Analyse dynamischer Abhängigkeitsgraphen zum Debugging von Hardwaredesigns

Jan Malburg[1] Alexander Finder[1] Görschwin Fey[1,2]

[1]Fachbereich 3 - Mathematik und Informatik
Universität Bremen
{malburg,final,fey}@informatik.uni-bremen.de

[2]Institut für Raumfahrtsysteme
DLR Bremen
goerschwin.fey@dlr.de

Zusammenfassung

Debugging, das Finden und Korrigieren von Fehlern, ist eine zeitaufwändige Arbeit für Hardwaredesigns. Jedoch ist die effektive Bearbeitung zwingend notwendig um komplexe zuverlässige Systeme kostengünstig zu entwickeln. Dieser Artikel stellt einen neuen Debuggingansatz für Hardwaredesigns basierend auf der Analyse von dynamischen Abhängigkeitsgraphen vor. Der Ansatz verbindet Reverse Debugging, dynamisches Slicing und programmspektrumbasierte Fehlerlokalisierung und passt sie zur Verwendung auf Hardwaredesigns an. In einer Fallstudie an zwei unterschiedlichen Designs mit mehreren Fehlern ließ sich die benötigte Zeit zum Finden und Korrigieren der Fehler um 50% im Vergleich zu konventionellen Techniken reduzieren.

Abstract

Debugging is a time consuming task in hardware design. Thus effective debugging techniques are required when developing complex reliable systems. In this paper a new debugging approach based on the analysis of dynamic dependency graphs is presented. Our approach combines reverse debugging, dynamic forward/backward slicing, as well as spectrum-based fault localization and adapts them for the use on hardware designs. In a case study on two different designs with multiple faults the proposed debugging methodology reduced the debugging time to 50% in comparison to conventional techniques.

1 Einleitung

Mit einem Anteil von über 46 Prozent an den Entwicklungskosten ist die Qualitätssicherung der Hauptkostenpunkt bei der Entwicklung neuer ASICs [1]. Der mit 60 Prozent größte Bereich in der Qualitätssicherung ist das Debugging des Designs, also das Finden und Korrigieren von Entwurfsfehlern. Größtenteils handelt es sich beim Debugging immer noch um eine manuelle Tätigkeit, bei der ein Entwickler fehlerhaftes Verhalten des Designs zuerst anhand einer Eingabesequenz verstehen muss. Anschließend muss der Entwickler die Ursache finden und den Fehler korrigieren. Insbesondere bei Systemen mit hohen Zuverlässigkeitsanforderungen werden während der Verifikation auch komplizierte und damit schwer zu verstehendes fehlerhaftes Verhalten erkannt. Für das Debugging wird typischerweise ein Simulator verwendet, der dem Entwickler die Werte der einzelnen Variablen als Waveform anzeigt.

Zu den Techniken, die einem Entwickler beim Debugging helfen, zählen unter anderem: Reverse Debugging, dynamisches Forward und Backward Slicing und programmspektrumbasierte Fehlerlokalisierung. Reverse Debugging kann beim Debugging von Softwaresystemen die benötigte Zeit fürs Debugging auf ein Viertel reduzieren [2] und ist beispielsweise im bekannten Softwaredebugger GDB implementiert [3]. Dynamisches Forward und Backward Slicing kann dafür verwendet werden die Menge des Quellcodes zu reduzieren, die ein Entwickler betrachten muss, um einen Entwurfsfehler zu finden und zu korrigieren. Schließlich erlaubt programmspektrumbasierte Fehlerlokalisierung Teile des Quellcodes ausfindig zu machen, die mit hoher Wahrscheinlichkeit einen Entwurfsfehler enthalten. Dafür werden Coveragedaten von Simulationsläufen verwendet, von denen einige korrekte und andere inkorrekte Ergebnisse liefern [4]. Dabei handelt es sich um Techniken zum Debugging von Softwaresystemen.

In vorangegangenen Arbeiten haben wir uns mit Methoden beschäftigt, die einem Entwickler dabei helfen ein neues Design schneller zu verstehen. Zu diesen Techniken gehört programmspektrumbasierte Featurelokalsierung [5]. Diese Techniken wurden unter Zuhilfenahme dynamischer Datenflussanalysen verbessert [6]. In der vorliegenden Arbeit untersuchen wir nun die Verwendung dieser Techniken für das Debugging von *Hardware Description Language* (HDL)-Designs. Dabei werden zusätzliche Techniken aus dem Bereich des Softwaredebuggings genutzt. Folgende Debuggingtechniken für (HDL)-Designs werden in diesem Artikel präsentiert:

- Reverse Debugging
- Dynamisches Program Slicing unter Berücksichtigung von Kontrollabhängigkeiten
- Programmspektrumbasierte Fehlerlokalisierung basierend auf dynamischen Abhängigkeitsgraphen

Für die Implementierung unseres Ansatzes verwenden wir dynamische Abhängigkeitsgraphen [7], die wir zusätzlich um die Simulationswerte der einzelnen Variablen erweitert haben. Für unsere Implementierung haben wir ein Benutzer-Interface entwickelt, das es dem Entwickler erlaubt durch

Diese Arbeit wurde co-finanziert durch Deutsche Forschungsgemeinschaft (DFG, grant no. FE 797/6-1)

den Abhängigkeitsgraphen zu navigieren und dabei einen fehlerhaften Wert zu seinem Ursprung zu verfolgen. Dabei wird auch das zeitliche Verhalten des Simulationslaufes dargestellt und die Knoten des Graphen werden mit den entsprechenden Positionen im Quellcode verknüpft.

Wir haben unseren Ansatz anhand einer Fallstudie getestet, bei der wir unseren Ansatz dazu verwendet haben Entwurfsfehler in einem Design mit mehreren Fehlern zu finden und zu beheben. In der Fallstudie reduziert unser Ansatz die benötigte Zeit durchschnittlich um den Faktor 2 im Vergleich zu konventionellen Debuggingtechniken mit Waveform-Ansicht, Einzelschrittausführung der Simulation und Haltepunktepunkten.

Der Rest des Artikels ist wie folgt aufgebaut: Abschnitt 2 gibt einen Überblick über verwandte Arbeiten. Danach werden in Abschnitt 3 wichtige Begriffe und Definitionen für diesen Artikel vorgestellt. Abschnitt 4 beschreibt unseren Ansatz. In Abschnitt 5 wird dieser anhand einer Fallstudie evaluiert und Abschnitt 6 fasst den Artikel zusammen.

2 Verwandte Arbeiten

Debugging ist ein aktives Forschungsgebiet für Softwaresysteme [2], [8], [4], [9], [10], [11], [12] sowie für Hardwaredesigns [13], [14], [15], [16], [17], [18].

Typischerweise beginnt ein Entwickler beim Debugging damit Quellcode auszuschließen, von dem er sicher sein kann, dass dieser nichts mit dem fehlerhaften Verhalten zu tun hat. Eine Technik, die ihm dabei hilft, ist statisches Program Slicing, ursprünglich von Weiser für Softwaresysteme entwickelt [9]. Statisches Program Slicing berechnet für eine Position im Quellcode (Slicing Criterion) alle Stellen des Quellcodes, die das Slicing Criterion beeinflussen können (Backward Slicing) oder vom Slicing Criterion beeinflusst werden können (Forward Slicing). Ein Ansatz für statisches Program Slicing für Hardwaredesigns wurde von Clarke et. al. [13] vorgestellt. Dabei werden zuerst HDL-Strukturen auf Strukturen von Software-Programmiersprachen abgebildet und danach statisches Program Slicing durchgeführt.

Während des Debuggings sind einem Entwickler üblicherweise eine oder mehrere Eingaben bekannt, die fehlerhaftes Verhalten des Systems aufzeigen. Korel und Laski [8] entwickelten dynamisches Program Slicing um diese Tatsache auszunutzen. Dynamisches Program Slicing berechnet nur den Bereich des Quellcodes, der unter einer bestimmten Eingabe Einfluss auf das Slicing Criterion hat bzw. vom Slicing Criterion beeinflusst wird. Zhang et. al. [19] beschreiben mehrere Arten von dynamischem Program Slicing. In *Data Slicing* werden nur die Datenabhängigkeiten berücksichtigt. Bei *Full Slicing* werden zusätzlich die Kontrollflussabhängigkeiten mit einbezogen und bei *Relevant Slicing* werden auch die Fälle betrachtet, in denen sich Variablenwerte dadurch ändern können, dass eine Kontrollflussbedingung anders ausgewertet wird.

Ein anderer Ansatz die Menge des Quellcodes zu verringern, die ein Entwickler betrachten muss um einen Entwurfsfehler zu finden, ist Fehlerlokalisierung. Bei diesem Ansatz wird berechnet für welchen Teil des Quellcodes die Wahrscheinlichkeit am höchsten ist, dass er einen Entwurfsfehler enthält. Eine Art von Fehlerlokalisierung ist programmspektrumbasierte Fehlerlokalisierung. Bei Programmspektren handelt es sich um Metriken bezüglich eines Programmdurchlaufes. Coveragemetriken sind eine Untermenge der Programmspektren. Bei programmspektrumbasierter Fehlerlokalisierung werden Programmspektren, üblicherweise Statementcoverage, von unterschiedlichen Programmdurchläufen miteinander verglichen. Hierbei führen einige Programmdurchläufe zu einem fehlerhaften Ergebnis und die anderen Programmdurchläufe liefern das erwartete Ergebnis [4].

Eine andere Variante von Fehlerlokalisierung wurde von Renieres und Reiss vorgestellt [10]. Ihr Ansatz sucht aus einer Menge von korrekten Programmdurchläufen denjenigen Durchlauf heraus, welcher sich am wenigsten von einem fehlerhaften Programmdurchlauf unterscheidet. Als Fehlerkandidaten werden dann die Teile des Quellcodes identifiziert, welche im fehlerhaften Durchlauf verwendet werden, nicht jedoch im gefundenen korrekten Programmdurchlauf. Groce et. al. [11] erweitern diesen Ansatz, indem sie Modellprüfung verwenden um einen korrekten Programmdurchlauf zu generieren, der sich minimal vom fehlerhaften Programmdurchlauf unterscheidet, anstelle in einer vorhandenen Menge von erfolgreichen Programmdurchläufen zu suchen.

Delta Debugging [12] versucht dem Entwickler beim Debugging zu helfen, indem es die möglichen Fehlerursachen eingrenzt. Die ursprüngliche Version versucht die Menge der Änderungen an einem Softwaresystem einzuschränken, welche zu einem Regressionsfehler führt. Spätere Erweiterungen werden dazu verwendet einen minimalen Testfall zu erzeugen, der das fehlerhafte Verhalten aufzeigt, beziehungsweise den minimalen Unterschied zwischen erfolgreichen und fehlerhaften Testfällen herauszufinden [20]. Die Annahme dahinter ist, dass ein kleinerer Testfall einfacher zu verstehen ist und auch weniger Teile des Systems verwendet, wodurch der Suchbereich eingeschränkt wird.

In [2] beschreibt Lewis das Programm *Omniscient Debugger*, ein Debugging-Werkzeug für Java-Programme. Dieses Tool erlaubt es rückwärts die Programmausführung zu inspizieren und somit festzustellen, wo Variablen ihren Wert zugewiesen bekommen haben. Diese Art des Debuggings wird auch als *Reverse Debugging* bezeichnet.

Ein Tool, das programmspektrumbasierter Fehlerlokalisierung für HDL-Designs unterstützt, ist ZamiaCAD [21]. Jedoch arbeitet ZamiaCAD im Gegensatz zu dem in diesem Artikel vorgestellten Tool, nicht mit dynamischen Abhängigkeitsgraphen und Slicing. Stattdessen verwendet ZamiaCAD als Programmspektrum die Schnittmenge von Statement- bzw. Branchcoverage und dem statischen Slice. Dadurch enthalten die von ZamiaCAD verwendeten Programmspektren größere Mengen von Quellcode, wodurch die Ergebnisse ungenauer sind als bei unserem Ansatz. Weiterhin unterstützt ZamiaCAD kein Reverse Debugging.

Path Tracing [14] ist ein Ansatz für Hardwaredesigns, bei dem Gatter gesucht werden, die Einfluss auf ein bestimmtes Signal haben. Dabei werden, beginnend vom zu untersuchenden Signal, immer die Eingänge eines Gatters weiterverfolgt die Einfluss auf den Ausgang eines Gatters haben. Die Suche endet,

wenn die primären Eingänge erreicht werden. Damit ist Path Tracing vergleichbar zu dynamischem Backward Slicing. In [15] ist ein Ansatz beschrieben um Path Tracing auf HDL-Ebene zu benutzen, jedoch werden dabei nur Datenzuweisungen betrachtet und keine Kontrollabhängigkeiten.

Eine Art von Fehlerlokalisierung für Hardwaredesigns ist Debugging basierend auf Boolescher Erfüllbarkeit [16]. Dabei werden der Schaltkreis, Eingaben, die fehlerhafte Werte erzeugen, und die korrekten Ergebnisse in einer Booleschen Formel beschrieben. Danach wird ein Beweiser eingesetzt um Teile des Designs zu finden, die das fehlerhafte Verhalten erklären können. Dieser Ansatz ist aufgrund der Limitierungen der Beweiser nicht auf größere Designs anwendbar.

In [17] werden zwei Analysen basierend auf dem dynamischen Datenfluss vorgestellt. Diese Analysen sind „What if“ und „How can“. Bei der „What if“-Analyse wird berechnet wie das Ändern einer oder mehrerer Variablen eine Zielvariable beeinflusst. Diese Analyse entspricht der Funktion in normalen Debuggingwerkzeugen, die es dem Benutzer erlauben während der Programmausführung den Wert einer Variablen zu ändern. Die „How can“-Analyse berechnet für eine Menge von vorgegebenen Signalen eine mögliche Belegung, so dass ein Zielsignal einen definierten Wert erhält. Damit ähnelt die Grundidee dem Debugging basierend auf Boolescher Erfüllbarkeit. Um zu lange Laufzeiten zu verhindern ist die Menge des Suchraums jedoch auf 70 binäre Variablen begrenzt, was die Nutzbarkeit des Ansatzes stark einschränkt.

Beer et. al. [18] beschreiben einen Ansatz, welcher die Ursache für eine Verletzung der Spezifikation berechnet. Hierbei ist die Spezifikation als eine *Lineare temporale Logik* (LTL)-Formel gegeben. Zusätzlich benötigt wird ein fehlerhafter Simulationslauf. Jedoch sucht ihr Ansatz die Verletzung der Spezifikation bezüglich des Zeitpunktes, an dem die Spezifikation zuerst verletzt wurde. Zusätzlich werden Signalwerte als Ursachen für das fehlerhafte Verhalten angesehen und nicht Teile des Quellcodes. Dies führt dazu, dass die gefundene Ursache ein Signalwert sein kann, dessen Änderung nur dazu führt, dass das gleiche fehlerhafte Verhalten zu einem späteren Zeitpunkt auftritt.

3 Definitionen

Dieser Abschnitt enthält Definitionen, die wir im weiteren Verlauf des Artikels verwenden werden. Hierfür werden wir folgendes Beispiel verwenden:

```
1. input wire[0:7] in;
2. input wire[0:4] tick;
3. input wire clock;
4.
5. output reg out;
6. reg parity;
7. always @(posedge clock)
8.   if(tick==0)
9.     parity<=in[0];
10.  else
11.    parity<=
12.      in[tick]^parity;
```

Ein **Abhängigkeitsgraph** ist ein gerichteter Graph. Es wird unterschieden zwischen statischen und dynamischen Abhängigkeitsgraphen. Ein **statischer Abhängigkeitsgraph** enthält einen Knoten für jede Anweisung und jeden Ausdruck im Quellcode des Designs. Zwei Knoten sind durch eine gerichtete Kante miteinander verbunden, wenn es eine Eingabesequenz für das Design gibt, so dass das Ergebnis des Startknoten der

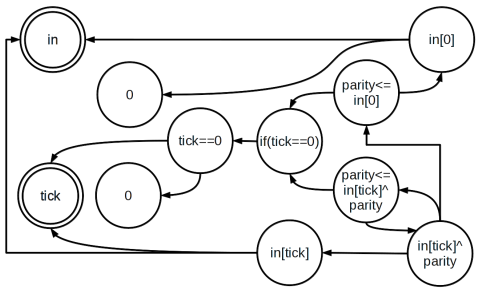

Abbildung 1. Der statische Abhängigkeitsgraph für unser Beispiel

Kante entweder von dem Endknoten der Kante gelesen wird oder wenn es sich beim Startknoten um eine Kontrollanweisung handelt, und die Ausführung des Endkonten abhängig ist von der Auswertung des Startknotens. Zum Berechnen des statischen Abhängigkeitsgraphen eines Designs ist der Quellcode des Designs ausreichend.

Abbildung 1 zeigt den statischen Abhängigkeitsgraphen für obiges Beispiel. In der Abbildung werden Ausdrücke und Anweisungen als einfach Kreise dargestellt, Eingabewerte als Doppelkreise und Abhängigkeiten als Pfeile.

Wir bezeichnen die Simulation eines Designs unter gegebenen Eingabewerten als **Simulationslauf**. Für unser Beispiel betrachten wir einen Simulationslauf, der über zwei Zeittakte läuft und bei dem alle Bits von *in* gleich 1 sind, im ersten Zeittakt *tick* gleich 0 und im zweiten Zeittakt gleich 1 ist.

Das Design ist **fehlerhaft**, wenn es mindestens eine Eingabesequenz gibt, bei der das Verhalten des Designs der Spezifikation widerspricht. Ein Grund für einen solchen Widerspruch wird als **Entwurfsfehler** bezeichnet.

Während eines Simulationslaufs l eines HDL-Designs H, können Ausdrücke und Anweisungen mehrmals ausgeführt werden. Der Grund hierfür kann sein, dass

- der Quellcode in verschiedenen Zeittakten ausgeführt wird,
- der Quellcode mehrmals instanziiert wurde,
- sich der Quellcode in einer Schleife befindet,
- die Ausführung des Quellcodes mehrmals pro Zeittick getriggert wird oder
- eine Kombination dieser Gründe auftritt

Wir bezeichnen eine einzelne Ausführung einer Anweisung oder eines Ausdruckes als **Ausführungspunkt**. Der Ausführungspunkt a ist eindeutig bestimmt durch seinen Instanziierungspfad, den Zeittakt in dem er ausgeführt wird und wie oft er für diesen Instanziierungspfad und Zeittakt bereits ausgeführt wurde. Je nachdem, ob der Ausführungspunkt zu einer Zuweisung, einer Kontrollflussanweisung oder einem Ausdruck korrespondiert, nennen wir ihn **Zuweisungs-Ausführungspunkt**, **Kontroll-Ausführungspunkt** oder **Ausdruck-Ausführungspunkt**. Die Menge aller Ausführungspunkte eines Simulationslaufs l wird mit A bezeichnet.

Ein **Datenpunkt** d ist definiert als der Wert einer Konstanten, der initiale Wert einer Variablen, der Wert einer Variablen nach einer Zuweisung, das Ergebnis eines Ausdrucks oder der Wert eines primären Eingangs, nachdem sich der Wert des Eingangs geändert hat. Die Menge aller Datenpunkte des Simulationslaufes l wird mit D bezeichnet.

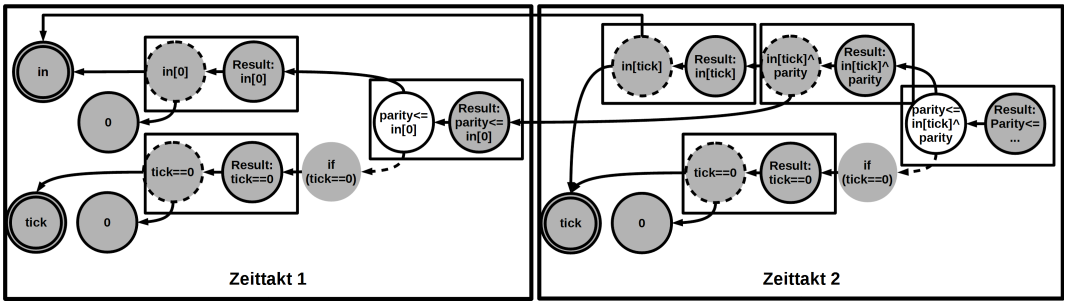

Abbildung 2. Der dynamische Abhängigkeitsgraph für unser Beispiel

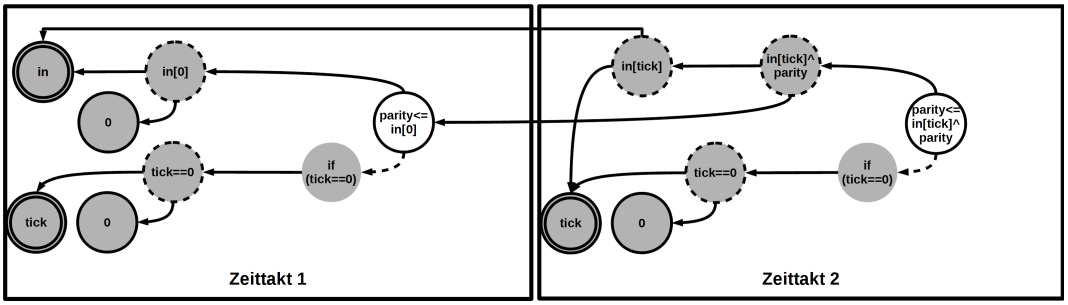

Abbildung 3. Der vereinfachte dynamische Abhängigkeitsgraph zu unserm Beispiel

Ein **dynamischer Abhängigkeitsgraph** $G = (V, E)$ ist ein gerichteter azyklischer Graph. Für seine Berechnung werden das Design H sowie ein Simulationslauf l von H benötigt. Die Knotenmenge $V = D \cup A$ von G ist die Vereinigungsmenge der Ausführungspunkte und Datenpunkte von l. Im Gegensatz zu dynamischen Abhängigkeitsgraphen, wie sie normalerweise für Softwaresysteme verwendet werden [7], enthalten die hier vorgestellten Graphen auch Datenpunkte. Dies ist der Tatsache geschuldet, dass es in HDL-Designs keine Aufrufe zum Einlesen von Eingaben gibt. Stattdessen werden die Eingabewerte dem Design durch das Ändern der primären Eingange übergeben. Daher würden die Eingabewerte ohne die Datenpunkte in unserem Graphen fehlen.

Abbildung 2 zeigt den dynamischen Abhängigkeitsgraphen für unser Beispiel. Dabei werden Knoten als Kreise repräsentiert, Zuweisungs-Ausführungspunkte werden als helle Kreise dargestellt, Ausdruck-Ausführungspunkte als Kreise mit einem gestrichelten Rand und Kontroll-Ausführungspunkte als Kreise ohne Rand. Bei den restlichen Kreisen handelt es sich um Datenpunkte. Datenpunkte, die zu primären Eingängen gehören, sind mit einem doppelten Rand gekennzeichnet.

Eine Kante $e = (v_1, v_2)$ in G ist eine gerichtete Verbindung von $v_1 \in V$ zu $v_2 \in V$. Die Menge der Kanten E ist definiert über die direkte Abhängigkeit zwischen den Knoten. Dabei gibt es zwei Arten von direkter Abhängigkeit: direkte Datenabhängigkeit und direkte Kontrollabhängigkeit.

Es besteht eine **direkte Datenabhängigkeit** von einem Datenpunkt d zu einem Ausführungspunkt a, wenn a ein Ausdrucks-Ausführungspunkt ist und d das Ergebnis von a repräsentiert oder wenn a ein Zuweisungs-Ausführungspunkt ist und d der von a zugewiesene Wert. Des Weiteren besteht eine direkte Datenabhängigkeit von a zu d, wenn d einen Operanden von a repräsentiert. Es besteht eine **direkte Kontrollabhängigkeit** von einem Ausführungspunkt a_1 zu einen Ausführungspunkt a_2, wenn a_2 der Kontroll-Ausführungspunkt ist, der die Ausführung von a_1 kontrolliert.

Es besteht eine **direkte Abhängigkeit** eines Knoten v_1 zu einem Knoten v_2, wenn entweder eine direkte Datenabhängigkeit oder eine direkte Kontrollabhängigkeit von v_1 zu v_2 besteht. In **Abbildung 2** werden direkte Datenabhängigkeiten als durchgezogene Pfeile dargestellt und direkte Kontrollabhängigkeiten als gestrichelte Pfeile.

Für einen Knoten $v \in V$ sei der ***Backwardtrace***$(v) \subseteq V$ die Menge der Knoten, die von v im Graphen unter Berücksichtigung der Kantenrichtung erreichbar sind. Entsprechend sei der ***Forwardtrace***$(v) \subseteq V$ die Menge der Knoten, welche von v aus erreichbar sind, wenn den Kanten in entgegengesetzter Richtung gefolgt wird.

Für jeden Zuweisungs-Ausführungspunkt und jeden Ausdrucks-Ausführungspunkt gibt es genau einen Knoten, der dazu eine direkte Abhängigkeit besitzt. Dieser Knoten ist immer ein Datenpunkt. Des Weiteren hat jeder Datenpunkt maximal einen Knoten, zu dem er eine direkte Abhängigkeit besitzt. Wenn ein solcher Knoten existiert, ist er immer entweder ein Zuweisungs-Ausführungspunkt oder ein Ausdrucks-Ausführungspunkt. Alle anderen Datenpunkte repräsentieren entweder Konstanten, Werte an primären Eingängen oder initiale Werte von Variablen. Dies erlaubt es die dynamischen Abhängigkeitsgraphen ohne Informationsverlust zu vereinfachen, indem Datenpunkte mit ihren zugehörigen Zuweisungs-Ausführungspunkten und Ausdrucks-Ausführungspunkten vereinigt werden. In **Abbildung 2** werden Knoten, die zusam-

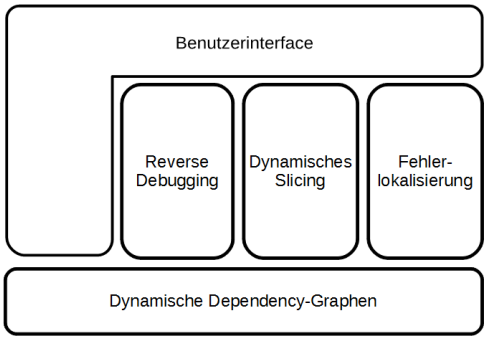

Abbildung 4. Eine schematische Übersicht der einzelnen Teile unseres Ansatzes.

mengefasst werden können, durch Rechtecke gekennzeichnet. **Abbildung 3** zeigt entsprechend den vereinfachten dynamischen Abhängigkeitsgraphen.

Die Abhängigkeitsgraphen, die wir für unseren Ansatz verwenden, werden darüber hinaus noch um Informationen bezüglich der Werte der einzelnen Variablen erweitert. Sowie, im Falle von Zuweisungs- und Kontroll-Ausführungspunkte, um die Simulationszeit, zu der sie ausgeführt werden. Für Daten- und Ausdrucks-Ausführungspunkte ist es nicht notwendig die Zeit zu speichern, da dies immer als Teil einer Anweisung geschieht und somit die Zeit mit der Zeit der entsprechenden Zuweisungs- oder Kontrollanweisung übereinstimmt. Im weiteren Verlauf dieses Artikels werden wir nur noch vereinfachte dynamische Abhängigkeitsgraphen betrachten, die um die oben genannten Informationen erweitert wurden.

4 Graphbasiertes Debugging

In diesem Abschnitt stellen wir unseren Ansatz für graphbasiertes Debugging vor. **Abbildung 4** gibt einen Überblick über unseren Ansatz. Als Grundlage für alle weiteren Funktionalitäten fungieren die dynamischen Abhängigkeitsgraphen. Berechnet werden diese mit Hilfe von instrumentalisierten Versionen des Designs. Ein grafisches Benutzerinterface erlaubt es die Graphen oder auch die Ergebnisse von weiterführenden Berechnungen auf den Graphen anzuzeigen. Reverse Debugging zeigt einem Entwickler, an welcher Stelle ein Signal oder eine Variable einen bestimmten Wert zugewiesen bekommen hat. Dabei kann der Entwickler den Graphen beliebig vorwärts und rückwärts durchlaufen. Mit Dynamic Slicing kann der Entwickler den Graph auf jene Bereiche eingrenzen, die für die Werte, die er untersucht, relevant sind und bei Fehlerlokalisierung handelt es sich um ein heuristisches Verfahren, welches dem Entwickler jene Bereiche im Graphen beziehungsweise im Quellcode aufzeigt, die mit hoher Wahrscheinlichkeit fehlerhaft sind.

4.1 Benutzerinterface

Abbildung 5 zeigt einen Screenshot von unserer GUI. Die GUI besteht aus drei Abschnitten (A-C). Der erste Abschnitt (A) ist die Graphanzeige. Hier wird der dynamische Abhängigkeitsgraph oder ein Slice dargestellt. In diesem Abschnitt sieht ein Entwickler den Daten- und Kontrollfluss innerhalb des Designs. Ebenfalls kann er erkennen, wie die einzelnen

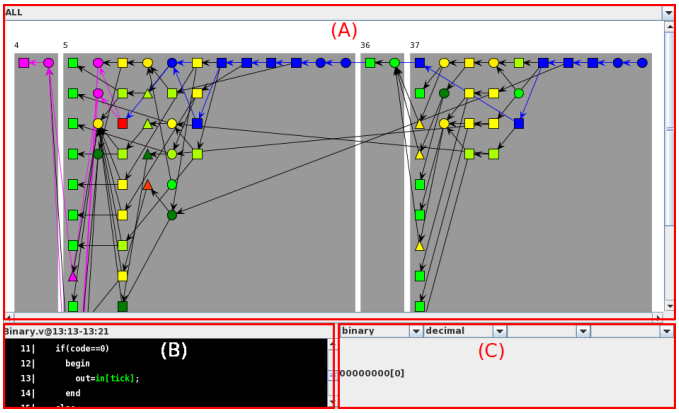

Abbildung 5. Ein Screenshot unserer GUI, bei dem die verschiedenen Teile hervorgehoben sind.

Teile des Designs voneinander abhängen. Der zeitliche Verlauf des Simulationslaufes wird am oberen Rand dargestellt. Um es dem Entwickler zu erlauben sich auch in großen Graphen zurechtzufinden, unterstützt die Graphanzeige außerdem verschiedene Detailstufen beim Darstellen der Knoten. Hierbei stehen dem Entwickler folgende Detailstufen zur Auswahl:

- Datei-Stufe: Knoten, die zu Quellcode gehören, der in der gleichen Datei ist, werden zusammengefasst.
- Modul-Stufe: Knoten, die zu Quellcode gehören, der sich im gleichen Modul befindet, werden zusammengefasst.
- Prozess-Stufe: Knoten, die zu Quellcode gehören, der sich im gleichen Prozess befindet, werden zusammengefasst.
- Basic-Block-Stufe: Knoten, die zu Quellcode gehören, der sich im gleichen Basic-Block befindet, werden zusammengefasst.
- Anweisungs-Stufe: Knoten, die zur gleichen Anweisung gehören, werden zusammengefasst.
- Ausdrucks-Stufe: Alle Knoten werden angezeigt.

Des Weiteren ist es dem Entwickler möglich sich unterschiedliche Knoten in unterschiedlichen Detailstufen anzeigen zu lassen, so dass er Teile des Graphen, die ihn besonders interessieren, genauer inspizieren kann, während er die restlichen Teile auf einer gröberen Detailstufe belassen kann. In der aktuellen Implementierung nehmen wir keine Zusammenfassung von Knoten über unterschiedliche Zeittakte hinweg vor, obwohl es technisch möglich wäre.

Im zweiten Abschnitt (B) wird jeweils der Teil des Quellcodes angezeigt, der dem momentan ausgewählten Knoten im Graphen entspricht. Und im letzten Abschnitt (C) werden die Werte der Operanden des ausgewählten Knoten angezeigt. Dabei kann der Entwickler zwischen dezimaler und binärer Darstellung der Werte wählen. Als Vereinfachung ist für jede Ausdrucksart eine Standarddarstellung definiert. Beispielsweise die dezimale Darstellung für arithmetische Operationen und die binäre Darstellung für Bit-Operationen.

4.2 Reverse Debugging

Reverse Debugging erlaubt es einem Entwickler auch eine Simulation rückwärts laufen zu lassen um so die Frage zu beantworten, warum eine Variable einen bestimmten Wert hat

und woher dieser Wert kommt. Dadurch wird es einfacher für einen Entwickler die eigentliche Ursache für einen fehlerhaften Wert zu finden. Dieser Ansatz hat mehrere Vorteile im Vergleich zu konventionellen Debuggingansätzen für HDL-Designs. Im Vergleich zu Waveform-Views sieht ein Entwickler zum Einen nicht nur, wenn sich ein Signal ändert, sondern auch aufgrund welcher Anweisung. Zum Anderen zeigt unser Ansatz Fälle bei denen eine Zuweisung zu einer Variablen mit dem gleichen Wert erfolgt. Schließlich zeigt unser Ansatz auch Zwischenergebnisse in größeren Ausdrücken. Im Vergleich zu Debugging mit Haltepunkten reduziert Reverse Debugging die Anzahl der Iterationen, da bei der Suche nach der Ursache nicht bei jedem Schritt der Simulationslauf neu ausgeführt werden muss. Darüber hinaus muss der Entwickler beim haltepunktbasiertem Debugging überlegen, an welchen Punkten die Zuweisung zu der Variablen passieren könnte. Auch dies ist beim Reverse Debugging nicht nötig, da er sich direkt zeigen lassen kann, wo ein Wert einer Variablen zugewiesen wurde.

4.3 Dynamisches Slicing

Mit dem dynamischen Abhängigkeitsgraphen sind bereits alle Informationen vorhanden um dynamische Slices zu berechnen. Dynamisches Backward Slicing zeigt einem Entwickler jene Teile des Quellcodes, die Einfluss auf ein bestimmtes Slicing Criterion haben. Verwendet man als Slicing Criterion den Wert, bei dem das fehlerhafte Verhalten beobachtet wurde, ist der Entwurfsfehler Teil des Slices mit Ausnahme einiger Sonderfälle, wie beispielsweise fehlendem Quellcode [19]. Entsprechend enthält der Forward Slice all jene Teile des Quellcodes, auf die ein Slicing Criterion Einfluss hat, und erlaubt somit abzuschätzen, wo Seiteneffekte im Falle einer Änderung des Quellcodes auftreten können. Für eine gegebene Menge von Daten- und Ausführungspunkten $C \in V$ als Slicing Criterion, lässt sich der entsprechende Backward Slice S_b durch die Vereinigung der entsprechenden Backtraces berechnen:

$$S_b = \bigcup_{c}^{C} \text{Backwardtrace}(c)$$

Entsprechend lässt sich der Forward Slice S_f durch die Vereinigung der entsprechenden Forwardtraces berechnen:

$$S_f = \bigcup_{c}^{C} \text{Forwardtrace}(c)$$

4.4 Programmspektrumbasierte Fehlerlokalisierung

Programmspektrumbasierte Fehlerlokalisierung berechnet für jeden Teil des Quellcodes die Wahrscheinlichkeit, dass dieser einen Entwurfsfehler enthält. Hierfür wird eine Menge von Simulationsläufen verwendet, von denen einige korrekt sind und andere fehlerhaft. Entsprechend kann Programmspektrumbasierte Fehlerlokalisierung auch nur eingesetzt werden, wenn korrekte und fehlerhafte Simulationsläufe vorhanden sind. In diesen Fällen kann es aber das Finden der Entwurfsfehler stark vereinfachen. Für die Berechnung der Wahrscheinlichkeiten wird ein Programmspektrum verwendet, zum Beispiel eine Coveragemetrik. Für Softwaresysteme wird typischerweise Statementcoverage verwendet. Beim Verwenden von Statementcoverage als Programmspektrum für HDL-Beschreibungen werden jedoch nur mäßige Ergebnisse erzielt [5]. Dies ist hauptsächlich der Tatsache geschuldet, dass große Teile des HDL-Quellcodes immer ausgeführt werden, aber keinen Einfluss auf die eigentlich durchgeführte Berechnung haben. Daher verwenden wir als Programmspektrum jene Codeteile, die zu Knoten im Abhängigkeitsgraphen oder einem Slice korrespondieren.

Tabelle I
DURCHSCHNITTLICHE ANZAHL VON KNOTEN PRO ZEITTAKT FÜR DIE UNTERSCHIEDLICHEN DETAIL-STUFEN.

Design	Ausdruck	Anweisung	Basic Block	Prozess	Modul / Datei[1]
Converter	43.7	14.9	14.4	5.6	4.3
FPU	205.9	75.2	30.6	12.3	3.1

5 Fallstudie

Mit dieser Fallstudie untersuchen wir in wieweit unser Ansatz einem Entwickler beim Debugging eines HDL-Designs hilft. Zuerst evaluieren wir, wieviel Abstraktion wir durch die unterschiedlichen Detailstufen erreichen können, danach die Zeitersparnis, die durch unseren Ansatz erreicht werden kann.

5.1 Abstraktion

Wir sind daran interessiert, wie stark die unterschiedlichen Detailstufen den Graphen übersichtlicher machen können. Dafür haben wir zwei unterschiedliche Designs und die dazugehörigen Testsuites untersucht. Das erste Design ist ein Parallel-Seriell-Konverter, den wir auch im zweiten Teil unserer Fallstudie benutzen werden. Dieses Design besteht aus 271 Codezeilen und die dazugehörige Testsuite besteht aus 3045 Codezeilen. Die Simulation der kompletten Testsuite benötigt 307 Taktzyklen. Der vollständige dynamische Abhängigkeitsgraph des Designs besteht aus 13401 Knoten, und benötigt dabei 358 KB an Speicherplatz. Unter Verwendung der Datei- oder Modul-Detailstufe[1] reduziert sich die Knotenzahl auf 1315 Knoten, was einer Reduktion um ungefähr den Faktor 10 entspricht. Das zweite Design ist eine FPU von der OpenCores.org-Website. Es besteht aus 2555 Zeilen Quellcode und die dazugehörende Testsuite aus 650 Zeilen Quellcode. Die Testsuite dieses Designs benötigt 4102 Zeittakte. Der Abhängigkeitsgraph hierfür benötigt rund 26,5 MB an Speicherplatz. Für das FPU-Design konnten wir eine Reduktion der Knoten um den Faktor 66 feststellen. **Tabelle I** gibt eine Übersicht über die Anzahl der Knoten für die unterschiedlichen Detail-Stufen für beide Designs. Aufgrund der unterschiedlichen Taktzyklen, welche die Testsuites benötigen um simuliert zu werden, haben wir die Werte als durchschnittliche Anzahl an Knoten je Zeittakt angegeben.

[1]In den betrachteten Designs ist jedes Modul in einer eigene Datei implementiert, wodurch die Anzahl der Knoten auf Modul- und Datei-Detailstufe identisch sind.

5.2 Debugging

In dieser Evaluation messen wir die Zeit, die ein Entwickler benötigt um ein Design zu debuggen. Dabei vergleichen wir unseren Ansatz mit konventionellem Debugging. Hierfür verwenden wir den Parallel-Seriell-Konverter. Der Konverter unterstützt vier unterschiedliche Betriebsmodi, bei denen jeweils immer ein Byte an Daten übertragen wird:

- Binär-Modus: Je Takt wird ein Bit übertragen.
- Paritäts-Modus: Wie Binär-Modus, jedoch wird zusätzlich am Ende des Bytes noch ein Paritätsbit gesendet.
- Return-To-Zero-Modus: Nach jedem Datenbit wird ein Bit mit dem Wert Null gesendet.
- Hamming-Modus: Das Byte wird mit Hilfe eines (7,4)-Hammingcodes versendet.

Der Benutzer für unseren Vergleich war der ursprüngliche Entwickler des Konverters und der dazu gehörigen Testsuite. Für die Evaluation haben wir fünf Sätze fehlerhafter Versionen des Designs generiert, von denen zwei zufällig für die Evaluation ausgesucht wurden. Die Sätze wurden von einem weiteren Entwickler erstellt, dem das Design unbekannt war. Jeder der Sätze enthält fünf fehlerhafte Versionen, wobei jede Version zwei bis vier Änderungen enthält. Zum Erzeugen der fehlerhaften Versionen werden folgende Mutations-Operatoren verwendet: Änderung eines Operators (opr), Änderung eines Operanden (opa), hinzugefügter Quellcode (zc), entfernter Quellcode (ec) und geänderte Modulparameter (mp). Bei der Generierung der verschiedenen Sätze wurde darauf geachtet, dass in den unterschiedlichen Sätzen vergleichbare Versionen des Designs enthalten waren. Das heißt, für jede fehlerhafte Version in einem Satz, ist im anderen Satz auch eine fehlerhafte Version vorhanden, die bezüglich Anzahl und Art der Mutationen übereinstimmt.

Unser Benutzer für diese Evaluation verwendet ModelSim [22] regelmäßig für seine normale Arbeit. Für die Evaluation verwendet er zum Finden der Entwurfsfehler die Waveform-Sicht von ModelSim sowie Haltepunkte und Einzelschrittausführung der Simulation.

Um sich mit unserem Tool vertraut zu machen erhielt er vor der eigentlichen Evaluation einen weiteren Satz fehlerhafter Versionen seines Designs, an denen er das Tool ausprobieren konnte. Der Evaluationsbenutzer hat sich für folgendes Vorgehen beim Debugging mit unserem Tool entschieden: Als Eingabe für die Fehlerlokalisierung verwendete er im Falle von erfolgreichen Tests die kompletten dynamischen Abhängigkeitsgraphen, im Falle von fehlgeschlagenen Tests jedoch den dynamischen Backward Slice, bei dem als Slicing Criterion, die Datenpunkte verwendet wurden, welche zu den Werten an den primären Ausgängen zu Zeittakten korrespondierten, an denen Assertions fehlschlagen. Entsprechend verwendete der Benutzer für die Graphansicht auch jene dynamischen Slices. Die eigentliche Suche nach den Entwurfsfehlern beginnt der Benutzer an Codeteilen, welche von der Fehlerlokalisierung als am wahrscheinlichsten markiert sind sowie deren direkten Vorgänger im Abhängigkeitsgraph. Im Falle von geänderten Operatoren und Operanden wird der Entwurfsfehler dadurch meistens schon gefunden. In dem Fall, dass diese Vorgehensweise zu keinem Ergebnis führt oder, dass die Fehlerlokalisierung keine Punkte findet, folgte der Entwickler einem fehlerhaften Ausgangswert rückwärts im Abhängigkeitsgraphen, bis er den Entwurfsfehler gefunden hatte. Fehlender Quellcode und geänderte Modulparameter werden meistens dadurch entdeckt, dass Quellcode der im Slice erwartet wird, nicht enthalten ist. In allen Fällen konnte das Design erfolgreich korrigiert werden.

Tabelle II
ZEITVERGLEICH ZWISCHEN DEBUGGING UNTER VERWENDUNG DES KONVENTIONELLEM ANSATZ UND UNSERM ANSATZ

Version	Fehlertypen	Konventionell	Unser Ansatz	Differenz	Faktor
1	opa	7min 24sec	1min 54sec	5min 30sec	3.89
2	opr,opa	18min 33sec	2min 46sec	15min 75sec	6.70
3	opr	6min 20sec	7min 52sec	-1min 32sec	0.81
4	zc,ec,mp	13min 32sec	9min 19sec	4min 13sec	1.45
5	zc,op	12min 10sec	7min 52sec	4min 18sec	1.55
$\sum$		57min 59sec	29min 45sec	28min 14sec	1.95
Ø		11min 35sec	5min 57sec	5min 38sec	1.95

Nach der Trainingsperiode, starteten wir mit den eigentlichen Messungen. Für beide Ansätze beginnt der Entwickler mit einer Auflistung, welche der Testfälle erfolgreich auf dem geänderten Design laufen und welche fehlschlagen. Im Fall von fehlgeschlagenen Testfällen erhält der Entwickler auch eine Liste welche Assertions fehlschlagen. Die Anzahl der Fehler pro Design ist dem Entwickler unbekannt und ein Design gilt als erfolgreich korrigiert, wenn alle Testfälle erfolgreich sind.

Tabelle II enthält die Ergebnisse unserer Evaluation. Um die Tabelle lesbarer zu machen, haben wir die zueinander gehörenden Versionen des Designs, das heißt, Versionen mit gleichem Typ und Anzahl der Änderungen, in dieselbe Zeile geschrieben. Der Entwickler kannte diese Zuordnung nicht. Genauso war dem Entwickler nicht bekannt, welche Arten von Änderungen am Design vorgenommen wurden. Das Ausführen der nicht instrumentalisierten Version der Testsuite dauerte 2 Minuten 48 Sekunden und der instrumentalisierten Version 2 Minuten 57 Sekunden. Eine Änderung kann eine andere Änderung maskieren. Beispielsweise hatten wir eine fehlerhafte Version, bei der eine Änderung eine konstante Ausgabe erzeugt, wodurch alle anderen Änderungen maskiert wurden. Eine zweite Änderung in dieser Version war eine fehlende Modulinstanziierung und eine dritte Änderung befand sich in genau dem Modul, dessen Instanziierung fehlte. In so einem Fall muss die Testsuite mindestens 3 mal ausgeführt werden um alle Fehler zu finden. Bei anderen fehlerhaften Versionen, waren die Änderungen derart verteilt, dass alle vier Änderungen bei nur einer Ausführung der Testsuite gefunden werden konnten. Aus diesem Grund sind die Zeitangaben in **Tabelle II** ohne die Simulationszeit angegeben, da ansonsten die Zeiten davon dominiert würden, ob sich Änderungen am Design gegenseitig maskieren oder nicht. Bei der Evaluation ließen wir den Entwickler zuerst unseren Ansatz verwenden und dann den konventionellen. Dadurch verhinderten wir, dass durch eventuelle Übungseffekte unser Ansatz bevorzugt wird. Sollten Übungseffekte aufgetreten sein, haben sie somit den konventionellen Ansatz bevorteilt.

Diese Evaluation zeigt, dass unser Ansatz die benötigte Zeit zum Debugging um circa den Faktor 2 verringert. Beim Ver-

gleich der unterschiedlichen Mutationsarten, die wir verwendet haben, zeigt sich, dass unser Ansatz besonders effizient ist beim Finden von fehlerhaften Operatoren. Bei diesen Änderungen findet die Fehlerlokalisierung den Entwurfsfehler fast immer direkt. Andererseits hilft unser Ansatz im Falle von fehlendem Quellcode nur bedingt. Dies ist auch der Grund für die große Debugdauer bei Version 3. Bei dieser Version erzeugte eine der Änderungen unerreichbaren Quellcode, unser Entwickler interpretierte diesen als zusätzlich hinzugefügten Quellcode und entfernte den entsprechenden Codebereich. Danach benötige er die meiste Zeit des Debuggings dafür den nun fehlenden Quellcode wiederherzustellen.

6 Zusammenfassung

In diesem Artikel präsentieren wir Reverse Debugging, dynamisches Forward und Backward Slicing mit Berücksichtigung von Kontrollflussabhängigkeiten und programmspektrumbasierte Fehlerlokalisierung für HDL-Designs. Als Basis für die Analyse dienen dynamische Abhängigkeitsgraphen In einer Fallstudie zeigen wir, dass unser Ansatz im Vergleich zur herkömmlichen Vorgehensweise die benötige Debuggingzeit ungefähr halbiert.

Der gezeigte Ansatz skaliert auch für große Designs, da seine Laufzeit und sein Speicherbedarf als simulationsbasierter Ansatz linear zur Designgröße und der Anzahl der Zeitschritte in der Testsuite ist. Unser Ansatz erlaubt durch mehrere Stufen von Abstraktionen die Komplexität zu reduzieren, so dass ein Entwickler auch noch in großen Designs sinnvoll navigieren kann.

Literatur

[1] H. Foster, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, 2008, vol. 5123, Seiten 5–10.

[2] B. Lewis, "Debugging backwards in time," in *Proceedings of International Workshop on Automated and Algorithmic Debugging*, 2003, Seiten 225–235.

[3] "GDB: The GNU Project Debugger." URL: http://www.gnu.org/software/gdb/

[4] J. A. Jones, M. J. Harrold, und J. T. Stasko, "Visualization for fault localization," in *Proceedings of ICSE Workshop on Software Visualization*, 2001, Seiten 71–75.

[5] J. Malburg, A. Finder, und G. Fey, "Automated feature localization for hardware designs using coverage metrics," in *Proceedings of Design Automation Conference*, 2012, Seiten 941–946.

[6] J. Malburg, A. Finder, und G. Fey, "Tuning dynamic data flow analysis to support design understanding," in *Proceedings of Design, Automation and Test in Europe*, 2013, Seiten 1179–1184.

[7] H. Agrawal und J. R. Horgan, "Dynamic program slicing," *ACM SIGPLAN Notices*, vol. 25, no. 6, Seiten 246–256, 1990.

[8] B. Korel und J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, Seiten 155–163, 1988.

[9] M. Weiser, "Program slicing," in *Proceedings of International Conference on Software Engineering*, 1981, Seiten 439–449.

[10] M. Renieres und S. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of International Conference on Automated Software Engineering*, 2003, Seiten 30–39.

[11] A. Groce, S. Chaki, D. Kroening, und O. Strichman, "Error explanation with distance metrics," *International Journal on Software Tools for Technology Transfer*, vol. 8, Seiten 229–247, 2006.

[12] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *Proceedings of the European Software Engineering Conference*, ser. Lecture Notes in Computer Science, vol. 1687, 1999, Seiten 253–267.

[13] E. Clarke, M. Fujita, S. Rajan, T. Reps, S. Shankar, und T. Teitelbaum, "Program slicing of hardware description languages," in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science, 1999, vol. 1703, Seiten 72–72.

[14] M. Abramovici, P. R. Menon, und D. T. Miller, "Critical path tracing - an alternative to fault simulation," in *Proceedings of Design Automation Conference*, 1983, Seiten 214–220.

[15] M.-C. Lai, C.-H. Lee, B.-H. Ho, und J.-S. Tsai, "Active trace debugging for hardware description languages," US Patent 6 546 526, 2003.

[16] A. Smith, A. Veneris, M. Ali, und A. Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 10, Seiten 1606–1621, 2005.

[17] Y.-C. Hsu, B. Tabbara, Y.-A. Chen, und F. Tsai, "Advanced techniques for RTL debugging," in *Proceedings of Design Automation Conference*, 2003, Seiten 362–367.

[18] I. Beer, S. Ben-David, H. Chockler, A. Orni, und R. Trefler, "Explaining counterexamples using causality," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, 2009, vol. 5643, Seiten 94–108.

[19] X. Zhang, H. He, N. Gupta, und R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *Proceedings of International Symposium on Automated and Analysis-Driven Debugging*, 2005, Seiten 33–42.

[20] A. Zeller und R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, Seiten 183–200, 2002.

[21] A. Tsepurov, J. Raik, J. H. Meza Escobar, und H.-D. Wuttke, "Localization of bugs in processor designs using zamiacad framework," in *Proceedings of International Workshop on Microprocessor Test and Verification*, 2012.

[22] "ModelSim Website," Mentor Graphics. URL: http://model.com/