Vladimir Herdt
Daniel Große
Rolf Drechsler

Verbessertes virtuelles Prototyping

Mit RISC-V-Fallstudien

Springer Vieweg

Verbessertes virtuelles Prototyping

Vladimir Herdt • Daniel Große • Rolf Drechsler

Verbessertes virtuelles Prototyping

Mit RISC-V-Fallstudien

Vladimir Herdt
Universität Bremen und DFKI GmbH
Bremen, Deutschland

Daniel Große
Johannes Kepler Universität Linz
Linz, Österreich

Rolf Drechsler
Universität Bremen und DFKI GmbH
Bremen, Deutschland

Dieses Buch ist eine Übersetzung des Originals in Englisch „Enhanced Virtual Prototyping“ von Herdt, Vladimir, publiziert durch Springer Nature Switzerland AG im Jahr 2021. Die Übersetzung erfolgte mit Hilfe von künstlicher Intelligenz (maschinelle Übersetzung durch den Dienst DeepL.com). Eine anschließende Überarbeitung im Satzbetrieb erfolgte vor allem in inhaltlicher Hinsicht, so dass sich das Buch stilistisch anders lesen wird als eine herkömmliche Übersetzung. Springer Nature arbeitet kontinuierlich an der Weiterentwicklung von Werkzeugen für die Produktion von Büchern und an den damit verbundenen Technologien zur Unterstützung der Autoren.

ISBN 978-3-031-18173-3 ISBN 978-3-031-18174-0 (eBook)
https://doi.org/10.1007/978-3-031-18174-0

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Springer Vieweg
© Der/die Herausgeber bzw. der/die Autor(en), exklusiv lizenziert an Springer Nature Switzerland AG 2022
Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.
Die Wiedergabe von allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jedermann benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des jeweiligen Zeicheninhabers sind zu beachten.
Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag, noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Lektorat/Planung: Axel Garbers
Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Nature Switzerland AG und ist ein Teil von Springer Nature.
Die Anschrift der Gesellschaft ist: Gewerbestrasse 11, 6330 Cham, Switzerland

Für Elena und Oleg,
Lukas
und
Lothar

Vorwort

Virtuelle Prototypen (VP) spielen eine sehr wichtige Rolle bei der Bewältigung der steigenden Komplexität im Entwurfsablauf von eingebetteten Geräten. Ein VP ist im Wesentlichen ein ausführbares abstraktes Modell der gesamten *Hardware* (HW) Plattform und wird überwiegend in SystemC TLM (*Transaction Level Modeling*) erstellt. Im Gegensatz zu einem traditionellen Entwurfsablauf, bei dem zuerst die HW und dann die *Software* (SW) erstellt wird, ermöglicht ein VP-basierter Entwurfsablauf die parallele Entwicklung von HW und SW, indem der VP für die frühe SW-Entwicklung und als Referenzmodell für die nachfolgenden Entwurfsablaufschritte genutzt wird. Dieser moderne VP-basierte Entwurfsablauf hat jedoch immer noch Schwächen, insbesondere aufgrund des erheblichen manuellen Aufwands für die Verifizierung und Analyse sowie für Modellierungsaufgaben, der sowohl zeitaufwändig als auch fehleranfällig ist. In diesem Buch werden mehrere neuartige Ansätze vorgestellt, die Modellierungs-, Verifikations- und Analyseaspekte abdecken, um den VP-basierten Entwurfsablauf deutlich zu verbessern. Darüber hinaus enthält das Buch mehrere RISC-V-Fallstudien zur Demonstration der entwickelten Ansätze. Die Beiträge des Buches sind im Wesentlichen in vier Bereiche unterteilt: Der erste Beitrag ist ein Open-Source RISC-V VP, der in SystemC TLM implementiert ist und sowohl funktionale als auch nicht-funktionale Aspekte abdeckt. Der zweite Beitrag verbessert den Verifikationsablauf für VPs durch die Berücksichtigung neuartiger formaler Verifikationsmethoden und fortgeschrittener automatisierter, abdeckungsgeführter Testverfahren, die auf SystemC-basierte VPs zugeschnitten sind. Der dritte Beitrag sind effiziente abdeckungsgeleitete Ansätze, die die VP-basierte SW-Verifikation und -Analyse verbessern. Der vierte und letzte Beitrag dieses Buches sind Ansätze, die eine Korrespondenzanalyse zwischen

Register-Transfer-Level (RTL) und TLM durchführen, um die auf verschiedenen Abstraktionsebenen verfügbaren Informationen zu nutzen. Alle Ansätze werden detailliert vorgestellt und mit mehreren Experimenten ausführlich evaluiert, die ihre Effektivität bei der Verbesserung des VP-basierten Entwurfsablaufs eindeutig belegen.

Bremen, Deutschland

Vladimir Herdt
Daniel Große
Rolf Drechsler

Danksagung

Zunächst möchten wir uns bei den Mitgliedern der Arbeitsgruppe *Rechnerarchitektur* (AGRA) an der Universität Bremen sowie bei den Mitgliedern des Forschungsbereichs *Cyber-Physical Systems* (CPS) des *Deutschen Forschungszentrums für Künstliche Intelligenz* (DFKI) in Bremen bedanken. Wir bedanken uns für die gute Atmosphäre und das anregende Umfeld. Außerdem möchten wir uns bei allen Co-Autoren der Beiträge bedanken, die den Ausgangspunkt für dieses Buch bildeten: Hoang M. Le, Muhammad Hassan, Mingsong Chen, Jonas Wloka, Tim Güneysu, und Pascal Pieper. Wir danken insbesondere Hoang M. Le und Muhammad Hassan für viele interessante Diskussionen und eine erfolgreiche Zusammenarbeit. Nicht zuletzt gilt unser besonderer Dank Rudolf Herdt für zahlreiche anregende und hilfreiche Diskussionen.

Bremen, Deutschland — Vladimir Herdt
Bremen, Deutschland — Daniel Große
Bremen, Deutschland — Rolf Drechsler
Mai 2020

Inhaltsverzeichnis

1 Einführung . . . 1
- 1.1 Auf virtuellen Prototypen basierender Entwurfsablauf . . . 3
- 1.2 Buchbeitrag . . . 6
- 1.3 Buchorganisation . . . 9

2 Präliminarien . . . 11
- 2.1 SystemC TLM . . . 11
  - 2.1.1 TLM-basierte Kommunikation . . . 12
  - 2.1.2 Semantik der Simulation . . . 13
- 2.2 RISC-V . . . 15
  - 2.2.1 ISA-Übersicht . . . 15
  - 2.2.2 Erweiterung des Atomic Instruction Set . . . 16
- 2.3 Abdeckungsgesteuertes Fuzzing . . . 16
  - 2.3.1 LibFuzzer-Kern . . . 17
  - 2.3.2 LibFuzzer Erweiterungen . . . 18
- 2.4 Symbolische Ausführung . . . 18
  - 2.4.1 Übersicht . . . 19
  - 2.4.2 Beispiel . . . 19

3 Eine Open-Source RISC-V Evaluierungsplattform . . . 21
- 3.1 RISC-V-basierter virtueller Prototyp . . . 23
  - 3.1.1 RISC-V-basierte VP-Architektur . . . 24
  - 3.1.2 VP Interaktion mit SW und Umwelt . . . 27
  - 3.1.3 VP Leistungsoptimierungen . . . 33
  - 3.1.4 Simulation von Multi-Core-Plattformen . . . 34
  - 3.1.5 VP Erweiterung und Konfiguration . . . 37
  - 3.1.6 VP Bewertung . . . 42
  - 3.1.7 Diskussion und zukünftige Arbeit . . . 49

3.2 Schnelle und akkurate Leistungsbewertung für RISC-V 50
3.2.1 Hintergrund: HiFive1-Karte 51
3.2.2 Kern-Timing-Modell 51
3.2.3 Experimente 58
3.2.4 Diskussion und zukünftige Arbeit 61
3.3 Zusammenfassung 62

4 Formale Verifikation von SystemC-basierten Entwürfen durch symbolische Simulation 63
4.1 Zustandsbezogene symbolische Simulation 64
4.1.1 SystemC Intermediate Verification Language 68
4.1.2 Übersicht Symbolische Simulation 71
4.1.3 Zustand Subsumtionsreduzierung (SSR) 77
4.1.4 Überprüfung der symbolischen Subsumtion 85
4.1.5 Experimente 88
4.1.6 Diskussion und zukünftige Arbeit 94
4.2 Formale Verifikation eines Interrupt-Controllers 95
4.2.1 TLM-Peripherie-Modellierung in SystemC 96
4.2.2 Überbrückung der Modellierungslücke 98
4.2.3 Fallstudie 100
4.3 Kompilierte symbolische Simulation 104
4.3.1 Übersicht 105
4.3.2 Optimierungen 111
4.3.3 Experimente 115
4.3.4 Diskussion und zukünftige Arbeit 119
4.4 Parallelisierte kompilierte symbolische Simulation 120
4.4.1 Details zur Implementierung 120
4.4.2 Bewertung und Schlussfolgerung 124
4.5 Zusammenfassung 124

5 Abdeckungsgesteuertes Testen für skalierbare Verifikation virtueller Prototypen 127
5.1 Datenflusstests für virtuelle Prototypen 128
5.1.1 SystemC Laufendes Beispiel 129
5.1.2 Def-Use-Zuordnung und Datenflusstest 132
5.1.3 Datenflussprüfung für SystemC 132
5.1.4 Details zur Implementierung 138
5.1.5 Experimentelle Ergebnisse 140
5.2 Verifizierung von Befehlssatzsimulatoren mit Coverage-Guided Fuzzing 141
5.2.1 Abdeckungsgesteuertes Fuzzing für die ISS-Verifikation . . . 142
5.2.2 Fallstudie: RISC-V ISS-Verifikation 146
5.2.3 Diskussion und zukünftige Arbeit 151
5.3 Zusammenfassung 152

6 Verifizierung von eingebetteten Software-Binärdateien mit Hilfe virtueller Prototypen . . . 153
6.1 Concolic Testing von eingebetteten Binärdateien . . . 154
6.1.1 Hintergrund zur konkurrierenden Prüfung von SW . . . 156
6.1.2 Concolic Testing Engine für eingebettete RISC-V-Binärdateien . . . 157
6.1.3 Experimente . . . 164
6.1.4 Diskussion und zukünftige Arbeit . . . 168
6.2 Verifikation von eingebetteten Binärdateien mit Coverage-Guided Fuzzing . . . 170
6.2.1 VP-basiertes abdeckungsgeleitetes Fuzzing . . . 171
6.2.2 Experiment 1: Testen eingebetteter Anwendungen . . . 179
6.2.3 Experiment 2: Testen des Zephyr-IP-Stacks . . . 184
6.2.4 Diskussion und zukünftige Arbeit . . . 187
6.3 Zusammenfassung . . . 187

7 Validierung von Firmware-basiertem Power Management mit virtuellen Prototypen . . . 189
7.1 Ein Constraint-basierter Zufallsansatz für die Workload-Generierung . . . 191
7.1.1 Frühzeitige Validierung von FW-basierten Power-Management-Strategien . . . 192
7.1.2 SoCRocket Fallstudie . . . 197
7.1.3 Ergebnisse . . . 202
7.1.4 Diskussion und zukünftige Arbeit . . . 204
7.2 Maximierung der Kreuzabdeckung von Powerzuständen . . . 205
7.2.1 Maximierung der Kreuzabdeckung von Powerzuständen . . . 206
7.2.2 Fallstudie . . . 212
7.2.3 Diskussion und zukünftige Arbeit . . . 218
7.3 Zusammenfassung . . . 219

8 Register-Transfer-Ebene Korrespondenzanalyse . . . 221
8.1 Auf dem Weg zur vollautomatischen Verfeinerung von TLM-zu-RTL-Eigenschaften . . . 222
8.1.1 UTOPIA Fallstudie . . . 224
8.1.2 Statische Analyse von Transaktoren . . . 226
8.1.3 Eigenschaft Verfeinerung . . . 228
8.1.4 Diskussion und zukünftige Arbeit . . . 231
8.2 Automatisierte RTL-zu-TLM-Fehlerkorrespondenzanalyse . . . 231
8.2.1 RTL-zu-TLM-Fehlerkorrespondenzanalyse . . . 233
8.2.2 Formale Analyse der Fehlerlokalisierung . . . 239
8.2.3 Fallstudie . . . 245
8.3 Zusammenfassung . . . 248

9 Schlussfolgerung . . . 249

Literatur . . . 253

Liste der Algorithmen

1 Scheduler zur Durchführung der DFS-basierten symbolischen Simulation (SymEx+POR) für IVL 76

2 Zustandsabhängige symbolische Simulation (SSR+ SymEx+POR) für IVL: erweitert Algorithmus 1 durch Überschreiben der Funktionen expand, initialize und backtrack 84

3 Fehlerkorrespondenzanalyse 235

Abbildungsverzeichnis

Abb. 1.1 Vergleich des Konzepts eines traditionellen Entwurfsablaufs (linke Seite) und eines VP-basierten Entwurfsablaufs (rechte Seite) 2
Abb. 1.2 Überblick über den VP-basierten Entwurfsablauf (linke Seite) und die entsprechenden Beiträge dieses Buches (rechte Seite) 3

Abb. 2.1 Ausführungsphasen eines SystemC-Programms (aus [85]) 14
Abb. 2.2 Schnittstellenfunktion [141], die das SUT bereitstellt und die von libFuzzer während des Fuzzing-Prozesses wiederholt aufgerufen wird 17
Abb. 2.3 Schnittstellenfunktion [141], die das SUT bereitstellen kann, um eigene Mutatoren zu implementieren 18
Abb. 2.4 Beispielprogramm zur Veranschaulichung der symbolischen Ausführung 20
Abb. 2.5 Symbolischer Ausführungsbaum für das in Abb. 2.4 gezeigte Beispiel. Die Knoten entsprechen den Zeilennummern und sind mit dem symbolischen Ausführungsstatus nach der Ausführung dieser Zeile versehen. Der Knoten Zeile 12 ist mit dem symbolischen Ausdruck versehen, den der SMT-Solver abfragt, um zu entscheiden, ob die Behauptung in Zeile 12 verletzt wird 20

Abb. 3.1 Überblick über die Architektur des virtuellen Prototyps 24
Abb. 3.2 Beispielanwendung, die auf der VP läuft, um die Interaktion zwischen Hardware und Software zu demonstrieren 28
Abb. 3.3 Bare-Metal-Bootstrap-Code zur Demonstration der Interrupt-Verarbeitung 29
Abb. 3.4 Mit der C-Bibliothek verknüpfter Stub für die Systemaufrufbehandlung (Gastseite, ausgeführt auf dem VP-Hostsystem). Dieses Beispiel-Listing basiert auf dem RISC-V *newlib* port https://github.com/riscv/riscv-newlib 31

Abb. 3.5 Konzept der Ausführung von Systemaufrufen auf der VP, die entweder an das Hostsystem weitergeleitet werden oder einen Trap übernehmen 32
Abb. 3.6 Bare-Metal-Bootstrap-Code für eine Multi-Core-Simulation mit zwei Kernen 35
Abb. 3.7 Kernspeicherschnittstelle mit Unterstützung atomarer Operationen 36
Abb. 3.8 SystemC-basiertes konfigurierbares Sensormodell, das periodisch mit Zufallsdaten gefüllt wird – demonstriert die Grundprinzipien bei der Modellierung von Peripheriegeräten 39
Abb. 3.9 Die Transportfunktion für das Beispiel in Abb. 3.8 40
Abb. 3.10 Qt-basierte virtuelle Umgebung, die die HiFive1-Platine mit einer Siebensegmentanzeige (Ausgabe) und einer Taste (Eingabe) zeigt, die über eine TCP-Verbindung mit der VP-Simulation verbunden ist 43
Abb. 3.11 Überblick über den RISC-V Torture- und CGF-Ansatz für VP-Tests 44
Abb. 3.12 Überblick über unser Kernzeitmodell und die Integration mit dem CPU-Kernmodell 52
Abb. 3.13 Beispiel zur Veranschaulichung des Pipeline-Zeitmodells 53
Abb. 3.14 Zeitmodell für die Verzweigungsvorhersage mit vier Schnittstellenfunktionen (*Sprung*, *Aufruf*, *Rückkehr* und *Verzweigung*) zur Aktualisierung des Zeitmodells nach einem Sprung, einem Funktionsaufruf/einer Rückkehr bzw. einer Verzweigungsanweisung 55
Abb. 3.15 Beispiel zur Veranschaulichung des Verzweigungszeitmodells 57
Abb. 3.16 Zusammenfassung der Ergebnisse zur Genauigkeit im Vergleich zum HiFive1-Board (oberes Diagramm) und zum Leistungsoverhead im Vergleich zum ursprünglichen RISC-V VP (unteres Diagramm) unseres Ansatzes (VP + Core Timing Model) 61

Abb. 4.1 Ein SystemC-Beispiel 70
Abb. 4.2 Das Beispiel in IVL 70
Abb. 4.3 Übersicht über die symbolische Simulation 71
Abb. 4.4 Vollständiger Suchbaum für Beispiel 1 72
Abb. 4.5 Ein IVL-Beispiel 78
Abb. 4.6 Zustandsraum für das IVL-Beispiel 79
Abb. 4.7 Abstrakter Zustandsraum zur Veranschaulichung des (Transitions-)Ignorierungsproblems 81
Abb. 4.8 Skizze der Beweisidee für das Hauptsatzes 83
Abb. 4.9 Optimierte ESS-Überprüfung 87
Abb. 4.10 Prinzip der Filterprüfung 88
Abb. 4.11 Beispiel für TLM-Registermodellierung 97
Abb. 4.12 Registerklasse in XIVL 99

Abb. 4.13 Beispiel einer Callback-Implementierung in XIVL. 99
Abb. 4.14 Registerinitialisierung in XIVL 100
Abb. 4.15 IRQMP-Übersicht 101
Abb. 4.16 Einfaches XIVL-Beispielprogramm mit einem einzigen Thread, der iterativ die Summe für einen gegebenen symbolischen Eingabewert *n* berechnet. Dies wird im weiteren Verlauf dieses Abschnitts als laufendes Beispiel dienen 105
Abb. 4.17 CSS-Übersicht 106
Abb. 4.18 Datenstrukturen für die CSS-Transformation von Thread A zum Speichern und Zugreifen auf den lokalen und globalen Programmzustand (in Block 2 von Abb. 4.17). 108
Abb. 4.19 CSS-Transformation des Fadens A (in Block 2 von Abb. 4.17), der in Abb. 4.16 definiert wurde 109
Abb. 4.20 Ausführung von Verzweigungen mit symbolischen Bedingungen (in Block 4 von Abb. 4.17) 110
Abb. 4.21 Relevante Anweisungen für einen Funktionsaufruf (in Block 2 von Abb. 4.17) 111
Abb. 4.22 Beispiel für die Zusammenführung von Zweigen (in Block 2 von Abb. 4.17) 113
Abb. 4.23 Werkzeugübersicht. 121
Abb. 4.24 Symbolische Verzweigungsausführung 122
Abb. 4.25 Implementierung der parallelisierten Gabelung. 123

Abb. 5.1 SystemC-Beispiel (Teil 1) 130
Abb. 5.2 SystemC-Beispiel (Teil 2) 131
Abb. 5.3 Ein Überblick über unseren Datenfluss-Testansatz für SystemC. . . . 133
Abb. 5.4 Überblick über abdeckungsgesteuertes Fuzzing (CGF) für die ISS-Verifikation 143
Abb. 5.5 Konzept der Zuordnung von Informationen zur funktionalen Abdeckung zu Merkmalen. 147

Abb. 6.1 Überblick über die Architektur der Concolic Testing Engine (CTE). 158
Abb. 6.2 Einfaches Sensorperipheriegerät, das das Konzept der Peripheriemodellierung in Kombination mit der CTE-Schnittstelle veranschaulicht 160
Abb. 6.3 Beispiel einer SW, die auf die Sensorperipherie zugreift (wie in Abb. 6.2 gezeigt) 161
Abb. 6.4 Beispielhafte konkolische Ausführungspfade durch die in den Abb. 6.3 und 6.2 dargestellte SW und Peripherie 163
Abb. 6.5 Wrapper für die echten FreeRTOS-Speicherverwaltungsfunktionen *pvPortMalloc* und *vPortFree* 168
Abb. 6.6 Überblick über VP-CGF: CGF mit SystemC-basierten VPs zur Verifikation von eingebetteten SW-Binärprogrammen 172
Abb. 6.7 Beispiel einer in SystemC TLM implementierten Sensorperipherie 175

Abb. 6.8 Implementierung der Transportfunktion für das in Abb. 6.7 gezeigte Beispiel der Sensorperipherie . . . 176
Abb. 6.9 Beispiel einer eingebetteten SW, die auf die in Abb. 6.7 gezeigte (SystemC TLM) Sensorperipherie zugreift . . . 177
Abb. 6.10 SW-Schleife zum Lesen und Verarbeiten von Zeichen vom Eingang (RX) UART . . . 181
Abb. 6.11 Wrapper für dynamische Speicherverwaltungsfunktionen, um den Überblick über die aktuell (dynamisch) zugewiesenen Blöcke zu behalten und zur Laufzeit auf Pufferüberläufe zu prüfen . . . 186

Abb. 7.1 Überblick über die Validierung des Energiemanagements . . . 193
Abb. 7.2 Ein Constraint-basiertes Workloadszenario . . . 196
Abb. 7.3 Workload der Anwendung (Constraint-Lösung) . . . 196
Abb. 7.4 Abrufen der statischen Leistung einer Komponente auf der Grundlage ihres aktuellen Powerzustands . . . 199
Abb. 7.5 Abrufen und Zurücksetzen der Schaltleistung eines Bauteils auf der Grundlage seines aktuellen Powerzustands . . . 199
Abb. 7.6 Regelmäßig ausgelöst durch Interrupts zur Aktualisierung der Powerzustände der Hardwarekomponenten . . . 200
Abb. 7.7 Aktualisierung des CPU-Powerstatus auf der Grundlage des Duty-Cycle der CPU – regelmäßig ausgelöst durch Interrupts . . . 201
Abb. 7.8 Firmwarefunktion zum Lesen von Daten aus einem IO-Gerät . . . 202
Abb. 7.9 Beispiel eines Power-FSM einer Komponente, die eine *bedarfsgesteuerte* PM-Strategie umsetzt. L bezeichnet die Einschaltdauer (d. h. die Last) des Bauteils in der letzten Periode (Zeit zwischen den FW-Aktualisierungszyklen) . . . 206
Abb. 7.10 Überblick über unseren Ansatz zur Testfallerstellung . . . 207
Abb. 7.11 Konzeptueller Überblick über den Testgenerierungs- und -ausführungsprozess in der Coverage-Schleife . . . 209
Abb. 7.12 Abstraktes Beispiel zur Veranschaulichung des Ansatzes der Linienmodenverfeinerung in einem zweidimensionalen Raum . . . 210
Abb. 7.13 Beispielhafte Befehlsblöcke mit unterschiedlichen Typen zur Veranschaulichung: ein Arithmetik-, Speicher- und SPU-Zugriffsblock . . . 214

Abb. 8.1 Überblick über die Verfeinerung von TLM-zu-RTL-Eigenschaften . . . 222
Abb. 8.2 UTOPIA Controller TLM und RTL-Modell IO-Schnittstelle (vereinfacht) . . . 225
Abb. 8.3 Pseudocode für die Implementierung von TLM-zu-RTL-Transaktoren . . . 226
Abb. 8.4 Symbolischer Ausführungszustandsraum des Eingangstransaktors für die RTL-Schnittstellenfunktion für den Eingangstransaktor in Abb. 8.3, Pfadbedingungen wurden durch Eliminierung unbenutzter Beschränkungen optimiert . . . 227

Abb. 8.5 FSM für den TLM-zu-RTL-Transaktor (Eingang) 228
Abb. 8.6 FSM für den RTL-zu-TLM-Transaktor (Ausgang) 228
Abb. 8.7 Übersicht über die Fehlerkorrespondenzanalyse 234
Abb. 8.8 Beispiel für einen IRQMP-Codeauszug in VHDL mit Registerkonfiguration zur Veranschaulichung der Fehlerinjektion auf RTL (Zeile mit Fehlerinjektion hervorgehoben) 238
Abb. 8.9 TLM-Code für die Registerkonfiguration, der einen entsprechenden TLM-Fehler (Zeile hervorgehoben) für Abb. 8.8 zeigt 239
Abb. 8.10 Überblick über die formale Fehlerlokalisierungsanalyse 240
Abb. 8.11 Auszug aus einer kommentierten TLM-Registerklasse in XIVL . . . 242
Abb. 8.12 Kodierungsdetails für transiente Ein-Bit-Fehlerinjektion 243

Tabellenverzeichnis

Tab. 3.1 Experimentergebnisse – alle Ausführungszeiten in Sekunden und die Anzahl der ausgeführten Anweisungen (#instr) in Milliarden (B) 46
Tab. 3.2 Versuchsergebnisse – alle Ergebnisse der Simulationszeit werden in Sekunden angegeben 59

Tab. 4.1 Beispieldaten für das IVL-Beispiel 79
Tab. 4.2 Vergleich der ESS-Optimierungen (Laufzeit in Sekunden) 90
Tab. 4.3 Vergleich mit KRATOS (1)-Benchmarks mit azyklischen Zustandsräumen. 92
Tab. 4.4 Vergleich mit KRATOS (2)-Benchmarks mit zyklischen Zustandsräumen. 93
Tab. 4.5 Überprüfungsergebnisse 103
Tab. 4.6 Bewertung der nativen Ausführung (Laufzeiten in Sekunden) 116
Tab. 4.7 Vergleich mit bestehenden SystemC-Verifikatoren auf öffentlich verfügbaren Benchmarks (Laufzeiten in Sekunden). 117
Tab. 4.8 Virtuelle Prototyp-Benchmarks in der realen Welt (Laufzeiten in Sekunden) 118
Tab. 4.9 Versuchsergebnisse, T.O. bezeichnet Timeout (Grenze 750 s) 124

Tab. 5.1 Datenflussassoziationen für das SystemC-Beispiel in Abb. 5.1 und 5.2, sortiert nach Klassifikation 137
Tab. 5.2 Bewertungsergebnisse – alle Ausführungszeiten sind in Sekunden angegeben – [V1 … V7] bedeutet, dass alle 7 Fehler V1-V7 gefunden wurden 148
Tab. 5.3 Weitere Einzelheiten zur funktionalen Abdeckung (ergänzt Tab. 5.2). 149
Tab. 5.4 Beschreibung aller Fehler, die wir in jeder ISS gefunden haben 150

Tab. 6.1 Versuchsergebnisse – Timeout (T.O.) auf 7200 s (2 h) eingestellt. 165
Tab. 6.2 Fehler, die beim Testen des FreeRTOS-TCP/IP-Stacks gefunden wurden – Implementierungszeit in Sekunden 169
Tab. 6.3 Experimentelle Ergebnisse zur Anwendung unseres VP-CGF-Ansatzes zum Testen eingebetteter Anwendungen 180

Tab. 7.1 Versuchsergebnisse (Simulationszeit in Sekunden, Stromverbrauch in μJ) auf der SoCRocket-Plattform 203
Tab. 7.2 Beispiel für die Erzeugung von Blöcken aus einem Mischvektor durch Verschachtelung 212
Tab. 7.3 Einzelne Blockinformationen, die durch den ausschließlichen Betrieb des entsprechenden Blocks auf der VP (alle Komponenten im Volllastmodus, d. h. kein PM in FW aktiviert) und die Messung der Arbeitszyklen (Last) und der durchschnittlichen Laufzeit gewonnen wurden. 213
Tab. 7.4 Experimentelle Ergebnisse für unseren Ansatz 216

Tab. 8.1 Abbildung von TLM-auf-RTL-Eigenschaften mit Verfeinerungsschritten. 230
Tab. 8.2 Entsprechende TLM-Fehlerinjektionskandidaten (entsprechender Fehler hervorgehoben) 237
Tab. 8.3 Kombinierte Ergebnisse der Experimente (Laufzeiten in Sekunden) 246

Kapitel 1
Einführung

Eingebettete Systeme sind heute in vielen verschiedenen Anwendungsbereichen weit verbreitet, vom *Internet der Dinge* (IoT) über die Automobilindustrie und die Produktion bis hin zu Kommunikations- und Multimediaanwendungen. Eingebettete Systeme bestehen aus Hardware- (HW) und *Software-* (SW) Komponenten und sind in der Regel kleine, ressourcenbeschränkte Systeme, die hoch spezialisiert sind, um anwendungsspezifische Lösungen zu implementieren. Daher erfordern Entwurfsabläufe für eingebettete Systeme effiziente und flexible Techniken zur Explorationdes Entwurfsraums, um alle anwendungsspezifischen Anforderungen wie zum Stromverbrauch und der Ausführungsleistung zu erfüllen.

Darüber hinaus nimmt die Komplexität eingebetteter Systeme ständig zu. Sie integrieren immer komplexere IP (*Intellectual Property*)-Komponenten (wie Prozessorkerne, benutzerdefinierte Beschleuniger und andere HW-Peripheriegeräte) auf der HW-Seite und verlassen sich weitgehend auf die SW, um die IPs zu steuern und darauf zuzugreifen sowie andere wichtige Funktionen auszuführen. Insbesondere die SW-Komplexität hat in den letzten Jahren stark zugenommen, weshalb die SW im Designprozess mit gleicher Priorität wie die HW betrachtet werden sollte. Heutzutage umfasst die (eingebettete) SW mehrere Abstraktionsschichten, die vom Bootcode über Gerätetreiber bis hin zu vollwertigen Betriebssystemen in Kombination mit verschiedenen Bibliotheken (z. B. einem drahtlosen Netzwerkstack für die Kommunikation) zusätzlich zum eigentlichen Anwendungscode reichen. Darüber hinaus wird die SW aufgrund ihrer Flexibilität zunehmend auch zur Berücksichtigung nicht-funktionaler Aspekte genutzt, z. B. zur Implementierung von Strategien zur Steuerung der Energieverwaltung des eingebetteten Systems. Neben der SW-Entwicklung ist auch die Verifizierung der SW entscheidend, um Fehler und Sicherheitslücken zu vermeiden. Ähnlich wie die SW-Entwicklungszeit steigt auch die SW-Verifikationszeit mit zunehmender SW-Komplexität. Daher ist es sehr wichtig, so früh wie möglich mit der SW-Entwicklung und -Verifizierung zu beginnen, um die engen Zeitvorgaben für die Markteinführung einzuhalten und ein qualitativ hochwertiges Produkt zu liefern.

© Der/die Autor(en), exklusiv lizenziert an Springer Nature Switzerland AG 2022

V. Herdt et al., *Verbessertes virtuelles Prototyping*,
https://doi.org/10.1007/978-3-031-18174-0_1

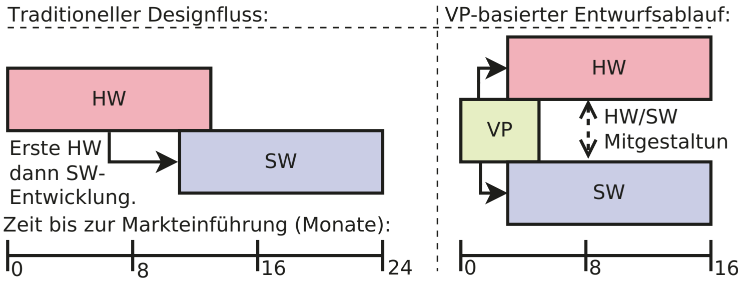

Abb. 1.1 Vergleich des Konzepts eines traditionellen Entwurfsablaufs (linke Seite) und eines VP-basierten Entwurfsablaufs (rechte Seite)

Der herkömmliche Entwurfsablauf für eingebettete Systeme ist unzureichend, um mit dieser zunehmenden Komplexität fertig zu werden. Der herkömmliche Entwurfsablauf funktioniert sequentiell, indem zuerst die HW und dann die SW entwickelt wird (wie auf der linken Seite von Abb. 1.1 dargestellt). Der Grund dafür ist, dass die SW die HW benötigt, um ausgeführt werden zu können, so dass die SW-Entwicklung beginnt, sobald die HW fast fertig ist. Dies wiederum führt zu erheblichen Verzögerungen im Entwurfsablauf, insbesondere aufgrund der steigenden Komplexität der SW.

Um dieses Problem zu lösen, werden zunehmend *virtuelle Prototypen* (VPs) für die SW-Ausführung in einem frühen Stadium des Entwurfsflusses eingesetzt und ermöglichen so die parallele Entwicklung von HW und SW [26, 31, 46, 139, 189]. Ein VP ist im Wesentlichen ein ausführbares abstraktes Modell der gesamten HW-Plattform und wird überwiegend in SystemC TLM (*Transaction Level Modeling*) erstellt [113, 163].[1] Aus Sicht der SW bildet ein VP die reale HW ab, d. h. er beschreibt die HW auf einer Ebene, die für die SW relevant ist. So bietet ein VP beispielsweise alle für die SW sichtbaren HW-Register, abstrahiert aber von komplexen internen Kommunikationsprotokollen. Da es sich jedoch um ein abstraktes Modell der realen HW handelt, kann der VP viel schneller entwickelt werden als die HW. Sobald der VP erstellt wurde, kann er zur Ausführung der SW verwendet werden und ermöglicht somit die SW-Entwicklung und -Tests zu einem frühen Zeitpunkt im Entwurfsablauf. Gleichzeitig dient der VP auch als ausführbares Referenzmodell für den nachfolgenden HW-Entwurfsablauf. Somit ermöglicht ein VP-basierter Entwurfsablauf die parallele Entwicklung von HW und SW (wie auf der rechten Seite von Abb. 1.1 dargestellt), was zu einer erheblichen Verkürzung der Markteinführungszeit führt und durch die Weitergabe von Feedback zwischen den HW- und SW-Entwicklungsflüssen auch einen agileren Entwicklungsfluss ermöglicht.

[1] Im Wesentlichen handelt es sich bei SystemC um eine C++-Klassenbibliothek, die einen ereignisgesteuerten Simulationskern enthält und grundlegende Bausteine zur Erleichterung der Entwicklung von VPs bereitstellt, während TLM die Beschreibung der Kommunikation in Form von abstrakten Transaktionen ermöglicht. Weitere Einzelheiten zu SystemC TLM werden später in den Vorbemerkungen in Abschn. 2.1 beschrieben.

Im Folgenden wird zunächst der VP-basierte Entwurfsablauf näher beschrieben und anschließend werden die Beiträge dieses Buches zur Verbesserung des VP-basierten Entwurfsablaufs vorgestellt.

1.1 Auf virtuellen Prototypen basierender Entwurfsablauf

Abb. 1.2 (auf der linken Seite) zeigt den VP-basierten Entwurfsablauf im Detail. Ausgangspunkt ist eine textuelle Spezifikation des eingebetteten Systems, die sowohl die funktionalen als auch die nicht-funktionalen Anforderungen an das eingebettete System spezifiziert. Der VP-basierte Entwurfsablauf selbst ist im Wesentlichen in vier verschiedene Schritte unterteilt (durch eine andere Hintergrundfarbe hervorgehoben):

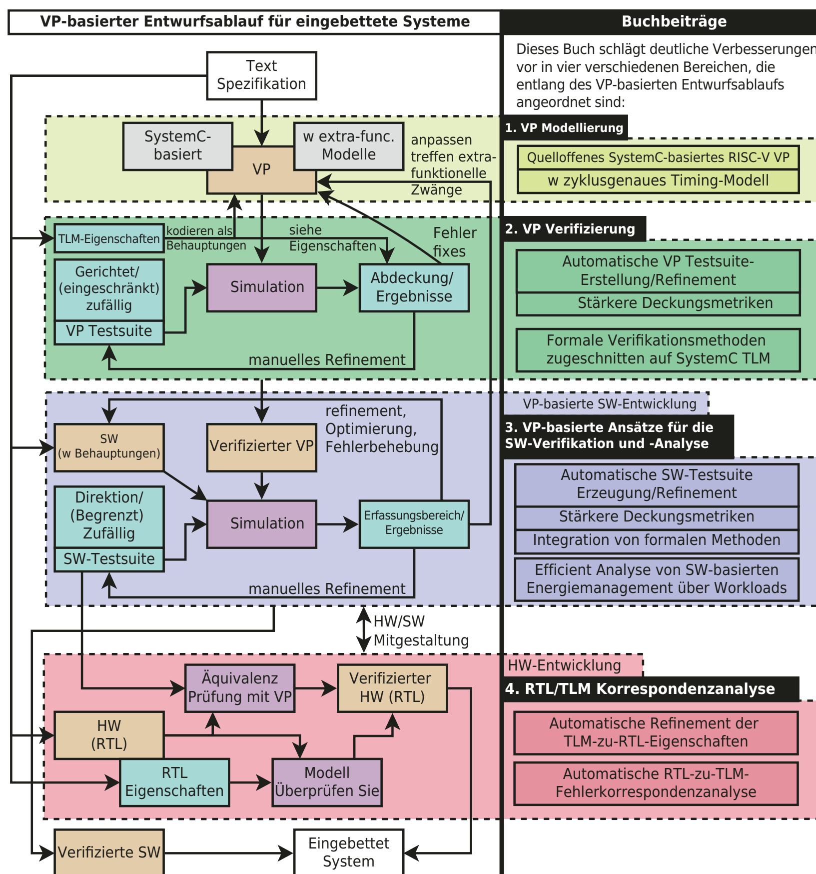

Abb. 1.2 Überblick über den VP-basierten Entwurfsablauf (linke Seite) und die entsprechenden Beiträge dieses Buches (rechte Seite)

1. VP Modellierung (gelb)
2. VP-Verifizierung (grün)
3. VP-basierte SW-Entwicklung (blau)
4. HW-Entwicklung (rot)

Bitte beachten Sie, dass diese vier Entwurfsschritte nicht sequentiell nacheinander ausgeführt werden, sondern ineinander verschachtelt sind, da z. B. die SW und die HW parallel entwickelt werden. Eine Übersicht über die einzelnen Schritte (einer nach dem anderen von oben nach unten, gruppiert durch gestrichelte Kästen und hervorgehoben durch die jeweilige Hintergrundfarbe) ist auf der linken Seite von Abb. 1.2 dargestellt. Im Folgenden werden die vier Schritte detaillierter beschrieben.

Schritt 1: VP-Modellierung
Im ersten Schritt wird der SystemC-basierte VP erstellt. Ein VP repräsentiert die gesamte HW-Plattform. Er kann aus einem oder mehreren allgemeinen oder speziellen Prozessoren sowie mehreren HW-Peripheriegeräten bestehen. Zusätzlich zum funktionalen Verhalten werden typischerweise nicht-funktionale Verhaltensmodelle in den VP integriert, um neben der SW-Ausführung frühe und genaue Schätzungen zum nicht-funktionalen Verhalten des Systems zu erhalten. Diese Schätzungen ermöglichen eine frühzeitige Exploration des Entwurfsraums und die Validierung nicht-funktionaler Einschränkungen wie Stromverbrauch und Ausführungsleistung/ Zeitplanung.

Schritt 2: VP-Überprüfung
Im nächsten Schritt wird der VP verifiziert, was sehr wichtig ist, da der VP als ausführbares Referenzmodell im nachfolgenden Entwurfsablauf dient. Zur Verifikation werden überwiegend simulationsbasierte Methoden eingesetzt. Sie erfordern eine umfassende Testsuite (d. h. eine Reihe von Testfällen), um eine gründliche Verifikation zu gewährleisten. Die Testfälle werden entweder manuell erstellt oder mit reinen Zufalls- und eingeschränkten Zufallsverfahren generiert. Jeder Testfall repräsentiert einen bestimmten Eingabestimulus für den VP. Um einen Testfall zu simulieren, wird eine Testbench erstellt, die die zu testende VP-Konfiguration instanziiert, dann die Eingangsstimuli an den VP weitergibt und das Ausgabeverhalten des VP beobachtet. Es ist möglich, den gesamten VP zu instanziieren oder einzelne VP-Komponenten (z. B. den Interrupt-Controller oder die CPU) isoliert zu testen. Das erwartete Verhalten wird in Form von TLM-Eigenschaften spezifiziert, die entweder direkt als (ausführbare) Assertions in den VP kodiert werden oder anhand der beobachteten Ergebnisse überprüft werden. Codeabdeckungsinformationen (z. B. Zeilen- und Verzweigungsabdeckung) werden verwendet, um die Qualität der Testsuite zu ermitteln und den Prozess der Testfallgenerierung zu steuern. Es ist jedoch ein erheblicher manueller Aufwand erforderlich, um gezielte Testfälle zu erstellen, die die Abdeckung maximieren. Der verifizierte VP

dient als ausführbares Referenzmodell für die nachfolgende SW- und HW-Entwicklung und ermöglicht die parallele Entwicklung von SW und HW.[2]

Schritt 3: VP-basierte SW-Entwicklung
Der primäre Anwendungsfall für den VP ist die *frühe* SW-Entwicklung. Neben der SW-Entwicklung und der Erkundung des Entwurfsraums wird der VP auch zur Durchführung zusätzlicher umfangreicher SW-Verifikations- und Analyseaufgaben verwendet. Ähnlich wie bei der VP-Verifikation selbst werden auch hier überwiegend simulationsbasierte Methoden eingesetzt. Diese simulationsbasierten Methoden arbeiten jedoch auf einer anderen Abstraktionsebene, da sie in erster Linie zum Testen der SW und nicht des VP konzipiert sind. Daher stellt jeder Testfall Eingabestimuli für die SW dar. Die Eigenschaften der SW werden entweder als Assertions in der SW selbst kodiert oder auf der Grundlage des beobachteten Ausgabeverhaltens des VP überprüft. Der Prozess der Testfallgenerierung basiert wiederum größtenteils auf manuell erstellten gerichteten Testfällen sowie auf (eingeschränkten) Zufallstechniken und wird durch die beobachteten Ergebnisse und Codeabdeckungsinformationen geleitet (in diesem Fall wird jedoch die SW-Abdeckung berücksichtigt). Ähnlich wie bei der Verifikation des VP ist ein erheblicher zeitaufwändiger und fehleranfälliger manueller Aufwand erforderlich, um eine umfassende Verifikation/Analyse der SW durchzuführen.

Auf der Grundlage der erzielten Ergebnisse wird die SW verfeinert und optimiert, um Fehler zu beheben und die anwendungsspezifischen Leistungs- und Stromverbrauchsanforderungen zu erfüllen. Insbesondere in den frühen Phasen des Entwurfsablaufs ist es auch möglich, den VP neu zu konfigurieren, um die Hardware so zu optimieren, dass sie die anwendungsspezifischen Anforderungen erfüllt.

Schritt 4: HW-Entwicklung
Die HW-Entwicklung findet parallel zur SW-Entwicklung statt. Das Ergebnis ist eine synthetisierbare *Register-Transfer-Level* (RTL)-Beschreibung der HW. Meistens werden zwei verschiedene Verifikationsmethoden eingesetzt:

- Durchführung eines Äquivalenztest der HW und des VP, indem die in Schritt 3 erhaltene(n) SW-Testsuite(n) wiederverwendet werden.
- Durchführung einer Modellprüfung anhand von RTL-Eigenschaften, die auf der Grundlage der textuellen Spezifikation erstellt wurden.

Schließlich können die verifizierte SW und HW in das endgültige eingebettete System integriert werden. Bitte beachten Sie, dass die HW-Entwicklung und HW-Verifikationsmethoden auf RTL-Ebene und darunter nicht im Fokus dieses Buches stehen.

[2] Es ist jedoch zu beachten, dass aufgrund der Komplexität der VP-Verifizierung und des erheblichen manuellen Aufwands zu Beginn nur eine vorläufige VP-Verifizierung durchgeführt wird und die VP-Verifizierung in der Regel ein laufender Prozess neben der SW- und HW-Entwicklung ist. Daher sind Verbesserungen im Bereich der VP-Verifikation sehr wichtig, um die Ausbreitung von Fehlern und damit kostspielige Iterationen zu vermeiden.

1.2 Buchbeitrag

Der VP-basierte Entwurfsablauf stellt eine erhebliche Verbesserung gegenüber dem traditionellen Entwurfsablauf dar und hat sich auch in mehreren industriellen Anwendungsfällen bewährt [46, 124, 126, 160]. Allerdings hat dieser moderne VP-basierte Entwurfsablauf immer noch Schwächen, insbesondere aufgrund des hohen manuellen Aufwands für Verifikations- und Analyseaufgaben, der sowohl zeitaufwendig als auch fehleranfällig ist. Dieses Buch schlägt mehrere neuartige Ansätze vor, um den VP-basierten Entwurfsablauf stark zu verbessern, und liefert damit Beiträge zu jedem der vier Hauptschritte des VP-basierten Entwurfsablaufs. Eine Übersicht über die Beiträge des Buches ist auf der rechten Seite von Abb. 1.2 dargestellt. Die Buchbeiträge sind in vier Bereiche gruppiert, die (größtenteils) den vier Schritten des VP-basierten Entwurfsablaufs entsprechen und im Folgenden näher erläutert werden:

1. VP Modellierung
2. VP Verifizierung
3. VP-basierte Ansätze für die SW-Verifikation und -Analyse
4. RTL/TLM-Korrespondenzanalyse

Beitragsbereich 1: VP-Modellierung
Der erste Beitrag dieses Buches ist ein in SystemC TLM implementierter Open-Source RISC-V VP.

RISC-V ist eine offene und freie *Instruction Set Architecture* (ISA) [218, 219], die lizenz- und lizenzgebührenfrei ist. Ähnlich wie die enorme Dynamik von Open-Source-SW gewinnt auch die quelloffene RISC-V ISA sowohl in der Industrie als auch in der Wissenschaft stark an Fahrt. Insbesondere für eingebettete Geräte, z. B. im IoT-Bereich, wird RISC-V zu einem Game Changer. Rund um RISC-V gibt es ein großes und ständig wachsendes Ökosystem, das von verschiedenen HW-Implementierungen (d. h. RISC-V-Cores) bis zu SW-Bibliotheken, Betriebssystemen, Compilern und Sprachimplementierungen reicht. Darüber hinaus sind mehrere *Open-Source-Hochgeschwindigkeits-Befehlssatzsimulatoren* (ISS) verfügbar. Diese ISSs sind jedoch in erster Linie für eine hohe Simulationsleistung ausgelegt und können daher kaum erweitert werden, um weitere Anwendungsfälle auf Systemebene zu unterstützen, wie z. B. die Exploration des Entwurfsraums, die Validierung von Energie/Timing/Leistung oder die Analyse komplexer HW/SW-Interaktionen. Das Ziel des vorgeschlagenen RISC-V VP ist es, diese Lücke im RISC-V-Ökosystem zu schließen und weitere Forschung und Entwicklung anzuregen.

Der VP bietet einen 32/64-Bit-RISC-V-Kern mit einer Reihe wichtiger Peripheriegeräte und Unterstützung für Multi-Core-Simulationen. Darüber hinaus bietet der VP auch SW-Debugging (über die Eclipse IDE) und Abdeckungsmessungen und unterstützt die Betriebssysteme FreeRTOS, Zephyr und Linux. Der VP ist als erweiterbare und konfigurierbare Plattform konzipiert (als Beispiel stellen wir eine

Konfiguration zur Verfügung, die dem RISC-V HiFive1 Board von SiFive entspricht), mit einem generischen Bussystem und in standardkonformem SystemC implementiert. Der letzte Punkt ist sehr wichtig, da er die Nutzung modernster SystemC-basierter Modellierungstechniken ermöglicht, die für die genannten Anwendungsfälle auf Systemebene benötigt werden. Schließlich ermöglicht der VP eine deutlich schnellere Simulation im Vergleich zu RTL und ist dabei genauer als bestehende ISSs.

Darüber hinaus integriert der VP ein effizientes Kern-Timing-Modell, das eine schnelle und genaue Leistungsbewertung für RISC-V-basierte Systeme ermöglicht. Das Timing-Modell ist mit dem Kern über eine Reihe von wohldefinierten Schnittstellen verbunden, die die funktionalen von den nicht-funktionalen Aspekten entkoppeln und eine einfache Neukonfiguration des Timing-Modells ermöglichen. Als Beispiel wird eine Timing-Konfiguration für das RISC-V HiFive1-Board von SiFive bereitgestellt.

Beitragsbereich 2: VP-Verifizierung
Dieses Buch verbessert den VP-Verifikationsablauf, indem es neuartige formale Verifikationsmethoden und fortgeschrittene automatisierte, abdeckungsgesteuerte Testverfahren berücksichtigt, die auf SystemC-basierte Designs zugeschnitten sind.

Mit formalen Verifikationsmethoden kann die Korrektheit eines SystemC-Entwurfs in Bezug auf eine Reihe von Eigenschaften nachgewiesen werden. Die formale Verifikation von SystemC-Entwürfen stellt jedoch eine große Herausforderung dar, da sie alle möglichen Eingaben sowie Prozessplanungsreihenfolgen berücksichtigen muss, was den Verifikationsprozess aufgrund des Problems der Zustandsraumexplosion sehr schnell unlösbar machen kann. In diesem Buch wurden fortgeschrittene symbolische Simulationstechniken für SystemC entwickelt, um die Zustandsraumexplosion zu entschärfen. Neben der grundlegenden symbolischen Simulationstechnik, die die Grundlage für eine effiziente Erkundung des Zustandsraums bildet, werden in diesem Buch mehrere wichtige Erweiterungen und Optimierungen vorgestellt, wie SSR (*State Subsumption Reduction*) und CSS (*Compiled Symbolic Simulation*). SSR unterstützt die Verifikation von zyklischen Zustandsräumen, indem es den erneuten Aufruf von symbolischen Zuständen verhindert und somit die Verifikation vollständig macht. CSS ist eine ergänzende Technik, die die symbolische Simulations-Engine eng in das zu verifizierende SystemC-Design integriert, um die Verifikationsleistung durch die Unterstützung der nativen Ausführung drastisch zu steigern. Obwohl die entwickelten formalen Techniken den Stand der Technik bei der formalen Verifikation von SystemC-Entwürfen erheblich verbessert haben, sind sie dennoch anfällig für eine Explosion des Zustandsraums. Formale Methoden sind auf einzelne VP-Komponenten isoliert anwendbar, z. B. um einen Interrupt-Controller zu verifizieren, wie dieses Buch später zeigen wird, aber sie sind immer noch nicht anwendbar, um den gesamten VP zu verifizieren.

Daher werden in diesem Buch auch fortgeschrittene abdeckungsgesteuerte Testverfahren untersucht, die auf Testfallgenerierung und Simulation beruhen. Im Vergleich zu den bestehenden simulationsbasierten Verifikationsabläufen werden in die-

sem Buch stärkere Überdeckungsmetriken sowie fortgeschrittene automatische Testfallgenerierung und -verfeinerungstechniken untersucht. Insbesondere werden die *Data Flow Testing* (DFT)-Abdeckung und das *Coverage-guided Fuzzing* (CGF) betrachtet, die sich beide im SW-Bereich als sehr effektiv erwiesen haben und auf die Verifikation von VPs zugeschnitten sind. Für DFT wurde eine Reihe von SystemC-spezifischen Abdeckungskriterien entwickelt, die die SystemC-Semantik der Verwendung von nicht-präemptivem Thread-Scheduling mit Shared-Memory-Kommunikation und ereignisbasierter Synchronisation berücksichtigen. CGF wird für die Verifikation von *Instruction Set Simulators* (ISSs) eingesetzt, d. h. ein abstraktes Modell eines Prozessorkerns, und wird durch die Integration von funktionaler Abdeckung und einem auf die ISS-Verifikation zugeschnittenen Mutationsverfahren weiter verbessert.

Beitragsbereich 3: VP-basierte Ansätze für die SW-Verifikation und -Analyse
Zunächst werden in diesem Buch neue VP-basierte Ansätze für die SW-Verifikation vorgeschlagen. Sie verbessern den bestehenden VP-basierten SW-Verifikationsablauf durch die Integration stärkerer Überdeckungsmetriken und die Bereitstellung automatischer Testfallgenerierungstechniken sowie die Nutzung formaler Methoden. Die Sicherstellung eines korrekten funktionalen Verhaltens ist sehr wichtig, um Fehler und Sicherheitslücken (z. B. Pufferüberläufe) zu vermeiden.

Der erste vorgeschlagene Ansatz integriert das konkolische Testen in den VP. Im Wesentlichen ist das konkolische Testen eine automatisierte Technik, die sukzessive neue Pfade durch das SW-Programm exploriert, indem sie symbolische Beschränkungen löst, die parallel zur konkreten Ausführung verfolgt werden. Die Integration von konkolischen Tests in VPs ist jedoch aufgrund der komplexen HW/SW-Interaktionen und Peripheriegeräte eine Herausforderung. Die vorgeschlagene Lösung ermöglicht eine hohe Simulationsleistung mit genauen Ergebnissen und vergleichsweise geringem Aufwand für die Integration von Peripheriegeräten mit konkolischen Ausführungsfunktionen.

Das konkolische Testen ist zwar sehr effektiv, beruht aber auf symbolischen Beschränkungen und ist daher anfällig für Skalierbarkeitsprobleme. Daher wird in diesem Buch ein zweiter SW-Verifizierungsansatz vorgeschlagen, der modernste CGF in Kombination mit VPs für die Verifizierung von eingebetteten SW-Binärdateien einsetzt. Um den Fuzzing-Prozess zu steuern, wird die Abdeckung der eingebetteten SW mit der Abdeckung der SystemC-basierten Peripherie der VP kombiniert. Dieser zweite Ansatz stützt sich nicht auf formale Methoden und leidet daher nicht unter umfangreichen Skalierbarkeitsproblemen. Gleichzeitig liefert er aufgrund der involvierten automatisierten Abdeckungsrückkopplungsschleife wesentlich bessere Ergebnisse als (eingeschränkte) Zufallstests.

Neben einem korrekten Funktionsverhalten ist eine hohe Leistung in Kombination mit einem geringen Stromverbrauch eine wichtige Anforderung für viele eingebettete Systeme. Aufgrund ihrer Benutzerfreundlichkeit und Flexibilität werden Power-Management-Strategien häufig in SW implementiert. PM-Strategien werden durch Beobachtung des nicht-funktionalen Verhaltens des VP neben der SW-Ausführung analysiert.