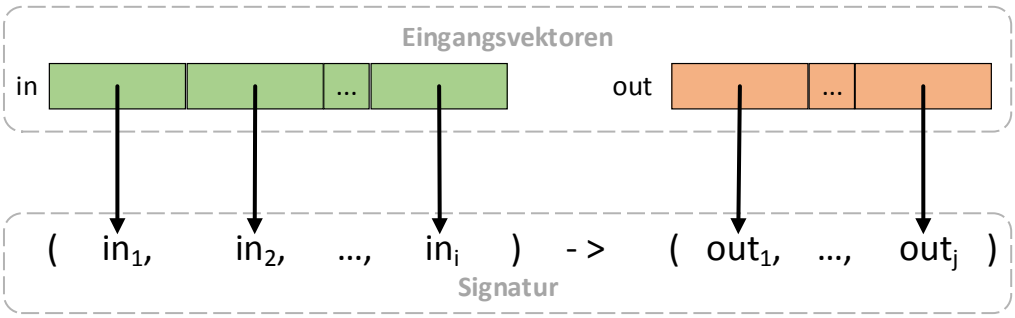

Abbildung 3.1: Eingangsvektoren + Signatur

In den folgenden Zeilen wird der Funktionsrumpf spezifiziert. Dieser besteht pro Zeile aus einer booleschen Funktion. Innerhalb dieser Funktionen können alle zuvor definierten Variablen und alle zur Verfügung stehenden Operatoren verwendet werden. Die Struktur muss mit der der Beschreibung der Custom Instruction aus dem Bedingungsdokument übereinstimmen. Damit geht einher, dass wie in SMT-Lib bzw. der Teilsprache alle Terme in Präfix-Notation angegeben und geklammert werden müssen. Beendet wird jeder Abschnitt eines Funktionsrumpfes durch ein Semikolon.

In der Teilsprache werden die einzelnen Funktionen der Custom Instruction durch den Operator *and* miteinander verbunden. Dadurch wird den Funktionen eine feste Reihenfolge vorgegeben. Diese ist mitverantwortlich für die konkrete Struktur der Beschreibung der Custom Instruction. Im Falle der Spezifikation werden die einzelnen Funktionen nicht durch einen Operator explizit miteinander verknüpft. Die Reihenfolge ist dennoch anhand der Leserichtung fest vorgeschrieben und zwar von oben nach unten und von links nach rechts.

Eine genaue Definition der möglichen Spezifikationen befindet sich in Form einer Grammatik im Anhang (siehe Anhang A.1).

Beispiel einer Custom Instruction Spezifikation

$(a, b, c) \; -> \; (d, e)$
$(= \; a \; d);$
$(and \; (> \; b \; e) \; (> \; c \; e));$

Die Signatur des Beispiels zeigt, dass über den ersten Eingangsvektor drei Werte und über den zweiten zwei Werte übertragen werden. Des Weiteren ist steht fest, wo die Werte der einzelnen Variablen anliegen. So ist z.B. bekannt, dass der Wert der Variablen c an dritter Stelle des ersten Eingangsvektors abgelesen werden kann.
Der Funktionsrumpf zeigt zwei boolsche Operationen. Die erste vergleicht den Wert einer eingehenden Variablen mit dem Wert einer ausgehenden Variablen. Die zweite besteht aus zwei Vergleichen, verknüpft durch den logischen Operator *and*.

Mit Hilfe dieses Schemas können alle Custom Instructions dargestellt werden, die auch innerhalb eines Bedingungsdokumentes dargestellt werden können, sodass ein Matching zwischen Beschreibung und Spezifikation hergestellt werden kann. Außerdem kann jeder Variablen einer Spezifikation ein Wertebereich auf einem der zwei Eingangsvektoren zugewiesen werden.

3.2.2 Erstellen eines Matchings

Die Informationen, die zum Zeitpunkt der Generierung des Eigenschaftsprüfers vorliegen, bestehen aus der Spezifikation der Custom Instruction und einem Bedingungsdokument inklusive Beschreibung der Custom Instruction. Um einen Eigenschaftsprüfer korrekt zu verschalten, muss es möglich sein, den Werten der Variablen des Bedingungsdokuments, die durch das Analyse-Tool benannt wurden, Bereiche auf den Eingangsvektoren zuzuordnen.
Um dieses Ziel zu erreichen, wird ein Matching zwischen den Variablen der Beschreibung und den Variablen der Spezifikation ermittelt, da die Wertebereiche passend zu den Variablen der Spezifikation bereits bekannt sind.

Zur Erstellung eines Matchings wird die Beschreibung und die Spezifikation der Custom Instruction jeweils in einen Graphen umgewandelt; beide bestehen aus einer Knoten- und einer Kantenmenge ($G_1 = (V_1, E_1), G_2 = (V_2, E_2)$).
Zur Erstellung dieser Graphen werden zunächst alle enthaltenen Terme bzw. booleschen Formeln betrachtet. Wie im letzten Abschnitt beschrieben ist die Struktur der Terme in der Spezifikation äquivalent zu der Struktur der Terme in der Beschreibung der Custom Instruction aus dem Bedingungsdokument. Daher besteht ein Term in beiden Quellen aus einem Operator und mindestens einer Komponente.

In der Grammatik G_{Spec} z.B. werden Terme durch zwei Produktionsregeln ab-

geleitet (siehe Anhang A.1):

$$p4 : \langle expression \rangle ::= \text{`(`}\ \langle operator \rangle\ \langle component \rangle^{+}\ \text{`)`},$$
$$p5 : \langle component \rangle ::= \langle valueORvariable \rangle\ \mid\ \langle expression \rangle,$$

$\langle operator \rangle$ entspricht dabei einem innerhalb der Teilsprache erlaubten Operator in einer booleschen Formel. $\langle valueORvariable \rangle$ wird auf eine Variable oder eine Konstante abgeleitet.

Der zugehörige Graph G_2 kann erstellt werden, indem für jede Komponente ($\langle component \rangle$) und jeden Operator ($\langle operator \rangle$) ein Knoten erstellt wird, der eindeutig anhand des Labels identifiziert werden kann. Die dadurch entstehende Menge von Knoten heißt $V_2 = \{v_1, .., v_i\}$ mit $i \in \mathbb{N}$. Die Kanten werden passend zu jedem Term ($\langle expression \rangle$) erstellt, wobei angenommen wird, dass $T = \{t_1, .., t_j\}$ mit $j \in \mathbb{N}$ die Menge aller Terme darstellt und jeder Term aus einem Tupel besteht: $t_x = (op_x, V_{2,x})$ mit $x \in \{1, .., j\}$. op_x entspricht dem Knoten des zugehörigen Operators. $V_{2,x}$ entspricht der Teilmenge von V_2 die alle Knoten beinhaltet, die zu Komponenten des jeweiligen Terms gehören. Für die Menge der Kanten (E_2) des Graphen passend zur Spezifikation gilt somit:

$$E_2 = \{(v_1, v_2)\ \mid\ \forall x \in \{1, .., j\}\ \ und\ \ \forall v \in V_{2,x} : v_1 = op_x, v_2 = v\}$$

Sollte dadurch kein zusammenhängender Graph entstehen, werden die Wurzeln der einzelnen Teilgraphen durch *and*-Knoten miteinander verbunden.
Dies ist der Fall, wenn die Spezifikation der Custom Instruction mehrere boolesche Funktionen im Funktionsrumpf trägt. Dann existieren mehrere Knoten, denen keine eingehende Kante innerhalb von E_2 zugeordnet werden kann. Die Menge dieser Knoten ist:

$$OP_2 = \{op_x\ \mid\ x \in \{1, .., j\}\ \ und\ \ \forall v \in V : (v, op_x) \notin E_2\}$$

Die Menge V_2 wird dementsprechend um $|OP_2| - 1$ *and*-Knoten erweitert, wobei der erste *and*-Knoten mit dem ersten Element aus OP_2 und dem nächsten *and*-Knoten über eine Kante verbunden wird. Dieser *and*-Knoten wiederum wird mit dem zweiten Element aus OP_2 und dem nächsten *and*-Knoten verbunden. Dieser Vorgang wird fortgesetzt, bis $|OP_2| - 1$ *and*-Knoten verwendet wurden. Der zuletzt betrachtete *and*-Knoten wird zusätzlich mit dem letzten Element aus OP_2 verbunden. Alle Kanten, die auf diese Weise erstellt werden, werden zu E_2 hinzugefügt.

Auf die gleiche Art kann der Graph G_1 entsprechend der Beschreibung der Custom Instruction aus dem Bedingungsdokument erstellt werden.

Wenn die Struktur von Beschreibung und Spezifikation übereinstimmt, entstehen auf diese Weise zwei deckungsgleiche Graphen, die als Bäume interpretiert werden können. Als Wurzeln dieser Bäume werden die Knoten definiert, die keine eingehenden Kanten besitzen. Anhand der Deckungsgleichheit kann jedem Knoten des einen Baums ein Knoten des anderen Baums zugewiesen werden. Unter anderem

kann somit den Knoten, die den Komponenten bzw. Variablen der Beschreibung entsprechen, jeweils ein Knoten, der einer Variablen der Spezifikation entspricht, zugeordnet werden.
Insgesamt bedeutet das, dass aus diesen beiden Bäume direkt das gesuchte Matching abgeleitet werden kann.

3.2.3 Beispiel (Teil 2/3): Erstellen des Matchings

Die Custom Instruction Beschreibung aus dem Bedingungsdokument (siehe Codeausschnitt 3.1) lautet:
(**define-fun** ci_1() **Bool** (= d a))
(**define-fun** ci_2() **Bool** (and (> b f) (> c f)))
(**define-fun** ci() **Bool** (**and** ci_1 ci_2))

Durch den Vergleich der Spezifikation des Beispiels aus Abschnitt 3.2.1 mit dieser Beschreibungen wird deutlich, dass die Variablen der beiden Komponenten unterschiedlich benannt wurden. Es existiert beispielsweise innerhalb der Spezifikation eine Variable mit dem Namen „e“, nicht aber in der Beschreibung aus dem Bedingungsdokument. Der Grund dafür ist, dass das Bedingungsdokument vom Analyse-Tool generiert wurde und die Variablen unabhängig von der Spezifikation benannt wurden.

Abbildung 3.2 zeigt die Graphen passend zur Spezifikation und zur Beschreibung des Beispiels. Die Sortierung der Knoten von links nach rechts entspricht dem Auftreten der Variablen, Konstanten und Operatoren.
Die äquivalenten Bäume sind in Abbildung 3.3 abgebildet. Die rot gestrichelten Pfeile stellen das Matching dar.

Das Matching kann somit passend zum Beispiel ermittelt werden (siehe Tabelle 3.3). Außerdem kann allen Variablen, mit Hilfe des Matchings, ein Wertebereich auf einem der Eingangsvektoren zugeordnet werden, da diese für die Variablen der Spezifikation bereits bekannt sind (siehe 3.2.1). Es wird angenommen, dass in diesem Fall jeder Integer-Wert genau 32 Bit nutzt.

Stelle in Spezifikation **Variable in Spezifikation**	1 a	2 b	3 c	4 d	5 e
Variable in Beschreibung (Bedingungsdokument)	d	b	c	a	f
Eingangsvektor **Bereich**	1 [0..31]	1 [32..63]	1 [64..95]	2 [0..31]	2 [32..63]

Tabelle 3.3: Matching der Variablen und Zuordnung der Wertebereiche

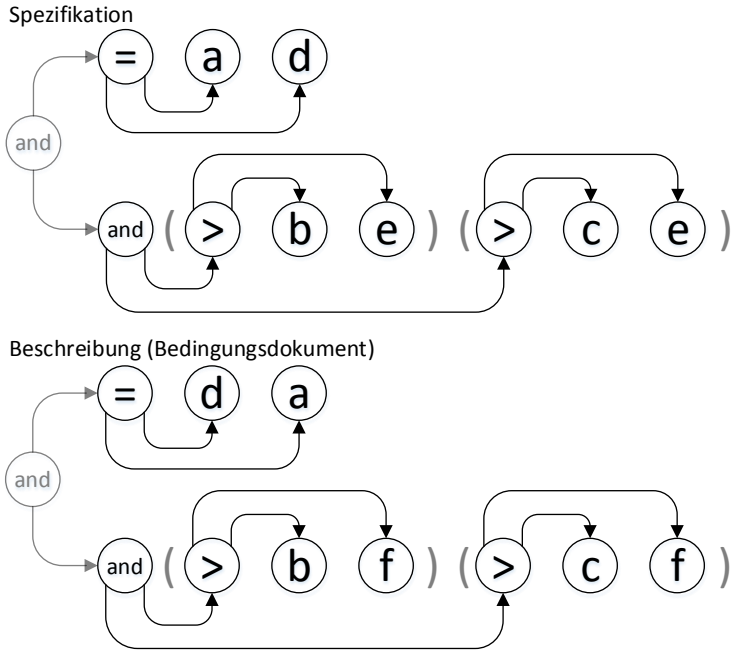

Abbildung 3.2: Graphen passend zu Spezifikation und Beschreibung

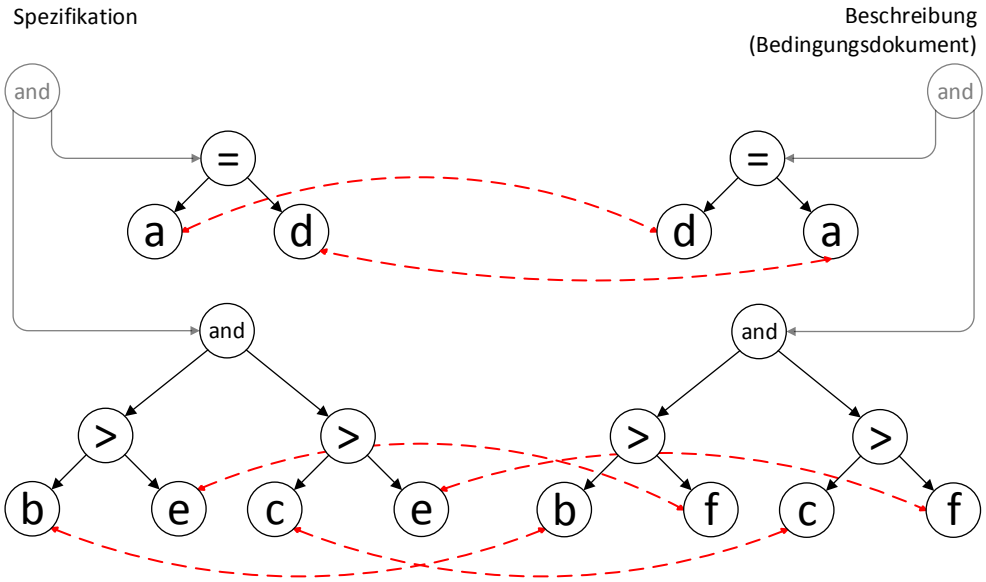

Abbildung 3.3: Umgewandelte Bäume + Matching

3.3 Transformationsregeln

Zur Generierung eines Eigenschaftsprüfers werden die Vor- und Nachbedingungen aus einem Bedingungsdokument in ein Verilog-Modul umgewandelt. Dieses Modul wird mit Hilfe des Matchings mit der Konfiguration verschaltet (siehe Ab-

bildung 2.4 – Kapitel 2.1.3). Wie diese Umwandlung und Verschaltung abläuft, wird anhand der Transformationsregeln festgelegt, die in diesem Kapitel vorgestellt werden.

Mittels dieser Transformationsregeln ist es möglich jedes Wort der Teilsprache $L(G)$ in Verilog zu übersetzen und somit aus einem beliebigen Bedingungsdokument einen Eigenschaftsprüfer in Form eines Verilog-Moduls zu generieren. Das generierte Verilog-Modul wird zusammengesetzt aus den folgenden Verilog-Befehlen, deren Funktion bereits in Abschnitt 2.4 vorgestellt wurde:

assign, endmodule, module, parameter, wire

Diese Befehlsmenge reicht aus, um einen Eigenschaftsprüfer in Form eines Verilog-Moduls zu definieren. Durch *wire* und *assign* definierte Verschaltungsregeln werden im Folgenden wie Variablen behandelt. Wann immer also von einer Verilog-Variablen und dessen Wert gesprochen wird, ist damit eine Verschaltungsregel und der anliegende Wert gemeint.

Alle Transformationsregeln basieren auf den Produktionsregeln der Grammatik G (siehe Grammatik 3.2), wobei zu jeder Produktionsregel eine Transformationsregel gehört.

Einige Produktionsregeln (p4, p6 - indirekt, p7, p8, p9, p10, p11) dienen ausschließlich der Abbildung auf Terminalsymbole. Die entsprechenden Transformationsregeln werden im Folgenden nicht aufgelistet, weil die Funktion dieser Regeln darauf beschränkt ist, die bereitgestellten Informationen durchzureichen. Wird beispielsweise mittels der Produktionsregel p4 die Information über den Datentyp einer Variablen bereitgestellt, so wird diese Information durch die zugehörige Transformationsregel an die übergeordnete(n) Transformationsregel(n) weitergeleitet. Welche Informationen passend zu den zuvor aufgezählten Produktionsregeln weitergereicht werden, ist in Tabelle 3.4 dargestellt.

Mit Beginn und Ende der Anwendung von Produktionsregel p1 startet bzw. endet die Generierung des Eigenschaftsprüfers, da diese Regel einerseits das Startsymbol verarbeitet und anderseits erst vollständig abgeleitet wird, wenn auch der letzte Befehl verarbeitet wurde.
Im Folgenden werden alle zur Generierung benötigten Transformationsregeln aufgelistet.

Startregel (t1)

Die Transformationsregel t1 wird beim Start der Generierung aufgerufen, da sie auf p1 basiert und p1 das Startsymbol verarbeitet. Mittels dieser Regel wird im Verilog-Code festgelegt, dass ein Modul mit dem Namen „Eigenschaftspruefer“ erstellt wird. Dieses Modul besitzt mindestens zwei Eingänge (*in*, *out*) und einen Ausgang (*error*). Der Eingangsvektor *in* entspricht dem Input der Konfiguration, *out* dem Output der Konfiguration und *error* dem Fehlerbit (siehe Abbildung 2.4

Produktionsregel	**Nicht-Terminal**	**Durchgereichte Information**	**Übergeordnete Transformationsregel(n)**
p4	⟨*sort*⟩	**Datentyp**: Bool oder Int	t2, t4
p6	⟨*spec_constant*⟩	**Konstantentyp**: Zahl oder Wahrheitswert	t3
p7	⟨*symbol*⟩	Angepasster **Bezeichner** (siehe Sonderregel s2)	t2, t3, t4
p8, p9, p10	⟨*numeral*⟩, ⟨*hexadecimal*⟩, ⟨*binary*⟩	**Wert einer Konstanten**: Positive, natürliche Zahl	t3
p11	⟨*bool*⟩	**Wert einer Konstanten**: Wahrheitswert: 1 (true/wahr) oder 0 (false/falsch)	t2, t3, t4

Tabelle 3.4: Durchgereichte Informationen

– Kapitel 2.1.3). Existieren Variablen, die ausschließlich in den Bedingungen genutzt werden, kommt ein weiterer Eingangsvektor hinzu (*extra*).
Aus der Signatur der Spezifikation, die auch zur Erstellung des Matchings genutzt wird, ergeben sich die Größen von *in* und *out*. Angenommen in der Signatur der Spezifikation wurden $x \in \mathbb{N}$ eingehende Variablen festgelegt, dann bietet der Eingangsvektor (*in*) Platz für x Werte von Variablen. Außerdem gilt $x = y + z$, wobei y der Anzahl Integer-Variablen und z der Anzahl boolescher Variablen entspricht. y und z werden mit Hilfe des Matchings und der Variablen-Definitionen im Bedingungsdokument ermittelt. Dementsprechend entspricht die genaue Größe von *in* der Summe aus z und dem Produkt von x und Parameter N.
Der Parameter N entspricht der Anzahl Bits, die pro Integer-Wert genutzt werden können und wird ebenfalls durch Regel t1 festgelegt. Standardmäßig ist $N = 32$ vorgesehen. Das implementierte Tool bietet dem Benutzer jedoch die Möglichkeit, diesen Wert selbst festzulegen.
In gleicher Weise wird die Größe von *out* anhand der ausgehenden Variablen ermittelt.
Der Ausgang *error* hat immer die Größe 1.

Beispiel

```
1 module Eigenschaftspruefer(in, out, extra, error);
2         parameter N = 32;
3
4         input [2*N-1:0] in;
5         input [N-1:0]   out;
```

```
6           input [3*N-1:0] extra;
7           output error;
```

Im Beispiel bietet der Eingangsvektor *in* zwei Werten von Integer-Variablen Platz, *out* nur einem Wert. Beispielhaft wurde angenommen, dass drei weitere Variablen innerhalb der Bedingungen, nicht aber innerhalb der Custom Instruction genutzt werden. Daher wurde der dritte Eingangsvektor *extra* eingefügt, der drei Integer-Variablen Platz bietet.

Transformation von declare-fun und define-fun (t2)

Die Transformationsregel t2 basiert auf der Produktionsregel p2 und greift auf Informationen zu, die durch andere Transformationsregeln bereitgestellt werden. Zum Beispiel tritt das Nichtterminal ⟨*symbol*⟩ innerhalb von p2 auf. Die Transformationsregel, die zu ⟨*symbol*⟩ bzw. zu Produktionsregel p7 gehört, gibt die Terminalsymbole, die ⟨*symbol*⟩ zugeordnet wurden, an t2 weiter. Auf die gleiche Art wird der jeweilige Datentyp passend zu ⟨*sort*⟩ an t2 übergeben.

Sobald die Produktionsregel p2 angewandt wird, wird eine Verilog-Variable erstellt, die entweder der Variablen, die durch *declare-fun*, oder dem Term, der durch *define-fun* festgelegt wurde, entspricht. Im ersten Fall (siehe Beispiel 1) wird der erstellten Verilog-Variablen ein Wert entsprechend des Matchings zugewiesen.

Beispiel 1

```
1           wire [N-1:0] x1;
2           assign x1 = in[2*N-1:N];
```

Dem Wire $x1$ wird der Wert an Position 2 auf dem Eingangsvektor *in* zugeordnet.

Im zweiten Fall (siehe Beispiel 2) wird der Verilog-Variablen ein Term zugewiesen. Terme werden verkörpert durch Verilog-Variablen, die bei der Anwendung von Transformationsregel t3 erstellt werden.

Beispiel 2

```
1           wire [N-1:0] x2;
2           assign x2 = term__1;
```

Der Verilog-Variablen $x2$ wird ein Term zugewiesen, der sich hinter der Verilog-Variablen *term__1* verbirgt.

In beiden Fällen wird einmal der Befehl *wire* und einmal der Befehl *assign* verwendet. Abbildung 3.4 zeigt einen Graphen, der demonstriert, wie die weitergeleiteten Informationen von untergeordneten Transformationsregeln verarbeitet werden. Jede Kante entspricht der Verarbeitung einer Information, wobei das Label der jeweiligen Kante angibt, welche Information verarbeitet wurde. Die Knoten zeigen das entsprechende Resultat.

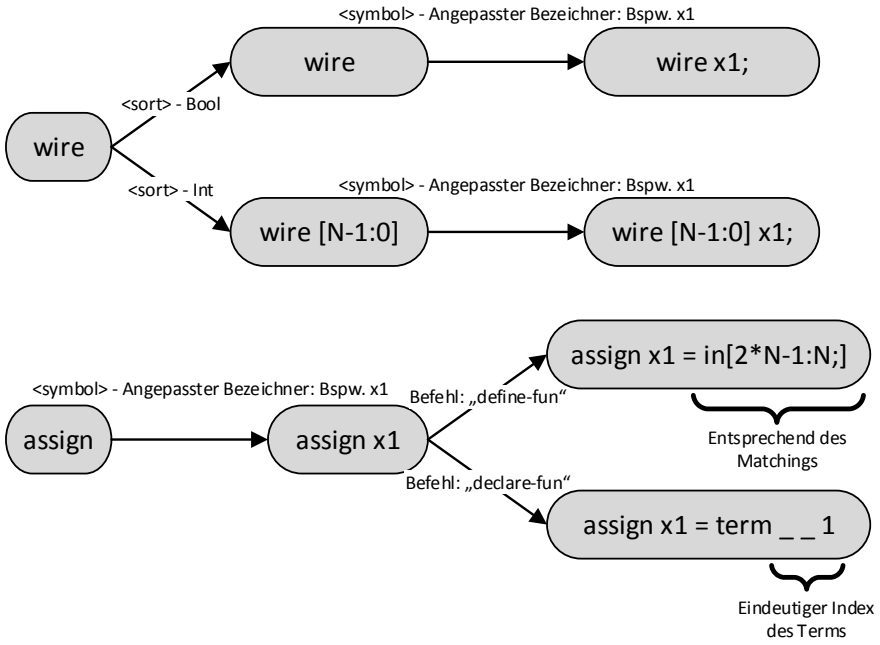

Abbildung 3.4: Verarbeitung weitergeleiteter Informationen

Transformation von Termen, Variablen und Konstanten (t3)

Durch die Anwendung der Produktionsregel p3 wird t3 ausgelöst, wodurch eine Verilog-Variable erstellt wird. Diese Verilog-Variable wird eindeutig benannt ($term\_\_x$ mit $x \in \mathbb{N}$) und mit einer Konstanten, einer Variablen oder einem Term verbunden, entsprechend der jeweiligen Alternative aus p3.
Wird $\langle term \rangle$ mittels p3 auf $\langle spec\_constant \rangle$ bzw. $\langle symbol \rangle$ abgebildet, heißt das, dass dem neu erstellten Wire eine Konstante bzw. eine Variable zugewiesen wird. Welche Konstante/Variable das ist, wird durch eine untergeordnete Transformationsregel an t3 weitergeleitet.
Wird die dritte Alternative innerhalb von p3 verwendet, so wird dem Wire ein geklammerter Term zugeordnet. Die innerhalb dieses Terms verwendeten Komponenten werden durch weitere Aufrufe der Regel t3 an diesen Aufruf von t3 übergeben.

Beispiel

```
1        // Konstante: (- 1)
2        wire [N-1:0] term__1;
3        assign term__1 = -1;
4
5        // Variable: x
6        wire [N-1:0] term__2;
7        assign term__2 = x;
8
```

```
9           // Term: term__1 * term__2
10          wire [N-1:0] term__3;
11          assign term__3 = (term__1 * term__2);
12
13          wire [N-1:0] y;
14          assign y = term__3;
```

Im Beispiel wurde der folgende Teilsprachenbefehl umgewandelt:

(**define-fun** $y()$ **Int** $(* \; x \; (- \; 1)))$

Dabei wurde drei mal t3 (Zeile 1-11) und einmal t2 (Zeile 13, 14) angewandt.

Transformation von lokalen Variablen (t4)

Sobald lokale Variablen im Bedingungsdokument verwendet werden, wird die Regel t4 aufgerufen (Auslösende Produktionsregel: p5).
Die Regel t4 entspricht der Regel t2 mit dem Unterschied, dass hier nur (lokale) Variablen und keine Terme transformiert werden. Infolge der Verwendung einer lokalen Variablen muss die generierte Verilog-Variable speziell markiert werden, damit diese nur im darauffolgenden Term verwendet werden kann. Dies wird realisiert, indem der Ausdruck *__inner* an den Namen des Wires angehängt wird.

Beispiel

```
1           wire [N-1:0] x__inner;
2           assign x__inner = extra[N-1:0];
```

Im Beispiel ist die lokale Variable x die erste zusätzliche Variable und befindet sich daher auf Position 1 des dritten optionalen Eingangsvektors *extra*.

Abschlussregel (t5)

t5 wird am Ende der Generierung aufgerufen und dient ausschließlich dazu, die Vor- und Nachbedingungen zu verschalten und das Modul zu beenden. Da das Fehlerbit den Wert 1 bzw. *true* liefern soll, falls eine Nachbedingung nicht aus der zugehörigen Vorbedingung folgt, wird dem Fehlerbit die Negation der Implikation $pre \rightarrow post$ zugewiesen. Außerdem wird die Implikation umgewandelt in $\neg pre \vee post$, da Verilog keine Implikationen unterstützt.

Beispiel

```
1           // Ein Paar von Bedingungen
2           assign error = ~(~pre | post);
3
4           // Mehrere Paare von Bedingungen
5           // (x entspricht einer positiven natürlichen Zahl)
6           assign error = ~(
```

```
7                                   (~pre_1 | post_1) &
8                                   (~pre_2 | post_2) &
9                                   ... &
10                                  (~pre_x | post_x)
11                              );
12  endmodule;
```

Das Beispiel besteht aus 2 Teilen: Zeile 1, 2 + 12 zeigen das Ergebnis der Regel t5 für nur ein Paar von Vor- und Nachbedingungen. Zeile 4-12 hingegen stellt das entsprechende Ergebnis für mehrere Paare dar.

Diese 5 Transformationsregeln genügen, um aus einem Bedingungsdokument einen Eigenschaftsprüfer zu generieren.

3.3.1 Sonderregeln

Neben diesen 5 Transformationsregeln, werden drei Sonderregeln festgelegt, die der Optimierung des Verilog-Codes und der Vermeidung von Fehlern während der Transformation dienen.

Entfernen überflüssiger Informationen (s1)

Alle zur Beschreibung der Custom Instruction gehörenden Informationen werden nur zur Erstellung des Matchings benötigt und müssen nicht in Verilog übersetzt werden bzw. im Eigenschaftsprüfer vorhanden sein.

Konvertierung der Variablennamen und Operatoren (s2)

Da Verilog andere Eigenschaften besitzt als die Teilsprache $L(G)$ bzw. als SMT-Lib, muss folgendes gelten:

- Die Variablennamen dürfen nach der Übersetzung nur die Zeichen „0“-„9“, „a“-„z“, „A“-„Z“ und „_“ enthalten. Außerdem muss jeder Variablenname mit einem Buchstaben oder dem Unterstrich beginnen [Hop06]. Daher werden alle nicht erlaubten Zeichen und doppelte Unterstriche in „S“ wie Sonderzeichen umgewandelt. Sollten durch die Umwandlung zwei gleiche Variablennamen entstehen, wird der umgewandelte Variablenname um einen Index erweitert, der solange erhöht wird bis der Variablenname einzigartig ist.

- Die in den Termen enthaltenen Operatoren müssen konvertiert werden:

Herkunft der Verilog-Variablen (s3)

Jede Variable des Bedingungsdokumentes, die nicht anhand des Matchings zugeordnet werden kann oder ausschließlich zu den Bedingungen gehört, wird bei der

SMT/Teilsprache	**Verilog**
and	&
or	|
not	$\sim$
=	==
div	/
mod	%
distinct	!=

Tabelle 3.5: Operatoren SMT & Verilog

Transformation in eine Verilog-Variable mit einem Index versehen. Dieser Index kann einer Datei des Bedingungsdokumentes zugeordnet werden; dabei gehört zur ersten Datei der Index 0, zur zweiten der Index 1 und so weiter. Dies vermeidet die Doppelbelegung von Variablennamen.

Beispielhaft angenommen die Interpretation von Datei 1 führt zur Definition einer zusätzlichen Verilog-Variablen mit dem Namen x und auch die Interpretation von Datei 2 führt zur Definition einer zusätzlichen Verilog-Variablen mit dem Namen x. Beide Variablen haben jedoch nichts miteinander gemeinsam. Damit beide auch nach der Transformation unterschieden werden können, wird jeweils ein Index angehängt:
x aus Datei 1 wird zu $x\_\_0$ und x aus Datei 2 zu $x\_\_1$.

3.3.2 Beispiel (Teil 3/3): Generierung des Eigenschaftsprüfers

Zusammen mit dem Matching (siehe Tabelle 3.3), das im vorherigen Kapitel erstellt wurde, können die Bedingungen des Bedingungsdokumentes (siehe Codeausschnitt 3.1) anhand der beschriebenen Transformationsregeln in einen Eigenschaftsprüfer transformiert werden:

```
1  module Eigenschaftspruefer(in, out, error);
2
3          parameter N = 32;
4
5          input [3*N-1:0] in;
6          input [2*N-1:0] out;
7          output error;
8
9          wire [N-1:0] b;
10         wire [N-1:0] c;
11         wire [N-1:0] e;
12         wire term__9;
13         wire pre__0;
14         wire term__11;
15         wire pre_1__0;
16         wire term__13;
17         wire post__0;
18         wire term__15;
19         wire post_1__0;
20
21         assign b = in[2*N-1:1*N];
22         assign c = in[3*N-1:2*N];
23         assign e = out[2*N-1:1*N];
24
25         assign term__9 = (b > 0);
26         assign pre__0 = term__9;
27         assign term__11 = (c > 0);
28         assign pre_1__0 = term__11;
29         assign term__13 = (e > b);
30         assign post__0 = term__13;
31         assign term__15 = (e > c);
32         assign post_1__0 = term__15;
33
34         assign error = !(
35                 (!pre__0 || post__0) &&
36                 (!pre_1__0 || post_1__0)
37         );
38
39 endmodule
```

Codeausschnitt 3.2: Eigenschaftsprüfer

Die Generierung beginnt durch die Anwendung der Transformationsregel t1. Darauf werden wiederholt die Regeln t2 und t3 angewandt. Die Anwendung dieser Regeln wird für Zeile 9 des Bedingungsdokumentes im Folgenden genauer betrachtet:

1)	(define-fun pre() Bool (> b 0))	$\xrightarrow{t2}$	wire pre__0; assign pre__0 = term__9;

Durch Anwendung der Regel t2 wird der *define-fun* Befehl verarbeitet. Dazu wird eine Verilog-Variable mit dem Namen der definierten Funktion *pre__0* erstellt, deren Größe 1 ist, da ihr Datentyp als *Bool* definiert wurde. In diesem Fall wird der Variablen ein Term zugewiesen. Die dabei im Namen des Terms verwendete Nummer wird bei jeder Termdefinition um 1 erhöht, wodurch jeder Term eindeutig identifiziert werden kann.

2)	(> b 0)	$\xrightarrow{t3}$	wire term__9; assign term__9 = (b > 0);

Durch Transformationsregel t3 wird der eigentliche Term transformiert, indem zunächst eine Verilog-Variable für den Term erstellt wird. Dieser Variablen wird der Term zugeordnet. Als Voraussetzung für die Transformation in Verilog wird die Präfix-Notation in eine Infix-Notation umgewandelt.

1 & 2)	(define-fun pre() Bool (> b 0))	$\xrightarrow{t2,t3}$	wire term__9;
			wire pre__0;
			assign term__9 = (b > 0);
			assign pre__0 = term__9;

Beide Regeln zusammen führen zum obigen Ergebnis (siehe Zeile 12, 13 + 25, 26 im Eigenschaftsprüfer).

Da keine lokalen Variablen im Bedingungsdokument auftreten, wird t4 nie angewandt. Außerdem ist anzumerken, dass die Variablen a und d im Eigenschaftsprüfer nicht auftauchen. Zurückzuführen ist dies darauf, dass diese Variablen nur in der Beschreibung bzw. Spezifikation der Custom Instruction, nicht aber in den Bedingungen enthalten sind. Nach der Ermittlung des Matchings werden diese Variablen dementsprechend nicht mehr benötigt.

Beendet wird die Generierung durch Anwendung der Regel t5.
Implizit wurden die zuvor genannten Sonderregeln (siehe 3.3.1) angewandt.

4 Implementierung

Das Tool zur Generierung von Eigenschaftsprüfern, das für diese Arbeit implementiert wurde, setzt das Konzept aus Kapitel 3 um. Die verwendete Programmiersprache ist Java und als Entwicklungsumgebung wurde Eclipse gewählt.

Während der Generierung eines Eigenschaftsprüfers durchläuft das Tool drei Phasen. Die erste Phase dient dem Einlesen der gegebenen Informationen (Bedingungsdokument und Spezifikation der Custom Instruction). In der zweiten Phase wird das Matching aus den eingelesenen Informationen generiert. Zuletzt werden in Phase drei alle Informationen zusammengefügt. Dementsprechend wird aus den eingelesenen Daten und dem ermittelten Matching ein Eigenschaftsprüfer gemäß der Transformationsregeln generiert.
Abbildung 4.1 spiegelt den gesamten Ablauf dieser drei Phasen wieder.

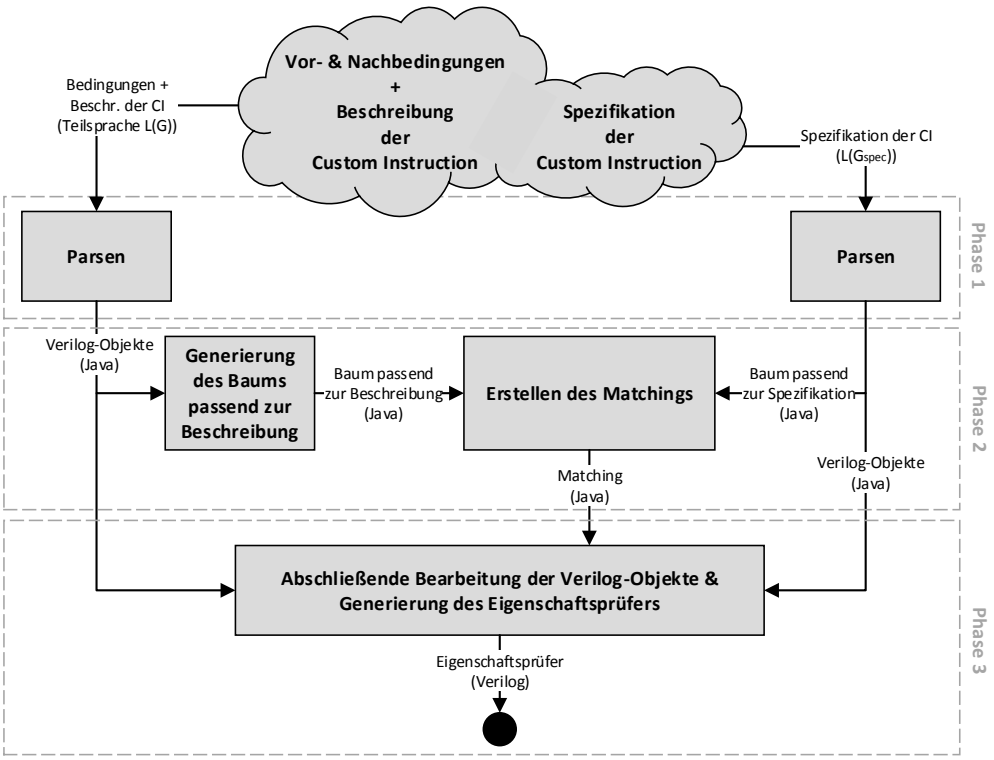

Abbildung 4.1: Workflow

Phase 1) Parsen - Einlesen der gegebenen Informationen

Die benötigten Informationen zur Generierung werden in Form von mindestens zwei Dateien an das Tool übergeben. Eine Datei enthält die Spezifikation der Custom Instruction. Das Bedingungsdokument wird in Form von mindestens einer Datei übergeben. Jede Datei, die zum Bedingungsdokument gehört, enthält mindestens ein Bedingungspaar bestehend aus einer Vor- und einer Nachbedingung und eine Beschreibung der Custom Instruction.
Die erste Aufgabe des Tools besteht darin, diese Dateien einzulesen und auszuwerten. Dazu wurden zwei Parser mit Hilfe des Tools JavaCC[1] entwickelt. Durch JavaCC ist es möglich, einen Parser direkt aus der Grammatik einer Sprache zu generieren. In diesem Fall heißt das, dass der erste Parser (zum Einlesen des Bedingungsdokumentes) aus der Grammatik der Teilsprache generiert werden kann. Für die Generierung des Parsers für Spezifikationen von Custom Instructions kann die im Konzept entwickelte Grammatik zur Darstellung von Spezifikationen verwendet werden (siehe Anhang A.1).
Die Generierung der Parser, bietet zwei entscheidende Vorteile im Vergleich zu handgeschriebenen Parsern:

> First, it substantially reduces the time required to produce a compiler. Second, it allows us to produce more reliable compilers. The majority of bugs in hand-written compilers are introduced during the tedious process of computing selection sets and grinding out the code from the translation grammar. [Rei]

Aus den eingelesenen Vor- und Nachbedingungen werden anhand der im Konzept festgelegten Transformationsregeln Verilog-Objekte erstellt. Jedes dieser Objekte entspricht einem oder mehreren Verilog-Befehlen und kann durch den Aufruf der Funktion $toVerilog()$ in Verilog-Quellcode umgewandelt werden. Dementsprechend repräsentieren diese Objekte bereits unvollständige Teile des Eigenschaftsprüfers. Innerhalb der nächsten zwei Phasen werden die Lücken in und zwischen diesen Teilen geschlossen.
Aufgrund der eingelesenen Daten aus der Spezifikationsdatei werden ebenfalls zwei Verilog-Objekte erzeugt. Diese definieren zwei der Eingangsvektoren des Eigenschaftsprüfers (in, out), die Platz für ebenso viele Variablen bieten, wie in der Signatur der Spezifikation festgelegt sind.

Phase 2) Erstellen des Matchings

Gemäß des Konzeptes werden zum Erstellen des Matchings zwei Bäume benötigt. Ein Baum, basiert auf der Beschreibung, der andere auf der Spezifikation der Custom Instruction. Der Baum passend zur Spezifikation wird direkt aus den Informationen erzeugt, die aus der Spezifikationsdatei über den Funktionsrumpf

[1] Java Compiler Compiler ™(JavaCC ™) - The Java Parser Generator (https://javacc.java.net/)

hervorgehen. Der zur Beschreibung passende Baum hingegen kann aus den zuvor erstellten Verilog-Objekten abgelesen werden. Dazu wird anhand des Schlüsselwortes „ci“ ermittelt, welche Verilog-Objekte zur Beschreibung der Custom Instruction gehören. Entsprechend des im Konzept beschriebenen Ablaufs ermittelt das Tool ein Matching zwischen den Variablen der Beschreibung und denen der Spezifikation. Nachdem das Matching erstellt wurde, können alle Verilog-Objekte, die zur Beschreibung der Custom Instruction gehörten, entfernt werden.

Phase 3) Generierung des Eigenschaftsprüfers

In Phase 3 findet die eigentliche Generierung des Eigenschaftsprüfers statt. Dazu wird eine Verilog-Datei erzeugt, die das Verilog-Modul, das den Eigenschaftsprüfer repräsentiert, enthält.
Bevor diese Verilog-Datei auf Basis der zuvor erstellten Verilog-Objekte generiert werden kann, müssen diese Objekte abschließend bearbeitet werden. Zur abschließenden Bearbeitung gehört einerseits, dass den Variablen Bereiche auf den Eingangsvektoren entsprechend des Matchings zugeordnet werden und anderseits, dass die in Abschnitt 3.3.1 betrachteten Sonderregeln umgesetzt werden.

4.1 Aufbau des Tools

Das in Abbildung 4.2 gezeigte Klassendiagramm spiegelt den Aufbau der Implementierung wieder.
Einige Klassen wurden im Diagramm nicht abgebildet und teilweise enthalten die Klassen innerhalb der Implementierung mehr Attribute und/oder Funktionen, als im Diagramm dargestellt. Diese nicht abgebildeten Informationen werden ausschließlich von Objekten einer Klasse benutzt und sind für den Gesamtablauf und das Zusammenspiel zwischen den Klassen nicht von Interesse. Weggelassen wurden diese Informationen damit das Diagramm überschaubar bleibt, sodass die Struktur und die Funktion des Tools am Diagramm erklärt werden kann.

Die Klasse *EPGenerator* enthält die *main*-Methode. Durch den Aufruf dieser Methode startet die Generierung des Eigenschaftsprüfers. Die Klassen *Parser* und *SpecificationParser* stellen die durch JavaCC generierten Parser dar. Neben diesen Klassen werden noch weitere Klassen von JavaCC generiert, die für die korrekte Funktion des Parsers benötigt werden, aber im Gesamtablauf keine Rolle spielen. Passend zu beiden Parsern wurden zwei Klassen implementiert ($ParserOutputHandler$, $SpecificationParserOutputHandler$), die die vom Parser eingelesenen Informationen interpretieren und verarbeiten. Alle bisher betrachteten Klassen werden für die Umsetzung der ersten Phase benötigt.

In der zweiten Phase wird das Matching bestimmt; dazu werden zwei Bäume mittels der Klasse *MTree* abgebildet. Diese Bäume bestehen aus einer Verkettung von Knoten, die durch Klasse *MNode* dargestellt werden. Mittels der Methode

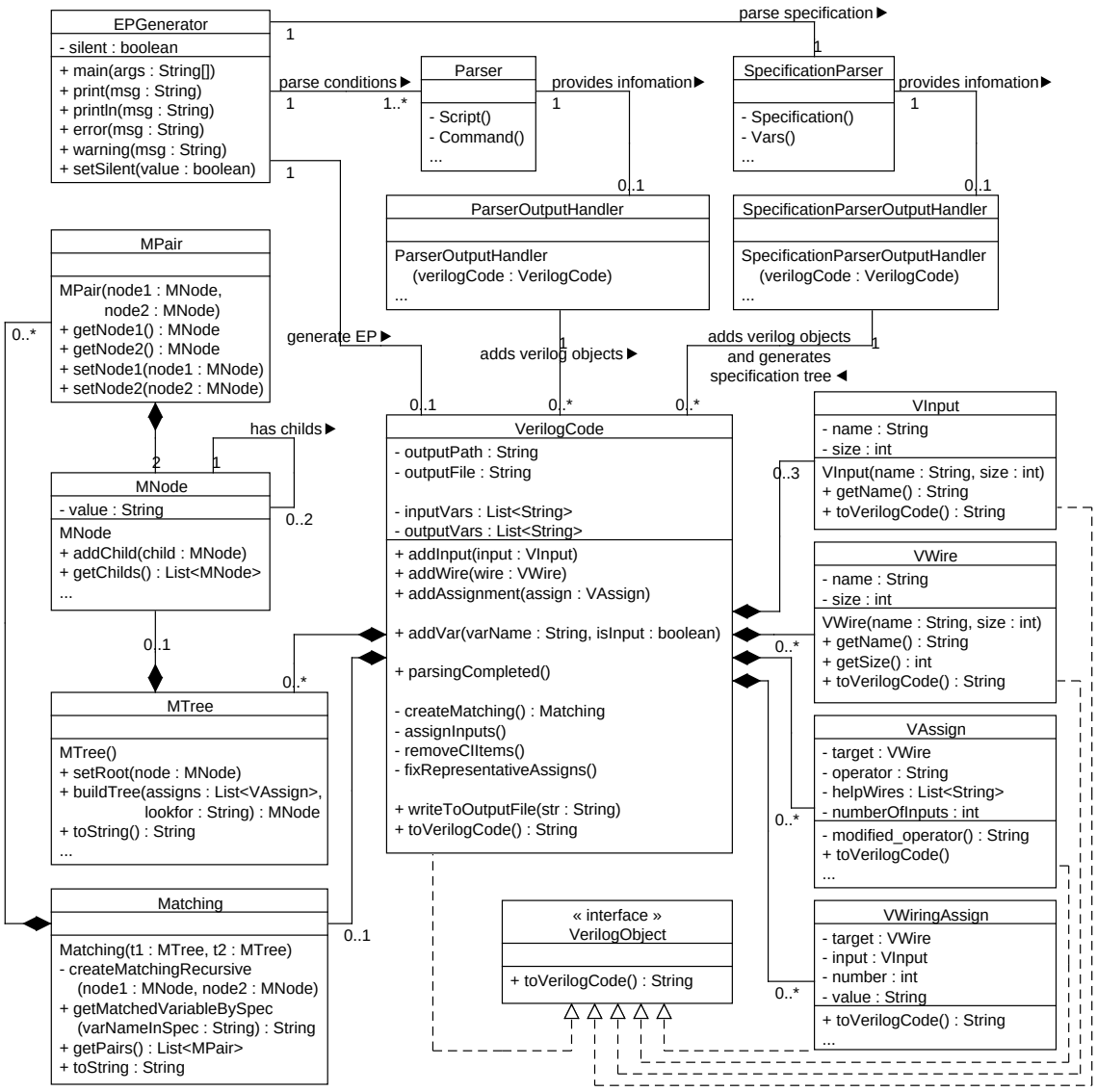

Abbildung 4.2: Klassendiagramm

buildTree wird z.B. ein Baum anhand der verfügbaren Verilog-Objekte erstellt. Der dadurch entstehende Baum passt zur Beschreibung der Custom Instruction. Das eigentliche Matching wird durch die Klasse *Matching* repräsentiert. Ein Objekt der Klasse *Matching* besteht dabei aus beliebig vielen Paaren (*MPair*) von Knoten (*MNode*). Durch die Funktion *createMatchingRecursive* wird das Matching erstellt ausgehend von den Wurzeln zweier Bäume.

Für die Umsetzung von Phase 3 wird ausschließlich ein Objekt der Klasse *VerilogCode* benötigt. Diese Klasse dient der Abbildung des Eigenschaftsprüfers. Alle erstellten Verilog-Objekte stehen dieser Klasse zur Verfügung. Außerdem hat diese Klasse Zugriff auf das Matching. Die Attribute *outputPath* und *outputFile* werden von der Funktion *writeToOutputFile* genutzt, die dafür sorgt, dass das Verilog-

Modul in einer Datei gespeichert wird, die mittels der Attribute spezifiziert wurde. Das Attribut $inputVars$ entspricht einer Liste, die die Namen aller eingehenden Variablen der Konfiguration enthält. $outputVars$ enthält äquivalent dazu alle Namen der ausgehenden Variablen. Über die Funktionen $addInput$, $addWire$ und $addAssignment$ werden einem Objekt der Klasse *VerilogCode* Verilog-Objekte angehängt, die jeweils mit einem Verilog-Befehl assoziiert werden können. Zum Beispiel dient die Klasse $VWire$ der Abbildung des Befehls „wire“. Die Funktion $parsingCompleteted$ wird aufgerufen, sobald das Parsen der Dateien abgeschlossen ist. Infolge des Aufrufs dieser Funktion wird eine Reihe weiterer Funktionen (*createMatching, assignInputs, removeCIItems, fixRepresentativeAssigns*) aufgerufen, die der Verarbeitung des Matchings und der Umsetzung der Sonderregeln (siehe Abschnitt 3.3.1) dienen.

Alle Verilog-Objekte und auch die Klasse *VerilogCode* implementieren das Interface *VerilogObject* und besitzen daher eine Funktion $toVerilogCode$. Mittels dieser Funktion können Verilog-Objekte direkt in Verilog-Code umgewandelt werden. Dementsprchend wird auch der finale Eigenschaftsprüfer durch $toVerilogCode$ anhand eines Objektes der Klasse *VerilogCode* erstellt.

4.2 Benutzerhandbuch

Das Tool besteht aus einer ausführbaren Java Archive (*.jar*) Datei und kann von der Kommandozeile aus gestartet und genutzt werden.
Zwei Parameter werden bei der Ausführung des Programms als notwendig angesehen. Einerseits ist dies die Angabe einer Spezifikationsdatei und anderseits die Angabe mindestens einer Datei, die Teil des Bedingungsdokumentes ist. Die Angabe einer Datei erfolgt in Anführungszeichen und inklusive Pfad. Für den Fall, dass das Bedingungsdokument aus mehreren Dateien besteht, werden alle Dateien hintereinander durch Leerzeichen getrennt und ebenfalls in Anführungszeichen angegeben. In allen Fällen können die Anführungszeichen weggelassen werden, falls weder Dateiname noch Pfad Leerzeichen enthalten.
Der Eigenschaftsprüfer wird standardmäßig in eine Datei mit dem Namen „output.v“ geschrieben. Der Pfad zu dieser Datei stimmt mit dem Pfad zum Programm überein. Das folgende Beispiel reicht aus, um einen Eigenschaftsprüfer auf Basis der Spezifikationsdatei *spec.txt* und den Dateien des Bedingungsdokumentes *bed1.smt2* und *bed2.smt2* generieren zu lassen:

java -jar EPGenerator.jar "spec.txt" "bed1.smt2" "bed2.smt2"

Alle Dateien des Beispiels befinden sich im gleichen Verzeichnis wie das Programm selbst; in diesem Fall muss der Pfad nicht angegeben werden. Da das Tool aus einer ausführbaren Java Archive Datei besteht, braucht zu Beginn der Zeile kein „java -jar“ vorangestellt werden. Wird dieser Ausdruck nicht vorangestellt kann keine automatische Ausgabe auf der Kommandozeile erfolgen, wodurch keine Informationen über eventuelle Fehler und/oder Warnungen, die während der Generierung

auftreten, gegeben werden können. Auch die allgemeinen Informationen zur Generierung können dann nicht ausgegeben werden. Deshalb wird empfohlen, den Ausdruck „java -jar“ immer voranzustellen.

Das Tool bietet zahlreiche optionale Parameter, die den Umgang mit dem Tool erleichtern und die Interpretation der Ausgabe vereinfachen. Alle optionalen Parameter und ihre Bedeutung sind in der folgenden Tabelle 4.1 aufgeführt:

Parameter	**Bedeutung**
-help, -h, -?, -man, -manpage	Dient dem Aufruf eines Kurzhandbuches inklusive einer Auflistung aller optionalen Parameter. (Alle anderen Parameter werden ignoriert. Auch die Angabe der notwendigen Parameter ist hinfällig.)
-out ”X”, -o ”X”	Wird genutzt zur expliziten Angabe einer Ausgabedatei. (X entspricht dem Pfad und Dateinamen der Datei)
-path ”X”, -p ”X”	Ermöglicht die Angabe eines Pfades zu allen Dateien: Spezifikationsdatei, Datei(en) des Bedingungsdokumentes, Ausgabedatei. Die Angabe eines Pfades im einzelnen wird somit überflüssig. (X entspricht dem Pfad zu allen Dateien)
-bits X, -b X	Standardmäßig werden 32 Bit pro Integer-Wert vorgesehen. Mittels dieses Parameters kann eine andere Anzahl Bits gewählt werden. (X entspricht dieser Anzahl)
-silent, -s	Die Standardausgabe zu Beginn und am Ende der Generierung wird durch diesen Parameter unterbunden. Auch Warnungen werden durch diesen Befehl ignoriert. Ausschließlich über das Auftreten von Fehlern bei der Generierung wird weiterhin informiert.
-trees, -t	Durch diesen Parameter werden die zur Herstellung des Matchings verwendeten Bäume auf der Kommandozeile ausgegeben.
-matching, -m	Mittels dieses Parameters wird das bei der Generierung ermittelte Matching auf der Kommandozeile ausgegeben.
-extra, -e	Auf der Kommandozeile wird eine Liste aller Variablen ausgegeben, die in den Bedingungen enthalten sind - nicht aber in der Custom Instruction auftreten.
-gui	Dient dem Aufruf einer graphischen Benutzeroberfläche die den Umgang mit dem Tool erleichtert aber im Funktionsumfang eingeschränkt ist. (Alle anderen Parameter werden ignoriert. Auch die Angabe der notwendigen Parameter ist hinfällig.)

Tabelle 4.1: Optionale Parameter

4.3 Test: Umsetzung des Konzepts

Das implementierte Tool soll das im Kapitel 3 festgelegte Konzept umsetzen und Eigenschaftsprüfer anhand der dort festgelegten Transformationsregeln generieren.

Zwei generierte Eigenschaftsprüfer werden zur exemplarischen Überprüfung der Einhaltung der Transformationsregeln genutzt. Die Bedingungsdokumente[2], die zur Generierung dieser Eigenschaftsprüfer genutzt wurden, wurden wiederum durch CPAchecker generiert.

Im Folgenden wird anhand von zwei Zeilen aus einer Datei eines Bedingungsdokumentes die Überprüfung der Einhaltung der Transformationsregeln detailliert veranschaulicht:

```
1 (declare-fun |main::x@1| () Int)
2 (define-fun .def_135 () Int (* (to_int (- 1)) |main::x@1|))
```

Codeausschnitt 4.1: Anforderung1_ex19.smt

In der ersten Zeile wird der Befehl *declare-fun* genutzt, um eine Variable zu deklarieren. Durch den Vergleich der Beschreibung der Custom Instruction mit der Spezifikation kann dieser Variablen ($|main :: x@1|$) die Variable y der Spezifikation zugeordnet werden. Das heißt, dass erwartet wird, dass eine Verilog-Variable mit dem Namen y erstellt und mit dem ersten Wert des Eingangsvektors *out* belegt wird. Passend zu dieser Erwartung wurden durch das Tool die folgenden Codezeilen generiert:

```
1         wire [N-1:0] y;
2         assign y = out[1*N-1:0];
```

Codeausschnitt 4.2: out_ex19.v

Der benutzte Parameter N wird festgelegt durch die Transforamtionsregel $t1$, die zu Beginn der Transformation immer aufgerufen wird. N entspricht dem Parameter, der angibt, wie viele Bits pro Integer-Wert genutzt werden. Demzufolge kann y genau ein Integer-Wert zugeordnet werden.
Mittels der *assign* Anweisung wird y der Wert zugewiesen, der im Bereich von 0 bis $N-1$ auf dem Eingangvektor *out* liegt.
Der Befehl *declare-fun* wurde dementsprechend erwartungsgemäß verarbeitet.

In Zeile 2 des Ausschnitts des Bedingungsdokumentes wird der Befehl *define-fun* verwendet. In diesem Fall wird erwartet, dass das Tool eine Verilog-Variable mit dem Namen $.def\_135$ erstellt; da dies jedoch kein gültiger Bezeichner innerhalb von Verilog ist, wird der Name leicht abgeändert: $.def\_135 \leftrightarrow Sdef\_135\_\_0$ (siehe s2, s3 im Kapitel 3.3.1 Sonderregeln). Zugewiesen wird dieser Verilog-Variablen

[2] Bedingungsdokument1: „Anforderung1_ex19.smt“, „Anforderung2_ex19.smt“; Bedingungsdokument2: „Anforderung1_inf6.smt“, „Anforderung2_inf6.smt“, „Anforderung3_inf6.smt“, „Anforderung4_inf6.smt“, „Anforderung5_inf6.smt“

ein Term. Terme werden laut Transformationsregel t3 durch eine Verilog-Variable mit dem Namen *term__x* mit $x \in \mathbb{N}$ dargestellt:

```
1        wire [N-1:0] Sdef_135__0;
2        assign Sdef_135__0 = term__7;
```

Codeausschnitt 4.3: Verilog Gegenstück

Wie erwartet enthält das Verilog-Modul eine Verilog-Variable mit leicht angepasstem Namen, die Platz für einen Integer-Wert bietet. Zugewiesen wird dieser Variablen ein Term mit dem Namen *term__7*.

Des Weiteren wird erwartet, dass der durch *term__7* verkörperte Term dem folgenden Ausdruck aus dem Bedingungsdokument entspricht:

$$(* \ (to\_int \ (- \ 1)) \ |main :: x@1|)$$

Laut Transformationsregel t3 wird für jeden geklammerten Term eine Verilog-Variable definiert. Für den obigen Ausdruck werden somit 3 Verilog-Variablen inklusive *term__7* erwartet:

```
1        wire [N-1:0] term__5;
2        wire [N-1:0] term__6;
3        wire [N-1:0] term__7;
4
5        assign term__5 = -1;
6        assign term__6 = term__5;
7        assign term__7 = (term__6 * y);
```

Codeausschnitt 4.4: Verilog Gegenstück

Den Erwartungen entsprechend repräsentiert *term__5* den geklammerten Ausdruck zur Darstellung einer negativen Zahl. *term__6* entspricht der *to_int* Anweisung und *term__7* der Multiplikation.
Auch der Befehl *define-fun* wurde demzufolge erwartungskonform transformiert.

Für alle nicht abgebildeten Codezeilen der Bedingungsdokumente konnten ebenfalls Verilog-Gegenstücke gefunden werden. Daher bleibt abschließend festzuhalten, dass die exemplarisch betrachteten Bedingungsdokumente wie erwartet entsprechend der Transformationsregeln in Verilog-Code übersetzt wurden.

5 Evaluierung

In diesem Kapitel wird das vorgestellte Tool genutzt, um sicherzustellen, dass die dadurch generierten Eigenschaftsprüfer für den Einsatz in einem Hardware/Software-Co-Verifikationsverfahren geeignet sind.
Dazu werden Eigenschaftsprüfer mit Hilfe des Tools generiert und anschließend in Bezug auf unterschiedliche Eigenschaften untersucht.

Zuerst wird getestet, ob die Syntax des generierten Verilog-Codes gültig ist bzw. alle innerhalb der Eigenschaftsprüfer genutzten Worte, gültige Worte innerhalb der Sprache Verilog sind. Dies ist unbedingt erforderlich, damit die Eigenschaftsprüfer in einem Hardware/Software-Co-Verifikationsverfahren eingesetzt werden können; andernfalls könnte der Verilog-Code nicht korrekt kompiliert werden.

Als zweites wird überprüft, dass die Semantik des Codes keine Fehler aufweist. Die Semantik einer Sprache beschreibt die Bedeutung der Sprachkonstrukte bzw. des Codes. Syntaktisch ungültige Wörter haben beispielsweise keine Semantik. Damit die Eigenschaftsprüfer im Einsatz funktionieren, muss demzufolge sichergestellt sein, dass der Einsatz aller Worte im Code semantisch korrekt erfolgt.

Zuletzt wird getestet, ob das Verhalten der Eigenschaftsprüfer dem erwarteten Verhalten entspricht. Das Fehlerbit soll nur dann 0 bzw. $false$ sein, wenn aus jeder Vorbedingung die Nachbedingung des entsprechenden Paares folgt. Ist dies nicht gewährleistet, kann das Tool zur Generierung von Eigenschaftsprüfern nicht innerhalb eines Verifikationsverfahrens eingesetzt werden.

5.1 Testmethoden

5.1.1 Syntax- und Semantiktest

Die syntaktische und semantische Überprüfung der generierten Eigenschaftsprüfer wird automatisch mit dem Tool Icarus Verilog[1] durchgeführt; dabei handelt es sich um ein Synthese- und Simulationstool für Verilog-Module. Mit Hilfe dieses Tools ist es möglich Verilog-Code zu kompilieren, sodass er auf Desktop-Computern simuliert werden kann. Sollten während der Kompilierung des Verilog-Codes syntaktische oder semantische Fehler/Warnungen auftreten wird der Anwender darüber informiert.

[1] Icarus Verilog - http://iverilog.icarus.com/

Alle Eigenschaftsprüfer, die in diesem Kapitel getestet werden, werden daher mittels Icarus Verilog kompiliert. Sollten dabei keine Fehler auftauchen, ist sichergestellt, dass die Eigenschaftsprüfer syntaktisch und semantisch fehlerfrei und diesbezüglich bereit für den Einsatz in einem Hardware/Software-Co-Verifikationsverfahren sind.

5.1.2 Verhaltenstest (Testsuites)

Um zu zeigen, dass das Verhalten der Eigenschaftsprüfer dem erwarteten Verhalten entspricht, wird passend zu allen Tests eine Testsuite (von Hand) implementiert bzw. angepasst. Jede Testsuite ist ein vom Eigenschaftsprüfer abgekoppeltes Verilog-Modul, das das erwartete Ergebnis des Fehlerbits mit dem tatsächlichen Ergebnis vergleicht.
Abbildung 5.1 zeigt schematisch den Funktionsumfang der Testsuites:

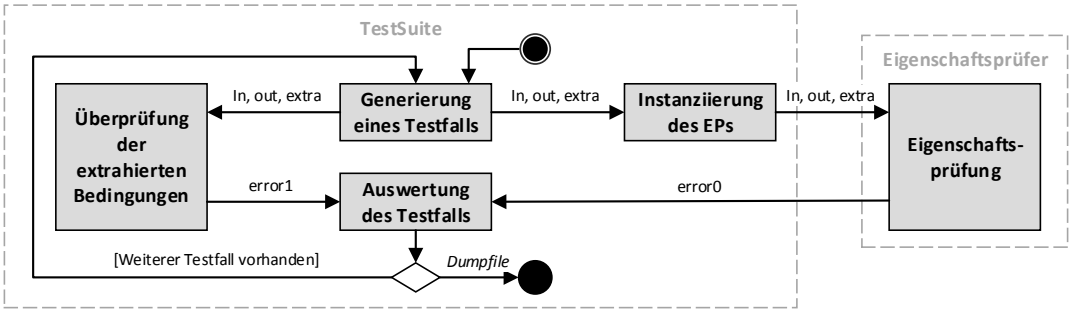

Abbildung 5.1: Testsuite

Zuerst werden Testfälle durch die Testsuite generiert. In diesem Sinne wird jeder Variablen des Eigenschaftsprüfers ein Wert im Bereich von 0 bis $2^y - 1$ zugewiesen, wobei $y \in \mathbb{N}$ der maximalen Anzahl Bits, die pro Wert genutzt werden können, entspricht. Getestet werden alle Kombinationen der Werte, wodurch passend zu einem Eigenschaftsprüfer mit x Variablen 2^{x*y} Testfälle entstehen. Die Laufzeit eines Testdurchlaufs entspricht daher auch 2^{x*y} simulierten Taktzyklen.

Die generierten Testfälle werden genutzt, indem die generierten Werte zu den zwei Eingangsvektoren (*in*, *out*) zusammengesetzt werden. Für den Fall, dass Variablen in den Bedingungen enthalten sind, die nicht in der Custom Instruction vorkommen, wird auch der Eingangvektor *extra* aus den generierten Werten zusammengesetzt. Anschließend werden die erstellten Eingangsvektoren mit dem generierten Eigenschaftsprüfer verschaltet. Diese Verschaltung wird wie im realen Einsatzszenario druchgeführt: Der Eigenschaftsprüfer erhält als Input die zwei bis drei zusammengesetzten Eingangsvektoren und gibt das Ergebnis in Form eines Fehlerbits zurück.

Des Weiteren werden die generierten Testfälle direkt innerhalb der Testsuite genutzt. Dabei wird ein weiteres Fehlerbit anhand von manuell extrahierten Bedingungen aus dem Bedingungsdokument ermittelt.

Letztendlich werden somit zwei Fehlerbits ermittelt: Einerseits das Fehlerbit $error0$, das dem Ergebnis des Eigenschaftsprüfers entspricht und andererseits das Fehlerbit $error1$, das das erwartete Ergebnis repräsentiert und durch die Testsuite berechnet wird. Diese zwei Ergebnisse werden miteinander verglichen und außerdem zur Veranschaulichung in einem Dumpfile gespeichert. Der Test ist dann als erfolgreich anzusehen, wenn sich die beiden Fehlerbits in keinem Testfall unterscheiden.

Im Folgenden wird das Tool GTKWave[2] zur Darstellung des Dumpfiles genutzt. GTKWave stellt die Ergebnisse in Form von Wellenzügen passend zu den Belegungen der Variablen bzw. der Ein- und Ausgänge in einem Graphen dar. Explizit können dadurch die Werte von den beiden generierten Fehlerbits $error0$ und $error1$ miteinander verglichen werden.

5.2 Tests

5.2.1 Test 1: Durch CPAChecker generiertes Testbeispiel 1

Dieser Test beruht auf einem *echten* Beispiel; das heißt, dass anhand des Programmcodes und der vorausgewählten Custom Instruction das Bedingungsdokument durch CPAchecker erstellt wird. Betrachtet werden zwei Bedingungspaare passend zur Custom Instruction mit der Spezifikation:

$$\begin{array}{c}(x)\ ->\ (y)\\ (=\ y\ (-\ x\ 1));\end{array}$$

Diese Custom Instruction wurde zweimal innerhalb des analysierten Programms verwendet. Daher hat CPAchecker passend zu jedem Auftreten der Custom Instruction eine Datei erzeugt, die zusammen das Bedingungsdokument ergeben. Die im Bedingungsdokument enthaltenen Bedingungen lauten (die Verwendeten Variablennamen können anhand der Tabelle A.2 im Anhang zugeordnet werden):

- Datei 1: Anforderung1_ex19.smt
  pre: $in = (extra\_3 + extra\_2 - extra\_1)$
  post: $out = (extra\_3 + extra\_2 - extra\_1 - 1)$

- Datei 2: Anforderung2_ex19.smt
  pre: $in = (1 + extra\_4 + extra\_5 - extra\_6)$
  post: $out = (extra\_4 + extra\_5 - extra\_6)$

Bei der Generierung des Eigenschaftsprüfers anhand des Bedingungsdokumentes und der zuvor genannten Spezifikation treten keine Fehler auf.

Während des Syntax- und Semantiktests bzw. während der Kompilierung des Eigenschaftsprüfers mittels Icarus Verilog treten ebenfalls keine Fehler auf. Aber eine Warnung wird wiederholt ausgegeben:

[2]GTKWave - http://gtkwave.sourceforge.net/

Assignment bit width mismatch: from 64 to 32

Diese Warnung tritt insgesamt 8 mal auf (siehe „out_ex19.v“ Zeile 122, 128, 144, 148, 168, 174, 190, 196). Zurückzuführen ist dies auf die Multiplikation von 2 Integer-Werten. Beide Werte werden durch eine Verilog-Variable repräsentiert, die einen Wert aus bis zu N Bits trägt. Das Produkt dieser Werte soll nun mit einer Verilog-Variablen verschaltet werden, die ebenfalls einen Wert mit maximal N Bits tragen kann. Abhängig von den Faktoren der Multiplikation besteht der Wert des Produkts jedoch aus bis zu $2 * N$ Bits, wodurch die Verilog-Variable das Ergebnis nicht korrekt abbilden kann und die Gefahr eines Überlaufs besteht. Bei der Verwendung von Multiplikationen muss daher beachtet werden, dass die Variabeln, die als Faktoren eingesetzt werden, nicht gleichzeitig mehr als die Hälfte der zulässigen Bits zur Integer-Darstellung nutzen.
Abgesehen von diesen Warnungen ist der generierte Verilog-Code syntaktisch und semantisch fehlerfrei.

Während des Verhaltenstests werden $2^{8*2} = 65536$ Testfälle betrachtet. Das heißt, dass für alle 8 unterschiedlichen, innerhalb des Eigenschaftsprüfers auftretenden Variablen Werte im Bereich von 0 bis 3 getestet werden. Dieser Wertebereich ist klein aber ausreichend, da die in den Bedingungen enthaltenen Konstanten nicht größer als 1 sind und damit das Ergebnis abhängig von den Variablen ist. Angenommen es existiere eine Konstante, die größer als 3 wäre, könnte das Ergebnis in diesem Fall unabhängig von den Variablen sein.

Der Verhaltenstest zeigt, dass in jedem Testfall das Ergebnis mit dem erwarteten Ergebnis übereinstimmt. Das durch den Eigenschaftsprüfer berechnete Fehlerbit $error0$ weicht also in keinem Fall von dem erwarteten Ergebnis ($error1$) ab.
Die Abbildung 5.2 zeigt einen Ausschnitt aus dem Ergebnisgraphen, der den Startbereich darstellt. Zeile 1 zeigt den Wert des Fehlerbits des Eigenschaftsprüfers

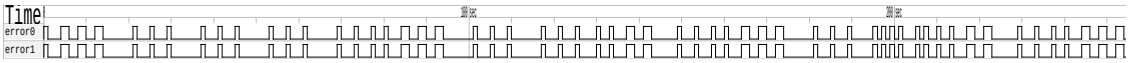

Abbildung 5.2: Ausschnitt aus Ergebnisgraph (Test 1)

($error0$) und Zeile 2 den Wert von $error1$. Ganz links am Rand sind alle Variablen mit dem Wert 0 belegt; im Verlauf des Graphen werden dann die Werte der einzelnen Variablen erhöht. Die Zeitangabe ist durch GTKWave vorgegeben und eine Zeiteinheit (eine Sekunde) entspricht in diesem Fall einem simulierten Taktzyklus.
An dem Graphen ist deutlich zu erkennen, dass wie auch hier im Ausschnitt sichtbar die beiden Fehlerbits immer übereinstimmen, der Eigenschaftsprüfer sich also in allen Fällen erwartungsgemäß verhält.

Die innerhalb dieses Tests verwendeten Dateien sind in der folgenden Tabelle aufgelistet und befinden sich im digitalen Anhang dieser Arbeit:

Spezifikationsdatei:	spec_ex19.txt
Bedingungsdokument:	Anforderung1_ex19.smt, Anforderung2_ex19.smt
Eigenschaftsprüfer:	out_ex19.v
Testsuite:	test_ex19.v
Dumpfile:	test_ex19.vcd

Da dieser Test auf einem *echten* Beispiel beruht, wird noch ein weiterer Test mit einer leicht veränderten Testsuite durchgeführt. Und zwar wird die Generation der Testfälle so angepasst, dass die Variable *out* bzw. der Output der Konfiguration entsprechend der Custom Instruction berechnet wird:

$$out = in \ - \ 1$$

Das erwartete Ergebnis dieses Tests ist, dass keines der Fehlerbits jemals den Wert 1 annimmt. Auch dieser Test kann erfolgreich durchgeführt werden. Anhand einer SAT-Kodierung der Testsuite und des Eigenschaftsprüfers wäre es möglich, einen Beweis zu führen, der zeigt, dass diese Formel unerfüllbar ist. Die veränderte und für diesen Test verwendete Testsuite befindet sich im digitalen Anhang in der Datei: *test_ex19_impCI.v*

5.2.2 Test 2: Durch CPAChecker generiertes Testbeispiel 2

Auch dieser Test beruht auf einem *echten* Beispiel, das durch CPAchecker erstellt wurde und basiert auf einer Custom Instruction und fünf Bedingungspaaren.
Die Spezifikation der Custom Instruction lautet:

```
(a) -> (status)
(and
        (or
                (not (> a 0))
                (= status 0)
        )
        (or
                (> a 0)
                (= status 1)
        )
);
```

Das Bedingungsdokument besteht aus 5 Dateien. Die enthaltenen Bedingungen lauten (zur Zuordnung der Variablennamen siehe Tabelle A.2 im Anhang):

- Datei 1: Anforderung1_inf6.smt
  pre: $flag = 0$
  post: $(flag = 0) \wedge [\ (\neg(out = 0) \wedge (in \leq 0)) \ \vee \ ((out = 0) \wedge \neg(in \leq 0))\ ]$

- Datei 2: Anforderung2_inf6.smt
  pre: $(flag = 0) \wedge (in \leq 0) \wedge (1 \leq extra\_1)$
  post: $(flag = 0) \wedge (1 \leq extra\_1) \wedge (out == 1)$

- Datei 3: Anforderung3_inf6.smt
  pre: $(flag = 0) \wedge (1 \leq in) \wedge (extra\_2 = 1)$
  post: $(flag = 0) \wedge \neg(extra\_2 = out)$

- Datei 4: Anforderung4_inf6.smt
  pre: $(flag = 0) \wedge (in > 0) \wedge (extra\_3 < 0)$
  post: $(flag = 0) \wedge \neg(out = 1) \wedge (extra\_3 < 0)$

- Datei 5: Anforderung5_inf6.smt
  pre: $(flag = 0) \wedge \neg(extra\_4 = 1) \wedge (in < 0)$
  post: $(flag = 0) \wedge \neg(extra\_4 = out)$

Die Generierung des Eigenschaftsprüfers verläuft fehlerfrei. Auch beim Syntax- und Semantiktest treten weder Fehler noch Warnungen auf.

Beim Verhaltenstest werden $2^{6*2+2*1} = 16384$ Testfälle überprüft. 6 Variablen werden mit Werten von 0 bis 3 belegt; der Variablen $flag$ werden die Werte 0 und 1 zugewiesen, da nur relevant ist, ob sie 0 oder nicht 0 ist (siehe Teilbedingung aller Bedingungen: flag = 0).

Abbildung 5.3 zeigt einen Ausschnitt des Ergebnisgraphen. Dargestellt ist der

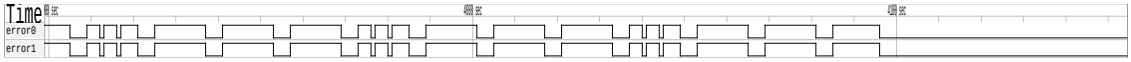

Abbildung 5.3: Ausschnitt aus Ergebnisgraph (Test 2)

Bereich, in dem der Wert für $flag$ von 0 auf 1 erhöht wird. Deutlich sichtbar ist, dass danach weder $error0$ noch $error1$ wahr wird.

Der Verhaltenstest gilt als bestanden, da der Wert anliegend an $error0$ nie vom Wert anliegend an $error1$ abweicht.

Spezifikationsdatei:	spec_inf6.txt
Bedingungsdokument:	Anforderung1_inf6.smt, Anforderung2_inf6.smt, Anforderung3_inf6.smt, Anforderung4_inf6.smt, Anforderung5_inf6.smt
Eigenschaftsprüfer:	out_inf6.v
Testsuite:	test_inf6.v
Dumpfile:	test_inf6.vcd

Da auch dieser Test auf einem *echten* Beispiel beruht, wird ebenfalls eine zweite Testsuite (*test_inf6_impCI.v*) im Zusammenhang mit diesem Beispiel zur Testdurchführung verwendet. Die Variable *out* wird dabei anhand der Custom Instruction berechnet:

$$\begin{aligned} in > 0 &\rightarrow out = 0 \\ in \leq 0 &\rightarrow out = 1 \end{aligned}$$

Auch die Durchführung dieses Test ist erfolgreich. In keinem Fall wird eines der Fehlerbits gesetzt.

5.2.3 Test 3: Kettenrechnungen

Innerhalb der ersten beiden Tests wurden ausschließlich Operatoren im Zusammenhang mit einem oder zwei Operanden benutzt. Entsprechend der Bedeutung der Operatoren und der Definition der Teilsprache können einige Operatoren jedoch beliebig viele Operanden verarbeiten. Folgender Test soll zeigen, dass auch bei mehr als 2 Operanden keine Fehler auftreten. Dazu wird eine einfache Custom Instruction gewählt:

$$\begin{array}{c} (a) \ -> \ (b) \\ (= \ a \ b); \end{array}$$

Ein Bedingungsdokument, bestehend aus einer Datei, enthält dazu passend ein Paar von Bedingungen (die Variable *in* bzw. *out* entspricht der Variablen a bzw. b der Spezifikation):
pre: *true*

post:
$$\begin{array}{lll} (6 = & (in \ + \ in \ + \ in)) & \wedge \\ (8 = & (in \ * \ in \ * \ in)) & \wedge \\ ((/ \ 1 \ 2) = & (out \ / \ out \ / \ out)) & \wedge \\ (-2 = & (out \ - \ out \ - \ out)) & \end{array}$$

Die Nachbedingung enthält den logischen Operator *and* mit 4 Operanden, die wiederum aus mathematischen Termen bestehen, die mehr als 2 Operanden enthalten.

Die Generierung des Eigenschaftsprüfer, sowie der Syntax- und Semantiktest läuft einwandfrei.

Vom Verhaltenstest wird erwartet, dass sowohl $error0$ als auch $error1$ nur falsch (0) wird, wenn beide Variablen den Wert 2 tragen. Der unter Abbildung 5.4 dargestellte Ausschnitt aus dem Ergebnisgraph zeigt die entsprechende Stelle.

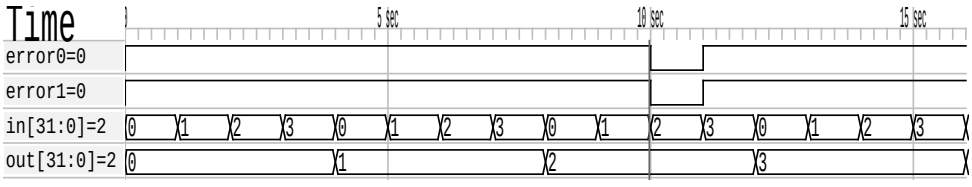

Abbildung 5.4: Ausschnitt aus Ergebnisgraph (Test 3)

Zeile 1 zeigt das vom Eigenschaftsprüfer berechnete Fehlerbit. Zeile 2 das in der

Testsuite ermittelte Fehlerbit. Die letzten zwei Zeilen zeigen die Variablen *in* und *out*. Deutlich zu erkennen ist das beide Fehlerbits auf falsch wechseln sobald beide Variablen den Wert 2 tragen.

Der Test konnte erfolgreich durchgeführt werden, dass heißt auch Operatoren mit mehr als zwei Operanden werden bei der Generierung eines Eigenschaftsprüfers korrekt verarbeitet.

Spezifikationsdatei:	spec_ab.txt
Bedingungsdokument:	kettenrechnung.smt2
Eigenschaftsprüfer:	out_kettenrechnung.v
Testsuite:	test_kettenrechnung.v
Dumpfile:	test_kettenrechnung.vcd

5.2.4 Test 4: Gleiche Variablennamen

Gemäß der Sonderregeln s2 und s3 werden die, innerhalb des Eigenschaftsprüfers verwendeten, Variablennamen abgeändert. Im Vergleich zu den Variablennamen im Bedingungsdokument, enthalten die abgeänderten Variablennamen keine Sonderzeichen mehr, die in Verilog nicht erlaubt sind. Außerdem sind alle Variablen eindeutig benannt. Das heißt, wenn innerhalb zweier Dateien desselben Bedingungsdokumentes ein Variablenname mehrfach benutzt wird und dieser nicht anhand der Custom Instruction zugeordnet werden kann, bleiben diese Variablen trotz des gleichen Namens unterscheidbar.

Dieser Test soll zeigen, dass die Verwendung von gleichen Variablennamen innerhalb von zwei Dateien eines Bedingungsdokumentes keine Probleme hervorruft. Dazu wird die selbe Custom Instruction wie im vorherigen Test verwendet. Anders als bei den vorherigen Tests werden anhand von vier Dateien, wovon jeweils zwei zu einem Bedingungsdokument gehören, zwei Eigenschaftsprüfer generiert. Beide Bedingungsdokumente sind genau gleich abgesehen von einem Unterschied.
Innerhalb des ersten Bedingungsdokumentes wird in beiden Dateien eine zusätzliche Variable mit dem Namen x benutzt. Im zweiten Bedingungsdokument heißt diese Variable in einer Datei x und in der anderen Datei y. Das heißt im ersten Bedingungsdokument ist ein Konflikt möglich, wo hingegen im zweiten Bedingungsdokument keiner möglich ist.
Die Bedingungen innerhalb der Bedingungsdokumente lauten (*in* entspricht a und *out* entspricht b. $extra\_1$ entspricht x und $extra\_2$ entspricht x bzw. y):

- Datei 1 des jeweiligen Bedingungsdokumentes.
  pre: $in > extra\_1$
  post: $out > extra\_1$

- Datei 2 des jeweiligen Bedingungsdokumentes.
  pre: $in > extra\_2$
  post: $out > extra\_2$

Innerhalb der Testsuite werden beide Eigenschaftsprüfer miteinander verglichen bzw. das Fehlerbit, das durch den ersten Eigenschaftsprüfer ($error0$) ermittelt wird, wird mit dem Fehlerbit des zweiten Eigenschaftsprüfers ($error1$) verglichen. Die Testsuite selbst berechnet in diesem Fall kein Fehlerbit. Stimmen beide Fehlerbits in allen Testfällen überein, gilt der Test als bestanden.

Die Generierung der beiden Eigenschaftsprüfer verläuft fehlerfrei. Auch die Überprüfung der Syntax und Semantik zeigt keine Fehler auf. Und der Verhaltenstest zeigt, dass sich beide Eigenschaftsprüfer genau gleich Verhalten:

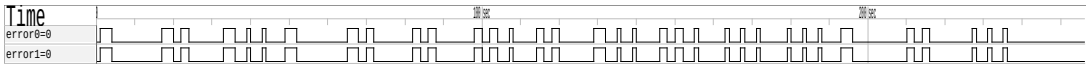

Abbildung 5.5: Ausschnitt aus Ergebnisgraph (Test 4)

Spezifikationsdatei:	spec_ab.txt
Bedingungsdokument:	variablennamen1_1.smt2, variablennamen1_2.smt2, variablennamen2_1.smt2, variablennamen2_2.smt2
Eigenschaftsprüfer:	out_variablennamen_1.v, out_variablennamen_2.v
Testsuite:	test_variablennamen.v
Dumpfile:	test_variablennamen.vcd

5.2.5 Test 5: Lokale Variablen

In keinem der bisherigen Tests wurden lokale Variablen verwendet. Dieser Test soll zeigen das auch diese Form der Variablendefinition korrekt verarbeitet wird. Der Aufbau dieses Test ähnelt dem Aufbau des letzten Tests. Auch bei diesem Test werden zwei Eigenschaftsprüfer generiert. Die beiden Fehlerbits dieser Eigenschaftsprüfer werden miteinander verglichen und die Testsuite selbst berechnet kein Fehlerbit. Verwendet wird die selbe Custom Instruction Spezifikation wie bei den letzten zwei Tests. Die Bedingungsdokumente aus denen die zwei Eigenschaftsprüfer generiert werden enthalten jeweils zwei zusätzliche Variablen. Die Bedingungsdokumente unterscheiden sich in nur einem Punkt: Innerhalb des ersten Bedingungsdokuments wird eine der zwei zusätzlichen Variablen als lokale Variable definiert.
Beide Bedingungsdokumente enthalten ein Bedingungspaar:
pre: $extra\_1 = 0$
post: $extra\_2 > 0$

Es treten keine Fehler während der Generierung der Eigenschaftsprüfer auf. Des Weiteren wird der Syntax- und der Sematiktest erfolgreich abgeschlossen. Auch der Verhaltenstest ist erfolgreich. Beide Fehlerbits tragen zu jedem Zeitpunkt den gleichen Wert:

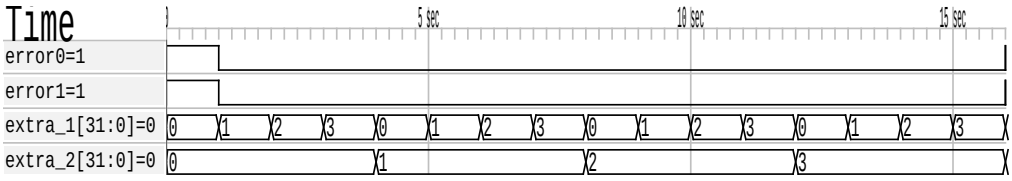

Abbildung 5.6: Ausschnitt aus Ergebnisgraph (Test 5)

Spezifikationsdatei:	spec_ab.txt
Bedingungsdokument:	lokvars_1.smt2, lokvars_2.smt2
Eigenschaftsprüfer:	out_lokvars_1.v, out_lokvars_2.v
Testsuite:	test_lokvars.v
Dumpfile:	test_lokvars.vcd

5.2.6 Test 6: Mehrere Paare von Bedingungen

Mehrere Paare von Bedingungen wurden bereits in den ersten beiden Tests betrachtet. In diesem Fall jedoch werden mehrere Bedingungspaare in einer Datei definiert.
Es werden wiederum zwei Eigenschaftsprüfer generiert. Einer beruht auf einem Bedingungsdokument bestehend aus zwei Dateien, die jeweils ein Bedingungspaar enthalten. Der Andere beruht auf einem Bedingungsdokument, das die gleichen Bedingungspaare in nur einer Datei enthält.
Die enthaltenen Vor- und Nachbedingungen lauten:

- Datei (1/2) des ersten Bedingungsdokumentes.
  pre: $true$
  post: $in \geq 2$

- Datei (2/2) des ersten Bedingungsdokumentes.
  pre: $true$
  post: $out > in$

- Einzige Datei des zweiten Bedingungsdokumentes.
  pre1: $true$
  post1: $in \geq 2$
  pre2: $true$
  post2: $out > in$

Die Spezifikation der Custom Instruction ist die selbe wie während der letzten drei Tests.

Die Generierung der beiden Eigenschaftsprüfer läuft fehlerlos. Bei der Kompilierung der Eigenschaftsprüfer mit Icarus Verilog treten keine Fehler auf. Beide Eigenschaftsprüfer liefern immer dasselbe Ergebnis; beide Fehlerbits stimmen zu

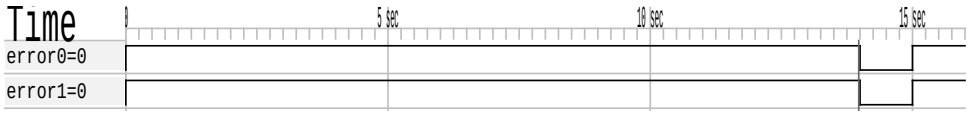

Abbildung 5.7: Ausschnitt aus Ergebnisgraph (Test 6)

jedem Zeitpunkt miteinander überein (siehe Abbildung 5.7).

Spezifikationsdatei:	spec_ab.txt
Bedingungsdokument:	bedingungen1_1.smt2, bedingungen1_2.smt2, bedingungen2.smt2
Eigenschaftsprüfer:	out_bedingungen_1.v, out_bedingungen_2.v
Testsuite:	test_bedingungen.v
Dumpfile:	test_bedingungen.vcd

5.2.7 Auflistung der getesteten Eigenschaften

Eigenschaft:	**Verwendet in den Tests:**	**Testergebnis:**
Bedingungsdokument bestehend aus..		
mehreren Dateien	1, 2, 4, 6	erfolgreich
einer Datei	3, 5, 6	erfolgreich
Negative Zahlen bzw. Konstanten	1, 3	erfolgreich
Operationen mit..		
einem Operanden	2	erfolgreich
zwei Operanden	1 - 6	erfolgreich
mehr als zwei Operanden	3	erfolgreich
Gleiche Variablennamen	4	erfolgreich
Lokale Variablen	5	erfolgreich
Mehrere Bedingungspaare	6	erfolgreich

Alle Test wurden erfolgreich bestanden, daher kann abschließend festgehalten werden, dass die generierten Eigenschaftsprüfer erwartungsgemäß funktionieren und sowohl das Tool zur Generierung als auch die generierten Eigenschaftsprüfer selbst bereit für den Einsatz in einem Hardware/Software-Co-Verifikationsverfahren sind.

6 Fazit und Ausblick

Das Hauptziel dieser Bachelorarbeit konnte erreicht werden: Es wurde ein Konzept entwickelt, dass es ermöglicht, Eigenschaftsprüfer zu generieren. Die Grundlage, auf der die Generierung aufbaut wird, wurde im Kontext des gesamten Verfahrens vorgestellt und später formal definiert. Diese formale Definition beruht auf einer Teilsprache, die aus der Sprache SMT-Lib abgeleitet wurde. Grundsätzlich kann die Generierung eines Eigenschaftsprüfer zusammengefasst werden als die Transformation eines mittels dieser Teilsprache definierten Dokuments in ein Modul, das mit Hilfe der Hardwarebeschreibungssprache Verilog dargestellt wird.

Auch die Teilziele konnten umgesetzt werden. Das im Konzept entwickelte Verfahren wurde in ein Tool umgesetzt, das die Generierung von Eigenschaftsprüfern automatisiert. Innerhalb der Evaluierung konnten anhand der Eigenschaftsprüfer, die mittels dieses Tools generiert wurden, Ergebnisse erzielt werden, die die Einsatzbereitschaft des Konzepts bzw. des Tools untermauern.

Insgesamt wurden alle angestrebten Ziele erreicht und ein weiterer Schritt in Richtung automatisierte und risikofreie Auslagerung von Programmteilen auf rekonfigurierbare Hardware zur Beschleunigung bzw. Verbesserung einer IT-Lösung wurde vollendet.

Eine Verbesserungsmöglichkeit des Konzepts läge in der Behandlung von Überläufen. Diese werden nicht gesondert behandelt und treten vor allem bei Multiplikationen von großen Zahlenwerten auf. Bisher wird auf diese möglichen Überläufe per Compiler-Warnungen hingewiesen; besser wäre jedoch eine Vermeidung durch eine alternative Darstellung der *zu großen* Zahlenwerte. Eine Lösung, die ins Konzept integriert werden könnte, bestünde darin, die möglichen Überläufe abzufangen und die Darstellung der überlauf-gefährdeten Variablen auf zwei Variablen aufzuteilen. Dies setze voraus, dass diese Variablen, wann immer sie im Verilog-Modul eingesetzt werden, bitweise bearbeitet werden. Dementsprechend müsste beispielsweise auch eine Multiplikation bitweise durchgeführt werden.

Nicht durchgeführt wurde der formale Beweis anhand einer SAT-Kodierung, die auf Grundlage einer mit einem Eigenschaftsprüfer verschalteten Konfiguration beruht. Die Erstellung eines solchen Beweises wäre der nächste Schritt im betrachteten Verifikationsverfahren. Wünschenswert wäre es, auch diesen Schritt zu automatisieren bzw. ein Tool zu erstellen oder zu benutzen, dass die SAT-Kodierung automatisch erstellt, sodass anhand dieser Kodierung automatisch mittels eines

SAT-Solvers bewiesen werden könnte, dass die Konfiguration korrekt synthetisiert wurde.

Durch diesen Schritt würde das gesamte Hardware/Software-Co-Verifikationsverfahren, begonnen bei der Erstellung des Bedingungsdokumentes durch CPAchecker über die Generierung des Eigenschaftsprüfers bis hin zur formalen Verifikation der Korrektheit, automatisiert sein. Es wäre möglich, ein Programm durch das Auslagern von Custom Instructions zu beschleunigen, ohne das Risiko einzugehen, eine nicht korrekt synthetisierte Konfiguration zu verwenden, da diese automatisch auf Korrektheit überprüft werden würde.

Abschließend ist festzuhalten, dass der Einsatzbereich des in dieser Arbeit konstruierten Konzepts nicht auf das erläuterte Proof-Carrying Hardware- und vorgestellte Hardware/Software-Co-Verifikationsverfahren limitiert ist. Überall dort, wo derartige Eigenschaftsprüfer eingesetzt werden können, kann das Konzept genutzt werden, da es ausschließlich auf standardisierten, fest-definierten Sprachkonstrukten aufbaut.

Anhang A

Tabellen & Grammatiken

A.1 Grammatik zur Darstellung der Spezifikationen

$$
\begin{array}{lcl}
G_{Spec} & = & (T, N, P, S) \\
 & & \\
T & = & \{\text{`␣`}, \text{`(`}, \text{`)`}, \text{` + `}, \text{` − `}, \text{` ∗ `}, \text{`_`}, A-Z, a-z, 0-9\} \\
N & = & \{\text{`}\langle bool\rangle\text{`}, \text{`}\langle calc\rangle\text{`}, \text{`}\langle component\rangle\text{`}, \text{`}\langle expression\rangle\text{`}, \text{`}\langle numeral\rangle\text{`}, \text{`}\langle operator\rangle\text{`}, \\
 & & \text{`}\langle specification\rangle\text{`}, \text{`}\langle symbol\rangle\text{`}, \text{`}\langle valueORvariable\rangle\text{`}, \text{`}\langle vars\rangle\text{`}\} \\
S & = & \{\text{`}\langle specification\rangle\text{`}\} \\
P & = & \{ \\
 & & p1 : \langle specification\rangle ::= \text{`(` } \langle vars\rangle \text{ `)}- > \text{(` } \langle vars\rangle \text{ `)` } \langle calc\rangle^{+}, \\
 & & p2 : \langle vars\rangle ::= \langle symbol\rangle \text{ (`, ` } \langle symbol\rangle)^{*}, \\
 & & p3 : \langle calc\rangle ::= \langle expression\rangle \text{ `;`}, \\
 & & p4 : \langle expression\rangle ::= \text{`(` } \langle operator\rangle \ \langle component\rangle^{+} \text{ `)`}, \\
 & & p5 : \langle component\rangle ::= \langle valueORvariable\rangle \mid \langle expression\rangle, \\
 & & p6 : \langle valueORvariable\rangle ::= \langle symbol\rangle \mid \langle numeral\rangle \mid \langle bool\rangle, \\
 & & p7 : \langle symbol\rangle ::= \text{a non-empty sequence of letters, digits and underscore} \\
 & & \qquad \text{not starting with a digit}, \\
 & & p8 : \langle operator\rangle ::= + \mid - \mid * \mid div \mid mod \mid and \mid or \mid not \mid > \\
 & & \qquad \mid >= \mid < \mid <= \mid = \mid distinct \ , \\
 & & p9 : \langle numeral\rangle ::= \text{a non-empty sequence of digits not starting with a zero}, \\
 & & p10 : \langle bool\rangle ::= \text{`}true\text{`} \mid \text{`}false\text{`} \\
 & & \}
\end{array}
$$

Grammatik A.1: G_{Spec}

A.2 Matching der Variablen

Bedingungsdokument	**Variable im Bedingungs-dokument**	**Variable im Eigen-schaftsprüfer**	**Variable in der Spezi-fikati-on**	**Variable in der Te-stum-gebung**
Anforderung1_ex19.smt	$\|main :: x@1\|$	y	y	out
	$\|main :: x@0\|$	x	x	in
	$\|main :: j\|$	$SmainSSjS\_\_0$	–	$extra\_1$
	$\|main :: i\|$	$SmainSSiS\_\_0$	–	$extra\_2$
	$\|main :: y\|$	$SmainSSyS\_\_0$	–	$extra\_3$
Anforderung2_ex19.smt	$\|main :: y@1\|$	y	y	out
	$\|main :: y@0\|$	x	x	in
	$\|main :: j\|$	$SmainSSjS\_\_1$	–	$extra\_4$
	$\|main :: x\|$	$SmainSSxS\_\_1$	–	$extra\_5$
	$\|main :: i\|$	$SmainSSiS\_\_1$	–	$extra\_6$
Anforderung1_inf6.smt	$\|main :: a\|$	a	a	in
	$\|main :: status\|$	$status$	$status$	out
	$\|main :: flag\|$	$SmainSSflagS\_\_0$	–	$flag$
Anforderung2_inf6.smt	$\|main :: a\|$	a	a	in
	$\|main :: as\|$	$status$	$status$	out
	$\|main :: b\|$	$SmainSSbS\_\_1$	–	$extra\_1$
	$\|main :: flag\|$	$SmainSSflagS\_\_1$	–	$flag$
Anforderung3_inf6.smt	$\|main :: b\|$	a	a	in
	$\|main :: bs\|$	$status$	$status$	out
	$\|main :: as\|$	$SmainSSasS\_\_2$	–	$extra\_2$
	$\|main :: flag\|$	$SmainSSflagS\_\_2$	–	$flag$
Anforderung4_inf6.smt	$\|main :: a\|$	a	a	in
	$\|main :: as\|$	$status$	$status$	out
	$\|main :: b\|$	$SmainSSbS\_\_3$	–	$extra\_3$
	$\|main :: flag\|$	$SmainSSflagS\_\_3$	–	$flag$
Anforderung5_inf6.smt	$\|main :: b\|$	a	a	in
	$\|main :: bs\|$	$status$	$status$	out
	$\|main :: as\|$	$SmainSSasS\_\_4$	–	$extra\_4$
	$\|main :: flag\|$	$SmainSSflagS\_\_4$	–	$flag$

Tabelle A.2: Matching der Variablen aus Test 1 & 2

Anhang B

Digitaler Anhang

Inhalt der CD:

- Die Datei *EPGenerator.jar* repräsentiert das Tool zur Generierung von Eigenschaftsprüfern in Form einer ausführbaren .jar-Datei. Sie befindet sich im Verzeichnis *EPGenerator*.
- Der gesamte Quellcode des Tools ist im Verzeichnis *Quellcode* zu finden.
- Das Tool als exportiertes Eclipse-Projekt befindet sich in der Datei *EPGenerator.zip*
- Alle Dateien, die im Rahmen der Tests genutzt wurden befinden sich im Verzeichnis *Tests*.

Literatur

[BSST08] Barrett, Clark; Sebastiani, Roberto; Seshia, Sanjit A.; Tinelli, Cesare: *Satisfiability Modulo Theories.* 2008

[BST12] Barrett, Clark; Stump, Aaron; Tinelli, Cesare: *The SMT-LIB Standard Version 2.0*, 2012

[CLRS09] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford: *Introduction to Algorithms.* 2009

[Cok13] Cok, David R.: *The SMT-LIBv2 Language and Tools: A Tutorial*, 2013

[DKP10] Drzevitzky, Stephanie; Kastens, Uwe; Platzner, Marco: Proof-Carrying Hardware: Concept and Prototype Tool Flow for Online Verification. In: *International Journal of Reconfigurable Computing* (2010)

[Hau10] Haubelt, Christian: Digitale Hardware/Software-Systeme: Spezifikation und Verifikation. (2010)

[Hop06] Hoppe, Bernhard. *Verilog : Modellbildung für Synthese und Verifikation.* 2006

[JPWW14] Jakobs, Marie-Christine; Platzner, Marco; Wehrheim, Heike; Wiersema, Tobias: Integrating Software and Hardware Verification. (2014)

[mik13] *www.mikrocontroller.net.* 2013

[Nec97] Necula, George C.: Proof-carrying Code. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* New York, NY, USA : ACM, 1997 (POPL '97). – ISBN 0–89791–853–3, S. 106–119

[Rei] Reis, Anthony J. D.: *Compiler Construction Using Java, JavaCC, and Yacc*

[SF10] Seto, Kenshu; Fujita, Masahiro: Custom Instruction Generation for Configurable Processors with Limited Numbers of Operands. In: *IPSJ Transactions on System LSI Design Methodology* (2010)