Hybrides Testverfahren für Simulink/TargetLink-Modelle

vorgelegt von
Dipl.-Inf.
Benjamin Wilmes
aus Berlin

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
— Dr.-Ing. —

genehmigte Dissertation

Promotionsausschuss

Prof. Dr. Uwe Nestmann (Vorsitzender)
Prof. Dr. Stefan Jähnichen (Berichter)
Prof. Dr. Andreas Spillner (Berichter)
Prof. Dr. Steffen Helke (Berichter)

Tag der wissenschaftlichen Aussprache:

19.11.2014

Berlin 2015
D 83

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Softwaretechnik
Ernst-Reuter-Platz 7
D-10587 Berlin

Hochschule Bremen
Fakultät Elektrotechnik und Informatik
Zentrum für Informatik und Medientechnologien
Flughafenallee 10
D-28199 Bremen

Brandenburgische Technische Universität Cottbus-Senftenberg
Juniorprofessur Sichere Softwaresysteme
Postfach 10 13 44
D-03013 Cottbus

Benjamin Wilmes: *Hybrides Testverfahren für Simulink/TargetLink-Modelle*

VERÖFFENTLICHUNGEN

Die in dieser Dissertation vorgestellte Arbeit entstand von **2010** bis **2014** an der Technischen Universität Berlin (Daimler Center for Automotive Information Technology Innovations). Ein Teil der Ideen und Darstellungen dieser Arbeit sind bereits in den im Folgenden aufgelisteten Veröffentlichungen erschienen:

KONFERENZBEITRÄGE

- **B.Wilmes**: *Automated Structural Testing of Simulink/TargetLink Models via Search-Based Testing assisted by Prior-Search Static Analysis*, 4th International Conference on Advances in System Testing and Validation Lifecycle (VALID), November 2012, Lissabon, Portugal, ISBN 978-1-61208-233-2

- **B.Wilmes**: *Toward a Tool for Search-Based Testing of Simulink/TargetLink Models*, 4th Symposium on Search Based Software Engineering (SSBSE), Fast Abstracts, Fondazione Bruno Kessler, September 2012, Riva del Garda, Italien, ISBN 978-88-905389-9-5

- **B.Wilmes**, F. Lindlar, A. Windisch: *Suchbasierter Test für den industriellen Einsatz*, 4. Symposium Testen im System- und Software Life-Cycle, Technische Akademie Esslingen, November 2011, Ostfildern, Deutschland, ISBN 3-92-4813-92-2

JOURNALBEITRÄGE

- **B. Wilmes**: *Static Preprocessing for Automated Structural Testing of Simulink Models*, International Journal On Advances in Systems and Measurements, IARIA, Ausgabe 6, Nummer 3&4 2013, Seiten 310-323, Dezember 2013, ISSN 1942-261x

KURZFASSUNG

Die modellbasierte Entwicklung der Software eingebetteter Systeme hat sich in vielen Industriezweigen durchgesetzt, allen voran in der Automobilindustrie. Die Programmierung von Softwarecode erfolgt hierbei auf grafischem Wege, meist durch spezielle Datenflussmodelle, wie sie das weit verbreitete Werkzeug Matlab/Simulink bereitstellt. Der Code wird aus diesen Modellen in der Regel automatisiert abgeleitet, häufig unter Nutzung der Simulink-Erweiterung TargetLink. Derlei Modelle besitzen als maschinell ausführbarer Ursprung der Softwarefunktionalität eine hohe Relevanz für die ohnehin im Automobilbereich sehr bedeutsame Qualitätssicherung. In Anbetracht des zunehmenden Anteils von Software in Fahrzeugen, einhergehend mit steigenden Aufwänden für deren Test, ist eine größtmögliche Testautomatisierung wünschenswert. Der suchbasierte Test hat sich als ein geeignetes, bezüglich seiner Effizienz allerdings noch verbesserungswürdiges Automatisierungsverfahren erwiesen. Er eignet sich unter anderem, um Eingaben (Testdaten) zur strukturellen Überdeckung eines Modells, welches unter Verwendung von Matlab/Simulink und TargetLink entwickelt wurde, zu finden.

Diese Arbeit verfolgt das Ziel, die Effizienz des suchbasierten Tests in Anwendung für solche Modelle zu steigern. Hierzu kombiniert sie den suchbasierten Test mit unterstützenden Techniken. Das entstehende hybride Testverfahren enthält zum einen drei speziell entwickelte statische Analysetechniken, welche der Vorbereitung eines suchbasierten Tests dienen. So wird ermittelt, ob das Finden gewünschter Testdaten überhaupt möglich ist, ob es Testdaten gibt, die vorab von einer Betrachtung ausgeschlossen werden können, und in welcher Reihenfolge für die einzelnen (strukturorientierten) Ziele geeignete Testdaten zu suchen sind. Zum anderen wird der suchbasierte Test um eine in geeigneten Fällen stattfindende Testdatengenerierung mittels symbolischer Ausführung und SMT-Solving ergänzt. Darüber hinaus wird ein alternativer Suchalgorithmus vorgestellt, welcher den Raum aller möglichen Testdaten im Vergleich zum sonst verwendeten genetischen Algorithmus nicht global, sondern lokal untersucht.

Fallstudien untersuchen die Auswirkungen dieser Beiträge anhand zweier realer Modelle aus der Automobilindustrie. Die Ergebnisse dieser Studien zeigen, dass die realisierte Hybridisierung des suchbasierten Tests im betrachteten Kontext zu einer deutlichen Effizienzsteigerung führt – die Laufzeiten der Testdatengenerierungsvorgänge fielen um über 90% geringer aus, ohne dass bezüglich der Qualität der Ergebnisse, wie dem Überdeckungsgrad, Einbußen hingenommen werden mussten.

ABSTRACT

Model-based development of embedded systems software has become a well-established practice in many industry branches, especially the automotive industry. Such an approach involves programming via graphical means, most often through the use of specialized data-flow models, such as those provided by the widely-used tool Matlab/Simulink. Normally, the software code is automatically generated from these models, often through the use of the Simulink extension TargetLink. Simulink models are usually the first executable artifacts in the development process. Testing these models is therefore particularly relevant to the aspect of the automotive industry dealing with quality assurance. Considering the ever-expanding role of software in modern automobiles, going hand in hand with the rising testing costs, the automation of testing activities is highly desirable. One promising technique which has shown its capabilities in automating software testing is search-based testing. Among other applications, it can be utilized to generate input data (test data) for structural coverage of a Simulink model.

The aim of this work is to raise the efficiency of search-based test data generation when applied to such models. To achieve this goal, search-based testing is combined with supportive techniques. The resulting hybrid test procedure contains three specially designed static analysis techniques which enhance the setup of search-based testing. These techniques determine (1) whether it is even possible to find desired test data, (2) if there is test data present that can be exempted from the search in advance, and (3) in what order the individual (structure-oriented) goals should be aimed at by the search. In addition, for certain goals, the hybrid test procedure is generating test data by symbolic execution and SMT-Solving, in place of a search. Furthermore, an alternative search algorithm is presented, which, unlike the elsewise used genetic algorithm, explores the test data space locally, rather than globally.

Case studies investigate the effect of these additional techniques on the basis of two real models from the automotive industry. The results of these studies show, that the provided and implemented hybridization of search-based testing results in a clear rise in efficiency. The run times of test data generation operations is over 90% shorter, yet the quality of the results, such as the structural coverage, are not compromised.

DANKSAGUNG

Als Autor der vorliegenden Arbeit möchte ich mich an dieser Stelle bei den Personen bedanken, die mich bei der Erarbeitung dieser Arbeit unterstützt und zu ihrem Gelingen beigetragen haben.

Mein ausdrücklicher Dank gilt der Daimler AG und Prof. Dr. Stefan Jähnichen von der Technischen Universität Berlin, ohne die eine Durchführung dieser Arbeit nicht möglich gewesen wäre. Insbesondere möchte ich Dirk Johanson, Christian Scheidler und Michael Weber von der Daimler AG für die jahrelange Unterstützung meines Vorhabens danken. Herzlich danke ich auch Prof. Dr. Andreas Spillner und Prof. Dr. Steffen Helke für die Begutachtung dieser Arbeit. Darüber hinaus gilt mein Dank Dr. Andreas Windisch, dessen Dissertation den Ausgangspunkt dieser Arbeit darstellte.

Mein freundschaftlicher Dank gilt den Kollegen meiner Zeit am DCAITI, namentlich Thomas Noack, Dr. Felix Lindlar, Quang Minh Tran, Kerstin Hartig und Dr. Thomas Karbe. Die vielen interessanten und kreativen Diskussionen, die ich mit Euch hatte, wie auch Hilfestellungen und Hinweise von Euch, haben maßgeblich zum Fortschreiten und Erfolg dieser Arbeit beigetragen. Dr. Thomas Karbe gebührt hierbei für seine ausgiebige Durchsicht der schriftlichen Ausarbeitung mein besonderer Dank. Auch den weiteren Mitarbeitern des Fachgebiets Softwaretechnik von Prof. Jähnichen bin ich für den gemeinsamen Austausch und die hilfreichen Anmerkungen bei gemeinsamen Klausurtagungen dankbar.

Von ganzem Herzen danke ich, nicht zuletzt, meiner Frau Tanya Wilmes, die mich stets unterstützt und mir den nötigen Rückhalt gegeben hat. Unseren beiden kleinen Söhnen Oscar und Elias, die das Laufen lernten, indem sie sich an dem Bürostuhl hochzogen, wenn ihr Vater auch an den Wochenenden an dieser Arbeit schrieb, sei gesagt: Papa hat nun endlich wieder etwas mehr Zeit.

Benjamin Wilmes
Berlin, März 2015

INHALTSVERZEICHNIS

Teil I

GRUNDLAGEN

1 EINFÜHRUNG

Die Begriffe Optimierung, Automatisierung und Effizienz haben den menschlichen Fortschritt seit seinen Anfängen bis zum heutigen Tage dominiert und geprägt. Angefangen mit einfachen Werkzeugen, über Produktionssysteme, bis hin zu Prozessgestaltungen und wirtschaftlichen Bestrebungen: Aus dem Leben einer fortschrittlichen Gesellschaft und aus der industriellen, betrieblichen Welt sind diese drei Begriffe nicht mehr wegzudenken. Dies gilt gleichermaßen für Wissenschaft sowie Forschung und erst recht für Informatik und Softwaretechnik.

Es lassen sich eine Vielzahl an Definitionen und Bedeutungen für diese Begriffe finden. Etliche Bereiche und Disziplinen, wie im Falle der Optimierung beispielsweise die Mathematik oder im Falle der Effizienz der Wirtschaftsbereich, haben sie für sich vereinnahmt. Und doch bleibt unabhängig von ihrer Verwendung stets festzustellen: Optimierung, Automatisierung und Effizienz stehen einander sehr nahe und bedingen einander häufig. Beispielsweise sorgt eine Automatisierung im Idealfall für erhöhte Effizienz. Andererseits sollte eine Automatisierung selbst ausreichend effizient sein, um ihren Einsatz zu rechtfertigen. Eine Automatisierung kann im eingesetzten Kontext eine Optimierung darstellen. Optimierungen können aber auch Teil einer Automatisierung sein.

Die vorliegende Arbeit hat sich diesen drei Begriffen ganz besonders verschrieben. Sie behandelt eine Automatisierung im Bereich des Softwaretests: ein automatisches Finden geeigneter Testfälle. Die Automatisierung entsteht dabei durch Techniken zur Optimierung: Testfälle werden hinsichtlich ihrer Eignung optimiert. Und das Hauptziel der vorliegenden Arbeit ist die Steigerung der Effizienz dieser Automatisierung. Die folgenden Seiten werden Klarheit in diese abstrakte Umschreibung bringen. Hierbei spielt auch der Begriff der Hybridisierung eine zentrale Rolle. Auch für diesen Begriff lassen sich eine Vielzahl an Definitionen finden, beispielsweise in der Kraftfahrzeugtechnik. Dort bezeichnet der Begriff das Zusammenspiel verschiedenartiger Antriebseinheiten, wie Verbrennungsmotor oder Elektromotor, und Energiespeichersysteme – mit dem Ziel, die Effizienz hinsichtlich Kraftstoffverbrauch und Leistung zu optimieren.

QUALITÄT DER SOFTWARE EINGEBETTETER SYSTEME

Nun ist diese Arbeit zwar auch im automobilen Umfeld entstanden, allerdings adressiert sie ein anderes Themengebiet. Dieses ist aus Kundensicht zwar weniger greifbar, aber für die Automobilindustrie von hoher Relevanz. Es geht um die Qualität von Software, welche für eingebettete Systeme entwickelt wird, und deren Absicherung. Eingebettete Systeme sind speziell auf ihren Anwendungsfall zugeschnittene Computer, welche üblicherweise zur Erfüllung ihrer Funktion eine Software betreiben. Derartige Systeme steuern heutzutage vielfältige Geräte, Maschinen und Anlagen [110]. Seit vielen Jahren sind eingebettete Systeme, in Form von sogenannten Steuergeräten, auch ein integraler Bestandteil moderner Fahrzeuge – sei es zu Zwecken einer effizienten Motorsteuerung, dem Auf- und Zuschließen des Fahrzeugs per Knopfdruck, dem Auslösen eines Airbags oder, betrachtet man aktuelle Entwicklungen, selbstständig (autonom) fahrenden Fahrzeugen. Die Anforderungen an Sicherheit, Zuverlässigkeit und Haltbarkeit sind in diesem Bereich in der Regel sehr hoch [106], denn Nachlässigkeiten in der Entwicklung haben mitunter drastische Folgen. Der Automobilhersteller Toyota sah sich zum Beispiel jüngst mit den Konsequenzen eines fehlerhaften Motorsteuergeräts konfrontiert. Eine Person starb, weil das Fahrzeug unkontrolliert beschleunigte. Eine Reihe von Untersuchungen machte Unzulänglichkeiten in der Software der Motorsteuerung sowie Verstöße gegen branchenübliche Entwicklungsrichtlinien aus [12]. Die Qualitätssicherung erfährt also, auch angesichts der steigenden Zahl softwarebasierter Fahrzeugfunktionen, einen erhöhten Stellenwert.

MODELLBASIERTE ENTWICKLUNG UND TEST

Der Test ist dabei die in der Praxis am intensivsten genutzte Maßnahme zur Qualitätssicherung. Das Vorgehen bei der Entwicklung orientiert sich üblicherweise am V-Modell. Dieses zeichnet sich unter anderem dadurch aus, dass das Produkt in seiner Planung graduell in kleinere Einheiten zerlegt wird, diese Einheiten entwickelt und anschließend integriert werden, um so das finale Produkt zu bilden. Wird beispielsweise ein System für ein Fahrzeug entwickelt, so wird dieses eventuell auf mehrere Steuergeräte verteilt. Die Software wiederum wird üblicherweise in mehrere Bausteine (Module) aufgeteilt, wie Klassen oder Funktionen. Die Testaktivitäten orientieren sich an den bei den jeweiligen Entwicklungs- und Integrationsstufen entstehenden System- und Softwareartefakten. Wird modellbasiert entwickelt, so bedeutet dies, dass vor allem die Funktionsmodelle, der hieraus automatisch generierte Softwarecode, der integrierte Softwarecode, das Steuergerät mit der eingebetteten Software, die integrierten Steuergeräte sowie das Gesamtsystem zu Testzwecken herangezogen werden. Mithilfe systematischer Testverfahren werden derlei Testobjekte überprüft und abgesichert. Die Entwicklung

der Funktionsmodelle, welche wie auch der resultierende Code Software-Module repräsentieren, erfolgt in der Automobilindustrie in der Regel mit dem Editor *Simulink* (SL) [114], der Teil des Werkzeugs *Matlab* (ML) [113] von *The Mathworks* ist. Zur Codegenerierung wird häufig das auf SL aufsetzende *TargetLink* (TL) [36] von *dSpace* eingesetzt. TL stellt gewisse Anforderungen an ein SL-Modell. Ist im Folgenden also von SL/TL-Modellen, oder manchmal auch einfach nur von Modellen, die Rede, so sind damit TL-konforme SL-Funktionsmodelle gemeint. Eine modellbasierte Entwicklung mit SL findet nicht nur in der Automobilindustrie, sondern auch in anderen Industriezweigen statt, beispielsweise bei der Entwicklung von Satelliten [112].

Erfolgt die Entwicklung wie zuvor beschrieben, so sind SL/TL-Modelle in der Regel die ersten entstandenen Entwicklungsartefakte, welche maschinell ausführbar oder simulierbar sind. Daher kommt ihnen bei der Absicherung in der Theorie zwar ein hoher Stellenwert zu – allerdings wird dieser Tatsache in der Testpraxis erfahrungsgemäß leider nicht immer Rechnung getragen. In einer historisch stark vom Maschinenbau dominierten Domäne liegt der Fokus auch heute noch oftmals zu sehr auf dem Test von Entwicklungsartefakten mit einem höheren Abstraktionsgrad. Dies bedeutet, der Test der Gesamt-Software oder des resultierenden Systems, insbesondere der Steuergeräte, erfolgt häufig gründlicher als der Test der SL/TL-Modelle (Modelltest). Ein hoher Zeit- oder Innovationsdruck in der Entwicklung verstärkt diese Priorisierung. Eine Vernachlässigung solcher Tests birgt jedoch das Risiko, dass bestimmte Fehler erst spät im Entwicklungsprozess gefunden werden und in Folge mit höherem Aufwand und Kosten behoben werden müssen. Zudem könnten manche Fehler auf späteren Integrationsstufen schwerer auffindbar sein.

AUTOMATISIERTE TESTDATENGENERIERUNG

Grundsätzlich ist man in der Entwicklung automobiler Softwaresysteme um eine Automatisierung der auszuführenden Testaktivitäten bemüht. Dies gilt insbesondere für die für gewöhnlich aufwändigste Testaktivität: die Wahl geeigneter Testfälle, beziehungsweise der Testobjekt-Eingaben (Testdaten). Nun ist eine Automatisierung dieser Aktivität beim Modelltest einerseits sehr attraktiv, da das Modell – wie erwähnt – das erste ausführbare Entwicklungsartefakt ist und viele Fehler bereits direkt an ihrer Quelle beseitigt werden können – und andererseits, weil beim Modelltest ein Blick in das Innere des Testobjekts möglich ist. Dies ist beim Test eines Steuergeräts beispielsweise nicht der Fall.

Automatisierte Testdatengenerierung ist ein Forschungsthema, welches eine Vielzahl an Ansätzen und Techniken hervorgebracht hat. Mit der *symbolischen Ausführung* [72], dem *Zufallstest* [99], dem *suchbasierten Test* [83], dem *Concolic Testing* [49, 109] und *Model-Checking*-Techniken [41]

seien an dieser Stelle nur die Grundtechniken der Arbeiten genannt. Vornehmlich wurden die Ansätze allerdings für den Test von Programmcode entwickelt. Wie sich in dieser Arbeit zeigen wird, bringt die Testdatengenerierung für SL/TL-Modelle aus dem Automobilbereich zusätzliche Herausforderungen mit sich. Der suchbasierte Test, welcher metaheuristische Such- und Optimierungsalgorithmen zur Testdatenfindung instrumentiert, hat sich in vorherigen Arbeiten als ein geeignetes Verfahren zur Testdatengenerierung für SL/TL-Modelle erwiesen. Die vorliegende Arbeit baut daher auf entsprechenden Vorarbeiten auf, insbesondere auf der Dissertation von Windisch [132].

Der suchbasierte Test eignet sich unter anderem, um Testdaten zur Überdeckung der inneren Struktur eines Testobjekts zu finden. Der Begriff Überdeckung bezeichnet hierbei das Erreichen bestimmter Zustände innerhalb des Testobjekts, beziehungsweise von strukturellen Elementen wie Programmanweisungen oder -verzweigungen, während dessen Ausführung. Diese Art von Tests wird auch als White-Box-Test oder Strukturtest bezeichnet. Eine weitere Anwendungsmöglichkeit liegt im Bereich des Funktionstests, bei welchem die Einhaltung der Spezifikation, auf deren Grundlage die Entwicklung erfolgt ist, durch das Testobjekt geprüft wird. Erste Arbeiten zum suchbasierten Test wurden bereits in den 1970er Jahren veröffentlicht. Obwohl der Ansatz seit Beginn der 1990er Jahre mit steigenden Veröffentlichungszahlen einen enormen Aufschwung erlebt hat, führt er sein Dasein nach wie vor hauptsächlich in der Forschungswelt. Dafür gibt es folgende Gründe: Erprobungen mit industriellen Fallbeispielen bestätigten zwar häufig die Fähigkeit des suchbasierten Tests, ein äußerst generisch einsetzbares Automatisierungsverfahren zu sein. Allerdings stehen den guten Ergebnissen teils hohe Laufzeiten in der Durchführung gegenüber. Diese Schwäche stellt, so lässt sich zumindest im Falle des suchbasierten Tests von SL/TL-Modellen urteilen, das wichtigste Hindernis für den Einsatz des Verfahrens in der Praxiswelt dar. Die vorliegende Arbeit hat sich zum Ziel gesetzt, dieses Hindernis zu überwinden.

FOKUS

Aufgrund der Relevanz von SL/TL-Modellen in der Automobilindustrie fokussiert sich die Arbeit in der Anwendung der Automatisierungsmethoden auf diese. Es ist allerdings zu betonen, dass SL/TL-Modelle letzten Endes nur als Vertreter einer ganzen Klasse von Modelltypen herangezogen werden: Datenflussdiagramme, welche dynamische Systeme beschreiben. Dynamische Systeme berücksichtigen in ihrer Funktion den Faktor Zeit. Auch wenn die vorgestellten Lösungen im Detail an Spezifika von SL/TL angepasst sind, so sind das grundlegende Konzept und die entstandenen Techniken auf verwandte Modelltypen übertragbar.

Wie zuvor angedeutet, gibt es mehrere Arten von Tests. Der Strukturtest ist dabei nur eine Testart, bei der sich eine automatisierte Testdatengenerierung anbietet oder diese vorstellbar ist. Eine Betrachtung der in Frage kommenden Testarten, beziehungsweise der Szenarien einer automatisierter Testdatengenerierung – namentlich der Strukturtest, der Funktionstest, der Back-to-Back-Test und der entwicklungsbegleitende Test – ist Teil dieser Arbeit. Diese Betrachtung kommt zu dem Schluss, dass die Gestaltung eines Verfahrens zur automatisierten Testdatengenerierung, im Falle eines Funktionstests allerdings nur zum Teil, unabhängig von der betrachteten Testart ist. In jedem der betrachteten Anwendungsszenarien gilt es, Testdaten zu finden, welche bei Ausführung des Modells das Erreichen bestimmter Zustände im Modell bewirken. Die Arbeit setzt daher den Fokus auf den Strukturtest und betrachtet ihn als Repräsentant für die anderen möglichen Testarten.

HERAUSFORDERUNG

Eine effiziente Testdatengenerierung für SL/TL-Modelle ist aus mehreren Gründen besonders herausfordernd. Der Umstand, dass solche Modelle dynamische Systeme beschreiben, erhöht ihren Zustandsraum ungemein. Um alle Zustände erreichen zu können, müssen Testdaten aus Signalen, also Sequenzen von Werten, bestehen. Dies führt zu einer sehr großen Menge möglicher Testdaten, aus der es die Richtigen auszuwählen gilt. Automatisch generierte Testdaten müssen zudem plausibel sein, das heißt bezüglich der Funktionalität des Testobjekts realitätsnahe Szenarien beschreiben. Entsprechende Randbedingungen gilt es zu berücksichtigen. Hinzu kommt, dass SL/TL-Modelle aus der industriellen Praxis häufig sehr umfangreich sind. Um Testdaten ermitteln zu können, muss also eine entsprechend mächtige Funktionalität untersucht werden.

IDEE

Zur Steigerung der Effizienz einer automatisierten Testdatengenerierung setzt diese Arbeit auf eine Hybridisierung verschiedener Techniken. Insbesondere geht es um die Kombination statischer und dynamischer Techniken. Im Bereich der Programmanalyse und des Softwaretests wird eine Technik als dynamisch bezeichnet, wenn das Programm oder das Testobjekt durch sie zur Ausführung kommt [38]. Andernfalls ist von einer statischen Technik die Rede. Hierzu jeweils ein Beispiel: Der suchbasierte Test ist in der in dieser Arbeit betrachteten Anwendung eine dynamische Technik, da das Modell mit den generierten Testdaten ausgeführt wird. Bei einer symbolischen Ausführung oder einer statischen Analyse hingegen handelt es sich, wie die Namen der Techniken bereits preisgeben, um statische Techniken.

Die Idee besteht nun darin, statische und dynamische Techniken gegenseitig von den Stärken des Anderen profitieren zu lassen und hierdurch individuelle Schwächen oder Restriktionen auszumerzen. So müssen statische Verfahren oftmals Abstriche in der Genauigkeit ihrer Ergebnisse machen, da Abstraktionstechniken notwendig sind, um überhaupt mit umfangreichen Problemstellungen umgehen zu können. Dynamische Verfahren hingegen arbeiten in der Regel zwar präzise, liefern aber keine verallgemeinernden Ergebnisse. Ein für die Effizienz einer Lösung wichtiges Kriterium besteht zudem in der Geschwindigkeit, mit der statische und dynamische Techniken ihre Aufgaben erledigen. Statische und dynamische Techniken unterscheiden sich diesbezüglich häufig in Abhängigkeit von der Komplexität des zu lösenden Problems. Während statische Techniken kleine, wenig komplizierte Problemstellungen meist schneller bearbeiten, lohnt sich der Einsatz dynamischer Techniken oftmals genau dann, wenn statische Techniken an Komplexitätsgrenzen stoßen.

ARBEITSTHESE UND BEITRÄGE

Aus dieser Motivation heraus stellt sich diese Arbeit folgender These:

These. *Die Effizienz des Ansatzes von Windisch zur suchbasierten Testdatengenerierung für SL-Modelle lässt sich durch eine Hybridisierung derselben mit geeigneten statischen oder dynamischen Techniken deutlich erhöhen.*

Um diese These zu belegen, identifiziert die Arbeit technische Probleme dieses suchbasierten Ansatzes, deren Lösung oder Abschwächung ein Potenzial zur Effizienzsteigerung bergen. So hängt der notwendige Aufwand für eine automatische Testdatensuche beispielsweise maßgeblich von der Größe des Raums aller theoretisch möglichen Testdaten, genannt Suchraum, ab. Ließe sich dieser einschränken, so könnte dies dem suchbasierten Test zu einer Effizienzsteigerung verhelfen. Auf derlei Aspekte wird die Arbeit im Detail eingehen. Die ermittelten Probleme adressiert die Arbeit auf zwei Wegen:

1. Statische Analysetechniken wurden geschaffen, um die Testdatengenerierung besser auf die sich ihr stellenden Probleme einzustellen und auszurichten. Da diese Techniken vor der eigentlichen Testdatengenerierung Anwendung finden, werden sie als *statische Voranalysen* bezeichnet. Es wurden drei derartige Techniken entwickelt, namentlich die *Überdeckungsziel-Erreichbarkeitsprüfung* (SIA), die *Modelleingang-Abhängigkeitsermittlung* (SDA) und die *Suchzielpriorisierung* (CGSeq).
2. Im Weiteren widmet sich die Arbeit *alternativen Techniken zur Testdatengenerierung*. In der zugrunde liegenden Vorarbeit von Windisch

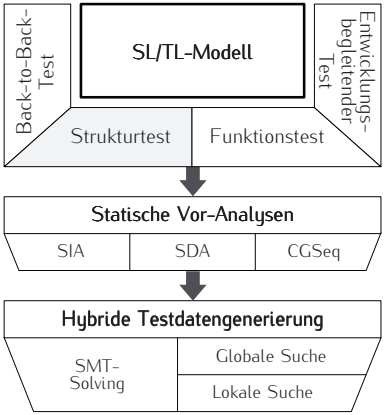


Abbildung 1: Übersichtsbild des entwickelten hybriden Testdatengenerierungsverfahrens für Simulink/TargetLink-Modelle

wurde zur automatisierten Testdatengenerierung ein globales Suchverfahren verwendet. Daneben existiert die Klasse lokaler Suchverfahren. Auf die Bedeutung dieser Unterscheidung wird an späterer Stelle eingegangen. Mit der Absicht, effizienter Testdaten zu generieren, wurde in dieser Arbeit ein auf die Bedürfnisse des Tests von SL/TL-Modellen zugeschnittenes *lokales Suchverfahren* entwickelt. Ein weiterer Beitrag besteht in der Testdatengenerierung durch eine Art symbolische Ausführung unter Anwendung von *SMT-Solving*, in hybridisierter Form mit einem Suchverfahren.

Die entwickelten Techniken wurden in einem prototypisches Werkzeug umgesetzt. Dieses demonstriert einerseits die Praxistauglichkeit des entstandenen Ansatzes und dient andererseits der Evaluierung der Beiträge dieser Arbeit. Zwei reale Modelle der Firma *Daimler* dienen hierbei als Fallbeispiele. Die Ergebnisse der Fallstudien zeigen, dass die hergestellte Hybridisierung einer suchbasierten Testdatengenerierung zu einer deutlichen Effizienzsteigerung führt.

Die einleitenden Ausführungen dieses Abschnitts bezüglich des Kontextes und der Beiträge dieser Arbeit sind in Abbildung 1 veranschaulicht. Dargestellt ist zum einen der Typ betrachteter Testobjekte, TL-konforme SL-Modelle, sowie die Wahl der Testart, zu deren Zweck eine automatisierte Erzeugung von Testdaten erfolgt. Von diesen Testarten rückt der Strukturtest in den Fokus dieser Arbeit. Zum anderen sind in der Abbildung die einzelnen Techniken des hybriden Testdatengenerierungsverfahrens aufgeführt. Zudem ist eine grobe Abfolge zu erkennen.

AUFBAU DER ARBEIT

Die Dissertation hat die im Folgenden geschilderte Struktur.

Kapitel 2 behandelt das Hintergrundwissen und die Motivation, welche den vorgestellten Ideen und Lösungen zu Grunde liegen. Einerseits wird der klassische, in der Automobilindustrie angewandte modellbasierte Entwicklungsprozess zur Erstellung von Software eingebetteter Systeme eingeführt. In diesem Zusammenhang erfolgt auch eine Einführung in SL und TL, um die in dieser Arbeit betrachteten Modelle zu definieren. Andererseits beschäftigt sich das Kapitel mit den Grundlagen des Softwaretests, dem Test von Modellen im Speziellen und den möglichen Anwendungsszenarien einer automatisierten Testdatengenerierung beim Modelltest. Anschließend werden existierende Automatisierungsansätze zur Testdatengenerierung beleuchtet, wobei der suchbasierte Testansatz eine intensivere Betrachtung erfährt. Das Kapitel endet mit einer Analyse der Problemstellungen, welche bei der Anwendung des suchbasierten Tests für SL/TL-Modelle existieren.

In *Kapitel 3* werden die drei entwickelten Verfahren zur statischen Voranalyse vorgestellt. Für die Techniken SIA, SDA und CGSeq werden jeweils ihre Zielsetzung, ihre Funktionsweise sowie verwandte Arbeiten und Ansätze betrachtet. Die Integration dieser Techniken miteinander und die ihre Anbindung an die Testdatensuche, im Sinne eines hybriden Testdatengenerierungsverfahrens, erfolgt im Anschluss.

Kapitel 4 befasst sich mit zwei alternativen Testdatengenerierungstechniken. Ein lokales Suchverfahren und ein Vorgehen zur Verwendung von SMT-Solving zu Zwecken der Testdatengenerierung für SL/TL-Modelle werden vorgestellt. Möglichkeiten zur Hybridisierung der Generierungstechniken sind ebenfalls Thema dieses Abschnitts.

Kapitel 5 stellt den im Rahmen dieser Arbeit entstandenen Testwerkzeug-Prototypen vor.

Die mit diesem Werkzeug durchgeführten Fallstudien zur Evaluierung der entwickelten Ansätze und Techniken finden sich in *Kapitel 6* wieder. Eine Bewertung des Entstandenen erfolgt ebenfalls an dieser Stelle.

Kapitel 7 fasst die in dieser Arbeit gemachten Ergebnisse und Erfahrungen zusammen und gibt einen Überblick über offene Themen und Fragestellungen im behandelten Gebiet, sowohl aus wissenschaftlicher als auch aus technischer Sicht.

Verwendete Kurzschreibweisen können übrigens im *Abkürzungsverzeichnis* nachgeschlagen werden. Dieses folgt, gemeinsam mit *Abbildungs-* und *Tabellenverzeichnis*, dem *Literaturverzeichnis* am Ende dieser Arbeit.

2 HINTERGRUND

Das Fundament, auf welches die Beiträge dieser Arbeit aufbauen, ist Gegenstand dieses Kapitels. Das Themenfeld, in welchem die Arbeit sich bewegt, als auch die Problemstellung, derer sich die Arbeit angenommen hat, werden hier eingehend behandelt. Das Grundlagenwissen und die Anforderungen an das Arbeitsthema entstammen dabei sowohl wissenschaftlichen Quellen und der Fachliteratur als auch Erfahrungen und Kenntnissen aus der Automobilindustrie.

2.1 MODELLTEST

Ein modellbasiertes Vorgehen bei der Softwareentwicklung führt zwangsläufig zu der Frage nach der Korrektheit der entwickelten Modelle. Diese Anforderung kommt umso stärker zum Tragen, wenn Modelle nicht nur der Unterstützung einer manuellen Implementierung dienen, wie es beispielsweise in der klassischen Softwareentwicklung häufig der Fall ist, sondern der Programmcode gänzlich automatisiert aus Modellen generiert wird. In der Automobilindustrie wird dies bei der modellbasierten Entwicklung mittels SL/TL so gehandhabt. Der Test ist in diesem Bereich neben Techniken wie Reviews oder statischen Analysen eine mögliche Maßnahme zur qualitativen Absicherung eines Modells. In diesem Abschnitt werden daher neben der modellbasierten Softwareentwicklung im Automobilbereich und der betrachteten Art von Modellen auch das Wichtigste aus Theorie und Praxis des Softwaretests einführend vorgestellt.

Der Modelltest sollte im Übrigen nicht mit dem *modellbasierten Test* verwechselt oder gleichgesetzt werden. Während es beim Modelltest um die Überprüfung eines Modells an sich geht, geht es beim modellbasierten Test um die Überprüfung eines Testobjekts mithilfe eines Modells. Ein solches Modell bildet üblicherweise Anforderungen (Soll-Verhalten) an das Testobjekt ab und dient der Ableitung von Testfällen, welche die Einhaltung der Anforderungen durch das Testobjekt untersuchen. Das Testobjekt kann dabei von beliebiger Natur sein, das heißt es könnte sich zum Beispiel um ein Steuergerät oder eben auch um ein SL/TL-Modell

handeln [22]. Der Begriff Modelltest hingegen setzt voraus, dass das Testobjekt ein Modell ist – stellt aber keinerlei Bedingungen an die Herkunft oder Herleitung der Testfälle/Testdaten.

2.1.1 MODELLBASIERTE ENTWICKLUNG

Modellbasiert Software zu entwickeln, bedeutet, die Implementierung einer Software durch grafische Notationen geeignet zu unterstützen. Klassischerweise finden Modellierungsaktivitäten dabei zeitlich vor anderen Aktivitäten zur Realisierung, wie der Programmierung, statt.

Das Bestreben einer modellbasierten Entwicklung gibt es im Bereich der eingebetteten Systeme wie auch bei klassischen Softwaresystemen. Als Hauptgründe für ein modellbasiertes Vorgehen werden typischerweise eine Senkung der Komplexität innerhalb der Entwicklung, eine Steigerung der Qualität, sowie eine positive Auswirkung auf Entwicklungszeiten und -kosten genannt [17]. Kirstan belegt diese Aussage in einer Studie [75] sogar durch Zahlen: Ein modellbasiertes Vorgehen bei der Entwicklung von eingebetteter Software im Automobilbereich führt zu Kosten- und Zeiteinsparungen von im Schnitt 27 bzw. 36 Prozent. Folgt man den weiteren Ausführungen dieser Studie, so liefert die Validierung auf Modellebene darüber hinaus im Schnitt bis zu 60 Prozent mehr gefundene Fehler im Software-Design im Vergleich mit der Validierung auf nachfolgenden Ebenen, wenn nicht modellbasiert implementiert wird.

In der klassischen Softwareentwicklung, insbesondere im Bereich der objektorientierten Programmiersprachen, zählen *UML*-Modelle [103] zu den prominentesten Notationen im Rahmen einer modellbasierten Entwicklung. Allerdings hat die Verwendung derartiger Modelle in der klassischen Softwareentwicklung meist nur einen unterstützenden Charakter. Dies bedeutet, dass die Modelle dem Entwickler zwar dabei helfen, seine Software sinnvoll zu strukturieren und deren Funktionsweise sauber zu spezifizieren – der Programmcode aber meist unter Zuhilfenahme der entstandenen Modelle manuell geschrieben wird. Im Bereich eingebetteter Software, insbesondere im Automobilbereich, erfolgt die Erstellung des Programmcodes (häufig in der Sprache C) heutzutage hingegen größtenteils automatisiert. Daher ist bei den der Codegenerierung zugrunde liegenden Modellen auch häufig von Implementierungsmodellen die Rede.

Der hohe Automatisierungsgrad bei der Entwicklung eingebetteter Software lässt sich vor allem durch die folgenden Feststellungen begründen. Die verwendeten Modellierungsnotationen liegen dem daraus abgeleiteten Code zum einen sehr nah. So nah, dass mancher die Modellierung auch als grafische Programmierung bezeichnet. Zum anderen sind die

Modellierungsnotationen meist sehr speziell. Das bedeutet, sie orientieren sich stark an den jeweiligen Besonderheiten eines eingebetteten Systems. Ein Beispiel: Bei eingebetteten Systemen, welche Steuerungs- und Regelungsaufgaben in einem Automobil übernehmen, handelt es sich meist um dynamische Systeme. Derartige Systeme berücksichtigen den Faktor Zeit. Im Fall einer Blinkersteuerung gilt es zum Beispiel, die Richtungsblinker eines Fahrzeugs in einem zeitlichen Takt synchron zueinander aufleuchten zu lassen. Die zu verarbeitenden und berechneten Werte und Variablen besitzen also eine zeitliche Dimension. Der daher notwendigen Signalverarbeitung bei der Softwareentwicklung wird in einer Modellierungsnotation wie SL Rechnung getragen.

Eine Modellbildung mitsamt Codegenerierung hat weitere Vorteile. Generierter Code hat eine einheitliche Form und moderne Codegeneratoren garantieren die Einhaltung von Code-Richtlinien, welche in der betreffenden Domäne gängig oder sogar verpflichtend sind. Ein Modell kann zudem als Formalisierung der Spezifikation gesehen werden und bildet somit eine Brücke zwischen textueller Spezifikation und Code [106]. Hinzu kommt, dass ein Modell verständlicher und einfacher lesbar als Code ist. Dies gilt nicht nur für den Entwickler selbst, sondern auch für den Austausch mit anderen Entwicklern und mit weiteren an der Entwicklung beteiligten Personen. Eine erhöhte Lesbarkeit erleichtert auch die Wiederverwendbarkeit und Wartbarkeit entwickelter Software. Die Modelle sind zudem ausführbar und ermöglichen somit Simulationen und Tests.

Ein modellbasiertes Vorgehen ist aus der automobilen Softwareentwicklung mittlerweile nicht mehr wegzudenken. Da die entwickelten Systeme allerdings, wie erwähnt, hohen Ansprüchen zu genügen haben, und die Entwicklung in der Regel unter Zulieferer-Beteiligung erfolgt, sind mit der Zeit eine Reihe von Firmen- oder sogar Industrie-übergreifenden Standards und Richtlinien entstanden. So haben beispielsweise die Norm *ISO 26262* [39] und die *AUTOSAR*-Initiative [71] wesentlich zur Gestaltung der heutigen Entwicklungsprozesse und der Nutzung einheitlicher Software-Architekturen beigetragen. Die modellbasierte Entwicklung, insbesondere die Wahl der Modellierungsnotationen und der Werkzeuge, erfolgt daher in der Automobilindustrie herstellerübergreifend sehr ähnlich.

Als Beispiele bekannter Modellierungsnotationen oder -werkzeuge für eingebettete Systeme und deren Software sind neben SL auch *LabView* [69], *SCADE* [111] und *Modelica* [7] zu nennen. In der Automobilindustrie ist SL allerdings die am häufigsten eingesetzte Notation, beziehungsweise ML/SL das am häufigsten eingesetzte Werkzeug. Wird SL verwendet, so kommt häufig der Codegenerator TL zum Einsatz.

2.1.2 SIMULINK UND TARGETLINK

In den vorherigen Abschnitten wurde bereits erklärt, dass SL eine Erweiterung des Werkzeugs ML ist und zur Modellierung von Regelungs- und Steuerungssoftware eingebetteter Systeme eingesetzt werden kann. Ebenso wurde der Codegenerator TL erwähnt. In diesem Abschnitt werden SL und TL nun im Detail vorgestellt. Teilweise werden die hinter SL stehenden Konzepte sehr detailliert eingeführt – dies ist allerdings für eine präzise Vorstellung der Beiträge dieser Arbeit notwendig.

SIMULINK

Die Grundzüge der SL-Notation lassen sich sehr kurz beschreiben. Bei SL-Modellen handelt es sich um Datenflussdiagramme. Dies sind gerichtete Graphen, bei denen die Knoten Funktionen und die Kanten jeweils Quelle und Ziel eines Datenflusses darstellen. Die Knoten werden in SL als *Blöcke* bezeichnet und die Kanten als *Signale*. Die Daten, welche von Block zu Block gereicht werden, sind Zahlenwerte. Beispielsweise enthält die Block-Bibliothek von SL einen Block mit der Summenfunktion, welcher die Werte der zu dem Block führenden Signale addiert und das Ergebnis mit einem ausgehenden Signal ausgibt. SL ergänzt diese Grundzüge um eine Reihe von Eigenschaften und Techniken. Die Wichtigsten sind im Folgenden aufgeführt.

Ports. Blöcke besitzen *Ports*. *Inports* (Eingangsports) und *Outports* (Ausgangsports) bilden dabei die *externen Schnittstellen* eines Blocks, beziehungsweise die Ein- und Ausgaben, über die der Block seine Funktionalität beschreibt. Einem Port gehört stets genau ein Signal an – vorausgesetzt, das Modell ist vollständig modelliert und somit auch ausführbar. Die In- und Outports eines Blocks sind jeweils durchnummeriert und haben bezüglich ihrer Darstellung eine entsprechend feste Position am Block.

Hierarchie. Um auch große Modelle mit einer Vielzahl an Blöcken anschaulich darstellen zu können, ermöglicht SL die Gruppierung von Blöcken zu Teilsystemen, sogenannte *Subsysteme*. Subsysteme verlagern die gruppierten Blöcke und deren Signalverbindungen auf eine tiefere Hierarchie-Ebene. Auf der darüber liegenden Hierarchie-Ebene stellt sich das Subsystem selbst wiederum als ein Block dar.

Schnittstellen. Auch ein SL-Modell als Ganzes ist im Grunde nichts anderes als ein Subsystem. Modell und Subsysteme enthalten beide Blöcke und besitzen im Normalfall Ein- und Ausgänge, welche die *internen Schnittstellen* dieser funktionalen Einheit bilden. Für diesen Zweck existieren spezielle *Inport-* und *Outport-Blöcke*. Innerhalb eines Subsystems sind die Inport- und Outport-Blöcke durchnummeriert – entsprechend der

Position des dazugehörigen In- oder Outports in der externen Schnittstelle des Subsystem-Blocks. Der Datenfluss ist in diesem Zusammenhang also nicht explizit im Modell dargestellt, sondern ergibt sich aus der Zuordnung zwischen interner und externer Block-Schnittstelle.

Zeit. SL-Modelle sind dynamische Systeme. Dies bedeutet, sie modellieren zeitabhängige Prozesse. Hierbei kann es sich sowohl um zeitkontinuierliche als auch um zeitdiskrete Prozesse handeln. In dieser Arbeit werden, wie auch in der Entwicklungspraxis bei Verwendung von TL, ausschließlich zeitdiskrete Modelle betrachtet. Die zeitliche Dynamik spiegelt sich in SL-Modellen in dem Umstand wieder, dass die Kanten Signale sind. Signale sind nichts Anderes als Sequenzen von Zahlenwerten, wobei jeder Wert einem Zeitpunkt zugeordnet ist. Ein Modell besitzt eine global definierte Abtastrate und jeder Block, falls angegeben, eine individuelle Abtastrate. Die Abtastrate bestimmt bei der Ausführung eines Modells, in welchen zeitlichen Zyklen ein jeder Block ausgeführt wird. Der daraus resultierende Ausführungszeitpunkt wird im Folgenden auch als Zeitschritt bezeichnet. Damit jeder Block bei seiner Ausführung auch auf die von ihm benötigten Werte seiner Eingangssignale zurückgreifen kann, berechnet SL für die Blöcke eines jeden Subsystems vor der Ausführung des Modells eine – für die Laufzeit statische – Ausführungsreihenfolge.

Signaleigenschaften. Ein jeder Outport eines jeden Blocks definiert jeweils eine Reihe von Eigenschaften, welche für das von ihm ausgehende Signal gelten. So wird an dieser Stelle der Datentyp des Signals spezifiziert. Des Weiteren ermöglicht SL es, Signale zu bündeln. Hierbei bietet SL zwei unterschiedliche Konzepte an. Zum einen können Signale Vektoren sein und somit mehrere Werte zu einem Zeitschritt bereitstellen. Dabei bezeichnet die Dimension eines Vektorsignals die Anzahl der Werte, die es enthält. Zum anderen kann ein Signal einen Container für beliebig viele andere Signale bilden. In diesem Fall ist von sogenannten Bus-Systemen, Bus-Signalen oder einfach nur Bussen die Rede. Ein Bus-Signal definiert eine Baumstruktur, in der andere Signale angeordnet sind. Die enthaltenen Signale können selbst wiederum Bus-Signale oder Vektorsignale sein.

Atomarität und bedingte Ausführung. Ein Subsystem kann als atomare Einheit deklariert werden. Dies wirkt sich auf die von SL berechnete Ausführungsreihenfolge aus. Und zwar kann die Ausführung der Blöcke eines atomaren Subsystems nicht durch die Ausführung eines Blockes, der außerhalb dieses Subsystems liegt, unterbrochen werden. SL stellt darüber hinaus eine Reihe von atomaren Subsystem-Typen zur Verfügung, deren enthaltene Blöcke nur zu den Zeitschritten ausgeführt werden, bei denen eine bestimmte Bedingung gilt. So besitzt das *Enabled Subsystem* einen zusätzlichen Eingang mit einem sogenannten Kontrollsignal. Die Blöcke eines Enabled Subsystems werden nur dann ausgeführt, wenn

dieses Kontrollsignal einen positiven Wert enthält. Für die Ausgänge eines solchen Subsystems kann festgelegt werden, ob es im Falle der Inaktivität den letztmalig berechneten oder einen spezifizierten Wert ausgeben soll. Ein Subsystem, welches nicht atomar ist, wird auch als virtuelles Subsystem bezeichnet.

Stateflow (SF). Dies ist eine Statechart-ähnliche Notation für Zustandsautomaten und Flussdiagramme. Mithilfe von SF können insbesondere ereignisgesteuerte Funktionalitäten effizienter modelliert werden als mit SL-Mitteln. In SL können SF-Automaten innerhalb eines SF-Blocks modelliert werden. Die Ein- und Ausgänge eines SF-Blocks lassen sich beliebig gestalten, je nachdem welche Signale in dem Automaten benötigt werden oder berechnet werden sollen.

TARGETLINK

Der Codegenerator TL integriert sich direkt in die Entwicklungsumgebung von SL. TL stellt zum einen zusätzliche Anforderungen an ein SL-Modell. So schränkt TL die Menge der erlaubten Blöcke unter anderem dahingehend ein, dass es in dem Modell keine zeitkontinuierlichen, sondern ausschließlich zeitdiskrete Blöcke geben darf. Das Modell muss so gestaltet sein, dass die Modellsimulation mit einem Solver erfolgen kann, der eine feste (statische) Abtastrate voraussetzt. Diesbezüglich geht TL konform mit dem Industriestandard ISO 26262 [39][1], welcher eine variable Abtastrate bei der Fahrzeug- und Umgebungsmodellierung als relevant erachtet, für die eingebettete Software aus Gründen einer effizienten Codegenerierung allerdings eine feste Abtastrate empfiehlt.

Darüber hinaus ergänzt TL die Block-Bibliothek von SL. Die Block-Bibliothek von TL umfasst sowohl Blocktypen, welche äquivalente, in SL vorhandene Blocktypen ersetzen, als auch Blocktypen, die in SL standardmäßig nicht enthalten sind. Der Grund, wieso TL eigene Blöcke anbietet, ist zweifältig. Einerseits haben TL-eigene Blöcke zusätzliche Parameter, welche für die Codegenerierung von Bedeutung sind. Andererseits existiert das Problem, dass es für die Blocktypen in SL keine frei verfügbare formale Semantik gibt. Um aber eine einwandfreie Codegenerierung zu garantierten, bringt TL eine eigene Block-Palette mit. Eine formale Semantik ist allerdings auch für den TL-Sprachumfang nicht veröffentlicht.

DEFINITION EINES SL/TL-MODELLS

Die Semantik der Blöcke ist auch für die in dieser Arbeit entwickelten Techniken von Relevanz. Verschiedene Arbeiten haben sich in der Vergangenheit mit der Definition einer Semantik für SL beschäftigt – allerdings

1 ISO/DIS 26262-6, Kapitel 6, Anhang B, Model Based Development

vornehmlich mit der SF-Notation. So stellte Hamon zum einen eine denotationelle Semantik für SF vor [53] und zum anderen, gemeinsam mit Rushby, eine operationale Semantik [55], ebenfalls beschränkt auf SF. Eine eigene Definition der Blocksemantik, zumindest für SL, ist also vonnöten. Diese findet nicht nur bei der Vorstellung der Beiträge dieser Arbeit, sondern auch in den weiteren Ausführungen dieses Kapitels Verwendung.

Neben der Semantik ist auch ein exaktes Verständnis der Syntax eines SL/TL-Modells wichtig. In diesem Zusammenhang haben mehrere Arbeiten Meta-Modelle für SL entwickelt. So benötigte Scheible [107] ein solches Meta-Modell zur Messung von Modellqualitätsmetriken. Ebenso wurde ein Meta-Modell im Rahmen des MATE-Ansatzes zur automatisierten Prüfung von Modellqualitätsrichtlinien und der Behebung von entsprechenden Verletzungen definiert [80]. Mit dem Ziel, SL-Modelle zu refaktorisieren und zu transformieren, stellten Tran et al. [119] ein Meta-Modell für SL auf. Die Meta-Modelle dieser Arbeiten sind jedoch entweder auf ihren eigenen Anwendungszweck zugeschnitten oder unzureichend für den Anwendungszweck dieser Arbeit. So benötigen die in dieser Arbeit vorgestellten Techniken beispielsweise keine Layout-Informationen. Inhalte aus dem ML-*Workspace* hingegen sind beispielsweise von Interesse.

In dieser Arbeit wird daher eine eigene Definition eines SL/TL-Modells aufgestellt. Die Definition ist dabei aus den Schilderungen der Dokumentationen von SL (Version R2009b) und TL (Version 3.1), sowie der Beobachtung des Simulationsverhaltens der Blöcke, abgeleitet. Sie hat keinen Anspruch auf Vollständigkeit, sondern bildet nur die Bestandteile von SL und TL ab, die in dieser Arbeit relevant sind. Im Folgenden wird bei Bedarf auf verwendete Hilfsfunktionen und -mengen verwiesen, welche in Tabelle 1 aufgeführt sind. Eine Erläuterung dieser erfolgt jeweils entweder an passender Stelle auf den folgenden Seiten oder bei späterer Verwendung.

Modell. Ein SL/TL-Modell sei durch folgendes Tupel aus der Menge aller möglichen Modelle M gegeben:

$$M = \{\, (B, P, S, W, DD, C) \mid B \subseteq ((B_{SLS} \subset B_{SL}) \cup B_{TL}) \,\} \tag{1}$$

Hierbei ist B die Menge aller im Modell enthaltenen Blöcke. B_{SLS} ist hierbei die Untermenge der von TL unterstützten SL-Blöcke B_{SL}, ergänzt um die TL-eigenen Blöcke B_{TL}. P ist die Menge aller Ports, S die Menge aller Signale. Die im ML-Workspace abgelegten Variablen und deren Belegung, auf welche ein Modell bei seiner Ausführung zurückgreifen kann, sind in der Menge W enthalten. Das zu TL gehörende *Data Dictionary*, in dem ebenfalls Variablenbelegungen sowie weitere Informationen abgelegt sein können, sei an dieser Stelle abstrakt als Menge DD bezeichnet.

(a) Eingangs-/Ausgangssignale eines Blocks:

$IS_b = \{s \mid s = (sp, DP) \in S \wedge \exists p \in IP_b : p \in DP\}$

$OS_b = \{s \mid s = (sp, DP) \in S \wedge \exists p \in OP_b : p = sp\}$

(b) Signal eines Ports:

$sig : P \to S$

$sig(p) = ps_p$

$sig_{in}, sig_{out} : B \times \mathbb{N}^+ \to S$

$sig_{in}(b, pos) = sig(ip(b, pos))$

$sig_{out}(b, pos) = sig(op(b, pos))$

(c) In-/Outport eines Blocks über Portposition:

$ip, op : B \times \mathbb{N}^+ \to P$

$ip(b, pos) = p' \quad$ mit $p' = (pos, ps, d) \in IP_b$

$op(b, pos) = p'' \quad$ mit $p'' = (pos, ps, d) \in OP_b$

(d) Quellport und Zielports eines Signals:

$src : S \to P$

$src(s) = sp_s$

$dest : S \to \mathcal{P}(P)$

$dest(s) = DP_s$

(e) Signal-/Inport-Dimension:

$d : S \to \mathbb{N}^+$

$d(s) = d_{src(s)}$

$d : P \to \mathbb{N}^+$

$d(p) = d_p$

$d : B \times \mathbb{N}^+ \to \mathbb{N}^+$

$d(b, pos) = d_{ip(b, pos)}$

(f) Block eines Ports:

$b : P \to B$

$b(p) = b'$

mit $b' \in B \wedge (p \in IP_{b'} \vee p \in OP_{b'})$

(g) Wert an Block-Inport für Zeitschritt:

$in^v_{b,pos} : \mathbb{N} \to \mathbb{R}$

$in^v_{b,pos}(ts) = out^v_{b(p'),pos(p')}(ts)$

mit $p' = src(sig(ip(b, pos)))$

(h) Maximale Dimension aller Blockeingänge:

$maxd_{in} : B \to \mathbb{N}^+$

$maxd_{in}(b) = x' \quad$ mit $(\exists p_1 \in IP_b : x' = d_{p_1})$

$\wedge (\forall p_2 \in IP_b : d_{p_2} \leqslant x')$

(i) Portposition:

$pos : P \to \mathbb{N}^+$

$pos(p) = pos_p$

(j) Übergeordnete Subsysteme eines Blocks (direktes/alle) und Hierarchietiefe:

$psys_{dir} : B \to B$

$psys_{dir}(b) = x' \quad$ mit $b \in BC_{x'}$

$psys_{all} : B \to \mathcal{P}(B)$

$$psys_{all}(b) = \begin{cases} \{x\} \cup psys_{all}(x) & , \exists x \in B \wedge b \in BC_x \\ \emptyset & , \text{sonst} \end{cases}$$

$dep : B \to \mathbb{N}$

$$dep(b) = \begin{cases} 1 + dep(x) & , \exists x \in B \wedge b \in BC_x \\ 0 & , \text{sonst} \end{cases}$$

(k) Vektorindex-Korrektur für Port oder Signal:

$vc : \mathbb{N}^+ \times P \to \mathbb{N}^+$

$vc(v, p) = \min(v, d_p)$

$vc : \mathbb{N}^+ \times S \to \mathbb{N}^+$

$vc(v, s) = \min(v, src(s))$

(l) Wahrheitswert eines Signalwerts:

$T, F : \mathbb{R} \to \mathbb{B}$

$$T(x) = \begin{cases} 1 & , \text{wenn } x \neq 0 \\ 0 & , \text{sonst} \end{cases}$$

$F(x) = \neg T(x)$

Tabelle 1: Hilfsfunktionen und -definitionen für Simulink/TargetLink-Modelle, welche in diesem Dokument zu weiteren Beschreibungen und Definitionen Anwendung finden

Die Menge C bildet die Ausführungskonfiguration des Modells – diese beinhaltet unter anderem die globale Abtastrate.

Block. Ein Block sei durch folgendes Tupel aus der Menge aller Blöcke B eines Modells gegeben:

$$B = \{\ (bt, IP, OP, BC, PV) \mid bt \in BT \wedge IP \subseteq P \wedge OP \subseteq P \wedge BC \subset B\ \} \tag{2}$$

Der Parameter bt bezeichnet den Blocktypen. Dieser wird nachfolgend definiert. IP ist die Menge der Eingangsports, OP die Menge der Ausgangsports. Handelt es sich bei einem Block um ein Subsystem, so bilden die enthaltenen Blöcke die Menge BC. Fast alle Blocktypen besitzen Parameter, deren Belegung für eine Blockinstanz in der Menge PV enthalten ist.

Blocktyp. Jeder Block im Modell ist eine Instanz eines Blocktypen. Ein Blocktyp sei als Element aller Blocktypen BT durch folgendes Tupel definiert:

$$BT = \{\ (PM, out^{\nu}_{b,pos}, st^{\nu}_{b}, act_{b}, rs_{b})\ \} \tag{3}$$

PM ist die Menge der Blocktyp-spezifischen Parameter (z.B. eine individuelle Abtastrate) mit ihren Bezeichnern und gegebenenfalls dazugehörigen Belegungsmöglichkeiten. Die Abbildung

$$out^{\nu}_{b,pos} : \mathbb{N} \to \mathbb{R} \tag{4}$$

beschreibt die Blockfunktionalität über den berechneten Ausgabewert zu einem bestimmten Zeitschritt und dient als Vorlage für eine Blockinstanz b. Der Parameter pos bezeichnet die Position eines der Ausgangsports und ν einen der Vektorindizes des von diesem Port abgehenden Signals.

Manche Blöcke verwalten darüber hinaus einen Zustand

$$st^{\nu}_{b} : \mathbb{N} \to \mathbb{R} \tag{5}$$

Dieser gilt ebenfalls für einen bestimmten Zeitschritt und kann die Form eines Vektors (Index-Angabe ν) haben.

Die Entscheidung über die Aktivität einer Blockinstanz zu einem Zeitschritt ist durch

$$act_{b} : \mathbb{N} \to \mathbb{B} \tag{6}$$

gegeben. Falls ein Blocktyp diese Abbildung nicht spezifisch definiert, gilt allgemein, dass ein Block aktiv ist, wenn kein übergeordnetes Subsystem inaktiv ist:

$$act_{b}(ts) = \neg \exists x \in psys_{all}(b) \colon \neg\, act_{x}(ts) \tag{7}$$

Die Funktion $psys_{all}$ (siehe Tabelle 1, Abschnitt j) bestimmt hierbei alle Subsystem-Blöcke, die einem Block in der Modellhierarchie übergeordnet sind.

Im Zuge der (Re-)Aktivierung eines Blocks besteht, je nach Parameter-Konfiguration, die Möglichkeit des Zurücksetzens des internen Zustands. Ob ein Zurücksetzen zu einem Zeitschritt erfolgt, gibt die Funktion

$$rs_b : \mathbb{N} \to \mathbb{B} \tag{8}$$

an. Falls für einen Blocktyp nicht anders definiert, gilt, dass eine Reaktivierung nur dann erfolgt, wenn das in der Subsystem-Hierarchie nach oben hin nächstfolgende bedingt ausgeführte Subsystem ein Zurücksetzen verlangt:

$$rs_b(ts) = \begin{cases} 1\,, \exists x \in psys_{all}(b): \\ \quad (rs_x(ts) \wedge \forall y \in psys_{all}(b):\ dep(x) \geqslant dep(y)) \\ 0\,, \text{sonst} \end{cases} \tag{9}$$

Die Funktion dep (siehe Tabelle 1, Abschnitt j) bezeichnet hierbei die Hierarchietiefe eines Blocks, das heißt die Anzahl aller Subsystem-Blöcke, die dem Block in der Modellhierarchie übergeordnet sind.

Port. Ein Port als Element aller Ports P sei definiert als Tupel seiner Position pos, dem angehörigen Signal ps, sowie der Dimension d und dem Datentyp dt des von diesem Port ausgehenden Signals:

$$P = \{\ (pos, ps, d, dt) \mid\ pos \in \mathbb{N}^+ \wedge\ ps \in S \wedge d \in \mathbb{N}^+\ \} \tag{10}$$

Eine Funktion, welche den zu einem Port gehörenden Block liefert, ist für eine spätere Verwendung in Tabelle 1 (Abschnitt f) angegeben. Die Position eines Ports ist in der Tabelle zudem auch über eine Funktion definiert (Abschnitt i).

Signal. Ein Signal aus der Menge aller Modellsignale S sei durch seinen Ursprung (Outport sp) und seine Ziele (Inport-Menge DP) definiert:

$$S = \{\ (sp, DP, BS) \mid sp \in P \wedge DP \subseteq P\ \} \tag{11}$$

Handelt es sich um ein Bus-Signal, so bildet die Menge BS die Bus-Struktur ab. Diesem Fall wird an dieser Stelle nicht weiter Rechnung getragen, da Bus-Signale im Rahmen des entwickelten Verfahrens zu Beginn aufgelöst werden und anschließend keine Rolle mehr spielen.

In Tabelle 1 (Abschnitt a) sind im Übrigen für eine spätere Verwendung in dieser Arbeit die Mengen IS_b und OS_b definiert. Sie enthalten jeweils alle ein-/ausgehenden Signale eines Blocks. Zudem sind in dieser Tabelle in Abschnitt b Funktionen definiert, mit welchen einerseits das Signal eines Ports und andererseits ein ein- oder ausgehendes Signal eines

Blocks durch Angabe der Portposition abgefragt werden können. Letztere nutzen die Funktionen ip und op (siehe Tabelle 1, Abschnitt c), welche den Ein-/Ausgangsport eines Blocks mit einer bestimmten Portposition liefern. Den Ursprungsport und die Zielports eines Signals geben die Funktionen src und dest an (siehe Tabelle 1, Abschnitt d). Die Dimension eines Signals kann durch die in der Tabelle in Abschnitt e definierten Funktionen abgefragt werden. Die Funktionen ermitteln die Dimension hierbei wahlweise über Angabe des Signals, des zugehörigen Ports oder des zu diesem Port gehörenden Blocks mitsamt Angabe der Portposition. Zudem existiert die Funktion maxd_{in}, welche die maximale Dimension aller in einen Block eingehenden Signale angibt (Abschnitt h).

Simulation. Die Simulation/Ausführung eines Modells wird in dieser Arbeit als eine Art *Black-Box* betrachtet. Das bedeutet, die Behandlung und Funktionsweise einer Simulation in SL ist irrelevant. Relevant sind nur die der Simulation zu Grunde liegenden Daten und die Ergebnisse der Simulation. Entsprechend fällt die Definition aus:

$$\text{SIM} = \{\ (\text{MI}, \text{CS}, \text{MO})\ \} \tag{12}$$

MI ist die Menge der in das Modell eingehenden Signale. Für jeden Modelleingang i enthält MI eine Funktion $\text{mi}_i : \mathbb{N} \to \mathbb{R}$ über die Zeitschritte der Simulation. MO ist entsprechend die Menge der ausgehenden Signale eines Modells (bestehend aus $\text{mo}_j : \mathbb{N} \to \mathbb{R}$ je Modellausgang j). CS ist eine optionale Konfiguration der Simulation, welche die Konfiguration C_m eines Modells $m \in M$ überschreibt, je nach Vorhandensein der einzelnen möglichen Parameterangaben in CS.

Blocksemantik. Die Definition der Funktionalität einiger ausgewählter Blöcke, welche in SL/TL-Modellen häufig verwendet werden, ist in Tabelle 2 durch Angabe der spezifischen Ausgabefunktionen $\text{out}^{v}_{b,\text{pos}}$ aufgeführt. Genau genommen sind dort exakt die Blocktypen gelistet, welche in Beispielen oder weiteren Betrachtungen dieser Arbeit vorkommen. Eine vollständige Auflistung für alle SL/TL-konformen Blocktypen würde den Rahmen dieser Arbeit sprengen. Wenige spezielle, in dem betrachteten Praxisumfeld selten vorkommende Sonderfälle sind in den Definitionen zudem außen vor gelassen, um die Darstellung anschaulich zu halten oder den Aufwand der Umsetzung der entwickelten Techniken kontrollierbar zu halten. Zum Beispiel entfällt die Behandlung dynamischer Signaldimensionen, das heißt sich während der Laufzeit ändernde Vektorbreiten. Bei den Definitionen der Blockfunktionalitäten in der Tabelle ist des Weiteren zu beachten: Ein Signal wird standardmäßig als Vektor-Signal verstanden. Ein Nicht-Vektor-Signal ist schlichtweg ein eindimensionales Vektor-Signal. Im Folgenden werden die Darstellungen der Tabelle je enthaltenem Blocktyp kurz in Worte gefasst.

Inport. Ein Inport-Block besitzt als einzigen Blockparameter eine Portnummer. Die Ausgabe eines Inport-Blocks ist abhängig davon, ob sich der

Block in einem Subsystem oder auf der höchsten Hierarchie-Ebene des Modells befindet. Ist er in einem Subsystem, so wird der Eingangswert des dem Inport-Block zugeordneten Subsystem-Inports ausgegeben. Handelt es sich um einen Modell-Eingang, so gibt der Block den durch die Simulation festgelegten Eingangswert aus. Die in der Tabelle verwendete Abbildung $\mathrm{in}^{\nu}_{\mathfrak{b},\mathrm{pos}}$ gibt den Wert des Eingangssignals des Blocks $\mathfrak{b}$ mit der Portposition pos und dem Vektorindex ν zu einem Zeitschritt wieder. Die Definition hierzu enthält Tabelle 1 (Abschnitt g).

Outport und Subsystem. Ein Outport-Block hat selbst keinen Ausgang. Dieser befindet sich, sofern es sich nicht um einen Modellausgang handelt, an dem Subsystem, in dem sich der Outport-Block befindet. Handelt es sich um ein virtuelles Subsystem, so gibt dieses den Wert, der in den Outport-Block eingeht, direkt aus. Ein Outport-Block hat allerdings neben einer Portnummer zwei weitere Parameter: einen Initialwert sowie die Einstellmöglichkeit des Ausgabeverhaltens des übergeordneten Subsystems bei Inaktivität. Diese zwei Parameter sind für die Funktionalität eines bedingt ausgeführten Subsystems von Interesse.

Constant. Der Block besitzt als Parameter eine Konstante. Diese kann auch ein Vektor sein. Der Block gibt diese Konstante unabhängig vom betrachteten Zeitschritt aus. Die Möglichkeit, dass der Konstantenparameter stattdessen eine Referenz auf eine im Workspace befindliche Variable enthält, ist der Übersichtlichkeit halber in dieser Definition nicht enthalten. Dies gilt ebenfalls für alle anderen in der Tabelle dargestellten Blocktypen und ihre Parameter.

Logical Operator. Dieser Block realisiert eine aussagenlogische Verknüpfung seiner Eingangswerte. Der Block-Parameter Operator gibt den Typ der Verknüpfung an: *AND* ($\wedge$), *OR* ($\vee$), *NAND* ($\uparrow$), *NOR* ($\downarrow$), *XOR* ($\oplus$) oder *NOT* ($\neg$). Ein weiterer Parameter legt die Anzahl der Eingänge fest. Handelt es sich nun um den Operator $\neg$, so gibt der Block den Wert 1 aus, wenn der Wert am einzigen Blockeingang mit dem gleichen Vektorindex gleich 0 ist – andernfalls gibt er den Wert 0 aus. Hat der Block einen der anderen Operatoren, so gibt es zwei Fälle. Hat der Block nur einen Eingang, so definiert sich die aussagenlogische Operation über alle Vektorelemente dieses Eingangs und gibt entsprechend der Funktionalität des Operators den Wert 0 oder 1 aus. Hat der Block mehrere Eingänge, so bezieht sich der Operator auf die Werte aller Eingänge mit dem gleichen Vektorindex. Die Dimension des Ausgangssignals entspricht dabei der größten Dimension einer der Eingangssignale. Bei unterschiedlichen Dimensionen wird ein Eingangsvektor gegebenenfalls durch sein letztes Vektorelement erweitert. In der Definition der Funktionsweise dieses Blocks in Tabelle 2 geben die Funktionen T und F (siehe Tabelle 1, Abschnitt l) die Wahrheitswerte *wahr* und *falsch* eines reellen (Signal-)Werts an. Die Funktion vc (siehe Tabelle 1, Abschnitt k) korrigiert des Weiteren die Angabe eines Vektorindex für ein Signal, wenn diese Angabe die

Dimension des Signals überschreitet. Die Funktion vc existiert in zwei Varianten: für ein Signal und für den Ursprungsport eines Signals.

Relational Operator. Der Block besitzt zwei Eingänge. Je nach verwendetem Vergleichsoperator, der als Parameter festgelegt ist, vergleicht der Block zwei Eingangswerte und gibt je nach Zutreffen der Operatorbedingung eine 0 oder 1 aus. Die Operation erfolgt je Vektorindex, wobei Eingangsvektoren bei ungleicher Dimensionalität erweitert werden.

Switch. Ein solcher Block hat zwei Daten-Eingänge (oben und unten) und einen Kontroll-Eingang (in der Mitte). Je nach Wert am Kontroll-Eingang leitet der Block einen der beiden Daten-Eingänge zum Ausgang durch. Die zwei Parameter Kriterium und Schwellwert bestimmen dabei die Bedingung, die für das Durchleiten des oberen Daten-Eingangs gelten muss. In der Definition wird zwischen dem Fall, dass Kontroll-Eingang oder Schwellwert mehrdimensional sind, und dem Fall eines eindimensionalen Kontroll-Eingangs und Schwellwerts unterschieden. In letzterem Fall wird auch berücksichtigt, dass die Daten-Eingänge möglicherweise unterschiedliche Dimensionen haben.

Unit Delay. Dieser Block gibt seinen Eingangswert um einen Zeitschritt verzögert aus. Hierzu greift er auf einen internen Zustand zurück. Existiert kein vorheriger Zeitschritt, so gibt der Block einen Initialwert aus. Befindet sich der Block in einem bedingt ausgeführten Subsystem und kommt es zu einem Zurücksetzen der Zustände innerhalb dieses Subsystems, so setzt der Block seinen Zustand auf den Initialwert zurück.

Product, Sum und Min/Max. Die Blöcke Product und Sum können zur Multiplikation oder Division, beziehungsweise zur Addition oder Subtraktion ihrer Eingangswerte verwendet werden. Hierbei kann über einen Parameter jedem Eingang ein individueller Operator zugewiesen werden. Die Operation erfolgt im Falle von nur einem Eingang über die Vektorelemente desselben oder, bei mehreren Eingängen, über alle Eingangswerte mit gleichem Vektorindex. Dies gilt auch für den Min/Max-Block, der je nach Parameterwahl den kleinsten oder größten Eingangswert ausgibt.

Abs und Gain. Ein Abs-Block berechnet den Absolutwert seines Eingangswerts und ein Gain-Block multipliziert seinen Eingangswert mit einem Faktor, der als Parameter vorliegt. Beide Blocktypen berechnen ihr Ergebnis dabei je Vektor-Element.

Enabled Subsystem und Enable. Die Funktionsweise des Enabled-Subsystem-Blocks ist im Vergleich zu den vorigen Blöcken ein wenig komplizierter. Ein solcher Block enthält stets einen Enable-Block. Dieser kann, sofern gewünscht, den Wert des Subsystem-Kontrolleingangs ausgeben. In der Definition hierzu in Tabelle 2 bezeichnet die Funktion psys_{dir} (siehe Tabelle 1, Abschnitt j) das diesem Block in der Modellhierarchie direkt übergeordnete Enabled Subsystem. Darüber hinaus besitzt ein Enable-Block einen Parameter, über den sich einstellen lässt, ob die im Subsystem

enthaltenen Blöcke ihre Zustände bei einer (Re-)Aktivierung zurücksetzen. Ein Enabled Subsystem besitzt eine individuelle Aktivitäts- und Zurücksetzungsfunktion. Erstere sagt aus, dass das Subsystem aktiv ist, wenn mindestens ein Vektor-Element des Kontrolleingangs einen positiven Wert enthält. Darauf basierend bestimmen die im Subsystem enthaltenen Blöcke ihre Aktivität über die in Definition 7 dargestellte allgemeine Aktivitätsfunktion. Die Zurücksetzungsfunktion zeigt für die im Subsystem enthaltenen Blöcke an (siehe Definition 9), dass sie ihren Zustand zurückzusetzen haben, wenn (a) der Parameter des Enable-Blocks entsprechend eingestellt ist und (b) das Subsystem zum vorherigen Zeitschritt inaktiv war und nun aktiv geworden ist. Das Ausführungsverhalten bedingt ausgeführter Subsysteme stellt im Übrigen eine Besonderheit dar. Normalerweise berechnen Blöcke ihre Ausgabe(n) nicht bei Inaktivität – ein bedingt ausgeführtes Subsystem allerdings schon. Die dabei ausgegebenen Werte hängen in erster Linie von der (In-)Aktivität ab. Ist das Subsystem aktiv, so werden je Ausgangsport die Eingangswerte des dazugehörigen Outport-Blocks ausgegeben. Bei Inaktivität gibt ein Parameter des jeweiligen Outport-Blocks an, ob ein Initialwert oder der zuletzt ausgegebene Wert erneut ausgegeben werden soll.

Mit den Ausführungen dieses Abschnitts sind die Grundlagen gelegt, um sich im Folgenden mit dem Test von SL/TL-Modellen und der Testdatengenerierung für sie auseinanderzusetzen.

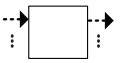
 Inport	PM	**Portnummer:** $n \in \mathbb{N}^{+}$
	out	**Fall 1:** $\exists sb \in B\colon b \in BC_{sb}$ $\forall v \in \mathbb{N}_{[1,d(sb,n)]}\colon \mathbf{out}^{v}_{b,1}(ts) = in^{v}_{sb,n}(ts)$ **Fall 2:** $\forall sb \in B\colon b \notin BC_{sb}$ $\mathbf{out}^{1}_{b,1}(ts) = x$ **mit** $x = mi_{n}(ts) \in MI_{sim} \wedge sim \in SIM$
 Outport	PM	**Portnummer:** $m \in \mathbb{N}^{+}$ **Initialwert:** $iv \in \mathbb{R}^{q}$ $(q \in \mathbb{N}^{+})$ **Subsystem-Ausgabeverhalten bei Inaktivität:** $io \in \mathbb{B}$ (Halten: io, Zurücksetzen auf iv: $\neg$ io)
 Subsystem	out	$\forall pos \in \mathbb{N}_{[1,\lvert OP_{b} \rvert]}\colon \forall v \in \mathbb{N}_{[1,d(op(b,pos))]}\colon$ $\mathbf{out}^{v}_{b,pos}(ts) = in^{v}_{b2,1}(ts)$ **mit** $b2 \in BC_{b} \wedge bt_{b2} = \text{Outport} \wedge (\text{„}m\text{“}, pos) \in PV_{b2}$
 Constant	PM	**Konstante:** $c^{k} \in \mathbb{R}^{\mathbb{N}_{[1,n]}}$
	out	$\forall v \in \mathbb{N}_{[1,n]}\colon \mathbf{out}^{v}_{b,1}(ts) = c^{v}$

Tabelle 2: Blocktypen TL-konformer SL-Modelle (Auszug)

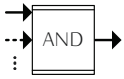
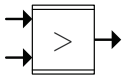
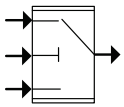
 Logical Operator	PM	**Operator:** $\bullet \in \{\wedge, \vee, \uparrow, \downarrow, \oplus, \neg\}$ **Eingang-Anzahl:** $n \in \mathbb{N}^+$
	out	**Fall 1:** $\bullet = \neg$ $\forall v \in \mathbb{N}_{[1,d(b,1)]}$: $\mathbf{out}^{v}_{b,1}(ts) = \neg(in^{v}_{b,1}(ts))$ **Fall 2:** $\bullet \neq \neg$ **Fall 2.1:** $n > 1$ $\forall v \in \mathbb{N}_{[1,maxd_{in}(b)]}$: $\mathbf{out}^{v}_{b,1}(ts) = \circ(\{in^{vc(v,p)}_{b,pos(p)}(ts) \mid p \in IP_b\})$ **Fall 2.2:** $n = 1$ $\mathbf{out}^{1}_{b,1}(ts) = \circ(\{in^{v}_{b,1}(ts) \mid v \in \mathbb{N}_{[1,d(b,1)]}\})$ **Es sei** $\circ : \mathcal{P}(\mathbb{R}) \to \mathbb{R}$ **mit** $\circ(X) = \begin{cases} 1\text{ , } ((\bullet = \wedge)_\wedge \forall x \in X{:}T(x))_\vee ((\bullet = \vee)_\wedge \exists x \in X{:}T(x))_\vee \\ \quad ((\bullet = \uparrow)_\wedge \exists x \in X{:}F(x))_\vee ((\bullet = \downarrow)_\wedge \forall x \in X{:}F(x))_\vee \\ \quad ((\bullet = \oplus)_\wedge ((\sum_{\substack{x \in X,\\ T(x)}} 1) \,\%\, 2) > 0) \\ 0\text{ , sonst} \end{cases}$
 Relational Operator	PM	**Operator:** $\bullet \in \{=, \neq, >, <, \geqslant, \leqslant\}$
	out	$\forall v \in \mathbb{N}_{[1,maxd_{in}(b)]}$: $\mathbf{out}^{v}_{b,1}(ts) = \begin{cases} 1, in^{vc(v,ip(b,1))}_{b,1}(ts) \bullet in^{vc(v,ip(b,2))}_{b,2}(ts) \\ 0, sonst \end{cases}$
 Switch	PM	**Kriterium:** $\bullet \in \{>, \geqslant, \neq\}$ **Schwellwert:** $t^k \in \mathbb{R}^{\mathbb{N}_{[1,n]}}$ $(\bullet = \text{„}\neq\text{“} \Rightarrow t^k = 0)$
	out	**Fall 1:** $\max(n, d(b,2)) > 1$ $\forall v \in \mathbb{N}_{[1,\max(n,d(b,2))]}$: $\mathbf{out}^{v}_{b,1}(ts) = \begin{cases} in^{vc(v,ip(b,1))}_{b,1}(ts)\text{ , }in^{vc(v,ip(b,2))}_{b,2}(ts) \\ \qquad\qquad \bullet t^{\min(v,n)} \\ in^{vc(v,ip(b,3))}_{b,3}(ts)\text{ , sonst} \end{cases}$ **Fall 2:** $n = d(b,2) = 1$ **Fall 2.1:** $in^{1}_{b,2}(ts) \bullet t^1$ $\forall v \in \mathbb{N}_{[1,d(b,1)]}$: $\mathbf{out}^{v}_{b,1}(ts) = in^{v}_{b,1}(ts)$ **Fall 2.2:** $\neg(in^{1}_{b,2}(ts) \bullet t^1)$ $\forall v \in \mathbb{N}_{[1,d(b,3)]}$: $\mathbf{out}^{v}_{b,1}(ts) = in^{v}_{b,3}(ts)$

Tabelle 2: Blocktypen TL-konformer SL-Modelle (Auszug)

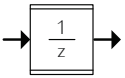
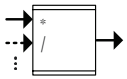
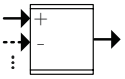
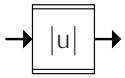
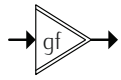
 Unit Delay	PM	**Initiale Ausgabe:** $a^k \in \mathbb{R}^{\mathbb{N}_{[1,n]}}$
	st	$\forall v \in \mathbb{N}_{[1,d(b,1)]}$: $\mathbf{st}_b^v(ts) = \begin{cases} a^{\min(v,n)}, & rs_b(ts) \\ in_{b,1}^v(ts-1), & sonst \end{cases}$
	out	$\forall v \in \mathbb{N}_{[1,d(b,1)]}$: $\mathbf{out}_{b,1}^v(ts) = \begin{cases} st_b^v(ts), & ts > 0 \\ a^{\min(v,n)}, & sonst \end{cases}$
 Product	PM	**Operator je Eingang:** $\bullet : \mathbb{N}_{[1,n]} \to \{\times, \div\}$ oder **Eingang-Anz.:** $n \in \mathbb{N}^+$ $(\forall i \in \mathbb{N}_{[1,n]} : \bullet(i) = \times)$
	out	**Fall 1:** $n > 1$ $\forall v \in \mathbb{N}_{[1,maxd_{in}(b)]}$: $\mathbf{out}_{b,1}^v(ts) =$ $1 \underset{(1)}{\bullet} in_{b,1}^{vc(v,p_1)}(ts) \ldots \underset{(i)}{\bullet} in_{b,i}^{vc(v,p_i)}(ts) \ldots \underset{(n)}{\bullet} in_{b,n}^{vc(v,p_n)}(ts)$ **mit** $i \in \mathbb{N}_{[1,n]} \wedge p_i \in IP_b \wedge i = pos(p_i)$ **Fall 2:** $n = 1$ $\mathbf{out}_{b,1}^1(ts) = 1 \underset{(1)}{\bullet} in_{b,1}^1(ts) \ldots \underset{(1)}{\bullet} in_{b,1}^j(ts) \ldots \underset{(1)}{\bullet} in_{b,1}^m(ts)$ **mit** $m = d(b,1) \wedge j \in \mathbb{N}_{[1,m]}$
	PM	**Operator je Eingang:** $\bullet : \mathbb{N}_{[1,n]} \to \{+, -\}$ oder **Eingang-Anz.:** $n \in \mathbb{N}^+$ $(\forall i \in \mathbb{N}_{[1,n]} : \bullet(i) = +)$
 Sum	out	**Fall 1:** $n > 1$ $\forall v \in \mathbb{N}_{[1,maxd_{in}(b)]}$: $\mathbf{out}_{b,1}^v(ts) =$ $0 \underset{(1)}{\bullet} in_{b,1}^{vc(v,p_1)}(ts) \ldots \underset{(i)}{\bullet} in_{b,i}^{vc(v,p_i)}(ts) \ldots \underset{(n)}{\bullet} in_{b,n}^{vc(v,p_n)}(ts)$ **mit** $i \in \mathbb{N}_{[1,n]} \wedge p_i \in IP_b \wedge i = pos(p_i)$ **Fall 2:** $n = 1$ $\mathbf{out}_{b,1}^1(ts) = 0 \underset{(1)}{\bullet} in_{b,1}^1(ts) \ldots \underset{(1)}{\bullet} in_{b,1}^j(ts) \ldots \underset{(1)}{\bullet} in_{b,1}^m(ts)$ **mit** $m = d(b,1) \wedge j \in \mathbb{N}_{[1,m]}$
 Abs	out	$\forall v \in \mathbb{N}_{[1,d(b,1)]}$: $\mathbf{out}_{b,1}^v(ts) = \lvert in_{b,1}^v(ts) \rvert$
 Gain	PM	**Faktor:** $gf^k \in \mathbb{R}^{\mathbb{N}_{[1,n]}}$
	out	$\forall v \in \mathbb{N}_{[1,\max(n,d(b,1))]}$: $\mathbf{out}_{b,1}^v(ts) = gf^{\min(v,n)} \cdot in_{b,1}^{vc(v,ip(b,1))}(ts)$

Tabelle 2: Blocktypen TL-konformer SL-Modelle (Auszug)

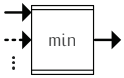
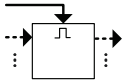
 Min/Max	PM	**Funktion:** $\bullet \in \{\min, \max\}$, $\bullet : \mathcal{P}(\mathbb{R}) \to \mathbb{R}$ **Eingang-Anzahl:** $n \in \mathbb{N}^+$
	out	**Fall 1:** $n > 1$ $\forall v \in \mathbb{N}_{[1, maxd_{in}(b)]}$: $\mathbf{out^v_{b,1}}(ts) = \bullet(\{in^{vc(v,p)}_{b,pos(p)}(ts) \mid p \in IP_b\})$ **Fall 2:** $n = 1$ $\mathbf{out^1_{b,1}}(ts) = \bullet(\{in^v_{b,1}(ts) \mid v \in \mathbb{N}_{[1,d(b,1)]}\})$
 Enable	PM	**Zustände der Subsystem-Blöcke** **bei (Re-)Aktivierung:** $az \in \mathbb{B}$ (Halten: az, Zurücksetzen: $\neg$ az) **Aktivierung des Ausgangs:** $a \in \mathbb{B}$
	out	**Voraussetzung:** a $\forall v \in \mathbb{N}_{[1,d(pb,\lvert IP_{pb}\rvert)]}$: $\mathbf{out^v_{b,1}}(ts) = in^v_{pb,\lvert IP_{pb}\rvert}(ts)$ **wobei** $pb = psys_{dir}(b)$
 Enabled Subsystem	act	$\mathbf{act_b}(ts) = \begin{cases} 1 & , \exists v \in \mathbb{N}_{[1,dc]}: in^v_{b,\lvert IP_b\rvert}(ts) > 0 \\ 0 & , \text{sonst} \end{cases}$ **wobei** $dc = d(b, \lvert IP_b\rvert)$
	rs	$\mathbf{rs_b}(ts) = \begin{cases} 1 & , ts > 0 \wedge \neg az \wedge \neg act_b(ts-1) \wedge act_b(ts) \\ 0 & , \text{sonst} \end{cases}$ **wobei** $eb \in BC_b \wedge bt_{eb} = \text{Enable} \wedge (\text{„az“}, az) \in PV_{eb}$
	out	$\forall pos \in \mathbb{N}_{[1,\lvert OP_b\rvert]}$: **Fall 1:** $act_b(ts)$ $\forall v \in \mathbb{N}_{[1,d(ob_{pos},1)]}$: $\mathbf{out^v_{b,pos}}(ts) = in^v_{ob_{pos},1}(ts)$ **Fall 2:** $\neg act_b(ts) \wedge io_{pos}$ **Fall 2.1:** $ts > 0$ $\forall v \in \mathbb{N}_{[1,d(ob_{pos},1)]}$: $\mathbf{out^v_{b,pos}}(ts) = out^v_{b,pos}(ts-1)$ **Fall 2.2:** $ts = 0$ $\forall v \in \mathbb{N}_{[1,q_{pos}]}$: $\mathbf{out^v_{b,pos}}(ts) = iv^v_{pos}$ **Fall 3:** $\neg act_b(ts) \wedge \neg io_{pos}$ $\forall v \in \mathbb{N}_{[1,q_{pos}]}$: $\mathbf{out^v_{b,pos}}(ts) = iv^v_{pos}$ **mit** $ob_{pos} \in BC_b \wedge bt_{ob_{pos}} = \text{Outport} \wedge (\text{„m“}, pos) \in PV_{ob_{pos}} \wedge$ $io_{pos} \in \mathbb{B} \wedge (\text{„io“}, io_{pos}) \in PV_{ob_{pos}} \wedge$ $iv^x_{pos} \in \mathbb{R} \wedge (\text{„iv“}, iv^x_{pos}) \in PV_{ob_{pos}} \wedge x \in \mathbb{N}_{[1,q_{pos}]} \wedge$ $q_{pos} \in \mathbb{N}^+ \wedge (\text{„q“}, q_{pos}) \in PV_{ob_{pos}}$

Tabelle 2: Blocktypen TL-konformer SL-Modelle (Auszug)

2.1.3 TESTGRUNDLAGEN

In diesem Abschnitt wird der Test von eingebetteten Systemen und deren Software grundlegend behandelt. Zudem werden wichtige Begriffe eingeführt. Der Modelltest nimmt dabei eine zentrale Rolle ein.

Testen bedeutet in der industriellen Praxis in erster Linie, für die in der Spezifikation eines Systems niedergeschriebenen funktionalen Anforderungen die Einhaltung durch das System zu prüfen. Das zu prüfende System wird als Testobjekt bezeichnet, um von seiner Natur (wie beispielsweise Steuergerät, Modell oder Code) zu abstrahieren. Diese Art des Testens wird üblicherweise als Funktionstest oder als Anforderungstest bezeichnet. Aus den Anforderungen werden dabei Testfälle abgeleitet. Ein Testfall definiert ein Szenario, welchem sich das Testobjekt stellen muss, und legt das erwartete Testobjekt-Verhalten fest. Das Szenario besteht hierbei aus der Umgebung, welche für die Durchführung des Tests herzustellen ist, sowie aus Testschritten. Die Testschritte beinhalten entweder Testdaten oder diese lassen sich aus ihnen ableiten. Testdaten sind die rohen Eingabedaten (Stimuli) für die Ausführung des Testobjekts. Findet eine automatisierte Testausführung statt, so wird ein Testfall in eine Testimplementierung überführt. Dies kann modellbasiert oder durch ein Skript erfolgen. Bei der Testausführung wird das Verhalten des Testobjekts überwacht und protokolliert. Anschließend gilt es, die Konformität der Ergebnisse mit dem erwarteten Testobjekt-Verhalten zu prüfen. Wurde das erwartete Verhalten zuvor in ein Testorakel überführt, beispielsweise in Form eines Prüfskripts, so kann die Konformitätsprüfung automatisiert erfolgen.

Die Entwicklung eingebetteter Systeme orientiert sich in der Automobilindustrie im Normalfall an dem Vorgehensmodell *V-Modell XT* [66]. Eine auf den Automobilbereich gemünzte Abwandlung hiervon ist in Abbildung 2 dargestellt. Der linke Ast im V-Modell beschreibt die Prozessschritte zur Planung und Spezifikation des Produkts. Die Realisierung der einzelnen Bausteine des Produkts finden im V-Modell im unteren Abschnitt statt. Der rechte Ast stellt die Teststufen dar, welche sich aus der phasenweisen Integration der Bausteine zum Gesamtprodukt ergeben. Klassischerweise dienen die Spezifikationsdokumente im linken Ast jeweils dem Funktionstest des Testobjekts auf gleicher Ebene im rechten Ast. Die Entwicklungspraxis weicht von dieser Grundidee aber meist ab: Spezifikationsdokumente werden beispielsweise gebündelt, das heißt, nicht für jede Test- oder Integrationsstufe existiert überhaupt eine 1:1 zuordenbare Spezifikation. Daher müssen aus den Anforderungen abgeleitete Testfälle den Teststufen erst zugeordnet werden. Derlei Aspekte werden in dieser Arbeit allerdings nicht vertieft.

Es existieren vielerlei Testmethoden und -techniken, um bei einem Funktionstest systematisch und lückenlos vorzugehen. Hierbei steht die Ab-

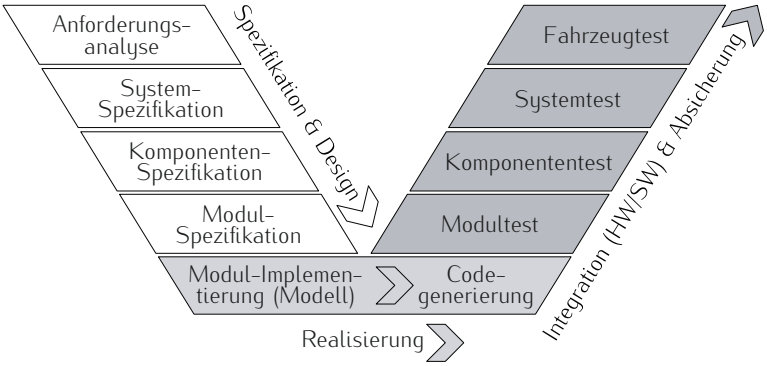


Abbildung 2: Entwicklungs- und Testprozess eingebetteter Systeme im Automobilbereich gemäß V-Modell

leitung der Testfälle im Vordergrund. Als Beispiel sei die Klassifikationsbaum-Methode von Grimm [51] genannt. Diese setzt auf eine systematische Analyse und Kategorisierung des zu testenden Aspekts. Conrad entwickelte die Methode für den Test von eingebetteten System im Automobilbereich weiter [28]. Trotz verschiedenster Vorgehensweisen bleibt die Ableitung geeigneter Testfälle, beziehungsweise der Testdaten, meist eine manuell durchzuführende Aufgabe. Der Strukturtest, welcher im Anschluss an diesen Abschnitt ausführlich behandelt wird, spielt im Vergleich zum Funktionstest in der Praxis eine untergeordnete Rolle. Dies liegt unter anderem daran, da er sich überhaupt nur in der unteren Ebene des V-Modells anbietet (Modell oder Code). Das manuelle Ableiten der Testdaten ist beim Strukturtest allerdings ungemein schwerer als beim Funktionstest. Um nämlich in den Tiefen eines Programms oder Modells einen bestimmten Zustand zu erreichen, gilt es, ausgehend von den Eingängen vielfältige Abhängigkeiten, Strukturen und Berechnungsschritte zu berücksichtigen. Daher ist insbesondere bei dieser Aktivität eine Automatisierung besonders wertvoll.

Im Bemühen um eine Automatisierung setzt diese Arbeit ihren Fokus auf den Modelltest und nicht etwa, wie viele Arbeiten zuvor, auf den Test von Code. Diese Fokussierung ist allerdings nicht mit dem Anspruch verbunden, der Modelltest könne den Test des Codes – selbst wenn dieser automatisch generiert ist – in Gänze ablösen. Die Motivation besteht vielmehr darin, Fehler in der Software eines eingebetteten Systems so nah wie möglich an ihrem Entstehungspunkt zu lokalisieren. Um Abweichungen zwischen Modell- und Codeverhalten festzustellen und um Fehlverhalten zu erkennen, dass durch den konfigurierbaren Prozess der Codegenerierung entsteht, sind Tests des Codes weiterhin notwendig. Der bereits mehrfach erwähnte Industriestandard ISO 26262 verlangt eine

Betrachtung der Überdeckung bei Entwicklung und Test von Software-Modulen und weist explizit auf die Möglichkeit hin, diese Betrachtung bereits auf Modellebene führen zu können. Baresel et al. schlagen in ihrer Studie zur Untersuchung von Zusammenhängen zwischen Code- und Modellüberdeckung [10] folgende Gestaltung des Testprozesses im unteren Teil des V-Modells vor:

1. Durchführung von Funktionstests auf Modellebene, um die korrekte Umsetzung der Anforderungen zu prüfen
2. Messung der erreichten Modellüberdeckung
3. Ableitung zusätzlicher Testfälle, um nicht-überdeckte Modellelemente zu überdecken (Strukturtest), und Durchführung/Auswertung dieser zusätzlichen Tests
4. Ausführung der Testfälle aus Schritt 1 auf dem generierten Code, um auch hier die Konformität mit den Anforderungen zu prüfen
5. Messung der erreichten Codeüberdeckung (nur, falls eine hohe Abdeckung sowohl des Modells als auch des Codes gefragt ist)
6. Ableitung zusätzlicher Testfälle, um nicht-überdeckte Code-Elemente zu überdecken, und Durchführung/Auswertung dieser zusätzlichen Tests (ebenfalls nur, falls eine hohe Abdeckung sowohl des Modells als auch des Codes verlangt wird)

Diese Arbeit widmet sich insbesondere der Automatisierung des dritten Schritts. Wie im nächsten Abschnitt zu erfahren ist, können die Ansätze dieser Arbeit auch beim ersten Schritt unterstützen. Der entwickelte Tool-Prototyp deckt darüber hinaus auch den zweiten Schritt ab. Für eine umfassende Auflistung anders gelagerter Forschungsarbeiten, welche sich mit der Qualitätssicherung von SL-Modellen beschäftigen, sei auf Elberzhager et al. [37] verwiesen.

2.2 SZENARIEN AUTOMATISIERTER TESTDATENGENERIERUNG

Für den Modelltest gibt es verschiedene Szenarien, in denen eine automatisierte Generierung von Testdaten denkbar ist. Sadeghipour und Lim haben die möglichen Anwendungsfelder in einer Publikation [104] sehr treffend herausgearbeitet:

- Strukturtests: Überdeckung von bei Funktionstests nicht überdeckten Modellelementen

- Funktionstests: (Teil-)Automatisierung bei der Ableitung von Testfällen aus der Spezifikation, sofern ein Testorakel über das erwartete Modellverhalten existiert
- Entwicklungsbegleitende Tests: Erzeugung von Testdaten zur Überprüfung eines ausgewählten Modellbestandteils während dessen Entwicklung
- Back-to-Back-Tests: Erzeugung von Testdaten für die Überprüfung der Äquivalenz von Modell und generiertem Code

Diese vier Bereiche werden in den nachfolgenden Abschnitten etwas näher beleuchtet. Der Strukturtest wird dabei bewusst intensiver behandelt, da er den Fokus dieser Arbeit darstellt. Allerdings wird auch gezeigt, dass die Problemstellung der strukturorientierten Testdatengenerierung durchaus auf die Problemstellungen der drei anderen Bereiche übertragbar ist – und der Strukturtest somit im Kontext dieser Arbeit als repräsentativer Anwendungsfall angesehen werden kann.

2.2.1 STRUKTURTEST

Der Strukturtest orientiert sich zur Herleitung von Testdaten an der inneren Struktur eines Testobjekts. Hintergrundgedanke ist, dass ein Testobjekt nur dann als vollständig getestet gelten kann, wenn auch alle Kommandos oder Verzweigungen im Inneren des Testobjekts durch Tests ausgeführt oder beschritten wurden. Aus den Anforderungen abgeleitete Testfälle erfüllen diese Maßgabe nicht zwingend. Nicht nur Baresel et al. [10] empfehlen daher, wie zuvor beschrieben, eine ergänzende Durchführung von Strukturtests. Auch Grimm [51] bezeichnet die Kombination von Funktionstest und Strukturtest als eine effektive Teststrategie.

Die Zielsetzung eines Strukturtests ist bestimmt durch die Wahl des gewünschten Überdeckungskriteriums. Der Strukturtest ist ein Verfahren, welches traditionell dem Test von Programmcode dient. In diesem Bereich werden primär kontrollflussorientierte Überdeckungskriterien verwendet – in wissenschaftlichen Arbeiten wie in der Praxis. Derlei Kriterien nutzen dabei folgende, vom Programmcode abstrahierende Strukturen: den *Kontrollflussgraph* (KFG) , wie er unter anderem von Liggesmeyer [82] behandelt wurde, oder den *Programmablaufplan* (PAP) , welcher auch *Flowchart* genannt wird und der in den Normen DIN 66001 und ISO 5807 definiert ist. Ein wesentlicher Unterschied zwischen diesen beiden Darstellungsformen ist, dass Kontrollflussbedingungen in einem PAP explizit dargestellt sind. Während für Kriterien wie der *Pfadüberdeckung*, *Anweisungsüberdeckung* oder der *Zweig-* bzw. *Entscheidungsüberdeckung* (DC) die Betrachtung des KFG genügt, sind für die *Bedingungsüberdeckung* (CC) oder das Kriterium der *Modified Condition/Decision Coverage* (MC/DC, zu

Programmcode

```
1: if (a > b)
2:     x = 0;
3: else
4:     x = 1;
5: c = x + 2;
6: if (a==c || b>c)
7:     x = x − 1;
8: return x;
```

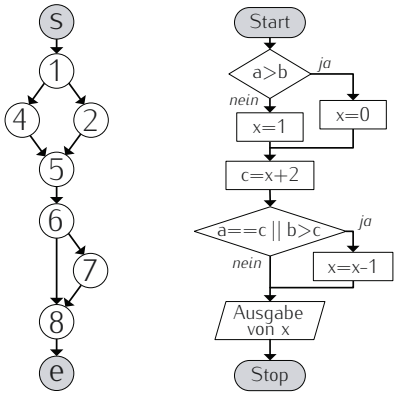


Abbildung 3: Darstellung des Kontrollflusses eines Programmcode-Beispiels (links) als Kontrollflussgraph (mitte) und als Programmablaufplan (rechts)

Deutsch: Modifizierte Bedingungs-/Entscheidungsüberdeckung) auch die Kontrollflussbedingungen von Relevanz. Beide Formen der Darstellung werden in der Literatur im Kontext des Strukturtests dennoch größtenteils synonym verwendet.

Abbildung 3 dient als Beispiel zum Verständnis des Strukturtests und seinen Überdeckungskriterien. Zu sehen sind ein Code-Fragment sowie dessen Kontrollfluss-Strukturen. Der PAP enthält sowohl die Bedingungen der Verzweigungen als auch die Anweisungen aus dem Code. Im KFG sind Bedingungen und Anweisungen hinter den entsprechenden Code-Zeilennummern verborgen. Wird nun die DC als Kriterium herangezogen, so gilt es, im KFG beispielsweise sowohl einmal von Knoten 1 zu 4 (Entscheidung *nein* im PAP) als auch einmal von 1 zu 2 (Entscheidung *ja* im PAP) überzugehen. Gesucht wären hierzu die Testdaten, also Belegungen für die Variablen a und b, welche diese Wege in KFG oder PAP beschreiten. Ein weiteres Beispiel: Um die CC für das gegebene Programm zu erfüllen, müssen alle Teilbedingungen der Bedingungen in Verzweigungen und Schleifen jeweils einmal mit *wahr* und einmal mit *falsch* ausgewertet werden. Für Knoten 6 betrifft dies die Teilbedingungen a==c und b>c, für Knoten 1 hingegen nur a>b. Insgesamt leiten sich für die CC also sechs Ziele ab.

Wie im Beispiel ersichtlich, sind zur Erfüllung eines Überdeckungskriteriums üblicherweise mehrere Strukturelemente oder Testobjekt-Zustände zu erreichen. Eine einzelne solche Zieldefinition wird in dieser Arbeit im Allgemeinen als *Überdeckungsziel* (CG, Plural: CGs) bezeichnet, wobei die Abkürzung dem englischen Begriff *Coverage Goal* entstammt. Eine konkrete Belegung der Testobjekt-Eingänge wird als Testdatum bezeich-

net. Die bei einem Strukturtest entstehende Menge von Testdaten, welche gemeinsam möglichst viele CGs erreichen, wird in dieser Arbeit Testsuite genannt. Die Qualität einer Testsuite hinsichtlich der Erfüllung eines Überdeckungskriteriums wird durch den Überdeckungsgrad bemessen. Diese Arbeit folgt der Definition des Überdeckungsgrads von Baluda et al. [9], da diese die potentielle Unerreichbarkeit einzelner CGs berücksichtigt:

$$\text{Überdeckungsgrad} = \frac{|\text{ Erreichte CGs }|}{|\text{ Gesamte CGs }| - |\text{ Unerreichbare CGs }|} \tag{13}$$

Die vollständige Erfüllung eines Überdeckungskriteriums ist dementsprechend dem Erreichen des maximalen Überdeckungsgrads von 100% gleichzusetzen.

Die für die Code-Welt existierenden Strukturtest-Konzepte lassen sich auf die Modell-Welt übertragen. Mögliche Überdeckungskriterien für SL/TL-Modelle [97, 114, 132] sind in Tabelle 3 aufgeführt. Für SF-Automaten existieren zwei Automaten-typische Kriterien, die Transitions- und die Zustandsüberdeckung. Sowohl für SF als auch für den Blockdiagramm-Anteil eines SL/TL-Modells lassen sich die aus der Code-Welt bekannten Kriterien DC und CC sowie MC/DC anwenden. Für zwei spezielle Klassen von Blocktypen existieren darüber hinaus eigene Kriterien. Für bedingte Subsysteme beispielsweise gibt es die *Subsystem-Bedingungsüberdeckung* (CSC) , welche die Aktivität eines jeden bedingten

Modellbestandteil	*Überdeckungskriterium*	*Abkürzung*
SL/TL	**Entscheidungsüberdeckung**	DC
	Bedingungsüberdeckung	CC
	Modifizierte Bedingungs-/ Entscheidungsüberdeckung	MC/DC
	Lookup-Tabellen-Überdeckung	LTC
	Subsystem-Bedingungsüberdeckung	CSC
SF	Zustandsüberdeckung	SC
	Transitionsüberdeckung	TC
	Entscheidungsüberdeckung	DC
	Bedingungsüberdeckung	CC

Tabelle 3: Modellüberdeckungskriterien für SL/TL-Blockdiagramme sowie SF-Zustandsdiagramme

Subsystems im Modell, jeweils zu mindestens einem beliebigen Zeitschritt, verlangt.

Während die zuvor genannten Code-Überdeckungskriterien stark standardisiert sind, ist die Verwendung und Interpretation der gelisteten Modell-Überdeckungskriterien noch stark werkzeugabhängig [29]. Ein Beispiel: Gemäß der Dokumentation von SL ist ein Relational-Operator-Block nicht relevant für die DC. In dieser Arbeit wird dieser Blocktyp allerdings bei der DC berücksichtigt, da er nach Ansicht des Autors eine Entscheidung impliziert. Darüber hinaus ist anzumerken: Die vorliegende Arbeit behandelt nur eine Auswahl der verfügbaren Überdeckungskriterien. Diese sind in der Tabelle durch Fettdruck hervorgehoben. Die entwickelten Techniken sind allerdings ebenso für die nicht behandelten Kriterien anwendbar. Einzig für die SF-spezifischen Kriterien ist eine Erweiterung der Techniken vonnöten.

In der Anwendung des Strukturtests für SL/TL-Modelle besteht ein CG aus einem Ausdruck cg, der das zu erfüllende Ziel beschreibt. Ein solcher Ausdruck stellt eine Bedingung an ein oder mehrere Modellsignal/-e. Die Menge E sei die Menge aller möglichen, für ein Modell definierbaren CGs. Der Bezeichner E steht hierbei für den englischen Begriff *Expression* (zu Deutsch: Ausdruck). Die folgenden Vorschriften definieren den Aufbau eines CG $cg \in E$:

$$E = E_{log} \cup E_{rel} \tag{14}$$

Die Menge E_{rel} enthält vergleichende Ausdrücke zwischen Signalen und/oder Konstanten über relationale Operatoren. In E_{log} sind hingegen logische Ausdrücke (Negation, Konjunktion, Disjunktion) über relationale Ausdrücke oder andere logische Ausdrücke enthalten.

$$E_{log} = \left\{ \bigwedge_{i=1}^{n} \varphi_i,\ \bigvee_{i=1}^{n} \varphi_i,\ \neg\varphi \right\} \tag{15}$$

$$E_{rel} = \{\xi_1 = \xi_2,\ \xi_1 \neq \xi_2,\ \xi_1 > \xi_2,\ \xi_1 < \xi_2,\ \xi_1 \geqslant \xi_2,\ \xi_1 \leqslant \xi_2\}$$

Hierbei gilt $n \in \mathbb{N}_{\geqslant 2}$ und $\varphi, \varphi_i \in E$. Mit ξ_1 und ξ_2 ist jeweils entweder ein Signal (mit Vektorindex-Angabe) oder eine Konstante bezeichnet, das heißt es gilt $\xi_1, \xi_2 \in L$ mit:

$$L = \{s^\nu \mid s \in S \wedge \nu \in \mathbb{N}_{[1,\,d(s)]}\} \cup \{c \mid c \in \mathbb{R}\} \tag{16}$$

Für eine Erweiterung dieser Definition zur Berücksichtigung SF-relevanter Überdeckungskriterien müssten unter anderem mathematische Operatoren miteinbezogen werden.

In Tabelle 4 ist die Überdeckungsziel-Menge $CG(b, k)$, die sich je nach Überdeckungskriterium k für einen Modellblock b ableitet, für eine Auswahl der in SL/TL-Modellen verfügbaren Blocktypen gegeben. So

ist der Logical-Operator-Block sowohl für DC als auch CC relevant. Für DC ist erforderlich, dass jedes Vektorelement des ausgehenden Signals jeweils einmal gleich 0 und einmal gleich 1 ist. Im Sinne der CC ist es notwendig, dass jedes Vektorelement eines jeden Blockeingangs jeweils einmal 0 und einmal ungleich 0 ist. Die Bedingung eines Relational-Operator-Blocks muss zur Erfüllung der DC ebenfalls einmal zutreffen und einmal nicht. Auch hier ist die Bedingung pro Vektorelement zu verstehen. Dies gilt ebenso für Switch- und Abs-Blöcke. Für den Switch-Block verlangt DC, dass die beiden Dateneingänge jeweils einmal zum Blockausgang geleitet werden. Ebenfalls für DC muss am Eingang eines Abs-Blocks einmal ein negativer und einmal ein nicht-negativer Wert vorliegen. Der Min/Max-Block muss zur Erfüllung der DC an jedem seiner Eingänge mindestens einmal den Minimal-/Maximalwert, den der Block ausgibt, anliegen haben. Ein Enabled-Subsystem-Block muss in seinem Kontrollsignal einmal mindestens einen positiven Wert enthalten und einmal muss das Gegenteil der Fall sein, um DC zu erfüllen. Für CSC genügt das einmalige Vorliegen eines positiven Werts im Kontrollsignal. CC verlangt, dass jedes Vektorelement des Kontrollsignals jeweils einmal positiv und einmal nicht-positiv ist.

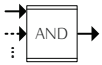
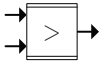
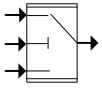
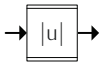
Blocktyp BT	*Krit.* k	*Überdeckungsziele* CG(b,k)
 Logical Operator	DC	$\{\langle s^{\nu}=0\rangle, \langle s^{\nu}=1\rangle \mid s \in OS_b \wedge \nu \in \mathbb{N}_{[1,d(s)]}\}$
	CC	$\{\langle s^{\nu}\neq 0\rangle, \langle s^{\nu}=0\rangle \mid s \in IS_b \wedge \nu \in \mathbb{N}_{[1,d(s)]}\}$
 Relational Operator	DC	$\left\{ \begin{array}{l} \left\langle s_1^{vc(\nu,s_1)} \bullet s_2^{vc(\nu,s_2)} \right\rangle, \\ \left\langle s_1^{vc(\nu,s_1)} \circ s_2^{vc(\nu,s_2)} \right\rangle \\ \mid s_1 = sig_{in}(b,1) \\ \wedge s_2 = sig_{in}(b,2) \\ \wedge \nu \in \mathbb{N}_{[1,maxd_{in}(b)]} \end{array} \right\}$ wobei $\circ := \begin{cases} \neq, \bullet = „=“ \\ =, \bullet = „\neq“ \\ \leqslant, \bullet = „>“ \\ \geqslant, \bullet = „<“ \\ <, \bullet = „\geqslant“ \\ >, \bullet = „\leqslant“ \end{cases}$
 Switch	DC	$\left\{ \begin{array}{l} \left\langle s^{vc(\nu,s)} \bullet t^{min(\nu,n)} \right\rangle, \\ \left\langle s^{vc(\nu,s)} \circ t^{min(\nu,n)} \right\rangle \\ \mid s = sig_{in}(b,2) \\ \wedge \nu \in \mathbb{N}_{[1,max(n,d(s))]} \end{array} \right\}$ wobei $\circ := \begin{cases} \leqslant, \bullet = „>“ \\ <, \bullet = „\geqslant“ \\ =, \bullet = „\neq“ \end{cases}$
 Abs	DC	$\{\langle s^{\nu}<0\rangle, \langle s^{\nu}\geqslant 0\rangle \mid s \in IS_b \wedge \nu \in \mathbb{N}_{[1,d(s)]}\}$

Tabelle 4: Überdeckungskriterien-relevante SL/TL-Blocktypen und die sich ableitenden Überdeckungsziele (Auszug)

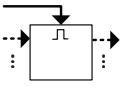
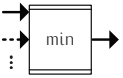
Blocktyp BT	*Krit.* k	*Überdeckungsziele* CG(b,k)
 Enabled Subsystem	CSC	$\left\{ \left\langle \bigvee_{\nu\in\mathbb{N}_{[1,d(s)]}} s^{\nu}>0 \right\rangle \mid s=\mathrm{sig}_{\mathrm{in}}(b,\lvert IP_b\rvert) \right\}$
	DC	$\left\{ \left\langle \bigvee_{\nu\in\mathbb{N}_{[1,d(s)]}} s^{\nu}>0 \right\rangle, \left\langle \bigvee_{\nu\in\mathbb{N}_{[1,d(s)]}} s^{\nu}\leqslant 0 \right\rangle \mid s=\mathrm{sig}_{\mathrm{in}}(b,\lvert IP_b\rvert) \right\}$
	CC	**Voraussetzung:** $d(s)>1$ für $s=\mathrm{sig}_{\mathrm{in}}(b,\lvert IP_b\rvert)$ $\left\{ \langle s^{\nu}>0\rangle, \langle s^{\nu}\leqslant 0\rangle \mid \nu\in\mathbb{N}_{[1,d(s)]} \right\}$
 Min/Max	DC	**Fall 1:** $n>1$: $\left\{ \left\langle \bigwedge_{\substack{j\in\mathbb{N}_{[1,i-1]},\\ s_2=\mathrm{sig}_{\mathrm{in}}(b,j)}} s^{vc(\nu,s)}\circ s_2^{vc(\nu,s_2)} \wedge \bigwedge_{\substack{k\in\mathbb{N}_{[i+1,n]},\\ s_3=\mathrm{sig}_{\mathrm{in}}(b,k)}} s^{vc(\nu,s)}\star s_3^{vc(\nu,s_3)} \right\rangle \mid i\in\mathbb{N}_{[1,n]}\wedge s=\mathrm{sig}_{\mathrm{in}}(b,i)\wedge\nu\in\mathbb{N}_{[1,\mathrm{maxd_{in}}(b)]} \right\}$ **Fall 2:** $n=1$: $\left\{ \left\langle \bigwedge_{\nu_2\in\mathbb{N}_{[1,\nu-1]}} s^{\nu}\circ s^{\nu_2} \wedge \bigwedge_{\nu_3\in\mathbb{N}_{[\nu+1,d(s)]}} s^{\nu}\star s^{\nu_3} \right\rangle \mid s=\mathrm{sig}_{\mathrm{in}}(b,1)\wedge\nu\in\mathbb{N}_{[1,d(s)]} \right\}$ wobei $\circ :=$ „$<$“, $\star :=$ „$\leqslant$“ für $\bullet=\min$ und $\circ :=$ „$>$“, $\star :=$ „$\geqslant$“ für $\bullet=\max$

Tabelle 4: Überdeckungskriterien-relevante SL/TL-Blocktypen und die sich ableitenden Überdeckungsziele (Auszug)

Die in der Tabelle enthaltenen Definitionen der Block-CGs vernachlässigen der Übersichtlichkeit halber ein Detail. Und zwar werden die CG-Ausdrücke jeweils mit einem weiteren Ausdruck verundet, wenn sich der zugrunde liegende Block in der Modellhierarchie unterhalb bedingt ausgeführter Subsysteme befindet. Ein CG-Ausdruck wird in diesem Fall mit der oder den relevanten Aktivierungsbedingung/-en verknüpft. Denn ein CG kann selbstverständlich nur dann erfüllt sein, wenn die an ihm beteiligten Signale zum betrachteten Zeitschritt auch einen Wert enthalten. Hierzu ein Beispiel: Ein AND-Block mit dem Ausgangssignal s_1 befindet sich in einem Enabled Subsystem, welches das Signal s_2 an seinem für die Aktivierung und Deaktivierung relevanten Eingang besitzt. Bezüglich des Kriteriums der Entscheidungsüberdeckung leitet

sich unter anderem das CG $s_1{=}1$ ab. Dieses kann jedoch bei einer Modellausführung nur dann erfüllt sein, wenn das Enabled Subsystem aktiv ist. Dies ist der Fall, wenn $s_2{>}0$ eintritt. Daher wird das CG $s_1{=}1$ um diese Bedingung erweitert, das heißt das CG lautet $s_2{>}0 \wedge s_1{=}1$.

In Ergänzung zu diesen Strukturtest-Grundlagen soll das Thema der Vergleichbarkeit von Code- und Modell-Überdeckungskriterien an dieser Stelle nicht unerwähnt bleiben. Baresel et al. [10] untersuchten den Zusammenhang zwischen Code- und Modell-Überdeckungskriterien anhand von SL-Modellen und dem aus diesen mit TL generierten Code. Betrachtet wurden Bedingungs- und Entscheidungsüberdeckung. Die Autoren kamen zu dem Schluss, dass eine deutlich stärkere Wechselbeziehung besteht als zuvor angenommen, wenn Modell und Code mit denselben Testdaten ausgeführt werden. Auf Grundlage ihrer Ergebnisse wagten die Autoren sogar die Schlussfolgerung, dass die Betrachtung von Modell-Überdeckungskriterien diejenige von Code-Überdeckungskriterien in Zukunft möglicherweise sogar ablösen könnte. Diese Aussage ist allerdings strikt auf den betrachteten Kontext der Verwendung von SL und dem Einsatz eines Codegenerators zu beziehen. Des Weiteren geben sowohl Baresel et al. als auch Kirner [74] zu bedenken, dass die Nähe zwischen Modellüberdeckung und Codeüberdeckung letzten Endes stark vom betrachteten Codegenerator abhängt.

2.2.2 FUNKTIONSTEST

Der Funktionstest wurde in Abschnitt 2.1.3 bereits grundlegend eingeführt. Eine automatisierte Testdatengenerierung zum Zwecke funktionaler Tests bedeutet in der Regel, automatisiert Testfälle oder Testdaten aus der dem Testobjekt zugrunde liegenden Spezifikation abzuleiten. Bei Verwendung natürlichsprachlicher Spezifikationsdokumente, wie es in der Automobilindustrie weitestgehend gang und gäbe ist, gestaltet sich dies sehr schwierig. Wird dennoch eine automatisierte Testdatengenerierung angestrebt, so greift man meist auf formalere Spezifikationsformen zurück. Die Verwendung von Zustandsautomaten (Testfallautomaten) ist hierbei eine weit verbreitete Form. Derlei sogenannte Testmodelle bilden die Umgebung des Testobjekts hinsichtlich einer oder mehrerer Anforderungen aus der Spezifikation ab. Oftmals spezifizieren sie auch erwartetes Testobjektverhalten. Aufgabe einer automatisierten Testdatengenerierung ist es in diesem Kontext, Testdaten aus einem Testmodell abzuleiten. Auch hierbei spielen Überdeckungskriterien eine Rolle. So wäre es beispielsweise sinnvoll, sicherzustellen, dass alle Zustände oder Transitionen eines Testfallautomaten erreicht werden. Dieses Testvorgehen fand bereits in Abschnitt 2.1 unter dem Begriff *modellbasierter Test* Erwähnung.

Das in dieser Arbeit vorgestellte Automatisierungsverfahren ist allerdings zu einer, zumindest in der Zielsetzung, anders gelagerten Art von Funktionstests anwendbar. Hierbei geht es um die Frage, ob das Testobjekt, in diesem Fall ein SL/TL-Modell, möglicherweise Eigenschaften verletzt, deren Einhaltung durch die Spezifikation verlangt wird oder die sich aus dieser ableiten lassen. Es gilt, Testdaten zu generieren, die bei Ausführung des Testobjekts einen spezifizierten Fehlerfall provozieren. Derartige Testdaten lassen sich manuell meist nur mit hohem Aufwand finden – insbesondere, weil ein solcher Negativtest vom Entwickler oder Tester eine pessimistische Grundhaltung bezüglich der Korrektheit des Testobjekts verlangt. Bei diesem Anwendungsfall einer automatisierten Testdatengenerierung stellt sich die Frage, wie der betrachtete Fehlerfall oder die zu verletzende Eigenschaft spezifiziert werden können oder sollten. Eine Antwort auf diese Frage richtet sich vor allem nach dem eingesetzten Automatisierungsverfahren. So ist es beispielsweise möglich, einen Fehlerfall durch eine Zielfunktion zu beschreiben, wie es bei Verwendung des in Abschnitt 2.4 vorgestellten suchbasierten Ansatzes in der Vergangenheit häufig gemacht wurde.

Eine andere Möglichkeit ist in Abbildung 4 beispielhaft veranschaulicht. Hierbei wird die zu verletzende Testobjekt-Eigenschaft modellbasiert formuliert. Denkbar ist, die Sprachmittel von SL und TL zur Definition einer Eigenschaft einzusetzen. Die Eingänge der Eigenschaftsdefinition

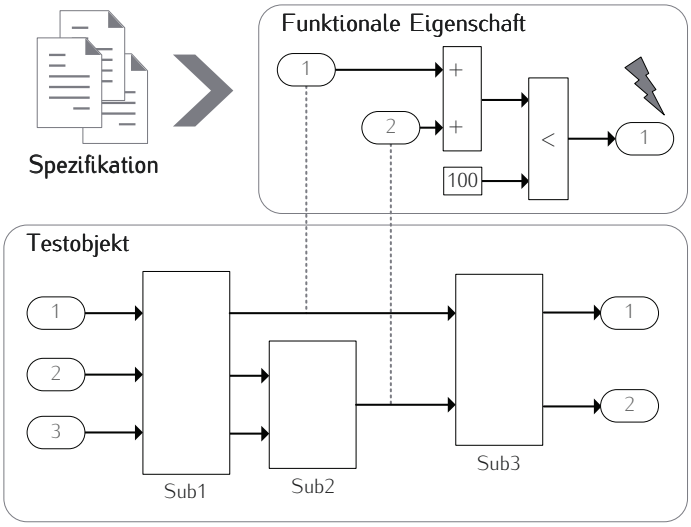


Abbildung 4: Automatisierte Testdatengenerierung mit dem Ziel der Provokation der Verletzung einer aus der Spezifikation abgeleiteten, modellbasiert definierten Eigenschaft

enthalten dabei Referenzen auf interne Signale des zu testenden SL/TL-Modells. Sollen nun Testdaten generiert werden, welche zu einer Verletzung der in der Abbildung dargestellten Eigenschaft führen, so muss es das Ziel sein, am Ausgang der modellbasierten Eigenschaftsdefinition den Wert 0 (*falsch*) zu erreichen. Nun entspricht eine solche Zieldefinition aus formaler Sicht der Definition eines Überdeckungsziels bei der strukturorientierten Testdatengenerierung. Daher behandelt die vorliegende Arbeit den Strukturtest als stellvertretenden Anwendungsfall für das entwickelte Automatisierungsverfahren. Dennoch empfehlen sich für die Zukunft Studien zur Einsetzbarkeit des Verfahrens zu Funktionstests.

2.2.3 ENTWICKLUNGSBEGLEITENDER TEST

Der Test eines SL/TL-Modells, sei es in funktionaler oder in strukturorientierter Ausrichtung, ist zwar eine wichtige qualitätssichernde Maßnahme – allerdings ist es nicht die einzig Verfügbare. Neben der Durchführung von statischen Analysen oder der Anwendung von Review-Techniken sind es vor allem entwicklungsbegleitende Tests, welche für eine korrekte Funktionalität eines Modells sorgen. Arbeitet ein Entwickler an einem Modell, so prüft er üblicherweise wichtige veränderte oder neu hinzugekommene Bestandteile direkt durch Simulationen des Modells in dem SL-Werkzeug. Die hierzu verwendeten Eingangsdaten lassen sich direkt im Modell durch verschiedene Blocktypen definieren, zum Beispiel unter Verwendung des *Signal-Builder*-Blocks oder des Constant-Blocks. Handelt es sich allerdings um die Implementierung einer Teilfunktionalität innerhalb eines großen Modells, so ist es auch für einen Entwickler manchmal nicht offensichtlich, wie die Eingangsdaten für das Modell auszusehen haben, damit das Modell einen gewünschten Zustand einnimmt. In diesem Fall stellt die automatisierte Testdatengenerierung eine sinnvolle Hilfestellung dar. Das Testdatengenerierungsproblem, also die Art der Zielstellung, ist dabei aus Sicht eines Automatisierungsverfahrens äquivalent zu dem des Funktionstests (siehe voriger Abschnitt).

2.2.4 BACK-TO-BACK-TEST

In einem Back-to-Back-Test werden klassischerweise zwei Versionen einer Programmierung gegeneinander getestet [124]. Im Rahmen einer modellbasierten Entwicklung werden Back-to-Back-Tests eingesetzt, um die funktionale Äquivalenz von Modell und Programmcode zu überprüfen. Diese Überprüfung erfolgt in der Regel durch einen Vergleich der von beiden Testobjekten bei ihrer Ausführung mit denselben Testdaten ausgegebenen Werten oder Signalen [104].

Wird eine Testsuite benötigt, welche zur Ausführung von Modell und Code verwendet wird, so bietet sich eine automatisierte Testdatengenerierung an. Beispielsweise könnte eine strukturorientiert generierte Testsuite, das heißt eine Menge von Testdaten mit hoher Modellüberdeckung, für einen Back-to-Back-Test herangezogen werden. So gesehen stellt diese Art des Testens keine zusätzliche Anforderung an das in dieser Arbeit behandelte Testverfahren – es handelt sich lediglich um einen weiteren Anwendungsfall für die durch einen automatisierten Strukturtest erzeugten Testdaten.

2.3 AUTOMATISIERUNGSANSÄTZE

Die vorigen Abschnitte dieses Kapitels behandelten zum einen Softwaretest- sowie Praxis-Grundlagen und steckten zum anderen den in dieser Arbeit betrachteten Anwendungsfall ab: Es geht um die Generierung von Testdaten für ein SL/TL-Modell, wobei das Hauptaugenmerk dabei dem Modell-Strukturtest gilt. Die verbleibenden Abschnitte dieses Kapitels widmen sich nun den für eine Testdatengenerierung existierenden Ansätzen und Techniken. Auch hierbei erfolgt eine Fokussierung – und zwar auf den suchbasierten Ansatz. Dieser wird daher im Anschluss an den folgenden Abschnitt ausführlich behandelt.

Die verschiedenen in der Literatur auffindbaren Techniken zur Testdatengenerierung lassen sich grundsätzlich in drei Kategorien einteilen: statische, dynamische und hybride Techniken. Wie im einführenden Kapitel beschrieben, führen dynamische Techniken das Testobjekt aus, wohingegen statische Techniken das Testobjekt entweder nur analysieren oder zumindest nur symbolisch ausführen. Hybride Techniken führen das Testobjekt unter Umständen ebenfalls aus, beinhalten allerdings auch statisch arbeitende Komponenten. Im Folgenden werden Ansätze aus allen drei Kategorien vorgestellt. Dabei handelt es sich primär um Ansätze in Anwendung für SL-Modelle, allerdings zum Teil auch um Arbeiten, welche die Testdatengenerierung für Programmcode behandeln. Einen im Vergleich hierzu deutlich allgemeineren Einblick in die verschiedenen Methodiken zur Testdatengenerierung stellen Anand et al. [3] zur Verfügung.

2.3.1 STATISCHE TESTDATENGENERIERUNG

Im Bereich der statischen Testdatengenerierungsverfahren haben sich, sowohl in der Anwendung für Programmcode als auch für SL/TL-Modelle,

zwei grundlegende Techniken hervorgetan: die symbolische Ausführung und das Model Checking.

SYMBOLISCHE AUSFÜHRUNG

Anders als der Name der symbolischen Ausführung [72] suggeriert, wird ein Testobjekt durch das Verfahren nicht im eigentlichen Sinne ausgeführt. Allerdings erfolgt eine statische Analyse des Testobjekts, welche einer abstrakten Ausführung gleichkommt. Diese Ausführung wird als symbolisch bezeichnet, weil die Eingangswerte und internen Werte des Testobjekts durch symbolische Werte repräsentiert werden. Im Falle der Anwendung für Programmcode wird durch die symbolische Ausführung eine Baumstruktur aufgebaut, welche die Ausführungslogik des Programms abbildet. Verzweigungen im Code führen dabei zu Verzweigungen im Ausführungsbaum. Jeder Knoten im Baum referenziert eine Zeile im Code und enthält den Zustand der Programm-Variablen als symbolischen Ausdruck. Ein solcher Ausdruck bildet beispielsweise die Berechnungsschritte innerhalb eines Programms ab. Des Weiteren beinhaltet jeder Knoten eine sogenannte Pfadbedingung (englisch: path constraint) – eine aussagenlogische Formel – welche alle Bedingungen an die Eingänge des Programms beschreibt, die durch Testdaten erfüllt sein müssen, um zu dieser Stelle im Code zu gelangen. Ist die Formel einer Pfadbedingung unerfüllbar, das heißt existiert keine Variablenbelegung, welche die Formel wahr macht, so ist die referenzierte Zeile im Code nicht erreichbar. Die Prüfung der Unerfüllbarkeit und die Lösung der Pfadbedingungen, also das Finden von erfüllenden Testdaten, erfolgt in der Regel mithilfe von *Constraint Solvern* oder verwandten Techniken und Werkzeugen.

Ein Großteil der Arbeiten zur symbolischen Ausführung beschäftigt sich mit der Anwendung der symbolischen Ausführung zur strukturorientierten Testdatengenerierung für verschiedenste Programmiersprachen. Anand et al. [3] bieten hierzu eine umfangreiche Übersicht an. Dabei stellen Anand et al. allerdings auch fest, dass die bisherigen Arbeiten die drei großen Probleme der symbolischen Ausführung nur zum Teil lösen oder adressieren. Ein Problem besteht in der sogenannten Pfadexplosion (englisch: path explosion). Mit diesem Begriff wird eine ausufernde Anzahl möglicher Programmpfade bezeichnet. Sowohl eine symbolische Ausführung als auch die Lösung von Pfadbedingungen gestaltet sich hierdurch sehr schwierig. Dies hat zur Folge, dass eine Testdatengenerierung mittels symbolischer Ausführung für viele reale, in der Praxiswelt vorkommende Programme nicht in angemessener Zeit bewerkstelligt werden kann. Ein weiteres Problem besteht in der Divergenz innerhalb vieler Programme. Oftmals sind Teile des Codes in anderen Sprachen implementiert, ausgelagert oder stehen nur in binärer Form zur Verfügung. Dies erschwert oder verhindert eine vollständige Ableitung der

Pfadbedingungen. Problematisch sind zudem Pfadbedingungen, in denen nicht-lineare Operationen, wie Division und Multiplikation, oder andere mathematische Funktionen, beispielsweise die Sinus-Funktion, enthalten sind.

MODEL CHECKING

Während die symbolische Ausführung ein Verfahren ist, welches primär zum Zwecke der Testdatengenerierung geschaffen wurde, hat das Model Checking einen anders gelagerten Hintergrund. Model Checking wird normalerweise dazu verwendet, um zu verifizieren, dass ein Modell oder Programm seine (formale) Spezifikation einhält. Entsprechende Werkzeuge werden als *Model Checker* bezeichnet. Deren grundlegende Funktionsweise haben Fraser und Wotawa [41] gut zusammengefasst: Ein Model Checker arbeitet üblicherweise mit einem Zustandsautomaten, welcher das Programm abbildet, und hat die Aufgabe, die Einhaltung oder Nicht-Einhaltung einer durch temporale Logiken spezifizierten Eigenschaft durch den Automaten zu prüfen. Hierzu untersucht der Model Checker den gesamten Zustandsraum des Automaten. Falls er eine Verletzung der Eigenschaft ausfindig machen kann, liefert er ein Gegenbeispiel. Dieses besteht bei den meisten aktuellen Model Checkern aus einer linearen Sequenz von Zuständen, die der Automat nacheinander auf dem Weg zur Eigenschaftsverletzung einnehmen würde.

Wie aber lässt sich dieses Verfahren zur Testdatengenerierung einsetzen? Um dies zu erreichen, wird die Fähigkeit eines Model Checkers zur Erzeugung von Gegenbeispielen ausgenutzt und zur Erzeugung von Testdaten uminterpretiert [41]. Bezogen auf einen Strukturtest werden im Testobjekt-Automaten sogenannte *Trap-Variablen* eingebaut, welche jeweils die Erfüllung eines Überdeckungsziels durch Einnahme eines bestimmten Zustands oder durch Begehen einer bestimmten Transition im Automaten anzeigen. Entsprechend ihrer deutschen Übersetzung bilden diese Trap-Variablen somit Fallen, in die der Model Checker tappen soll. Die durch den Model Checker je Überdeckungsziel zu prüfende Eigenschaft verkörpert nämlich die Behauptung, dass das Überdeckungsziel nicht erreichbar sei. Ist es doch erreichbar, so liefert der Model Checker ein Gegenbeispiel, aus dem sich die gesuchten Testdaten ableiten lassen. Fraser und Wotawa [41] haben sich in einer Publikation intensiv mit der Anwendung von Model Checkern zum Zwecke der Testdatengenerierung auseinandergesetzt. Hierbei führen sie eine ganze Reihe von Defiziten auf, welche dieser Ansatz mit sich bringt – insbesondere im Vergleich mit Verfahren, welche gezielt für den Zweck der Testdatengenerierung entwickelt wurden. Hierbei kommt vor allem das Problem der sogenannten Zustandsexplosion zur Sprache. Ähnlich der Pfadexplosion bei der symbolischen Ausführung bezeichnet diese eine ausufernde Größe des Zustandsraums, den ein Model Checker erschöpfend untersucht. Eine

Abstraktion innerhalb des Zustandsautomaten ist ein Weg, um dieses Problem zu lindern. Allerdings, so stellen Fraser und Wotawa fest, eignen sich die existierenden Abstrahierungskonzepte nicht, wenn Model Checking zur Testdatengenerierung verwendet wird.

ANWENDUNG FÜR SIMULINK-MODELLE

Sowohl die symbolische Ausführung als das Model Checking wurden zur Testdatengenerierung für SL-Modelle oder SF-Automaten angewandt.

Hamon et al. [54, 56] setzen auf den zuvor beschriebenen Model-Checking-Ansatz, um neben der Durchführung eines klassischen Model Checkings auch strukturorientiert Testdaten für SL-Modelle und SF-Automaten generieren zu können. Hierbei erfolgt eine Übersetzung der Modelle in eine für den verwendeten Model Checker verständliche Sprache (SAL). Die Autoren merken an, dass ihr Ansatz Einschränkungen bezüglich der Unterstützung von verschiedenen SL-Blocktypen macht. Beispielsweise werden Blöcke mit nicht-linearen Operationen oder trigonometrischen Funktionen nicht unterstützt. Der Ansatz wurde in der Anwendung zur Testdatengenerierung für SF-Automaten mit Erfolg evaluiert, SL-Blockdiagramme wurden in den vorgestellten Fallbeispielen allerdings nicht betrachtet.

Gadkari et al. verfolgen gemeinsam mit Mohalik et al. einen sehr ähnlichen Ansatz [43, 44, 90]. Ihr mit *AutoMOTGen* bezeichneter Ansatz setzt ebenfalls sowohl auf Model-Checking-Techniken als auch auf eine Transformation des Modells in die Sprache SAL. Die Autoren berücksichtigen allerdings nur eine stark eingegrenzte Untermenge der verfügbaren SL-Blocktypen in ihrer Umsetzung. Zwei SL-Modelle aus dem Automobilbereich wurden in Fallstudien zur Evaluierung herangezogen und die Ergebnisse mit denen eines kommerziellen Werkzeugs, welches auf eine zufallsbasierte Testdatengenerierung setzt, verglichen. Der Model-Checking-Ansatz führte zwar zu vielversprechenden Ergebnissen, das heißt einer ähnlich hohen Modellüberdeckung wie das kommerzielle Tool, allerdings sahen sich die Autoren auch mit den technischen Grenzen ihres Ansatzes konfrontiert – vor allem bezüglich der Skalierbarkeit.

Weitere Arbeiten, in denen Model Checking zur Testdatengenerierung für SL-Modelle zur Anwendung kommt, stammen von He et al. [64] und von Brillout et al. [21]. Diese beschäftigen sich allerdings mit der Generierung von Tests im Rahmen von sogenannten Mutationstests und seien daher an dieser Stelle nur der Vollständigkeit halber genannt.

Die symbolische Ausführung wird in Arbeiten von Roy und Shankar [101, 102] sowie Păsăreanu et al. [96] im Kontext von SL betrachtet. Roy und Shankar verwenden SMT-Solving, um Testdaten für SL-Modelle zu erzeugen. Da ein Teil der vorliegenden Arbeit einen ähnlichen Ansatz

verfolgt, sei hier darauf verwiesen, dass das Thema SMT-Solving an späterer Stelle eingehend behandelt wird. Die Arbeit von Roy und Shankar unterscheidet sich allerdings grundlegend bezüglich dreier Aspekte: Erstens liegt das Hauptaugenmerk der Autoren nicht in der strukturorientierten Testdatengenerierung, sondern in der Generierung von Testdaten, die sogenannte *Contracts*, welche unter Berücksichtigung der Signaldatentypen zu gelten haben, verletzen. Mit dem Begriff Contracts bezeichnen die Autoren Aussagen über Beziehungen zwischen Signalen. Zweitens erörtert die Arbeit von Roy und Shankar nicht die Grenzen und die Skalierbarkeit des vorgestellten Ansatzes. Drittens verwenden die Autoren ausschließlich SMT-Solving zur Testdatengenerierung. In der vorliegenden Arbeit ist eine ähnliche Technik aber Teil eines hybriden Ansatzes.

Păsăreanu et al. generieren Testdaten für SL ebenfalls mittels symbolischer Ausführung. Allerdings übersetzen sie ein Modell zuerst in eine mit der Programmiersprache Java beschriebenen Repräsentation und nutzen dann ihr Werkzeug *Symbolic PathFinder* zur Testdatengenerierung. Dieses Werkzeug führt die symbolische Ausführung auf Java-Byte-Code-Ebene durch und verwendet Constraint Solver, um Testdaten aus den Pfadbedingungen abzuleiten. Unklar ist, in welchem Zusammenhang die hierdurch erzielte Codeüberdeckung mit der in der vorliegenden Arbeit betrachteten Modellüberdeckung steht. Die Autoren berichten des Weiteren von Skalierungsproblemen ihres Ansatzes.

Zusammenfassend bleibt festzustellen, dass die Unterstützung aller in SL/TL-Modellen relevanten Funktionalitäten und die Skalierbarkeit mit steigender Modellgröße entscheidende Hürden für eine rein statische Testdatengenerierung darstellen. Hinzu kommt, wie Kanade et al. [70] treffend feststellen, dass das Fehlen einer klar spezifizierten Semantik für SL (siehe Abschnitt 2.1.2) die Konstruktion formaler Werkzeuge erschwert. In der vorliegenden Arbeit werden dennoch statische Techniken behandelt und eingesetzt. Grund hierfür ist, dass statische Techniken auch ihre Vorteile haben. So können sie allgemeingültige Aussagen treffen und arbeiten oftmals schneller als entsprechende dynamische Techniken. Allerdings sind die statischen Techniken dieser Arbeit stets als Ergänzung einer dynamischen Testdatengenerierung anzusehen. Das dynamische Verfahren funktioniert auch dann, wenn eine der statischen Techniken an ihre Grenzen stößt. Ergänzend ist zu erwähnen, dass keine der aufgeführten Arbeiten die Plausibilität der generierten Testdaten berücksichtigt. Dies bedeutet, dass nicht sichergestellt wird, dass die generierten Testdaten – welches im Falle von SL/TL-Modellen Signale sein sollten – gewisse Anforderungen an ihre Form (wie beispielsweise die Signalcharakteristik) erfüllen. In der Praxis könnten derlei Testdaten unbrauchbar sein, weil sie keine realistischen Szenarien abbilden.

2.3.2 DYNAMISCHE TESTDATENGENERIERUNG

In der Kategorie dynamische Testdatengenerierung sind im Wesentlichen der Zufallstest und der suchbasierte Test zu nennen. Diese beiden Ansätze werden im Folgenden behandelt, insbesondere in ihrer Anwendung für SL/TL-Modelle.

ZUFALLSTEST

Die Testdatengenerierung durch verschiedenste Formen einer zufälligen Bestimmung oder Auswahl von Testdaten ist seit den frühen 1960er Jahren [99] und auch in aktuellen Arbeiten ein Thema in der Forschung. In seiner Reinform werden bei einem Zufallstest so lange zufällig Testdaten generiert und das Testobjekt mit diesen ausgeführt, bis die gewünschten Testdaten, beispielsweise zum Erreichen aller Überdeckungsziele eines Strukturtests, gefunden wurden – oder die zufällige Generierung anderweitig abgebrochen wird. Die Testdaten werden dabei üblicherweise mit einer einheitlichen Wahrscheinlichkeit p aus dem Eingabedatenraum D des Testobjekts zufällig gewählt, das heißt für jedes Testdatum gilt: $p = 1/|D|$ [4].

Selbst kommerzielle Werkzeuge zur Testdatengenerierung setzen, wie in Abschnitt 2.3.4 ersichtlich wird, auf diesen simplen Ansatz. Die Gründe hierfür sind naheliegend, wie Chen et al. [24] feststellen: Der Zufallstest ist leicht zu verstehen, kann in der Regel einfach implementiert werden, hat in vielen Arbeiten unter Beweis gestellt, dass er Fehler aufdecken kann, und ist möglicherweise gerade deswegen so beliebt, weil er das Testobjekt auch mit eher ungewöhnlichen Szenarien konfrontiert – auf die ein Tester gar nicht erst kommen würde. Arcuri et al. stellen fest, dass der Zufallstest in vielen Anwendungsfällen ein brauchbares Mittel zur Testdatengenerierung darstellt [6]. Insbesondere dann, wenn andere Testdatengenerierungstechniken nicht ausreichend gut skalieren, hat der Zufallstest Vorteile [5]. Daher wird der Zufallstest auch gerne in hybriden Ansätzen zur Testdatengenerierung verwendet, wie der folgende Abschnitt 2.3.3 zeigt. Es zeigt sich allerdings häufig, dass ein reiner Zufallstest ineffizient ist, da keinerlei Versuch unternommen wird, die Generierung der Testdaten in irgendeiner Art und Weise mithilfe von Informationen über das Testobjekt zu steuern [24]. Daher wird der Zufallstest auch gerne zum Vergleich herangezogen, wenn es gilt, ein anderes Testdatengenerierungsverfahren zu untersuchen.

In den letzten Jahren hat sich eine Reihe von Arbeiten mit einer Variation des Zufallstests beschäftigt: dem sogenannten adaptiven Zufallstest (ART, englisch: adaptive random testing) [24]. Die grundlegende Idee des ART besteht darin, eine stärkere Verteilung der generierten Testdaten im Eingabedatenraum anzustreben. Befinden sich generierte Testdaten in

ähnlichen Regionen des Eingabedatenraums und sind diese Testdaten nicht die Gesuchten, so wird bei der Erzeugung weiterer Testdaten darauf geachtet, dass diese im Eingabedatenraum weit von dieser Region entfernt sind. Wie weit, hängt von dem betrachteten Anwendungsfall und dem Testobjekt ab [4]. In einer groß angelegten Studie [4] stellen Arcuri und Briand allerdings fest, dass ART nur geringfügig effizienter und effektiver als ein klassischer Zufallstest ist. Die Autoren zeigen sich angesichts der hohen Anzahl an Arbeiten bezüglich ART sehr überrascht ob der ernüchternden Ergebnisse.

SUCHBASIERTER TEST

Die Anfänge des suchbasierten Tests reichen bis in die 1970er Jahre zurück. Im Jahre 1976 hatten Miller und Spooner [89] die Idee, Testdaten zu generieren, die einen bestimmten Programmpfad zur Ausführung bringen, indem erzeugte Testdaten innerhalb eines simplen Optimierungsprozesses mithilfe einer Kostenfunktion bewertet werden. Testdaten, welche der Ausführung des gewünschten Pfades näher kamen, bekamen durch die Kostenfunktion (im Folgenden als Fitnessfunktion bezeichnet) einen niedrigen Wert zugewiesen und dienten der Optimierung zur Erzeugung weiterer Testdaten. Testdaten mit hohen „Kosten“ wurden hingegen verworfen. Erst Anfang der 1990er Jahre wurde dieser Ansatz von Korel [76] wieder aufgegriffen und erfährt seit Mitte der 1990er Jahre eine steigende Beachtung in der Forschung [83].

Heute wie damals ist der suchbasierte Test durch den Einsatz von metaheuristischen Algorithmen gekennzeichnet [59, 83]. Dies sind Optimierungsalgorithmen, welche aus abstrakten Schritten bestehen und somit grundsätzlich für jedes beliebige Optimierungsproblem anwendbar sind. Kern der meisten metaheuristischen Algorithmen ist die Bewertung generierter Lösungsvorschläge durch eine Fitnessfunktion. Diese hat die Aufgabe, gute Lösungsvorschläge von schlechten Lösungsvorschlägen zahlenmäßig zu unterscheiden und leitet die Suche somit im Idealfall zum gewünschten Ziel. Die verschiedenen existierenden Algorithmen, wie beispielsweise der Bergsteigeralgorithmus oder der genetische Algorithmus, unterscheiden sich im Wesentlichen in der Art und Weise, wie sie von bereits generierten Lösungsvorschlägen zu neuen Lösungsvorschlägen gelangen. Im Kontext des Tests stellen die Lösungsvorschläge die Testdaten dar. Als Strukturtest interpretiert, sucht ein metaheuristischer Algorithmus nach Testdaten, die eines oder mehrere Überdeckungsziele erfüllen [120–122, 126, 127]. Die jeweilige Fitnessfunktion lässt sich dabei automatisch aus der inneren Struktur des Testobjekts ableiten [126].

ANWENDUNG FÜR SIMULINK-MODELLE

Die Zahl der Arbeiten, die sich mit einer Anwendung des Zufallstests oder des suchbasierten Tests für SL-Modelle beschäftigen, ist noch relativ überschaubar. Die existierenden Arbeiten zum suchbasierten Test behandelt dennoch nicht nur den Strukturtest, sondern auch den Funktionstest und sogar den Mutationstest. Der Zufallstest wurde, zumindest in der Forschung, zwar des Öfteren für den Modelltest angewandt, allerdings primär zu Vergleichszwecken. Das Konzept des adaptiven Zufallstests ist noch nicht in der SL-Welt angewandt worden.

Im Bereich des Funktionstests verwenden Boström und Björkqvist [20] sowie Vos et al. [123] einen suchbasierten Ansatz, um Spezifikationsverletzungen innerhalb von SL-Modellen ausfindig zu machen. Während Boström und Björkqvist die durch das Modell einzuhaltenden Anforderungen mittels sogenannter Assertions ausdrücken und aus diesen jeweils automatisch die benötigte Fitnessfunktion ableiten, setzen Vos et al. auf eine direkte Definition der Fitnessfunktion. Der Ansatz von Boström und Björkqvist ist im Übrigen eng verwandt mit der Interpretation des Funktionstests in der vorliegenden Arbeit. Im Unterschied zu dem Ansatz der Beiden schlägt diese Arbeit allerdings, wie in Abschnitt 2.2.2 beschrieben, eine Formulierung von Assertions mit den Sprachmitteln von SL/TL vor.

Windisch [132] sowie Zhan und Clark [135, 137] wenden den suchbasierten Test zur strukturorientierten Testdatengenerierung für SL-Modelle an. Während Zhan und Clark hierbei lediglich die grundsätzliche Anwendbarkeit behandeln und sich ansonsten auf den Einsatz zu Mutationstests konzentrieren, widmet sich die Arbeit von Windisch primär der Anwendbarkeit in der Praxis. Hierzu entwickelte er gemeinsam mit Al Moubayed eine Abstraktionstechnik, welche die Generierung plausibler Signale ermöglicht [133]. Ebenfalls im Sinne einer Plausibilität der erzeugten Testdaten ist eine ergänzende Arbeit von Wilmes und Windisch zu sehen [130]. Diese behandelt die Berücksichtigung zusätzlicher Bedingungen innerhalb der Testdatensuche. Dabei handelt es sich um temporal-logisch spezifizierte Bedingungen, welche für die zu generierenden Signale zu gelten haben. Des Weiteren berücksichtigt der Ansatz von Windisch auch die strukturorientierte Testdatengenerierung für SF-Automaten [131]. In diesem Zusammenhang stellen Oh et al. im Übrigen einen speziellen genetischen Algorithmus vor, der die Transitionsüberdeckung von SF-Automaten adressiert [94]. Zu guter Letzt seien Ghani und Clark genannt, welche die Arbeit von Zhan und Clark zum suchbasierten Mutationstest um dynamische Fitnessfunktionen ergänzen [47]. Hierbei wird die Zieldefinition einer Suche bewusst manipuliert, um der Suche die Annäherung an das eigentliche Suchziel zu erleichtern. Zhan und Clark erweitern ihren Ansatz des Weiteren selbst um eine statische Analyse des betrachteten SL-Modells zur Bestimmung einer geeigneten Länge der

Signale, welche die Testdaten darstellen und die es demnach zu erzeugen gilt [136].

Sowohl Windisch als auch Zhan und Clark vergleichen ihre Ansätze mit dem Zufallstest [132, 135, 137]. Hierbei liefert der suchbasierte Ansatz durchweg bessere Ergebnisse. Besser bedeutet in diesem Zusammenhang, dass der suchbasierte Strukturtest in Anwendung für SL-Modelle eine höhere Modellüberdeckung mit geringerem Aufwand erzielt. Der Aufwand bemisst sich dabei in der Anzahl generierter Testdaten und Modellausführungen. Diese Beobachtungen bestätigen Ergebnisse, die im Bereich des suchbasierten Strukturtests von Programmcode erzielt wurden – beispielsweise durch Wegener et al. [127].

Windisch evaluierte seinen Ansatz anhand von realen Modellen aus der automobilen Entwicklung. Stellt man den suchbasierten Test in diesem Zusammenhang den statischen Ansätzen gegenüber, so ist unabhängig von der Anwendbarkeit rein statisch operierender Testdatengenerierungsverfahren festzustellen, dass der suchbasierte Ansatz zwar keinerlei Einschränkungen hinsichtlich Blockunterstützung oder Modellgröße hinnehmen muss – allerdings ist die Effizienz des Verfahrens noch verbesserungswürdig.

2.3.3 HYBRIDE TESTDATENGENERIERUNG

Die Beiträge der vorliegenden Arbeit sind nicht die Ersten, welche der Grundidee der Kombination verschiedener Techniken folgen. So existiert eine Vielzahl an Arbeiten, die sich mit einer Verzahnung von symbolischer Ausführung und Zufallstest auseinandersetzen. Zudem gibt es erste Impulse, die sich um eine Nutzung suchbasierter Techniken innerhalb dieser Kombination bemühen. Derlei Arbeiten und in diesem Bereich unternommene Anwendungsversuche zur Testdatengenerierung für SL werden in diesem Abschnitt behandelt.

HYBRIDE ANSÄTZE

Die erwähnte Kombination von symbolischer Ausführung und Zufallstest zum Zwecke einer strukturorientierten Testdatengenerierung hat ihre Ursprünge in zwei parallel zueinander entstandenen Arbeiten. Während Godefroid et al. [49] die Technik als dynamische symbolische Ausführung und ihr dazu entstandenes Werkzeug mit dem Namen *DART* bezeichnen, nennen Sen et al. [109] die Technik Concolic Testing und ihr Werkzeug *CUTE*. Der Begriff Concolic beschreibt hierbei, dass es in dem Verfahren sowohl zu einer konkreten (englisch: concrete) als auch zu einer symbolischen (englisch: symbolic) Ausführung des Testobjekts kommt. Beide

Begriffe haben sich in der Literatur gehalten, in den folgenden Ausführungen verwendet diese Arbeit jedoch den Begriff Concolic Testing.

Die grundlegende Idee beider Ansätze besteht in der tatsächlichen Ausführung eines Testobjekts (in diesem Fall C-Code), falls eine Testdatengenerierung durch Constraint Solving bei der symbolischen Ausführung nicht möglich ist. In Abschnitt 2.3.1 wurde erklärt, dass die Bildung von Pfadbedingungen oder das Ableiten von Testdaten aus diesen in verschiedenen Situationen entweder nicht möglich oder nur schwer zu bewerkstelligen ist. Dies ist beispielsweise der Fall, wenn der zur Ableitung der Testdaten verwendete Constraint Solver mit einem Ausdruck in der Pfadbedingung nicht umgehen kann. In einem solchen Fall wird das Testobjekt beim Concolic Testing mit zufällig generierten Testdaten ausgeführt. Anschließend werden Variablen des problematischen Ausdrucks in der Pfadbedingung zum Teil mit Werten ersetzt, die bei der realen Ausführung entstanden sind. Handelt es sich zum Beispiel um die Multiplikation zweier Variablen, so würde ein Multiplikand durch einen Wert ersetzt werden. Der verwendete Constraint Solver wäre nun in der Lage, auch mit diesem Ausdruck umzugehen. Da die symbolische Ausführung durch das Einsetzen konkreter Werte in die Pfadbedingungen allerdings punktuell ihren verallgemeinernden Charakter verliert, beinhaltet das Concolic Testing die Möglichkeit eines *Backtrackings*. Für eine detaillierte Einführung dieser Technik sei auf die Arbeiten von Godefroid et al. [49] und Sen et al. [109] verwiesen.

Lakhotia et al. vergleicht in einer umfangreichen Fallstudie den suchbasierten Test und das Concolic Testing [77, 78]. Hierbei betrachtet er die strukturorientierte Testdatengenerierung für C-Code mit dem Werkzeug CUTE und einem eigenen suchbasierten Testwerkzeug namens *AUSTIN*. Zusammenfassend betrachtet bringt die Studie keinen eindeutigen Sieger hervor. Beide Verfahren oder Werkzeuge hatten zum Zeitpunkt der Durchführung der Studie mit technischen Problemen im Umgang mit den verwendeten Testobjekten zu kämpfen. Im Schnitt arbeitete CUTE zwar etwas effizienter als AUSTIN, bezüglich des erreichten Überdeckungsgrads gab es jedoch gleichermaßen Fälle, in denen das Concolic Testing oder der suchbasierte Test bessere Ergebnisse erzielten.

In einer weiteren Arbeit ergänzen Lakhotia et al. das Testwerkzeug *Pex* [100] von Microsoft um eine suchbasierte Testdatengenerierung [79]. Pex ist eine Erweiterung der Entwicklungsumgebung *Visual Studio* und kann zur strukturorientierten Testdatengenerierung für *.NET*-Programme eingesetzt werden. Pex verwendet hierzu das Concolic Testing. Lakhotia et al. betrachten das Problem von Fließkommazahl-Berechnungen in einer Pfadbedingung. Diese sind für den in Pex verwendeten Constraint Solver problematisch. Kommt in diesem Spezialfall statt Constraint Solving eine metaheuristische Suche zum Einsatz, so kann die Effektivität der Testdatengenerierung deutlich gesteigert werden, wie die Ergebnisse

einer dazugehörigen Fallstudie zeigen. Diesem Vorteil stehen allerdings deutlich längere Laufzeiten gegenüber.

Inkumsah und Xie befassen sich ebenfalls mit einer Kombination des suchbasierten Tests und des Concolic Testing [68]. Ihr Framework *EVACON* dient dabei der strukturorientierten Testdatengenerierung für mit Java geschriebenen objektorientierten Code. EVACON greift auf zwei andere Werkzeuge zurück: das suchbasierte Testwerkzeug *eToc* [118] und *jCUTE* [108], eine Implementierung von CUTE für Java-Code. Während EVACON das suchbasierte Werkzeug einsetzt, um Sequenzen von Methodenaufrufen zu generieren, wird jCUTE genutzt, um in Ergänzung hierzu Werte für die Methodenparameter zu ermitteln.

ANWENDUNG FÜR SIMULINK-MODELLE

Zwei Ansätze einer hybriden Testdatengenerierung für SL-Modelle lassen sich in der Literatur aktuell finden.

Satpathy et al. übertragen die Idee des Concolic Testing auf die Modellebene [105]. Ihr Verfahren namens *REDIRECT* nutzt hierbei einen SMT-Solver zur Lösung der Pfadbedingungen. Zusätzlich wendet REDIRECT eine Reihe von Heuristiken an, um mit nicht-linearen Blöcken umgehen zu können. Allerdings fokussieren sich die Autoren in ihrer Veröffentlichung auf SF-Automaten und betrachten den Blockdiagramm-Anteil eines SL-Modells nur am Rande. In Fallbeispielen generiert REDIRECT Testdaten mit einer höheren Transitions- und Zustandsüberdeckung als ein kommerzielles Werkzeug. Bei den Anwendungsbeispielen handelt es sich jedoch um relativ kleine SF-Automaten. Darüber hinaus beinhaltet der Ansatz keinerlei Konzepte zur Gewährleistung einer Plausibilität der generierten Testdaten.

Peranandam et al. [95] stellen einen Ansatz vor, der ähnlich wie ein Beitrag der vorliegenden Arbeit ausgerichtet ist. In ihrem mit dem Namen *SmartTestGen* bezeichneten Verfahren kommen sowohl Zufallstest als auch symbolische Ausführung, Model Checking und eigene Heuristiken zur Testdatengenerierung zum Einsatz. Die verschiedenen Techniken operieren dabei nicht ineinander verzahnt, sondern es erfolgt je Überdeckungsziel eine statische Auswahl der mutmaßlich erfolgversprechendsten Technik. Über die Auswahlkriterien oder -regeln verraten die Autoren allerdings nicht viel, es ist von Regeln die Rede, die auf Erfahrungswerten basieren. Fallstudien belegen, dass der SmartTestGen-Ansatz im Schnitt eine höhere Modellüberdeckung als ein kommerzielles Werkzeug erzielt. Die Autoren schließen daraus, dass der Fall-spezifische Einsatz verschiedener Testdatengenerierungsverfahren ein geeigneter Ansatz ist, um der Vielfältigkeit und Komplexität in industriellen SL-Modellen zu begegnen. Die Plausibilität der Testdaten ist in ihrer Arbeit ebenfalls kein Thema.

Wie zu sehen ist, setzen existierende hybride Ansätze vornehmlich auf den Zufallstest als dynamische Komponente, um Limitierungen verwendeter statischer Techniken zu überwinden. Die vorliegende Arbeit vertritt in diesem Zusammenhang eine umgekehrte Sichtweise: Sie verwendet statische Techniken in erster Linie, um Schwächen der dynamischen Komponente zu kompensieren. Vergleicht man des Weiteren die Funktionsweise eines Zufallstest mit der eines suchbasierten Tests, so ist festzustellen, dass der suchbasierte Ansatz eine verbesserte Form des Zufallstests darstellt. Auch beim suchbasierten Test spielt der Zufall in der Erzeugung der Testdaten eine entscheidende Rolle. Die Erzeugung erfolgt jedoch nicht vollständig zufällig, sondern erfährt durch die Bewertung der Testdaten eine Führung zum betrachteten Ziel. Dies ist ein Grund, wieso die vorliegende Arbeit im Bemühen um eine Hybridisierung den suchbasierten Ansatz aufgreift. Hinzu kommt, dass dieser Ansatz bereits sehr stark an die Bedürfnisse der industriellen Praxis angepasst ist – unter anderem durch die Arbeiten von Windisch [130–133].

2.3.4 KOMMERZIELLE WERKZEUGE

Neben den in der Literatur dokumentierten Arbeiten darf nicht vergessen werden, dass es auch kommerzielle Werkzeuge gibt, die sich zur strukturorientierten Testdatengenerierung für SL/TL-Modelle einsetzen lassen. Die Mehrheit dieser Werkzeuge generiert allerdings Testdaten zur strukturellen Überdeckung des aus dem Modell generierten Codes. Modellüberdeckungskriterien, wie in Abschnitt 2.2.1 vorgestellt, betrachtet neben dem Werkzeug *Reactis* [97] der Firma *Reactive Systems* auch der *Simulink Design Verifier* [116] von ML/SL-Hersteller *The MathWorks*. Reactis wurde unter anderem von Windisch in Fallstudien zum Vergleich des eigenen Testdatengenerierungsansatzes herangezogen. Hierbei konnte der suchbasierte Ansatz von Windisch, wie zuvor bereits geschildert, einen höheren Modellüberdeckungsgrad erzielen [132].

Über die Techniken, welche Reactis und der Simulink Design Verifier zur Testdatengenerierung einsetzen, ist öffentlich nicht allzu viel bekannt. Während Reactis grundlegend auf den Zufallstest setzt und diesen um sogenannte „gesteuerte Simulationen“ ergänzt, verwendet der Simulink Design Verifier sowohl Model-Checking- als auch Constraint-Solving-Techniken.

2.4 SUCHBASIERTER ANSATZ

Aufbauend auf der Einführung des suchbasierten Tests in Abschnitt 2.3.2 vertieft dieser Abschnitt das dort geschilderte Grundprinzip. Darüber hinaus wird der Ansatz zum suchbasierten Strukturtest für SL von Windisch etwas detaillierter als zuvor vorgestellt, da die Beiträge der vorliegenden Arbeit hierauf aufsetzen.

2.4.1 LOKALE, GLOBALE UND HYBRIDE SUCHE

Zur Durchführung eines suchbasierten Tests kann grundsätzlich ein beliebiger metaheuristischer Optimierungsalgorithmus zum Einsatz kommen. Im Folgenden werden solche Algorithmen auch als Suchverfahren oder schlicht als Suche bezeichnet. Die vielfältige Welt der Suchverfahren lässt sich in drei Kategorien unterteilen: lokale, globale und hybride Suchverfahren. Um diese Unterscheidung erklären zu können, empfiehlt es sich, zuerst einen Blick auf die Relevanz der Fitnessfunktion zu werfen. Eine Fitnessfunktion bewertet die von einem Suchverfahren generierten Lösungsvorschläge und hilft hierdurch wiederum dem Suchverfahren, denn das Suchverfahren kann nun ermitteln, in welche Regionen des Suchraums es als Nächstes steuert. Bildlich betrachtet spannt die Fitnessfunktion über dem Suchraum, also dem Raum aller möglichen Lösungsvorschläge, eine sogenannte Fitnesslandschaft auf. Die Fitnesslandschaft kann lokale und globale Optima enthalten. Betrachtet man dies wiederum bildlich, so stellen lokale Optima Hügel beliebiger Größe dar. Ein Optimum wird dann als global angesehen, wenn es den höchsten Hügel in der gesamten Fitnesslandschaft darstellt. Vor diesem Hintergrund definieren Harman und McMinn [59] die Klassifikation der Suchverfahren wie folgt: Eine *globale Suche* (GS) ist darauf spezialisiert, trotz möglicher Ankunft in lokalen Optima dennoch das globale Optimum im Suchraum zu finden. Eine *lokale Suche* (LS) hingegen nimmt in Kauf, in lokalen Optima zu verharren und somit nicht das globale Optimum zu finden. Eine LS navigiert zu jedem Zeitpunkt des Suchvorgangs im Suchraum üblicherweise nur von einem Punkt (Lösungsvorschlag) aus. Eine GS hingegen betrachtet zu jedem Zeitpunkt des Suchvorgangs klassischerweise eine Vielzahl an Lösungsvorschlägen [83]. Daher gelten lokale Suchverfahren im Allgemeinen als effizienter, sofern sich ein Suchproblem mit ihnen lösen lässt. Eine GS findet allerdings häufiger Anwendung, da sie im Allgemeinen besser mit verschiedenartigen Fitnesslandschaften umgehen kann und somit Problem-unabhängiger ist [59]. Hybride Suchverfahren kombinieren LS und GS.

Abbildung 5 stellt die grundlegende Funktionsweise eines suchbasierten Tests dar. Die Darstellung ist einerseits unabhängig vom tatsächlich

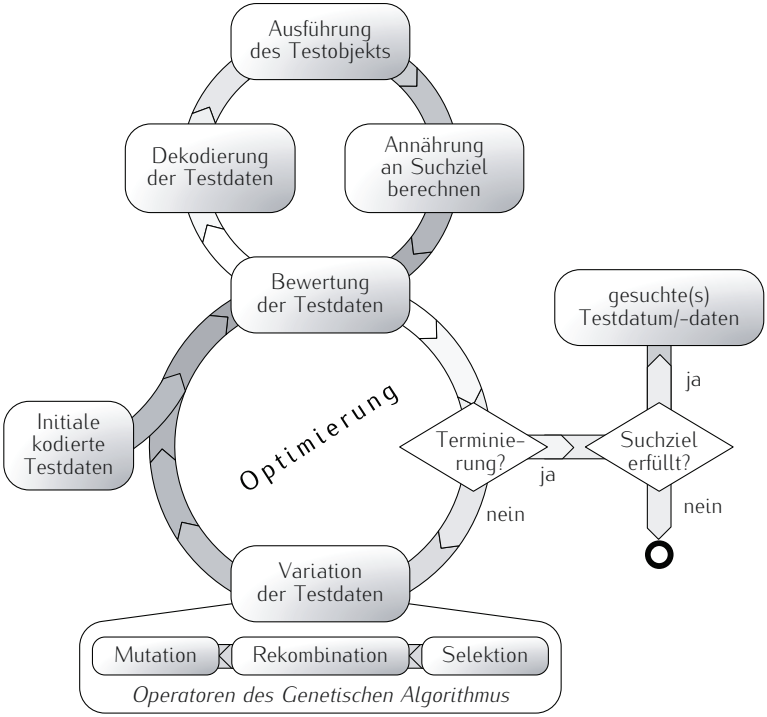


Abbildung 5: Ablauf eines suchbasierten Tests im Allgemeinen mit dem Einsatz eines genetischen Algorithmus zu Optimierungszwecken als Beispiel für eine mögliche Ausprägung (in Anlehnung an Windisch [132])

verwendeten Suchverfahren gehalten, enthält andererseits aber auch als Beispiel die speziellen Schritte eines *genetischen Algorithmus* als Beispiel eines Suchverfahrens. Ein genetischer Algorithmus realisiert eine GS. Die Anwendung eines Suchverfahrens zu Zwecken einer Testdatengenerierung macht es häufig notwendig, die Testdaten innerhalb der Optimierung (unterer Kreislauf in der Abbildung) in einer kodierten Form zu betrachten. Enthält ein Testdatum beispielsweise eine Zeichenkette, so müsste das Suchverfahren in der Lage sein, einerseits Zeichenketten zu generieren und diese andererseits gemäß seiner spezifischen Änderungsoperatoren zu variieren. Ist ein Suchverfahren allerdings nur auf den Umgang mit Zahlenwerten ausgerichtet, so bietet es sich an, eine entsprechende Kodierung für Zeichenketten zu verwenden. Natürlich ist es auch möglich, die Operatoren des Suchverfahrens für den direkten Umgang mit Zeichenketten anzupassen. Unabhängig hiervon gilt

es, eine Repräsentation eines Testdatums zu definieren, welche mit den Operatoren des Suchverfahrens kompatibel ist.

Zurück zu der Darstellung: Ein suchbasierter Test beginnt mit der, in der Regel zufälligen, Erzeugung eines initialen Testdatums oder einer Menge initialer Testdaten – je nachdem, ob eine LS oder GS verwendet wird. Anschließend erfolgt eine Bewertung der Testdaten. Bevor allerdings die Fitnessfunktion die Nähe der Testdaten zum betrachteten Ziel bemessen kann, wird das Testobjekt in der Regel mit den Testdaten ausgeführt. Vor einer jeden Ausführung werden die kodierten Testdaten dekodiert. Während oder nach einer Ausführung werden Daten, die Grundlage der Bewertung durch die Fitnessfunktion darstellen, gesammelt – wie beispielsweise die Ausgaben des Testobjekts. Liegen letzten Endes die Bewertungen vor, werden Abbruchkriterien geprüft. Diese sind von Anwendungsfall zu Anwendungsfall verschieden. So kann das Ziel der Suche darin bestehen, dass ein bestimmter Fitness-Schwellwert über- oder unterschritten wird. Des Weiteren ist häufig eine Obergrenze an Optimierungsiterationen definiert, damit eine Suche auch bei Misserfolg endet. Trifft keines der geprüften Abbruchkriterien zu, so verwenden die spezifischen Operatoren des Suchverfahrens die Fitnesswerte der Testdaten, um für die nächste Iteration der Suche neue Testdaten zu erzeugen. Ein genetischer Algorithmus wendet hierbei Prinzipien der Evolutionstheorie an: Erst werden Testdaten mit einer guten Fitness selektiert (Selektion), dann Bestandteile dieser Testdaten miteinander getauscht (Rekombination) und zusätzlich einzelne Bestandteile variiert (Mutation). Lösungsvorschläge werden im Kontext eines genetischen Algorithmus auch als Individuen bezeichnet und die Menge aller Lösungsvorschläge einer Optimierungsiteration auch Population genannt. In den Operatoren eines genetischen Algorithmus werden Entscheidungen getroffen, bei denen meist der Zufall miteinbezogen wird. Von der Gestaltung der Operatoren hängt auch wesentlich ab, ob das Suchverfahren in der Lage ist, lokale Optima in der Fitnesslandschaft zu überwinden und das globale Optimum – sofern es denn gesucht ist – zu finden. Die Gestaltung der Fitnessfunktion kann auf der anderen Seite darauf Einfluss nehmen, wie einfach oder schwer das Ziel einer Suche gefunden werden kann.

In der Kategorie der globalen Suchverfahren ist der genetische Algorithmus das mit Abstand am häufigsten verwendete Verfahren [59]. Eine gute Übersicht über existierende Suchverfahren bietet Weicker [128]. Bei den lokalen Suchverfahren wird als Beispiel gerne der sogenannte Bergsteigeralgorithmus (englisch: *Hill Climbing*, HC) angeführt. Die in den publizierten Arbeiten am häufigsten behandelte LS ist hingegen die *Alternating Variable Method* (AVM) von Korel [76]. Die AVM ist eine Variation des HC. Das Vorgehen des HC gestaltet sich sehr simpel: Gestartet wird mit einem Lösungsvorschlag. Für im Suchraum benachbarte Lösungsvorschläge wird anschließend geprüft, ob einer der Nachbarn eine bessere Fitness aufweist. Ist dies der Fall, so rückt dieser Lösungsvorschlag in

den Fokus und seine Nachbarschaft im Suchraum wird wiederum untersucht. Gibt es keinen Nachbarn mit besserer Fitness, so endet der Algorithmus. Die Nachbarschaft eines Lösungsvorschlags gilt es vorab gemäß dessen Repräsentation zu definieren [57]. Eine Nachbarschaft besteht typischerweise aus allen Lösungsvorschlägen, die von einem Lösungsvorschlag aus betrachtet mit einem möglichst kleinen Schritt im Suchraum, unter Anwendung eines Bewegungsoperators, erreichbar sind [57, 93]. Bei Anwendungsfällen mit einem vieldimensionalen Suchraum, also bei einer hohen Anzahl zu optimierender Parameter, ist die Nachbarschaft entsprechend groß. Die Nachbarschaftsdefinition gestaltet sich dann schwierig, da ansonsten das Durchsuchen einer Nachbarschaft nicht effizient geleistet werden kann [1].

Die AVM adressiert dieses Problem. Die Parameter eines Lösungsvorschlags werden bei der AVM nacheinander einzeln betrachtet. Bei einem suchbasierten Test könnten dies beispielsweise die einzelnen Eingaben des Testobjekts sein. Die Nachbarschaftsbetrachtung bezieht sich somit stets nur auf die Veränderung eines einzelnen Parameters. AVM geht dabei je Parameter wie folgt vor: Zu Beginn wird geprüft, ob sich in der durch Veränderung dieses Parameters aufgespannten Nachbarschaft ein Lösungsvorschlag mit besserer Fitness befindet. Dieser Schritt wird als Erkundungssuche bezeichnet. Gibt es einen solchen Lösungsvorschlag, so wird eine Richtungssuche durchgeführt. Hierbei erfolgen fortlaufend weitere Änderungen des betrachteten Parameters in die gleiche Richtung. Handelt es sich beispielsweise um eine ganze Zahl und führte eine Inkrementierung bei der Erkundungssuche zu einem besseren Lösungsvorschlag, so wird bei der Richtungssuche weiter inkrementiert. Sowohl Erkundungs- als auch Richtungssuche enden, sofern kein besserer Lösungsvorschlag entsteht. In diesem Fall wird zum nächsten Parameter übergegangen. Auf den letzten Parameter folgt erneut der erste Parameter. Die Suche endet insgesamt, sofern deren Ziel erreicht ist oder die Erkundungssuche aller Parameter in Folge zu keinem besseren Lösungsvorschlag führt.

Auch im Bereich der Suchverfahren existieren Ansätze einer Hybridisierung, in erster Linie um die Effizienz einer Suche zu steigern. Sogenannte memetische Algorithmen bilden hierbei den bekanntesten Ansatz [93]. Ihr Name leitet sich von dem Begriff der Memetik ab. Dieser stammt aus der Biologie und setzt sich aus dem englischen Wort Memory (zu Deutsch: Gedächtnis oder Speicher) und Genetik zusammen. Memetische Algorithmen führen grundsätzlich eine GS durch, zeichnen sich aber dadurch aus, dass sie ihre GS unterbrechen, um für ausgewählte oder alle aktuell betrachteten Lösungsvorschläge jeweils eine LS durchzuführen. Dieser Verfahrensweise liegt folgender Hintergrundgedanke zu Grunde: Befindet sich die GS innerhalb des Suchraums bereits in der Nähe des gesuchten Optimums, so ist eine LS in dieser Region des Suchraums zum einen erfolgsversprechend, da durch die Nachbarschaftsbetrachtung

der LS eine äußerst systematische Untersuchung dieser Suchraumregion stattfindet und zum anderen ist es nicht unwahrscheinlich, dass die LS diese Untersuchung effizienter als eine direkte Fortführung der GS durchführt. In der Literatur lassen sich sowohl Anwendungsbeispiele finden, in denen eine memetische Suche effizienter und effektiver als eine rein lokale oder globale Suche ist, als auch Fälle, in denen Gegenteiliges festgestellt wird [42, 59].

2.4.2 ANWENDUNG FÜR SIMULINK/TARGETLINK-MODELLE

Das im vorigen Abschnitt vertiefte Prinzip des suchbasierten Tests wendet Windisch unter Nutzung eines genetischen Algorithmus für den Strukturtest von SL-Modellen an [132]. Die in diesem Zusammenhang von ihm entwickelten Konzepte sind eins zu eins auf den Strukturtest von TL-konformen SL-Modellen übertragbar.

Abbildung 6 veranschaulicht den Ablauf der Testdatensuche im betrachteten Kontext. Gemäß der Ausführungen in Abschnitt 2.2.1 leitet sich von einem Modell entsprechend der durch den Anwender gewählten Überdeckungskriterien eine Menge von Überdeckungszielen (CGs) ab. Die CGs werden nacheinander mit Suchvorgängen abgearbeitet – in einer beliebigen Reihenfolge. Aus dem Ausdruck eines jeden CGs lässt sich dabei automatisch eine Fitnessfunktion ableiten. Windisch ermöglicht eine Spezifikation der durch die Suche zu generierenden Modelleingangssignale durch den Nutzer. Wie im Folgenden detailliert erläutert, schlägt Windisch auch eine spezielle Kodierung für Signale vor. Die kodierte Form eines Signals bezeichnet er dabei als Optimierungssequenz. Wird ein Testdatum bewertet, so werden Optimierungssequenzen dekodiert, das heißt in sogenannte Simulationssequenzen umgewandelt. Diesen Vorgang bezeichnet Windisch als Signalgenerierung. Mit den Simulationssequenzen kann das Modell ausgeführt werden. Während der Ausführung findet eine Protokollierung der resultierenden Signalverläufe innerhalb des Modells statt. Diese werden anschließend von der Fitnessfunktion des anvisierten CG zur Distanzberechnung verwendet. Darüber hinaus werden auch die Fitnessfunktionen aller noch nicht erreichten CGs ausgeführt, um zu prüfen, ob eines der CGs eventuell zufälligerweise mit-erreicht wurde. In diesem Fall ist von einer Kollateralüberdeckung (englisch: collateral coverage) die Rede. Sobald ein CG erstmalig erreicht wurde, wird das dafür verantwortliche Testdatum in die resultierende Testsuite aufgenommen.

Die angesprochene Kodierung der Testdaten ist in Abbildung 7a veranschaulicht. Da Windisch einen speziell angepassten genetischen Algorithmus zur Testdatensuche nutzt und dieser eine GS darstellt, wird je

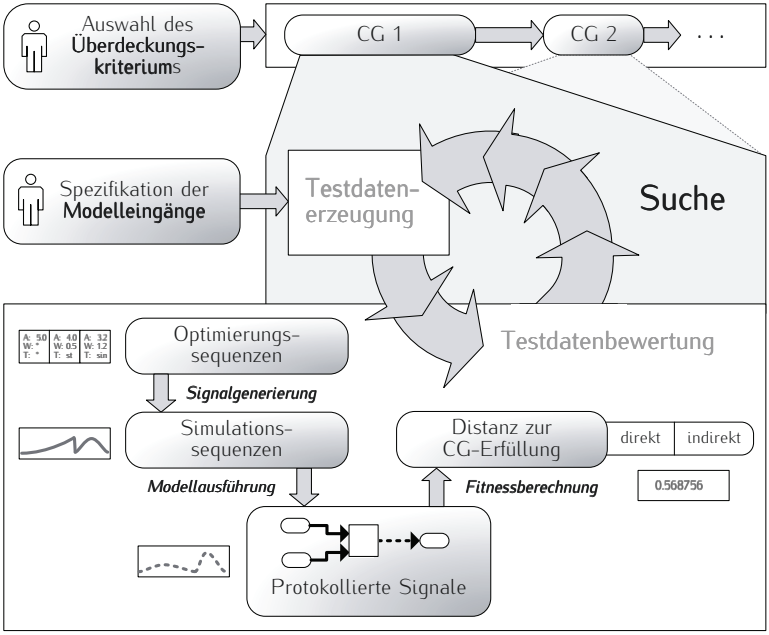
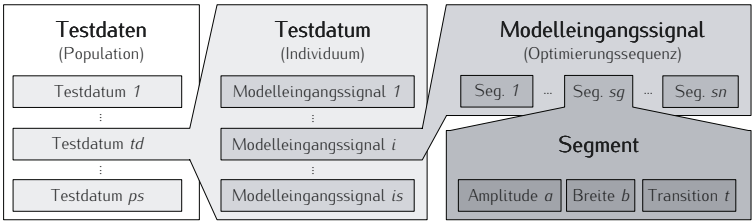


Abbildung 6: Strukturorientierte Testdatensuche für SL/TL-Modelle

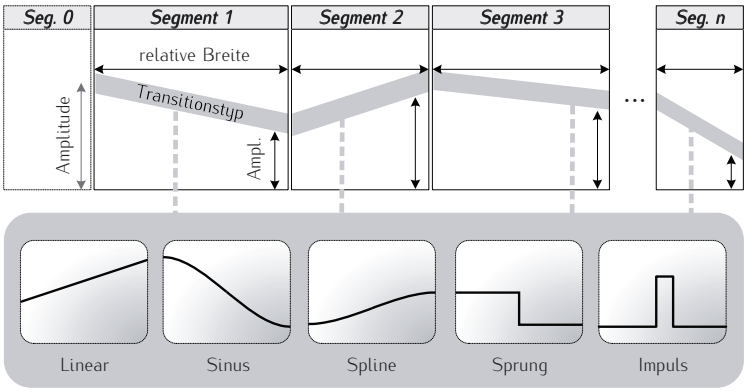
Optimierungsiteration eine Menge von Testdaten betrachtet. Diese Menge sei im Folgenden Population genannt. Ein Testdatum wird auch als Individuum bezeichnet. Ein Individuum ist Träger einer geordneten Menge von Optimierungssequenzen. Die Menge beinhaltet dabei exakt so viele Elemente wie das Modell Eingänge hat. In diesem Zusammenhang sei die Menge der Modelleingänge MB, bestehend aus den Inport-Blöcken der obersten Hierarchie-Ebene im Modell, gegeben durch:

$$\mathrm{MB} = \{\mathrm{ib} \mid \mathrm{ib} \in \mathrm{B}_{\wedge}\ \mathrm{bt}_{\mathrm{ib}} = \mathrm{Inport}_{\wedge} \forall \mathrm{sb} \in \mathrm{B}\colon \mathrm{ib} \notin \mathrm{BC}_{\mathrm{sb}}\} \tag{17}$$

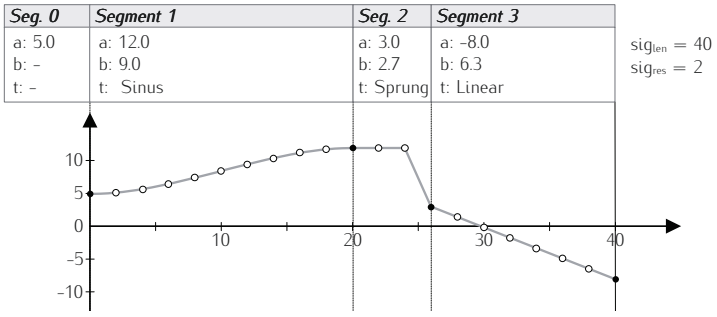
Um nun die Motivation hinter der Einführung einer Signalkodierung und einer Trennung zwischen Optimierungs- und Simulationssequenz zu verstehen, ist zu bedenken, was ein Signal eigentlich strukturell darstellt und welche Auswirkung dies auf die Anwendung eines Suchverfahrens hat. Ein (diskretes) Signal ist eine Sequenz von Werten. Für ein SL/TL-Modell gilt es, ausreichend lange Eingangssignale zu generieren, da ein Modell in der Regel zeitbehaftete Funktionalitäten enthält. Dieses Problem erkannten auch McMinn und Holcombe [85], verallgemeinernd für zustandsbehaftete Systeme. Im Falle von SL machen zu kurze Eingangssignale und eine entsprechend kurze Simulationsdauer es unter Umständen unmöglich, einzelne CGs zu erreichen. Werden allerdings



(a) Struktur der Testdatenrepräsentation



(b) Aufbau einer Optimierungssequenz im Allgemeinen



(c) Optimierungssequenz mit dazugehöriger Simulationssequenz (Beispiel)

Abbildung 7: Aufbau der Testdaten- und Signalkodierung im Detail und Funktionsweise der Signalgenerierung anhand eines Beispiels (in Anlehnung an Windisch [132])

Signale mit sehr vielen Werten generiert und wird jeder Signalwert dabei als zu optimierender Parameter aufgefasst, so führt dies zum einen aufgrund der hohen Parameterzahl zu einer ineffizienten Suche. Zum anderen würden dabei sehr sprunghafte Signalverläufe entstehen, die in der Praxis selten gewollt sind.

Baresel et al. schlagen daher einen Ansatz zur Signalkodierung vor [11]. Dieser sieht vor, ein Signal als eine Sequenz von Signalsegmenten aufzufassen. Windisch und Al Moubayed [133] greifen diese Idee auf und bauen den Ansatz wie im Folgenden beschrieben aus. Jedes Segment einer Optimierungssequenz besitzt drei Parameter: Einen Signalwert, den es am Ende des Segments einnimmt (Amplitude), eine relativ zu den restlichen Segmenten zu betrachtende Breite und einen Transitionstyp, der die Art des Signalverlaufs zwischen Beginn und Ende des Segments beschreibt. Auf diese Art und Weise reduziert sich die Parameterzahl eines Signals drastisch. Zusätzlich ergeben sich Möglichkeiten, das Aussehen und die Charakteristik der zu generierenden Signale zu kontrollieren.

So kann der Anwender vor Durchführung einer Testdatensuche die Menge der möglichen Amplitudenwerte je Modelleingang einschränken. Auch kann je Modelleingang bei Bedarf ein Startamplitudenwert für den ersten Zeitschritt angegeben werden. Diese zwei Angaben der Modelleingangsspezifikation seien wie folgt definiert:

$$\begin{aligned}&\forall\, ib \in MB:\\ &\quad spec_{dom}(ib) = [a;b]_q \quad (a,b,q \in \mathbb{R}_{\wedge}\, a \geqslant b_{\wedge}((b-a) \bmod q) = 0) \\ &\quad spec_{init}(ib) = x \in spec_{dom}(ib)\end{aligned} \tag{18}$$

Im Falle der Wertebereich-Angabe bezeichnet q die Quantisierung eines Signals. Dies ist der Abstand zwischen den einzelnen aufeinanderfolgenden Werten zwischen a und b. So entspricht $[1;10]_3$ beispielsweise der Menge $\{1,4,7,10\}$.

Neben Wertebereich und Startamplitude kann je Modelleingang eine Mindest- und Höchstzahl an Segmenten spezifiziert werden. Hierüber lässt sich unter anderem steuern, wie sprunghaft ein Signalverlauf sein darf. Auch kann der Anwender die Menge der für einen Modelleingang möglichen Transitionstypen einschränken. Zur Auswahl stehen hierbei grundsätzlich folgende Typen: ein linearer Übergang, ein Sinus-förmiger Verlauf, eine Spline-Kurve, ein Sprung und ein Impuls. Hierzu empfiehlt sich ein Blick auf Abbildung 7b, welche den Aufbau einer Optimierungssequenz veranschaulicht. Windisch betrachtet eine Reihe weiterer Angaben, die in der sogenannten Modelleingangsspezifikation gemacht werden können – an dieser Stelle seien aber nur diejenigen Angaben definiert, welche in den Beiträgen dieser Arbeit eine Rolle spielen. Hierzu gehören neben den oben angegebenen Modelleingangs-spezifischen

Angaben auch zwei Parameter, die für die Signalgenerierung aller Eingangssignale gelten:

$$\begin{aligned} &\mathrm{spec}_{\mathrm{len}} \in \mathbb{R}_{>0} \\ &\mathrm{spec}_{\mathrm{res}} \in \{x \mid x \in \mathbb{R}_{>0} \wedge \mathrm{spec}_{\mathrm{res}} \leqslant \mathrm{spec}_{\mathrm{len}} \wedge (\mathrm{spec}_{\mathrm{len}} \bmod \mathrm{spec}_{\mathrm{res}}) = 0\} \end{aligned} \tag{19}$$

Der Parameter $\mathrm{spec}_{\mathrm{len}}$ definiert die Länge der zu generierenden Signale in Sekunden. Die Abtastrate, in dieser Arbeit auch als Auflösung bezeichnet, wird durch $\mathrm{spec}_{\mathrm{res}}$, ebenfalls in Sekunden, festgelegt. $\mathrm{spec}_{\mathrm{len}}$ hat dabei ein Vielfaches von $\mathrm{spec}_{\mathrm{res}}$ zu sein. Aus Länge und Auflösung leitet sich die Anzahl der Datenpunkte eines Signals ab:

$$\mathrm{spec}_{\mathrm{steps}} = \mathrm{spec}_{\mathrm{len}} \div \mathrm{spec}_{\mathrm{res}} \tag{20}$$

All diese Angaben werden bei der Erzeugung von Optimierungssequenzen und deren Überführung zu Simulationssequenzen berücksichtigt. Der letztgenannte Schritt, die Signalgenerierung, ist beispielhaft in Abbildung 7c dargestellt. Zu sehen ist eine Optimierungssequenz mit vier Segmenten. Da in diesem Beispiel kein Startamplitudenwert spezifiziert ist, wird stattdessen der Amplitudenwert des ersten Segments als Startwert genutzt. Wie in der entstehenden Simulationssequenz zu erkennen ist, verwendet jedes der folgenden Segmente den Amplitudenwert des Vorgängersegments als Startwert. Der Amplitudenwert eines Segments bildet hingegen den Wert am Segmentende. Da die Segmentbreiten relative Angaben sind, erfolgt entsprechend der spezifizierten Signallänge und Auflösung eine Berechnung der Datenpunkt-Anzahl, welche jedes Segment in der Simulationssequenz einnimmt. Ist die Anzahl der Datenpunkte für ein Segment bekannt, so kommt die Transitionsfunktion des betreffenden Transitionstyps zum Einsatz, um den Signalwert eines jeden Datenpunkts zu bestimmen.

Da die Operatoren des eingesetzten Suchverfahrens für die Variation von Testdaten zuständig sind, müssen auch diese mit der Signalkodierung umgehen können. Windisch setzt primär auf einen genetischen Algorithmus mit angepasstem Rekombinations- und Mutationsoperator. Diese Operatoren tauschen oder verändern hierbei nicht nur die verschiedenartigen Segmentparameter. Sie sind auch dazu in der Lage, Segmente zu einer Optimierungssequenz hinzuzufügen oder zu entfernen. Daher wird dieses globale Suchverfahren im Folgenden als *längenvariabler genetischer Algorithmus* (LGA) bezeichnet. Für die verschiedenen Operatoren des LGA existieren eine Reihe von Parametern, welche unter anderem die Wahrscheinlichkeiten der einzelnen Operatoraktionen festlegen. Für eine detaillierte Betrachtung des LGA sei auf die Arbeit von Windisch verwiesen [132].

Zur Durchführung einer suchbasierten Testdatengenerierung wird des Weiteren je CG eine Fitnessfunktion benötigt. Diese lässt sich, wie bereits

angedeutet, automatisch aus einem CG-Ausdruck (siehe Definition 14) ableiten. Die vorliegende Arbeit folgt hierbei dem Vorgehen von Windisch [132]. Zunächst wird für das betrachtete Überdeckungsziel $cg \in E$ je diskretem Zeitschritt ts der zuvor erfolgten Modellausführung ein Fitnesswert berechnet. Grundlage der Fitnessberechnung sind die bei der Modellausführung protokollierten Signalverläufe genau der Signale, welche Teil des CG-Ausdrucks sind. Die Fitnessberechnung erfolgt je Zeitschritt über die in Tabelle 5 definierte Distanzfunktion $\delta_i : E \times \mathbb{N} \to \mathbb{R}$, wobei $i \in \{T, F\}$ angibt, ob das Ziel die Erfüllung oder die Nicht-Erfüllung des betrachteten CG oder eines CG-Teilausdrucks ist. κ bezeichnet eine möglichst kleine positive reelle Zahl. Da jedes $\xi_1, \xi_2 \in L$ eines CG-Ausdrucks jeweils entweder ein Signal oder eine Konstante bezeichnet, gibt die Funktion $r : L \times \mathbb{N} \to \mathbb{R}$ dementsprechend entweder den bei der Modellausführung protokollierten Wert eines Signals zu einem Zeitschritt oder den Wert der Konstanten an:

$$r(\xi, ts) = \begin{cases} out^{v}_{b(src(s)),pos(src(s))}(ts) & , \xi = s^{v} \\ c & , \xi = c \end{cases} \tag{21}$$

Auf diesem Wege wird eine Sequenz von Fitnesswerten berechnet:

$$fs(cg) = (\delta_T(cg, 0), \ldots, \delta_T(cg, spec_{steps} - 1)) \tag{22}$$

Diese wird anschließend in einen einzelnen Fitnesswert überführt:

$$f(cg) = \begin{cases} 0 & , \min(fs(cg)) \leqslant 0 \\ \overline{fs(cg)} + \min(fs(cg)) & , \text{sonst} \end{cases} \tag{23}$$

φ	*Distanz zur Erfüllung* ($i = T$)	*Distanz zur Nicht-Erfüllung* ($i = F$)
$\xi_1 < \xi_2$	$r(\xi_1, ts) - r(\xi_2, ts) + \kappa$	$\delta_T(\xi_1 \geqslant \xi_2, ts)$
$\xi_1 > \xi_2$	$r(\xi_2, ts) - r(\xi_1, ts) + \kappa$	$\delta_T(\xi_1 \leqslant \xi_2, ts)$
$\xi_1 \leqslant \xi_2$	$r(\xi_1, ts) - r(\xi_2, ts)$	$\delta_T(\xi_1 > \xi_2, ts)$
$\xi_1 \geqslant \xi_2$	$r(\xi_2, ts) - r(\xi_1, ts)$	$\delta_T(\xi_1 < \xi_2, ts)$
$\xi_1 = \xi_2$	$\lvert r(\xi_1, ts) - r(\xi_2, ts) \rvert$	$\delta_T(\xi_1 \neq \xi_2, ts)$
$\xi_1 \neq \xi_2$	1	$\delta_T(\xi_1 = \xi_2, ts)$
$\varphi_1 \wedge \varphi_2$	$\delta_T(\varphi_1, ts) + \delta_T(\varphi_2, ts)$	$\frac{\delta_F(\varphi_1, ts) \cdot \delta_F(\varphi_2, ts)}{\delta_F(\varphi_1, ts) + \delta_F(\varphi_2, ts)}$
$\varphi_1 \vee \varphi_2$	$\frac{\delta_T(\varphi_1, ts) \cdot \delta_T(\varphi_2, ts)}{\delta_T(\varphi_1, ts) + \delta_T(\varphi_2, ts)}$	$\delta_F(\varphi_1, ts) + \delta_F(\varphi_2, ts)$
$\neg \varphi_1$	$\delta_F(\varphi_1, ts)$	$\delta_T(\varphi_1, ts)$

Tabelle 5: Distanz $\delta_i(\varphi, ts)$ zur (Nicht-)Erfüllung eines CG-Ausdrucks φ zum Zeitschritt ts (in Anlehnung an Windisch [132])

Hierbei bezeichnet $\overline{\text{fs}(\text{cg})}$ das arithmetische Mittel aller Werte aus der Fitness-Sequenz und $\min(\text{fs}(\text{cg}))$ den kleinsten Wert dieser Sequenz.

Der so entstehende Fitnesswert bewertet das Testdatum, das bei der Modellausführung zum Einsatz kam, und wird von den Operatoren des Suchverfahrens zu Optimierungszwecken genutzt. Das betrachtete CG gilt als erreicht, wenn ein Testdatum den Fitnesswert 0 besitzt. Neben diesem Abbruchkriterium für die Suche begrenzt Windisch in seinem Ansatz die Anzahl der Optimierungsiterationen.

2.5 PROBLEMSTELLUNG

Der suchbasierte Ansatz hat sich in den Arbeiten von sowohl Windisch [131–133] als auch Zhan [135, 136] als geeignet für eine Testdatengenerierung aus SL-Modellen erwiesen. Während Windisch, wie beschrieben, eine GS verwendet, setzt Zhan eine LS ein. Windisch und Zhan vergleichen ihre Verfahren beide mit einem Zufallstest und dem kommerziellen Werkzeug Reactis. Diese werden hinsichtlich des erzielten Überdeckungsgrads vom suchbasierten Ansatz in beiden Fällen übertrumpft. Windisch nutzte in seinen Fallstudien allerdings deutlich größere Modelle.

Auch wenn diese Ergebnisse als sehr positiv zu werten sind, so existieren dennoch Hürden, die einer Anwendung in der industriellen Praxis im Wege stehen. Aus Nutzersicht spiegelt sich dies vor allem in den teils sehr hohen Laufzeiten einer suchbasierten Testdatengenerierung wider. Wie die Fallstudien der vorliegenden Arbeiten zeigen werden, kann es bei der Betrachtung von Modellen aus der industriellen Entwicklungspraxis durchaus zu Laufzeiten im zweistelligen Stundenbereich kommen. Ohne Anwendung des zusätzlichen Abbruchkriteriums einer Fitness-Stagnation werden aus diesen Stunden schätzungsweise sogar Tage. Ähnliche Erfahrungen machte Vos et al. bei der strukturorientierten Testdatensuche für C-Code [121].

Aus dieser Betrachtung leitet sich die Zielsetzung dieser Arbeit ab: die Steigerung der Effizienz der suchbasierten Testdatengenerierung in Anwendung für SL/TL-Modelle. Zusätzlich muss es das Ziel sein, die Effizienzsteigerung ohne Einbußen bei den Überdeckungsgraden zu erreichen. Um dies beides erreichen zu können, sind folgende zwei Fragen zu beantworten:

1. Gibt es Potenzial zur Verbesserung der Effizienz einer suchbasierten Testdatengenerierung für SL/TL-Modelle?
2. Lassen sich Änderungen am suchbasierten Verfahren oder erweiternde Techniken formen, welche dieses Potenzial ausschöpfen?

Die erste Frage wird in den folgenden Zeilen dieses Abschnitts adressiert. Auf die drei zuletzt genannten der sechs hierbei aufgeführten Problemstellungen weist auch Windisch im Ausblick seiner Arbeit hin [132]. Die zweite Frage wird in den folgenden Kapiteln dieser Arbeit beantwortet.

Unerreichbarkeit. Die erste Problemstellung besteht in der Unerreichbarkeit mancher CGs. Auch wenn ein Modell seine beabsichtigte Funktionalität vollständig korrekt implementiert, gibt es in der Regel Modellzustände, deren Eintreten durch keinerlei Testdaten bewirkt werden kann. Der in der Code-Welt bekannte „tote" Code existiert auch in der SL-Welt. In der Entwicklungspraxis werden beispielsweise vordefinierte Bibliotheksblöcke in Modellen verwendet. Solche Blöcke realisieren häufig benötigte und wiederkehrende Funktionalitäten. De facto handelt es sich bei diesen Blöcken um referenzierte Subsysteme, mit Ein- und Ausgängen. Nun werden allerdings oftmals nicht alle Parameter der auf diesem Wege abgebildeten Bibliotheksfunktionen benötigt und einzelne Blockeingänge daher mit einem Konstantenblock verbunden. Dies führt innerhalb eines Bibliotheksblocks dann dazu, dass bestimmte CGs nicht erreichbar sind. Für eine suchbasierte Testdatengenerierung ist dieser Umstand höchst problematisch, da das Suchverfahren unerbittlich und ohne Aussicht auf Erfolg bis zum Eintreten eines anderweitigen Abbruchkriteriums nach nicht existierenden Testdaten fahndet. Dieser Aspekt stand nicht im Fokus der Arbeit von Windisch [132]. Allerdings betrachtete er diese Problematik in seinen Fallstudien. Die dort verwendeten Modelle wurden vor Durchführung einer Testdatensuche manuell analysiert und unerreichbare CGs vorab herausgefiltert. Die vorliegende Arbeit schlägt eine Technik (SIA) vor, welche diese Aufgabe automatisiert.

Suchraumgröße. Die Größe des Raumes aller, für die Eingänge eines Modells generierbaren Signale kann ein entscheidender Faktor sein, wenn es darum geht, wie einfach oder schwer eine Testdatensuche für ein bestimmtes CG erledigt werden kann. Je enger der Suchraum auf das gesuchte Optimum eingegrenzt werden kann, desto einfacher hat es die Suche in der Regel. Auf ein SL/TL-Modell bezogen wächst die Größe des Suchraums allen voran mit der Anzahl der Modelleingänge. Nun ist es allerdings so, dass das Erreichen vieler Zustände und CGs nicht von der Stimulation aller Modelleingänge abhängig ist – sondern häufig nur von einer Teilmenge der Eingänge. An genau dieser Stelle setzt ein in dieser Arbeit vorgestelltes Verfahren (SDA) an.

Überdeckungsziel-Abhängigkeiten. Wie in Abschnitt 2.2.1 geschildert, bedingt ein Strukturtest in der Regel das Erreichen einer Vielzahl von CGs. Zwischen den CGs existieren jedoch unzweifelhaft Abhängigkeiten: Das Erreichen eines CG hat unter Umständen das Erreichen eines anderen CG zur Folge (Kollateralüberdeckung). Je höher die Kollateralüberdeckung bei einer strukturorientierten Testdatensuche ausfällt, desto weniger Suchvorgänge müssen insgesamt unternommen werden.

Der Ansatz von Windisch berücksichtigt diesen Aspekt nicht [132]. Die vorliegende Arbeit ergänzt den Ansatz daher um eine Technik (CGSeq), die den beschriebenen Sachverhalt ausnutzt, um die Kollateralüberdeckung und somit auch die Effizienz eines automatisierten Strukturtests zu steigern.

Blindheit der Fitnessfunktion. Eine der Stärken des suchbasierten Ansatzes ist gleichzeitig eine seiner Schwächen. Zur Erfüllung eines CG betrachtet er nämlich den Teil des Modells zwischen Modelleingängen und den an dem CG beteiligten Signalen als eine Art *Black-Box*. Dies bedeutet, das Modell wird an den Eingängen mit Testdaten stimuliert, die in den relevanten Signalen entstandenen Werte werden ausgelesen und anschließend zur Fitnessberechnung verwendet. Jegliche Berechnungsschritte dazwischen werden nicht berücksichtigt. Diese Eigenschaft macht den suchbasierten Ansatz auf der einen Seite hoch-skalierbar – vor allem im Vergleich mit statischen Verfahren wie der symbolischen Ausführung. Auf der anderen Seite können in diesem als Black-Box bezeichneten Teil eines Modells jedoch Bedingungen oder Berechnungen enthalten sein, von deren Berücksichtigung oder Erfüllung das Erreichen des verfolgten CG entscheidend abhängt. Dieses Problem erkannten bereits McMinn et al. [84, 86], wenn auch nicht auf den Modelltest bezogen. Die CGSeq-Technik der vorliegenden Arbeit adressiert auch dieses Problem, löst es allerdings nur für einen Teil der möglichen Problemfälle.

Boolesche Signale. Je nach Anwendungsbereich bilden SL/TL-Modelle teils mehr, teils weniger berechnende oder aussagenlogische Funktionalitäten ab. Ist ein Modell entsprechend Logik-lastig, so enthält es viele Signale mit Booleschem Wertebereich. Existieren CGs, welche Bedingungen über solche Booleschen Signale definieren, gestaltet sich eine Bewertung der Testdaten schwierig. Gemäß der in Abschnitt 2.4.2 beschriebenen Vorgehensweise zur Distanzberechnung ergibt sich für ein Testdatum, welches ein solches CG nicht erfüllt, die Distanz 1. Die Fitnessfunktion bietet dem Suchverfahren demnach keinerlei Unterstützung bei der Unterscheidung von Testdaten. Die CGSeq-Technik der vorliegenden Arbeit und eine Testdatengenerierung via SMT-Solving, wie in der vorliegenden Arbeit behandelt, adressieren dieses Problem zum Teil.

Anzahl aufgesuchter Testdaten. Die Effizienz einer automatischen Testdatensuche hängt maßgeblich von der Anzahl an Testdaten ab, welche im Zuge der Suche erzeugt, bewertet und möglicherweise verworfen werden. Denn je betrachtetem Testdatum erfolgt eine Ausführung des SL/TL-Modells. Je nach Größe und Gestaltung des Modells variiert die Dauer einer Modellausführung. Eine Ausführungsdauer von zehn Sekunden oder sogar mehr ist bei Modellen aus der industriellen Praxis nicht unüblich. Aus diesem Grund wirkt sich die Vorgehensweise eines Suchverfahrens direkt auf die Laufzeit der gesamten Testdatengenerierung aus. Globale Suchverfahren wie der LGA sind bekannt dafür, eine

hohe Anzahl an Testdaten im Zuge ihrer Suchiterationen zu betrachten. Eine LS kann effizienter sein, sofern sie es schafft, das Ziel der Suche zu finden. Die Arbeit untersucht daher, ob sich im betrachteten Kontext eine LS anwenden lässt und wie effizient, beziehungsweise erfolgreich, eine LS Testdaten für ein SL/TL-Modell generieren kann. Darüber hinaus wird die Möglichkeit einer Testdatengenerierung mittels SMT-Solving untersucht.

Die aufgeführten Potenziale, welche der suchbasierte Ansatz von Windisch aufweist, bilden die Grundlage der in den nachfolgenden Kapiteln vorgestellten Lösungen. Der suchbasierte Ansatz von Windisch wird dabei um eine Reihe von Techniken ergänzt. Die Einführung zusätzlicher Techniken ist hierbei um eine wichtige, zusätzliche Anforderung begleitet: Erweiternde Techniken dürfen keinerlei Einschränkungen bezüglich der Beschaffenheit verwendbarer SL/TL-Modelle machen. Die nachfolgend vorgestellten Techniken zur statischen Voranalyse eines SL/TL-Modells erfüllen diese Anforderung, da sie auch mit unbekannten oder durch sie nicht explizit unterstützten Blocktypen umgehen können.

Teil II

HYBRIDES VERFAHREN ZUR TESTDATENGENERIERUNG

3 STATISCHE VORANALYSEN

Eine Hybridisierung des suchbasierten Ansatzes mit ergänzenden statischen oder dynamischen Techniken ist auf drei Ebenen möglich. Unterstützende Techniken können vor Durchführung einer Testdatensuche, während (oder zum Teil anstelle) dieser oder danach aktiv sein. In diesem Kapitel werden drei Techniken vorgestellt, die einer Testdatensuche für SL/TL-Modelle vorausgehen. Die drei Techniken operieren statisch, sie sind vollständig automatisierbar und sie haben die Aufgabe, die Suche besser auf die sich ihr stellenden Probleme einzustellen und hierdurch die Effizienz der Testdatengenerierung zu steigern.

Allen drei als statische Voranalysen bezeichneten Techniken gehen grundlegende Modelltransformationen voraus. In diesem Zusammenhang ist anzumerken, dass die Voranalysen mit einer Repräsentation des betrachteten SL/TL-Modells arbeiten. Das originale Modell bleibt von etwaigen Transformationen unberührt. Bei den vorausgehenden Transformationen handelt es sich einerseits um die Auflösung von Bus-Systemen, das heißt deren Zerlegung in ihre enthaltenen Signale. Andererseits geht es um das Entfernen von Modellbestandteilen, die für die Analysen irrelevant sind. Dies betrifft Blöcke und Signale, die sich zwischen den Ausgängen des Modells und Signalen oder Blöcken, welche in Zusammenhang mit Überdeckungszielen stehen, befinden. Derlei Blöcke und Signale sind am Datenfluss zwischen Modelleingängen und Überdeckungszielen nicht beteiligt und können daher aus der Modellrepräsentation entfernt werden. Die vorausgehenden Transformationen erleichtern die Abläufe der statischen Voranalysen und reduzieren ihren Aufwand.

3.1 ÜBERDECKUNGSZIEL-ERREICHBARKEITSPRÜFUNG

Der folgende Abschnitt behandelt Sinn und Zweck der ersten statischen Voranalyse, der sogenannten Überdeckungsziel-Erreichbarkeitsprüfung (SIA). Die Bedeutung der Abkürzung SIA wird ebenfalls im folgenden Abschnitt erklärt. Danach wird SIA im Detail vorgestellt und verwandte Arbeiten aus dem Teilbereich, in dem SIA sich bewegt, beleuchtet.

3.1.1 ZIEL

Bei einem Strukturtest eines SL/TL-Modells besteht die Möglichkeit, dass ein CG unerreichbar ist. Dies bedeutet, dass keinerlei Testdaten existieren, die zu einer Erfüllung führen. Dieser Umstand bedeutet allerdings nicht zwangsläufig, dass das Modell seine Spezifikation nicht korrekt umsetzt oder dass es dem Modell an Qualität mangelt. In Abschnitt 2.5 wurde bereits eine mögliche Ursache für die Unerreichbarkeit von CGs aufgeführt: die unvollständige Verwendung von Bibliotheksblöcken. Eine weitere Ursache besteht in der Auflösung von Variabilität zum Zeitpunkt der Modellausführung. Systeme und Funktionen, die mit SL/TL-Modellen beschrieben werden, sind häufig konfigurierbar gestaltet. Ein Modell kann zum Beispiel verschiedene Softwarevarianten gleichzeitig abbilden. Die Variabilität entsteht dabei in der Regel durch die Verwendung von Variablen in Blockparametern, beispielsweise bei einem Constant-Block. Die Konfigurierung, das heißt die Festlegung auf eine Variante, erfolgt dann für gewöhnlich über eine Variablenbelegung im ML-Workspace. Abhängig hiervon können Teile des Modells bei dessen Ausführung inaktiv oder bestimmte Zustände nicht erreichbar sein.

Der suchbasierte Ansatz erkennt die eventuelle Unerreichbarkeit eines CG nicht. In Folge dessen wird, ohne Aussicht auf Erfolg, eine Testdatensuche durchgeführt. Je nachdem, welche Abbruchkriterien zum Einsatz kommen, wirkt sich das Vorhandensein eines unerreichbaren CG oder mehrerer unerreichbarer CGs entsprechend negativ auf die Laufzeit der gesamten Strukturtestautomatisierung aus. Erfahrungen mit Modellen aus der industriellen Praxis zeigen, dass oftmals nicht wenige unerreichbare CGs in Modellen existieren [132]. Daher ist eine automatisierte Erkennung dieser für eine automatisierte Testdatengenerierung von großem Nutzen. Dies gilt nicht nur, weil die Testdatengenerierung hierdurch effizienter erfolgen kann. Auch ermöglicht eine solche Erkennung eine präzisere Bestimmung des Überdeckungsgrads beim Strukturtest. Des Weiteren kann die Tatsache, dass ein CG unerreichbar ist, für den Entwickler eines Modells möglicherweise doch ein Hinweis auf eine fehlerhafte oder nicht ideale Modellierung sein.

Im Allgemeinen ist es nicht mit vertretbarem Aufwand möglich, mit einer statischen Analyse jede mögliche Kombination von Zuständen innerhalb eines beliebig großen Modells zu betrachten. Grund hierfür ist die explodierende, oftmals sogar unendliche Anzahl möglicher Zustandskombinationen [50, 78, 134]. Statische Analysetechniken nutzen daher meist Abstraktionstechniken, um nicht an Komplexitätsgrenzen zu stoßen. Eine Abstraktion führt dabei in der Regel zu einem Verlust an Detailinformationen, die Korrektheit der produzierten Ergebnisse ist jedoch gewährleistet [38]. Für das Vorhaben, die unerreichbaren CGs eines SL/TL-Modells zu identifizieren, bedeutet dies, dass eine auf Abstraktionsprinzipien beruhende statische Analyse nicht garantieren kann,

alle unerreichbaren CGs auch tatsächlich zu finden. Dies gilt auch für die im folgenden vorgestellte SIA-Technik.

SIA greift das Konzept der abstrakten Interpretation auf. Der Kerngedanke dieses von dem Ehepaar Cousot begründeten Konzepts für statische Programmanalysen besteht in der Interpretation eines Programms mittels einer abstrakten Semantik anstatt seiner konkreten Semantik [31]. Als eine Form der Abstrahierung von den bei der Ausführung eines Programms entstehenden konkreten Werten schlagen Cousot und Cousot die Verwendung von Intervallen vor [30]. Dieser Ansatz bildet die Grundlage von SIA. Innerhalb der Modellanalyse, die SIA durchführt, kommt statt der Semantik eines jeden Modellblocks eine sogenannte Intervallsemantik zum Einsatz. Unter Einbeziehung der Modelleingangsspezifikation (siehe Abschnitt 2.4.2) wird so der Wertebereich eines jeden Modellsignals bestimmt. Eine im Anschluss durchgeführte Erreichbarkeitsanalyse ist in der Lage, für die betrachteten CGs anhand der Signalwertebereiche abzuleiten, ob ein CG unerreichbar ist. Wird dies festgestellt, so ist das betreffende CG tatsächlich unerreichbar. Im Umkehrschluss bedeutet dies allerdings nicht, dass ein CG, für welches die Unerreichbarkeit auf diesem Wege nicht festgestellt wird, zwingend erreichbar ist.

Die für diese statische Voranalyse verwendete Abkürzung orientiert sich übrigens an dem ihr zugrunde liegenden Prinzip, gemünzt auf den betrachteten Kontext: SIA steht für den Begriff Signalintervallanalyse.

3.1.2 ABSTRAKTE INTERVALL-INTERPRETATION

Das in Abbildung 8 dargestellte Beispiel soll als Einstieg für die nachfolgenden Ausführungen dienen. Zu sehen ist ein kleines Modell mit drei Eingängen (Inport-Blöcke) und zwei Ausgängen (Outport-Blöcke). Das Modell ist umgeben von einer Modelleingangsspezifikation, wie sie im Rahmen des suchbasierten Ansatzes zur Testdatengenerierung von Windisch zum Einsatz kommt. Jedem Eingang ist hierbei ein Wertebereich per Intervallangabe, gegebenenfalls inklusive Quantisierung, zugewiesen. Darüber hinaus ist über die spezifizierte Signallänge und Abtastrate die Gesamtanzahl der Zeitschritte einer Modellausführung, im Falle des Beispiels 300, bekannt.

Gemäß der Ausführungsreihenfolge der Modellblöcke kommt es nun zu einer abstrakten Ausführung des Modells. Hierbei berechnet jeder Block auf Grundlage der Wertebereiche seiner eingehenden Signale die Wertebereiche für seine Ausgangssignale (Intervallpropagierung). Jedem Signal ist dabei nicht nur ein einziges Intervall zugeordnet, sondern eine Menge von Intervallmengen. Jede Intervallmenge ist einer Zeitphase zugeordnet. Diese Zusammenhänge werden im Folgenden noch exakt definiert und erläutert. An dieser Stelle soll lediglich das Grundprinzip vermittelt

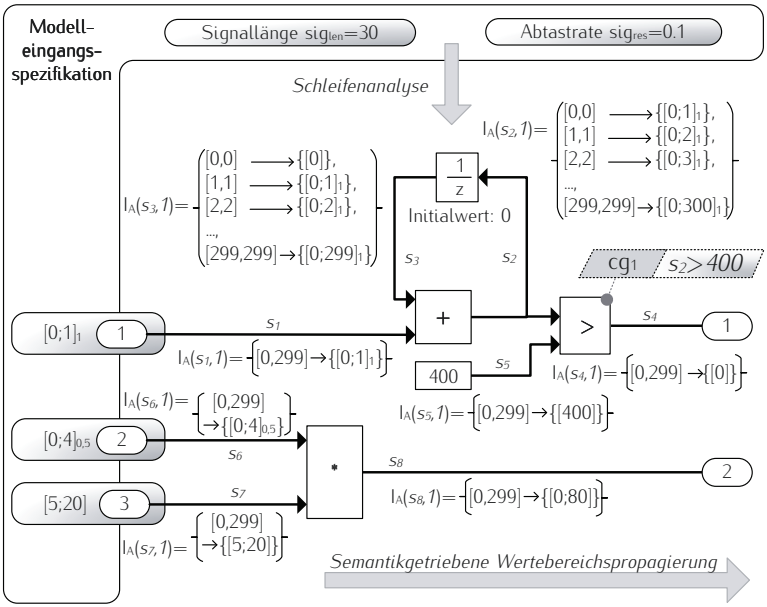


Abbildung 8: Funktionsweise der Überdeckungsziel-Erreichbarkeitsprüfung anhand eines Beispiels

werden. Hierzu seien die Modelleingänge zwei und drei im Beispiel betrachtet. Ihre Ausgangssignale besitzen zu jedem möglichen Zeitschritt einer Modellausführung, das heißt im Zeitintervall $[0,299]$, den in der Modelleingangsspezifikation entsprechend angegebenen Wertebereich. Beide Signale gehen im Modell in einen Product-Block ein. Unter Berücksichtigung der eingehenden Wertebereiche und der Intervallsemantik dieses Blocks ergibt sich der Wertebereich seines Ausgangssignals.

Diese wesentliche Verfahrensweise findet auch vom ersten Modelleingang aus betrachtet Anwendung. Allerdings kann der im Modell enthaltene Sum-Block den Wertebereich seines Ausgangssignals nicht ohne Weiteres bestimmen, da er Teil einer sogenannten Rückkopplung im Modell ist. Eine solche Rückkopplung wird in dieser Arbeit als Schleife bezeichnet. Im Falle des Beispiels wird die Schleife über einen Unit-Delay-Block gebildet. Dieser bewirkt eine zeitliche Verzögerung seiner Ausgabe um einen Zeitschritt. SIA enthält eine Schleifenanalyse, welche sich die spezifizierte Dauer einer Modellausführung zu Nutze macht, um die Wertebereiche der an der Schleife beteiligten Signale schrittweise zu bestimmen. Für das betrachtete Beispiel kommt SIA zu dem Schluss, dass das Ausgangssignal des Relational-Operator-Blocks zu jedem Zeitschritt den Wert 0 einnimmt. Das dargestellte CG ist demzufolge unerfüllbar.

Die Intervallsemantik der verschiedenen Blocktypen wird im folgenden Abschnitt behandelt. Die Vorgehensweisen der Intervallpropagierung und der Schleifenanalyse werden anschließend betrachtet.

3.1.2.1 *Intervallsemantik*

Zur Bildung einer Intervallsemantik für die in SL/TL verfügbaren Blocktypen wird in dieser Arbeit die Intervall-Arithmetik von Moore [91] genutzt. Die Arithmetik von Moore definiert unter anderem die mathematischen Operatoren für die Addition, Subtraktion, Division und Multiplikation in Anwendung für Intervalle. So gilt für zwei beliebige abgeschlossene, reelle Intervalle:

$$\begin{aligned}
[a,b]+[c,d] &= [a+c,b+d] \\
[a,b]-[c,d] &= [a-d,b-c] \\
[a,b]\cdot[c,d] &= [\min(a\cdot c,a\cdot d,b\cdot c,b\cdot d),\max(a\cdot c,a\cdot d,b\cdot c,b\cdot d)] \\
[a,b]\div[c,d] &= [\min(a\div c,a\div d,b\div c,b\div d),\max(a\div c,a\div d,b\div c,b\div d)] \\
&\quad (\text{nur definiert für } 0\notin[c,d])
\end{aligned} \tag{24}$$

Wang et al. verwenden dieses Grundkonzept ebenfalls, allerdings für die statische Analyse von Java-Programmen [125]. Um die Genauigkeit ihrer Analysen zu erhöhen, weisen sie den Programmvariablen Intervallmengen anstatt einzelner Intervalle zu. Hierzu ein Beispiel: Nimmt eine Variable Werte zwischen 10 und 20 sowie 30 und 40 an, so ist die Angabe einer Intervallmenge $\{[10,20],[30,40]\}$ zweifelsfrei exakter und verlustfreier als die Angabe des Intervalls $[10,40]$. Auch SIA verwendet Intervallmengen je Modellsignal. Um die Intervallmengen allerdings so kompakt wie möglich zu halten, enthält SIA ein Vorgehen zur Vereinigung von Intervallen – ohne hierbei allerdings an Genauigkeit einzubüßen.

Alle Intervalle haben die folgend dargestellte Form. IV bildet hierbei die Menge aller möglichen Intervalle.

$$\begin{aligned}
\mathrm{IV} = &\{[a;b] \mid a,b\in\mathbb{R}_\wedge a\leqslant b\} \cup \\
&\{[a;b]_q \mid a,b,q\in\mathbb{R}_\wedge a\leqslant b_\wedge((b-a) \bmod q)=0)\} \cup \\
&\{[a] \mid a\in\mathbb{R}\}
\end{aligned} \tag{25}$$

Wie in dieser Definition zu sehen ist, existieren drei verschiedene Varianten. Zum einen kann ein Intervall optionalerweise mit einer Quantisierung q versehen sein. Zum anderen wird für einelementige Intervalle eine verkürzende Schreibweise verwendet. Als Trennzeichen innerhalb eines Intervalls wird ein Semikolon verwendet, da es sich bei den Intervallgrenzen um reelle Zahlen handelt.

Im Zusammenhang mit dem einführenden Beispiel wurde bereits erwähnt, dass je Signal eine Aufteilung seiner Intervallmenge in zeitliche Phasen möglich ist. Auch dieser Schritt dient einer größtmöglichen Genauigkeit der Analyse. Ist SIA mit seiner Analyse fertig, so decken die zeitlichen Phasen eines jeden Signals die Gesamtzahl der für die Signalgenerierung spezifizierten Zeitschritte ab. Die Zeitphasen überlappen sich dabei nicht. Eine Zeitphase wird ebenfalls in Form eines Intervalls, mit Anfangs- und Endzeitschritt, angegeben:

$$\mathrm{TI} = \{[t_1,t_2]^{\mathsf{T}} \mid t_1,t_2 \in \mathbb{N} \wedge 0 \leqslant t_1 \leqslant t_2 \leqslant \mathrm{spec}_{\mathrm{steps}} - 1\} \tag{26}$$

Im Folgenden werden Zeitphasen auch als Zeitintervalle bezeichnet. Um zwischen Werte- und Zeitintervallen unterscheiden zu können, wird ein Zeitintervall mit einem hochgestellten „T“ markiert.

Für die Zuordnung von Intervallmengen zu Signalen über Zeitintervalle seien folgende zwei Abbildungen eingeführt, welche im Folgenden zur Definition der Intervallsemantik wahlweise Anwendung finden:

$$\begin{aligned} &I_S : (S \times \mathbb{N}^+ \times \mathrm{TI}) \to \mathcal{P}(\mathrm{IV}) \\ &I_A : (S \times \mathbb{N}^+) \to \mathcal{P}(\mathrm{TI} \times \mathcal{P}(\mathrm{IV})) \end{aligned} \tag{27}$$

Während die Abbildung I_S für ein konkretes Zeitintervall eines Signals die dazugehörige Intervallmenge liefert, gibt I_A sämtliche durch SIA gebildete Zuordnungen von Zeitintervallen zu Intervallmengen für ein Signal an. Beide Abbildungen erhalten als zweiten Parameter die Angabe eines Vektor-Index für das betreffende Signal.

Die Gestaltung der Intervallmenge eines Signals hängt im Allgemeinen sowohl von der konkreten Semantik des Modellblocks ab, aus welchem das Signal hervorgeht, als auch von den Intervallmengen der Eingangssignale dieses Blocks. Die Art und Weise, wie ein Blocktyp die Intervallmenge seiner Ausgangssignale berechnet oder ableitet, sei als Intervallsemantik bezeichnet. Bei der folgenden Beschreibung dieser für verschiedene Blocktypen kommen folgende Abbildungen zum Einsatz, um die Definitionen kompakt zu gestalten.

Die Abbildung T gibt für das ν-te Vektorelement eines Signals s die Menge der dazugehörigen, durch SIA gebildeten Zeitintervalle an:

$$\begin{aligned} &T : S \times \mathbb{N}^+ \to \mathcal{P}(\mathrm{TI}) \\ &T(s,\nu) = \{ti \mid (ti,Z) \in I_A(s,\nu)\} \end{aligned} \tag{28}$$

Die Definition der Intervallsemantik greift darüber hinaus auf die Abbildungen I_P und I_V zurück. Diese liefern jeweils die Intervallmengen der Eingangssignale eines Blocks – unterscheiden dabei aber zwischen Blöcken, welche mehrere Eingänge besitzen sowie ihre Funktionalität je Vektorelement anwenden (zum Beispiel ein *AND*-Block), und Blöcken,

welche nur einen Eingang besitzen und die ihre Funktionalität über alle Vektorelemente dieses Eingangs anwenden (zum Beispiel ein *Sum*-Block mit einem Eingang). Die Abbildung I_P beschreibt also für die v-ten Vektorelemente aller Eingangssignale eines Blocks b die Menge der dort anliegenden Intervalle, welche in ein Zeitintervall $[t_1,t_2]^T$ fallen. Die Ausgabe der Intervallmengen erfolgt dabei zugeordnet zu den Eingängen des Blocks (über die Portnummer):

$$I_P : (B \times TI \times \mathbb{N}^+) \to (\mathbb{N}^+ \times \mathcal{P}(IV)) \tag{29}$$

$$I_P(b,ti,v) = \left\{ (i,X) \;\middle|\; i \in \mathbb{N}_{[1,|IP_b|]} \wedge X = \bigcup_{Z \in OI_{ti,s,w}} Z \right\}$$

Hierbei bezeichnet $s = sig_{in}(b,i)$ das Eingangssignal des Inports mit der Portnummer i und $w = vc(v, ip(b,i))$ einen, je nach Signaldimensionalität gegebenenfalls nach unten korrigierten Vektorindex. OI_{ti} ist eine Menge all derer Intervallmengen, welche Zeitintervallen zugeordnet sind, die mit dem betrachteten Zeitintervall ti mindestens einen Zeitschritt teilen:

$$OI_{ti,s,u} = \{\, F \mid (ti',F) \in I_A(s,u) \wedge \text{overlap}(ti,ti') \,\} \tag{30}$$

Die Funktion $\text{overlap} : TI \times TI \to \mathbb{B}$ gibt an, ob ein Zeitintervall mindestens einen Zeitschritt mit einem anderen Zeitintervall teilt:

$$\text{overlap}([t_1,t_2]^T, [t_1',t_2']^T) = t_2' \geqslant t_1 \wedge t_1' \leqslant t_2 \tag{31}$$

Die Abbildung I_V funktioniert im Vergleich hierzu ähnlich, dient aber der Intervallsemantik-Definition von Blöcken, die nur einen Eingang besitzen und die ihre Funktionalität über alle Vektorelemente des einen Eingangssignals definieren. Dementsprechend bildet I_V die Intervallmengen des Eingangssignals, welche im Zeitintervall $[t_1,t_2]^T$ gelten, in Zuordnung mit dem jeweiligen Vektorindex ab:

$$I_V : (B \times TI) \to (\mathbb{N}^+ \times \mathcal{P}(IV)) \tag{32}$$

$$I_V(b,ti) = \left\{ (i,X) \;\middle|\; i \in \mathbb{N}_{[1,d(b,1)]} \wedge X = \bigcup_{Z \in OI_{ti,s,i}} Z \right\}$$

Hierbei bezeichnet $s = sig_{in}(b,1)$ das einzige Eingangssignal.

Die Funktionen min und max werden des Weiteren verwendet, um die minimale untere Grenze oder die maximale obere Grenze einer Intervallmenge, das heißt aller ihrer enthaltenen Intervalle, zu bestimmen:

$$\min, \max : \mathcal{P}(IV) \to \mathbb{R} \tag{33}$$

$$\min(X) = a \quad \textbf{mit } ([a;b] \in X \vee [a;b]_q \in X \vee [a] \in X) \wedge (\forall [c;d],[c;d]_r,[c] \in X\text{: } a \leqslant c)$$

$$\max(X) = b \quad \textbf{mit } ([a;b] \in X \vee [a;b]_q \in X \vee [b] \in X) \wedge (\forall [c;d],[c;d]_r,[d] \in X\text{: } b \geqslant d)$$

Die untere und obere Grenze sowie Quantisierung eines einzelnen Intervalls seien durch die Funktionen lb, ub und q gegeben:

$$
\begin{aligned}
&\mathrm{lb}, \mathrm{ub}, q : \mathrm{IV} \to \mathbb{R} \\
&\mathrm{lb}(\mathrm{iv}) = a \quad \textbf{mit } \mathrm{iv} = [a;b]_{\vee} \; \mathrm{iv} = [a;b]_{q\vee} \; \mathrm{iv} = [a] \\
&\mathrm{ub}(\mathrm{iv}) = b \quad \textbf{mit } \mathrm{iv} = [a;b]_{\vee} \; \mathrm{iv} = [a;b]_{q\vee} \; \mathrm{iv} = [b] \\
&q(\mathrm{iv}) = \begin{cases} q & , \mathrm{iv} = [a;b]_q \\ 0 & , \text{sonst} \end{cases}
\end{aligned}
\tag{34}
$$

Eine Definition der Intervallsemantik einiger wichtiger Blocktypen ist in Tabelle 6 angegeben. Die folgenden Absätze erläutern diese.

Constant, Inport und Subsystem. Wie aus der Definition für den Constant-Block hervorgeht, gibt dieser unabhängig von dem betrachteten Zeitintervall stets ein einelementiges Intervall mit seinem Konstantenwert aus. Im Falle des Inport-Blocks gilt es zu unterscheiden, ob sich dieser in einem Subsystem befindet oder ob es sich um einen Modelleingang handelt. In ersterem Fall gibt der Block die Intervallmenge aus, welche dem dazugehörigen Eingang des übergeordneten Subsystems für das betrachtete Zeitintervall zugewiesen ist. Handelt es sich um einen Modelleingang, so leitet sich das ausgehende Intervall von der Wertebereichsangabe in der Modelleingangsspezifikation ab. Für die Ausgänge eines virtuellen Subsystems wird die Intervallmenge jeweils ohne Änderung von dem Eingang des dazugehörigen Outport-Blocks im Subsystem übernommen.

Logical und Relational Operator. Der Logical-Operator-Block prüft zur Bestimmung seiner auszugebenden Intervallmenge in Abhängigkeit des logischen Operators, ob die in den Block eingehenden Intervallmengen für das betrachtete Zeitintervall zu einer Auswertung der Operatorbedingung mit ausschließlich *falsch* (oder *wahr*) führen. Für den *And*-Operator wird beispielsweise geprüft, ob in den Intervallmengen aller Eingänge der Wert 0 nicht enthalten ist. Ist dies der Fall, besteht die auszugebende Intervallmenge ausschließlich aus dem Intervall [1]. Enthält die Intervallmenge mindestens eines Eingangs des *And*-Blocks hingegen ausschließlich das Intervall [0], so lautet das Ausgabe-Intervall [0]. Trifft keiner dieser beiden Fälle zu, wird das Intervall $[0;1]_1$ ausgegeben. Der Relational-Operator-Block verfährt in seiner Intervallsemantik sehr ähnlich. Es wird ebenfalls geprüft, ob das Zutreffen oder Nicht-Zutreffen seiner Operatorbedingung mit den gegebenen Intervallmengen der Blockeingänge überhaupt möglich ist. Entsprechend gibt der Block anstatt des Booleschen Intervalls $[0;1]_1$ gegebenenfalls [0] oder [1] aus.

Sum. Der Sum-Block berechnet seine auszugebende Intervallmenge paarweise über die eingehenden Intervallmengen. Hat der Block nur einen Eingang, so sind dies die Intervallmengen der einzelnen Vektorelemente des Eingangssignals. Hat der Block mehrere Eingänge, so sind dies die

Intervallmengen der unterschiedlichen Eingangssignale mit gleichem Vektorindex. Das weitere Vorgehen ist in beiden Fällen identisch, daher sei an dieser Stelle der Einfachheit halber der Fall mit mehreren Blockeingängen betrachtet. Die Intervallberechnung erfolgt dann im ersten Schritt für die Intervallmenge des ersten Blockeingangs und der hinsichtlich einer Addition/Subtraktion neutralen Intervallmenge $\{[0]\}$. Im zweiten Schritt erfolgt die Intervallberechnung mit der aus dem ersten Schritt resultierenden Intervallmenge und der Intervallmenge des zweiten Blockeingangs. Dieser Schritt wiederholt sich entsprechend der Anzahl an Blockeingängen. Die aus dem allerletzten Schritt resultierende Intervallmenge ist die gesuchte Intervallmenge. Die Intervallberechnung erfolgt paarweise, da jeder Blockeingang einer unterschiedlichen Operation, das heißt Addition oder Subtraktion, unterliegen kann. Im Kern wendet die Intervallsemantik des Sum-Blocks die grundlegenden Rechenregeln für Intervalle an. Ergänzend hierzu wird allerdings auch die Quantisierung von Intervallen berücksichtigt. Besitzen zwei zu addierende oder subtrahierende Intervalle jeweils eine Quantisierung, so bildet der größte gemeinsame Teiler dieser Quantisierungen die Quantisierung des resultierenden Intervalls.

Switch. Bei einem Switch-Block gilt es, zwei Fälle zu unterscheiden. So können der Kontrolleingang oder der Schwellwert-Parameter des Blocks mehrdimensional sein - oder eben nicht. Unabhängig von dieser Unterscheidung lässt sich die Intervallsemantik eines Switch-Blocks folgendermaßen zusammenfassen: Es wird geprüft, ob die am Kontrolleingang anliegende(n) Intervallmenge(n) des betrachteten Zeitintervalls zu einer ausschließlichen Weiterleitung des ersten oder dritten Blockeingangs zum Blockausgang führen. Diese Prüfung erfolgt durch die Funktionen rt_1 und rt_3. Ist rt_1 *wahr*, so entspricht die Intervallmenge am Ausgang des Blocks der Intervallmenge am ersten Blockeingang. Entsprechendes gilt für rt_3 und den dritten Blockeingang. Tritt keiner dieser beiden Fälle ein, so wird die auszugebende Intervallmenge aus den Intervallmengen von sowohl dem ersten als auch dem dritten Blockeingang gebildet.

Unit Delay. Der Unit-Delay-Block gibt für ein Zeitintervall $[t_1,t_2]^T$ an seinem Ausgang die Intervallmenge seines Eingangs für das Zeitintervall $[t_1-1,t_2-1]^T$, das heißt um einen Zeitschritt versetzt, aus. Ist allerdings der erste Zeitschritt einer Modellausführung im betrachteten Zeitintervall enthalten, so fließt auch der Initialwert des Unit-Delay-Blocks in die auszugebende Intervallmenge ein.

Unbekannter Block. In Abschnitt 2.5 wurde an Techniken wie SIA die Anforderung gestellt, dass sie mit unbekannten oder nicht-unterstützten Blocktypen umgehen können müssen. Aus diesem Grund wird eine Intervallsemantik für unbekannte Blöcke definiert. Ein Signal, welches aus einem als unbekannt betrachteten Block ausgeht, erhält dabei den Wertebereich seines Datentyps zugewiesen. Dieses Vorgehen führt zwar

dazu, dass die Unerreichbarkeit mancher CGs unter Umständen nicht erkannt wird - es ist aber alternativlos, es sei denn der Wertebereich eines solchen Signals wird vorab durch eine manuelle Angabe spezifiziert.

Um in den Darstellungen der Intervallsemantik-Definitionen die Übersichtlichkeit zu wahren, wurden zwei Aspekte bewusst weggelassen. Für die berechneten Intervalle findet im Anschluss noch eine Prüfung der Konformität mit dem Datentyp des betreffenden Signals statt. Gegebenenfalls wird ein Intervall korrigiert, das heißt Intervallgrenzen oder Quantisierung angepasst. Zudem enthalten die dargestellten Definitionen keine Behandlung der möglichen Inaktivität eines Blocks sowie der möglichen Zurücksetzung eines Blockzustands. Befindet sich ein Block beispielsweise in einem bedingt ausgeführten Subsystem, so berücksichtigt seine Intervallsemantik auch die Intervallmengen der Kontrollsignale dieses Subsystems.

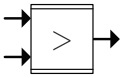
Blocktyp	*Intervallmenge der ausgehenden Signale*
 Constant	$\forall \nu \in \mathbb{N}_{[1,n]}\colon \mathbf{I_S(s,v,ti)} = \{[c^\nu]\}$ **mit** $s = \mathrm{sig}_{\mathrm{out}}(b,1)$
 Relational Operator	$\forall \nu \in \mathbb{N}_{[1,\mathrm{maxd}_{\mathrm{in}}(b)]}$: $\mathbf{I_S(s,v,ti)} = \begin{cases} \{[1]\} & ,\star_T(X,Z) \\ \{[0]\} & ,\star_F(X,Z) \\ \{[0;1]_1\} & ,\text{sonst} \end{cases}$ **mit** $s = \mathrm{sig}_{\mathrm{out}}(b,1) \wedge (1,X),(2,Z) \in I_P(b,ti,\nu)$ **und** $\star_T, \star_F\colon \mathcal{P}(IV) \times \mathcal{P}(IV) \to \mathbb{B}$ $\star_T(X,Z) = \begin{cases} 1\ , & (((\bullet = \text{„}>\text{“}) \vee (\bullet = \text{„}\geqslant\text{“})) \wedge (\min(X) \bullet \max(Z))) \vee \\ & (((\bullet = \text{„}<\text{“}) \vee (\bullet = \text{„}\leqslant\text{“})) \wedge (\max(X) \bullet \min(Z))) \vee \\ & ((\bullet = \text{„}=\text{“}) \wedge (\exists [c] \in X \cup Z\colon \\ & \quad c = \min(X \cup Z) = \max(X \cup Z))) \vee \\ & ((\bullet = \text{„}\neq\text{“}) \wedge (\forall\, iv_1 \in X\colon \forall\, iv_2 \in Z\colon iv_1 \cap iv_2 = \emptyset)) \\ 0\ , & \text{sonst} \end{cases}$ $\star_F(X,Z) = \begin{cases} 1\ , & (((\bullet = \text{„}>\text{“}) \vee (\bullet = \text{„}\geqslant\text{“})) \wedge (\max(X) \circ \min(Z))) \vee \\ & (((\bullet = \text{„}<\text{“}) \vee (\bullet = \text{„}\leqslant\text{“})) \wedge (\min(X) \circ \max(Z))) \vee \\ & ((\bullet = \text{„}=\text{“}) \wedge (\forall\, iv_1 \in X\colon \forall\, iv_2 \in Z\colon iv_1 \cap iv_2 = \emptyset)) \\ & ((\bullet = \text{„}\neq\text{“}) \wedge (\exists [c] \in X \cup Z\colon \\ & \quad c = \min(X \cup Z) = \max(X \cup Z))) \\ 0\ , & \text{sonst} \end{cases}$

Tabelle 6: Intervallsemantik TL-konformer Blocktypen (Auszug)

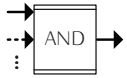
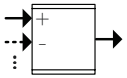
Blocktyp	*Intervallmenge der ausgehenden Signale*
 Logical Operator	**Fall 1:** $n>1_{\vee} \bullet =\neg$ $\forall \nu \in \mathbb{N}_{[1,\max d_{in}(b)]}$: $\mathbf{I_S(s,\nu,ti)} = \begin{cases} \{[1]\} & , \star_T(I_P(b,ti,\nu)) \\ \{[0]\} & , \star_F(I_P(b,ti,\nu)) \\ \{[0;1]_1\} & , \text{sonst} \end{cases}$ **Fall 2:** $n=1_{\wedge} \bullet \neq \neg$ $\mathbf{I_S(s,1,ti)} = \begin{cases} \{[1]\} & , \star_T(I_V(b,ti)) \\ \{[0]\} & , \star_F(I_V(b,ti)) \\ \{[0;1]_1\} & , \text{sonst} \end{cases}$ **mit** $s=\text{sig}_{\text{out}}(b,1)$ **und** $\star_T, \star_F: \mathcal{P}(\mathbb{N}^+ \times \mathcal{P}(IV)) \to \mathbb{B}$ $\star_T(X) = \begin{cases} 1 \text{ , } ((\bullet=\wedge)_{\wedge}(\forall(i,Z)\in X: \forall iv \in Z: 0 \notin iv))_{\vee} \\ \quad ((\bullet=\vee)_{\wedge}(\exists(i,Z)\in X: \forall iv \in Z: 0 \notin iv))_{\vee} \\ \quad ((\bullet=\uparrow)_{\wedge}(\exists(i,Z)\in X: \forall iv \in Z: iv=[0]))_{\vee} \\ \quad (((\bullet=\downarrow)_{\vee}(\bullet=\neg))_{\wedge}(\forall(i,Z)\in X: \forall iv \in Z: iv=[0]))_{\vee} \\ \quad ((\bullet=\oplus)_{\wedge}(\lvert\{i \mid (i,Z)\in X_{\wedge} \forall iv \in Z: 0 \notin iv\}\rvert \bmod 2=1)_{\wedge} \\ \quad (\forall(i,Z)\in X: \forall iv \in Z: 0 \in iv \Rightarrow iv=[0])) \\ 0 \text{ , sonst} \end{cases}$ $\star_F(X) = \begin{cases} 1 \text{ , } ((\bullet=\wedge)_{\wedge}(\exists(i,Z)\in X: \forall iv \in Z: iv=[0]))_{\vee} \\ \quad ((\bullet=\vee)_{\wedge}(\forall(i,Z)\in X: \forall iv \in Z: iv=[0]))_{\vee} \\ \quad (((\bullet=\uparrow)_{\vee}(\bullet=\neg))_{\wedge}(\forall(i,Z)\in X: \forall iv \in Z: 0 \notin iv))_{\vee} \\ \quad ((\bullet=\downarrow)_{\wedge}(\exists(i,Z)\in X: \forall iv \in Z: 0 \notin iv))_{\vee} \\ \quad ((\bullet=\oplus)_{\wedge}(\lvert\{i \mid (i,Z)\in X_{\wedge} \forall iv \in Z: 0 \notin iv\}\rvert \bmod 2=0)_{\wedge} \\ \quad (\forall(i,Z)\in X: \forall iv \in Z: 0 \in iv \Rightarrow iv=[0])) \\ 0 \text{ , sonst} \end{cases}$
 Sum	**Fall 1:** $n>1$ $\forall \nu \in \mathbb{N}_{[1,\max d_{in}(b)]}$: $\mathbf{I_S(s,\nu,ti)} = \star(n,Y,\ldots \star(2,Y,\star(1,Y,\{[0]\}))\ldots)$ **wobei** $Y=I_P(b,ti,\nu)$ **Fall 2:** $n=1$ $\mathbf{I_S(s,1,ti)} = \star(d(b,1),Y,\ldots \star(2,Y,\star(1,Y,\{[0]\}))\ldots)$ **wobei** $Y=I_V(b,ti)$ **In beiden Fällen gilt:** $s=\text{sig}_{\text{out}}(b,1)$ **und** $\star: \mathbb{N}^+ \times (\mathbb{N}^+ \times \mathcal{P}(IV)) \times \mathcal{P}(IV) \to \mathcal{P}(IV)$ $\star(i,Y,X) = \bigcup_{iv_1 \in X} \bigcup_{(i,Z)\in Y} \bigcup_{iv_2 \in Z} \{\circ(iv_1,iv_2,i)\}$ **und** $\circ: IV \times IV \times \mathbb{N}^+ \to IV$ $\circ([a;b],[c;d],i) = \begin{cases} [a+c;b+d] & , \bullet(i)=\text{„+“} \\ [a-d;b-c] & , \bullet(i)=\text{„−“} \end{cases}$ $\circ([a;b]_q,[c;d]_r,i) = \circ([a;b],[c;d],i)_{ggT(q,r)}$

Tabelle 6: Intervallsemantik TL-konformer Blocktypen (Auszug)

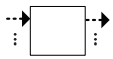
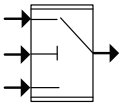
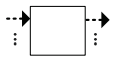
Blocktyp	*Intervallmenge der ausgehenden Signale*
 Inport	**Fall 1:** $\exists sb \in B\colon b \in BC_{sb}$ $\forall \nu \in \mathbb{N}_{[1,d(sb,n)]}\colon \mathbf{I_S}(\mathbf{s_1},\mathbf{v},\mathbf{ti}) = I_S(s_2,\nu,ti)$ **mit** $s_1 = sig_{out}(b,1) \wedge s_2 = sig_{in}(sb,n)$ **Fall 2:** $\forall sb \in B\colon b \notin BC_{sb}$ $\mathbf{I_S}(\mathbf{s},\mathbf{1},\mathbf{ti}) = \{spec_{dom}(b)\}$ **mit** $s = sig_{out}(b,1)$
 Subsystem	$\forall pos \in \mathbb{N}_{[1,\lvert OP_b \rvert]}\colon \forall \nu \in \mathbb{N}_{[1,d(op(b,pos))]}\colon$ $\mathbf{I_S}(\mathbf{s_1},\mathbf{v},\mathbf{ti}) = I_S(s_2,\nu,ti)$ **mit** $s_1 = sig_{out}(b,pos) \wedge s_2 = sig_{in}(ob,1) \wedge ob \in BC_b$ $\wedge\ bt_{ob} = \text{Outport} \wedge (\text{„m“}, pos) \in PV_{ob}$
 Switch	**Fall 1:** $\max(n, d(b,2)) > 1$ $\forall \nu \in \mathbb{N}_{[1,\max(n,d(b,2))]}\colon$ $\mathbf{I_S}(\mathbf{s_0},\mathbf{v},\mathbf{ti}) = \begin{cases} RI(1,ti,\nu) & ,rt_1(\nu) \\ RI(3,ti,\nu) & ,rt_3(\nu) \\ RI(1,ti,\nu) \cup RI(3,ti,\nu) & ,\text{sonst} \end{cases}$ **Fall 2:** $n = d(b,2) = 1$ **Fall 2.1:** $rt_1(1)$ $\forall \nu \in \mathbb{N}_{[1,d(b,1)]}\colon \mathbf{I_S}(\mathbf{s},\mathbf{v},\mathbf{ti}) = RI(1,ti,\nu)$ **Fall 2.2:** $rt_3(1)$ $\forall \nu \in \mathbb{N}_{[1,d(b,3)]}\colon \mathbf{I_S}(\mathbf{s},\mathbf{v},\mathbf{ti}) = RI(3,ti,\nu)$ **Fall 2.3:** $\neg\, rt_1(1) \wedge \neg\, rt_3(1)$ $\forall \nu \in \mathbb{N}_{[1,\max(d(b,1),d(b,3))]}\colon$ $\mathbf{I_S}(\mathbf{s},\mathbf{v},\mathbf{ti}) = RI(1,ti,\nu) \cup RI(3,ti,\nu)$ **mit** $s = sig_{out}(b,1)$ **und** $RI : \mathbb{N}^+ \times TI \times \mathbb{N}^+ \to \mathcal{P}(IV)$ $RI(i,ti,\nu) = X \mid (i,X) \in I_P(b,ti,\nu)$ **und** $rt_1, rt_3 : \mathbb{N}^+ \times TI \to \mathbb{B}$ $rt_1(\nu,ti) = \begin{cases} 1 & ,(\bullet \in \{>,\geq\}_\wedge \min(CI) \bullet t^{\min(n,\nu)}) \\ & \vee (\bullet = \text{„}\neq\text{“}_\wedge \forall iv \in CI : 0 \notin iv) \\ 0 & ,\text{sonst} \end{cases}$ $rt_3(\nu,ti) = \begin{cases} 1 & ,(\bullet \in \{>,\geq\}_\wedge \max(CI) \circ t^{\min(n,\nu)}) \\ & \vee (\bullet = \text{„}\neq\text{“}_\wedge \min(CI) = \max(CI) = 0) \\ 0 & ,\text{sonst} \end{cases}$ **mit** $CI := RI(2,ti,\nu)$
 Unbekannt	$\forall pos \in \mathbb{N}_{[1,\lvert OP_b \rvert]}\colon \forall \nu \in \mathbb{N}_{[1,d(s)]}\colon$ $\mathbf{I_S}(\mathbf{s},\mathbf{v},\mathbf{ti}) = \{range(dt_{op(b,pos)})\}$ **mit** $s = sig_{out}(b,pos)$

Tabelle 6: Intervallsemantik TL-konformer Blocktypen (Auszug)

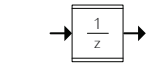
Blocktyp	*Intervallmenge der ausgehenden Signale*
 Unit Delay	$\forall v \in \mathbb{N}_{[1,d(b,1)]}$: $\mathbf{I_S(s_1,v,[t_1,t_2]^T)} = \begin{cases} I_S(s_2,v,[t_1-1,t_2-1]^T) & , t_1>0 \\ \{[a^{\min(v,n)}]\} & , t_1=0 \\ \quad \cup I_S(s_2,v,[0,t_2-1]^T) & \quad \wedge t_2>0 \\ \{[a^{\min(v,n)}]\} & , \text{sonst} \end{cases}$ **mit** $s_1 = sig_{out}(b,1) \wedge s_2 = sig_{in}(b,1)$

Tabelle 6: Intervallsemantik TL-konformer Blocktypen (Auszug)

3.1.2.2 *Intervallpropagierung und Schleifenanalyse*

Die aufgestellte Intervallsemantik findet bei SIA innerhalb einer Intervallpropagierung Anwendung. Ergebnis dieser Propagierung ist, dass jedes Signal des betrachteten Modells für die gesamte Länge einer Modellausführung über Wertebereichsinformationen verfügt. Die Länge einer Modellausführung ist hierbei durch die aus der Modelleingangsspezifikation ableitbare Anzahl der Zeitschritte $spec_{steps}$ gegeben.

SIA erstellt und nutzt für eine Intervallpropagierung die Ausführungsreihenfolge der Blöcke eines Modells, wie sie auch für die konkrete Ausführung eines Modells verwendet wird. Im Vergleich hierzu wird die Ausführungsreihenfolge aber nicht wiederholend Zeitschritt für Zeitschritt angewandt. Die Ausführungsreihenfolge wird zwar auch iterativ angewandt, das heißt sie wird gegebenenfalls mehrfach durchlaufen - allerdings berechnet ein jeder Block hierbei seine Ausgabe für so viele Zeitschritte wie möglich, also nicht zwangsläufig für nur einen Zeitschritt. Hat ein Block seinen Ausgangssignalen Intervallangaben für sämtliche Zeitschritte zugewiesen, so wird dieser Block aus der Ausführungsreihenfolge entfernt und in den folgenden Iterationen nicht weiter betrachtet.

Dieses Vorgehen impliziert die Durchführung einer Schleifenanalyse. Hierzu sei erneut das Beispiel aus Abbildung 8 betrachtet. Der in dem Beispielmodell enthaltene Unit-Delay-Block befindet sich in der Ausführungsreihenfolge vor dem Sum-Block. Wird der Unit-Delay-Block erstmals abstrakt ausgeführt, so bestimmt er seine auszugebende Intervallmenge lediglich für den ersten Zeitschritt, das heißt für das Zeitintervall $[0,0]^T$. Der in der Ausführungsreihenfolge nachfolgende Sum-Block ist nun in der Lage, seine Intervallberechnung ebenfalls für das Zeitintervall $[0,0]^T$ durchzuführen. Der Unit-Delay-Block betrachtet dann bei seiner nächsten abstrakten Ausführung das Zeitintervall $[1,1]^T$. Dieser Zyklus innerhalb des Modells wird so lange abgearbeitet, bis für alle 300 Zeitschritte im Falle des Beispiels Intervallangaben gemacht sind.

Bei jeder seiner abstrakten Ausführungen wird für einen Block also zuerst bestimmt, für welche Zeitintervalle er seine Intervallsemantik überhaupt anwendet. Der Unit-Delay-Block bildet hierbei eine Ausnahme. Denn wie zuvor gesehen, wird für die Intervallbestimmung dieses Blocks ein Zeitintervall gewählt, welches einen Zeitschritt über die an seinem Eingang verfügbaren Zeitintervalle hinausgeht. Für alle gewöhnlichen Blöcke hingegen reichen die am Ausgang betrachteten Zeitintervalle nicht über die an den Blockeingängen vorliegenden Zeitintervalle hinaus. Dieser Umstand ist in Abbildung 9 veranschaulicht. Je abstrakter Ausführung eines Blocks erfolgt die Intervallbestimmung mitunter für mehrere Zeitintervalle. Der Grund: Hat ein Block mehr als einen Eingang, so kann es vorkommen, dass an seinen Eingängen unterschiedliche Zeitintervalle vorliegen. Diese überlappen sich eventuell. Um nicht an Genauigkeit bei der zeitlichen Zuordnung von Wertebereichsangaben einzubüßen, werden daher alle unteren und oberen Grenzen der an den Eingängen anliegenden Zeitintervalle bei der Bildung der Zeitintervalle für den oder die Ausgänge des Blocks berücksichtigt (siehe Abbildung 9).

3.1.2.3 *Intervallvereinigung*

Nachdem ein Block seine Wertebereichsbestimmung für eine oder mehrere Zeitintervalle vorgenommen hat, erfolgt eine sogenannte Intervallvereinigung. Hierbei wird einerseits versucht, die enstandenen Intervallmengen jeweils, ohne Verlust in ihrer Aussagekraft, zu reduzieren. Andererseits werden aufeinanderfolgende Zeitintervalle mit identischen

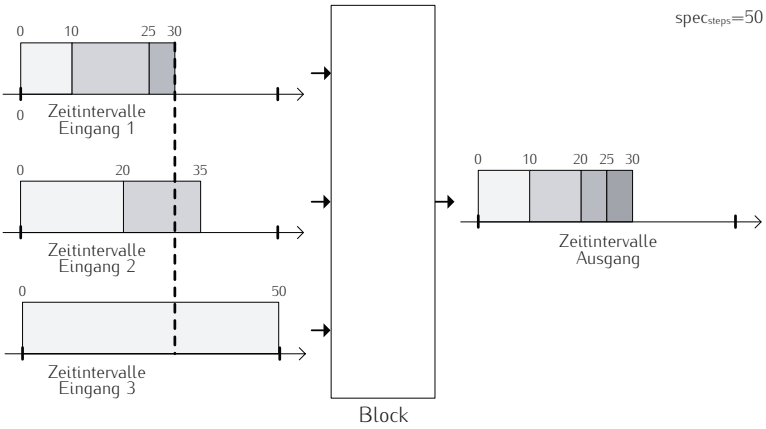


Abbildung 9: Ableitung der Zeitintervalle für das Ausgangssignal eines Blockes von den Zeitintervallen der für den Ausgang relevanten Eingangssignale

Intervallmengen zusammengeführt. Beide Varianten einer Intervallvereinigung sind für die Durchführbarkeit von SIA nicht zwingend notwendig. Sie vereinfachen jedoch die Intervallpropagierung, die einzelnen Intervallbestimmungen und die anschließende Erreichbarkeitsanalyse.

Betrachtet man beispielsweise die Intervallsemantik eines Sum-Blocks (siehe Tabelle 6), in der je Blockeingang eine paarweise Intervallbestimmung erfolgt und hierbei zudem alle möglichen Intervallpaare aus den zwei betrachteten Intervallmengen miteinander verrechnet werden, sollte ersichtlich sein, dass es durchaus zu großen resultierenden Intervallmengen kommen kann. Darüber hinaus sind die Intervallmengen oftmals nicht redundanzfrei, das heißt sie enthalten Intervalle, die – zumindest teilweise – identische Werte(-bereiche) abdecken.

Die daher durchgeführte Intervallvereinigung verfährt wie folgt: Zunächst wird für jedes neu hinzugekommene Zeitintervall eine Reduzierung der zugeordneten Intervallmenge $A \subseteq \mathcal{P}(\mathrm{IV})$ angestrebt. Hierbei wird zwischen der Behandlung einelementiger und mehrelementiger Intervalle unterschieden. Im Anschluss daran werden die Intervallmengen aufeinanderfolgender Zeitintervalle auf Gleichheit überprüft und die Zeitintervalle gegebenenfalls zusammengelegt.

VEREINIGUNG EINELEMENTIGER INTERVALLE

Gegebenenfalls lassen sich einelementige Intervalle zu mehrelementigen Intervallen zusammenfassen. Die Intervallmenge $\{[0], [1], [2], [3]\}$ könnte beispielsweise in $\{[0;3]_1\}$ überführt werden. Hierzu wird folgendes Vorgehen vorgeschlagen. Abbildung 10 dient dabei zur Veranschaulichung. Aus den Werten der in A enthaltenen einelementigen Intervalle wird in einem ersten Schritt ein Distanznetz aufgebaut. Dieses beschreibt den zahlenmäßigen Abstand zwischen jedem Wertepaar. In Abbildung 10 ist ein solches für die Intervallmenge $A = \{[1], [3], [5], [9], [12], [25]\}$ gegeben.

Im zweiten Schritt werden aus diesem Netz Gruppen von Werten abgeleitet. Eine Gruppe besteht hierbei aus Werten, die im Netz über Kanten mit identischen Distanzen verbunden sind. So ergibt sich für das Beispiel unter anderem die Gruppe $\{1, 3, 5\}_2$, wobei 2 die Distanz bezeichnet. Die Gruppen werden in einem dritten Schritt gemäß folgender Kriterien sortiert: nach Gruppengröße (größere Anzahl enthaltener Werte) und, bei identischer Größe, nach Distanz (kleiner vor größer). Die Auflistung der Gruppen für das Beispiel in Abbildung 10 ist bereits nach diesen Vorgaben sortiert.

Im vierten Schritt erfolgt eine Auswahl der Gruppen. Die sortierte Gruppenliste wird hierzu durchgegangen und eine Gruppe selektiert, wenn keiner ihrer enthaltenen Werte Teil einer anderen bereits selektierten Gruppe ist. Für das betrachtete Beispiel werden daher die Gruppen

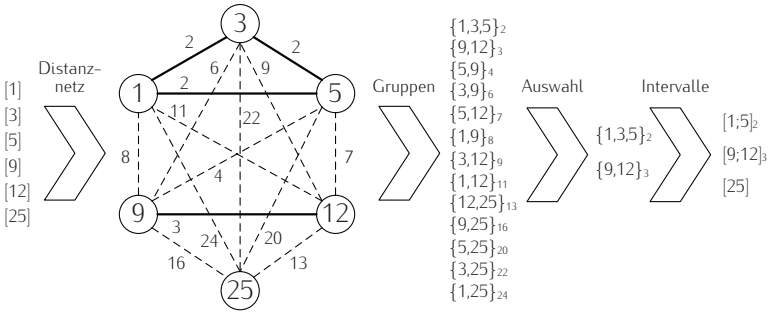


Abbildung 10: Systematische Vereinigung von einelementigen Intervallen (Konstanten) durch Bildung eines Distanznetzes und Ableitung von Konstanten-Gruppen (Beispiel)

$\{1,3,5\}_2$ und $\{9,12\}_3$ ausgewählt. Zum Schluss werden aus den selektierten Gruppen Intervalle gebildet und die darin enthaltenen Werte in der ursprünglichen Intervallmenge ersetzt. Dies bedeutet im Falle des Beispiels: $A' = (A \setminus \{[1], [3], [5], [9], [12]\}) \cup \{[1;5]_2, [9;12]_3\}$

BEHANDLUNG MEHRELEMENTIGER INTERVALLE

Es erfolgt ein paarweiser Vergleich aller Intervalle $[a;b] \in A$ und $[a;b]_q \in A$ untereinander, wobei nach fünf verschiedenen Fällen Ausschau gehalten wird. Der Vergleich sei dabei verallgemeinernd wie folgt definiert.

$$\forall\, iv_1 \in A: \forall\, iv_2 \in A: iv_1 \neq iv_2 \wedge \mathbf{x} \Rightarrow A' = \mathbf{Y} \tag{35}$$

Die jeweilige Fallbedingung ist hierbei für **x** einzusetzen. Die in dem jeweiligen Fall resultierende Intervallmenge entspricht **Y**.

FALL 1 (GLEICHHEIT):

Grundvoraussetzung sind identische Intervallgrenzen:

$lb(iv_1) = lb(iv_2) \wedge ub(iv_1) = ub(iv_2)$

Wenn zudem teilbare Intervall-Quantisierungen angegeben sind:

$q(iv_1) > 0 \wedge q(iv_2) > 0$

$\wedge (\max(q(iv_1), q(iv_2)) \bmod \min(q(iv_1), q(iv_2))) = 0$

Dann gilt für die resultierende Intervallmenge:

$(A \setminus \{iv_1, iv_2\}) \cup \{[\, lb(iv_1); ub(iv_1)\,]_{\min(q(iv_1), q(iv_2))}\}$

FALL 2 (TEILMENGE):

Grundvoraussetzung ist, dass sich das Intervall iv_2 innerhalb der Grenzen von iv_1 befindet: $lb(iv_1) \leqslant lb(iv_2) \wedge ub(iv_1) \geqslant ub(iv_2)$

FALL 2.1:

Wenn die Werte aus iv_2 unter Beachtung der Quantisierungen beider Intervalle tatsächlich in iv_1 enthalten sind:

$q(iv_2) \geqslant q(iv_1) > 0 \wedge (q(iv_2) \bmod q(iv_1)) = 0 \wedge lb(iv_2) \in iv_1$

Dann gilt für die resultierende Intervallmenge: $A \setminus \{iv_2\}$

FALL 2.2:

Wenn für iv_1 keine Quantisierung angegeben ist: $q(iv_1) = 0$

Dann gilt für die resultierende Intervallmenge: $A \setminus \{iv_2\}$

FALL 3 (ÜBERLAPPUNG ODER DIREKTER ANSCHLUSS):

Grundvoraussetzung ist, dass die Intervalle mit Blick auf ihre Grenzen potenziell mindestens einen Wert miteinander teilen:

$lb(iv_1) \leqslant lb(iv_2) \wedge ub(iv_1) \geqslant lb(iv_2)$

FALL 3.1:

Wenn Quantisierungen vorliegen, müssen diese identisch sein und es ist sicherzustellen, dass die Intervalle tatsächlich mindestens einen Wert miteinander teilen:

$q(iv_1) = q(iv_2) > 0 \wedge (ub(iv_1) \bmod q(iv_1)) = (lb(iv_2) \bmod q(iv_1))$

Dann gilt für die resultierende Intervallmenge:

$(A \setminus \{iv_1, iv_2\}) \cup \{[\, lb(iv_1); ub(iv_2) \,]_{q(iv_1)}\}$

FALL 3.2:

Wenn keine Quantisierungen angegeben sind: $q(iv_1) = q(iv_2) = 0$

Dann gilt für die resultierende Intervallmenge:

$(A \setminus \{iv_1, iv_2\}) \cup \{[\, lb(iv_1); ub(iv_2) \,]\}$

FALL 4 (INDIREKTER ANSCHLUSS):

Wenn zwei Intervalle eine identische Quantisierung besitzen und wenn durch einen Schritt mit dieser Quantisierung über die obere Grenze des einen Intervalls hinaus die untere Grenze des anderen Intervalls erreicht wird:

$q(iv_1) = q(iv_2) > 0 \wedge ub(iv_1) + q(iv_1) = lb(iv_2)$

Dann gilt für die resultierende Intervallmenge:

$(A \setminus \{iv_1, iv_2\}) \cup \{[\, lb(iv_1); ub(iv_2) \,]_{q(iv_1)}\}$

Für jeden Vergleich zweier Intervalle aus einer Intervallmenge A gilt: Sobald einer der aufgeführten Fälle eintritt und eine modifizierte Intervallmenge A' entsteht, wird der Vereinigungsvorgang (siehe Definition 35) abgebrochen und für A' erneut gestartet. Auf diese Weise werden auch transitive Vereinigungsmöglichkeiten identifiziert. Dieses Prozedere endet, sobald in einer Intervallmenge keiner der gelisteten Fälle auftritt.

VEREINIGUNG VON ZEITINTERVALLEN

Die für Zeitintervalle eines Signals berechneten Intervallmengen werden zuerst nach dem zuvor beschriebenen Schema minimiert. Anschließend wird geprüft, ob die Intervallmengen A_1 und A_2 zweier aufeinanderfolgender Zeitintervalle $[t_1, t_2]^T$ und $[t_2{+}1, t_3]^T$ identisch sind. Zwei Intervallmengen werden hierbei genau dann als identisch angesehen, wenn sie die gleiche Anzahl an Intervallen beinhalten und jedes Intervall aus der einen Menge auch in der anderen Menge enthalten ist. Ist dies der Fall, so werden die zwei Zeitintervalle durch ein neues Zeitintervall $[t_1, t_3]^T$ ersetzt und diesem die Intervallmenge A_1 zugeordnet.

SIA beschränkt sich an dieser Stelle auf identische Intervallmengen und vernachlässigt gleiche Intervallmengen. Zwei Intervallmengen sind genau dann gleich, wohlgemerkt aber nicht identisch, wenn alle in den Intervallen einer der Intervallmengen enthaltenen Werte jeweils auch in mindestens einem Intervall der anderen Intervallmenge enthalten sind. Die Intervallmengen $\{[1,2]_1, [4,5]_1\}$ und $\{[1,4]_3, [2,5]_3\}$ sind beispielsweise gleich. Ein Vorgehen zur Erkennung der Gleichheit vorausgesetzt, könnten auch die Zeitintervalle gleicher Intervallmengen vereinigt werden.

3.1.3 MODELLTRANSFORMATIONEN

Die durch die abstrakte Intervall-Interpretation ermittelten Signalwertebereiche machen es in bestimmten Fällen möglich, die durch SIA betrachtete Repräsentation eines SL/TL-Modells zu vereinfachen. Diese (gegebenenfalls vereinfachte) Modellrepräsentation wird nicht nur durch SIA, sondern auch durch die anderen in dieser Arbeit vorgestellten statischen Voranalysen sowie durch den SMT-Testdatengenerierungsansatz verwendet. Wie in den Abschnitten zu diesen Techniken ersichtlich wird, können sich Vereinfachungen der Modellrepräsentation im Rahmen von SIA vorteilhaft auf die Effizienz und die erzielten Ergebnisse der anderen Techniken auswirken. Da die Modelltransformationen im Rahmen von SIA und auch in ihrer Auswirkung auf die anderen Techniken dennoch eine untergeordnete Rolle einnehmen, wird in den folgenden Zeilen auf eine formale und umfassende Betrachtung der Transformationen verzichtet.

Die zur Vereinfachung der Modellrepräsentation durchgeführten Modelltransformationen erfolgen bei SIA während der Intervallpropagierung. Wird ein Block abstrakt ausgeführt, so wird zunächst geprüft, ob die Zeitintervalle aller seiner Eingänge bereits vollständig sind. Vollständig bedeutet, dass die Zeitintervalle sämtliche betrachtete Zeitschritte abdecken. Ist dies der Fall, so erfolgt vor der Intervallberechnung eine Prüfung von Transformationsbedingungen. Diese Transformationsbedingungen

sind Block-spezifisch und betrachten die an den Eingängen anliegenden Intervallmengen. Trifft eine Transformationsbedingung zu, so wird die Modellrepräsentation durch Anwendung von Transformationsoperationen transformiert. Die nachfolgende Intervallbestimmung für die Blockausgänge wird, je nach der Art der Transformationsoperationen, anschließend durchgeführt oder auch nicht. Wird sie durchgeführt, so werden auch nach ihr noch weitere Transformationsbedingungen geprüft und gegebenenfalls Transformationsoperationen angewandt. Die Transformationsbedingungen betrachten hierbei allerdings die Intervallmengen der Blockausgänge. Zudem sind diese nicht Block-spezifisch. In jedem Fall gilt aber auch hier die Vorbedingung der Zeitintervall-Vollständigkeit.

TRANSFORMATIONSOPERATIONEN

Folgende drei Arten von Transformationsoperationen werden angewandt:

1. Ersetzung des betrachteten Blocks durch einen anderen Block oder mehrere andere Blöcke
2. Entfernung des betrachteten Blocks und Vereinigung seiner Ein- und Ausgangssignale
3. Entfernung ausgewählter Eingangsports des betrachteten Blocks

In allen drei Fällen kann es sein, dass ein im Modell rückwärtsgerichteter Entfernungsprozess angestoßen wird. Dieser entfernt Signale und Blöcke, die zu keinem anderen Block mehr führen und die in keinem Zusammenhang mit einem CG stehen. Der Entfernungsprozess geht sogar so weit, dass beispielsweise Signale ohne Zielport oder Ausgangsports ohne Signale existieren können. Die Umsetzungen von SIA und den anderen in dieser Arbeit vorgestellten Techniken können mit derart unvollständigen Modellrepräsentationen allerdings umgehen.

Mit dem Entfernen und Hinzufügen von Blöcken gilt es selbstverständlich auch, die Ausführungsreihenfolge der laufenden Intervallpropagierung zu aktualisieren.

TRANSFORMATIONSBEDINGUNGEN

Die Transformationsbedingungen, welche die Intervallmengen der Blockeingänge betrachten, können vielfältiger Natur sein. Für fast jeden für SL/TL-Modelle verfügbaren Blocktypen lassen sich Überlegungen anstellen, wie sich die Gestaltung der Wertebereiche an den Blockeingängen auf die Blockfunktionalität auswirkt und ob sich hieraus Vereinfachungen im Modell ableiten lassen. Folgend seien drei Beispiele aufgeführt:

- **Logical Operator:** Setzt ein solcher Block beispielsweise den AND-Operator ein und gibt es einen Eingang, dessen Intervallmengen aller vorhandener Zeitintervalle durchgängig ausschließlich das Intervall [1] beinhalten, so kann dieser Eingang entfernt werden. Bleibt hierdurch nur ein Eingang übrig, so bietet es sich an, den Block durch einen Relational-Operator-Block (mit Ungleich-Operator) zu ersetzen. In diesen Block geht dann zum einen das verbliebene Eingangssignal des AND-Blocks ein. Zum anderen wird ein Constant-Block mit der Konstanten 0 hinzugefügt und mit dem Relational-Operator-Block verbunden.
- **Abs:** Ist der Wertebereich des Blockeingangs durchgehend nichtnegativ, so kann der Block entfernt werden. Das Ein- und Ausgangssignal werden vereinigt.
- **Switch:** Falls sich der Wertebereich des zweiten Eingangs derart gestaltet, dass für alle Zeitintervalle eine ausschließliche Weiterleitung des ersten oder dritten Eingangs zum Ausgang erfolgt, kann der Block ebenfalls entfernt werden. Das Signal des ersten oder dritten Eingangs wird dann mit dem Ausgangssignal vereinigt.

Für die Intervallmengen der Ausgänge eines Blocks berücksichtigt SIA eine Transformationsbedingung, die für alle Blocktypen gültig ist. Zur Erfüllung dieser Bedingung muss für jeden Blockausgang Folgendes gelten: die Intervallmengen aller Zeitintervalle enthalten ausschließlich ein einelementiges Intervall mit dem gleichen Wert. Handelt es sich um ein Vektorsignal, so muss diese Eigenschaft je Vektorelement gelten. Ist die Bedingung erfüllt, so wird der betrachtete Block durch einen Constant-Block oder mehrere Constant-Blöcke ersetzt - je nachdem wie viele Ausgänge der betrachtete Block hat.

3.1.4 ERREICHBARKEITSANALYSE

Ist die Intervallpropagierung beendet, so liegen die Wertebereichsangaben für alle Modellsignale vor. SIA überprüft dann die Erreichbarkeit der CGs. Die diesem Zweck dienende Erreichbarkeitsanalyse erfolgt in drei Schritten:

1. Ersetzung von Signalen in CG-Ausdrücken durch Konstanten
2. Normierung der CG-Ausdrücke
3. Prüfung der CG-Erreichbarkeit

Der erste Schritt betrifft lediglich die Signale, welche durch die Intervallbestimmung als konstant identifiziert wurden. Ein Signal wird im Kontext von SIA als konstant bezeichnet, wenn die Intervallmengen aller Zeitintervalle ausschließlich ein einelementiges Intervall mit dem

gleichen Wert beinhalten. Ist ein Signal konstant, so werden alle Vorkommnisse dieses Signals in CG-Ausdrücken durch den entsprechenden konstanten Wert ersetzt. Im zweiten Schritt erfolgt eine Sortierung innerhalb der CG-Ausdrücke, um die anschließende Erreichbarkeitsprüfung zu erleichtern. Hierbei werden insbesondere Vergleiche von Signalen mit Konstanten in eine einheitliche Ordnung gebracht. Aus dem Ausdruck $5>s$ wird beispielsweise $s<5$. Die im dritten Schritt durchgeführte Erreichbarkeitsprüfung wird im Folgenden detailliert vorgestellt.

Jedes CG ist durch seinen Ausdruck gegeben. Wie in Definition 14 angegeben, kann ein CG aus Teilausdrücken bestehen. Sowohl CG-Ausdrücke als auch darin enthaltene Teilausdrücke seien in den folgenden Ausführungen verallgemeinernd als $\varphi \in E$ bezeichnet. Jedes φ besitzt einen Zustand, der angibt, ob φ unerreichbar ist (Zustand UNSAT), ob φ unabhängig von den bei einer Modellausführung verwendeten Testdaten stets erreicht ist (Zustand SAT) oder ob φ nur bei Verwendung passender Testdaten erreicht werden kann (Zustand OPEN). Den Zustand gibt die Abbildung st wieder:

$$\mathrm{st} : E \to \{\mathrm{UNSAT}, \mathrm{SAT}, \mathrm{OPEN}\} \tag{36}$$

Wie in den folgenden Definitionen ersichtlich wird, ist es für ein φ mitunter relevant, zu welchen Zeitschritten einer potentiellen Modellausführung der Zustand SAT oder UNSAT eintritt. Die Abbildung $S@(\varphi)$ gibt daher die Menge der Zeitschritte an, welche für den Zustand SAT ursächlich sind. Analog hierzu ist $U@(\varphi)$ für UNSAT zu sehen.

$$S@, U@ : E \to \mathcal{P}(\mathbb{N}) \tag{37}$$

Voraussetzung zur Anwendung von $S@(\varphi)$ ist allerdings, dass $\mathrm{st}(\varphi) = \mathrm{SAT}$ gilt. Denn nur dann ist $S@(\varphi) \neq \emptyset$. Für $U@(\varphi)$ gilt eine solche Einschränkung nicht, da es auch Zeitschritte enthalten kann, bei denen φ unerfüllbar ist, ohne dass jedoch φ dadurch insgesamt unerfüllbar wird. Hierzu müsste φ nämlich zu jedem betrachteten Zeitschritt unerfüllbar sein.

Zudem sei AT die Menge aller Zeitschritte einer Modellausführung:

$$\mathrm{AT} = \mathbb{N}_{[0, \mathrm{spec}_{\mathrm{steps}} - 1]} \tag{38}$$

Der Zustand von φ ergibt sich unter Verwendung der Signalwertebereiche wie im Folgenden dargestellt. Es wird hierbei zwischen den verschiedenen Formen, die φ haben kann, unterschieden.

RELATIONALER AUSDRUCK MIT ZWEI KONSTANTEN

Kommt es im ersten Schritt der Erreichbarkeitsanalyse zur Ersetzung von Signalen mit Konstanten in CG-Ausdrücken, so können sich Ausdrücke der Form $\varphi = c_1 \bullet c_2$ mit $\bullet \in \{>, \geqslant, <, \leqslant, =, \neq\}$ und $c_1, c_2 \in \mathbb{R}$ ergeben. Der

Zustand von φ sowie die Mengen der Zeitschritte, zu denen φ stets erfüllt und nicht erfüllbar ist, werden folgendermaßen definiert. Da keine Signale in den Ausdruck involviert sind, spielen Signalwertebereiche an dieser Stelle keine Rolle.

$$\begin{aligned}
\mathrm{st}(c_1 \bullet c_2) &= \begin{cases} \mathrm{SAT} & , c_1 \bullet c_2 \\ \mathrm{UNSAT} & , \text{sonst} \end{cases} \\
\mathrm{S@}(c_1 \bullet c_2) &= \begin{cases} \mathrm{AT} & , \mathrm{st}(c_1 \bullet c_2) = \mathrm{SAT} \\ \emptyset & , \text{sonst} \end{cases} \\
\mathrm{U@}(c_1 \bullet c_2) &= \begin{cases} \mathrm{AT} & , \mathrm{st}(c_1 \bullet c_2) = \mathrm{UNSAT} \\ \emptyset & , \text{sonst} \end{cases}
\end{aligned} \tag{39}$$

Der Zustand ergibt sich einzig aus dem Zutreffen des relationalen Ausdrucks. Je nach Auswertung zu *wahr* oder *falsch* enthalten $\mathrm{S@}(\varphi)$ oder $\mathrm{U@}(\varphi)$ alle betrachteten Zeitschritte.

RELATIONALER AUSDRUCK MIT SIGNAL UND KONSTANTE

Für die Verknüpfung eines Signals $s \in S$ (Vektorindex $v \in \mathbb{N}_{[1,\,d(s)]}$) mit einer Konstanten $c \in \mathbb{R}$ über einen relationalen Operator $\bullet \in \{>, \geqslant, <, \leqslant, =, \neq\}$ greift die Zustandsdefinition auf den Wertebereich des Signals zurück:

$$\mathrm{st}(s^v \bullet c) = \begin{cases} \mathrm{SAT} & , \exists\, \mathrm{ti} \in \mathrm{T}(s,v)\colon \circ(\mathrm{I_S}(s,v,\mathrm{ti}),c) \\ \mathrm{UNSAT} & , \forall\, \mathrm{ti} \in \mathrm{T}(s,v)\colon \star(\mathrm{I_S}(s,v,\mathrm{ti}),c) \\ \mathrm{OPEN} & , \text{sonst} \end{cases}$$

$$\mathrm{S@}(s^v \bullet c) = \bigcup_{\substack{[t_1,t_2]^\mathsf{T} \in \mathrm{T}(s,v)\colon \\ \circ(\mathrm{I_S}(s,v,[t_1,t_2]^\mathsf{T}),c)}} \mathbb{N}_{[t_1,t_2]} \tag{40}$$

$$\mathrm{U@}(s^v \bullet c) = \bigcup_{\substack{[t_1,t_2]^\mathsf{T} \in \mathrm{T}(s,v)\colon \\ \star(\mathrm{I_S}(s,v,[t_1,t_2]^\mathsf{T}),c)}} \mathbb{N}_{[t_1,t_2]}$$

Die Funktionen $\star, \circ : \mathcal{P}(\mathrm{IV}) \times \mathbb{R} \to \mathbb{B}$ prüfen für die Intervallmenge eines Zeitintervalls ti und die in φ enthaltene Konstante c, ob die Intervalle dieser Intervallmenge dazu führen, dass φ in ti unerfüllbar oder stets erfüllt ist. $\mathrm{S@}(\varphi)$ und $\mathrm{U@}(\varphi)$ sammeln jeweils die Zeitschritte der Zeitintervalle, in denen einer dieser beiden Fälle gilt.

$$\circ(X,c) = \begin{cases} \min(X) \bullet c & , \bullet \in \{>, \geqslant\} \\ \max(X) \bullet c & , \bullet \in \{<, \leqslant\} \\ \min(X) = \max(X) = c & , \bullet = \text{„}=\text{“} \\ \forall\, \mathrm{iv} \in X\colon c \notin \mathrm{iv} & , \bullet = \text{„}\neq\text{“} \end{cases} \tag{41}$$

$$\star(X,c)=\begin{cases} c\geqslant \max(X) & ,\bullet=„>“ \\ c>\max(X) & ,\bullet=„\geqslant“ \\ c\leqslant \min(X) & ,\bullet=„<“ \\ c<\min(X) & ,\bullet=„\leqslant“ \\ \forall\, iv\in X\colon c\notin iv & ,\bullet=„=“ \\ \min(X)=\max(X)=c & ,\bullet=„\neq“ \end{cases}$$

Ginge es beispielsweise um den Ausdruck $s>5$, so ist dieser unerfüllbar, wenn der größtmögliche Wert aus dem Wertebereich von s in jedem Zeitintervall nicht größer als die Konstante 5 ist.

RELATIONALER AUSDRUCK MIT ZWEI SIGNALEN

Für die Verknüpfung zweier Signale $s_1,s_2\in S$ (Vektorindizes $v\in\mathbb{N}_{[1,d(s_1)]}$ und $w\in\mathbb{N}_{[1,d(s_2)]}$) über einen relationalen Operator $\bullet\in\{>,\geqslant,<,\leqslant,=,\neq\}$ leitet sich der Zustand aus den Wertebereichen beider Signale ab:

$$st(s^v\bullet s^w)=\begin{cases} \text{SAT} & ,\exists[t_1,t_2]^T\in T(s_1,v)\colon\exists[t_3,t_4]^T\in T(s_2,w)\colon \\ & \quad (t_1\leqslant t_4\wedge t_3\leqslant t_2)\Rightarrow \circ(G,H) \\ \text{UNSAT} & ,\forall[t_1,t_2]^T\in T(s_1,v)\colon\forall[t_3,t_4]^T\in T(s_2,w)\colon \\ & \quad (t_1\leqslant t_4\wedge t_3\leqslant t_2)\Rightarrow \star(G,H) \\ \text{OPEN} & ,\text{sonst} \end{cases}$$

$$S@(s^v\bullet s^w)=\bigcup_{\substack{[t_1,t_2]^T\\ \in T(s_1,v)}}\ \bigcup_{\substack{[t_3,t_4]^T\in T(s_2,w)\colon\\ t_1\leqslant t_4\wedge t_3\leqslant t_2\\ \wedge\circ(G,H)}}\mathbb{N}_{[\max(t_1,t_3),\min(t_2,t_4)]} \tag{42}$$

$$U@(s^v\bullet s^w)=\bigcup_{\substack{[t_1,t_2]^T\\ \in T(s_1,v)}}\ \bigcup_{\substack{[t_3,t_4]^T\in T(s_2,w)\colon\\ t_1\leqslant t_4\wedge t_3\leqslant t_2\\ \wedge\star(G,H)}}\mathbb{N}_{[\max(t_1,t_3),\min(t_2,t_4)]}$$

Hierbei bezeichnen $G=I_S(s_1,v,[t_1,t_2]^T)$ und $H=I_S(s_2,w,[t_3,t_4]^T)$ die Intervallmengen eines Zeitintervalls der beiden Signale. Die Hilfsfunktionen $\star,\circ:\mathcal{P}(IV)\times\mathcal{P}(IV)\to\mathbb{B}$ geben für zwei Intervallmengen an, ob diese zu einer Unerfüllbarkeit oder garantierten Erfüllung des relationalen Ausdrucks führen. $S@(\varphi)$ und $U@(\varphi)$ sammeln auch hier jeweils die Zeitschritte, in denen einer dieser beiden Fälle eintritt.

$$\circ(X,Y)=\begin{cases} \min(X)\bullet\max(Y) & ,\bullet\in\{>,\geqslant\} \\ \max(X)\bullet\min(Y) & ,\bullet\in\{<,\leqslant\} \\ \min(X)=\min(Y)=\max(X)=\max(Y) & ,\bullet=„=“ \\ \forall\, iv_1\in X\colon\forall\, iv_2\in Y\colon iv_1\cap iv_2=\emptyset & ,\bullet=„\neq“ \end{cases} \tag{43}$$

$$\star(X,Y)=\begin{cases}\min(Y)\geqslant\max(X) & ,\bullet=\text{„}>\text{“}\\ \min(Y)>\max(X) & ,\bullet=\text{„}\geqslant\text{“}\\ \max(Y)\leqslant\min(X) & ,\bullet=\text{„}<\text{“}\\ \max(Y)<\min(X) & ,\bullet=\text{„}\leqslant\text{“}\\ \forall iv_1\in X:\forall iv_2\in Y:\, iv_1\cap iv_2=\emptyset & ,\bullet=\text{„}=\text{“}\\ \min(X)=\min(Y)=\max(X)=\max(Y) & ,\bullet=\text{„}\neq\text{“}\end{cases}$$

Auch hierzu ein Beispiel: Der Ausdruck $s_1=s_2$ ist unerfüllbar, wenn die Schnittmenge der Signalwertebereiche von s_1 und s_2 leer ist, das heißt die Wertebereiche keinen Wert gemeinsam haben.

LOGISCHER AUSDRUCK

Besteht φ aus einer logischen Verundung von $n\geqslant 2$ Teilausdrücken φ_i (mit $i\in\mathbb{N}_{[1,n]}$), so gilt für den Zustand:

$$st(\bigwedge_{i=1}^{n}\varphi_i)=\begin{cases}\text{UNSAT} & ,\exists j\in\mathbb{N}_{[1,n]}:\, st(\varphi_j)=\text{UNSAT}\\ \text{SAT} & ,(\forall j\in\mathbb{N}_{[1,n]}:\, st(\varphi_j)=\text{SAT})\wedge\bigcap_{k=1}^{n}S@(\varphi_k)\neq\emptyset\\ \text{OPEN} & ,\text{sonst}\end{cases}$$

$$S@(\bigwedge_{i=1}^{n}\varphi_i)=\bigcap_{j=1}^{n}S@(\varphi_j) \tag{44}$$

$$U@(\bigwedge_{i=1}^{n}\varphi_i)=\bigcup_{\substack{j=1,\\ st(\varphi_j)=\text{UNSAT}}}^{n}U@(\varphi_j)$$

Existiert also ein unerfüllbarer Teilausdruck, so ist die Verundung aller Teilausdrücke ebenfalls unerfüllbar. Der Gesamtausdruck ist hingegen stets erfüllt, wenn alle seine Teilausdrücke stets erfüllt sind.

Im Falle einer logischen Veroderung von $n\geqslant 2$ Teilausdrücken φ_i (mit $i\in\mathbb{N}_{[1,n]}$) gilt für den Zustand:

$$st(\bigvee_{i=1}^{n}\varphi_i)=\begin{cases}\text{UNSAT} & ,\forall j\in\mathbb{N}_{[1,n]}:\, st(\varphi_j)=\text{UNSAT}\\ \text{SAT} & ,\exists j\in\mathbb{N}_{[1,n]}:\, st(\varphi_j)=\text{SAT}\\ \text{OPEN} & ,\text{sonst}\end{cases}$$

$$S@(\bigvee_{i=1}^{n}\varphi_i)=\bigcup_{\substack{j=1,\\ st(\varphi_j)=\text{SAT}}}^{n}S@(\varphi_j) \tag{45}$$

$$U@(\bigvee_{i=1}^{n}\varphi_i)=\bigcap_{j=1}^{n}U@(\varphi_j)$$

Anders als bei der Verundung müssen alle Teilausdrücke unerfüllbar sein, damit der Gesamtausdruck unerfüllbar ist. Existiert hingegen mindestens ein stets erfüllter Teilausdruck, so ist der Gesamtausdruck automatisch auch stets erfüllt.

Der Zustand einer logischen Negation $\varphi = \neg\varphi'$ sei wie folgt definiert:

$$\mathrm{st}(\neg\varphi') = \begin{cases} \text{UNSAT} & , \mathrm{st}(\varphi') = \text{SAT} \wedge |\,\mathrm{S@}(\varphi')| = \mathrm{spec}_{\mathrm{steps}} \\ \text{SAT} & , \mathrm{U@}(\varphi') \neq \emptyset \\ \text{OPEN} & , \text{sonst} \end{cases}$$

$$\mathrm{S@}(\neg\varphi') = \mathrm{U@}(\varphi') \tag{46}$$

$$\mathrm{U@}(\neg\varphi') = \mathrm{S@}(\varphi')$$

Der Ausdruck φ ist unerfüllbar, wenn φ' zu allen betrachteten Zeitschritten stets erfüllt ist. Existiert hingegen mindestens ein Zeitschritt, zu dem φ' unerfüllbar ist, so ist φ als stets erfüllt anzusehen.

Mit dieser Betrachtung ist die Erreichbarkeitsanalyse von SIA vollumfänglich beschrieben. Zusammenfassend ist zu betonen, dass SIA durch seine abstrakte Intervall-Analyse eines SL/TL-Modells, gepaart mit der in diesem Abschnitt vorgestellten Erreichbarkeitsanalyse, dazu in der Lage ist, eine Teilmenge unerreichbarer CGs zu identifizieren. Darüber hinaus erkennt SIA auch CGs, welche unabhängig von der Testdatenwahl erfüllt sind. Beide Kategorien von CGs können bei einer Testdatengenerierung, ganz gleich ob ein suchbasiertes Verfahren oder ein anderer Ansatz zum Einsatz kommen, ausgeschlossen werden. Der in dieser Arbeit verwendete suchbasierte Ansatz sollte seine Effizienz durch eine vorherige Anwendung von SIA steigern können – vorausgesetzt, für das betrachtete Modell existieren unerreichbare CGs und der Aufwand von SIA selbst hält sich, wie aufgrund der Vorgehensweise dieser Technik zu vermuten ist, in einem überschaubaren Rahmen. Die in Kapitel 6 vorgestellten Fallstudien untersuchen neben der Auswirkung von SIA auf die Testdatengenerierung auch diesen Aspekt.

3.1.5 VERWANDTE ARBEITEN

Die Durchführung einer Intervallanalyse, das heißt die Betrachtung der Wertebereiche von Programmvariablen, stellt seit den Anfängen der abstrakten Interpretation [30] eine algorithmische Herausforderung dar [45]. Die statische Interpretation eines Programms ist aus Komplexitätsgründen meist an Abstraktionstechniken oder Abschätzungen gebunden. Genauigkeit und Optimalität derartiger Analysen stehen im Allgemeinen im Konflikt mit der Durchführbarkeit derselben. Eine Vielzahl an Arbeiten beschäftigt sich daher mit Intervallanalysen. Insbesondere im

Bereich des Compilerbaus, der sich dem Entwurf von Übersetzern (englisch: Compiler) für die Wandlung von Programmcode in ausführbaren Maschinencode widmet, finden Intervallanalysen Anwendung. Wertebereichsinformationen werden dabei vielfältig genutzt, beispielsweise zur Erkennung von totem, also nicht-erreichbarem Code [23, 35].

Intervalle als Teil einer abstrakten Interpretation nutzt beispielsweise das Werkzeug ASTRÉE [32]. Dieses dient der Ermittlung von Laufzeitfehlern in C-Programmen. Geht es um den Anwendungsfall der vorliegenden Arbeit, die Erkennung unerreichbarer Zustände, so existieren wohlgemerkt auch Ansätze, die nicht auf eine abstrakte Interpretation bauen. Goldberg et al. prüfen die Erreichbarkeit von Programmpfaden zum Beispiel mithilfe von Constraint-Solving-Techniken und durch Theorembeweis [50]. Im betrachteten Kontext, der Anwendung für SL/TL-Modelle, ist die Skalierbarkeit eines solchen Ansatzes allerdings als kritisch zu erachten (siehe Abschnitt 2.3.1).

Der Codegenerator TL sowie aktuelle Versionen des SL-Werkzeugs betrachten ebenfalls die Wertebereiche der Signale innerhalb eines SL- oder SL/TL-Modells. Die Analysen dienen hierbei der Optimierung des generierten Codes und der Ermittlung geeigneter Signal-Datentypen und -Skalierungen für die Codegenerierung. Die verwendete Technik ist in beiden Fällen allerdings nicht veröffentlicht. Es ist allerdings zu erkennen, dass die Fähigkeiten der Werkzeuge zur Wertebereichserkennung begrenzt sind. Im Falle von Schleifen beispielsweise kann der Nutzer Angaben zu den Wertebereichen machen. Dies spricht dafür, dass keine Schleifenanalyse, wie sie SIA enthält, erfolgt. Neben diesen Funktionalitäten in SL und TL existiert des Weiteren das Werkzeug Polyspace [115] zur statischen Analyse von C-Code. Polyspace ist in der Lage, Wertebereichsangaben für Programmvariablen zu machen. Selbstverständlich lässt sich das Werkzeug auch für den aus einem SL/TL-Modell generierten C-Code einsetzen. Seine statischen Analysen führt Polyspace allerdings nicht im Modell durch.

Die am nächsten mit SIA verwandte Arbeit stammt von Wang et al. [125]. Obwohl sie sich mit der Anwendung für Java-Programme und nicht etwa für SL/TL-Modelle oder andere verwandte Modellsprachen beschäftigt, so steht ihr Ansatz der SIA-Technik bezüglich Vorgehen und Ausrichtung doch sehr nahe. Wang et al. verwenden eine abstrakte Interpretation mittels Intervallen, um unerreichbare Pfade in Java-Programmen zu ermitteln. Hierbei setzen sie Intervallmengen je Programmvariable ein. Zudem definieren sie Mengenoperationen und die grundlegenden mathematischen Operationen für den Umgang mit Intervallmengen. SIA baut auf diesen Grundlagen auf und zeichnet sich durch folgende Eigenschaften und Beiträge aus:

1. Verwendung einer Spezifikation der Eingänge (Modelleingangsspezifikation), um präzisere Wertebereichsangaben (mit engeren Grenzen) zu erhalten
2. Definition einer Intervallsemantik für SL/TL-Blöcke
3. Erweiterung des Intervallmengen-Konzepts von Wang et al. um zeitliche Dynamik (Zuordnung über Zeitintervalle)
4. Intervallpropagierung mit Schleifenanalyse unter Berücksichtigung der Ausführungssemantik eines SL-Modells
5. Modelltransformationen innerhalb der Intervallpropagierung

Zum letzten Punkt ist zu ergänzen, dass es Arbeiten gibt, die sich explizit mit der Transformation von SL-Modellen auseinandersetzen – beispielsweise von Tran et al. [119]. Der Sinn und Zweck von Transformationen liegt bei SIA allerdings nicht in der Unterstützung und Teilautomatisierung der Modellbildung. Modelltransformationen werden in SIA durch Eigenheiten der ermittelten Wertebereiche getriggert und dienen der Vereinfachung einer Modellrepräsentation für an SIA anschließende Analysen. Das zugrunde liegende SL/TL-Modell wird von SIA dabei zudem nicht abgeändert.

3.2 MODELLEINGANG–ABHÄNGIGKEITS–ERMITTLUNG

Die zweite der drei in dieser Arbeiten vorgestellten statischen Voranalysen wird mit dem Kürzel SDA bezeichnet. SDA steht für die englische Bezeichnung *Signal Dependency Analysis* (zu Deutsch: Signalabhängigkeitsanalyse). In diesem Abschnitt werden die Zielsetzung und das Vorgehen von SDA beschrieben. Eine Einordnung der Technik im Vergleich mit verwandten Arbeiten erfolgt im Anschluss.

3.2.1 ZIEL

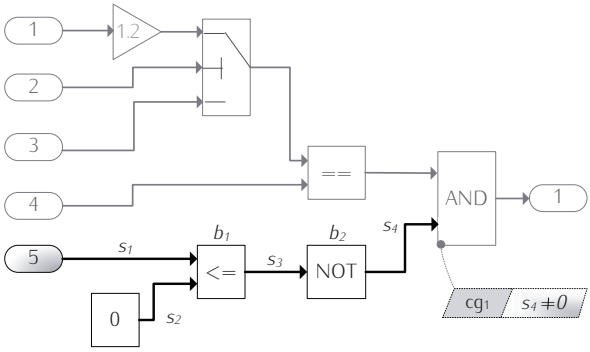
Das Problem der Größe des bei einem suchbasierten Test abzusuchenden Suchraums wurde bereits in Abschnitt 2.5 erörtert. Die Anzahl der Dimensionen des Suchraums wächst im betrachteten Anwendungsbereich maßgeblich mit der Anzahl der Modelleingänge, in zweiter Instanz allerdings auch mit den in der Modelleingangsspezifikation gemachten Angaben. Hierbei kommen insbesondere die spezifizierte zeitliche Länge der Signale sowie Parameter, welche sich auf die Anzahl der Signalsegmente auswirken, zum Tragen. Die Modelleingangsspezifikation stellt so gesehen bereits ein Instrument dar, mit dem manuell, durch Expertenwissen über das Testobjekt, Einschränkungen des Suchraums getroffen werden können.

Einen Umstand adressiert die Modelleingangsspezifikation allerdings nicht: Die Erfüllung der meisten CGs hängt zwar von einer geeigneten Testdatenwahl ab, allerdings gibt es im Falle vieler CGs Modelleingänge, deren Belegung mit wie auch immer gearteten Signalen keinerlei Einfluss auf die Erfüllung des CG nimmt. Derartige Modelleingänge sind irrelevant für die Testdatensuche eines bestimmten CG. Das eingesetzte Suchverfahren besitzt allerdings keinerlei Kenntnis von solch einer Tatsache. Es erzeugt nicht nur einmalig Signale für unter Umständen irrelevante Modelleingänge, sondern die Operatoren des Suchverfahrens konzentrieren sich zum Zwecke der Suche auch auf Änderungen dieser Signale. Dieses Vorgehen ist der Effizienz einer suchbasierten Testdatengenerierung nicht zuträglich.

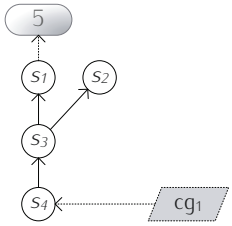
Die SDA-Technik ist in der Lage, festzustellen, welche Modelleingänge für die Erfüllung eines CG irrelevant sind. Wie der Name der Technik verrät, analysiert sie hierzu Abhängigkeiten zwischen den Signalen eines SL/TL-Modells. Das Vorgehen dieser Analyse und die Miteinbeziehung der Analyseergebnisse in die Testdatensuche werden im Folgenden behandelt.

3.2.2 TECHNIK

Ein motivierendes Beispiel soll als Einstieg in die folgenden Erläuterungen dienen. Abbildung 11 zeigt ein kleines Modell mit fünf Modelleingängen. Wird bei einem Strukturtest beispielsweise eine Bedingungsüberdeckung angestrebt, so gilt es unter anderem, an den Eingängen des im Modell enthaltenen AND-Blocks ein *wahr* und *falsch* zu erreichen. Ein *wahr* bedeutet in diesem Zusammenhang, dass jedes in den Block eingehende Signal mindestens einmal einen Wert enthält, der ungleich 0 ist. Ein solches CG ist in der Abbildung dargestellt: $s_4 \neq 0$. Um nun festzustellen, welche Modelleingänge für das Erreichen dieses CG relevant oder irrelevant sind, verfolgt SDA die Einwirkungen auf die an dem CG beteiligten Signale bis hin zu den Modelleingängen zurück. Das an dem Beispiel-CG beteiligte Signal s_4 entsteht in seiner Wertebelegung durch die Anwendung des NOT-Blocks für Signal s_3. Dies bedeutet, s_4 hängt von s_3 ab – wie in dem Abhängigkeitsgraphen im unteren Teil der Abbildung dargestellt. Signal s_3 wiederum hängt von den Signalen s_2 und s_1 ab. Signal s_2 entstammt einem Constant-Block, daher endet



(a) Beispielmodell mit ausgewähltem Überdeckungsziel



(b) Ausschnitt aus dem Signalabhängigkeitsgraphen für das Modell in (a)

Abbildung 11: Einführendes Beispiel zur Modelleingang-Abhängigkeitsermittlung

die Zurückverfolgung an dieser Stelle. Signal s_1 hingegen stammt aus einem Modelleingang. Dieser stellt somit den einzigen für das betrachtete CG relevanten Modelleingang dar. Dieses Beispiel demonstriert die Grundzüge der SDA-Technik. Während der Zusammenhang zwischen CG und Modelleingängen in einem solch kleinen Modell wohlgemerkt offensichtlich ist, ist dieser in größeren Modellen im Allgemeinen weder derart einfach ablesbar, noch derart trivial ableitbar.

SIGNALABHÄNGIGKEITSGRAPH

Wie im Beispiel angedeutet, sammelt SDA die zwischen Modellsignalen existierenden Abhängigkeiten in einem Abhängigkeitsgraphen. Dies ist ein gerichteter Graph, in dem die Knoten Signale, beziehungsweise Signal-Vektorelemente im Falle vektorisierter Signale, darstellen. Eine Kante bezeichnet die Abhängigkeit eines Signals von einem Anderen. Die Abhängigkeitsbeziehung trifft dabei keine Aussage darüber, wie sich die Beschaffenheit eines Signals auf die Beschaffenheit eines anderen Signals auswirkt, sondern gibt lediglich an, dass die Wertebelegung des einen Signals Einfluss auf die Wertebelegung des Anderen nimmt. Horwitz et al. unterscheiden bei einer hierzu verwandten Analyse für Programmcode zwischen Datenabhängigkeit und Kontrollabhängigkeit [67]. Bezugnehmend auf diese Unterscheidung ist festzustellen, dass es sich bei dem Großteil der Signalabhängigkeiten innerhalb eines SL/TL-Modell für gewöhnlich um Datenabhängigkeiten handelt. In wenigen Ausnahmen gilt es allerdings auch Kontrollabhängigkeiten zu berücksichtigen, beispielsweise im Falle bedingt ausgeführter Subsysteme. Diese spiegeln sich in dem Abhängigkeitsgraph, den SDA aufbaut, jedoch letzten Endes auch als Datenabhängigkeiten wieder. Es wird nämlich nicht erfasst, unter welcher Bedingung beispielsweise ein solches Subsystem ausgeführt wird, sondern lediglich die Tatsache, dass das Kontrollsignal des bedingt ausgeführten Subsystems an der Wertebelegung eines Signals innerhalb dieses Subsystems beteiligt ist.

Ein Signal kann im Allgemeinen nicht nur von einem, sondern von mehreren anderen Signalen abhängig sein. Dies berücksichtigend beschreibt die folgende Abbildung eine Abhängigkeitsbeziehung zwischen einem Signal (mit Vektorindex-Angabe) und einer Menge von Signalen (ebenfalls jeweils mit Vektorindex-Angabe):

$$\xrightarrow{\text{dep}} : S^{\mathbb{N}^+} \times \mathcal{P}(S^{\mathbb{N}^+}) \tag{47}$$

Im Folgenden wird diese Abbildung in Infixnotation verwendet. Hängt ein Signal s_1 beispielsweise von den Signalen s_2 und s_3 ab, so wird dies durch $s_1 \xrightarrow{\text{dep}} \{s_2, s_3\}$ ausgedrückt. Vektoren-Indizes werden auch hier weggelassen, sofern Signale eindimensional sind.

Grundsätzlich gilt, dass die aus einem Block ausgehenden Signale von den in den Block eingehenden Signalen abhängen – und die Analyse der Abhängigkeiten somit basierend auf der Modellsyntax erfolgt. Genau genommen sind die Signalabhängigkeiten jedoch Block-spezifisch. So kann es zum einen sein, dass bei einem Block mit mehreren Ausgangssignalen ein Ausgangssignal nur von einem Teil der Eingangssignale abhängt. Zum anderen verarbeiten Blöcke gegebenenfalls Vektorsignale. Hierbei hängt ein Vektorelement des Ausgangssignals eines Blocks häufig nur von bestimmten Vektorelementen der Blockeingangssignale ab. Anders ausgedrückt lässt sich auch sagen: Rein Syntax-basiert ermittelte Abhängigkeiten können durch Berücksichtigung der Blockfunktionalitäten zum Teil wieder aus dem Abhängigkeitsgraph entfernt werden. Der so entstehende Abhängigkeitsgraph ist somit präziser als ein rein Syntax-basierter Abhängigkeitsgraph. Diese Sichtweise dient allerdings lediglich der Verdeutlichung, warum die Signalabhängigkeiten als Block-spezifisch aufgefasst werden. SDA sammelt keine Abhängigkeitsbeziehungen, die anschließend wieder gestrichen werden – sondern, soweit möglich, nur tatsächlich existierende Abhängigkeiten zwischen Signalen.

Für eine Reihe repräsentativer Blocktypen führt Tabelle 7 eine Definition der Abhängigkeitsbeziehung $\xrightarrow{\text{dep}}$ auf. Diese wird im Folgenden erläutert. Für die sonstigen SL/TL-Blocktypen gestaltet sich die Definition ähnlich. Daher wird auf eine vollständige Auflistung verzichtet.

Logical Operator. Für einen Logical-Operator-Block ist hierbei zu unterscheiden, ob dieser mehrere Eingänge oder nur einen Eingang hat – beziehungsweise ob er den NOT-Operator oder einen der anderen logischen Operatoren verwendet. Dies ist deshalb notwendig, da der Block seine Funktionalität dementsprechend entweder je Vektorelement der/des Eingänge/Eingangs oder für alle Vektorelemente seines einen Eingangs anwendet. In ersterem Fall hängt jedes Vektorelement des aus dem Block ausgehenden Signals jeweils von den Vektorelementen der Eingangssignale mit identischem Index ab. In zweiterem Fall hängt das Ausgangssignal, welches eindimensional ist, von allen Vektorelementen des einzigen Eingangssignals ab.

Inport. Ein Inport-Block befindet sich entweder in einem Subsystem oder auf der höchsten Hierarchie-Ebene – und stellt in diesem Fall einen Modelleingang dar. Ist er in einem Subsystem, so hängt jedes ν-te Vektorelement seines Ausgangssignals von dem ν-ten Vektorelement des Signals ab, welches in den entsprechenden Inport des Subsystems eingeht. Handelt es sich um einen Modelleingang, so besteht keine weitere Abhängigkeit des Blockausgangssignals (oder dessen Vektorelemente) von anderen Signalen.

Subsystem. Die Ausgangssignale eines Subsystem-Blocks hängen jeweils von dem Eingangssignal des entsprechenden Outport-Blocks, der sich

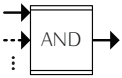
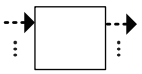
Blocktyp	*Signal-Abhängigkeit*
 Logical Operator	**Fall 1:** $n > 1_{\vee} \bullet = \neg$ $\forall \nu \in \mathbb{N}_{[1,\max d_{in}(b)]}$: $(sig_{out}(b,1))^{\nu} \xrightarrow{\textbf{dep}} \bigcup_{p \in IP_b} \{(sig(p))^{vc(\nu,p)}\}$ **Fall 2:** $n = 1_{\wedge} \bullet \neq \neg$ $(sig_{out}(b,1))^{1} \xrightarrow{\textbf{dep}} \bigcup_{\nu=1}^{d(b,1)} \{(sig_{in}(b,1))^{\nu}\}$
 Inport	**Fall 1:** $\exists sb \in B: b \in BC_{sb}$ $\forall \nu \in \mathbb{N}_{[1,d(sb,n)]}$: $(sig_{out}(b,1))^{\nu} \xrightarrow{\textbf{dep}} \{(sig_{in}(sb,n))^{\nu}\}$ **Fall 2:** $\forall sb \in B: b \notin BC_{sb}$ $(sig_{out}(b,1))^{1} \xrightarrow{\textbf{dep}} \emptyset$
 Subsystem	$\forall pos \in \mathbb{N}_{[1,\lvert OP_b \rvert]}: \forall \nu \in \mathbb{N}_{[1,d(op(b,pos))]}:$ $(sig_{out}(b,pos))^{\nu} \xrightarrow{\textbf{dep}} \{(sig_{in}(ob,1))^{\nu}\}$ **mit** $ob \in BC_b \wedge bt_{ob} = \text{Outport} \wedge (\text{„}m\text{“}, pos) \in PV_{ob}$
 Unbekannt	$\forall pos \in \mathbb{N}_{[1,\lvert OP_b \rvert]}: \forall \nu \in \mathbb{N}_{[1,d(op(b,pos))]}:$ $(sig_{out}(b,pos))^{\nu} \xrightarrow{\textbf{dep}} \bigcup_{p \in IP_b} \bigcup_{i=1}^{d(p)} \{(sig(p))^{i}\}$

Tabelle 7: Abhängigkeit der Ausgangssignale eines SL/TL-Blocks von dessen Eingangssignalen (Auszug)

im Subsystem befindet, ab. Handelt es sich um vektorisierte Signale, so bezieht sich die Abhängigkeit jeweils auf die Vektorelemente der Signale mit identischem Index.

Unbekannter Block. Auch die SDA-Technik soll, gemäß der in Abschnitt 2.5 gemachten Vorgabe, Modelle mit teils unbekannten oder nicht-berücksichtigten Blocktypen unterstützen können. Aus diesem Grund enthält SDA eine Definition der Abhängigkeitsbeziehung für unbekannte Blöcke. Diese Definition sagt aus, dass jedes Vektorelement eines Ausgangssignals eines unbekannten Blocks von allen Vektorelementen aller Eingangssignale des Blocks abhängt. Diese Annahme stellt, entsprechende Kenntnis über die Funktionalität des Blocks vorausgesetzt, zweifelsfrei eine Über-Approximation der tatsächlichen Signalabhängigkeiten dar. Dies kann dazu führen, dass ein Modelleingang letzten Endes

als relevant für ein CG erachtet wird, obwohl er eigentlich irrelevant ist. Dieser Umstand ist allerdings hinnehmbar, da eine Testdatensuche ohne Verwendung der SDA-Technik ohnehin jeden Modelleingang als relevant betrachten würde. Viel wichtiger ist: Eine solche Über-Approximation führt unter keinen Umständen dazu, dass relevante Modelleingange als irrelevant erachtet werden.

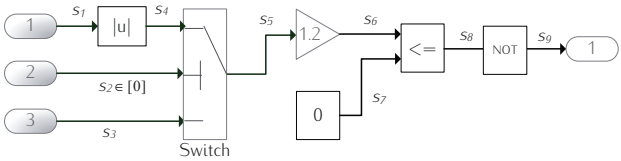
In Ergänzung zu den Definitionen der Abhängigkeitsbeziehungen in der Tabelle ist zu berücksichtigen, dass je Block gegebenenfalls weitere Signalabhängigkeiten hinzukommen. Befindet sich ein Block, wie zuvor bereits angedeutet, in einem bedingt ausgeführten Subsystem, beziehungsweise ist er in der Modellhierarchie unterhalb eines bedingt ausgeführten Subsystems angesiedelt, so hängen seine Ausgangssignale zusätzlich von dem Kontrollsignal des in der Hierarchie nach oben hin nächsten bedingt ausgeführten Subsystems ab. Wird dieses Subsystem nämlich zu einem Zeitschritt einer beliebigen Modellausführung nicht aktiviert, so sind dessen enthaltene Blöcke nicht aktiv und die Ausgangssignale dieser Blöcke enthalten keinen Wert. Daher hängt ein jedes solches Signal zusätzlich von etwaigen Kontrollsignalen ab.

Wird dies für jeden Modellblock ergänzend berücksichtigt, so gilt: Durch Anwendung der Abbildung $\xrightarrow{\text{dep}}$ für die Ausgangssignale eines jeden Blocks eines SL/TL-Modells entsteht der gewünschte Abhängigkeitsgraph. Die Reihenfolge, in der die Blöcke betrachtet werden, ist hierbei nicht von Bedeutung. Da jeder Block auch nur einmal betrachtet werden muss, steigt und fällt der Aufwand des Aufbaus eines Signalabhängigkeitsgraphen mit der Anzahl der Blöcke eines Modells.

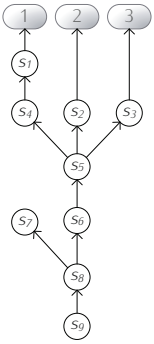
NUTZEN DER SIA-MODELLTRANSFORMATIONEN FÜR SDA

In Abschnitt 3.1.3 wurde im Zuge der Vorstellung der statischen Voranalyse SIA erklärt, dass innerhalb dieser Analyse gegebenenfalls Transformationen der betrachteten Modellrepräsentation durchgeführt werden. Die SDA-Technik verwendet für den Aufbau eines Signalabhängigkeitsgraphen die gleiche Modellrepräsentation wie SIA. Anhand eines Beispiels soll an dieser Stelle aufgezeigt werden, dass die Transformationen von SIA für die SDA-Technik von Vorteil sein können.

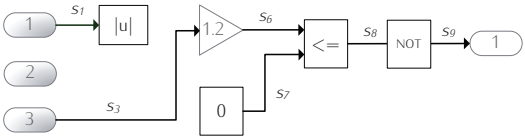
Ausgangspunkt des Beispiels ist das in Abbildung 12a dargestellte SL/TL-Modell. Der dazugehörige Signalabhängigkeitsgraph, wie ihn SDA ableiten würde, ist in Abbildung 12b zu sehen. Für das Beispiel wird nun folgende Annahme getroffen: SIA stellt fest, dass der Wertebereich des zweiten Modelleingangs für alle betrachteten Zeitschritte ausschließlich den Wert 0 enthält. Da dies im betrachteten Beispiel dazu führen würde, dass der Switch-Block stets den an seinem dritten Eingang anliegenden Wert ausgibt, führt SIA eine Transformation durch. Der Switch-Block wird entfernt und das ursprünglich dritte Eingangssignal des Switch-Blocks mit



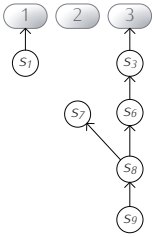
(a) Modellrepräsentation ohne Durchführung von Transformationen bei SIA



(b) Signalabhängigkeitsgraph für die Modellrepräsentation in (a)



(c) Modellrepräsentation mit Durchführung von Transformationen bei SIA



(d) Signalabhängigkeitsgraph für die Modellrepräsentation in (b)

Abbildung 12: Ausnutzung der Modelltransformationen von SIA bei SDA anhand eines Beispiels

dessen ursprünglichem Ausgangssignal vereinigt. Nicht mehr benötigte Modellbestandteile werden gemäß der Erläuterungen in Abschnitt 3.1.3 entfernt. Das transformierte Beispiel-Modell ist in Abbildung 12c zu sehen. Interessant wird es nun bei einem Blick auf den Signalabhängigkeitsgraphen, den SDA für das transformierte Modell ableitet. Dieser enthält, wie in Abbildung 12d dargestellt, weniger Abhängigkeitsbeziehungen als sein Gegenstück in Abbildung 12b. Die Signalabhängigkeiten konnten durch die Transformationen von SIA präzisiert werden. Für ein CG, welches sich von dem NOT-Block des Beispiel-Modells ableitet, wird zum Beispiel nun erkannt, dass Modelleingang eins und zwei für die Erfüllung des CG irrelevant sind. Ohne Durchführung dieser Transformation bei SIA hätte sich eine Testdatensuche auf alle drei Modelleingänge konzentriert.

MODELLEINGANGSABHÄNGIGKEIT

Den für ein SL/TL-Modell erstellten Signalabhängigkeitsgraphen verwendet SDA, um für jedes der betrachteten CGs zu prüfen, von welchen Modelleingängen dieses abhängt. Die Abhängigkeit eines CG von einem Modelleingang hat zur Folge, dass dieser Eingang gezielt mit passenden Testdaten stimuliert werden muss, um das Erreichen des CG zu gewährleisten.

Zur Ermittlung der Menge relevanter Modelleingänge sei zuerst die Abbildung $\text{esig} : E \to \mathcal{P}(S^{\mathbb{N}^+})$ definiert. Diese gibt für einen CG-Ausdruck oder -Teilausdruck an, welche Signale (jeweils inklusive Vektorindex-Angabe) in ihm vorkommen.

$$\begin{aligned}
\text{esig}(\xi_1 \bullet \xi_2) &= \begin{cases} \{\xi_1, \xi_2\} & , \xi_1 \in S^{\mathbb{N}^+} \wedge \xi_2 \in S^{\mathbb{N}^+} \\ \{\xi_1\} & , \xi_1 \in S^{\mathbb{N}^+} \wedge \xi_2 \notin S^{\mathbb{N}^+} \\ \{\xi_2\} & , \xi_1 \notin S^{\mathbb{N}^+} \wedge \xi_2 \in S^{\mathbb{N}^+} \\ \emptyset & , \text{sonst} \end{cases} \\
\text{esig}(\circ \varphi_i) &= \bigcup_{i=1}^{n} \text{esig}(\varphi_i) \qquad (48) \\
\text{esig}(\neg \varphi) &= \text{esig}(\varphi)
\end{aligned}$$

Hierbei gilt: $\bullet \in \{>, \geqslant, <, \leqslant, =, \neq\}$ und $\circ \in \{ \bigwedge_{i=1}^{n}, \bigvee_{i=1}^{n} \}$.

Die Abhängigkeit eines CG φ von einer Menge von Modelleingängen sei dann durch die Abbildung $\xrightarrow{\text{in}} : E \times \mathcal{P}(\text{MB})$ gegeben. Wie in der folgenden Definition ersichtlich, wird die Abbildung ebenfalls in Infixnotation verwendet.

$$\varphi \xrightarrow{\text{in}} \bigcup_{s^\nu \in \text{esig}(\varphi)} \text{vis}(s^\nu, \emptyset) \tag{49}$$

Die Abbildung $\text{vis} : S^{\mathbb{N}^+} \times \mathcal{P}(S^{\mathbb{N}^+}) \to \mathcal{P}(\text{MB})$ wird hierbei für jedes in dem CG-Ausdruck vorkommende Signal angewandt. Ausgehend von dem betreffenden Signal traversiert sie den Abhängigkeitsgraphen durch Abgehen der Abhängigkeitsbeziehungen bis hin zu den auf diesem Wege erreichbaren Modelleingängen. Diese werden der gesuchten Menge hinzugefügt. Mithilfe des zweiten Parameters der Abbildung, eine Signalmenge, wird festgehalten, welche Signale bei der Traversierung bereits besucht wurden. Dies ist notwendig, damit etwaige Zyklen im Abhängigkeitsgraphen nur einmalig beschritten werden.

$$\text{vis}(s^\nu, H) = \begin{cases} \emptyset & , s^\nu \in H \\ \text{ib} & , s^\nu \xrightarrow{\text{dep}} \emptyset \\ & \quad \wedge\, \text{ib} = b(\text{src}(s)) \in \text{MB} \\ \bigcup\limits_{\substack{x^w \in X \\ (s^\nu \xrightarrow{\text{dep}} X)}} \text{vis}(x^w, H \cup \{s^\nu\}) & , \text{sonst} \end{cases} \tag{50}$$

Die Menge der für ein CG φ irrelevanten Modelleingänge ist demnach $\text{MB} \setminus X$, wobei $\varphi \xrightarrow{\text{in}} X$ gilt. Diese irrelevanten Modelleingänge werden bei einer Modellausführung im Rahmen einer Testdatensuche für φ mit zufällig generierten Signalen belegt. Für jedes Testdatum wird hierzu pro irrelevantem Modelleingang eine Optimierungssequenz erzeugt, deren Parameter unter Berücksichtigung der Modelleingangsspezifikation zufällig gewählt sind. Diese Optimierungssequenzen werden von den Operatoren des verwendeten Suchverfahrens nicht angefasst. Dies bedeutet, dass bei der Erzeugung neuer oder veränderter Testdaten keine gezielte Modifikation dieser Optimierungssequenzen erfolgt.

Dennoch bleiben diese für das betrachtete CG irrelevanten Optimierungssequenzen von Testdatum zu Testdatum nicht unverändert. Für jedes Testdatum erfolgt eine erneute zufällige Erzeugung. Hierdurch wird bei der Suche die Wahrscheinlichkeit erhöht, dass andere, bislang unerreichte CGs, also Andere als das anvisierte CG φ, zufälligerweise auch erreicht werden (Kollateralüberdeckung). Ohne diesen Hintergedanken könnte man irrelevante Eingänge auch schlichtweg mit Signalen belegen, welche durchgängig den gleichen Wert haben. Versuche haben allerdings gezeigt, dass dies das Ausmaß der Kollateralüberdeckung, wie sie eine Suche normalerweise hat, verringern würde. Eine zufällige Generierung der Signale für irrelevante Modelleingänge individuell für jedes erzeugte Testdatum beseitigt diesen Nachteil, den der Einsatz der SDA-Technik ansonsten mit sich bringen würde.

3.2.3 VERWANDTE ARBEITEN

Bei der Anwendung des suchbasierten Ansatzes zur Testdatengenerierung für Programmcode beschäftigte sich Korel bereits zu Beginn der 1990er Jahre mit dem Einfluss der Eingangsvariablen eines Programms auf das Erreichen eines Strukturelements im Programmcode [76]. In diesem Zusammenhang schlägt er eine Datenflussabhängigkeitsanalyse vor. Diese Analyse ist jedoch dynamisch gestaltet, das heißt Abhängigkeiten werden während der Programmausführungen im Rahmen einer Testdatensuche gesammelt. Korel regt in seiner Arbeit an, dass das durch ihn betrachtete Vorhaben durch statische Analysen möglicherweise effizienter zu realisieren wäre. Harman et al. [61] und McMinn et al. [87] griffen diesen Vorschlag auf und ergänzten den suchbasierten Strukturtest für Code um eine entsprechende statische Analyse. Um irrelevante Eingangsvariablen von einer der Suche dienenden Optimierung auszuschließen, verwenden die Autoren beider Arbeiten eine vor der Suche stattfindende Variablen-Abhängigkeitsanalyse. Die Arbeit von McMinn et al. ist, bezüglich der Zielsetzung und des grundlegenden Vorgehens, die am nächsten mit der SDA-Technik verwandte Arbeit. SDA überträgt die Konzepte von Harman, McMinn et al. von der Code- auf die Modell-Welt.

Harman et al. [61] untersuchten des Weiteren in einer Studie die Auswirkungen des Einsatzes eines solchen Eingangsvariablen-Ausschlusses auf eine Testdatengenerierung. Als Testdatengenerierungsverfahren verwenden sie sowohl einen Zufallstest als auch zwei suchbasierte Ansätze. Ihre Experimente mit Open-Source-Code und Code aus der Serienentwicklung von Daimler bestätigen die Vermutung, dass eine ergänzende Durchführung einer Variablen-Abhängigkeitsanalyse für die rein zufällige Testdatengenerierung keinerlei Vorteile mit sich bringt. Dies ist wenig überraschend, da in diesem Fall relevante und irrelevante Eingangsvariablen bei Programmausführungen ohnehin gleichermaßen mit Zufallswerten belegt werden. Für „intelligentere“ Testdatengenerierungsverfahren, wie lokale und globale Suchverfahren, stellen Harman et al. allerdings fest, dass der Ausschluss von irrelevanten Eingangsvariablen dazu im Stande ist, die Effizienz einer Suche zu steigern – in den betrachteten Fällen allerdings ohne dabei einen höheren Überdeckungsgrad zu erreichen. Als Suchverfahren kamen in dieser Studie das Hill Climbing und ein Genetischer Algorithmus zum Einsatz.

Abgesehen von dem betrachteten Kontext der Testdatengenerierung ist festzustellen, dass der SDA-Ansatz ein Problem behandelt, welches in seinem Kern der Variablenabhängigkeit in Programmcode [60] gleichkommt. So gesehen ist SDA auch eng verwandt mit *Slicing*-Techniken [117, 129]. Slicing (zu Deutsch: schneiden oder abschneiden) bezeichnet die Vereinfachung von Programmen durch eine Fokussierung auf ausgewählte Aspekte der Programmsemantik, wie Harman und Hierons

[58] definieren. Slicing entfernt Programmbestandteile, für welche ermittelt werden kann, dass sie keine Auswirkung auf den betrachteten Aspekt haben. Slicing-Techniken finden in den Bereichen Testen, Debugging, Re-Engineering und Programmanalyse sowie bei der Ableitung von Software-Metriken Anwendung [58].

Bei SDA handelt es sich zwar nicht um eine Slicing-Technik, allerdings hat SDA die Ermittlung von Abhängigkeiten innerhalb eines Programmablaufs mit Slicing-Techniken gemein. Der Unterschied besteht grundsätzlich in der Verwendung der ermittelten Abhängigkeiten. So dienen diese bei SDA der Erkennung der Relevanz von Programmeingaben (Modelleingänge) hinsichtlich ihres Einflusses auf Programmvariablen (Signale). Slicing-Techniken hingegen nutzen die ermittelten Abhängigkeiten, um einen Ausschnitt des betrachteten Programms zu extrahieren oder Programmteile zu entfernen.

Aus technischer Sicht kann die SDA-Technik daher als verwandt mit einer Arbeit von Reicherdt und Glesner [98], die zeitlich parallel zu der vorliegenden Arbeit entstanden ist, angesehen werden. Reicherdt und Glesner schlagen einen Ansatz zum Slicing von SL-Modellen vor. Bei den Abhängigkeiten, welche dabei gesammelt werden, handelt es sich ebenfalls um sowohl Daten- als auch Kontrollabhängigkeiten. Im Unterschied zur SDA-Technik betrachtet der Slicing-Ansatz für SL-Modelle allerdings Abhängigkeiten zwischen Modellblöcken. SDA hingegen analysiert die Abhängigkeiten zwischen den Signalen eines Modells. Aus rein syntaktischer Sicht auf ein Modell ist dieser Unterschied unerheblich. Für das durch die SDA-Technik verfolgte Ziel ist die Ermittlung der Signalabhängigkeit dennoch das präzisere Mittel. Enthält ein Modell nämlich beispielsweise vektorisierte Signale, so stellt die Erfassung von Blockabhängigkeiten statt Signalabhängigkeiten eine Abstraktion dar, welche zu einem Verlust an Detailinformationen führt.

Die Beiträge der SDA-Technik im wissenschaftlichen Diskurs lassen sich folgendermaßen zusammenfassen:

1. Transfer der Idee des Eingangsvariablen-Ausschlusses bei einer Testdatengenerierung für Programmcode auf den Kontext der Testdatengenerierung für SL/TL-Modelle
2. Analyse der Signalabhängigkeiten eines SL/TL-Modells unter Berücksichtigung von Abhängigkeiten, die sich nicht direkt aus der Modellsyntax ergeben

3.3 SUCHZIELPRIORISIERUNG

Analog zu den zuvor vorgestellten statischen Voranalysen wird in diesem Abschnitt eine dritte statische Voranalyse in ihrer Zielsetzung, Funktionsweise und ihrem Verhältnis zu verwandten Arbeiten behandelt. Bei der dritten statischen Voranalyse handelt es sich um die mit dem Kürzel CGSeq (in Englisch: *Coverage Goal Sequencing*) bezeichnete Suchzielpriorisierung.

3.3.1 ZIEL

Die CGSeq-Technik adressiert drei der in Abschnitt 2.5 identifizierten Probleme einer suchbasierten Testdatengenerierung für SL/TL-Modelle. Zum einen berücksichtigt die Technik Abhängigkeiten zwischen CGs, mit dem Ziel, die bei einer Testdatensuche erzielte Kollateralüberdeckung zu erhöhen. Hierdurch kann sich die Anzahl der insgesamt notwendigen Suchvorgänge verringern. Entsprechend steigt die Effizienz der gesamten Automatisierung. Zum anderen kann CGSeq in vielen Fällen, wo die Problematik einer „blinden" oder einer Booleschen Fitnessfunktion auftritt, helfen. Die zwischen CGs ermittelten Abhängigkeiten werden nämlich auch genutzt, um anstatt eines problematischen CG gegebenenfalls ein anderes CG, dessen Erfüllung die Erfüllung des problematischen CG zur Folge hat, mit einer Testdatensuche anzuvisieren.

Ein einführendes Beispiel soll diese Überlegungen verdeutlichen. Abbildung 13 zeigt ein kleines Modell und zusätzlich eine Reihe hieraus abgeleiteter CGs. So existieren beispielsweise die CG-Ausdrücke $s_1 \geqslant 90$, $s_1 \geqslant 70$ und $s_1 \geqslant 50$. Ohne gesonderte Berücksichtigung der Zusammenhänge zwischen diesen CGs könnte es dazu kommen, dass das CG mit dem Ausdruck $s_1 \geqslant 50$ als erstes durch einen Suchvorgang anvisiert wird. Angenommen, die Suche findet Testdaten, die zu einem Wert von 55 für das Signal s_1 führen, so gilt das CG als erfüllt. Die CG-Ausdrücke $s_1 \geqslant 90$ und $s_1 \geqslant 70$ werden hierdurch allerdings nicht erfüllt. Würde man hingegen zuerst das CG mit dem Ausdruck $s_1 \geqslant 90$ betrachten, so sind bei Erfolg der Testdatensuche all diese drei CGs auf einmal erfüllt.

Neben dem Aspekt der Kollateralüberdeckung sei anhand des Beispiels auch der Nutzen von CGSeq im Zusammenhang mit der Problematik Boolescher Fitnessfunktionen aufgezeigt. Die Fitnessfunktionen der CGs, die sich in dem Beispiel-Modell von dem NOT-Block sowie dem Switch-Block ableiten, unterliegen beispielsweise dieser Problematik. Die Signale, auf welche sich die CGs jeweils beziehen, haben einen Booleschen Wertebereich. Dies lässt sich durch die SIA-Technik feststellen. Die Fitnessfunktion eines CGs besitzt in diesen Fällen ebenfalls einen nur zweielementigen Wertebereich. Eine derartige Fitnessfunktion ist

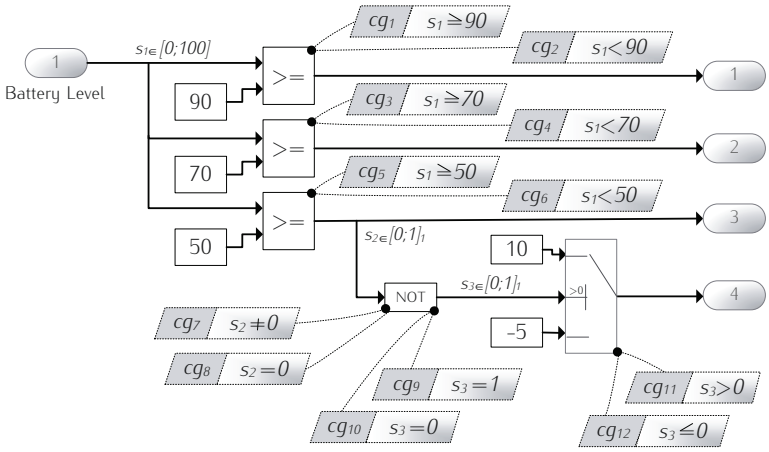


Abbildung 13: Motivierendes Beispiel zur Suchzielpriorisierung

für das verwendete Suchverfahren unvorteilhaft, da sie keine qualitative Unterscheidung von Testdaten liefert, die das betrachtete CG nicht erfüllen. CGSeq hilft in diesem Fall, da die Technik erkennen würde, dass beispielsweise aus der Erfüllung des CG-Ausdrucks $s_1<50$ auch die Erfüllung der Ausdrücke $s_2=0$, $s_3=1$ und $s_3>0$ folgt. Das CG mit dem Ausdruck $s_1<50$ wird in diesem Fall priorisiert durch eine Testdatensuche anvisiert und das Problem Boolescher Fitnessfunktionen somit umgangen.

Der Einsatz der CGSeq-Technik eignet sich insbesondere bei Testdatengenerierungen im Rahmen eines Strukturtests, darüber hinaus aber auch in den anderen möglichen Anwendungsszenarien (siehe Abschnitt 2.2). Der Ansatz geht allerdings in jedem Fall davon aus, dass initial alle CGs aller strukturorientierten Überdeckungskriterien (siehe Abschnitt 2.2.1) aus dem betrachteten SL/TL-Modell abgeleitet werden. Eine Nutzerwahl des gewünschten Überdeckungskriteriums, einzelner CGs oder anderweitiger CG-ähnlicher Zieldefinitionen, wie bei einem suchbasierten Funktionstest (siehe Abschnitt 2.2.2), wird bei Verwendung der CGSeq-Technik erst innerhalb derselben berücksichtigt.

CGSeq versucht also, aus den Zusammenhängen zwischen möglichst vielen markanten Zustandszielen, welche durch CGs gegeben sind, einen Nutzen für die Effizienz der Testdatengenerierung zu ziehen. Hierzu werden, unter Berücksichtigung der zuvor angesprochenen Nutzerwahl, aus den CG-Zusammenhängen sogenannte *Suchziele* (SGs, Singular: SG) abgeleitet. Die Abkürzung leitet sich vom englischen Begriff *Search Goal* ab. Ein SG kann ein CG oder mehrere CGs enthalten. Die abgeleiteten SGs werden anschließend für die nachfolgende Bearbeitung durch Suchvor-

gänge sortiert. Die Sortierung erfolgt durch die Verwendung von eigens hierfür definierten Priorisierungsmetriken. Diese berücksichtigen unter anderem, wie hoch die Kollateralüberdeckung eines CG ist und, sofern die SIA-Technik zuvor eingesetzt wurde, ob ein CG ein Boolesches Signal adressiert. Die Berechnungsgrundlage der Metriken sind hauptsächlich die ermittelten CG-Zusammenhänge.

Zwei Teilvorgänge bilden zusammen die CGSeq-Technik: die *Überdeckungszielanalyse* und die *Abarbeitungsreihenfolgenbildung*. Die folgenden beiden Abschnitte stellen beide Vorgänge ausführlich vor. Hierzu begleitend fasst Abbildung 14 die Funktionsweise der beiden Vorgänge und von CGSeq insgesamt zusammen.

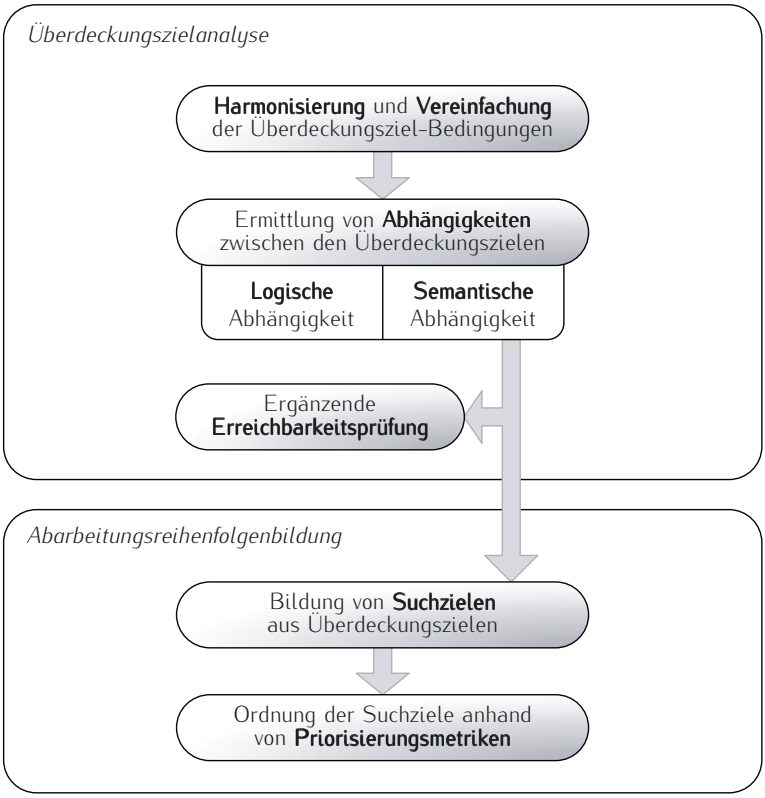


Abbildung 14: Funktionsweise der Suchzielpriorisierung

3.3.2 ÜBERDECKUNGSZIELANALYSE

Die Überdeckungszielanalyse ermittelt für die Gesamtmenge der CGs, ob zwischen den enthaltenen CGs prädikatenlogische Zusammenhänge existieren. In einem vorbereitenden Schritt werden die Ausdrücke der CGs in ein einheitliches Muster gebracht, um die Fallunterscheidungen der darauffolgenden Schritte schlanker gestalten zu können (Abschnitt 3.3.2.1). Die erfassten CG-Zusammenhänge formen einen speziellen Abhängigkeitsgraphen. Die Syntax eines solchen Graphen wird in Abschnitt 3.3.2.2 vorgestellt. Die Ermittlung der Abhängigkeiten zwischen CGs erfolgt in zwei Schritten. Zuerst werden *logische* Abhängigkeiten (Abschnitt 3.3.2.3) und anschließend *semantische* Abhängigkeiten (Abschnitt 3.3.2.4) erfasst. Erstere ergeben sich einzig und allein aus den CG-Ausdrücken, Letztere durch Berücksichtigung der Modellsemantik. Als Nebenprodukt der CGSeq-Technik können durch die ermittelten Abhängigkeiten unter Umständen weitere unerreichbare CGs identifiziert werden. Die hierzu existierende Erreichbarkeitsprüfung, welche die Erreichbarkeitsprüfung von SIA ergänzt, wird in Abschnitt 3.3.2.5 behandelt.

3.3.2.1 *Harmonisierung und Vereinfachung*

Die bei der CG-Abhängigkeitsanalyse durchgeführten Vergleiche einzelner CGs gestalten sich einfacher, wenn die CG-Ausdrücke zuvor in ein einheitliches Format gebracht werden. Die hierzu durchgeführte Harmonisierung und Vereinfachung erfolgt in drei Schritten:

1. Anpassung von CG-Ausdrücken über die für deren beteiligte Signale ermittelten Wertebereiche (falls SIA verwendet wird)
2. Sortierung der Parameter innerhalb von CG-Ausdrücken
3. Anwendung von Vereinfachungsregeln für CG-Ausdrücke

Für jeden dieser Schritte sei ein Beispiel betrachtet. Eine vollständige Auflistung aller Fälle oder eine Formalisierung der angewendeten Regeln und Umformungen würde den Rahmen an dieser Stelle sprengen – insbesondere, da eine Harmonisierung und Vereinfachung der CG-Ausdrücke für CGSeq nicht zwingend erforderlich ist, sondern für die folgenden Analysen lediglich eine einfachere Gestaltung zulassen.

Schritt 1. Angenommen, ein CG besitzt den Ausdruck $s>0$ und für das Signal s wurde durch SIA der Wertebereich $s\in[0;1]_1$ identifiziert. Dieser Wertebereich verfügt über eine Quantisierung, ist also diskretisiert. Er enthält ausschließlich die Werte 0 und 1. Da es nur einen Wert gibt, der größer als 0 ist, würde CGSeq den relationalen Operator des CG-Ausdrucks in diesem Fall austauschen: Aus $s>0$ wird $s=1$.

Schritt 2. Die relationalen Teilausdrücke innerhalb eines CG-Ausdrucks werden geordnet. Dies bedeutet, dass bei einem Vergleich eines Signals mit einer Konstanten das Signal zuerst aufgeführt wird. Der Ausdruck $0>s$ würde demnach in $s<0$ geändert werden. Für den relationalen Vergleich zweier Signale ist zu berücksichtigen: In der Umsetzung der Ansätze dieser Arbeit wird einem Signal eines SL/TL-Modells eine eindeutige Identifikationsnummer (ID) zugewiesen. Eine solche Signal-ID wird innerhalb dieses Dokuments, sofern eine Angabe notwendig ist, im Index einer Signalvariablen notiert (Beispiel: s_5 hat die Signal-ID 5). In Schritt 2 gilt hierauf basierend: Werden in einem relationalen CG-Teilausdruck zwei Signale miteinander verglichen, so wird das Signal mit der kleineren Signal-ID zuerst aufgeführt. Aus $s_3<s_1$ wird beispielsweise $s_1>s_3$.

Schritt 3. In diesem Schritt werden unter anderem Negationen in den CG-Ausdrücken eliminiert. Hierzu werden zuerst die De Morganschen Gesetze angewandt, so dass Negationen nur noch direkt vor relationalen Teilausdrücken vorkommen. Eine Negation kann dann entfernt werden, indem der relationale Operator einer Teilaussage umgekehrt wird. Aus dem Ausdruck $\neg(s=0)$ wird beispielsweise $s\neq0$. Neben der Beseitigung von Negationen ist es auch sinnvoll, Redundanzen, wie im Falle des Ausdrucks $s>4\wedge s>10$, aufzulösen.

3.3.2.2 *Überdeckungsziel-Abhängigkeitsgraph*

Bevor es um die Ermittlung von CG-Abhängigkeiten gehen soll, sei die Struktur des dabei entstehenden Abhängigkeitsgraphen vorgestellt. Bei einem CG-Abhängigkeitsgraphen handelt es sich um einen zum Teil gerichteten, zum Teil aber auch ungerichteten Graphen, in dem die Knoten Gruppen äquivalenter CGs und die gerichteten Kanten zwischen derartigen Knoten Implikationsbeziehungen repräsentieren. Ein Graph kann allerdings eine Reihe weiterer logischer Beziehungen enthalten. Diese sind in einem Graphen als ungerichtete Kanten dargestellt. Zudem gibt es zwei Arten von Junktoren als weitere Knotentypen, über welche sich zusammengesetzte Implikationsbeziehungen zwischen CGs, beziehungsweise CG-Gruppen, ausdrücken lassen.

Ein CG-Abhängigkeitsgraph sei gegeben als

$$\mathrm{CGDG} = (V_{\mathrm{CG}}, V_{\mathrm{K}}, V_{\mathrm{D}}, E_{\rightarrow}, E_{\uparrow}, E_{\oplus}, E_{\dashleftarrow}) \tag{51}$$

wobei $V_{\mathrm{CG}}\subseteq\mathcal{P}(E)$ die Menge derer Knoten ist, welche gruppierte, also äquivalente CGs beinhalten, und V_{K} die Knotenmenge für Konjunktoren sowie V_{D} die Knotenmenge für Disjunktoren sind. Die Menge $E_{\rightarrow} \subseteq (V_{\mathrm{CG}}\cup V_{\mathrm{K}}\cup V_{\mathrm{D}})\times(V_{\mathrm{CG}}\cup V_{\mathrm{K}}\cup V_{\mathrm{D}})$ enthält gerichtete Graphkanten zur Beschreibung der Implikationsbeziehung zwischen CG-Gruppen,

Disjunktoren und Konjunktoren. Die Mengen $E_{\uparrow}, E_{\oplus} \subseteq V_{CG} \times V_{CG}$ beinhalten ungerichtete Kanten, welche der Beschreibung von *NAND*- und *XOR*-Beziehungen zwischen CG-Gruppen dienen. Die Kantenmenge $E_{\leftarrow\!\!-} \subseteq V_{CG} \times V_{CG}$ enthält rückwärtsgerichtete Implikationsbeziehungen. Die verschiedenen Beziehungstypen sind in Tabelle 8 veranschaulicht und werden nachfolgend erläutert.

Die Erkennung äquivalenter CGs dient einer direkten Reduktion der Anzahl an CGs, die es durch separate Suchvorgänge zu bearbeiten gilt. Für eine Gruppe äquivalenter CGs gilt: Ist eines der CG einer Gruppe erfüllt, so sind alle anderen CGs der Gruppe ebenfalls erfüllt. Gleiches gilt für die Nicht-Erreichbarkeit. Ein Hinweis zur Schreibweise: Eine Gruppe äquivalenter CGs wird als Menge $CGN_{a,\ldots,n} \in V_{CG}$ geschrieben. Die Indizes $a, \ldots, n$ geben hierbei an, welche CGs in der Gruppe enthalten sind: $CGN_{a,\ldots,n} = \{cg_a, \ldots, cg_n\}$. In den folgenden Darstellungen werden die Mengenklammern einer CG-Gruppe in der Regel weggelassen, wenn diese nur ein CG enthält. Des Weiteren wird die Abbildung $cgn : E \rightarrow V_{CG}$ verwendet, um für ein CG den dazugehörigen Knoten im Abhängigkeitsgraphen zu erhalten:

$$cgn(cg_i) = CGN_{a,\ldots,n} \quad (i \in \{a, \ldots, n\}) \tag{52}$$

Die Implikationsbeziehung sagt aus, dass aus der Erfüllung eines CG die Erfüllung eines anderen CG folgt. Das Erfassen derartiger Beziehungen ist wesentliche Grundlage für CGSeq, um die Kollateralüberdeckung einer Testdatensuche zu erhöhen. Neben der normalen Implikationsbeziehung existiert der Beziehungstyp rückwärtsgerichteter Implikationen. In ihrer Bedeutung unterscheiden sich beide Beziehungstypen nicht. Rückwärtsgerichtete Implikationen bilden allerdings ausschließlich semantische Abhängigkeiten ab. Diese ergeben sich unter Berücksichtigung der Semantik eines Modellblocks gegebenenfalls für die CGs, die mit dessen Ein- und Ausgangssignalen in Verbindung stehen. Während Abhängigkeiten der CGs der Blockausgangssignale von CGs der Blockeingangssignale durch normale Implikationsbeziehungen abgebildet werden, werden rückwärtsgerichtete Implikationen ausschließlich für Abhängigkeiten der CGs der Blockeingangssignale von CGs der Blockausgangssignale verwendet. Rückwärtsgerichtete Implikationen werden einzig und allein bei der ergänzenden Erreichbarkeitsprüfung genutzt.

Die NAND- und XOR-Beziehung ermöglichen es, auszudrücken, dass zwei CGs (oder Gruppen von CGs) nicht gleichzeitig erfüllt sein können. Gleichzeitig bedeutet hierbei: zu dem gleichen Zeitschritt einer Modellausführung. Der Unterschied zwischen NAND- und XOR-Beziehung besteht darin, dass die NAND-Beziehung eine gleichzeitige Nicht-Erfüllung der beteiligten CGs erlaubt. Die XOR-Beziehung hingegen gibt an, dass zu jedem Zeitschritt einer Modellausführung stets eines der beiden in

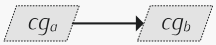
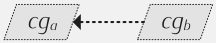
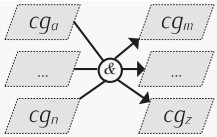
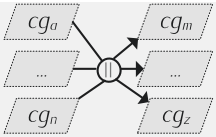
Grafische Darstellung	*Beschreibung*	*Formale Schreibweise*
	Knoten (äquivalenter Überdeckungsziele)	$CGN_{a,...,n} = \{cg_a, ..., cg_n\}$
	Implikation	$CGN_a \rightarrow CGN_b$
	Verneinte Konjunktion (NAND)	$CGN_a \uparrow CGN_b$
	Kontravalenz (XOR)	$CGN_a \oplus CGN_b$
	Rückwärtsgerichtete Implikation	$CGN_a \dashleftarrow CGN_b$
	Konjunktive Implikation	$(CGN_a \wedge ... \wedge CGN_n) \rightarrow (CGN_m \wedge ... \wedge CGN_z)$
	Disjunktive Implikation	$(CGN_a \vee ... \vee CGN_n) \rightarrow (CGN_m \wedge ... \wedge CGN_z)$
Es gilt: $cg_a, cg_b, cg_n, cg_m, cg_z \in E$		$(a,b,n,m,z \in \mathbb{N}^+)$

Tabelle 8: Syntax eines Überdeckungsziel-Abhängigkeitsgraphen

Beziehung gesetzten CGs erfüllt ist. Die XOR-Beziehung ist also eine striktere Variante der NAND-Beziehung, das heißt es gilt:

$$(n_1, n_2) \in E_{\oplus} \Rightarrow (n_1, n_2) \in E_{\uparrow} \tag{53}$$

Die Erfassung dieser beiden Beziehungstypen dient in erster Linie der Erkennung weiterer unerreichbarer sowie stets erreichter CGs – in Ergänzung zu denen, die bereits durch die SIA-Technik erkannt werden. Bei den nachfolgenden Betrachtungen der logischen und semantischen Abhängigkeiten ist die Kommutativität dieser beiden Ausschlussbeziehungen zu berücksichtigen:

$$(n_1, n_2) \in E_{\oplus} \Leftrightarrow (n_2, n_1) \in E_{\oplus} \text{ und } (n_1, n_2) \in E_{\uparrow} \Leftrightarrow (n_2, n_1) \in E_{\uparrow} \tag{54}$$

Mithilfe des Junktors *konjunktive Implikation* (Konjunktor) lässt sich beschreiben, dass aus der gleichzeitigen Erfüllung mehrerer CGs die Erfüllung eines oder mehrerer CGs folgt. Der Junktor *disjunktive Implikation* (Disjunktor) hingegen drückt aus, dass die Erfüllung mindestens eines mehrerer CGs die Erfüllung eines oder mehrerer CGs zur Folge hat. Junktoren beider Typen können über eine Implikationsbeziehung miteinander verbunden sein. Dieser Fall ist in der Tabelle nicht veranschaulicht.

3.3.2.3 *Logische Abhängigkeiten*

Die zwischen CGs möglichen Beziehungen lassen sich zum einen durch eine ausschließliche Betrachtung der CG-Ausdrücke erkennen. Anhand der in den Ausdrücken enthaltenen Operatoren, Signale und Konstanten lassen sich Zusammenhänge entsprechend der im vorigen Abschnitt beschriebenen Beziehungstypen erkennen. Die so gewonnenen Abhängigkeiten werden als logische Abhängigkeiten bezeichnet. Sofern vor der CGSeq-Technik die SIA-Technik verwendet wird, so werden bei Betrachtung der CG-Ausdrücke an dieser Stelle auch die durch SIA bestimmten Signalwertebereiche berücksichtigt. Die Beziehung $(s_1{>}0) \rightarrow (s_1{\geqslant}0)$ lässt sich zum Beispiel durch eine reine Betrachtung der in den Ausdrücken involvierten Operatoren erkennen. Die Beziehung $(s_1{==}5) \rightarrow (s_1{\geqslant}0)$ hingegen bedarf zu ihrer Erkennung eine zusätzliche Berücksichtigung der beteiligten Konstanten. Die Signalwertebereiche sind beispielsweise für die Erkennung der Beziehung $(s_1{==}3) \rightarrow (s_1{>}s_2)$ zu beachten. Hierfür müsste nämlich $\max(s_2){<}3$ gelten, das heißt der größtmögliche Wert, den das Signal s_2 annehmen kann, ist kleiner als die Konstante 3.

Bevor das Vorgehen zur Erkennung logischer CG-Abhängigkeiten im Detail vorgestellt wird, sei dieser Schritt für das Beispiel-Modell aus Abbildung 13 veranschaulicht. Abbildung 15 gibt hierzu den CG-Abhängigkeitsgraphen an. Dieser enthält ausschließlich logische Abhängigkeiten. Wie zu sehen ist, schließen sich die CG-Paare cg_1 ($s_1{\geqslant}90$) und cg_2 ($s_1{<}90$), cg_3 ($s_1{\geqslant}70$) und cg_4 ($s_1{<}70$) sowie cg_5 ($s_1{\geqslant}50$) und cg_6 ($s_1{<}50$) jeweils bezüglich ihrer Erfüllung gegenseitig aus. Hierbei handelt es sich um XOR-Beziehungen. Die CG-Paare cg_1 und cg_4, cg_1 und cg_6 sowie cg_3 und cg_6 können ebenfalls nicht gleichzeitig erfüllt sein, hierbei handelt es sich allerdings lediglich um NAND-Beziehungen. Aus der Erfüllung von cg_1 folgt logischerweise auch die Erfüllung von cg_3 und cg_5. Eine Implikationsbeziehung existiert ebenso zwischen cg_3 und cg_5. Ähnliches gilt für cg_6, cg_4 und cg_2. Für die CGs cg_9 ($s_3{=}1$) und cg_{11} ($s_3{>}0$), beziehungsweise cg_{10} ($s_3{=}0$) und cg_{12} ($s_3{\leqslant}0$), erkennt die CG-Analyse von CGSeq durch Berücksichtigung des Booleschen Signalwertebereichs von s_3, dass die beiden CGs jeweils äquivalent sind. Die beiden hierdurch entstehenden CG-Gruppen schließen sich zudem über eine XOR-Beziehung bezüglich ihrer Erfüllung aus. Letzteres gilt auch für die CGs cg_7 ($s_2{\neq}0$) und cg_8 ($s_2{=}0$).

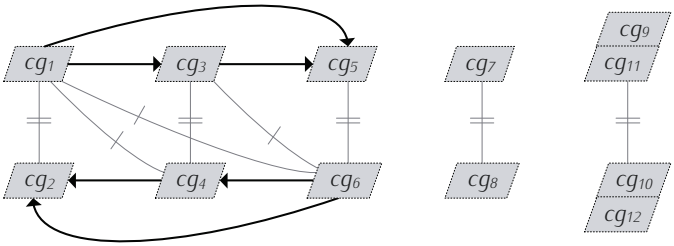


Abbildung 15: Logische Abhängigkeiten für das Beispiel-Modell aus Abbildung 13

Zur Erkennung solcher logischen Abhängigkeiten führt CGSeq eine Fallunterscheidung durch. Es findet ein paarweiser Vergleich aller vorhandenen CGs statt. Hierbei bilden drei Arten von Beziehungen, welche jeweils für ein CG-Paar gelten können, die Grundlage für daraus folgende Modifikationen des Abhängigkeitsgraphen:

$$\xrightarrow{\text{dom}}, \lightning, \lightning_{0/1} : E \times E \to \mathbb{B} \tag{55}$$

Diese in Infixnotation verwendeten Abbildungen geben für zwei CGs an, ob die jeweils durch sie beschriebene Beziehung gilt oder nicht. Für zwei CGs cg_a und cg_b steht $cg_a \xrightarrow{\text{dom}} cg_b$ hierbei für die Aussage, dass cg_b erfüllt ist, wann immer cg_a erfüllt ist. Im Folgenden wird diesbezüglich auch folgende Ausdrucksweise verwendet: cg_a dominiert cg_b. Die Abbildung $\lightning$ beschreibt den gegenseitigen Ausschluss, d.h. die CGs können unter keinen Umständen gleichzeitig erfüllt sein. Die Abbildung $\lightning_{0/1}$ ist eine striktere Form dieser Beziehung, welche zusätzlich verlangt, dass die beiden betrachteten CGs unter keinen Umständen gleichzeitig unerfüllt sind. Im Grunde spiegeln diese Abbildungen die Implikations-, XOR- und NAND-Beziehung in Anwendung für je zwei einzelne CGs wieder. Analog zu Definition 53 und Definition 54 gilt demnach auch:

$$\begin{aligned} &cg_a \lightning_{0/1} cg_b \Rightarrow cg_a \lightning cg_b \\ &cg_a \lightning_{0/1} cg_b \Leftrightarrow cg_b \lightning_{0/1} cg_a \\ &\quad cg_a \lightning cg_b \Leftrightarrow cg_b \lightning cg_a \end{aligned} \tag{56}$$

Aus der Überprüfung jeglicher CG-Paare cg_a und cg_b auf diese drei Beziehungstypen hin lassen sich folgendermaßen Modifikationen des Abhängigkeitsgraphen ableiten. Die Feststellung dieser Fälle ist hierbei entsprechend der Reihenfolge ihrer Auflistung priorisiert.

$$
\begin{aligned}
&1.\ \text{Äquivalenz: } cg_a \xrightarrow{dom} cg_b \wedge cg_b \xrightarrow{dom} cg_a \Rightarrow cgn(cg_a) = cgn(cg_b)\\
&2.\ \text{Implikation: } cg_a \xrightarrow{dom} cg_b \Rightarrow cgn(cg_a) \rightarrow cgn(cg_b)\\
&3.\ \text{XOR: } cg_a \not\downarrow_{0/1} cg_b \Rightarrow cgn(cg_a) \oplus cgn(cg_b)\\
&4.\ \text{NAND: } cg_a \not\downarrow cg_b \Rightarrow cgn(cg_a) \uparrow cgn(cg_b)
\end{aligned}
\tag{57}
$$

Um nun die Abbildungen $\xrightarrow{dom}$, $\not\downarrow$ und $\not\downarrow_{0/1}$ kompakt beschreiben zu können, seien folgende Definitionen vorausgesetzt. Den Wertebereich eines Signals s (bzw. dessen ν-ten Vektorelements), über sämtliche existierenden Zeitintervalle hinweg gesehen, beschreibt die Abbildung $D : S^{\mathbb{N}^+} \rightarrow \mathcal{P}(IV)$ folgendermaßen:

$$
D(s^\nu) = \bigcup_{ti \in T(s,\nu)} I_S(s, \nu, ti) \tag{58}
$$

Die aus diesem Wertebereich kleinstmöglichen und größtmöglichen Werte eines Signals seien durch $min, max : S^{\mathbb{N}^+} \rightarrow \mathbb{R}$, unter Verwendung der min/max-Funktionen aus Definition 33, wie folgt definiert:

$$
\begin{aligned}
&min(s^\nu) = min(D(s^\nu))\\
&max(s^\nu) = max(D(s^\nu))
\end{aligned}
\tag{59}
$$

Ein Wert $x \in \mathbb{R}$ ist darüber hinaus genau dann in einem Signalwertebereich enthalten, ausgedrückt durch $x \dot{\in} D(s^\nu)$, wenn der Wert in mindestens einem Intervall enthalten ist:

$$
x \dot{\in} D(s^\nu) \Leftrightarrow \exists\, iv \in D(s^\nu) : x \in iv \tag{60}
$$

Die Anzahl möglicher Signalwerte, ausgedrückt durch $|D(s^\nu)|$, ist folgendermaßen definiert:

$$
|D(s^\nu)| = \sum_{iv \in D(s^\nu)} |iv| \tag{61}
$$

Hierbei sind $|[a]| = 1$, $|[a;b]_q| = 1 + ((b-a)/q)$ und $|[a;b]| = \infty$.

Die Werte, welche die Wertebereiche zweier Signale (jeweils in Form einer Intervallmenge) gemeinsam haben, sind darüber hinaus über den speziellen Schnittmengenoperator $\dot{\cap}$ definiert:

$$
D(s_1^\nu) \dot{\cap} D(s_2^w) = \{\, x \in \mathbb{R} \mid x \dot{\in} D(s_1^\nu) \wedge x \dot{\in} D(s_2^w) \,\} \tag{62}
$$

Zuletzt sei die Funktion $BOOL : S^{\mathbb{N}^+} \rightarrow \mathbb{B}$ definiert, welche angibt, ob der Wertebereich eines Signals ein Boolescher ist:

$$
BOOL(s^\nu) = (0 \dot{\in} D(s^\nu) \wedge 1 \dot{\in} D(s^\nu) \wedge |D(s^\nu)| = 2) \tag{63}
$$

Die Definitionen der Beziehungen $\xrightarrow{\text{dom}}$, $\not\downarrow$ und $\not\downarrow_{0/1}$ sind fallbasiert in Tabelle 9 dargestellt. Geht es beispielsweise um die Prüfung der Beziehung $cg_a \xrightarrow{\text{dom}} cg_b$ und cg_b besteht aus der Konjunktion mehrerer Teilausdrücke, so muss aus der Erfüllung von cg_a die Erfüllung eines jeden Teilausdrucks von cg_b folgen, damit die Beziehung $cg_a \xrightarrow{\text{dom}} cg_b$ gilt. Ähnliche Zusammenhänge gibt es bei allen drei Beziehungstypen, falls cg_b mehrere Teilausdrücke logisch verknüpft (Tabellenzeilen 1/2).

Handelt es sich bei cg_b um einen relationalen Ausdruck, so sind bei allen drei Beziehungstypen jeweils drei Fälle zu unterscheiden: cg_a kann eine Konjunktion mehrerer Teilausdrücke, eine Disjunktion mehrerer Teilausdrücke oder ebenfalls ein relationaler Ausdruck sein (siehe Definition 14). Die Negation eines Teilausdrucks muss an dieser Stelle nicht berücksichtigt werden, da Negationen, wie in Abschnitt 3.3.2.1 beschrieben, in einem vorbereitenden Schritt von CGSeq entfernt werden.

Fall cg_b	*Beziehung* $cg_a \xrightarrow{\text{dom}} cg_b$	*Beziehung* $cg_a \not\downarrow cg_b$	*Beziehung* $cg_a \not\downarrow_{0/1} cg_b$
$\bigwedge_{i=1}^{n} \varphi_i$	$\forall \varphi_i : cg_a \xrightarrow{\text{dom}} \varphi_i$	$\exists \varphi_i : cg_a \not\downarrow \varphi_i$	$\forall \varphi_i : cg_a \not\downarrow_{0/1} \varphi_i$
$\bigvee_{i=1}^{n} \varphi_i$	$\exists \varphi_i : cg_a \xrightarrow{\text{dom}} \varphi_i$	$\forall \varphi_i : cg_a \not\downarrow \varphi_i$	$\forall \varphi_i : cg_a \not\downarrow_{0/1} \varphi_i$
$\xi_1 \bullet \xi_2$	*Fall 1:* $cg_a = \bigwedge_{i=1}^{n} \varphi_i$		
	$\exists \varphi_i : \varphi_i \xrightarrow{\text{dom}} cg_b$	$\exists \varphi_i : \varphi_i \not\downarrow cg_b$	$\forall \varphi_i : \varphi_i \not\downarrow_{0/1} cg_b$
	Fall 2: $cg_a = \bigvee_{i=1}^{n} \varphi_i$		
	$\forall \varphi_i : \varphi_i \xrightarrow{\text{dom}} cg_b$	$\forall \varphi_i : \varphi_i \not\downarrow cg_b$	$\forall \varphi_i : \varphi_i \not\downarrow_{0/1} cg_b$
	Fall 3: $cg_a = \xi_3 \circ \xi_4$		
	siehe Tabelle 10	siehe Tabelle 11	

Tabelle 9: Bedingungen zur Erfüllung der logischen Beziehungen $cg_a \xrightarrow{\text{dom}} cg_b$, $cg_a \not\downarrow cg_b$ und $cg_a \not\downarrow_{0/1} cg_b$ (mit $cg_a, cg_b \in E$) in Abhängigkeit von den in den Ausdrücken enthaltenen Operatoren (wobei $\bullet, \circ \in \{>, \geqslant, <, \leqslant, =, \neq\}$)

Im Falle einer Konjunktion gilt beispielsweise die Beziehung $cg_a \xrightarrow{dom} cg_b$, wenn es mindestens einen Teilausdruck in cg_a gibt, dessen Erfüllung die Erfüllung von cg_b zur Folge hat. Handelt es sich um eine Disjunktion, so müssen sich beispielsweise alle Teilausdrücke von cg_a mit cg_b gegenseitig ausschließen, damit $cg_a \nmid cg_b$ gilt. Bilden sowohl cg_a als auch cg_b relationale Ausdrücke, so ist eine spezifische Fallunterscheidung notwendig. Diese ist auszugsweise in Tabelle 10 sowie Tabelle 11 dargestellt. Tabelle 10 betrachtet hierbei die Beziehung $\xi_3 \circ \xi_4 \xrightarrow{dom} \xi_1 \bullet \xi_2$, für $\bullet \in \{>, \geqslant\}$ und $\circ \in \{>, \geqslant, <, \leqslant, =, \neq\}$. Tabelle 11 hingegen definiert die Beziehungen $\xi_1 \circ \xi_2 \nmid \xi_3 \bullet \xi_4$ und $\xi_1 \circ \xi_2 \nmid_{0/1} \xi_3 \bullet \xi_4$, für $\bullet \in \{>, =\}$ und $\circ \in \{>, \geqslant, <, \leqslant, =, \neq\}$.

Die Spalten in Tabelle 10 stellen vier verschiedene Fälle dar. Grundvoraussetzung ist dabei, dass es sich bei ξ_3 in $\xi_3 \circ \xi_4$ und bei ξ_1 in $\xi_1 \bullet \xi_2$ um Signale handelt. ξ_2 und ξ_4 können jeweils ein Signal oder eine Konstante sein. Daraus ergeben sich vier Kombinationsmöglichkeiten, beziehungsweise die in den Spalten dargestellten vier Fälle. Je nach Operator für $\bullet$ und $\circ$ gelten unterschiedliche Bedingungen, damit $\xi_3 \circ \xi_4 \xrightarrow{dom} \xi_1 \bullet \xi_2$ gilt. Zum Beispiel gilt die Beziehung $\xi_3 = \xi_4 \xrightarrow{dom} \xi_1 > \xi_2$ (mit ξ_2 und ξ_4 als Konstanten) genau dann, wenn es sich bei ξ_3 und ξ_1 einerseits um das selbe Signal handelt ($\xi_1 = \xi_3$) und die Konstante ξ_2 andererseits kleiner als die Konstante ξ_4 ist. Es gilt zum Beispiel: $s = 5 \xrightarrow{dom} s > 2$.

Die Fallunterscheidung in Tabelle 11 ist ähnlich gestaltet. Auch hier sind ξ_1 und ξ_3 in jedem Fall Signale sowie ξ_2 und ξ_4 wahlweise jeweils ein Signal oder eine Konstante. Die Bedingungen, nach denen es sich bei zwei der in beiden Ausdrücken enthaltenen Signale um das selbe Signal handeln muss, sind in dieser Tabelle allerdings bereits in den Spaltenüberschriften verarbeitet. Ein Beispiel: Es seien die Ausdrücke $\xi_1 > \xi_2$ und $\xi_3 \leqslant \xi_4$ betrachtet, wobei es sich bei allen enthaltenen Parametern um Signale handelt. Die beiden Ausdrücke schließen sich wechselseitig gegenseitig aus ($\nmid_{0/1}$), wenn nicht nur ξ_1 das selbe Signal wie ξ_3, sondern auch ξ_2 das selbe Signal wie ξ_4 ist. Ein konkretes Beispiel hierzu ist $s_1 > s_2 \nmid_{0/1} s_1 \leqslant s_2$. Bezeichnen ξ_2 und ξ_4 allerdings unterschiedliche Signale, so kann lediglich der einfache gegenseitige Ausschluss ($\nmid$) gelten. Hierzu müsste der kleinstmögliche Wert des Signals ξ_2 größer als oder gleich dem größtmöglichen Wert des Signals ξ_4 sein. Auch hierzu sei ein konkretes Beispiel angeführt: $s_1 > s_2 \nmid s_1 \leqslant s_4$ gilt, wenn beispielsweise $\min(\xi_2) = 7$ und $\max(\xi_4) = 5$ sind.

Im Rahmen der vorgestellten Arbeit wurde die Fallunterscheidung selbstverständlich für alle relationalen Operatoren vorgenommen – sie wird an dieser Stelle jedoch aus Platzgründen nicht voll umfassend vorgestellt.

$\bullet$	$\xi_1, \xi_3 \in S^{\mathbb{N}^+} \wedge \xi_2, \xi_4 \in \mathbb{R}$		$\xi_1, \xi_2, \xi_3, \xi_4 \in S^{\mathbb{N}^+}$		$\xi_1, \xi_2, \xi_3 \in S^{\mathbb{N}^+} \wedge \xi_4 \in \mathbb{R}$		$\xi_1, \xi_3, \xi_4 \in S^{\mathbb{N}^+} \wedge \xi_2 \in \mathbb{R}$	
$>$	$\circ$	$\xi_1 = \xi_3$	$\circ$		$\circ$		$\circ$	$\xi_2 < \max(\min(\xi_3), \min(\xi_4))$
	$>$	$\xi_2 \leqslant \xi_4$	$>$	$\xi_1 = \xi_3 \wedge \xi_2 = \xi_4$	$=$	$(\xi_1 = \xi_3 \wedge \max(\xi_2) < \xi_4) \vee$ $(\xi_2 = \xi_3 \wedge \min(\xi_1) > \xi_4)$	$>, \geqslant$	$\xi_1 = \xi_3$
	$\neq$	$\xi_2 = \xi_4 = \min(\xi_3)$			$>$	$\xi_1 = \xi_3 \wedge \max(\xi_2) \leqslant \xi_4$	$<, \leqslant$	$\xi_1 = \xi_4$
			$<$	$\xi_2 = \xi_3 \wedge \xi_1 = \xi_4$	$\geqslant$	$\xi_1 = \xi_3 \wedge \max(\xi_2) < \xi_4$		
	$\geqslant, =$	$\xi_2 < \xi_4$			$<$	$\xi_2 = \xi_3 \wedge \min(\xi_1) \geqslant \xi_4$	$=$	$\xi_1 = \xi_3 \vee \xi_1 = \xi_4$
					$\leqslant$	$\xi_2 = \xi_3 \wedge \min(\xi_1) > \xi_4$		
$\geqslant$	$\circ$	$\xi_1 = \xi_3$	$\circ$		$\circ$		$\circ$	$\xi_2 \leqslant \max(\min(\xi_3), \min(\xi_4))$
	$=, >, \geqslant$	$\xi_2 \leqslant \xi_4$	$=$	$(\xi_1 = \xi_3 \wedge \xi_2 = \xi_4) \vee$ $(\xi_2 = \xi_3 \wedge \xi_1 = \xi_4)$	$=$	$(\xi_1 = \xi_3 \wedge \max(\xi_2) \leqslant \xi_4) \vee$ $(\xi_2 = \xi_3 \wedge \min(\xi_1) \geqslant \xi_4)$	$>, \geqslant$	$\xi_1 = \xi_3$
	$\neq$	$\min(\xi_1) = \xi_4$ $\wedge (\exists [a;b] \in D(\xi_1)$ $\Rightarrow \xi_2 \leqslant \xi_4) \wedge$ $(\forall [\xi_4;b]_q \in D(\xi_1):$ $\xi_2 \leqslant \xi_4 + q)$ $\wedge (\forall [a] \in D(\xi_1):$ $a > \xi_4 \Rightarrow \xi_2 \leqslant a)$					$<, \leqslant$	$\xi_1 = \xi_4$
			$>, \geqslant$	$\xi_1 = \xi_3 \wedge \xi_2 = \xi_4$	$>, \geqslant$	$\xi_1 = \xi_3 \wedge \max(\xi_2) \leqslant \xi_4$	$=$	$\xi_1 = \xi_3 \vee \xi_1 = \xi_4$
			$<, \leqslant$	$\xi_2 = \xi_3 \wedge \xi_1 = \xi_4$	$<, \leqslant$	$\xi_2 = \xi_3 \wedge \min(\xi_1) \geqslant \xi_4$		

Tabelle 10: Bedingungen zur Erfüllung der logischen Beziehung $\xi_3 \circ \xi_4 \xrightarrow{\text{dom}} \xi_1 \bullet \xi_2$ (Auszug für $\bullet \in \{>, \geqslant\}$)

$\bullet$	$\xi_2 \in \mathbb{R}$					$\xi_2 \in S^{\mathbb{N}^+}$			
	$\xi_4 \in \mathbb{R} \wedge \xi_1 = \xi_3$		$\xi_4 \in S^{\mathbb{N}^+}$			$\xi_4 \in S^{\mathbb{N}^+}$			$\xi_4 \in \mathbb{R}$
			$\xi_1 = \xi_3$		$\xi_1 = \xi_4$	$\xi_1 = \xi_3$		...	...
$>$	$\circ$		$\circ$		...	$\circ$		...	...
	$=$	$\lightning_{0/1}$: $\mathrm{BOOL}(\xi_1) \wedge \xi_2 = \xi_4 = 0$ $\lightning$: $\xi_2 \geqslant \xi_4$	$=$ $<$ $\leqslant$	$\lightning$: $\xi_2 \geqslant \max(\xi_4)$		$=$	$\lightning$: $\min(\xi_2) \geqslant \min(\max(\xi_1), \max(\xi_4))$		
	$\neq$	$\lightning_{0/1}$: $\mathrm{BOOL}(\xi_1) \wedge \xi_2 = 0 \wedge \xi_4 = 1$				$<$	$\lightning$: $\min(\xi_2) \geqslant \max(\xi_4)$		
	$<$	$\lightning_{0/1}$: $\mathrm{BOOL}(\xi_1) \wedge \xi_2 = 0 \wedge \xi_4 = 1$ $\lightning$: $\xi_2 \geqslant \xi_4$				$\leqslant$	$\lightning_{0/1}$: $\xi_2 = \xi_4$ $\lightning$: $\min(\xi_2) \geqslant \max(\xi_4)$		
	$\leqslant$	$\lightning_{0/1}$: $\xi_2 = \xi_4$ $\lightning$: $\xi_2 \geqslant \xi_4$							
$=$	$\circ$		$\circ$		...	$\circ$		...	...
	$=$	$\lightning_{0/1}$: $\mathrm{BOOL}(\xi_1) \wedge \xi_2 \neq \xi_4$ $\lightning$: $\xi_2 \neq \xi_4$	$=$	$\lightning$: $\neg(\xi_2 \dot{\in} \mathrm{D}(\xi_4))$		$=$	$\lightning$: $\mathrm{D}(\xi_2) \dot{\cap} \mathrm{D}(\xi_4) = \emptyset$		
	$\neq$	$\lightning_{0/1}$: $\xi_2 = \xi_4$	$>$	$\lightning$: $\xi_2 \leqslant \min(\xi_4)$		$\neq$	$\lightning_{0/1}$: $\xi_2 = \xi_4$		
	$>$	$\lightning_{0/1}$: $\mathrm{BOOL}(\xi_1) \wedge \xi_2 = \xi_4 = 0$ $\lightning$: $\xi_2 \leqslant \xi_4$	$\geqslant$	$\lightning$: $\xi_2 < \min(\xi_4)$		$>$	$\lightning$: $\min(\xi_4) \geqslant \min(\max(\xi_1), \max(\xi_2))$		
	$\geqslant$	$\lightning_{0/1}$: $\mathrm{BOOL}(\xi_1) \wedge \xi_2 = 0 \wedge \xi_4 = 1$ $\lightning$: $\xi_2 < \xi_4$	$<$	$\lightning$: $\xi_2 \geqslant \max(\xi_4)$		$\geqslant$	$\lightning$: $\min(\xi_4) > \min(\max(\xi_1), \max(\xi_2))$		
	$<$	$\lightning_{0/1}$: $\mathrm{BOOL}(\xi_1) \wedge \xi_2 = \xi_4 = 1$ $\lightning$: $\xi_2 \geqslant \xi_4$	$\leqslant$	$\lightning$: $\xi_2 > \max(\xi_4)$		$<$	$\lightning$: $\min(\xi_4) \leqslant \min(\max(\xi_1), \max(\xi_2))$		
	$\leqslant$	$\lightning_{0/1}$: $\mathrm{BOOL}(\xi_1) \wedge \xi_2 = 1 \wedge \xi_4 = 0$ $\lightning$: $\xi_2 > \xi_4$				$\leqslant$	$\lightning$: $\min(\xi_4) < \min(\max(\xi_1), \max(\xi_2))$		

Tabelle 11: Bedingungen zur Erfüllung der logischen Beziehungen $\xi_1 \circ \xi_2 \not\lightning \xi_3 \bullet \xi_4$ und $\xi_1 \circ \xi_2 \lightning_{0/1} \xi_3 \bullet \xi_4$ (Auszug aus der Fallunterscheidung für $\bullet \in \{>, =\}$) - Grundvoraussetzung ist in jedem Fall: ξ_1 und ξ_3 sind Signale ($\xi_1, \xi_3 \in S^{\mathbb{N}^+}$)

3.3.2.4 *Semantische Abhängigkeiten*

Die Erfassung semantischer Abhängigkeiten stellt eine Erweiterung des durch die Sammlung logischer Abhängigkeiten entstandenen CG-Abhängigkeitsgraphen dar. Während bei der Prüfung logischer Abhängigkeiten zweier CGs vorausgesetzt wird, dass beide Ausdrücke mindestens ein Signal miteinander teilen, so ist dies im Falle semantischer Abhängigkeiten nicht notwendig. Diese ergeben sich nämlich, wie bereits angedeutet, aus der Semantik des betrachteten SL/TL-Modells.

Hierzu sei einleitend erneut das Beispiel-Modell aus Abbildung 13 betrachtet. Den vollständigen CG-Abhängigkeitsgraph für dieses Modell, also inklusive der semantischen Abhängigkeiten, ist in Abbildung 16 dargestellt. Vergleicht man diesen mit dem rein aus logischen Abhängigkeiten bestehenden Abhängigkeitsgraph in Abbildung 15, so fällt auf, dass es zu einem Zusammenschluss der CGs cg_5, cg_7, cg_{10} und cg_{12}, beziehungsweise cg_6, cg_8, cg_9 und cg_{11}, zu je einer Gruppe äquivalenter CGs gekommen ist. Ein Blick in das Beispiel-Modell klärt auf, wieso: cg_5 und cg_7 beziehen sich auf ein Ein- oder Ausgangssignal des unteren Relational-Operator-Blocks. Aus der Funktionalität dieses Blocks lässt sich ableiten, dass bei Erfüllung von cg_5 auch cg_7 erfüllt ist. Da cg_7 nur genau dann erfüllt ist, wenn auch cg_5 ist, sind diese CGs als äquivalent anzusehen. Das gleiche gilt für cg_6 und cg_8. Die weiteren Äquivalenzen ergeben sich analog hierzu durch Betrachtung des NOT- und des Switch-Blocks.

Zur Ermittlung semantischer Abhängigkeiten geht CGSeq folgendermaßen vor: Da die Prüfung dieser Abhängigkeiten Block-spezifisch ist, wird für jeden Block der Repräsentation des betrachteten SL/TL-Modells

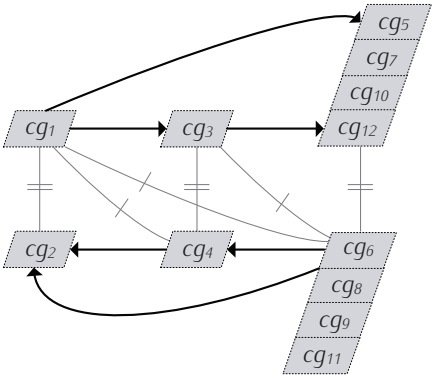


Abbildung 16: Vollständiger Abhängigkeitsgraph für das Beispielmodell aus Abbildung 13

eine entsprechende Analyse durchgeführt. Die Blöcke werden dabei in der Ausführungsreihenfolge, wie sie auch innerhalb einer Modellausführung gilt, abgearbeitet. Die Analyse eines jeden Blocks untersucht die CGs, welche mit dessen Ein- und Ausgangssignalen in Verbindung stehen und leitet gegebenenfalls Beziehungen zwischen diesen CGs für den CG-Abhängigkeitsgraphen ab. Blöcke mit zeitlicher Dynamik, wie ein Unit-Delay-Block, werden hierbei ausgelassen. Wäre dies nicht so, müssten die CG-Beziehungen oder die CGs im Abhängigkeitsgraphen über Zeitangaben verfügen. Da dies, insbesondere im Fall von Schleifen im Modell, die Komplexität eines CG-Abhängigkeitsgraphen unter Umständen deutlich vergrößern würde, wird an dieser Stelle auf ein erweiterndes Konzept verzichtet. Die Realisierung einer entsprechenden Erweiterung könnte Gegenstand von Folgearbeiten sein.

Werden bei der Analyse eines Blocks semantische Abhängigkeiten in Form von Implikationsbeziehungen registriert, so werden, wie in Abschnitt 3.3.2.2 erklärt, zwei verschiedene Typen von Implikationsbeziehungen verwendet: vorwärts- und rückwärtsgerichtete Implikationsbeziehungen. Betrachtet man die CGs, durch Zuordnung zu den in ihnen jeweils enthaltenen Signalen, eingegliedert in das betrachtete SL/TL-Modell, so folgen vorwärtsgerichtete (normale) Implikationsbeziehungen der Richtung des Signalflusses im Modell. Rückwärtsgerichtete Implikationen stehen diesem entgegen. Anders ausgedrückt: Folgt aus der Erfüllung eines CGs, welches ein Ausgangssignal eines Blocks referenziert, die Erfüllung eines CGs, das ein Eingangssignal des selben Blocks referenziert, so handelt es sich um eine rückwärtsgerichtete Implikation. Ein Beispiel: Ein CG, das verlangt, dass das Ausgangssignal eines AND-Blocks eine 1 enthält (Wahrheitswert *wahr* am Ausgang), dominiert ein jedes CG, das beschreibt, dass eines der Eingangssignale des Blocks ungleich 0 ist (Wahrheitswerte *wahr* an den Eingängen). Während vorwärtsgerichtete Implikationen vor allem der Bildung einer Abarbeitungsreihenfolge der CGs für die Testdatensuche dienen, werden rückwärtsgerichtete Implikationen durch CGSeq einzig und allein für die erwähnte ergänzende Erreichbarkeitsprüfung genutzt.

Die Blockanalysen zur semantischen CG-Abhängigkeit fügen der Gesamtmenge an CGs des Weiteren unter Umständen weitere CGs hinzu. Ein jedes solches CG wird *virtuelles Überdeckungsziel* (VCG, Plural: VCGs) genannt. Ein VCG dient im Abhängigkeitsgraph der „Brückenbildung" zwischen anderen CGs. Zum Verständnis dieser Idee sei das Beispiel-Modell in Abbildung 17 betrachtet. Ein partieller CG-Abhängigkeitsgraph für einige der ableitbaren CGs ist ebenfalls in dieser Abbildung dargestellt. Von dem linken der beiden Switch-Blöcke leitet sich unter anderem das CG cg_2 ab, welches den Wert 1 für das Kontrollsignal am zweiten Eingang des Blocks verlangt. Dieser Switch-Block erkennt bei seiner Analyse semantischer Abhängigkeiten zwar keine Zusammenhänge zwischen existierenden CGs an seinen Ein- und Ausgängen, jedoch nutzt er die

durch SIA für seine zwei Dateneingänge gewonnen Wertebereichsangaben, um hieraus VCGs für sein Ausgangssignal zu erstellen. Daher entsteht vcg_1 mit dem Ausdruck $s_5{=}2$. Da die Erfüllung von cg_2 für die Funktionalität des Switch-Blocks eine Durchleitung des ersten Dateneingangs bedeutet, lässt sich schlussfolgern: cg_2 dominiert vcg_1. Der nachfolgende Gain-Block kann bei seiner Analyse ebenfalls keinen direkten Zusammenhang zwischen existierenden CGs herstellen. Allerdings können auch hier VCGs entstehen. Da dieser Gain-Block eine Multiplikation seiner Eingabe mit dem Faktor 10 durchführt, kann das VCG vcg_2 mit dem Ausdruck $s_6{=}20$ abgeleitet werden. Der im Signalfluss darauffolgende Switch-Block überträgt dieses VCG auf sein Ausgangssignal, mit der Bedingung dass gleichzeitig auch sein Kontrolleingang entsprechend gestaltet ist (Konjunktion mit cg_4). Aus der gleichzeitigen Erfüllung von cg_4 und vcg_2 folgt also die Erfüllung von vcg_3, welches den Ausdruck $s_8{=}20$ besitzt. Der Abhängigkeitsgraph wird nach der Erfassung aller semantischen Abhängigkeiten komplettiert, indem für jedes hinzugefügte VCG eine Überprüfung möglicher logischer Abhängigkeiten mit allen anderen VCGs und CGs durchgeführt wird. Daher wird auch erkannt, dass vcg_3 das CG cg_5 dominiert. Aus Sicht einer Testdatensuche erscheint es mit Blick auf den entstandenen Abhängigkeitsgraphen nun sinnvoll, die CGs cg_1 und cg_3 gleichzeitig als Ziel anzuvisieren. Werden nämlich dabei die gesuchten Testdaten gefunden, so werden alle in der Abbildung dargestellten CGs auf einen Schlag erreicht.

Die Block-spezifischen Analysen semantischer Abhängigkeiten, inklusive der Erstellung von VCGs, werden im Folgenden, zumindest auszugs-

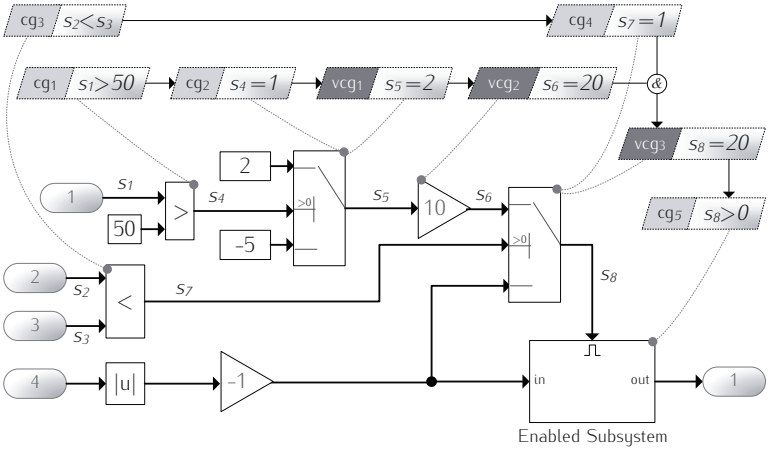


Abbildung 17: Block-spezifisches Hinzufügen virtueller Überdeckungsziele (Beispiel)

weise, definiert. Hierzu sei auf drei Hilfskonstrukte oder -abbildungen verwiesen, welche in den Definitionen aufgegriffen werden. Die Menge all derer CGs, in deren Ausdrücken ein bestimmtes Signal vorkommt, sei durch $SCG : S^{\mathbb{N}^+} \to \mathcal{P}(E)$ gegeben. Da es in den Blockanalysen möglich sein soll, eine solche Menge um VCGs zu erweitern, sei der Operator $\biguplus : \mathcal{P}(E) \times E \to \mathcal{P}(E)$ eingeführt. Dieser fügt einer CG-Menge ein weiteres CG oder VCG hinzu. Ist im Folgenden übrigens von CGs die Rede, so sind damit auch eventuell vorhandene VCGs gemeint – es sei denn, es wird explizit unterschieden. Zuletzt wird die Funktion $\curvearrowright : E \times S^{\mathbb{N}^+} \times S^{\mathbb{N}^+} \to E$ benötigt. Diese tauscht in dem Ausdruck eines CG alle Vorkommnisse eines bestimmten Signals durch ein anderes gewünschtes Signal aus. Tabelle 12 enthält die Definition der Blockanalysen, im Auszug für die Blöcke Inport und Logical Operator.

Inport. Für diesen Block sei zunächst der Fall betrachtet, dass der Block sich in einem virtuellen Subsystem befindet (Fall 1). Im Grunde überträgt die Blockanalyse in diesem Fall jegliche CGs von dem entsprechenden Eingangssignal des Subsystems auf das Ausgangssignal des Inport-Blocks (Fall 1.2). Ein CG cg des Subsystem-Eingangssignals und das korrespondierende VCG vcg des Inport-Ausgangssignals werden in Folge im Abhängigkeitsgraphen als äquivalent aufgefasst. Existiert allerdings bereits ein CG rcg am Ausgangssignal, welches äquivalent zu vcg ist, so wird vcg nicht hinzugefügt und stattdessen rcg als äquivalent mit cg aufgefasst (Fall 1.1). Befindet sich der Inport-Block ein einem bedingt ausgeführten Subsystem, wie einem Enabled Subsystem (Fall 2), so müssen bei der Erfassung des Zusammenhangs zwischen cg und vcg/rcg (Fall 2.2/Fall 2.1) auch all die CGs acg berücksichtigt werden, deren Erfüllung eine Aktivierung des Subsystems bedeuten. Ist eines davon erfüllt (Disjunktion) und zusätzlich cg erfüllt (Konjunktion), dann ist auch vcg/rcg erfüllt (Implikation).

Logical Operator. In der Tabelle ist die Definition ausschließlich für den AND-Operator und den Fall, dass der Block mehrere Eingänge besitzt, angegeben. Das Vorgehen ist allerdings relativ einfach auf die weiteren Operatortypen und die Blockvariante mit nur einem Eingang zu übertragen. Im dargestellten Fall werden je Vektorelement der Eingangssignale folgende drei Zusammenhänge erfasst.

1. Von allen Eingangssignalen des Blocks (Konjunktion) muss mindestens ein CG icg, welches einer Belegung des Blockeingangs mit dem Wahrheitswert *wahr* (Signalwert ungleich 0) gleichzusetzen ist, erfüllt sein (Disjunktion), damit in Folge auch all die CGs ocg des Ausgangssignals erfüllt sind, die durch den Wahrheitswert *wahr* am Ausgang (Signalwert gleich 1) dominiert werden.

2. Ist mindestens ein CG icg der Eingangssignale erfüllt (Disjunktion), welches einer Belegung des entsprechenden Eingangs mit dem Wahrheitswert *falsch* gleichkommt (Signalwert gleich 0), so sind

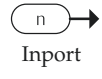
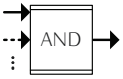
Blocktyp	*Semantische Abhängigkeiten und virtuelle Überdeckungsziele*
 Inport	$\forall v \in \mathbb{N}_{[1,d(s_1)]}: \forall cg \in SCG(s_1^v)$: **Fall 1:** $\exists sb \in B: b \in BC_{sb} \wedge bt_{sb}$=Subsystem **Fall 1.1:** $\exists rcg \in SCG(s_2^v)$: $rcg \xrightarrow{dom} vcg \wedge vcg \xrightarrow{dom} rcg$ **Abh.:** $cgn(cg) \cup cgn(rcg)$ **Fall 1.2:** sonst **VCG:** $SCG(s_2^v) \uplus vcg$ **Abh.:** $cgn(cg) \cup cgn(vcg)$ **Fall 2:** $\exists sb \in B: b \in BC_{sb} \wedge bt_{sb}$=Enabled Subsystem **Fall 2.1:** $\exists rcg \in SCG(s_2^v)$: $rcg \xrightarrow{dom} vcg \wedge vcg \xrightarrow{dom} rcg$ **Abh.:** $(cgn(cg) \wedge (\bigvee_{\forall acg \in A} cgn(acg))) \rightarrow rcg$ **Fall 2.2:** sonst **VCG:** $SCG(s_2^v) \uplus vcg$ **Abh.:** $(cgn(cg) \wedge (\bigvee_{\forall acg \in A} cgn(acg))) \rightarrow vcg$ **mit** $s_1 = sig_{in}(sb,n) \wedge s_2 = sig_{out}(b,1) \wedge vcg = \curvearrowright(cg, s_1^v, s_2^v)$ $\wedge A = \bigcup_{\forall w \in \mathbb{N}_{[1,d(s_3)]}} \{ccg \mid ccg \in SCG(s_3^w) \wedge ccg \xrightarrow{dom} \left\langle \bigvee_{\forall u \in \mathbb{N}_{[1,d(s_3)]}} s_3^u > 0 \right\rangle\}$
 Logical Operator	Exemplarisch für $\bullet = \wedge$ und $n > 1$: $\forall v \in \mathbb{N}_{[1,maxd_{in}(b)]}$: **Abh.:** *(1)* $(\bigwedge_{\forall p \in POS} (\bigvee_{\forall icg \in SCG(s_1^w):\ icg \xrightarrow{dom} \langle s_1^w \neq 0 \rangle} cgn(icg)))$ $\rightarrow (\bigwedge_{\forall ocg \in SCG(s_2^v):\ \langle s_2^v = 1 \rangle \xrightarrow{dom} ocg} cgn(ocg))$ *(2)* $(\bigvee_{\forall p \in POS: \forall icg \in SCG(s_1^w):\ icg \xrightarrow{dom} \langle s_1^w = 0 \rangle} cgn(icg)) \rightarrow (\bigwedge_{\forall ocg \in SCG(s_2^v):\ \langle s_2^v = 0 \rangle \xrightarrow{dom} ocg} cgn(ocg))$ *(3)* $\forall ocg \in SCG(s_2^v)$: $ocg \xrightarrow{dom} \langle s_2^v = 1 \rangle$: $\forall p \in POS: \forall icg \in SCG(s_1^w): \langle s_1^w \neq 0 \rangle \xrightarrow{dom} icg$: $cgn(icg) \dashleftarrow cgn(ocg)$ **mit** $s_1 = sig_{in}(b,p) \wedge s_2 = sig_{out}(b,1)$ $\wedge POS = \mathbb{N}_{[1,\lvert IP_b \rvert]} \wedge w = vc(v, s_1)$

Tabelle 12: Block-spezifische Abhängigkeiten zwischen an den Ein- und Ausgangssignalen eines Blocks anliegenden Überdeckungszielen und Bildung virtueller Überdeckungsziele (Auszug)

auch alle CGs ocg des Ausgangssignals erfüllt, die durch den Wahrheitswert *falsch* am Ausgang (Signalwert gleich 0) dominiert werden.

3. Die Erfüllung aller CGs ocg des Ausgangssignals, die den Wahrheitswert *wahr* am Ausgang (Signalwert gleich 1) zur Folge haben, implizieren jeweils die Erfüllung aller CGs icg der Blockeingangssignale, welche erfüllt wären, wenn der Wahrheitswert des jeweiligen Eingangs *wahr* ist (Signalwert ungleich 0). Bei dieser Implikationsbeziehung handelt es sich allerdings um eine rückwärtsgerichtete Implikation.

Die auf diesem Wege erfolgende Ermittlung semantischer Abhängigkeiten formt gemeinsam mit zuvor ermittelten logischen Abhängigkeiten den finalen CG-Abhängigkeitsgraphen eines SL/TL-Modells. Dieser dient sowohl der ergänzenden Erreichbarkeitsprüfung als auch der Bildung einer Abarbeitungsreihenfolge für die CGs. Gegebenenfalls im Abhängigkeitsgraphen entstandene Zyklen der Form $CGN_a \rightarrow ... \rightarrow CGN_n$ werden übrigens in einem letzten Schritt beseitigt. In einem Zyklus enthaltene CGs, beziehungsweise deren Graphknoten, sind de facto äquivalent und können dementsprechend auch im Graph vereinigt werden – das heißt: $CGN_{a,...,n} = CGN_a \cup ... \cup CGN_n$. Ein Zyklus wird wohlgemerkt nicht beseitigt, wenn eine der im Zyklus enthaltenen Implikationsbeziehungen rückwärtsgerichtet ist.

3.3.2.5 *Ergänzende Erreichbarkeitsprüfung*

Die in Abschnitt 3.2 vorgestellte SIA-Technik dient einer Prüfung der Erreichbarkeit der betrachteten CGs. Allerdings können über diesen Ansatz unter Umständen nicht alle unerreichbaren CGs identifiziert werden. In manchen solcher Fälle kann die CGSeq-Technik helfen. Der durch sie erzeugte CG-Abhängigkeitsgraph kann genutzt werden, um aus der durch SIA festgestellten Unerreichbarkeit (UNSAT) oder garantierten Erfüllung (SAT) eines CG Rückschlüsse auf andere CGs zu ziehen.

Zu diesem Zweck werden die Propagierungsregeln der folgend vorgestellten drei Kategorien „Äquivalenz“, „Implikation“ und „XOR“ auf jeden Knoten eines CG-Abhängigkeitsgraphen, der ein unerfüllbares oder stets erfülltes CG enthält, angewendet. Im Falle der vierten Kategorie („Widerspruch“) werden alle Knoten des Graphen, welche zu diesem Zeitpunkt weder als unerfüllbar noch als stets erfüllt gelten, betrachtet. Für alle Propagierungsregeln gilt: Erfolgt eine Propagierung eines CG-Zustands auf die CGs eines anderen Graphknotens, so wird auch dieser bezüglich möglicher weiterer Propagierungsmöglichkeiten untersucht. Der Nachweis der Gültigkeit aller Regeln kann relativ trivial über Wahrheitstabellen geführt werden.

Äquivalenz. Wurde durch SIA festgestellt, dass ein CG unerreichbar (Fall 1) oder stets erreicht (Fall 2) ist, so gilt dieser Zustand auch für alle anderen CGs der Gruppe äquivalenter CGs, welcher dieses CG angehört.

$$\forall n \in V_{CG}\colon \forall\, cg_1, cg_2 \in n\colon \tag{64}$$
$$\textbf{Fall 1:}\ st(cg_1) = UNSAT \Rightarrow (st(cg_2) = UNSAT \wedge U@(cg_2) = U@(cg_1))$$
$$\textbf{Fall 2:}\ st(cg_1) = SAT \Rightarrow (st(cg_2) = SAT \wedge S@(cg_2) = S@(cg_1))$$

Hierauf basierend sei die Funktion st, die den Zustand eines einzelnen CG angibt, für die weiteren Betrachtungen auch für CG-Gruppen definiert. Gleichermaßen werden auch die Funktionen U@/S@, welche die Zeitschritte einer Unerfüllbarkeit/garantierten Erfüllung angeben, übertragen.

$$st : V_{CG} \to \{UNSAT, SAT, OPEN\}$$
$$st(n) = \begin{cases} UNSAT & ,\forall\, cg \in n\colon st(cg) = UNSAT \\ SAT & ,\forall\, cg \in n\colon st(cg) = SAT \\ OPEN & ,sonst \end{cases} \tag{65}$$
$$S@, U@ : V_{CG} \to \mathcal{P}(\mathbb{N})$$
$$S@(n) = \bigcap_{cg \in n} S@(cg) = \bigcup_{cg \in n} S@(cg)\ ,\quad U@(n) = \bigcap_{cg \in n} U@(cg) = \bigcup_{cg \in n} U@(cg)$$

Implikation. Wird ein unerreichbares CG durch ein anderes CG impliziert, so ist auch dieses CG, von dem die Implikation ausgeht, unerreichbar (Fall 1). Handelt es sich um eine Implikation über einen Disjunktoren, so gilt dies für alle CGs, die im Abhängigkeitsgraph zu diesem Junktoren führen. Ein impliziertes CG wird hingegen als stets erreicht erkannt, wenn das CG, von dem die Implikation ausgeht, stets erreicht ist (Fall 2). Führt diese Implikationsbeziehung über einen Disjunktor, so müsste eines der zu dem Junktor führenden CGs stets erreicht sein. Führt sie über einen Konjunktor, so müssten alle zu dem Junktor führenden CGs stets erreicht sein. Fall 1 und 2 gelten auch für rückwärtsgerichtete Implikationsbeziehungen.

$$\forall n_1 \in (V_{CG} \cup V_K \cup V_D), n_2 \in V_{CG}\colon (n_1, n_2) \in (E_{\rightarrow} \cup E_{\dashleftarrow})\colon$$
$$\textbf{Fall 1:}\ st(n_2) = UNSAT \Rightarrow (n_1 \notin V_K \Rightarrow (unsat(n_1) \wedge U@(n_1) = U@(n_2)))$$
$$\textbf{Fall 2:}\ sat(n_1) \Rightarrow (st(n_2) = SAT \wedge S@(n_2) = S@(n_1))$$
$$\text{mit}\ \ sat, unsat : (V_{CG} \cup V_K \cup V_D) \to \mathbb{B} \tag{66}$$

$$\text{sat}(e) = \begin{cases} \text{st}(e) = \text{SAT} & , e \in V_{CG} \\ \exists (x,e) \in E_{\rightarrow} : \text{sat}(x) & , e \in V_D \\ \forall (x,e) \in E_{\rightarrow} : \text{sat}(x) & , e \in V_K \end{cases}$$

$$\text{unsat}(e) = \begin{cases} \text{st}(e) = \text{UNSAT} & , e \in V_{CG} \\ \forall (x,e) \in E_{\rightarrow} : \text{unsat}(x) & , e \in V_D \\ \emptyset & , e \in V_K \end{cases}$$

XOR. Schließen sich zwei CGs wechselseitig gegenseitig aus und ist eines der beiden CGs unerreichbar, so gilt das andere CG als stets erreicht (Fall 1). Ist eines der beiden CGs hingegen stets erreicht und gilt dies für alle betrachteten Zeitschritte einer Modellausführung, so ist das andere an der XOR-Beziehung beteiligte CG unerreichbar (Fall 2).

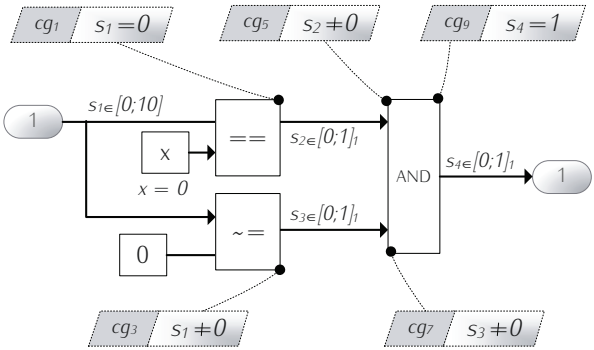
$$\begin{aligned} &\forall n_1, n_2 \in V_{CG} : (n_1, n_2) \in E_{\oplus} : \\ &\textbf{Fall 1:}\ \text{st}(n_1) = \text{UNSAT} \Rightarrow (\text{st}(n_2) = \text{SAT} \wedge \text{S@}(n_2) = \text{U@}(n_1)) \\ &\textbf{Fall 2:}\ (\text{st}(n_1) = \text{SAT} \wedge \text{S@}(n_1) = \text{AT}) \\ &\qquad \Rightarrow (\text{st}(n_2) = \text{UNSAT} \wedge \text{U@}(n_2) = \text{AT}) \end{aligned} \tag{67}$$

Widerspruch. Die CGs einer CG-Gruppe sind unerreichbar, wenn von dieser Gruppe Implikationsbeziehungen ausgehen und es in der Menge aller, auch transitiv implizierten CGs ein Paar zweier CGs bilden lässt, die sich bezüglich ihrer Erfüllung gegenseitig ausschließen. Hierbei werden neben normalen, vorwärtsgerichteten Implikationsbeziehungen auch rückwärtsgerichtete Implikationsbeziehungen berücksichtigt.

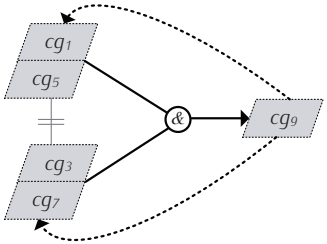
$$\begin{aligned} &\forall n \in V_{CG} : \\ &(\exists n_1, n_2 \in \text{IG}(n) : (n_1, n_2) \in E_{\oplus} \cup E_{\uparrow}) \Rightarrow (\text{st}(n) = \text{UNSAT} \wedge \text{U@}(n) = \text{AT}) \\ &\text{mit}\ \text{IG} : (V_{CG} \cup V_D) \rightarrow \mathcal{P}(V_{CG}) \end{aligned}$$

$$\text{IG}(e) = \bigcup_{(e,x) \in (E_{\rightarrow} \cup E_{\dashleftarrow})} \begin{cases} \{x\} \cup \text{IG}(x) & , x \in V_{CG} \\ \text{IG}(x) & , x \in V_D \\ \emptyset & , \text{sonst} \end{cases} \tag{68}$$

Abbildung 18 zeigt ein Beispiel für eine solche Situation. Dargestellt sind ein Modellausschnitt mit einem Teil der hieraus ableitbaren CGs und der dazugehörige CG-Abhängigkeitsgraph. Die durch SIA ermittelten Signalwertebereiche sind ebenfalls im Modellausschnitt zu sehen. Basierend auf diesen Wertebereichen erkennt die Erreichbarkeitsanalyse von SIA für keines der dargestellten CGs eine Unerreichbarkeit. Die ergänzende Erreichbarkeitsprüfung von CGSeq stellt allerdings durch Anwendung



(a) Modellausschnitt mit abgeleiteten Überdeckungszielen (Auszug)



(b) Abhängigkeitsgraph für die Überdeckungsziele aus Abbildung (a)

Abbildung 18: Beispiel zur ergänzenden Erreichbarkeitsprüfung

der Regel aus Definition 68 einen Widerspruch fest. Verfolgt man ausgehend von cg_9 die Implikationsbeziehungen im Abhängigkeitsgraph, so lassen sich mit cg_1 und cg_3 zwei CGs finden, die nicht gleichzeitig erfüllt sein können. Das CG cg_9 ist daher unerfüllbar.

Aus der Erkennung von weiteren unerreichbaren oder stets erreichten CGs können sich geringfügige Transformationen des CG-Abhängigkeitsgraphen ergeben. Wird beispielsweise für ein CG, welches im Graphen zu einem Disjunktoren führt, festgestellt, dass dieses unerreichbar ist, so kann die Kante von diesem CG zum Disjunktoren entfernt werden.

3.3.3 ABARBEITUNGSREIHENFOLGENBILDUNG

Wie einführend erläutert, dient die Erfassung der Abhängigkeiten zwischen CGs letzten Endes der Ableitung einer Abarbeitungsreihenfolge für die CGs. Diese Reihenfolge wird nach Durchführung von CGSeq verwendet, um die CGs nacheinander jeweils mit Testdatensuchen anzuvisieren.

Genau genommen werden nicht die CGs anvisiert, sondern sogenannte Suchziele (SGs), welche CGSeq bildet. Bei einem SG kann es sich sowohl um ein einzelnes CG als auch um eine Kombination von CGs handeln. Bei der Behandlung des Beispiels aus Abbildung 17 in Abschnitt 3.3.2.4 wurde bereits angedeutet, dass es unter Umständen von Vorteil sein kann, bei einer Testdatensuche mehrere CGs gleichzeitig anzuvisieren. Die Abarbeitungsreihenfolge bestimmt CGSeq dementsprechend für die gebildeten SGs und somit nicht direkt für die CGs. Die SG-Bildung und die Priorisierungsmetriken, welche die Reihenfolgenbildung bewirken, werden in den nächsten zwei Abschnitten vorgestellt.

3.3.3.1 *Ableitung von Suchzielen*

Ein SG ist formal betrachtet eine Menge von CGs:

$$sg \subseteq E \tag{69}$$

Für ein SG wird die folgende Schreibweise verwendet:

$$(a, ..., n) \Leftrightarrow sg = \{cg_a, ..., cg_n\} \tag{70}$$

Die statische Voranalyse CGSeq bildet die SGs in drei Schritten:

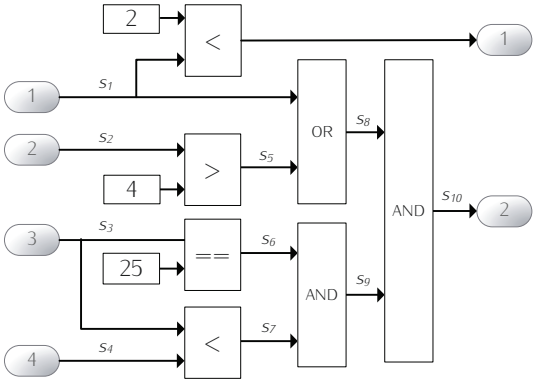
1. Reduzierung des Abhängigkeitsgraphen gemäß Testziel oder Nutzerwahl
2. Bildung singulärer SGs (enthalten je ein CG)
3. Bildung zusammengesetzter SGs (enthalten mehrere CGs)

Gilt die Testdatengenerierung der Automatisierung eines Strukturtests, so erfolgt üblicherweise eine Auswahl des oder der betrachteten Überdeckungskriteriums/-kriterien. Möglich ist auch, dass manuell eine Auswahl aus den verfügbaren CGs getroffen wird. Eine funktionale Eigenschaft, welche es im Rahmen eines Funktionstests – wie er in dieser Arbeit betrachtet wird (siehe Abschnitt 2.2.2) – zu erreichen oder zu widerlegen gilt, kann ebenfalls als ein einzelnes ausgewähltes CGs betrachtet werden. Unabhängig von der Testart gibt es eine Teilmenge der CGs, welche es zu erreichen gilt. Im Folgenden wird ein in dieser Teilmenge enthaltenes CG auch als selektiertes CG bezeichnet. Der CG-Abhängigkeitsgraph wird durch CGSeq entsprechend dieser Teilmenge reduziert. Hierzu werden zum einen nicht-selektierte CGs aus Gruppen äquivalenter CGs entfernt – vorausgesetzt, die Gruppe besitzt mindestens ein selektiertes CG. Zum anderen werden nicht-selektierte CGs und gegebenenfalls deren Knoten im Abhängigkeitsgraphen entfernt, falls von diesem CG ausgehend keinerlei Implikationsbeziehungen zu einem selektierten CG existieren, das heißt auch keine transitiven Verbindungen.

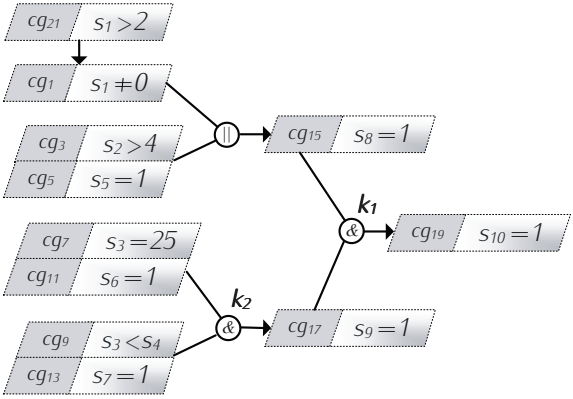
Nachdem ein CG-Abhängigkeitsgraph auf diese Weise zugeschnitten wurde, werden singuläre SGs gebildet. Hierzu wird nacheinander jedes selektierte CG cg herangezogen, das durch die Erreichbarkeitsprüfungen von SIA und CGSeq nicht als unerreichbar (UNSAT) oder stets erreicht (SAT) identifiziert wurde – das heißt st(cg)=OPEN. Aus jedem solchen CG wird ein singuläres SG gebildet. Gegebenenfalls bildet für jedes cg zusätzlich ein anderes, dieses CG repräsentierendes CG ein SG. Hierzu wird durch Ablaufen der Implikationsbeziehungen im Abhängigkeitsgraphen, welche direkt oder transitiv zu cg führen, eine Menge von CGs gebildet. In dieser Menge sind all die CGs enthalten, deren singuläre Erfüllung jeweils die Erfüllung von cg zur Folge hätte – und cg selbst. Aus dieser Menge wird ein CG ausgewählt, welches ein singuläres SG bildet und cg somit repräsentiert. Die Auswahl erfolgt auf Grundlage der Metriken, welche primär der nachfolgenden Sortierung aller gesammelten SGs dienen und daher erst im nächsten Abschnitt vorgestellt werden. Ein CG wird selbstverständlich nur dann in Form eines singulären SG der entstehenden SG-Menge hinzugefügt, wenn diese noch kein singuläres SG mit dem selben CG beinhaltet.

Wieso aber werden sowohl ein selektiertes CG als auch ein hierzu passendes repräsentierendes CG als SG herangezogen? Nun, handelt es sich bei dem selektierten CG beispielsweise um eines, welches durch eine Testdatensuche schwer zu erreichen ist, so besteht die Hoffnung, dass das repräsentierende CG im Vergleich hierzu einfacher zu erreichen ist. Dennoch wird das selektierte CG ebenfalls als singuläres SG aufgenommen, da auch das Gegenteil der Fall sein kann. Für das Beispiel in Abbildung 13 gilt beispielsweise: Das CG cg_1 ($s \geqslant 90$) weist eine hohe Kollateralüberdeckung auf und repräsentiert somit unter anderem cg_5 ($s \geqslant 50$). Der Wertebereich des Signals s lässt Werte von 0 bis 100 zu. Im Vergleich zu cg_5 ist cg_1 somit möglicherweise schwerer zu erreichen. Für den Fall, dass die Testdatensuche für cg_1 nicht erfolgreich abgeschlossen werden kann, ist es daher ratsam, nachfolgend auch cg_5 mit einer Testdatensuche anzuvisieren.

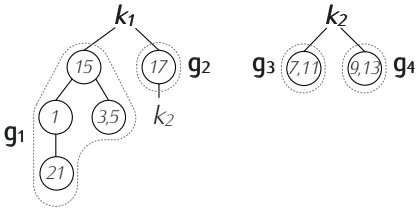
Ausgangspunkt der Bildung zusammengesetzter SGs sind gegebenenfalls in einem CG-Abhängigkeitsgraphen vorhandene Konjunktoren. Das Vorgehen hierbei sei anhand von Abbildung 19 erklärt. Dargestellt sind ein Beispiel-Modell und ein Auszug des aus diesem Modell hervorgehenden CG-Abhängigkeitsgraphen. Wie zu sehen ist, enthält dieser Graph zwei Konjunktoren. Diese sind mit den Bezeichnern k_1 und k_2 benannt. Für jeden Konjunktor wird, wie dargestellt, eine Baumstruktur aufgebaut, welche diejenigen Graphknoten enthält, die über direkte oder transitive Implikationsbeziehungen zu dem Konjunktor führen. Gelangt CGSeq bei dieser Rückverfolgung der Implikationsbeziehungen an einen weiteren Konjunktor, so endet die Rückverfolgung und in der Baumstruktur wird eine Referenz auf die Baumstruktur des anderen Konjunktors gesetzt. Abbildung 19c zeigt die nach diesem Schema aufgebauten Baumstrukturen



(a) Beispielmodell



(b) Auszug des Überdeckungsziel-Abhängigkeitsgraphen zu (a)



(c) Bildung zusammengesetzter Suchziele über Gruppierung von Überdeckungszielen gemäß der Abhängigkeiten aus (b)

Abbildung 19: Vorgehen zur Bildung von zusammengesetzten Suchzielen anhand eines Beispielmodells

für k_1 und k_2. Die in den Baumstrukturen enthaltenen CG-Gruppen werden anschließend bezüglich ihres Einflusses auf die jeweilige Konjunktion gruppiert. Dieser Schritt verhindert, dass zu viele zusammengesetzte SGs gebildet werden. Für das Beispiel ergeben sich die vier Gruppen g_1, g_2, g_3 und g_4. Zur Erfüllung der Konjunktion k_2 ist es notwendig, dass jeweils ein CG aus den Gruppen g_3 und g_4 erfüllt ist. Im Falle von k_1 werden zum einen die Gruppen g_1 und g_2 als Kombination gewählt und zum anderen die Kombination g_1, g_3 und g_4, welche durch Berücksichtigung der referenzierten Baumstruktur von k_2 entsteht. Um aus diesen Gruppen-Kombinationen abschließend zusammengesetzte SGs zu bilden, wird unter Verwendung der im nachfolgenden Abschnitt vorgestellten Metriken aus jeder Gruppe ein CG ausgewählt. Für das Beispiel ergeben sich die SGs $(7, 9)$, $(21, 7, 9)$ und $(21, 17)$.

In Abschnitt 2.4.2 wurde die Fitnessberechnung für ein CG im Zuge einer Testdatensuche behandelt. Bei Nutzung von CGSeq berechnet sich die Fitness für ein SG anstatt direkt für ein CG. Im Falle eines singulären SG entspricht die Fitness der des enthaltenen CG. Im Falle eines zusammengesetzten SG ist die Fitness das arithmetische Mittel der Fitness der enthaltenen CGs.

3.3.3.2 *Priorisierungsmetriken*

Die hier vorgestellten Metriken dienen unter anderem, wie im vorigen Abschnitt festgestellt, der Bildung von SGs durch Bestimmung geeigneter CGs. Der hauptsächliche Verwendungszweck dieser Metriken ist allerdings ein anderer. Die untereinander priorisierten Metriken bewirken nämlich eine Sortierung einer SG-Menge. Diese Sortierung bestimmt, in welcher Reihenfolge ein jedes SG anschließend mit einem eigenen Suchvorgang zum Zwecke einer Testdatensuche anvisiert wird. Im Folgenden werden die vier Priorisierungsmetriken vorgestellt. Abbildung 20 fasst diese Ausführungen zusammen.

Anzahl Boolescher Signale (M1). In einem CG-Ausdruck enthaltene Signale mit einem Booleschen Wertebereich sind für eine Testdatensuche problematisch (siehe Abschnitt 2.5). Im Sinne einer effizienten Testdatengenerierung ist es daher sinnvoll, SGs, deren CGs in Summe eine höhere Anzahl Boolescher Signale enthalten, an das Ende der Liste abzuarbeitender SGs zu setzen. Die Nutzung dieser Metrik setzt voraus, dass die SIA-Technik verwendet wird. Die Metrik hat unter den vier Metriken die höchste Priorität. Der Grund hierfür ist, dass eine Bewertung generierter Testdaten mit einer Fitnessfunktion, die im Extremfall einen Booleschen Wertebereich besitzt, bei Nutzung eines Suchverfahrens zur Testdatengenerierung sehr nachteilig für deren Effizienz ist.

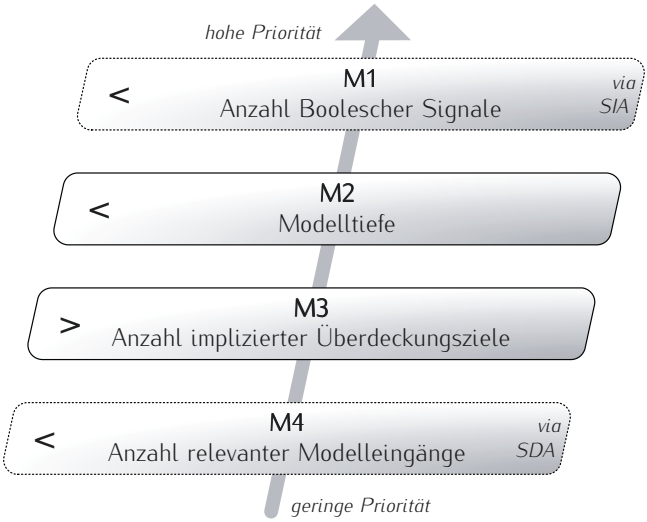


Abbildung 20: Übersicht über die zur Suchzielpriorisierung verwendeten Metriken

Anzahl implizierter CGs (M3). Hauptziel der CGSeq-Technik ist eine Steigerung der Kollateralüberdeckung bei einer Testdatensuche für ein SL/TL-Modell. Aus dem CG-Abhängigkeitsgraphen lässt sich für ein CG ermitteln, wie viele andere CGs mindestens erfüllt wären, wenn das betrachtete CG bei einer Modellausführung erfüllt wäre. Hierzu gilt es, die direkten und transitiven vorwärtsgerichteten Implikationsbeziehungen, ausgehend von dem betrachteten CG, zu traversieren. Hierbei ist jedes selektierte CG der besuchten CG-Gruppen in eine Menge implizierter CGs aufzunehmen. Im Falle eines zusammengesetzten SG ist dies für jedes enthaltene CG durchzuführen. Die dabei entstehenden Mengen der jeweils implizierten CGs werden anschließend vereinigt. Die Anzahl der CGs in dieser vereinigten Menge ist die gesuchte Anzahl. Unter den vier Metriken hat die Anzahl implizierter CGs nur die dritthöchste Priorität. Warum, wird im Zusammenhang mit der nachfolgend vorgestellten Metrik deutlich.

Modelltiefe (M2). Für jedes an einem CG-Ausdruck beteiligten Signal lässt sich eine minimale Entfernung zu einem der Modelleingänge bestimmen. Diese als Modelltiefe bezeichnete Entfernung wird durch die Anzahl der Signale zwischen betrachtetem Signal und einem Modelleingang bemessen. Wird neben CGSeq auch die SDA-Technik genutzt, so verwendet CGSeq hierzu den durch SDA erstellten Signalabhängigkeitsgraphen. Andernfalls wird die Modelltiefe direkt aus der Struktur des betrachteten SL/TL-Modells abgeleitet. Für die Modelltiefe eines

CG gilt: Enthält das CG mehrere Signale, so ist dessen Modelltiefe das arithmetische Mittel der Modelltiefen der Signale. Letzten Endes wird die Modelltiefe eines SG benötigt. Im Falle eines zusammengesetzten SGs ist dies ebenfalls das arithmetische Mittel der Modelltiefen der enthaltenen CGs, andernfalls die des einzigen enthaltenen CG. Die Metrik der Modelltiefe besitzt die zweithöchste Priorität. Dies hat folgenden Grund: Um eine möglichst hohe Kollateralüberdeckung bei einer Testdatensuche zu erreichen, sollte normalerweise die Metrik der Anzahl implizierter CGs eine höhere Priorität besitzen. Allerdings handelt es sich bei dieser Anzahl nur um einen Mindestwert, da die Analyse der CG-Abhängigkeiten unter Umständen nicht alle tatsächlich existierenden Abhängigkeiten erfasst. Gibt es beispielsweise einen unbekannten Block, für den sich keine semantischen Abhängigkeiten ableiten lassen, so wird hierdurch möglicherweise eine Kette von Implikationsbeziehungen unterbrochen, beziehungsweise ein Glied dieser Kette erst gar nicht erkannt. Betrachtet man demzufolge einen CG-Abhängigkeitsgraphen als eine Ansammlung von nicht zusammenhängenden Teilgraphen, so stellt die Modelltiefe der in den Teilgraphen enthaltenen CGs ein auf dieser übergeordneten Ebene betrachtet geeigneteres Maß für die zu erwartende Kollateralüberdeckung dar. Denn je näher ein CG sich bezüglich der Lage seiner Signale an den Modelleingängen befindet, desto mehr im Signalfluss folgende CGs sollten durch das Erreichen dieses CG mit erreicht werden.

Anzahl relevanter Modelleingänge (M4). Wird die SDA-Technik verwendet, so ist bekannt, von welchen Modelleingängen die Erfüllung eines CG abhängt. Die Erfüllung eines SG ist entsprechend von all denen Modelleingängen abhängig, von denen mindestens eines der enthaltenen CGs abhängt. Die Verwendung dieser Metrik als weiteres Priorisierungskriterium adressiert nicht die Kollateralüberdeckung, sondern die erwartungsgemäß effizientere Durchführung einer Testdatensuche bei geringerer Suchraumgröße (siehe Abschnitt 3.2.1).

Die aus dem CG-Abhängigkeitsgraphen abgeleiteten SGs werden unter Zuhilfenahme dieser Metriken sortiert. Hierbei gilt: Liefert eine Metrik für zwei oder mehrere SGs das gleiche Ergebnis, so entscheidet die nachfolgende, geringer priorisierte Metrik über die Sortierung der betreffenden SGs untereinander. Unterscheiden sich zwei oder mehrere SGs auch nach Anwendung aller vier Metriken nicht bezüglich der ermittelten Maße, so erfolgt eine zufällige Sortierung dieser SGs untereinander.

Durch Beachtung dieser Sortierung bei der Abarbeitung der SGs mittels Suchvorgängen ist eine insgesamt effizientere Testdatengenerierung zu erwarten. Diese Annahme fußt auch auf kleineren Experimenten, welche im Zuge der Entwicklung von CGSeq – insbesondere zur Bestimmung der Reihenfolge der Priorisierungsmetriken – durchgeführt wurden. Die in Kapitel 6 vorgestellten Fallstudien untersuchen in diesem Zusammenhang, welche Auswirkungen die Nutzung einer statischen Voranalyse

wie CGSeq auf eine automatisierte Testdatengenerierung hat. Dabei wird auch der zusätzliche Aufwand, den eine vorherige Anwendung von CGSeq mit sich bringt, berücksichtigt und untersucht.

3.3.4 VERWANDTE ARBEITEN

Das Ziel, die Kollateralüberdeckung einer suchbasierten Testdatengenerierung im Kontext eines Strukturtests zu steigern, wurde bereits in anderen Arbeiten adressiert. Die existierenden Ansätze unterscheiden sich allerdings grundlegend von der durch CGSeq verfolgten Idee.

Fraser und Arcuri streben in einer ihrer Arbeiten [40] die Erhöhung der Kollateralüberdeckung bei einem suchbasierten Strukturtest für Java-Code an. Hierbei generieren sie mittels eines genetischen Algorithmus ganze Testsuiten anstatt einzelner Testdaten. Ein durch die Suche erzeugter Lösungsvorschlag enthält also eine ganze Menge an Testdaten. Die Fitnessfunktion ist derart gestaltet, dass die Suche neben der Erfüllung eines bestimmten CG auch eine Maximierung des durch die erzeugten Testsuiten erzielten Überdeckungsgrads anstrebt. Haben zwei Testsuiten hierbei den gleichen Überdeckungsgrad, so erhält die Testsuite mit weniger Testdaten einen vergleichsweise besseren Fitnesswert. Experimente zeigen, dass dieser Ansatz im Vergleich mit einem klassischen suchbasierten Ansatz tatsächlich zu einer höheren Kollateralüberdeckung, sowie kleineren Testsuiten, führen kann. Harman et al. verfolgen einen hierzu ähnlichen Ansatz [62]. Die Suche generiert in ihrem Fall zwar keine Testsuiten, sondern einzelne Testdaten, jedoch beziehen sie das Kriterium der Kollateralüberdeckung ebenfalls mit in eine Suche ein. Dies geschieht im Rahmen einer sogenannten *Multi-Objective Optimization*, welche einer gleichzeitigen Anvisierung mehrerer Ziele dient. Experimente der Autoren zeigen, dass ihr Vorgehen zu geringeren Testsuitegrößen und hiermit verbunden, zu einer höheren Kollateralüberdeckung führt.

Ein weiterer Ansatz weist Gemeinsamkeiten mit CGSeq auf – allerdings beschränkt auf einen bestimmten Teilaspekt. McMinn und Holcombe beschäftigen sich in einer Arbeit [84] mit der suchbasierten Testdatengenerierung für C-Code. Der klassische suchbasierte Ansatz wird dabei um folgende Idee ergänzt: Für bestimmte, schwer erreichbare Zustände oder Suchziele wird auf statischem Wege aus dem Programmcode des Testobjekts eine Folge von Ereignissen (englisch: *event sequence*) abgeleitet. Hierbei gilt, dass das Eintreten einer Folge von Ereignissen bei Ausführung des Testobjekts das Erreichen des gewünschten Zustands oder Suchziels impliziert. Das Ziel, eine solche Ereignisfolge zu erreichen, wird entsprechend in die Fitnessfunktion aufgenommen und somit durch die Testdatensuche berücksichtigt. Dieses Vorgehen weist Ähnlichkeiten

zu CGSeq auf. Auch CGSeq versucht, für ein anvisiertes, unter Umständen schwer erreichbares CG zu ermitteln, ob es andere Zustände (in Form anderer CGs) gibt, deren Erfüllung die Erfüllung des anvisierten CG impliziert. Eine derartige Hilfestellung zum Erreichen schwer erreichbarer (siehe hierzu auch Abschnitt 2.5 zum Thema „Blindheit der Fitnessfunktion“) oder auch Boolescher Zustände ist allerdings nur ein Teilaspekt von CGSeq. Primär geht es CGSeq um eine Steigerung der Kollateralüberdeckung und Effizienz einer Testdatensuche. Das gewählte Mittel besteht bei CGSeq, anders als bei den hier genannten Arbeiten, grundsätzlich auch nicht in der zusätzlichen Aufnahme weiterer Ziele in eine Fitnessfunktion, sondern in der Gestaltung einer möglichst optimalen CG-Abarbeitungsreihenfolge.

Eine Arbeit, die sich mit einer solchen Reihenfolgenbildung auseinandersetzt, stammt von Li et al. [81]. Die Autoren betrachten die strukturorientierte Testdatengenerierung für C-Code. In diesem Kontext ist es das Ziel ihres Ansatzes, nur eine minimale Menge an Pfaden im Kontrollflussgraphen betrachten zu müssen und dennoch den größtmöglichen Überdeckungsgrad zu erzielen. Hierzu werden Pfade aus dem Kontrollflussgraphen in geordneter Folge ausgewählt und anschließend für diese Pfade jeweils Testdaten abgeleitet – Li et al. verwenden hierzu eine binäre Suche. Die Pfadauswahl und -ordnung basiert auf einer Gewichtung der Knoten des Kontrollflussgraphen. Die Gewichtung eines Knoten gibt an, wie viele andere Knoten durch ihn dominiert werden. Ein Knoten a dominiert einen anderen Knoten b, wenn die Ausführung der Codezeile, welche a referenziert, die Ausführung von der zu b gehörenden Codezeile zur Folge hat. Dieses Vorgehen ist dem von CGSeq grundlegend sehr ähnlich. Bei CGSeq ist jedoch nicht etwa ein Kontrollflussgraph, sondern ein CG-Abhängigkeitsgraph Grundlage für die Reihenfolgenbildung. Zudem existieren zum Zwecke dieser Reihenfolgenbildung neben der Metrik der Anzahl implizierter CGs, welche mit der Knotengewichtung bei Li et al. vergleichbar ist, weitere Metriken.

Eingebettet in den hier behandelten wissenschaftlichen Kontext zeichnet sich CGSeq durch folgende Eigenschaften und Beiträge aus:

1. Ermittlung logischer und semantischer (das heißt aus der Modellsemantik abgeleiteter) Abhängigkeiten zwischen CGs
2. CG-Erreichbarkeitsprüfung auf Basis von CG-Abhängigkeiten, in Ergänzung zur SIA-Technik
3. Bildung einer Abarbeitungsreihenfolge der CGs über Metriken, die vergleichsweise „einfach“ erreichbare CGs und solche mit einer potentiell hohen Kollateralüberdeckung bevorzugen

3.4 KOMBINATION DER TECHNIKEN

Bei der Vorstellung der drei statischen Voranalysen SIA, SDA und CGSeq in diesem Kapitel wurde bereits herausgestellt, dass die Ergebnisse der Analysen nicht nur der Vorbereitung der Testdatengenerierung dienen, sondern diese auch zum Teil von den Analysen untereinander genutzt werden. In diesem Abschnitt soll das Zusammenwirken zwischen den statischen Voranalysen und deren Einbettung in das in dieser Arbeit behandelte Testdatengenerierungsverfahren daher zusammenfassend betrachtet werden.

Abbildung 21 gibt hierzu einen Überblick. Die statischen Voranalysen (Bereich in der Mitte) betten sich zwischen der Initialisierung der Testdatengenerierung (Bereich oben) und der Testdatengenerierung selbst (Bereich unten) ein. Die Vorgänge und entstehenden Artefakte innerhalb

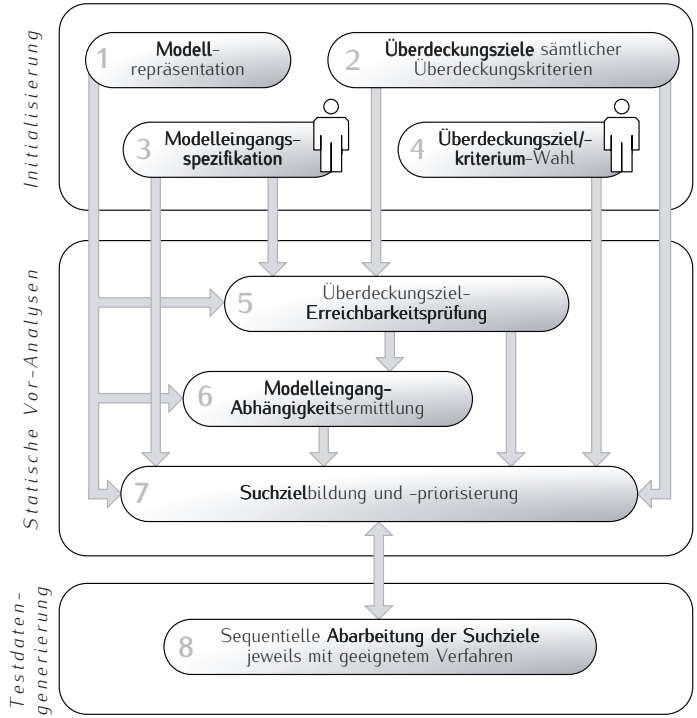


Abbildung 21: Zusammenspiel der statischen Voranalysen untereinander und Einbettung in den Anwendungskontext

dieser Bereiche sind in der Darstellung entsprechend der Reihenfolge ihrer Durchführung oder ihres Auftretens nummeriert.

Im ersten Schritt wird das betrachtete SL/TL-Modell in eine Repräsentation überführt, mit der die nachfolgenden Vorgänge und Analysen arbeiten. Hierbei werden, wie zu Anfang dieses Kapitels erläutert, grundlegende Modelltransformationen durchgeführt. Anschließend wird das Modell auf alle betrachteten Überdeckungskriterien hin untersucht und hierbei eine Gesamtmenge von CGs abgeleitet (siehe Abschnitt 2.2.1). Der Nutzer unterstützt die Testdatengenerierung durch Angabe einer Modelleingangsspezifikation (siehe Abschnitt 2.4.2). Des Weiteren wählt der Nutzer, sofern es sich um eine Testdatengenerierung im Sinne eines Strukturtests handelt, eines oder mehrere Überdeckungskriterien aus. Handelt es sich um einen anderen Anwendungsfall, wie beispielsweise einen Funktionstest, so kann es sich hierbei auch um eine Auswahl einzelner Zieldefinitionen handeln, welche CG-ähnlich formuliert sind. Unabhängig davon resultiert diese Nutzerwahl in einer Menge selektierter CGs.

Bevor nun die Testdatengenerierung stattfindet, kommen die statischen Voranalysen ins Spiel. Die Überdeckungsziel-Erreichbarkeitsprüfung (SIA) ermöglicht eine Ermittlung unerreichbarer und stets erreichter CGs in der Menge sämtlicher CGs. Diese müssen bei der Testdatengenerierung nicht weiter betrachtet werden. SIA verwendet für diese Ermittlung die Modellrepräsentation und die Modelleingangsspezifikation.

Die anschließend durchgeführte Modelleingang-Abhängigkeitsermittlung (SDA) prüft für ein jedes erfüllbares CG der Menge aller CGs, welche Modelleingänge bei einer Testdatengenerierung mit passenden Daten versorgt werden müssen, damit eine Erfüllung des jeweiligen CG erzielt werden kann. Irrelevante Modelleingänge können bei einer späteren Testdatengenerierung zum Teil ausgeklammert werden. SDA nutzt nicht nur die Modellrepräsentation, sondern über diese indirekt auch Ergebnisse von SIA– denn SIA führt unter Umständen Modelltransformationen durch, welche im Falle von SDA die Genauigkeit der Ergebnisse erhöhen können.

Die Suchzielbildung und -priorisierung (CGSeq) analysiert Abhängigkeiten zwischen sämtlichen CGs, bildet hierauf basierend unter Verwendung der Menge selektierter CGs sogenannte Suchziele (SGs) und bestimmt für diese eine sinnvolle Abarbeitungsreihenfolge, die bei der darauffolgenden Testdatengenerierung Berücksichtigung findet und dort eine Effizienzsteigerung bewirken soll. Zur Abhängigkeitsanalyse nutzt CGSeq unter anderem auch die Ergebnisse von SIA. Zur Bildung der Abarbeitungsreihenfolge werden sowohl Ergebnisse von SIA als auch von SDA genutzt.

Die statischen Voranalysen sind wohlgemerkt so gestaltet, dass sie auch unabhängig voneinander arbeiten können. Jedoch kann ein solch partieller Einsatz zu Abstrichen bei Umfang und Genauigkeit der Analyse-Ergebnisse führen. Dieser Aspekt ist insbesondere für die Untersuchungen der Fallstudien dieser Arbeit von Relevanz.

Wird die Testdatengenerierung mittels einer, wie in Abschnitt 2.4.2 beschriebenen, globalen Suche durchgeführt, so schließt sich diese in der Darstellung in Schritt acht an. In der Darstellung ist jedoch zu lesen, dass jedes SG mit einem geeigneten Testdatengenerierungsverfahren bearbeitet wird. Diese Formulierung greift bereits auf die Inhalte des nächsten Kapitels vor. Denn das hybride Testdatengenerierungsverfahren dieser Arbeit verwendet anstatt einer globalen Suche mitunter auch SMT-Solving zur Erzeugung von Testdaten.

Nach jedem Suchvorgang für ein SG erfolgt übrigens eine Filterung und Neuordnung der Abarbeitungsreihenfolge. Filterung bedeutet, dass gegebenenfalls SGs entfernt werden. Ein SG ist hiervon in zwei Fällen betroffen: einerseits, wenn alle enthaltenen CGs erreicht wurden – andererseits, wenn es kein unerreichtes, selektiertes CG enthält und in der Menge der von diesem SG implizierten, selektierten CGs kein unerreichtes CG mehr vorkommt. Anlass für eine Neuordnung ist ebenfalls, dass während des vorherigen Suchvorgangs möglicherweise eines oder mehrere CGs erreicht wurden. Dies ändert die Berechnungsgrundlage der initialen Abarbeitungsreihenfolge, da diese zur Sortierung die Metrik der Anzahl der durch ein SG implizierten und gleichzeitig bislang unerreichten CGs nutzt. Es erfolgt allerdings lediglich eine erneute Berechnung der Metriken und eine entsprechende Sortierung der SGs. Die Abhängigkeitsanalyse von CGSeq wird nicht wiederholt.

4 GENERIERUNGSVERFAHREN

Die Hybridisierung der Testdatensuche für SL/TL-Modelle durch eine Angliederung statischer Voranalysen stellt den Hauptbeitrag dieser Arbeit dar. Darüber hinaus wurde allerdings auch untersucht, inwiefern eine Erweiterung des in der zugrundeliegenden Arbeit von Windisch [132] genutzten globalen Suchverfahrens oder der Einsatz alternativer Generierungstechniken zu einer effizienten Testdatengenerierung für SL/TL-Modelle beitragen können.

Zwei Ansätze wurden in diesem Zusammenhang verfolgt. Ersterer adressiert die in Abschnitt 2.4.1 behandelte Unterscheidung lokaler und globaler Suchverfahren. Der folgende Abschnitt stellt ein lokales Suchverfahren zur Testdatengenerierung für SL/TL-Modelle vor. Der zweite Ansatz, welcher Thema des darauffolgenden Abschnitts ist, befasst sich mit einer Testdatengenerierung mittels SMT-Solving. In letzterem Fall wurde eine Hybridisierung der alternativen Generierungstechnik mit der globalen Testdatensuche realisiert.

4.1 GENERIERUNG MITTELS LOKALER SUCHE

Das im Rahmen dieser Arbeit geschaffene lokale Suchverfahren basiert auf der AVM. Entsprechend seines spezifischen Anwendungsbereichs, der Testdatengenerierung für SL/TL-Modelle, wird es als *alternierende Signalparameteroptimierung* (ASPO) bezeichnet. Die Zielstellung von ASPO wird im folgenden Teilabschnitt erläutert. Anschließend wird die Funktionsweise von ASPO erklärt und der Beitrag, den dieses Verfahren darstellt, im Kontext verwandter Arbeiten diskutiert.

Vorweg ein Hinweis: ASPO ist im Rahmen einer Masterarbeit entstanden [63]. Die Inhalte von Abschnitt 4.1 stammen daher zum Teil aus dieser Arbeit oder sind an die dortigen Darstellungen und Beschreibungen angelehnt.

4.1.1 ZIEL

Es wurde bereits einführend behandelt, dass bei Suchverfahren grundlegend zwischen lokalen Suchen (LS) und globalen Suchen (GS) zu unterscheiden ist. Hierbei wurde hervorgehoben, dass eine LS tendenziell effizienter im Umgang mit Problemstellungen ist, bei denen es um das Finden eines lokalen Optimums im betrachteten Suchraum geht. Eine GS hingegen besitzt ihre Stärken, wenn es um das Auffinden des globalen Optimums geht – oder allgemeiner ausgedrückt, wenn es unerwünscht ist, dass die Suche in beliebigen lokalen Optima endet. Die vorliegende Arbeit geht, wie in Abschnitt 2.4.2 behandelt, vom Einsatz einer GS zur Testdatengenerierung für SL/TL-Modelle aus. Bei dieser GS handelt es sich um einen längenvariablen genetischen Algorithmus (LGA).

Die Frage, ob sich für das betrachtete Anwendungsgebiet eine LS oder eine GS besser eignet, lässt sich ohne eine Gegenüberstellung zweier entsprechender Ansätze nicht beantworten. Der an dieser Stelle vorgestellte Teilbeitrag dieser Arbeit ist durch diese Fragestellung motiviert. Erst nach Beantwortung dieser Fragestellung lässt sich über Möglichkeiten zur Hybridisierung beider Varianten einer suchbasierten Testdatengenerierung nachdenken. Um zu diesem Punkt zu gelangen, wird eine LS benötigt, die für SL/TL-Modelle anwendbar ist und die imstande ist, Testdaten in ähnlicher Form wie der LGA zu generieren. Die Forschungsfrage lautet daher an dieser Stelle: Lässt sich eine LS gestalten, die im Vergleich zum LGA effizienter und mit zumindest gleichbleibendem Erfolg plausible Testdaten für ein SL/TL-Modell generiert?

Bei der hierzu entwickelten ASPO handelt es sich um eine LS, welche Testdaten, wie auch der LGA, in Form von Optimierungssequenzen betrachtet. Dies bedeutet, dass ein Signal, welches für jeden Modelleingang erzeugt und im Rahmen einer Suche variiert wird, als eine Sequenz von Signalsegmenten aufgefasst wird – wobei jedes Segment drei Parameter besitzt. Trotz einer solchen Abstrahierung von der tatsächlichen Komplexität eines Signals stellt die Verwendung von Optimierungssequenzen eine Herausforderung für die Anwendung von gängigen lokalen Suchverfahren dar. Der Grund hierfür ist, dass lokale Suchverfahren die Nachbarschaft eines während der Suche betrachteten Lösungsvorschlags nicht nur sehr systematisch, sondern auch nahezu erschöpfend untersuchen. Mit steigender Anzahl zu optimierender Parameter wächst die Komplexität einer solchen Nachbarschaftsuntersuchung. Eine LS, welche sich diesem Problem im besonderen Maße angenommen hat, ist die AVM [76]. ASPO ist eine Variante der AVM, welche Signale anstatt gängiger Programmvariablentypen (Zahlen oder Strings) optimiert.

ASPO führt, entsprechend der in dieser Arbeit betrachteten spezifischen Testdatenrepräsentation (Optimierungssequenzen), eine Nachbarschaftsuntersuchung für Signale durch. Im folgenden Abschnitt 4.1.2 wird daher

die Nachbarschaft eines Signals definiert. Der Ablauf einer LS mit ASPO wird im daran anschließenden Abschnitt 4.1.3 vorgestellt.

4.1.2 NACHBARSCHAFTSDEFINITION FÜR SIGNALE

Der Nachbarschaftsbegriff im Rahmen einer LS wurde bereits in Abschnitt 2.4.1 eingeführt. Im Allgemeinen wird ein Lösungsvorschlag im Suchraum als benachbart bezeichnet, wenn er vom betrachteten Lösungsvorschlag aus gesehen durch Anwendung eines sogenannten Bewegungsoperators in einem Schritt erreicht werden kann [93]. Die Größe und Art einer Nachbarschaft ist stark von der verwendeten Repräsentation eines Lösungsvorschlags und den verwendeten Bewegungsoperatoren abhängig. Im Folgenden werden Überlegungen bezüglich einer geeigneten Nachbarschaftsdefinition für Signale, inklusive geeigneter Bewegungsoperatoren, angestellt.

4.1.2.1 *Parametermodifikationen*

Die Modifikationen eines Signals oder mehrerer Signale im Rahmen einer LS sollten eine möglichst systematische und ganzheitliche Untersuchung des Suchraums ermöglichen. Eine bloße Verschiebung eines Signals auf seiner Werte- oder Zeitachse wird diesem Anspruch beispielsweise nicht gerecht. Eine gezielte Änderung einzelner Signalwerte ist ebenfalls nicht ratsam. Dieses Vorgehen könnte zu Signalen führen, die im Konflikt mit der geforderten Plausibilität der Testdaten stehen. Unabhängig davon, ob Testdaten mit einer GS oder einer LS generiert werden, ist es aus praktischen Gründen nämlich wünschenswert, dass die automatisch erzeugten Testdaten realistische Daten darstellen. Es besteht daher die Anforderung, dass eine LS, welche zur Signalgenerierung genutzt wird, ebenfalls die in Abschnitt 2.4.2 behandelte segmentbasierte Signalrepräsentation nutzt.

Um die Nachbarschaft eines Signals mitsamt Bewegungsoperatoren zu definieren, gilt es also, die Parameter einer Optimierungssequenz zu adressieren. Jedes Segment einer Optimierungssequenz besteht aus drei verschiedenartigen Parametern: einem Amplitudenwert, einer Segmentbreite und einem Transitionstypen. Modifikationen dieser Parameter ermöglichen eine Generierung fast beliebiger Signale [133] – eine einschränkende Wirkung wird einzig durch Angaben in der Modelleingangsspezifikation und durch eine mögliche Festsetzung der Anzahl von Signalsegmenten erzielt. Führt eine LS eine Nachbarschaftsuntersuchung mittels Modifikation der Parameter einer Optimierungssequenz durch, so ist die eingangs gestellte Forderung, eine möglichst ganzheitliche Untersuchung des Suchraums zu ermöglichen, daher erfüllt.

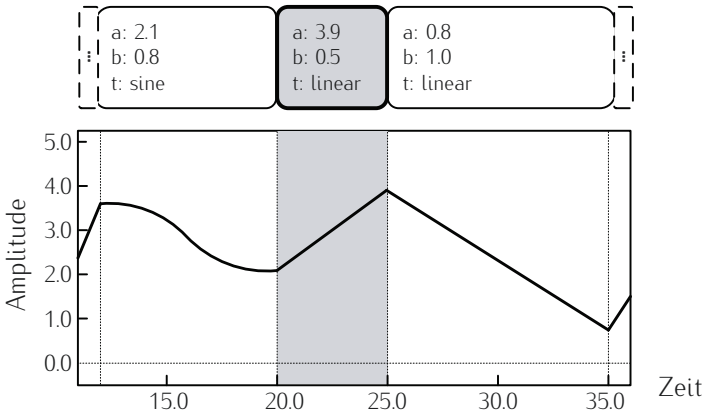
Die Nachbarschaft einer Optimierungssequenz ergibt sich durch die Betrachtung der Nachbarschaften ihrer einzelnen Segmente. Die Nachbarschaft eines Segments wiederum ergibt sich durch Betrachtung der Nachbarschaften der drei Parameter des Segments. Die Nachbarschaft eines Amplitudenwerts, einer Segmentbreite und eines Transitionstyps wird im Folgenden durch Vorstellung der jeweiligen Bewegungsoperatoren definiert. Der Auszug einer Optimierungssequenz, dargestellt in Abbildung 22a, dient hierbei der Veranschaulichung. Zu sehen sind drei aufeinanderfolgende Segmente einer Optimierungssequenz, wobei das mittlere, grau eingefärbte Segment Gegenstand der Nachbarschaftsuntersuchung sein soll.

AMPLITUDE

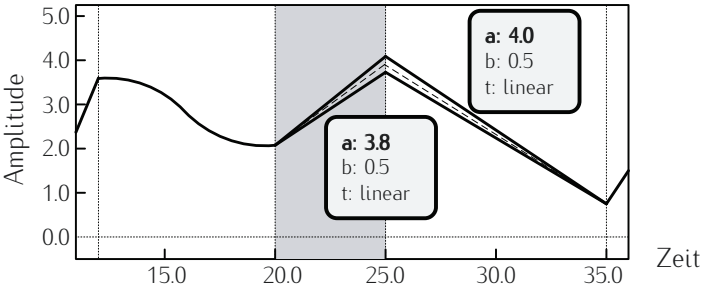
Der Amplitudenwert eines Segments beschreibt den Wert, welches das Signal, in seinem zeitlichen Verlauf betrachtet, am Ende des Segments einnimmt. Der Amplitudenwert ist eine reelle Zahl. Als Bewegungsoperatoren kommen daher Inkrement- und Dekrementoperatoren in Frage, um die direkte Nachbarschaft eines Amplitudenwerts abzustecken. Die Schrittweite der Inkrementierung und Dekrementierung kann hierbei der Modelleingangsspezifikation entnommen werden. In dieser ist für jeden Modelleingang angegeben, in welchen unteren und oberen Amplitudengrenzen sich ein generiertes Signal zu bewegen hat – und in diesem Zusammenhang auch, welche Quantisierung die Signalwerte aufzuweisen haben (siehe Definition 18). Die Quantisierung entspricht der gesuchten Schrittweite.

Das Beispiel-Segment in Abbildung 22a besitzt den Amplitudenwert $3,9$. Als Schrittweite wird der Wert $0,1$ angenommen. Abbildung 22b veranschaulicht die Anwendung des Inkrement- und Dekrementoperators. Es resultieren zwei benachbarte Segmente, in denen der Amplituden-Parameter des Segments die Werte $3,8$ und $4,0$ besitzt. Falls der Ausgangsamplitudenwert (in diesem Fall $3,9$) bereits die obere oder untere spezifizierte Grenze der Amplitudenwerte darstellt, kann dementsprechend nur einer der beiden Bewegungsoperatoren angewendet werden. Die Nachbarschaft bezüglich des Amplitudenparameters besteht dann aus nur einem benachbarten Segment.

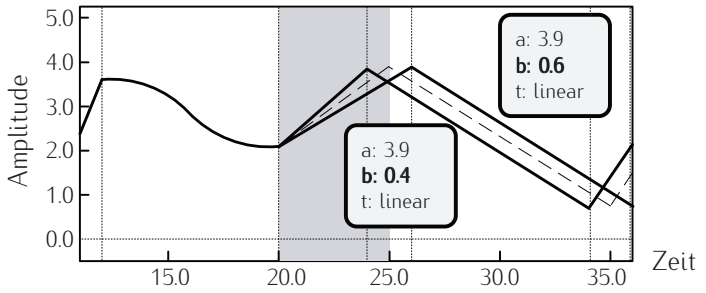
Im Falle der meisten Transitionstypen bewirkt eine Amplituden-Inkrementierung oder -dekremtierung eine Änderung der Steigung des Übergangs zwischen Segmentbeginn und -ende. In der Regel wirken sich die Bewegungsoperationen dabei auch auf das darauffolgende Segment aus – zumindest auf das durch die gesamte Optimierungssequenz beschriebene und als Simulationssequenz bezeichnete Signal bezogen. Der Amplitudenwert eines Segments beschreibt nämlich nicht nur den Signalwert am Segmentende, sondern gleichzeitig auch den Signalwert



(a) Teil einer Optimierungssequenz mit Fokus auf das mittlere Segment



(b) Nachbarschaft des Amplitudenparameters



(c) Nachbarschaft des Breitenparameters

Abbildung 22: Betrachtung der Nachbarschaft eines Segments (Parameter Amplitude und Breite) einer Optimierungssequenz anhand eines Beispiels

zu Beginn des darauffolgenden Segments. Daher wirkt sich eine Modifikation des Amplitudenwerts eines Segments zwangsläufig auch auf das Folgesegment aus. Ausgenommen hiervon sind Segmente mit dem Transitionstyp Impuls, da bei diesen der Anfangsamplitudenwert des folgenden Segments dem Anfangsamplitudenwert des Impulssegments entspricht. Der Amplitudenwert eines Impulssegments beschreibt hierbei nicht den Amplitudenwert am Segmentende, sondern den temporären Impulsausschlag.

SEGMENTBREITE

Bei einer Segmentbreite handelt es sich ebenfalls um einen reellen Wert. Daher bilden auch hier Inkrement- und Dekrementoperatoren die Bewegungsoperatoren. Eine Vergrößerung oder Verkleinerung einer Segmentbreite bewirkt eine Dehnung oder Stauchung des entsprechenden Signalabschnitts auf der Zeitachse. Ein Beispiel hierzu findet sich in Abbildung 22c. Als spezifizierte Abtastrate (siehe Definition 19) für die zu erzeugenden Signale wird im Falle des Beispiels ein Wert von $0,1$ Sekunden angenommen. Unter Berücksichtigung dieser Angabe ergeben sich im Rahmen der Nachbarschaftsbetrachtung zwei benachbarte Segmente: eines, welches innerhalb der aus der Optimierungssequenz generierten Simulationssequenz einen Zeitbereich einnimmt, der um $0,1$ Sekunden kürzer ist – und eines, welches um $0,1$ Sekunden länger ist.

Als Schrittweite, welche durch die Bewegungsoperatoren zu der Segmentbreite addiert oder subtrahiert wird, lässt sich allerdings nicht direkt die Abtastrate verwenden. Die Schrittweite muss zuerst bestimmt werden. Hierzu werden folgende Hintergrundkenntnisse benötigt: Bei einer Segmentbreite handelt es sich, wie in Abschnitt 2.4.2 erläutert, nicht etwa um eine Zeitangabe, sondern um eine relative Breitenangabe. Bei Überführung einer Optimierungssequenz in eine Simulationssequenz werden die aufsummierten Breiten aller Segmente einer Optimierungssequenz nämlich der spezifizierten Signallänge (siehe Definition 19) gleichgesetzt und die zeitlichen Längen eines jeden Segments hierzu in Relation berechnet. Die zeitliche Länge wird hierbei durch die Anzahl an Datenpunkten angegeben. Wird die relative Breite eines Segments g mit $\mathrm{sg}_{\mathrm{width}}(g)$ bezeichnet und handelt es sich bei $\mathrm{spec}_{\mathrm{steps}}$ um die Gesamtanzahl an Signal-Datenpunkten (siehe Definition 20), so bestimmt sich die Anzahl der Datenpunkte, welches das Segment in der Simulationssequenz einnimmt, folgendermaßen:

$$\mathrm{sg}_{\mathrm{steps}}(g) = \frac{\mathrm{sg}_{\mathrm{width}}(g)}{\mathrm{sg}_{\mathrm{width},\Sigma}} \cdot \mathrm{spec}_{\mathrm{steps}} \tag{71}$$

Hierbei bezeichnet $sg_{width,\Sigma}$ die Summe der Breiten aller Segmente der Optimierungssequenz.

$$sg_{width,\Sigma} = \sum_{i=1}^{sg_\Sigma} sg_{width}(sg(i)) \tag{72}$$

Bei sg_Σ handelt es sich um die Anzahl der Segmente der Optimierungssequenz und $sg(i)$ bezeichnet das i-te Segment.

Bezogen auf das Beispiel-Segment aus Abbildung 22a, welches die relative Breitenangabe $0,5$ besitzt, bedeutet dies: Geht man neben der angenommenen Abtastrate von $0,1$ Sekunden von einer spezifizierten Signallänge von 30 Sekunden aus, so ergibt sich in einem ersten Schritt eine Datenpunktanzahl für die Simulationssequenz von 300 ($spec_{steps}$). Wird nun zudem eine aufsummierte Gesamtbreite aller Segmente von 10 angenommen ($sg_{width,\Sigma}$), so ergeben sich für das betrachtete Segment 15 Datenpunkte ($sg_{steps}(g)$).

Der Genauigkeit halber sei angemerkt, dass der für ein jedes Segment berechnete Wert $sg_{steps}(g)$ möglicherweise nicht ganzzahlig ist und daher anschließend noch aufgerundet wird. Hierdurch kann für das gesamte Signal allerdings ein Überhang an Datenpunkten entstehen. Dieser Überhang wird im Nachhinein unter Berücksichtigung einer Mindestlänge für Segmente bei entsprechend gewählten Segmenten wieder abgeschnitten. Ein Segment, bei welchem im Zuge einer Nachbarschaftsuntersuchung eine Segmentbreitenänderung erfolgt, bleibt hiervon allerdings unberührt.

Um nun unter Zuhilfenahme der effektiven Breite eines Segments (Definition 71) einen kleinstmöglichen Schritt zur Änderung der Segmentbreite in der Simulationssequenz zu bewirken, ist es notwendig, die Auswirkungen der Modifikation einer Segmentbreite auf die Skalierung der Breiten aller Segmente zu berücksichtigen. Gesucht sind die Schrittweiten $ms_{incr}(g)$ und $ms_{decr}(g)$, welche zur relativen Segmentbreite addiert oder subtrahiert werden, um die einem Signalsegment zugeordnete Anzahl der Datenpunkte um genau einen Datenpunkt zu erhöhen oder zu verringern. Diese lassen sich folgendermaßen berechnen:

$$\begin{aligned} sg_{steps}(g) + 1 &= \frac{sg_{width}(g) + ms_{incr}(g)}{sg_{width,\Sigma} + ms_{incr}(g)} \cdot spec_{steps} \\ &\Downarrow \\ ms_{incr}(g) &= \frac{sg_{width,\Sigma}}{spec_{steps} - sg_{steps}(g) - 1} \end{aligned} \tag{73}$$

und

$$\begin{aligned} sg_{steps}(g) - 1 &= \frac{sg_{width}(g) - ms_{decr}(g)}{sg_{width,\Sigma} - ms_{decr}(g)} \cdot spec_{steps} \\ &\Downarrow \\ ms_{decr}(g) &= \frac{sg_{width,\Sigma}}{spec_{steps} - sg_{steps}(g) + 1} \end{aligned} \tag{74}$$

Bei den oberen Gleichungen in Definition 73 und Definition 74 handelt es sich um Umformulierungen der effektiven Segmentbreite aus Definition 71 für ein Segment, welches in der Simulationssequenz einen Datenpunkt mehr, beziehungsweise einen Datenpunkt weniger, einnimmt. Die Gleichungen wurden in beiden Fällen schlichtweg umgestellt und vereinfacht, so dass einzig $ms_{incr}(g)$, beziehungsweise $ms_{decr}(g)$, auf der linken Seite stehen. Somit sind die Schrittweiten zur Inkrementierung und Dekrementierung einer Segmentbreite im Sinne einer Nachbarschaftsbildung gefunden.

Im Falle des Beispiels ergeben sich für das dort betrachtete Segment, unter Berücksichtigung der zuvor gemachten Annahmen, die Schrittweiten $ms_{incr}(g) = \frac{5}{142}$ für den Inkrementoperator und $ms_{decr}(g) = \frac{5}{143}$ für den Dekrementoperator.

TRANSITIONSTYP

Der Transitionstyp eines Segments ist im Vergleich zu Amplitudenwert und Segmentbreite keine reelle Zahl, sondern ein Element aus einer Menge möglicher Transitionstypen. Diese Menge ist gegebenenfalls über eine entsprechende Auswahl in der Modelleingangsspezifikation eingeschränkt. Dies bedeutet, dass für Optimierungssequenzen, die einem bestimmten Modelleingang gelten, unter Umständen nicht alle der verfügbaren Transitionstypen (siehe Abschnitt 2.4.2) zugelassen sind.

Für die Elemente dieser Menge kann ohne Weiteres keine Ordnung angenommen werden. Das heißt, es kann nicht davon ausgegangen werden, dass auf einen linearen Transitionstypen beispielsweise ein sinusförmiger Transitionstyp folgt oder Ähnliches. Die Zweckmäßigkeit einer solchen Ordnung im Rahmen einer der Testdatengenerierung dienenden LS ist zudem fraglich. Zwar existieren zweifelsfrei Ähnlichkeiten zwischen der linearen und sinusförmigen Transition sowie der Spline-Transition, da sie alle zu einem kontinuierlichen, positiven oder negativen Anstieg innerhalb eines Signalsegments führen, jedoch ist es für eine automatische Suche nicht zwingend von Vorteil, zum Beispiel die Sinus-Transition als eine gesteigerte Form einer linearen Transition aufzufassen. Dies würde zu einer Priorisierung bei der Transitionstyp-Wahl führen und somit den Freiheitsgrad der Suche unnötig einschränken. Die Transitionstypen

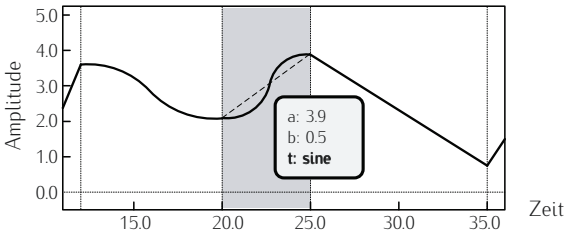
werden daher alle als miteinander benachbart betrachtet. Ein einzelner Transitionstyp besitzt also keine ausgewählten Nachbarn in der Menge aller für eine Optimierungssequenz erlaubten Transitionstypen. Die Anzahl der Nachbarn richtet sich einzig und allein nach den Einschränkungen der verfügbaren Transitionstypen in der Modelleingangsspezifikation.

Der Bewegungsoperator für den Transitionstyp generiert dementsprechend keinen, einen oder mehrere modifizierte Signalsegmente. Dies ist in Abbildung 23 für die zuvor bereits als Beispiel verwendete Optimierungssequenz dargestellt. In diesem Fall sind alle Transitionstypen erlaubt. Im Sinne der Nachbarschaftsbildung wird der ursprüngliche lineare Transitionstyp, welcher in den einzelnen Darstellungen gestrichelt dargestellt ist, jeweils durch einen der vier anderen Transitionstypen (Sinus, Spline, Impuls und Sprung) ausgetauscht. Im Gegensatz zu den anderen Transitionstypen entspricht der Endamplitudenwert eines Segments mit Impuls-Transitionstyp dessen Anfangsamplitudenwert, wie zuvor bereits herausgestellt wurde. Der Anfangsamplitudenwert eines Impulssegments wird dabei auch vom darauffolgenden Segment als Anfangsamplitudenwert verwendet. Daher sorgt eine Veränderung des Transitionstyps hin zu dem Impuls-Typ auch für eine Veränderung innerhalb der Simulationssequenz im darauffolgenden Signalsegment. Dieser Umstand kann in Abbildung 23c nachvollzogen werden.

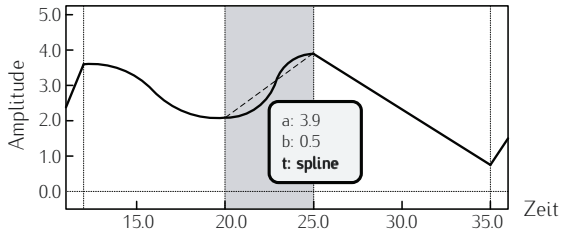
4.1.2.2 *Die Nachbarschaftsgröße als Komplexitätsfaktor*

Aus den Überlegungen zu der Nachbarschaft eines Signalsegments lassen sich Folgerungen bezüglich der Komplexität einer Nachbarschaftsuntersuchung im Rahmen einer LS, die der Suche nach Signalen dient, ableiten.

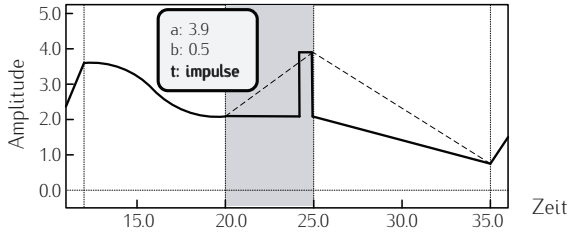
Abbildung 24 stellt in diesem Zusammenhang die Nachbarschaft eines Segmentes einer Optimierungssequenz dar. Die benachbarten Segmente ergeben sich hierbei durch Anwendung der Segmentparameterspezifischen Bewegungsoperatoren für jeweils einen einzelnen Segmentparameter. Für den Amplitudenparameter ergeben sich ein oder zwei Nachbarn, je nachdem ob die Anwendung eines der Bewegungsoperatoren zu einem Amplitudenwert außerhalb des spezifizierten Wertebereichs führen würde oder nicht. Bezüglich der Segmentbreite ergeben sich zwei weitere Nachbarn. Für den Parameter Transitionstyp leiten sich bis zu vier benachbarte Segmente ab. Die genaue Anzahl hängt dabei davon ab, inwieweit die Menge der erlaubten Transitionstypen in der Modelleingangsspezifikation eingeschränkt ist. Insgesamt besitzt ein Signalsegment bis zu acht benachbarte Segmente – mindestens jedoch drei. Sofern in der Modelleingangsspezifikation kein fixer Startamplitudenwert für einen Modelleingang angegeben ist (siehe Definition 18), so wird dieser durch



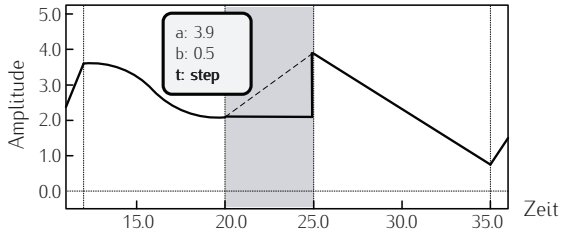
(a) Nachbar mit dem Transitionstyp Sinus



(b) Nachbar mit dem Transitionstyp Spline



(c) Nachbar mit dem Transitionstyp Impuls



(d) Nachbar mit dem Transitionstyp Sprung

Abbildung 23: Betrachtung der Nachbarschaft eines Segments (Parameter Transitionstyp) einer Optimierungssequenz anhand eines Beispiels

den Amplitudenparameter des ersten Segments einer Optimierungssequenz bestimmt. Das erste Segment besitzt in diesem Fall keinen Breiten- und Transitionstyp-Parameter. Daher leiten sich für das erste Segment in diesem Fall nur ein oder zwei benachbarte Segmente ab.

Geht man von einer festen Anzahl an Signalsegmenten aus, bezeichnet mit sg_Σ, so bewegt sich die Anzahl nb an Nachbarn einer gesamten Optimierungssequenz, welche durch minimale Veränderungen eines jeden Parameters eines jeden Segments erreicht werden können, zwischen folgenden Werten:

$$3 \cdot sg_\Sigma - 2 \leqslant nb \leqslant 8 \cdot sg_\Sigma \tag{75}$$

Entsprechend der Anzahl der betrachteten Modelleingänge steigt die Anzahl an Nachbarn bei einer LS nach Eingangssignalen für ein SL/TL-Modell auf ein Vielfaches dieses Werts.

Hierzu sei ein kleines Beispiel betrachtet: Besitzt ein Modell fünf Eingänge und wird festgelegt, dass die Optimierungssequenzen für jeden Modelleingang aus jeweils zehn Segmenten bestehen, so leitet sich für ein generiertes Testdatum eine stattliche Anzahl von bis zu 400 benachbarten Testdaten ab. Zieht man als lokales Suchverfahren den in Abschnitt 2.4.1 einführend vorgestellten Bergsteigeralgorithmus heran, so bedeutet dies, dass in jeder Iteration der Suche bis zu 400 Testdaten generiert werden und diese zu evaluieren sind. Diese Herangehensweise würde zu einer höchst ineffizienten Suche führen. Würde man zusätzlich bei der Nachbarschaftsbetrachtung auch die gleichzeitige Modifikation mehrerer Parameter zulassen, steigt die Anzahl an generierten Testdaten sogar exponentiell.

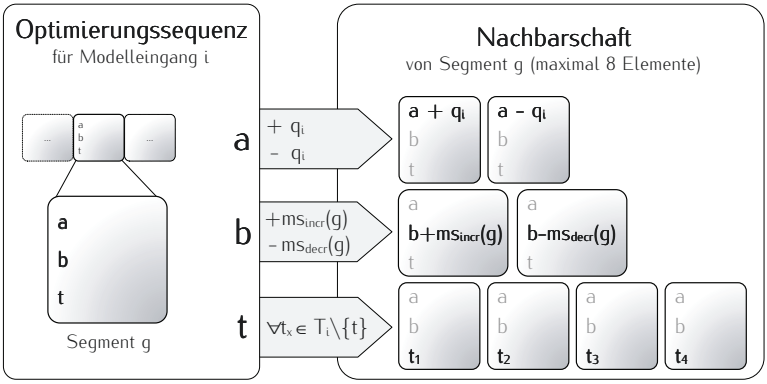


Abbildung 24: Betrachtung der Nachbarschaftsgröße eines Segments einer Optimierungssequenz

Die Betrachtung der Komplexität einer Nachbarschaftsuntersuchung soll an dieser Stelle verdeutlichen, dass die Durchführung einer LS im betrachteten Kontext, also bei der Generierung von Signalen mittels Optimierungssequenzen, nicht unproblematisch ist. Hier bedarf eine LS einer anderen Herangehensweise, als sie lokale Suchverfahren wie der Bergsteigeralgorithmus in seiner klassischen Form bieten. Das entwickelte Suchverfahren ASPO setzt daher auf ein Vorgehen, welches, wie im folgenden Abschnitt ersichtlich, der AVM ähnelt.

4.1.3 ALTERNIERENDE SIGNALPARAMETEROPTIMIERUNG

Das lokale Suchverfahren AVM zeichnet sich im Kern dadurch aus, dass es für jeden veränderlichen Parameter abwechselnd eine eigene LS durchführt (siehe auch Abschnitt 2.4.1). Der gesamte Vorgang startet mit einem initialen, zufällig gewählten Lösungsvorschlag. Sobald die LS bei ihren fortwährenden Nachbarschaftsuntersuchungen bezüglich eines betrachteten Parameters zu keinem, bezüglich seiner Fitness besseren Lösungsvorschlag mehr führt, wird zum nächsten Parameter übergegangen. Auch für diesen erfolgt eine entsprechende LS. AVM beendet seine Suche, wenn das Ziel der Suche erreicht wurde oder für alle Parameter in Folge eine jeweilige Nachbarschaftsuntersuchung zu keinem besseren Lösungsvorschlag führt. Die Vorgehensweise von AVM bewirkt, dass sich die Größe der während einer LS aufzubauenden Nachbarschaft stets auf die Größe der Nachbarschaft eines einzelnen Parameters beschränkt.

Dieses Vorgehen macht sich ASPO zunutze. Die Segmentparameter der für alle relevanten Modelleingänge betrachteten Optimierungssequenzen werden nacheinander behandelt. Je nach Art des Parameters reduziert sich die Zahl der bei einer Nachbarschaftsuntersuchung generierten Testdaten auf maximal zwei im Falle eines Amplitudenparameters, zwei im Falle eines Segmentbreitenparameters und maximal vier im Falle eines Transitionstyp-Parameters. Die Funktionsweise von ASPO im in dieser Arbeit betrachteten Anwendungskontext ist in Abbildung 25 dargestellt. Sie wird im Folgenden erläutert.

Um Testdaten für ein SL/TL-Modell zu generieren, übernimmt ASPO, wie in der Abbildung zu sehen, schlichtweg die Rolle des LGA. Die einzelnen SGs werden ebenfalls sequentiell mittels Suchvorgängen abgearbeitet. ASPO beginnt mit einem initialen Testdatum, welches wie die Population initialer Testdaten im Falle des LGA ebenfalls zufällig erzeugt wird. Die Repräsentation eines Testdatums ist ebenfalls identisch, das heißt dieses enthält je relevantem Modelleingang eine Optimierungssequenz. Im Unterschied zur Verwendung des LGA ist die initial, unter

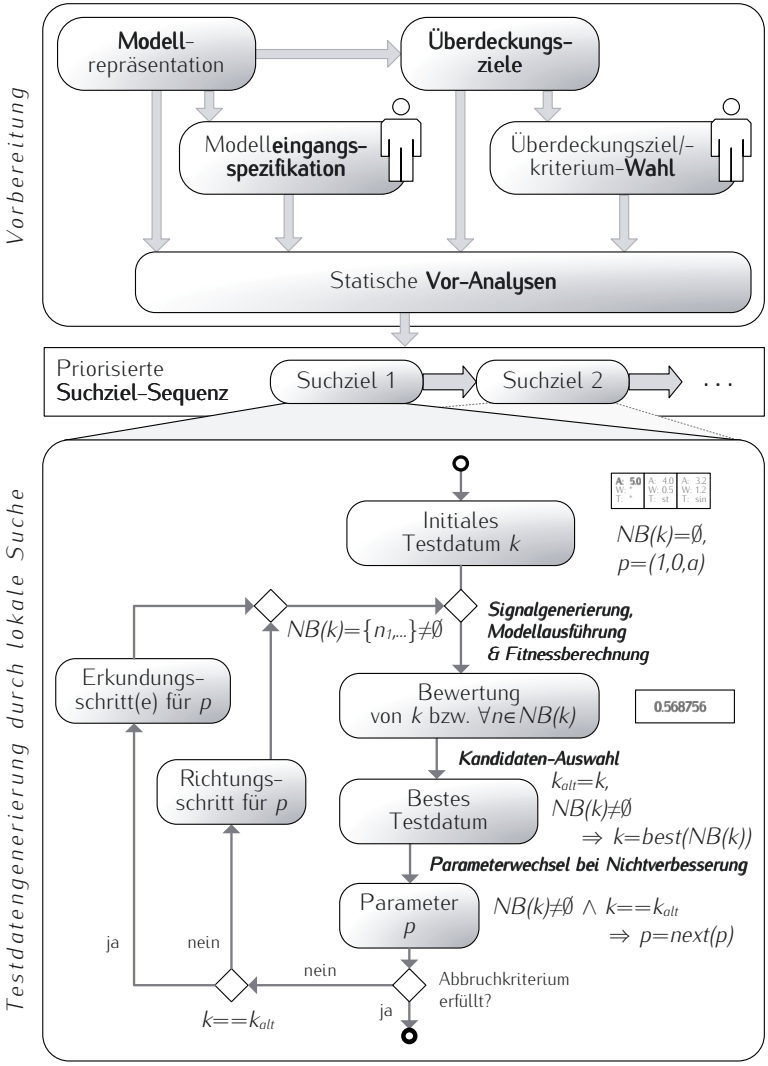


Abbildung 25: Ablauf der strukturorientierten Testdatengenerierung für SL/TL-Modelle mithilfe der alternierenden Signalparameteroptimierung

Berücksichtigung der Modelleingangsspezifikation je Optimierungssequenz zufällig gewählte Anzahl an Segmenten allerdings fix. Das heißt, während der Suche werden keine Segmente entfernt oder hinzugefügt. Um einen höheren Freiheitsgrad bei einer Suche zu erreichen, ist es vorstellbar, ASPO an dieser Stelle zukünftig noch zu erweitern.

Zurück zum Ablauf von ASPO: Gemäß einer Abarbeitungsreihenfolge der Optimierungssequenz-Parameter, welche nachfolgend noch detailliert behandelt wird, betrachtet ASPO zuerst den Parameter a (Amplitude) des 0-ten (also ersten) Segments der Optimierungssequenz für den ersten relevanten Modelleingang als zu untersuchenden Parameter p. Ein Parameter sei hierbei durch einen Tupel aus Nummer des Modelleingangs, Nummer des Signalsegments und Parametertyp bezeichnet. Der zuerst betrachtete Parameter ist demnach $p=(1,0,a)$.

Die Menge der Nachbarn des aktuell betrachteten Testdatums, bezeichnet durch $NB(k)$, ist initial leer. Wie für metaheuristische Suchverfahren üblich, erfolgt eine Bewertung der generierten Testdaten mittels einer Fitnessfunktion. Diese unterscheidet sich hierbei nicht von der, welche auch bei Verwendung des LGA zum Einsatz kommt. Zu Beginn bewertet ASPO nur das initiale Testdatum k, in folgenden Suchiterationen stattdessen die benachbarten Testdaten aus $NB(k)$. Wurde hierbei ein Testdatum mit besserer Fitness entdeckt, so übernimmt dieses die Rolle des aktuell betrachteten Testdatums. Bildlich betrachtet verschiebt die LS in diesem Fall ihre aktuelle Position im Suchraum. Kam es nicht zu einer Verbesserung, so rückt ASPO unter Berücksichtigung der Parameterabfolge zum nächsten Optimierungssequenz-Parameter vor.

Erfüllt das aktuell beste Testdatum k das SG oder wurden alle Parameter aller Optimierungssequenzen in Folge als Parameter p betrachtet, ohne dass in der jeweiligen Nachbarschaft ein Testdatum mit verbesserter Fitness registriert werden konnte, so endet die LS für das betrachtete SG. In ersterem Fall wird das Testdatum, welches das SG erfüllt, in die resultierende Testsuite aufgenommen. Um den Suchalgorithmus in seiner Laufzeit zu beschränken, existiert zudem eine maximale Anzahl von Fitnessfunktionsberechnungen, beziehungsweise generierter und bewerteter Testdaten. Die LS für ein SG terminiert daher auch, sobald diese Grenze erreicht ist.

Wird die LS fortgeführt, so wird die Nachbarschaft des aktuellen Testdatums k bezüglich des aktuell zur Modifikation herangezogenen Parameters p gebildet. Hierzu existieren zwei unterschiedliche Vorgehensweisen, von denen jeweils stets genau eine zur Anwendung kommt. Ist aufgrund einer Nicht-Verbesserung der bislang besten Fitness ein Parameterwechsel erfolgt, so wird die Nachbarschaft zunächst über sogenannte Erkundungsschritte gebildet. Kam es bezüglich des aktuellen Parameters p zu einer Verbesserung der Fitness und somit zu einer Änderung des aktuellen Testdatums k, so werden sogenannte Richtungsschritte unternommen,

um die Nachbarschaft zu formen. Eine ausführliche Unterscheidung dieser Schrittarten erfolgt im nächsten Teilabschnitt. Unabhängig davon, welche dieser Varianten in einer Suchiteration Anwendung findet, werden durch die Parametermodifikationen neue Testdaten gebildet. Diese formen die Menge NB(k). Da an dieser Stelle, wie in der Abbildung zu sehen ist, eine neue Suchiteration beschritten wird, verfährt ASPO im Weiteren wie in den vorigen Absätzen beschrieben.

4.1.3.1 *Erkundungs- und Richtungssuche*

Die in Abschnitt 4.1.2 vorgestellte Nachbarschaftsbetrachtung basiert auf einer Anwendung von Bewegungsoperatoren, welche per Definition minimale Veränderungen eines Optimierungssequenz-Parameters vornehmen. Derlei Veränderungsschritte werden als Erkundungsschritte bezeichnet. Richtungsschritte hingegen verändern Parameter in nur eine Richtung, beispielsweise durch ausschließliche Anwendung des Inkrementoperators. Zudem verwendet ASPO bei Richtungsschritten gegebenenfalls eine Schrittweite, die über das Maß einer minimalen Veränderung hinausgeht.

ERKUNDUNGSSUCHE

Die Anwendung von Erkundungsschritten zur Nachbarschaftsbildung wird auch als Erkundungssuche bezeichnet. Ein einzelner Erkundungsschritt wird durch genau einen Bewegungsoperator ausgeführt. Geht es beispielsweise um einen Amplitudenwert, so kann dies ein Inkrement- oder Dekrementoperator sein. Alle für einen Parameter verfügbaren Erkundungsschritte zusammen definieren dessen gesamte, direkte Nachbarschaft. Es handelt sich um direkte Nachbarn, da beispielsweise bei einer Inkrementierung eines Amplitudenwerts eine minimale Schrittweite verwendet wird (siehe Abschnitt 4.1.2.1).

Aufgabe einer Erkundungssuche ist es, die Richtung von Parametermodifikationen zu ermitteln, welche zu einer Annäherung an die Erfüllung des betrachteten SG führt. Erhält man durch Dekrementierung eines Amplitudenparameters beispielsweise ein Testdatum mit besserer Fitness, so gilt die Richtung einer Verringerung des Amplitudenwerts als erfolgversprechend. Eine solche Erkundungssuche, die in einem Testdatum mit besserer Fitness resultiert, ist Grundlage einer anschließend durchgeführten Richtungssuche. Eine Durchführung mehrerer, direkt aufeinanderfolgender Erkundungssuchen für denselben Parameter erfolgt nicht. Wird ein Parameter allerdings nach Betrachtung aller anderen Parameter zum wiederholten Male betrachtet, so erfolgt auch dann zuerst eine Erkundungssuche für diesen Parameter.

RICHTUNGSSUCHE

Führt ein Erkundungsschritt in der anschließenden Bewertung zu einem Testdatum mit verbessertem Fitnesswert, so wird daraufhin ein Richtungsschritt in die Richtung unternommen, die für die Verbesserung verantwortlich war. Sofern ein Richtungsschritt wiederum zu einem besseren Testdatum führt, so wird ein erneuter Richtungsschritt unternommen. Dieses Vorgehen endet, sobald keine Verbesserung des Fitnesswerts auftritt. Diese Herangehensweise gilt allerdings nur für die Amplituden- und Segmentbreitenparameter. Für einen Transitionstyp-Parameter lassen sich nach einer Verbesserung durch einen Erkundungsschritt keine weiteren Nachbarn finden. Bei der Erkundungssuche wurden nämlich bereits alle Variationen des Transitionstyp-Parameters untersucht. Daher geht ASPO nach einer Erkundungssuche für einen Transitionstyp-Parameter zu dem in der Abfolge nächsten Parameter über – unabhängig davon, ob die Erkundungssuche zu einem besseren Testdatum führt oder nicht. Diese Feinheit ist aus Darstellungsgründen nicht in Abbildung 25 enthalten.

Grundsätzlich wird für Richtungsschritte ebenfalls eine minimale Schrittweite genutzt. Im Falle des Amplitudenparameters wendet ASPO jedoch sogenannte Richtungssprünge an. Dieser Idee liegt folgende Problematik zugrunde: Bei wiederholter Anwendung minimaler Richtungsschritte kann es passieren, dass eine Annäherung an das SG trotz stetiger Verbesserungen des Fitnesswertes nur sehr langsam geschieht. Korel schlägt in diesem Zusammenhang vor, die Schrittweite mit zunehmender Anzahl erfolgreicher Richtungsschritte zu erhöhen [76]. Dieses Vorgehen hat allerdings den Nachteil, dass die Richtungsschritte immer größer werden, je näher die Suche ihrem Ziel kommt. Dies birgt die Gefahr, einen zur Erfüllung des Ziels gesuchten Parameterwert zu überspringen.

Die Richtungssprünge von ASPO nutzen daher stattdessen Schrittweiten, welche auf Basis der bei der vorherigen Erkundungssuche erzielten Fitnessverbesserung und der dabei verwendeten minimalen Schrittweite prognostiziert werden. Die Schrittweite s_{jump} eines Richtungssprungs berechnet sich hierbei folgendermaßen:

$$s_{jump} = \frac{s_{step}}{f_{improve}} \cdot f_{best} \tag{76}$$

Hierbei bezeichnet s_{step} die bei dem zuvor mit Erfolg durchgeführten Erkundungsschritt verwendete Schrittweite. Die bei diesem Erkundungsschritt erzielte Verbesserung des Fitnesswerts, also die Differenz zwischen vorherigem und dem neuen, aktuell besten Fitnesswert f_{best}, sei mit $f_{improve}$ benannt. Die Berechnung geht davon aus, dass die Fitnesswerte positiv sind und es stets das Ziel ist, den Fitnesswert 0 zu erreichen.

Die tatsächlich verwendete Schrittweite ist allerdings zusätzlich durch die spezifizierten Amplitudengrenzen beschränkt. Hierdurch wird verhin-

dert, dass Richtungssprünge zu Amplitudenwerten führen, die außerhalb des in der Modelleingangsspezifikation angegebenen Wertebereichs liegen. Darüber hinaus ist zu berücksichtigen, dass die Schrittweite eines Richtungssprungs lediglich eine Abschätzung darstellt. Um die Gefahr eines Überspringens des gesuchten Testdatums im Suchraum einzudämmen, führt ASPO nach jedem Richtungssprung eine Erkundungssuche durch. Je nach Ausgang dieser kann anschließend eine Richtungssuche in sowohl dieselbe Richtung wie zuvor als auch in die entgegengesetzte Richtung erfolgen. Somit ermöglicht ASPO im Falle zu großer Richtungssprünge ein Zurückgehen. Dieses Vorgehen sowie die Anwendung von Richtungssprüngen im Falle von Amplitudenparametern insgesamt ist in der Übersicht in Abbildung 25 nicht dargestellt.

4.1.3.2 *Parameterpriorisierung*

ASPO arbeitet die Segmentparameter der Optimierungssequenzen eines jeden relevanten Modelleingangs in einer bestimmten Abfolge ab. Die Überlegungen, welche Grundlage der Reihenfolgenbildung sind, werden in diesem Teilabschnitt vorgestellt und erklärt. Die durch ASPO erstellte Parameterabfolge ist in Abbildung 26 dargestellt. Dort ist zu erkennen, dass die Parameter in erster Linie nach der Art des Parameters geordnet sind. Zuerst werden alle Amplitudenparameter betrachtet, anschließend die Segmentbreiten eines jeden Segments und erst dann die Transitionstypen der Segmente. Die Parameter eines jeden Typs werden nachfolgend nach Zugehörigkeit zu einem Modelleingang und der Position des dazugehörigen Signalsegments innerhalb seiner Optimierungssequenz geordnet. Bei Verwendung der statischen Voranalyse SDA werden hierbei selbstverständlich nur die Modelleingänge berücksichtigt, welche für das bei der LS betrachtete SG relevant sind. Gelangt ASPO bei seiner LS am letzten Parameter der Abfolge an und erfolgt auch nach dessen Bearbeitung keine Beendigung der Suche, so wird als nächstes erneut der erste Parameter der Abfolge betrachtet.

Das markanteste Merkmal der Parameterabfolge ist die Priorisierung nach Parameterart. Dieser Priorisierung liegt die Annahme zugrunde, dass der Einfluss einer Modifikation eines Parameters auf die Annäherung an die Erfüllung eines SG in vielen Fällen je nach Art des Parameters unterschiedlich stark ausfällt. Werden Parameter, deren Veränderung mit höherer Wahrscheinlichkeit zu einem Testdatum mit verbesserter Fitness führen, durch ASPO bevorzugt behandelt, so kann der Aufwand und die Laufzeit einer Suche geringer ausfallen.

Mit Blick auf die Gestaltung der CG-Ausdrücke, wie sie Gegenstand beispielsweise einer strukturorientierten Testdatengenerierung sind (siehe Abschnitt 2.2.1), so fällt auf, dass es in der Regel um das Erreichen bestimmter Amplitudenwerte oder das Unter-/Überschreiten bestimmter

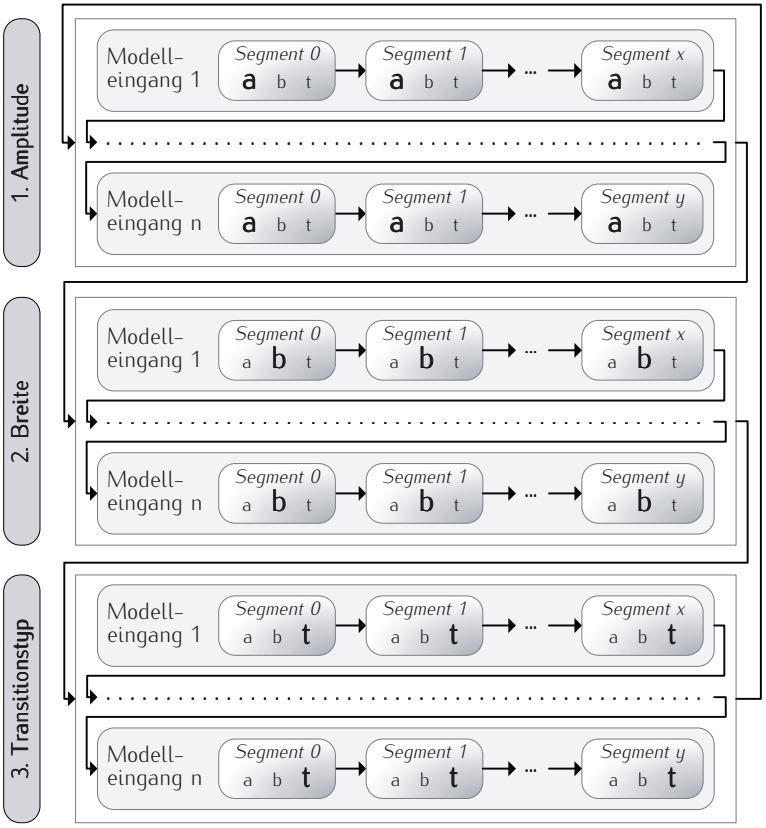


Abbildung 26: Abarbeitungsreihenfolge der einzelnen Signalparameter bei der alternierenden Signalparameteroptimierung

Amplitudengrenzen für Signale eines SL/TL-Modells geht. Ein jeder Amplitudenwert eines Signals innerhalb eines Modells ergibt sich im Allgemeinen aus Amplitudenwerten an den Eingängen des Modells. Daher ist die Vermutung naheliegend, dass insbesondere die Modifikation der Amplitudenparameter der Optimierungssequenzen zu einem erfolgreichen Fortschreiten einer LS beitragen können. Diese werden daher in der Parameterabfolge am höchsten priorisiert.

Wie in Abschnitt 4.1.2.1 herausgestellt, bewirken Modifikationen der Segmentbreitenparameter und Transitionstyp-Parameter indirekt Änderungen der Amplitudenwerte in einer aus einer Optimierungssequenz erzeugten Simulationssequenz. Insbesondere bei der Anvisierung von Modellzuständen, welche nur bei Stimulation des Modells mit bestimmten Werten zu unterschiedlichen Zeitschritten erreicht werden können, kann eine Modifikation von Breite oder Transitionstyp bestimmter Segmente notwendig sein. Die Modifikation einer Segmentbreite beeinflusst hierbei vor allem den Grad einer Steigung innerhalb eines Abschnitts der entstehenden Simulationssequenz. Der Transitionstyp ist hingegen meist für die Art der Steigung, beispielsweise durch einen linearen oder kurvenförmigen Verlauf, verantwortlich. ASPO weist Veränderungen von Steigungsgraden über Modifikationen von Segmentbreiten eine höhere Priorität als der Modifikation von Transitionstypen zu. Kleinere Versuche während der Entwicklung von ASPO haben bestätigt, dass eine solche Priorisierung entsprechend der drei Parameterarten zu einer effizienteren Durchführung einer LS führt [63].

Mit diesen Ausführungen ist die Vorstellung des lokalen Suchverfahrens ASPO, welches auf eine Testdatengenerierung für SL/TL-Modelle ausgerichtet ist, abgeschlossen. Ein Vergleich der LS von ASPO mit der GS des LGA erfolgt im Rahmen der in Kapitel 6 vorgestellten Fallstudien. Die statischen Voranalysen werden der LS hierbei, wie zuvor in Abbildung 25 dargestellt, ebenfalls vorangestellt. Eine darüber hinausgehende Hybridisierung der LS mit der GS oder einem anderen Generierungsverfahren ist nicht Teil dieser Arbeit. Sie wird aber bei der Bewertung von ASPO in Abschnitt 6.3 und im Ausblick dieser Arbeit in Abschnitt 7.2 thematisiert.

4.1.4 VERWANDTE ARBEITEN

Im betrachteten Anwendungsgebiet, der Testdatengenerierung für SL- oder SL/TL-Modelle, hat sich bislang nur eine recht überschaubare Anzahl von Arbeiten mit einer Anwendung eines suchbasierten Ansatzes auseinandergesetzt. Diese Arbeiten wurden bereits in Abschnitt 2.3.2 aufgeführt und diskutiert. Unter diesen Arbeiten gibt es zwei, welche sich mit der Nutzung einer LS beschäftigt haben. Beide betrachten zum

Zwecke einer Testdatengenerierung für SL-Modelle das Suchverfahren der *simulierten Abkühlung* (englisch: Simulated Annealing). Dieses ursprünglich der Simulation von Abkühlungsprozessen verschiedener Materialien dienende Verfahren [88] wurde einst von Kirkpatrick et al. [73] zu einem Optimierungsverfahren umfunktioniert. In diesem Zusammenhang adressiert das Verfahren die Schwäche des Bergsteigeralgorithmus, lokale Optima bei einer Suche nicht überwinden zu können. Hierzu lässt das Verfahren zum Teil Bewegungen im Suchraum zu Lösungsvorschlägen mit einer schlechteren Fitness zu. Diese Bewegungen sind an Wahrscheinlichkeiten gebunden. Da das Verfahren hierdurch eine globale Untersuchung des Suchraums anstrebt, wird es trotz seiner lokalen Betrachtung des Suchraums des Öfteren auch als GS klassifiziert.

Ghani et al. vergleichen die simulierte Abkühlung in Anwendung zur Testdatengenerierung für SL-Modelle mit einer GS, einem genetischen Algorithmus [48]. Bezüglich Erfolg und Effizienz der Suchvorgänge unterscheiden sich die beiden Ansätze meist nicht signifikant – und wenn, dann führt die GS zu besseren Ergebnissen. Allerdings ist die Aussagekraft dieser Untersuchung fraglich, da es sich bei den betrachteten SL-Modellen lediglich um kleine „Spielzeugmodelle“ handelt. Zudem legen die Autoren keinen Wert auf die Plausibilität der generierten Testdaten, das heißt eine Erzeugung von Signalen steht nicht im Vordergrund ihrer Bemühungen. Das Problem der Komplexität einer Nachbarschaftsuntersuchung, welches sich vor allem durch Berücksichtigung dieses Aspekts über einen segmentbasierten Signalgenerierungsansatz ergibt, tritt demnach im Falle ihrer Arbeit ebenfalls nicht auf. Diese Unterschiede und Einschränkungen gelten auch für die Arbeit von Zhan und Clark [137]. Die beiden Autoren, welche ebenfalls bei der Arbeit von Ghani et al. mitwirken, wenden auch das Verfahren der simulierten Abkühlung zur Testdatengenerierung für SL-Modelle an. Während ihr Vorgehen sehr ähnlich zu dem in der Arbeit von Ghani et al. ist, geht es ihnen allerdings nicht um eine Testdatengenerierung für Strukturtests, sondern für Mutationstests. Des Weiteren vergleichen sie ihre LS lediglich mit dem Zufallstest.

In diesem Kontext betrachtet lassen sich die Beiträge des ASPO-Verfahrens folgendermaßen zusammenfassen:

1. Gestaltung einer LS unter Berücksichtigung der Plausibilität generierter Testdaten
2. Aufstellung einer Nachbarschaftsdefinition bei Nutzung einer segmentbasierten Signalrepräsentation
3. Anwendungsfall-spezifische Erweiterung der AVM durch eine Parameterpriorisierung und durch Richtungssprünge bei der Modifikation von Amplitudenparametern

4.2 GENERIERUNG MITTELS SMT-SOLVING

Die Kombination der drei statischen Voranalysen mit einer automatischen Testdatensuche stellt eine erste Hybridbildung zum Zwecke einer effizienten Testdatengenerierung für SL/TL-Modelle dar. Während das im vorigen Abschnitt behandelte lokale Suchverfahren ASPO im Rahmen dieser Arbeit lediglich eine Alternative zum globalen Suchverfahren LGA ist, erfolgt die in diesem Abschnitt betrachtete Anwendung des SMT-Solving zur Testdatengenerierung in hybridisierter Form mit einer Suche. So gesehen handelt es sich bei dem hier behandelten Beitrag um eine zweite Hybridbildung. Folgt man der Unterscheidung der Hybridisierungsmöglichkeiten zu Beginn von Kapitel 3, so ordnet sich dieser Beitrag in die Kategorie der Ansätze ein, welche während oder anstelle einer Testdatensuche unterstützend wirken.

4.2.1 ZIEL

Eine Testdatengenerierung mittels ausschließlich statischer Techniken wurde einführend in Abschnitt 2.3.1 behandelt. Hierbei kam vor allem das Verfahren der symbolischen Ausführung zur Sprache. Der Begriff symbolische Ausführung beschreibt hierbei, dass das Testobjekt zum Zwecke einer Testdatengenerierung nicht tatsächlich ausgeführt wird, sondern eine statische Analyse der Testobjekt-Struktur und -Semantik stattfindet, welche in ihrem Vorgehen einer konkreten Ausführung ähnelt. Handelt es sich beispielsweise um Programmcode, so wird wie bei einer konkreten Ausführung ebenfalls der Kontrollfluss, wie er sich in einem KFG darstellen lässt, von den Programmeingängen ausgehend abgegangen. Im Unterschied zur konkreten Ausführung der enthaltenen Bedingungen und Anweisungen werden allerdings Pfadbedingungen gesammelt, deren Erfüllung ein Erreichen bestimmter Zustände, Bedingungen oder Anweisungen innerhalb des Testobjekts zur Folge hätte. Aus der Lösung solcher Pfadbedingungen, beispielsweise durch SMT-Solving, lassen sich Testdaten ableiten.

Eine solche symbolische Ausführung stößt allerdings an Grenzen, denn aus realen Testobjekten leiten sich mitunter Ausdrücke in Pfadbedingungen ab, deren Lösung durch Techniken wie dem SMT-Solving nicht bewerkstelligt werden kann. Ist eine Lösbarkeit allerdings gegeben, so liefern das SMT-Solving in der Regel sehr zügig Ergebnisse. Aus diesen Ergebnissen lassen sich direkt die gesuchten Testdaten bilden. Eine Testdatengenerierung mittels SMT-Solving besitzt daher das Potential, sofern deren Anwendbarkeit gegeben ist, deutlich effizienter als eine Testdatensuche zu sein. Die Zahl der Arbeiten im Bereich des SMT-Solving und die

dabei gemachten Fortschritte lassen vermuten, dass die Leistungsgrenzen einer symbolischen Ausführung mittels SMT-Solving heutzutage deutlich höher liegen. Dies betrifft nicht nur die Fähigkeit moderner SMT-Solver, beispielsweise mathematische Ausdrücke verarbeiten zu können, sondern auch deren Skalierbarkeit. Da SL/TL-Modelle aus der industriellen Praxis häufig sehr komplex und mächtig sind, ist dieser Aspekt für eine Anwendung des SMT-Solving zur Testdatengenerierung besonders relevant.

Dem in diesem Teil der Arbeit behandelten Ansatz liegen daher Überlegungen und Versuche zugrunde, welche die Anwendbarkeit des SMT-Solving zur Testdatengenerierung für SL/TL-Modelle adressieren. Das Resultat dieser Betrachtungen ist eine Definition von Einsatzkriterien. Aus diesen Kriterien ergibt sich eine hybride Testdatengenerierung – für einzelne CG beziehungsweise SG wahlweise mittels SMT-Solving oder über eine Testdatensuche. Das Ziel dieses Beitrags ist es, über eine solch hybride Lösung für die Gesamtheit der betrachteten CGs/SGs effizienter Testdaten erzeugen zu können als es der ausschließliche Einsatz einer Testdatensuche zu leisten imstande ist.

Der folgende Abschnitt behandelt das SMT-Solving im Allgemeinen. Anschließend werden die angesprochenen Einsatzkriterien und der Ansatz dieser Arbeit zur Testdatengenerierung mittels SMT-Solving vorgestellt. Dessen Hybridisierung mit einer Testdatensuche wird danach behandelt. Wie bei der Vorstellung der vorherigen Beiträge endet auch dieser Teil der Arbeit mit einer Betrachtung verwandter Arbeiten.

4.2.2 SATISFIABILITY MODULO THEORIES

Die Abkürzung SMT steht für *Satisfiability Modulo Theories* und bezeichnet eine Klasse sogenannter Solver (zu Deutsch: Löser), welche der Lösung mathematischer, durch aussagenlogische und arithmetische Ausdrücke beschriebener Erfüllbarkeitsprobleme dienen. Im Allgemeinen bilden diese Probleme eine Kategorie des *Constraint-Satisfaction-Problems* (CSP, zu Deutsch: Bedingungserfüllungsproblem) [34]. Die Aktivität des Lösens solcher Probleme wird in dieser Arbeit als SMT-Solving bezeichnet.

Wie De Moura und Bjørner in einem einführenden Artikel [34] zum Thema SMT erklären, kann SMT als eine Erweiterung von *SAT* angesehen werden. SAT ist eine Abkürzung für den englischen Begriff *Satisfiability* (zu Deutsch: Erfüllbarkeit). Mit SAT werden Erfüllbarkeitsprobleme der Aussagenlogik bezeichnet. Ein SAT-Solver adressiert also die Frage, ob ein aussagenlogischer Ausdruck wie beispielsweise $(a \wedge b) \vee \neg a$, in dem a und b Boolesche Variablen darstellen, erfüllbar oder unerfüllbar ist.

Kann die Erfüllbarkeit festgestellt werden, liefert ein SAT-Solver üblicherweise ein sogenanntes Modell, welches als Beleg der Erfüllbarkeit eine Variablenbelegung enthält. Dies gilt gleichermaßen für SMT-Solver.

Das durch SAT betrachtete Boolesche CSP ist für praktische Anwendungen jedoch oftmals nicht ausreichend. In der Praxis lassen sich Probleme manchmal nicht oder nur schwer mit ausschließlich Booleschen Operatoren und Variablen ausdrücken. Die Verwendung der Prädikatenlogik erster Stufe ist für viele Anwendungsfälle beispielsweise deutlich komfortabler. SMT ergänzt SAT daher dahingehend, dass zusätzlich sogenannte *Hintergrund-Theorien* berücksichtigt werden können [92]. Um beispielsweise Ausdrücke der Form $a > b \wedge b = c + 5$ mit $a, b, c \in \mathbb{Z}$ lösen zu können, ist eine Theorie notwendig, welche eine entsprechende Arithmetik und somit letzten Endes die Bedeutung der Symbole $>$, $=$, $+$ und 5 definiert. Darüber hinaus existieren auch Theorien zum Umgang mit Arrays, Listen, Zeichenketten und mehr.

Zur Lösung von Ausdrücken spezieller Theorien übersetzen SMT-Solver einen Ausdruck entweder in ein SAT-lösbares Problem oder verwenden Theorie-spezifische Lösungsstrategien und Entscheidungsprozeduren. Ansätze, welche auf die erstere Herangehensweise setzen, werden häufig auch als *eager* (zu Deutsch: begierig oder eifrig) bezeichnet – Ansätze der zweiten Kategorie hingegen als *lazy* (zu Deutsch: faul oder träge). Diese Wortwahl verdeutlicht, dass eine unter Umständen schwierige und herausfordernde Überführung hin zu einem SAT-tauglichen Problem bei der Konstruktion eines SMT-Solvers in der Regel die präferierte Lösung darstellt. Allerdings ist dies nicht für alle ergänzenden Theorien überhaupt möglich. Die hintergründige Funktionsweise von SMT- und SAT-Solvern soll an dieser Stelle allerdings nicht zu weit vertieft werden. Bei Interesse empfiehlt sich ein Blick in das Buch *Handbook of Satisfiability* von Biere et al. [18] und insbesondere in das dort enthaltene Kapitel von Barrett et al. bezüglich SMT [13].

Bedeutsamer für das in dieser Arbeit mittels SMT-Solving verfolgte Ziel ist, welche Theorien und logischen Konstrukte einzelne SMT-Solver unterstützen und inwiefern sie zu einer effizienten Lösung einer größeren Menge entsprechender Ausdrücke imstande sind. Denn während SAT-Probleme bereits NP-vollständig sind, gilt die Lösung von in der Prädikatenlogik erster Stufe definierten Problemen in der Theorie sogar als unentscheidbar [92].

Beispiele aktueller SMT-Solver sind *Z3* [19, 33], *MathSAT 5* [25] und *CVC4* [15]. Viele SMT-Solver ermöglichen unter anderem die Nutzung von Quantoren oder nicht-linearer Funktionen. Z3 kann sogar mit der Multiplikation zweier reellwertiger Variablen in Ausdrücken umgehen. Eine Lösung von Problemdefinitionen, welche derlei Sprachmittel verwenden, können die Solver allerdings nicht garantieren. Aufgrund seiner Breite an unterstützten Theorien und Konstrukten ist die Wahl eines

SMT-Solvers im Rahmen dieser Arbeit auf Z3 gefallen. Auch bezüglich Laufzeit und Effizienz überzeugt der Z3-Solver im Vergleich mit anderen SMT-Solvern [16, 26]. Der Z3-Solver findet übrigens auch in dem in Abschnitt 2.3.3 erwähnten Testwerkzeug *Pex* [100] zum Zwecke einer Testdatengenerierung für *.NET*-Programme Anwendung.

Zum Verständnis der folgenden Abschnitte ist ein weiterer Aspekt von Relevanz. Im Bemühen um eine höhere Akzeptanz und einfachere Anwendbarkeit der SMT-Solver sowie eine einfachere Vergleichbarkeit und Austauschbarkeit derselben haben Forscher und Entwickler, welche im Bereich des SMT-Solving aktiv sind, ein einheitliches Austauschformat namens *SMT-LIB* geschaffen [14, 27]. Der Großteil der derzeit verfügbaren SMT-Solver kann in diesem Format aufgestellte Problemdefinitionen verarbeiten. Auch wenn in der Umsetzung der in dieser Arbeit vorgestellten Testdatengenerierung mittels SMT-Solving eine direkte Kommunikation über eine Schnittstelle des Z3-Solvers erfolgt, wird für einzelne Darstellungen im Folgenden die SMT-LIB-Sprache genutzt. Die hierfür wichtigsten Grundzüge dieser Sprache lassen sich kurz erklären: Jegliche Operationen, das heißt sowohl aussagenlogische, relationale als auch andere mathematische Operationen, werden eingeklammert in Präfix-Schreibweise ausgedrückt. Als Beispiele seien die Ausdrücke `(= x y)` (Gleichheit), `(=> a b)` (Implikation), `(and a b)` (Konjunktion), `(or a b)` (Disjunktion), `(not a)` (Negation) und `(* x y)` (Multiplikation) genannt – wobei `x, y` als Zahlenwerte und `a, b` als Boolesche Variablen deklariert sind.

4.2.3 EINSATZKRITERIEN

Eine Testdatengenerierung mittels SMT-Solving stellt im Grunde eine symbolische Ausführung dar. Vergleicht man eine symbolische Ausführung von SL/TL-Modellen allerdings mit derer von Programmcode, so ist Folgendes festzustellen: SL/TL-Modelle enthalten im Vergleich zu herkömmlichem Programmcode einen deutlich geringeren Anteil an Kontrollfluss. Während Blöcke von Anweisungen in Programmcode sehr häufig bedingt ausgeführt werden, gesteuert beispielsweise über *if-else*-Anweisungen oder *for*-Schleifen, so kommen hiermit vergleichbare Strukturen in realen SL/TL-Modellen, umgesetzt beispielsweise durch die Verwendung bedingter Subsysteme, weniger häufig vor. Bei einer Testdatengenerierung für Programmcode durch symbolische Ausführung wird üblicherweise jeweils nur ein einzelner Programmpfad zur Ableitung von Testdaten herangezogen. Dies ist im Falle von SL/TL-Modellen grundsätzlich nicht anders. Allerdings bewirkt der geringere Anteil an Kontrollfluss in Modellen, dass die Betrachtung eines einzelnen CG oder SG nicht zu einer derart starken Reduktion der Komplexität der jeweiligen Problemstellung für das SMT-Solving führt, wie es

bei Programmcode im Allgemeinen der Fall wäre. Daher ist eine Testdatengenerierung durch symbolische Ausführung für SL/TL-Modelle grundsätzlich herausfordernder.

Wie im vorigen Abschnitt angedeutet, ist die Anwendbarkeit des SMT-Solving entsprechend der Beschaffenheit und Größe einer Problemstellung begrenzt oder unter dem Gesichtspunkt der Effizienz der Lösung einer Problemstellung durch SMT-Solving zumindest fraglich. Eine systematische Analyse, unter welchen Bedingungen eine Testdatengenerierung mittels SMT-Solving praktisch durchführbar ist oder nicht, gestaltet sich schwierig, da der Sprachumfang von SL/TL-Modellen sehr groß ist und die Möglichkeiten der Kombination verschiedener Blocktypen und der Verwendung unterschiedlicher Strukturierungskonzepte zu komplex sind. Daher konzentrieren sich die Bemühungen dieser Arbeit in dieser Hinsicht auf das Sammeln von Erfahrungswerten in der Anwendung des SMT-Solving zur Testdatengenerierung für reale SL/TL-Modelle der Firma Daimler. Hierbei wurden Modellausschnitte verschiedenen Umfangs betrachtet. Es erfolgte eine schrittweise Skalierung, das heißt eine schrittweise Vergrößerung der betrachteten Modellausschnitte. Auf diese Weise konnte identifiziert werden, ab welcher Modellgröße – welche sich beispielsweise durch die Anzahl der enthaltenen Blöcke bemessen lässt – oder bei Vorhandensein welcher Blocktypen eine Anwendung des SMT-Solving problematisch ist. Die Versuche dienten darüber hinaus dem Finden einer für das SMT-Solving geeigneten Definition der Problemstellungen einer Testdatengenerierung. Diese Definition wird wohlgemerkt erst im nachfolgenden Abschnitt vorgestellt.

Bezüglich der Grenzen eines SMT-Solving-Einsatzes konnte festgestellt werden, dass die Größe eines SL/TL-Modells, zumindest im Falle der betrachteten Beispiele, keinen signifikanten Einfluss auf die Anwendbarkeit des SMT-Solving hat. Es wurde jedoch deutlich, dass die Art der in den Modellausschnitten vorkommenden Blöcke einen starken Einfluss auf eine grundsätzliche oder effiziente Lösbarkeit der Testdatengenerierungsaufgaben mittels SMT-Solving nimmt. So ist das Vorhandensein von Blöcken, welche gewisse mathematische Operationen wie die Anwendung trigonometrischer Funktionen (Sinus, Kosinus und Weitere) bewerkstelligen, problematisch. Zudem erhöht die Existenz zeitdynamischer Blöcke, wie *Unit Delay* oder *Discrete-Time Integrator*, die Komplexität einer Problemstellung ungemein – insbesondere wenn mehrere solche Blöcke in Kombination auftreten.

In Konsequenz setzt diese Arbeit beim Einsatz von SMT-Solving auf Einsatzkriterien, welche das Vorhandensein bestimmter Blocktypen in den Signalpfaden eines CG adressieren. Die Signalpfade eines CG umfassen in diesem Zusammenhang alle Blöcke des betrachteten SL/TL-Modells, die visuell betrachtet zwischen den für ein CG relevanten Modelleingängen (siehe Abschnitt 3.2) und den am CG beteiligten Modellsignalen

liegen – und die direkt oder transitiv über Signale verbunden sind. Ein einzelner Signalpfad ist hierbei eine jede Abfolge von Signalverbindungen, die von einem der am CG beteiligten Modellsignale zu einem der Modelleingänge führen. Die Einsatzkriterien werden für die CGs eines SG geprüft. Kommt in den hierbei betrachteten Signalpfaden ein Block vor, dessen Typ durch die Einsatzkriterien ausgeschlossen ist, so erfolgt die Testdatengenerierung für dieses SG nicht per SMT-Solving.

Die Menge der bei diesem Vorgehen akzeptierten Blocktypen basiert auf den zuvor geschilderten Erfahrungswerten. Die Blocktyp-Untermenge ist dabei recht strikt definiert, damit ein SMT-Solving letzten Endes nur dann stattfindet, wenn dieses vom verwendeten SMT-Solver zügig, das heißt innerhalb weniger Sekunden, durchgeführt werden kann. Über die Einsatzkriterien sind zum einen Blöcke ausgeschlossen, deren Funktionsweise nicht bekannt ist oder nicht ohne weitergehende Analysen ermittelt werden kann (Beispiel: *S-Function*-Blöcke). Zum anderen sind Blöcke ausgeschlossen, welche für ein SMT-Solving problematische mathematische Funktionen anwenden – wie beispielsweise die zuvor erwähnten trigonometrischen Funktionen. Blöcke mit weniger problematischen mathematischen Operationen wie Summen- oder Produktbildungen sind hingegen zugelassen. Darüber hinaus sind keine Blöcke erlaubt, welche eine zeitliche Dynamik besitzen. Eine zumindest teilweise Miteinbeziehung zeitlicher Dynamik in das SMT-Solving ist wohlgemerkt denkbar, wird in dieser Arbeit allerdings nicht weiter untersucht. Betreffend einer weniger strikten Auslegung der Einsatzkriterien insgesamt ergibt sich vermutlich noch Potenzial zu einer weiteren Effizienzsteigerung des in dieser Arbeit insgesamt entstandenen hybriden Testdatengenerierungsverfahrens.

4.2.4 DEFINITION DES TESTDATENGENERIERUNGS-PROBLEMS

Damit SMT-Solving zur Testdatengenerierung eingesetzt werden kann, erfolgt für jedes SG eine Problemdefinition. Diese stellt die Grundlage zur Erzeugung entsprechender Testdaten via SMT-Solving dar. Eine solche Problemdefinition wird im Folgenden als *Testdatengenerierungsproblem* (TDGP, Plural: TDGPs) bezeichnet. Zur Erstellung eines TDGP wird die selbe Repräsentation des SL/TL-Modells verwendet, welche auch für die statischen Voranalysen genutzt wird. Diese Modellrepräsentation wird im Zuge der statischen Voranalysen gegebenenfalls transformiert (siehe Absatz zwei zu Beginn von Kapitel 3 und Abschnitt 3.1.3). Die durchgeführten Transformationen stellen lediglich eine Zuschneidung des Modells auf die für eine Testdatengenerierung relevanten Aspekte sowie eine Vereinfachung aufgrund der Ergebnisse der statischen Voranalysen dar. Daher kann diese Modellrepräsentation ohne Probleme zur

Ableitung von TDGPs genutzt werden. Die zuvor durchgeführten Transformationen sind sogar von Vorteil, denn Vereinfachungen im Modell führen zu Vereinfachungen der TDGPs.

Ein TDGP wird durch eine rückwärts gerichtete symbolische Ausführung je CG, welches Teil eines SG ist, geformt. Dies bedeutet, die Blöcke des betrachteten Modells werden nicht entsprechend ihrer Ausführungsreihenfolge, beginnend mit den Modelleingängen, symbolisch ausgeführt – sondern in umgekehrter Reihenfolge, ausgehend von den an einem CG beteiligten Modellsignalen. Ein solches Vorgehen wird im Englischen als *Backward Symbolic Execution* bezeichnet [52]. Der Vorteil einer rückwärts gerichteten symbolischen Ausführung besteht in ihrer Orientierung an einer bestimmten Zielstellung. Diese verhindert, dass irrelevante Pfade – im betrachteten Anwendungsfall Signalpfade – analysiert werden. Würde nämlich eine klassische symbolische Ausführung durchgeführt werden, so würden von allen Modelleingängen ausgehend auch Signalpfade analysiert werden, welche in keinerlei Zusammenhang mit dem betrachteten CG stehen. Würde man übrigens fest davon ausgehen, dass die statische Voranalyse SDA genutzt wird, so ist auch eine klassische symbolische Ausführung denkbar. Denn dann wäre bekannt, welche Modelleingänge für ein CG relevant sind und die symbolische Ausführung könnte ausgehend von nur diesen Eingängen beginnen. Das nachfolgend vorgestellte Vorgehen zur Bildung der TDGPs lässt rein theoretisch beide Vorgehensweisen zu. Abbildung 27 veranschaulicht das Vorgehen.

VORFORMULIERUNG VON BEDINGUNGEN

Grundsätzlich besteht das TDGP eines jeden SG aus einer Menge von Bedingungen (englisch: Constraints), welche im Folgenden, wo notwendig, in der SMT-LIB-Sprache formuliert sind. In die Bedingungsmenge eines TDGP fließen nicht nur Bedingungen ein, die sich (1) aus den Blockfunktionalitäten entlang der relevanten Signalpfade ableiten, sondern auch Bedingungen, welche sich (2) für die beteiligten Modelleingänge aus der Modelleingangsspezifikation ergeben. Hinzu kommen selbstverständlich Bedingungen, welche die Zielstellung der Testdatengenerierung, das heißt (3) die Ausdrücke der in dem SG enthaltenen CGs abbilden. Da einzelne Modellblöcke und Modelleingänge im Falle von (1) und (2), sowie einzelne Modellsignale in allen drei Fällen, meist für die Bedingungsmengen mehrerer TDGPs relevant sind, eine einmalige Bildung der dazugehörigen Bedingungen allerdings genügt, erfolgt diese in vier vorbereitenden Schritten.

Im *ersten* vorbereitenden Schritt wird für jedes Modellsignal über eine Bedingung eine Variable deklariert. Bei einem vektorisierten Signal erfolgt für jede Signaldimension eine Variablendeklaration. Spielt ein Signal bei der Ableitung von Bedingungen aus Blockfunktionalität, Modelleingangsspezifikation oder CG-Ausdrücken eine Rolle, so wird die hierzu

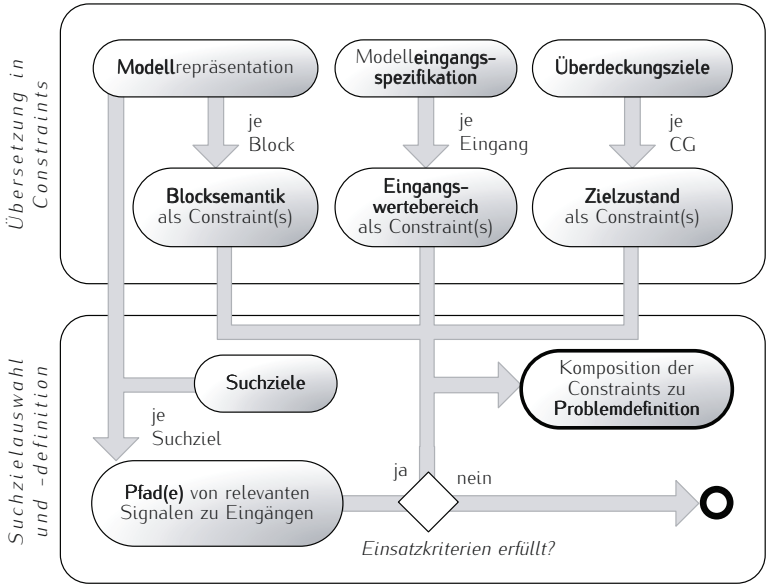


Abbildung 27: Ermittlung der Suchziel-Problemdefinitionen für den Einsatz eines SMT-Solvers

deklarierende Bedingung in die Bedingungsmenge des TDGP aufgenommen – vorausgesetzt, sie ist in dieser Menge noch nicht enthalten.

Im *zweiten* vorbereitenden Schritt werden für jeden Modelleingang die in der Modelleingangsspezifikation angegebenen Wertebereiche, falls angegeben inklusive Quantisierung, in Bedingungen übertragen. Hierdurch wird sichergestellt, dass die durch SMT-Solving erzeugten Testdaten diese Wertebereichseinschränkungen einhalten. Lautet der Wertebereich eines Modelleingangs beispielsweise $[10;20]_2$ und bezeichnet s das vom Inport-Block ausgehende Signal, so leiten sich die Bedingungen $(\geqslant \texttt{s 10})$, $(\leqslant \texttt{s 20})$ und $(\texttt{is\_int}\ (\texttt{div}\ (-\ \texttt{s 10})\ \texttt{2}))$ ab. Letztere Bedingung verlangt, dass der für s generierte Wert die Quantisierung 2 berücksichtigt.

Im *dritten* vorbereitenden Schritt werden für jeden im Modell enthaltenen Block Bedingungen abgeleitet, die dessen Funktionalität formulieren. Für drei ausgewählte Blocktypen sind die abzuleitenden Bedingungen in Tabelle 13 aufgeführt. Im Folgenden werden die dortigen Formulierungen erläutert. Allgemein gilt: Die Sammlung von Bedingungen erfolgt je Ausgang eines Blocks, das heißt gegebenenfalls auch je Vektorelement eines Ausgangssignals. Die Abbildung $\mathrm{OC}: S^{\mathbb{N}^+} \to \mathcal{P}(\mathrm{SMT})$ ordnet dementsprechend jedem aus dem Block ausgehenden Signal (mit Vektorindex-Angabe) eine Menge von Bedingungen zu. Die Menge

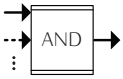
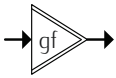
Blocktyp	*SMT-Definition*
 Inport	**Voraussetzung:** $\exists\, sb \in B : b \in BC_{sb}$ $\forall v \in \mathbb{N}_{[1,d(s_1)]}$: $\mathbf{OC}(s^v) = \{\ (= s^v\ s_1^v)\ \}$ **mit** $s = sig_{out}(b,1) \wedge s_1 = sig_{in}(sb,n)$
 Logical Operator	**Exemplarisch für:** $\bullet = \wedge$ und $n > 1$ $\forall v \in \mathbb{N}_{[1,maxd_{in}(b)]}$: $\mathbf{OC}(s^v) =$ $\left\{ \begin{array}{l} (=> \text{ (and (not } (= s_1^w\ 0)) \ldots \text{ (not } (= s_n^w\ 0))) \ (= s^v\ 1)) \text{ ,} \\ (=> \text{ (or } (= s_1^w\ 0) \ldots (= s_n^w\ 0)) \ (= s^v\ 0)) \end{array} \right\}$ **mit** $s = sig_{out}(b,1) \wedge s_x = sig_{in}(b,x) \wedge w = vc(v,s)$
 Gain	$\forall v \in \mathbb{N}_{[1,max(n,d(b,1))]}$: $\mathbf{OC}(s^v) = \{\ (= s^v\ (*\ gf^{min(v,n)}\ s_1^{vc(v,s_1)}))\ \}$ **mit** $s = sig_{out}(b,1) \wedge s_1 = sig_{in}(b,1)$

Tabelle 13: Übersetzung der Funktionalität eines jeden Blocks in für das SMT-Solving verwendbare Bedingungen (Auszug)

SMT bezeichnet hierbei die Menge aller möglichen SMT-LIB-konformen Bedingungen. Von Blöcken, die gemäß der Einsatzkriterien nicht zu berücksichtigen sind, leiten sich keine Bedingungen ab.

Inport. Für einen Inport-Block sind nur dann Bedingungen zu formulieren, wenn sich dieser in einem Subsystem befindet. Andernfalls handelt es sich um einen Modelleingang und ein solcher wurde bereits im zweiten vorbereitenden Schritt behandelt. Ist der Inport-Block allerdings innerhalb eines Subsystems, so wird eine Bedingung formuliert, die ausdrückt, dass die seinem Ausgangssignal (gegebenenfalls je Vektorindex) zugeordnete Variable der Variablen des entsprechenden Eingangssignals des übergeordneten Subsystems (mit gleichem Vektorindex) entspricht.

Logical Operator. Die Bedingungen, welche sich für einen Logical-Operator-Block ableiten, sind in der Tabelle beispielhaft für den AND-Operator und die Blockvariante mit mehreren Eingängen angegeben. Eine Bedingung formuliert hierbei, unter welcher Voraussetzung der Block den Wert 1 (*wahr*) ausgibt – nämlich dann, wenn der Signalwert eines jeden Blockeingangs ungleich 0, also *wahr*, ist. Die zweite Bedingung gibt an, dass der Block den Wert 0 (*falsch*) ausgibt, sofern einer der in den Block eingehenden Signalwerte gleich 0 (*falsch*) ist.

Gain. Für einen Gain-Block leitet sich eine Bedingung ab. Diese gibt an, dass die dem Ausgangssignal des Blocks (gegebenenfalls je Vektorindex) zugeordnete Variable der mit einem Faktor multiplizierten Variable des Eingangssignals des Blocks (mit gleichem Vektorindex) entspricht. Der Faktor ist hierbei, wie in Abschnitt 2.1.2 definiert, ein unter Umständen vektorisierter Blockparameter.

Die in der Tabelle angegebenen Bedingungsdefinitionen gelten, wie auch die Bedingungsdefinitionen aller weiteren SL/TL-Blocktypen, im Übrigen auch für Blöcke, welche sich in bedingten Subsystemen befinden. Die Bedingung zur Ausführung eines bedingten Subsystems ist nämlich schon an anderer Stelle berücksichtigt: Befinden sich die an einem CG beteiligten Signale innerhalb des bedingten Subsystems, so fließt dessen Ausführungsbedingung bereits über den CG-Ausdruck (siehe Abschnitt 2.2.1) mit in die Bedingungsmenge eines TDGP ein. Befinden sich die an einem CG beteiligten Signale entlang des Signalflusses im Modell betrachtet hinter einem bedingt ausgeführten Subsystem, so wird dessen Ausführungsbedingung bei der Bildung der Bedingungen für den oder die sich im Subsystem befindlichen Outport-Block/-Blöcke aufgenommen.

Im *vierten* vorbereitenden Schritt wird für jedes CG, welches Teil eines SG ist, einmalig eine Bedingung gebildet. Diese beschreibt die Erfüllung des CG und kann daher direkt aus dem CG-Ausdruck abgeleitet werden. Für das CG mit dem Ausdruck $s_1 > 5 \wedge s_2 = 1$ leitet sich beispielsweise die Bedingung $(\texttt{and}\ (>\ \texttt{s}_1\ \texttt{5})\ (=\ \texttt{s}_2\ \texttt{1}))$ ab.

KOMPOSITION DER BEDINGUNGEN

Die für Signale, Modelleingänge, Blöcke und CGs vorformulierten Bedingungen werden anschließend je SG in ein TDGP überführt. Dies geschieht durch eine rückwärts gerichtete symbolische Ausführung. Initial werden die vorformulierten Bedingungen der CGs des jeweiligen SG in das TDGP aufgenommen. Anschließend werden, ausgehend von den Signalen, welche an diesen CGs beteiligt sind, alle Signalpfade bis hin zu den Modelleingängen traversiert. Die in den Signalpfaden enthaltenen Blöcke werden hierbei in umgekehrter Reihenfolge zu ihrer eigentlichen Ausführungsreihenfolge (siehe Abschnitt 2.1.2) besucht.

Je Block kommt es zu einer symbolischen Ausführung. Hierbei werden die vorformulierten Bedingungen für die Blockausgänge, welche in einem der Signalpfade liegen, in das TDGP aufgenommen. Für das Signal eines jeden solchen Blockausgangs wird (gegebenenfalls je Vektorindex) zudem die Bedingung, welche die dazugehörige Variablendeklaration enthält, aufgenommen. Handelt es sich bei dem betrachteten Block um einen Inport-Block, der zugleich einen Modelleingang darstellt, so wird das

TDGP um die vorformulierten Bedingungen ergänzt, welche für diesen Modelleingang aus der Modelleingangsspezifikation gebildet wurden.

Kommt es zu der symbolischen Ausführung eines Blocks, der den zuvor vorgestellten Einsatzkriterien nicht entspricht, so wird die Traversierung der Signalpfade abgebrochen und das in diesem Zusammenhang behandelte SG als für SMT-Solving ungeeignet markiert. Dadurch ist bei der späteren Abarbeitung aller SGs nachvollziehbar, für welche SGs eine Testdatengenerierung nicht mittels SMT-Solving durchzuführen ist. Dieser Aspekt der TDGP-Bildung ist auch in Abbildung 27 dargestellt.

4.2.5 SIGNALGENERIERUNG

Ein TDGP stellt die Eingabe für einen SMT-Solver dar. Der Solver prüft die Erfüllbarkeit des TDGP. Seine Antwort lautet *sat* (erfüllbar), *unsat* (unerfüllbar) oder *unknown* (unbekannt). Letztere Antwort bedeutet, dass der Solver das TDGP nicht lösen konnte. Durch die Verwendung von Einsatzkriterien für das SMT-Solving ist die Wahrscheinlichkeit im Anwendungsfall dieser Arbeit jedoch gering, dass ein TDGP mit *unknown* ausgewertet wird. Auch sollte die Antwort *unsat* selten vorkommen, sofern die statischen Voranalysen SIA und CGSeq eingesetzt werden, um einen Großteil der unerfüllbaren CGs vorab zu identifizieren. Dennoch ist eine Behandlung dieser möglichen Fälle notwendig. Bevor allerdings die Einbettung des SMT-Solving in den Testdatengenerierungsprozess Thema dieses Kapitels ist, soll es an dieser Stelle erst einmal darum gehen, wie im Falle der Antwort *sat* Testdaten gebildet werden.

Das zur Erfüllung eines SG gesuchte Testdatum lässt sich über ein Modell bilden, welches der SMT-Solver für ein erfüllbares TDGP erzeugt. Da der Modellbegriff in dieser Arbeit bereits anderweitig genutzt wird, sei dieses Modell als *Lösungsmodell* bezeichnet. Ein Lösungsmodell stellt einen Beleg der Erfüllbarkeit dar, also ein Beispiel, für welches die betrachtete Bedingungsmenge erfüllt ist. Im vorliegenden Anwendungsfall enthält ein Lösungsmodell Wertebelegungen für jene Variablen eines TDGP, welche die aus den Inport-Blöcken der obersten Hierarchie-Ebene des SL/TL-Modells ausgehenden Signale und somit die Modelleingänge bezeichnen. Da die Einsatzkriterien dazu führen, dass kein TDGP zeitliche Dynamik enthält, liegt im Lösungsmodell je Modelleingang nur ein Variablenwert und nicht etwa mehrere Werte für unterschiedliche Zeitschritte vor.

Um aus den Variablenbelegungen ein plausibles Testdatum zu formen, gilt es bei einer Testdatengenerierung mittels SMT-Solving, wie schon im Falle des LGA (siehe Abschnitt 2.4.2) und der ASPO (siehe Abschnitt 4.1), die Modelleingangsspezifikation zu berücksichtigen. Auf einfachstem Wege ist dies durch eine Überführung des Solver-Lösungsmodells in

Optimierungssequenzen, wie sie auch durch LGA und ASPO verwenden werden, realisierbar. Je Modelleingang ist demnach eine Optimierungssequenz zu erzeugen. Hierbei ist zu berücksichtigen: Die Variablenbelegungen des Lösungsmodells adressieren nur einen einzigen Zeitschritt der für die Modelleingänge zu erzeugenden Signale (Simulationssequenzen). Daher reicht es aus, wenn die Variablenbelegung des Lösungsmodells für einen Modelleingang im Verlauf des hierfür erzeugten Signals nur einmal enthalten ist. Dies kann erzielt werden, indem der Amplitudenwert eines einzigen Segments der entsprechenden Optimierungssequenz auf den Wert der Variablenbelegung gesetzt wird. Auf alle auf diese Weise adressierten Modelleingänge bezogen müssen diese Amplitudenwerte allerdings alle zum gleichen Zeitschritt in den aus Optimierungssequenzen generierten Simulationssequenzen enthalten sein.

Dies wird folgendermaßen sichergestellt: Ist für keinen der Modelleingänge in der Modelleingangsspezifikation ein Startamplitudenwert angegeben, so bilden die Variablenbelegungen des Lösungsmodells die Startamplitudenwerte der jeweiligen Optimierungssequenzen. Mit Startamplitudenwert ist hierbei der Amplitudenparameter des ersten Signalsegments gemeint. Da dieses Segment keine weiteren Parameter besitzt, sondern lediglich den Startamplitudenwert für das Folgesegment angibt, enthält eine jede Simulationssequenz die dazugehörige Variablenbelegung zum ersten Zeitschritt. Gibt es für mindestens einen Modelleingang hingegen einen Startamplitudenwert, so bildet eine jede Variablenbelegung den Amplitudenwert des jeweils letzten Segments einer jeden Optimierungssequenz. Hierdurch tauchen die Variablenbelegungen in den Simulationssequenzen stets im allerletzten Zeitschritt auf.

Alle anderen, zu diesem Zeitpunkt noch offenen Parameter der Optimierungssequenzen, das heißt Amplituden-, Breiten- und Transitionstyp-Parameter der enthaltenen Segmente, werden unter Berücksichtigung der Modelleingangsspezifikation zufällig bestimmt. Dies gilt übrigens auch für alle Parameter der Optimierungssequenzen, welche für Modelleingänge erzeugt werden, denen durch das Solver-Lösungsmodell keine Variablenbelegung zugewiesen ist. Der Zufall wird an dieser Stelle aus folgendem Grund miteinbezogen: Wie im nachfolgenden Abschnitt zu erfahren ist, wird das betrachtete SL/TL-Modell mit dem wie hier beschrieben erzeugten Testdatum ausgeführt. Auf das Ziel der Testdatengenerierung für alle betrachteten SGs bezogen ist es wünschenswert, dass die hierbei erzielte Kollateralüberdeckung möglichst hoch ist. Eine zufällige Bestimmung der irrelevanten Segmentparameter bei der Erzeugung eines jeden Testdatums ist hierzu dienlicher als ein Festsetzen dieser Parameter auf immer gleiche Standardwerte. Diese Überlegung spielte bereits bei der Erzeugung von Signalen für irrelevante Modelleingänge im Rahmen der statischen Voranalyse SDA eine Rolle (siehe Abschnitt 3.2.2).

4.2.6 KOMBINATION VON SUCHE UND SMT-SOLVING

Die Einbettung der Testdatengenerierung mittels SMT-Solving in das hybride Testverfahren dieser Arbeit ist in Abbildung 28 dargestellt. Insbesondere ist dort Form und Ablauf des Hybrids aus SMT-Solving und Testdatensuche zu erkennen. Die Formulierung der TDGPs für jedes SG erfolgt, wie zu sehen, nach Durchführung der statischen Voranalysen. Hierbei werden auch die Einsatzkriterien geprüft, so dass für jedes SG angegeben ist, ob der Versuch einer Testdatengenerierung mittels SMT-Solving unternommen werden soll.

Anschließend werden für die SGs, gemäß der durch die statische Voranalyse CGSeq bestimmten Abarbeitungsreihenfolge, Testdaten generiert. Hierbei kommt vorzugsweise SMT-Solving zum Einsatz, sofern ein SG als hierfür geeignet gilt. Andernfalls wird eine Testdatensuche angestoßen. Das Vorgehen einer Testdatensuche, sei es mittels des LGA oder der ASPO, wurde in dieser Arbeit bereits ausführlich behandelt. Sollen für ein SG Testdaten per SMT-Solving generiert werden, so wird der SMT-Solver mit dem vorbereiteten TDGP ausgeführt. Antwortet der Solver mit *unsat*, so wird zum nächsten SG übergegangen. Enthält das SG zudem nur ein einziges CG, so wird dieses als unerfüllbar markiert. Im Falle eines zusammengesetzten SG bleibt hingegen offen, welches der enthaltenen CGs für die Unerfüllbarkeit verantwortlich ist. Antwortet der SMT-Solver mit *unknown*, so war er nicht erfolgreich. Daher wird in diesem Fall eine Testdatensuche für das SG angestoßen. Antwortet der Solver mit *sat*, so wird, wie im vorigen Abschnitt ausführlich beschrieben, aus dem Lösungsmodell des Solvers ein Testdatum geformt.

Um zu verifizieren, dass dieses Testdatum tatsächlich das Erreichen der CGs des betrachteten SG bewirkt, wird das SL/TL-Modell mit dem Testdatum ausgeführt. Dieser Schritt dient außerdem der Ermittlung, welche anderen CGs und somit SGs eventuell zufällig mit erreicht werden (Kollateralüberdeckung). Bestätigt die Modellausführung, dass das Testdatum das aktuell betrachtete SG erfüllt, so wird es in die insgesamt zu erstellende Testsuite aufgenommen. Anschließend wird zum nächsten SG übergegangen. Kommt die statische Voranalyse CGSeq zum Einsatz, so wird zuvor noch die Abarbeitungsreihenfolge der SGs aktualisiert. Der Sinn und Zweck dieses Schritts ist in Abschnitt 3.4 dargelegt.

Mit der Hybridisierung von Suche und symbolischer Ausführung erfährt das in Abschnitt 3.3.3.2 vorgestellte Vorgehen zur SG-Reihenfolgenbildung durch CGSeq eine Änderung. Die über eine Priorisierung der SGs gebildete Reihenfolge unterscheidet nun zwischen SGs, für welche SMT-Solving angewendet wird, und denen, für die vorab schon feststeht, dass eine Testdatensuche durchgeführt wird. Diese Boolesche Metrik erhält den Bezeichner *Mo*. SMT-Solving-taugliche SGs werden hierbei bevorzugt, das heißt sie landen in der Abarbeitungsreihenfolge vor sonstigen SGs.

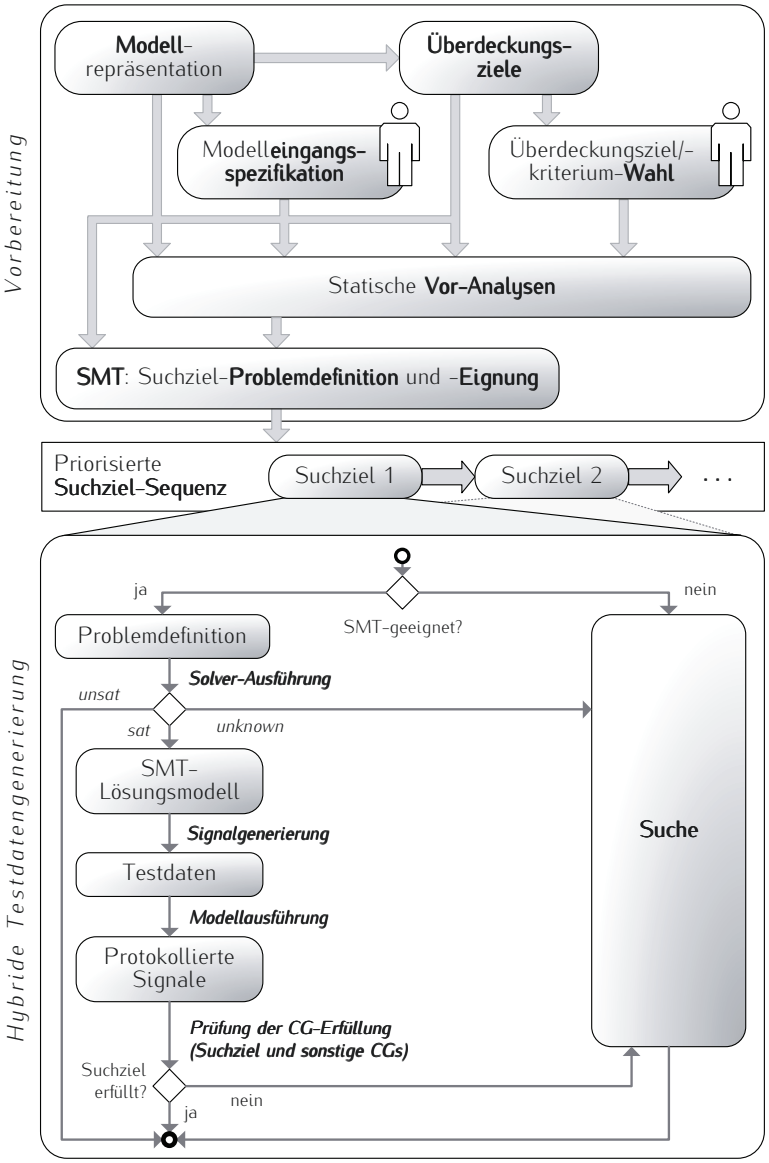


Abbildung 28: Kombination von Suche und SMT-Solving zu einem hybriden Testdatengenerierungsverfahren für SL/TL

Die sonstigen SGs werden anschließend entsprechend der ursprünglich verwendeten Metriken priorisiert. SMT-Solving-taugliche SGs hingegen werden zuerst nach der geringeren Modelltiefe eines SG (M2) und bei diesbezüglicher Gleichheit nach der höheren Anzahl implizierter CGs (M3) geordnet. Dies bedeutet, dass in diesem Fall die Priorisierungsmetriken der Anzahl Boolescher Signale (M1) und der Anzahl relevanter Modelleingänge (M4) entfallen. Der Grund hierfür ist, dass diese Metriken für eine Testdatengenerierung durch SMT-Solving keine Relevanz haben. Boolesche Zieldefinitionen stellen zum einen kein Problem für eine solche Testdatengenerierung dar und zum anderen betrachtet die rückwärts gerichtete symbolische Ausführung bedingt durch ihre Funktionsweise ohnehin ausschließlich relevante Modelleingänge. Vergleichend zur Darstellung der ursprünglichen Priorisierung in Abbildung 20 stellt Abbildung 29 die erweiterte Priorisierung dar.

Da die Bildung der TDGP, wie in Abbildung 28 erkennbar, erst nach den statischen Voranalysen und somit erst nach CGSeq stattfindet, ist die SG-Reihenfolge nach der TDGP-Bildung zu aktualisieren. Alternativ dazu könnte die TDGP-Bildung auch innerhalb von CGSeq erfolgen – zwischen Bildung der SGs und deren Priorisierung.

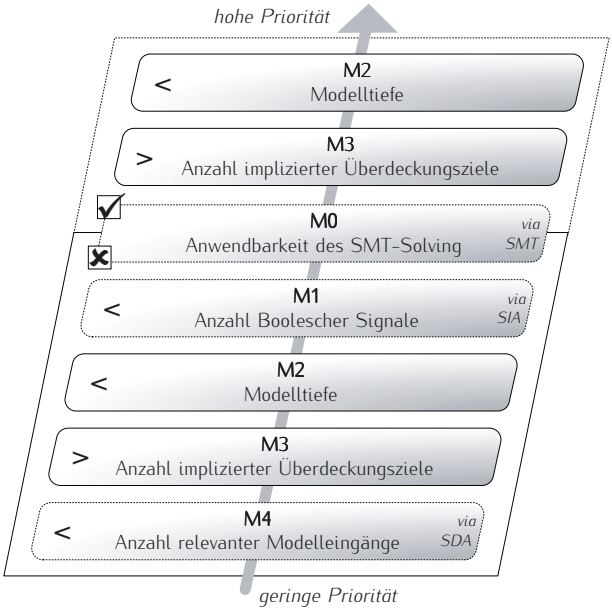


Abbildung 29: Anpassung der Suchzielpriorisierung bei Verwendung von SMT-Solving und Suche in Kombination

Mit diesen Ausführungen ist der letzte Baustein des hybriden Testverfahrens dieser Arbeit vorgestellt. Der Baustein ergänzt die bereits um statische Voranalysen erweiterte Testdatensuche um eine Testdatengenerierung durch symbolische Ausführung. Zwischen Testdatensuche und symbolischer Ausführung erfolgt hierbei eine SG-spezifische, Kriterien-basierte Auswahl. Die in Kapitel 6 vorgestellten Fallstudien untersuchen, ob und inwieweit sich eine derartige Hybridisierung positiv auf die Effizienz einer automatisierten Testdatengenerierung auswirkt.

4.2.7 VERWANDTE ARBEITEN

Im Grundlagen-Kapitel dieser Arbeit in Abschnitt 2.3.1 wurden bereits zwei verwandte Arbeiten, eine von Roy und Shankar [101, 102] sowie eine von Păsăreanu et al. [96], im Kontext einer symbolischen Ausführung für SL-Modelle behandelt. Wie dort bereits herausgestellt wurde, unterscheiden sich die Ansätze der beiden Arbeiten vom hier präsentierten Ansatz in mehreren Punkten. Die zwei wesentlichsten Punkte sind hierbei, dass die Plausibilität der generierten Testdaten dort nicht betrachtet wird und die Grenzen einer symbolischen Ausführung in Anwendung für SL-Modelle entweder nicht erörtert oder nicht adressiert werden.

Eine Hybridisierung mehrerer Testdatengenerierungstechniken über eine Kriterien-basierte Auswahl ist Gegenstand einer Arbeit von Peranandam et al. [95] (behandelt in Abschnitt 2.3.3). Die verwendeten Kriterien nennen die Autoren allerdings nicht. Die Plausibilität der generierten Testdaten steht zudem nicht im Fokus ihres Ansatzes.

Eine verwandte Arbeit, welche sich mit einer Anwendung des SMT-Solving für SL-Modelle beschäftigt, stammt von Herber et al. [65]. Auch wenn es den Autoren nicht um die Generierung von Testdaten geht, sondern um eine formale Verifikation von Modellen (Model-Checking und Invariantenprüfung), so behandeln sie doch auch eine Übersetzung der Funktionalität eines SL-Modells hin zu einer für SMT-Solving tauglichen Repräsentation. Die Übersetzung erfolgt allerdings in die Eingabesprache des Verifikationswerkzeugs *UCLID*. Dieses nutzt zur Erfüllung seiner Verifikationsaufgaben SMT-Solving.

In Bezug zu diesen Arbeiten zeichnet sich der in diesem Teil der vorliegenden Arbeit vorgestellte Ansatz durch folgende Eigenschaften aus:

1. Definition von Einsatzkriterien zur Testdatengenerierung mittels SMT-Solving für SL/TL-Modelle
2. Beachtung der Signalplausibilität bei Erzeugung über SMT-Solving
3. Hybridisierung von Suche, symbolischer Ausführung und statischen Voranalysen

Teil III

REALISIERUNG UND EVALUIERUNG

5 TESTWERKZEUG-PROTOTYP

Die in dieser Arbeit vorgestellten Ideen und Techniken wurden in ein prototypisches Testwerkzeug überführt. Dieses ermöglicht eine automatisierte Testdatengenerierung für SL/TL-Modelle. Das Werkzeug, welches im Folgenden unter Verwendung des Namens *TASMO* (Testing via Automated Search for Models) vorgestellt wird, dient im Rahmen der vorliegenden Arbeit hauptsächlich der Evaluierung der zuvor behandelten Beiträge. Darüber hinaus dient der Werkzeug-Prototyp allerdings auch der Heranführung der Konzepte des suchbasierten Tests mitsamt ergänzender Techniken an die Praxiswelt. TASMO nutzt einen Teil der Implementierung, die im Rahmen der Arbeit von Windisch [132] entstanden ist. Dies betrifft im Wesentlichen die Umsetzung des Suchverfahrens LGA und des segmentbasierten Signalgenerierungsansatzes.

In den folgenden drei Abschnitten werden technische Hintergründe zu TASMO, der Ablauf einer Anwendung des Werkzeugs sowie Herausforderungen bei der Entwicklung eines solchen Werkzeugs beleuchtet.

5.1 ARCHITEKTUR

Ein Blick in Abbildung 30 verrät, dass TASMO grundlegend aus zwei Komponenten besteht: einer Client- und einer Server-Anwendung. Die Funktionalitäten des Testwerkzeugs sind zum wesentlichen Teil in der Programmiersprache *Java* realisiert. Dieser Teil von TASMO stellt die Server-Anwendung dar. Die Client-Anwendung kommuniziert mit dieser. Die Client-Anwendung ist mit der ML-eigenen Skriptsprache *m* geschrieben und kann in das ML-Werkzeug integriert werden. Auf diese Weise kann der Anwender direkt aus dem SL-Editor heraus für ein geöffnetes SL/TL Modell oder für einen Ausschnitt aus einem solchen (virtuelles Subsystem) eine Testdatengenerierung initiieren. TASMO ist hierbei strukturell so aufgebaut, dass verschiedene Testdatengenerierungsszenarien (siehe Abschnitt 2.2) unterstützt werden können. Eine vollständige Implementierung ist im Rahmen dieser Arbeit allerdings nur für den Strukturtest erfolgt.

Nachfolgend wird die Architektur von TASMO vorgestellt. Die Darstellung dieser in Abbildung 30 begleitet die Ausführungen.

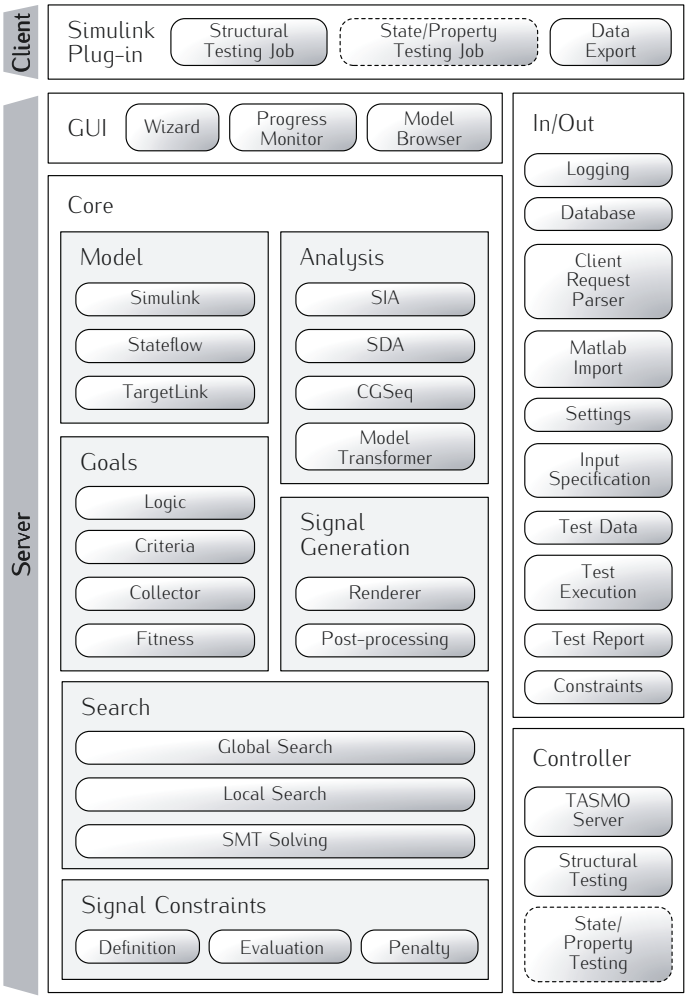


Abbildung 30: Architektur des entwickelten Tool-Prototypen

Simulink Plug-in. Diese Komponente enthält die Client-Anwendung und realisiert die Integration von TASMO in den SL-Editor. Ist ein Modell in diesem geöffnet, so kann über einen Menüeintrag die Testdatengenerierung angestoßen werden. Der Client prüft dann, ob die Server-Anwendung bereits läuft. Ist dies der Fall, so übermittelt der Client dem Server die vom Nutzer gewünschte Aufgabe (Testart und Modellinformationen) – andernfalls wird zuerst ein Start der Server-Anwendung initiiert. Bevor aber beispielsweise die Aufgabe einer Testdatengenerierung zum

Zwecke eines Strukturtests übermittelt wird, prüft die Client-Anwendung eine Reihe von Vorbedingungen, wie die Ausführbarkeit des im SL-Editor geöffneten Modells, und sammelt im Falle eines positiven Ergebnisses Informationen über das Modell (Unterkomponente *Data Export*). Diese stehen der Server-Anwendung im Anschluss zur Verfügung. Insbesondere wird die Struktur des betrachteten SL/TL-Modells exportiert. Ein Modellparser nutzt hierzu die Programmierschnittstelle (API) von ML und überführt das Modell in eine XML-Datei. Bei dem Parser handelt es sich um eine Weiterentwicklung des SL-Parsers von Scheible [107].

Controller. Die *Controller*-Komponente der Server-Anwendung bildet die Funktionslogik der verschiedenen Anwendungsfälle ab, wobei der Prototyp nur die strukturorientierte Testdatengenerierung umsetzt. Zur Erfüllung ihrer Teilaufgaben greift die Komponente auf die Komponenten *Core* und *In/Out* zurück. Zur visuellen Darstellung der Abläufe und zur Interaktion mit dem Anwender nutzt sie die *GUI*-Komponente.

GUI. Diese Komponente regelt auf Anweisung der *Controller*-Komponente die grafische Darstellung der Benutzeroberfläche von TASMO und stellt der *Controller*-Komponente zudem Benutzerinteraktionen und -eingaben zur Verfügung. Die Vorbereitung einer Testdatengenerierung durch den Anwender erfolgt hierbei in Form eines *Wizard*. Bei diesem handelt es sich um einen aus mehreren Schritten bestehenden Benutzerdialog. Während einer Testdatengenerierung wird zudem deren Fortschritt grafisch dargestellt (Unterkomponente *Progress Monitor*). Sowohl innerhalb des Wizards als auch während der Testdatengenerierung kann die Unterkomponente *Model Browser* genutzt werden. Diese stellt das betrachtete SL/TL-Modell grafisch dar und reichert die Darstellung durch zusätzliche Informationen, wie die sich von den Blöcken des Modells ableitenden oder die durch die Testdatengenerierung aktuell verfolgten CGs, an.

In/Out. Diese Komponente stellt zum einen Kommunikationsschnittstellen zur Verfügung, wie beispielsweise zur Interaktion von TASMO mit einer Instanz des ML-Werkzeugs. Zum anderen ermöglicht sie das Ablegen und Einlesen verschiedener Daten, welche entweder Grundlage einer Testdatengenerierung sind oder welche während einer Testdatengenerierung entstehen. Es existieren folgende Unterkomponenten.

Client Request Parser. Dieser verarbeitet die XML-basierten Aufträge der Client-Anwendung. Im Falle einer strukturorientierten Testdatengenerierung enthält ein Auftrag unter anderem Angaben zu dem Speicherpfad des SL/TL-Modells, dessen Dateinamen sowie den Namen und Pfad eines vom Client erzeugten Arbeitsordners für abzulegende Dateien und Ergebnisse. In diesem Ordner legt die Client-Anwendung übrigens auch das als XML-Datei extrahierte SL/TL-Modell ab.

Matlab Import. Diese Unterkomponente liest die von der Client-Anwendung exportierten Modellinformationen ein und erstellt aus ihnen eine

Modellrepräsentation: Von dieser werden im Falle eines Strukturtests CGs abgeleitet. Zudem dient sie der Durchführung der statischen Voranalysen (siehe Kapitel 3) und der symbolischen Ausführung via SMT-Solving (siehe Abschnitt 4.2).

Input Specification. Diese Unterkomponente kann eine vom Nutzer erstellte Modelleingangsspezifikation in einem XML-Format abspeichern und in umgekehrter Richtung auch wieder einlesen.

Test Data. Erzeugte Testdaten können durch diese Unterkomponente in einem XML-Format abgelegt werden. Zudem können Testdaten, welche unter Umständen aus anderweitigen Tests stammen, eingelesen werden. Diese können beispielsweise bei einer automatisierten Testdatengenerierung im Sinne eines Strukturtests als Grundlage dienen. Auf diesen Aspekt wird im nächsten Abschnitt noch näher eingegangen. Der Prototyp TASMO unterstützt aktuell allerdings nur das eigene XML-Format.

Test Execution. Diese Unterkomponente bewerkstelligt die Ausführung einer Kopie des betrachteten SL/TL-Modells mit Testdaten. Hierzu gehören auch die Initialisierung und Instrumentierung der Modell-Kopie sowie das Auslesen der protokollierten Signalverläufe. Zur Kommunikation mit ML und zur Auswertung der protokollierten Daten werden die Java-Bibliotheken *matlabcontrol*[1] und *JMatIO*[2] eingesetzt.

Test Report. Statistiken und Ergebnisse einer abgeschlossenen Testdatengenerierung werden von dieser Unterkomponente in einem Web-basierten Bericht zusammengefasst.

Constraints. Wie in Abschnitt 2.3.2 erwähnt, können in dem suchbasierten Ansatz von Windisch [132] zusätzliche Testdaten-Bedingungen berücksichtigt werden. Derartige Bedingungen kann die Unterkomponente *Constraints* in eine Datei überführen oder aus einer Datei einlesen.

Settings. Diese Unterkomponente verwaltet, das heißt speichert und lädt, generelle Einstellungen des TASMO-Werkzeugs. Für die Durchführung der Fallstudien dieser Arbeit enthalten die Einstellungen übrigens auch Parameter, über die sich die einzelnen statischen Voranalysen (SIA, SDA, CGSeq) de-/aktivieren lassen und über die sich das zu verwendende Testdatengenerierungsverfahren (LGA, ASPO oder der Hybrid aus LGA und SMT-Solving) auswählen lässt.

Database. Für die Auswertung der Fallstudien dieser Arbeit sammelt diese Unterkomponente relevante Daten in einer *SQLite*-Datenbank.

Core. Die *Core*-Komponente stellt den Kern der Funktionalität von TASMO dar und enthält verschiedene Datenstrukturen und Algorithmen. Sie ist in die nachfolgend aufgeführten Unterkomponenten gegliedert.

1 `https://code.google.com/p/matlabcontrol` (letzter Zugriff: 01.04.2014)
2 `http://sourceforge.net/projects/jmatio` (letzter Zugriff: 01.04.2014)

Model. Die Struktur der internen Repräsentation des betrachteten SL/TL-Modells wird unter Verwendung dieser Unterkomponente gebildet. Sie bildet sowohl generische Strukturelemente wie Signale oder Blöcke ab, als auch spezifische Blocktypen – unterschieden nach SL-eigenen und TL-eigenen Blocktypen sowie dem speziellen SF-Blocktyp.

Goals. Der Aufbau eines CG-Ausdrucks sowie eines SG ist in dieser Unterkomponente definiert. Die Ableitung der CGs von einem Modell und deren Zuordnung zu Überdeckungskriterien ist ebenfalls hier enthalten. Darüber hinaus stellt die Unterkomponente Funktionen zur Berechnung der Fitness eines CG oder SG zur Verfügung.

Analysis. Diese Unterkomponente enthält die Umsetzungen der drei statischen Voranalysen SIA, SDA und CGSeq. Zusätzlich existieren eine Reihe von Funktionen zur Modelltransformation. Diese werden unter anderem von SIA verwendet.

Search. Die zur Testdatengenerierung verwendeten Algorithmen sind Bestandteil dieser Unterkomponente: das globale Suchverfahren LGA, das lokale Suchverfahren ASPO und die symbolische Ausführung mittels SMT-Solving. Die einzelnen Algorithmen greifen zu ihrer Erfüllung insbesondere auf die folgenden drei Komponenten zurück: *Signal Generation*, *Test Execution* und *Fitness* aus der Komponente *Goals*.

Signal Generation. Die Erzeugung von Optimierungssequenzen und deren Umwandlung in Simulationssequenzen realisiert diese Komponente.

Signal Constraints. Die Struktur zusätzlicher Testdaten-Bedingungen und das Vorgehen zur Berücksichtigung solcher bei einer Testdatensuche ist Gegenstand dieser Unterkomponente. Da dieser Aspekt nicht im Fokus der vorliegenden Arbeit steht, unterscheidet sich diese Komponente nicht von der diesbezüglichen Umsetzung im Rahmen der Arbeit von Windisch [132]. Dies bedeutet auch, dass zusätzliche Testdaten-Bedingungen von TASMO zwar neben dem LGA auch von ASPO berücksichtigt werden, nicht aber durch das SMT-Solving.

5.2 NUTZERSICHT UND FUNKTIONSWEISE

Die zuvor beschriebenen Bausteine werden vom TASMO-Werkzeug zum Zweck einer automatisierten Testdatengenerierung für ein SL/TL-Modell benutzt. Hierbei steht, wie auch im Rahmen der vorliegenden Arbeit insgesamt, der Strukturtest im Vordergrund. In diesem Szenario ergibt sich bei Verwendung von TASMO der folgend erläuterte Ablauf. Dieser ist in Abbildung 31 dargestellt und betrachtet sowohl die Sicht des Anwenders auf den gesamten Vorgang, als auch die wichtigsten internen Funktionalitäten von TASMO. Diese bleiben dem Anwender allerdings verborgen.

Eine Interaktion mit dem Anwender ist nur an den Stellen notwendig, die in der Darstellung mit einer kleinen Figur markiert sind.

Eine Testdatengenerierung zur strukturellen Überdeckung eines im SL-Editor geöffneten Modells wird vom Anwender durch Anklicken des entsprechenden Menüeintrags von TASMO im SL-Editor eingeleitet. TASMO ermittelt daraufhin die benötigten Modelldaten und erzeugt eine eigene Modellrepräsentation. Diese nutzt TASMO im nächsten Schritt, um die Beschaffenheit der Eingänge des Modells zu analysieren. Bei den Ein- und Ausgängen des Modells handelt es sich nämlich unter Umständen um Bus-Systeme oder Vektorsignale, wobei Beides sogar in verschachtelter Form vorkommen kann. Damit TASMO weiß, wie viele und welche Signale tatsächlich zu generieren sind, ist eine Schnittstellenanalyse notwendig. Nach dieser geht TASMO alle Blöcke der

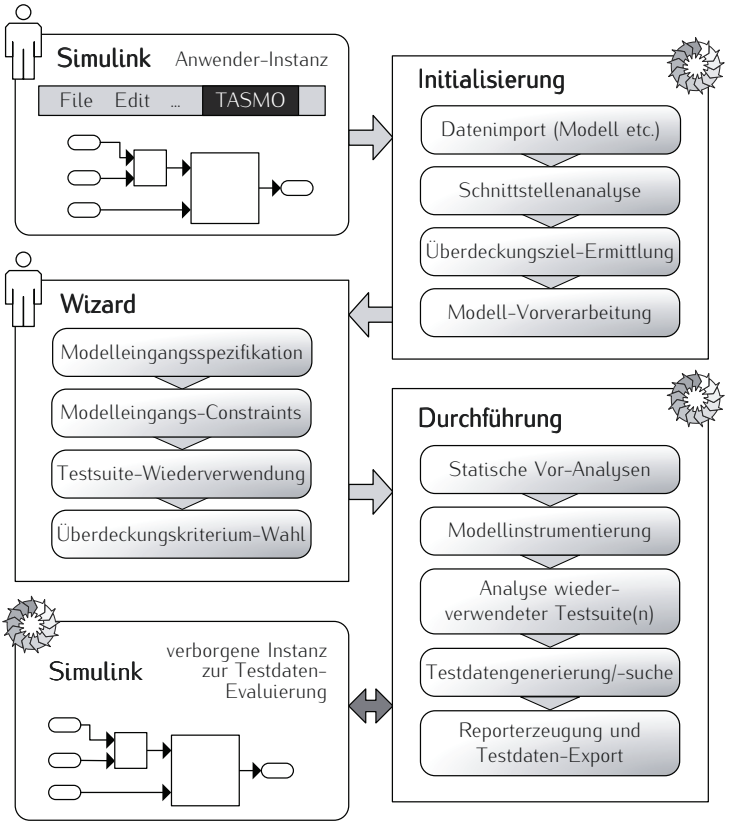
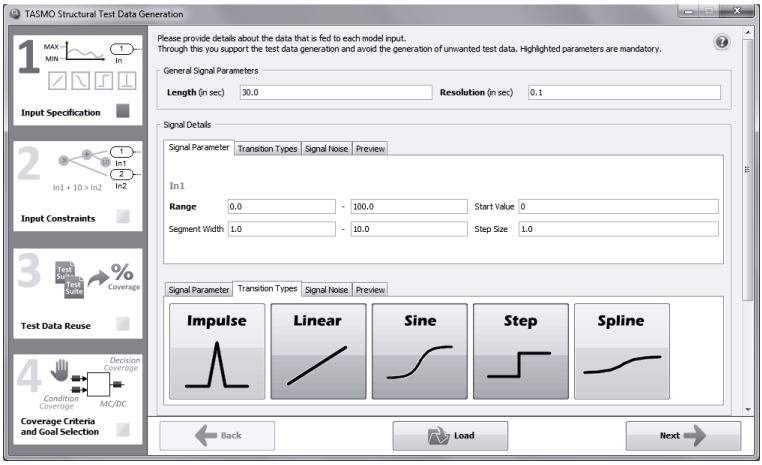


Abbildung 31: Ablauf der strukturorientierten Testdatengenerierung und Nutzerinteraktion mit dem Tool-Prototypen

Modellrepräsentation durch und leitet die CGs aller in dieser Arbeit unterstützten Überdeckungskriterien ab (siehe Abschnitt 2.2.1). Wie zu Beginn des dritten Kapitels erwähnt, führt TASMO zudem initiale Modelltransformationen durch. Diese reduzieren das Modell auf den für die Testdatengenerierung relevanten Modellanteil.

Dem Anwender bleibt diese Initialisierung verborgen. Nachdem er im SL-Editor die strukturorientierte Testdatengenerierung angewählt hat, zeigt sich ihm als Nächstes das Fenster des TASMO-Werkzeugs mit dem ersten Schritt eines Wizards (siehe Abbildung 32a). Der Wizard dient der Vorbereitung der Testdatengenerierung durch den Anwender. Im ersten Schritt erfolgt die Spezifikation der Modelleingänge, wie in Abschnitt 2.4.2 behandelt. Zum Beispiel müssen für jeden Modelleingang Wertebereichsangaben gemacht werden. Anschließend besteht die Möglichkeit, für jeden Modelleingang zusätzliche Bedingungen in einer temporalen Logik anzugeben. Obwohl dieser Aspekt, wie im vorigen Abschnitt angesprochen, nicht im Fokus dieser Arbeit steht, enthält der TASMO-Wizard einen entsprechenden Schritt, um die Möglichkeiten einer suchbasierten Testdatengenerierung zu verdeutlichen. Der dritte Schritt des Wizards erlaubt es dem Anwender, bereits existierende Testdaten wiederzuverwenden. Da einem Strukturtest in der Praxis häufig ein Funktionstest, beziehungsweise ein anforderungsbasierter Test, vorausgeht (siehe Abschnitt 2.1.3), existieren demnach auch bereits Testdaten oder Testfälle. Diese erzielen bereits einen gewissen Überdeckungsgrad. Um TASMO nur nach Testdaten für noch nicht erreichte CGs fahnden zu lassen, ist im Wizard daher die Möglichkeit eines Imports von Testfällen oder Testdaten vorgesehen. Im letzten Schritt des Wizards wählt der Anwender das oder die gewünschte/-n Überdeckungskriterium/-kriterien, oder wahlweise einzelne CGs der unterstützten Überdeckungskriterien aus (siehe Abbildung 32b). Die Auswahl einzelner CGs kann der Anwender hierbei auch im erwähnten Modellbrowser tätigen.

Ist der Anwender am Ende des Wizards angelangt, das heißt hat er alle verpflichtenden Angaben (Modelleingangsspezifikation und Kriterien/CGs-Wahl) und gegebenenfalls optionale Angaben (Testdaten-Bedingungen und Testdaten-Import) gemacht, so startet die automatische Durchführung des gesamten Testdatengenerierungsprozesses (siehe Abbildung 33a). Hierbei werden im ersten Schritt die drei statischen Voranalysen SIA, SDA und CGSeq ausgeführt. Wegen der Fallstudien dieser Arbeit ist es möglich, dass einzelne Voranalysen deaktiviert sind. Im Falle der Deaktivierung von CGSeq gilt: Es werden trotzdem SGs gebildet, diese enthalten aber jeweils nur ein selektiertes CG (siehe Abschnitt 3.3.3.1). Die SGs werden in diesem Fall zudem zufällig sortiert. Im zweiten Schritt wird eine Kopie des betrachteten SL/TL-Modells für spätere Ausführungen mit Testdaten vorbereitet. Hierzu wird eine neue, für den Anwender nicht sichtbare Instanz des ML-Werkzeugs geöffnet und in dieser Workspace-Daten sowie die Modellkopie geladen. Zudem

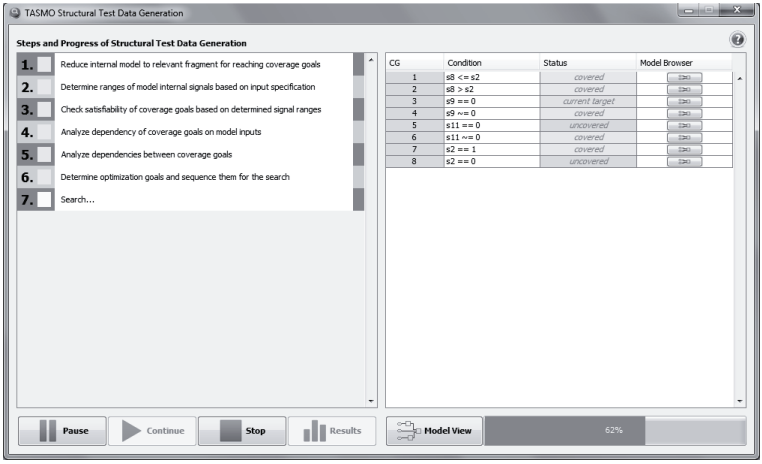


(a) Modelleingangsspezifikation

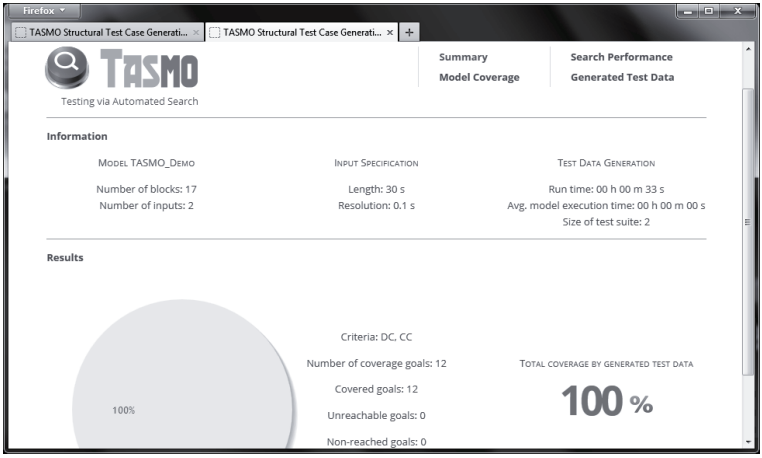
(b) Überdeckungskriterien-Wahl oder Überdeckungsziel-Auswahl

Abbildung 32: Einblick in den Wizard des Werkzeug-Prototypen

wird die Modellkopie instrumentiert, das heißt für alle Signale, welche zur Auswertung der CGs protokolliert werden müssen, wird ein entsprechendes Feature des SL-Werkzeugs aktiviert. Die protokollierten Daten werden hierdurch nach einer Modellausführung im ML-Workspace abgelegt. Im dritten Schritt werden gegebenenfalls importierte Testdaten ausgeführt und registriert, welche CGs, und somit SGs, durch diese bereits erreicht sind. Im vierten Schritt erfolgt die Testdatengenerierung für alle noch nicht erreichten SGs, je nach Voreinstellung durch die AS-



(a) Testdatengenerierung



(b) Testreport

Abbildung 33: TASMO: Testdatengenerierung und Testreport

PO, den LGA oder den Hybrid aus LGA und SMT-Solving. Die hierbei erstellte Testsuite wird nach Beendigung der Testdatengenerierung in einem XML-Format abgespeichert. Zudem wird jedes in der Testsuite enthaltene Testdatum als ausführbares m-Skript abgelegt, so dass der Anwender das SL/TL-Modell später noch einmal mit diesem Testdatum ausführen kann. Im letzten Schritt erzeugt TASMO einen Bericht mit Statistiken, wie dem erreichten Überdeckungsgrad, und einer Detailansicht der erzeugten Testdaten (siehe Abbildung 33b).

5.3 TECHNISCHE UND PRAKTISCHE HERAUSFORDERUNGEN

Die Entwicklung eines Werkzeugs wie dem Prototypen TASMO birgt eine Reihe von technischen und praktischen Herausforderungen. Diese sollen an dieser Stelle in aller Kürze angerissen werden. Neben der bereits erwähnten Schnittstellenanalyse, welche notwendig ist, um die Signalstruktur an den Modelleingängen zu bestimmen, stellen bestimmte Eigenheiten und Sonderfälle der ML- und SL-Umgebung sowie Individuallösungen in der Entwicklungspraxis weitere Herausforderungen.

So erlaubt es SL beispielsweise nicht, Signale zu protokollieren, welche direkt aus bedingten Subsystemen ausgehen und die zu einem *Merge*-Block führen. TASMO löst dieses Problem, indem es im Zuge der Modellinstrumentierung zwischen Subsystem und Merge-Block einen Gain-Block mit dem Faktor eins einfügt. Ein solcher Gain-Block verändert die Signalwerte nicht, sein Ausgangssignal lässt sich jedoch protokollieren. Eine weitere Schwierigkeit besteht in der effizienten Übertragung einer hohen Anzahl protokollierter Signalverläufe von ML in ein Java-basiertes Werkzeug. Eine direkte Übertragung, bei der jeder Signalverlauf innerhalb einer Programmschleife übermittelt wird, dauert außerordentlich lange. Um die Übertragung protokollierter Daten möglichst schnell zu gestalten, lässt TASMO diese von ML in eine sogenannte *MAT*-Binärdatei exportieren und importiert diese im Java-Werkzeug. Diese beiden Operationen laufen selbst bei recht vielen protokollierten Signalen sehr schnell ab.

Auch wenn das prototypische Werkzeug TASMO bereits viele Eigenheiten und Sonderfälle berücksichtigt, so bedarf es dennoch einer weitergehenden Entwicklung, um aus TASMO ein in der industriellen Praxis einsetzbares Werkzeug zu formen. Beispielsweise gilt es, das Vorhandensein individueller Entwicklungsumgebungen in SL zu berücksichtigen. In vielen Entwicklungsprojekten kommen nämlich individuelle, in der m-Sprache implementierte Tool-Suiten zum Einsatz. Diese schaffen beispielsweise direkte Verknüpfungen zu im Entwicklungsprozess benachbarten Werkzeugen oder regeln den Zugriff auf gemeinsame Ressourcen. Da ein SL/TL-Modell ohne die Initialisierung einer solchen Entwicklungsumgebung oftmals nicht ausgeführt werden kann, ist eine entsprechende Initialisierung in der ML-Instanz, welche bei TASMO der Modellausführung dient, zu bewerkstelligen. Dies ist aufgrund der vielen Formen einer Initialisierung – beispielsweise über ein m-Skript mit grafischer Benutzeroberfläche oder über eine Batch-Datei – nicht immer unproblematisch. Ein weiteres Problem besteht in dem Vorhandensein verschiedener Programmversionen von SL und TL in Entwicklungsprojekten. Ein praxistaugliches Werkzeug muss die Unterschiede dieser Versionen berücksichtigen.

6 FALLSTUDIEN

Zur Bewertung der Beiträge dieser Arbeit wurde der Werkzeug-Prototyp TASMO eingesetzt. Es galt, im Sinne eines Strukturtests Testdaten für zwei reale SL/TL-Modelle aus der Automobilindustrie zu erzeugen. Die einzelnen Beiträge dieser Arbeit, das heißt die statischen Voranalysen SIA, SDA und CGSeq sowie das lokale Suchverfahren ASPO und die symbolische Ausführung mittels SMT-Solving, wurden hierbei jeweils einer gesonderten Untersuchung unterzogen. Dieses Kapitel stellt zunächst den Versuchsaufbau der Fallstudien vor. Anschließend werden deren Ergebnisse präsentiert, eingeordnet und die Beiträge dieser Arbeit bewertet.

6.1 VERSUCHSAUFBAU

Der Versuchsaufbau adressiert neben den Eckdaten der verwendeten SL/TL-Modelle die zu beantwortenden Thesen und die der Beantwortung dienenden Kriterien. In diesem Zusammenhang werden verschiedene Konfigurationen des hybriden Testverfahrens aufgestellt. Die Ergebnisse der mit diesen Konfigurationen durchgeführten Fallstudien werden im Anschluss an diesen Abschnitt vorgestellt.

6.1.1 TESTOBJEKTE/MODELLE

Die für die Fallstudien verwendeten SL/TL-Modelle entstammen aktuellen Entwicklungsprojekten der Daimler AG. Es werden zwei verschiedene Modelle betrachtet. Das im folgenden als Modell *A* bezeichnete SL/TL-Modell realisiert eine Funktion in einem alternativen (elektrischen/hybriden) Antriebsstrang eines Fahrzeugs. Das zweite, als Modell *B* bezeichnete SL/TL-Modell implementiert Funktionalitäten, die im Bereich des Fahrzeuginnenraums zur Anwendung kommen. Über die Funktionalitäten, Schnittstellen und Inhalte beider Modelle können an dieser Stelle aus Gründen der Geheimhaltung leider keine detaillierten Angaben gemacht werden. Um aber dennoch einen Eindruck bezüglich der Komplexität dieser Modelle und einer Testdatengenerierung für

diese zu erhalten, seien folgende Größenangaben und Charakteristiken genannt.

Modell *A* enthält eine hohe dreistellige Anzahl an Blöcken – Modell *B* ist bezüglich der Blockanzahl fast dreimal so umfangreich. Modell *B* weist im Vergleich zu Modell *A* einen höheren Anteil logischer Blocktypen auf – das heißt Blocktypen, deren ausgehende Signale einen Booleschen Wertebereich haben. Modell *A* enthält mehr Schleifen (Rückkopplungen) und somit eine höhere zeitliche Dynamik als Modell *B*. Die Schnittstellen beider Modelle enthalten eine niedrige zweistellige Anzahl an Eingängen – wobei Modell *B* mehr Eingänge als Modell *A* besitzt.

Für beide Modelle wurde im Rahmen der Fallstudien jeweils eine Modelleingangsspezifikation gebildet. Die dort gemachten Angaben basieren zum einen auf Informationen, die – wie zum Beispiel die Wertebereichsangaben – aus Modell-begleitenden Entwicklungsdokumenten stammen, und zum anderen auf Auskünften der Modellentwickler. Trotz einer geringeren Anzahl an Eingängen ist der für die Testdatengenerierung so entscheidende Suchraum im Fall von Modell *A* größer (beziehungsweise komplexer) als bei Modell *B*. Während nämlich nur 17% der Eingänge von Modell *A* einen Booleschen Wertebereich aufweisen, so sind dies bei Modell *B* ganze 50%.

Neben der individuellen Spezifikation der Modelleingänge gilt es, die zeitliche Länge und Abtastrate der zu generierenden Signale vorab Modelleingangs-übergreifend festzulegen. Die Abtastrate ist in den Simulationseinstellungen eines Modells vermerkt und kann direkt übernommen werden. Sie liegt im Falle beider Modelle im Millisekundenbereich. Als zeitliche Länge für die zu generierenden Signale wurden 30 Sekunden im Fall von Modell *A* und 20 Sekunden im Fall von Modell *B* gewählt. Dieser Festlegung liegt eine manuelle Betrachtung der zeitlichen Dynamik der Modelle zugrunde, denn es ist sicherzustellen, dass die generierten Signale ausreichend lang sind. Sind sie nämlich zu kurz, so könnte dies zur Unerreichbarkeit mancher CGs führen.

Das hybride Testverfahren dieser Arbeit hatte in den Fallstudien die Aufgabe, Testdaten zur strukturellen Überdeckung beider Modelle zu generieren. Eventuell in den Entwicklungsprojekten bereits existierende Testfälle oder Testdaten, beispielsweise aus anforderungsbasierten Tests, wurden nicht berücksichtigt. Als Überdeckungskriterien wurde sowohl die Bedingungsüberdeckung (CC) als auch die Entscheidungsüberdeckung (DC) herangezogen. Für diese beiden Kriterien leiten sich für Modell *A* insgesamt 600 CGs und für Modell *B* insgesamt 1113 CGs ab.

6.1.2 THESEN

In diesem Abschnitt werden Thesen aufgestellt. Diese verkörpern Erwartungen an die Beiträge dieser Arbeit. Die Fallstudien dienen der Stützung, Relativierung oder Widerlegung dieser Erwartungen. In Kapitel 1 wurde auf Seite 8 die zentrale These dieser Arbeit aufgestellt. Diese lautet:

Die Effizienz des Ansatzes von Windisch zur suchbasierten Testdatengenerierung für SL-Modelle lässt sich durch eine Hybridisierung derselben mit geeigneten statischen oder dynamischen Techniken deutlich erhöhen.

Diese These wird für die Fallstudien nun in einzelne Thesen aufgeschlüsselt. Diese adressieren jeweils den Einfluss der einzelnen Beiträge dieser Arbeit auf die Effizienz des suchbasierten Ansatzes. Die Gültigkeit dieser Thesen impliziert die Gültigkeit der zentralen These dieser Arbeit. Darüber hinaus werden ergänzende Thesen aufgestellt, welche weitergehende Aspekte der einzelnen Beiträge oder die automatisierte Testdatengenerierung insgesamt adressieren. Solche Thesen sind in der folgenden Auflistung jeweils mit einem kleinen Sternchen markiert. Eine ausführliche Erklärung der aufgelisteten Thesen erfolgt, sofern jeweils notwendig, im Zuge der Analyse der Fallstudienergebnisse hinsichtlich dieser Thesen.

Für die statische Voranalyse SIA werden folgende Thesen aufgestellt:

These T1a. *Existieren in einem Modell unerreichbare CGs, so steigert der Einsatz von SIA die Effizienz der Testdatengenerierungsvorgänge insgesamt.*

These T1b. *Der zeitliche Aufwand von SIA ist geringer als der Zeitgewinn, den der Einsatz von SIA für die Testdatengenerierungsvorgänge insgesamt bewirkt.*

These T1c. *Der durch Testdatensuchen erzielte Überdeckungsgrad ist bei Einsatz von SIA mindestens genauso hoch wie ohne diesen.*

Die zweite statische Voranalyse SDA wird unter Berücksichtigung folgender Thesen bewertet:

These T2a. *Der Einsatz von SDA steigert die Effizienz der Testdatengenerierungsvorgänge insgesamt.*

These T2b. *Der zeitliche Aufwand von SDA ist geringer als der Zeitgewinn, den der Einsatz von SDA für die Testdatengenerierungsvorgänge insgesamt bewirkt.*

These T2c. *Der durch Testdatensuchen erzielte Überdeckungsgrad ist bei Einsatz von SDA mindestens genauso hoch wie ohne diesen.*

Die dritte statische Voranalyse CGSeq wird anhand folgender Thesen evaluiert:

These T3a. *Der Einsatz von CGSeq steigert die Effizienz der Testdatengenerierungsvorgänge insgesamt.*

These T3b. *Der zeitliche Aufwand von CGSeq ist geringer als der Zeitgewinn, den der Einsatz von CGSeq für die Testdatengenerierungsvorgänge insgesamt bewirkt.*

These T3c. *Der durch Testdatensuchen erzielte Überdeckungsgrad ist bei Einsatz von CGSeq mindestens genauso hoch wie ohne diesen.*

These T3d*. *Die Bildung einer Abarbeitungsreihenfolge der SGs durch CGSeq führt bei den Testdatensuchen zu einer höheren Kollateralüberdeckung oder einer erhöhten Anzahl erfolgreich abgeschlossener Suchen.*

These T3e*. *Der Einsatz von CGSeq bewirkt, dass die durch Testdatensuchen erzeugte Testsuite weniger Testdaten enthält.*

Da die drei statischen Voranalysen zum Teil untereinander auf ihre Ergebnisse zurückgreifen, wird auch die Kombination aller drei Voranalysen betrachtet. Folgende Thesen werden in diesem Zusammenhang untersucht:

These T4a. *Der gemeinsame Einsatz von SIA, SDA und CGSeq bewirkt, im Vergleich zum Einsatz nur eines Teils dieser Techniken, eine zusätzliche Effizienzsteigerung der Testdatengenerierungsvorgänge insgesamt.*

These T4b. *Der zeitliche Aufwand von SIA, SDA und CGSeq insgesamt ist geringer als der Zeitgewinn, den der Einsatz dieser Techniken für die Testdatengenerierungsvorgänge insgesamt bewirkt.*

Die Generierung von Testdaten über den Hybrid aus (globaler) Testdatensuche und symbolischer Ausführung wird über die folgenden Thesen bewertet:

These T5a. *Die Kriterien-basierte Testdatengenerierung mittels Suche und symbolischer Ausführung steigert die Effizienz der Testdatengenerierungsvorgänge insgesamt.*

These T5b. *Der durch den Hybrid aus Suche und symbolischer Ausführung erzielte Überdeckungsgrad ist mindestens genauso hoch wie der durch ausschließliche Suchen erreichte.*

Das lokale Suchverfahren ASPO, welches in dieser Arbeit wohlgemerkt nicht in die Bildung eines Hybrids aus Testdatengenerierungstechniken eingeflossen ist, wird innerhalb der Fallstudien über folgende These bewertet:

These T6. *Das lokale Suchverfahren ASPO ist für zumindest einen Teil der CGs in der Lage, die jeweils gesuchten Testdaten effizienter zu finden als das globale Suchverfahren LGA.*

Je nachdem, in welchen Situationen und wie häufig ASPO dies gelingt, stellt sich die Frage, inwiefern eine solche lokale Suche innerhalb des hybriden Testdatengenerierungsverfahrens eine für deren Ergebnisse oder Effizienz gewinnbringende Rolle übernehmen könnte. Eine Untersuchung dieser Fragestellung steht allerdings nicht im Fokus der Arbeit.

6.1.3 KONFIGURATIONEN

Zur Untersuchung der einzelnen Thesen erfolgt eine Aufstellung verschiedener Konfigurationen des hybriden Testverfahrens. Jede dieser Konfigurationen wird verwendet, um strukturorientiert Testdaten für die Modelle *A* und *B* zu generieren. Hierbei werden Daten bezüglich bestimmter Metriken und Kriterien gesammelt – wie im nächsten Abschnitt behandelt. Diese Daten werden durch Gegenüberstellungen verschiedener Konfigurationen verglichen, um Rückschlüsse auf die Gültigkeit der einzelnen Thesen zu ziehen.

Die in den Fallstudien betrachteten Konfigurationen sind in Tabelle 14 dargestellt. Die Konfiguration *C1* stellt hierbei den Ansatz von Windisch dar, das heißt *C1* verwendet ausschließlich die globale Suche LGA zur Testdatengenerierung. *C1* nutzt zudem keine der in dieser Arbeit eingeführten statischen Voranalysen. *C2* bis *C7* nutzen hingegen jeweils eine oder mehrere der in dieser Arbeit vorgestellten zusätzlichen Techniken. Vergleiche dieser Konfigurationen mit beispielsweise dem Zufallstest oder einem kommerziellen Werkzeug erfolgen in dieser Arbeit nicht. Windisch führte einen solchen Vergleich bereits in seiner Arbeit durch [132]. Da sein Ansatz dabei zu überlegenen Ergebnissen führte, müssen

	Statische Voranalysen			*Generierung*		
Konfiguration	*SIA*	*SDA*	*CGSeq*	*GS*	*LS*	*SMT*
C1	-	-	-	x	-	-
C2	x	-	-	x	-	-
C3	x	x	-	x	-	-
C4	x	-	x	x	-	-
C5	x	x	x	x	-	-
C6	x	x	x	x	-	x
C7	x	x	x	-	x	-

Tabelle 14: In den Fallstudien betrachtete Konfigurationen des hybriden Testdatengenerierungsverfahrens

sich die verschiedenen Konfigurationen des hier behandelten hybriden Testverfahrens nur mit der Konfiguration *C1* messen.

Die Aufstellung der Konfigurationen ist in der in der Tabelle dargestellten Vollständigkeit nicht vorab erfolgt, sondern ist das Ergebnis schrittweiser Untersuchungen. So wurde beispielsweise zuerst die SIA-Technik evaluiert und hierfür *C2* mit *C1* verglichen. Da *C2* hierbei bezüglich der erzielten Überdeckung und der Effizienz als Gewinner hervorging, wurde in der Konfiguration *C3*, die der Evaluierung der SDA-Technik dient, auch die SIA-Technik aktiviert. *C3* wurde dann mit *C2* verglichen. Wäre die SIA-Technik in *C3* nicht aktiviert, so hätte ein Vergleich mit *C1* erfolgen müssen – durch die Betrachtung unerreichbarer CGs hätte dies die Laufzeit der Fallstudien allerdings unnötig ausgedehnt. Eine Ausnahme dieses schrittweisen Vorgehens bildet die Evaluierung von CGSeq. Diese erfolgt über die Konfiguration *C4*. Hierbei wurde auf eine Aktivierung der SDA-Technik verzichtet, um unterscheiden zu können, wie stark die Auswirkungen der beiden Techniken SDA und CGSeq jeweils sind. Eine gemeinsame Aktivierung von SDA und CGSeq erfolgt stattdessen in *C5* zur Evaluierung aller statischen Voranalysen in Kombination.

Zusammenfassend stellt sich die Evaluierung der verschiedenen Beiträge dieser Arbeit anhand der zuvor definierten Thesen und über Vergleiche der Konfigurationen folgendermaßen dar:

Evaluierung der SIA-Technik. Auswertung der Thesen *T1a*, *T1b* und *T1c* durch Vergleich von *C2* mit *C1*

Evaluierung der SDA-Technik. Auswertung der Thesen *T2a*, *T2b* und *T2c* durch Vergleich von *C3* mit *C2*

Evaluierung der CGSeq-Technik. Auswertung der Thesen *T3a*, *T3b*, *T3c*, *T3d* und *T3e* durch Vergleich von *C4* mit *C2*

Evaluierung der Voranalysen-Kombination. Auswertung der Thesen *T4a* und *T4b* durch Vergleich von *C5* mit *C1*/*C2*/*C3*/*C4*

Evaluierung des Hybrids aus LGA und SMT. Auswertung der Thesen *T5a* und *T5b* durch Vergleich von *C6* mit *C5*

Evaluierung der lokalen Suche ASPO. Auswertung der These *T6* durch Vergleich von *C7* mit *C5*

6.1.4 KRITERIEN

Bei Ausführung der aufgestellten Konfigurationen des hybriden Testverfahrens sind Daten zu sammeln, die eine Bewertung der aufgestellten Thesen zulassen. Die im Folgenden aufgeführten Metriken stellen hierbei die wesentlichen Datenquellen dar.

Anzahl unerreichbarer CGs (D1). Die SIA-Technik, darauf aufbauend auch die CGSeq-Technik und zuletzt auch das SMT-Solving (siehe Abschnitt 4.2.6) sind in der Lage, unerreichbare CGs zu identifizieren. Ein CG φ, welches unerreichbar ist, bekommt den Zustand $\text{st}(\varphi) = \text{UNSAT}$ zugewiesen. Die Metrik *D1* gibt nach Abschluss des hybriden Testverfahrens die Anzahl aller CGs mit diesem Zustand an.

Anzahl erreichter CGs (D2). All jenen CGs φ, für welche durch eine Testdatensuche oder SMT-Solving erfüllende Testdaten generiert werden konnten, wird der Zustand $\text{st}(\varphi) = \text{SAT}$ zugewiesen. Die Metrik *D2* gibt deren Anzahl an. Werden SIA und zusätzlich gegebenenfalls CGSeq eingesetzt, so enthält diese Anzahl auch die Anzahl der als stets erreicht erkannten CGs. Auch wenn *D2* als ein absolutes Maß nicht unterscheidet, welche der Techniken für den SAT-Zustand verantwortlich ist, wird die Anzahl erreichter CGs in den Darstellungen der Ergebnisse zum Teil nach verantwortlicher Technik aufgeschlüsselt.

Laufzeit (D3). Während der Ausführungen der einzelnen Konfigurationen werden verschiedene zeitliche Messungen vorgenommen. So werden die Laufzeiten von SIA (Metrik *D3a*), SDA (*D3b*) und CGSeq (*D3c*) gemessen – vorausgesetzt, die jeweilige Technik kommt zum Einsatz. Zudem werden die Laufzeiten aller bei Ausführung einer Konfiguration stattfindenden Testdatensuchen aufsummiert (*D3d*). Finden Testdatengenerierungen mittels SMT-Solving statt, so werden deren Laufzeiten ebenfalls aufsummiert (*D3e*).

Anzahl betrachteter Testdaten (D4). Jegliche erzeugten Testdaten, ganz gleich ob durch den LGA, die ASPO oder SMT-Solving geschehen, werden durch diese Metrik erfasst.

Größe der Testsuite (D5). Ein Testdatum, welches während der Ausführung einer Konfiguration ein bislang unerreichtes CG erstmalig erreicht, wird in die resultierende Testsuite aufgenommen. Die Metrik *D5* bezeichnet die Anzahl der Testdaten in dieser Testsuite.

Anzahl anvisierter SGs (D6). Die Testdatengenerierungstechniken LGA, ASPO und SMT-Solving arbeiten jeweils ein oder mehrere SGs ab. Dies gilt auch, wie in Abschnitt 5.2 erklärt, wenn die statische Voranalyse CGSeq nicht zum Einsatz kommt. Da die CGs eines SG bei der Abarbeitung eines anderen SG unter Umständen erreicht werden (Kollateralüberdeckung), kann die Anzahl der bei Ausführung einer Konfiguration tatsächlich anvisierten und abgearbeiteten SGs variieren.

Erfolgsquote (D7). Der Versuch, mittels LGA, ASPO oder SMT-Solving Testdaten zur Erfüllung aller CGs eines SG zu finden, kann erfolgreich oder erfolglos sein. Die Erfolgsquote bemisst prozentual, wie häufig der Erfolgsfall eintritt. Die Metrik wird hierbei auf zweierlei Weise interpretiert – und zwar zum einen bezogen auf mehrere Anvisierungen eines bestimmten SG bei mehrfachen Ausführungen einer Konfiguration (*D7a*),

und zum anderen bezogen auf die Anvisierungen mehrerer verschiedener SGs innerhalb der Ausführung einer Konfiguration (*D7b*).

Tabelle 15 stellt dar, welche dieser Metriken in die Auswertung einer jeden der aufgestellten Thesen einfließen. Bei der Analyse der Ergebnisse werden diese Zusammenhänge aufgegriffen. Daher sei an dieser Stelle nur ein Beispiel genannt: Die These *T1a* adressiert die erwartete Effizienzsteigerung der Testdatensuchen durch den Einsatz der statischen Voranalyse SIA (Konfiguration *C2*). Grundvoraussetzung einer höheren Effizienz ist, dass die Anzahl der erreichten CGs mindestens genauso hoch ist wie ohne den Einsatz von SIA (*C1*). Des Weiteren setzt die These die Existenz unerreichbarer CGs voraus. Daher werden die Anzahlen unerreichbarer und erreichter CGs benötigt (*D1* und *D2*). Die Effizienz der Testdatensuchen wird in den Fallstudien durch deren aufsummierte

These	*D1*	*D2*	*D3a*	*D3b*	*D3c*	*D3d*	*D3e*	*D4*	*D5*	*D6*	*D7a*	*D7b*
T1a	x	x	-	-	-	x	-	x	-	-	-	-
T1b	-	-	x	-	-	x	-	-	-	-	-	-
T1c	x	x	-	-	-	-	-	-	-	-	-	-
T2a	-	x	-	-	-	x	-	x	-	-	-	-
T2b	-	-	-	x	-	x	-	-	-	-	-	-
T2c	x	x	-	-	-	-	-	-	-	-	-	-
T3a	-	x	-	-	-	x	-	x	-	-	-	-
T3b	-	-	-	-	x	x	-	-	-	-	-	-
T3c	x	x	-	-	-	-	-	-	-	-	-	-
T3d	-	-	-	-	-	-	-	-	-	x	-	x
T3e	-	-	-	-	-	-	-	-	x	-	-	-
T4a	-	x	-	-	-	x	-	x	-	-	-	-
T4b	-	-	x	x	x	x	-	-	-	-	-	-
T5a	-	x	-	-	-	x	x	x	-	-	-	-
T5b	x	x	-	-	-	-	-	-	-	-	-	-
T6	-	-	-	-	-	-	-	x	-	-	x	-

Tabelle 15: Relevanz der Metriken für die Auswertung der Thesen

Laufzeiten und, zur Absicherung der auf diesem Wege gewonnenen Erkenntnisse, auch durch die Anzahl generierter Testdaten bewertet. Daher fließen zusätzlich die Metriken *D3d* und *D4* in die Auswertung von These *T1a* ein.

6.1.5 EINSTELLUNGEN UND RANDBEDINGUNGEN

Die in den Fallstudien verwendeten Einstellungen der verwendeten Suchalgorithmen sowie weitere Randbedingungen, welche nicht bei der Vorstellung der Thesen, Konfigurationen und Kriterien Erwähnung fanden, sind Thema dieses Abschnitts.

GRUNDLEGENDES

Wie zu Beginn dieses Kapitels bemerkt, wurde zur Durchführung der Fallstudien das im Rahmen dieser Arbeit entstandene prototypische Werkzeug TASMO verwendet. TASMO wurde für die Fallstudien mit zusätzlichen Möglichkeiten zur Protokollierung verschiedener Daten ausgestattet. Hierdurch sammelt TASMO während der Durchführung seiner Aufgaben vor allem für die zuvor aufgeführten Metriken relevante Daten. Die Protokollierung wirkt sich durch eine Erhöhung der Laufzeiten auf die Messergebnisse der Fallstudien aus – bezogen auf die Gesamtlaufzeiten ist der Einfluss hierbei allerdings so gering, dass er vernachlässigt werden kann. Die Benutzeroberfläche von TASMO wurde bei Durchführung der Fallstudien nicht genutzt. Stattdessen wurden Skripte erstellt, welche die Abläufe vollständig automatisieren. Notwendige Nutzerangaben, wie die Angabe einer Modelleingangsspezifikation oder die Wahl der Überdeckungskriterien, wurden zuvor vorbereitet und durch die Skripte gesteuert. Auf diesem Wege konnte der manuelle Aufwand bei der Fallstudiendurchführung so gering wie möglich gehalten werden.

Der Rechner, mit dem die Studien durchgeführt wurden, verfügt über einen *Intel Core 2 Duo* Prozessor (P8600, 2.4 GHz), 4 GB Arbeitsspeicher und das Betriebssystem Windows 7 (64-bit). Zur Ausführung des TASMO-Werkzeugs wurde das *Java Runtime Environment* in Version 7 genutzt. ML/SL lag in der Version 2009b und TL in der Version 3.1 vor.

Die Suchalgorithmen LGA und ASPO treffen in ihren Abläufen zum Teil zufällige Entscheidungen, sei es bei der Bildung initialer Testdaten oder, im Falle des LGA, bei der Kombination, Mutation und Selektion von Testdaten. Um die Aussagekraft und die statistische Signifikanz der bei den Fallstudien gesammelten Daten zu stützen, werden daher alle Konfigurationen (*C1* bis *C7*) mehrfach ausgeführt und für die Analyse der Ergebnisse jeweils das Mittel der gemessenen Werte gebildet. In der Literatur zum Thema suchbasiertes Testen wird in der Regel eine

30-fache Wiederholung genutzt, um zufallsbedingte Ausschläge in den Ergebnissen zu kompensieren. Dies wurde auch in den Fallstudien der vorliegenden Arbeit so gehandhabt. Die Resultate der Fallstudien weisen allerdings, so viel sei an dieser Stelle vorweggenommen, nur sehr geringe Schwankungen auf, so dass auch eine geringere Wiederholungszahl genügt hätte.

LOKALE SUCHE

Die Untersuchung des lokalen Suchverfahrens ASPO, welche wie dessen Entwicklung auch im Rahmen einer Masterarbeit [63] stattfand, unterscheidet sich von den Untersuchungen der sonstigen Beiträge dieser Arbeit. In den Konfigurationen *C1* bis *C6*, welche der Evaluierung von SIA, SDA, CGSeq, der Voranalysen-Kombination und des Hybrids aus LGA und SMT-Solving dienen, ist jeweils die Erfüllung aller für die Überdeckungskriterien CC und DC relevanten CGs das Ziel. Für die Evaluierung von ASPO wurden hingegen 12 CGs ausgewählt, allesamt aus Modell *A*. Diese wurden durch Ausführung der Konfigurationen *C5* (LGA mit statischen Voranalysen) und *C7* (ASPO mit statischen Voranalysen) jeweils einzeln anvisiert, ebenfalls 30-mal. Die Konfiguration *C5* kam also in zweierlei Form zur Anwendung.

Die Auswahl von 12 CGs dient der Fokussierung des Vergleichs zwischen lokaler und globaler Suche. Bei den Ausführungen der Konfigurationen *C1* bis *C6* zur Evaluierung der anderen Beiträge dieser Arbeit wurde nämlich erkannt, dass die Testdatensuchen sich in der Regel nur mit einem kleinen Kreis an CGs wirklich auseinandersetzen müssen. Ein Großteil der CGs wird nämlich bereits durch die ersten, initial generierten Testdaten erreicht. Drei der bei Modell *A* als schwierig erreichbar erkannten CGs fließen daher in die Menge der zur Analyse von ASPO betrachteten 12 CGs ein. Zudem wurden nach folgenden Kriterien neun weitere CGs ausgewählt: unterschiedliche Tiefe im Modell, Nicht-/Vorhandensein Boolescher Signale im CG-Ausdruck und eine unterschiedliche Anzahl relevanter Modelleingänge. Auf die Charakteristika der einzelnen CGs wird bei der Analyse der Ergebnisse eingegangen.

Für die der Evaluierung von ASPO dienenden Ausführungen der Konfigurationen *C5* und *C7* gelten darüber hinaus folgende Einstellungen. Die Belegungen der Parameter des LGA entsprechen zum Großteil den in der Arbeit von Windisch gewählten Belegungen (siehe Tabelle 10 auf Seite 161 seiner Arbeit [132]). Folgende Ausnahmen gelten: Die Population besteht in jeder Iteration des LGA aus 20 Individuen (Testdaten). Die Anzahl der Iterationen ist zudem auf maximal 200 Iterationen begrenzt. Da der LGA unter diesen Randbedingungen insgesamt maximal 4000 Testdaten erzeugen kann, wird die Erzeugung des 4000. Testdatums auch für die ASPO als zusätzliches Abbruchkriterium verwendet. In der Modelleingangsspezifikation wird sowohl für den LGA als auch für ASPO

festgelegt, dass die zu erzeugenden Optimierungssequenzen zwischen 4 und 50 Segmente enthalten dürfen. Die genaue Anzahl wird bei der initialen Erzeugung von Testdaten jeweils zufallsbasiert bestimmt. Während der LGA die Segmentanzahl während seiner Suche allerdings variieren kann, ändert sich diese bei ASPO nicht (siehe Abschnitt 4.1.3).

GLOBALE SUCHE

Für den LGA, welcher in den Konfigurationen *C1* bis *C6* zur Evaluierung von SIA, SDA, CGSeq, der Voranalysen-Kombination und des Hybrids aus LGA und SMT-Solving eingesetzt wurde, gelten zum Großteil ebenfalls die Einstellungen, die Windisch für seine Fallstudien wählte. Die Populationsgröße wurde allerdings auch hier auf 20 Individuen gesetzt. Darüber hinaus wurden die (neben der Erfüllung des anvisierten SG) zusätzlichen Abbruchkriterien der Suche ein wenig strenger definiert. Insbesondere im Fall von Konfiguration *C1*, bei der keinerlei unterstützende Techniken zum Einsatz kommen, wären die Laufzeiten der Testdatensuchen ansonsten zu lang ausgefallen, um die Fallstudien dieser Arbeit insgesamt in angemessener Zeit durchführen zu können. Ein zusätzliches Abbruchkriterium besteht daher im Erreichen der 40. Suchiteration. Eine Suche wird zudem abgebrochen, wenn in acht aufeinanderfolgenden Iterationen keine Verbesserung des bis dato besten Fitnesswerts auftritt. Diese Einschränkungen gelten wohlgemerkt nicht für die Ausführungen von *C5*, welche der Evaluierung der lokalen Suche dienen.

6.2 ERGEBNISSE

Bei den wie zuvor beschrieben durchgeführten Experimenten wurden für die genannten Kriterien Daten gesammelt. Diese werden im Folgenden präsentiert. Über Vergleiche dieser Daten von verschiedenen Konfigurationen werden die Thesen anschließend beantwortet – und die einzelnen Beiträge dieser Arbeit somit bewertet.

6.2.1 MESSDATEN

Die gesammelten Daten werden gruppiert nach verschiedenen Aspekten vorgestellt. Für die Ausführungen der Konfigurationen *C1* bis *C6*, bei denen eine Betrachtung aller für die Überdeckungskriterien CC und DC relevanten CGs stattfand, wird zunächst die erzielte Modellüberdeckung thematisiert (Metriken *D1* und *D2*). Anschließend werden die verschiedenen Laufzeiten sowie die Anzahl der betrachteten Testdaten behandelt

(*D3* und *D4*). Die Ergebnisse für die restlichen Metriken (*D5*, *D6* und *D7*) folgen danach. Zum Schluss werden die Ergebnisse der Ausführungen von Konfiguration *C5* und *C7*, welche dem Vergleich von globaler und lokaler Suche dienen, präsentiert (Metriken *D4* und *D7*).

ÜBERDECKUNGSGRAD

Zu Beginn sei der Überdeckungsgrad betrachtet, welchen die Konfigurationen *C1* bis *C6* erzielen. Die Diagramme in Abbildung 34b (für Modell *A*) und in Abbildung 34c (für Modell *B*) stellen diesen dar. Der Überdeckungsgrad ergibt sich hierbei gemäß Definition 13 (siehe Seite 33) aus der Anzahl der erreichten CGs (*D2*) im Verhältnis zur CG-Gesamtanzahl, abzüglich der Anzahl der gegebenenfalls als unerreichbar identifizierten CGs (*D1*). Es ist zu beachten, dass die dargestellten Werte gemittelt sind, und zwar über die jeweils 30 Durchläufe einer Konfiguration.

Modell A. Die Konfiguration *C1* konnte für 556 der insgesamt 600 CGs erfüllende Testdaten finden. Da *C1* nicht in der Lage ist, unerreichbare CGs zu erkennen, ergibt sich ein Überdeckungsgrad von 92,6%. Die Aktivierung der SIA-Technik in *C2* führt dazu, dass in jedem der 30 Durchläufe 33 unerreichbare und 69 stets erreichte CGs identifiziert wurden. Über Testdatensuchen konnten neben den bereits als stets erreicht bekannten CGs weitere 487 CGs erfüllt werden. Der Überdeckungsgrad von *C2* beträgt daher 98%. Die zusätzliche Aktivierung der SDA-Technik in *C3* führte dazu, dass die Anzahl der CGs, für welche über Testdatensuchen erfüllende Testdaten gefunden werden konnten, um eines gestiegen ist. Dies resultiert in einem Überdeckungsgrad von 98,2%. Bei der Konfiguration *C4*, welche neben SIA die CGSeq-Technik, allerdings nicht die SDA-Technik nutzt, beträgt der Überdeckungsgrad wie schon bei *C2* daher auch nur 98%. Abgesehen davon, dass die CGSeq-Technik weitere 38 stets erreichte CGs identifizieren konnte und somit insgesamt 107 stets erreichte CGs bekannt sind, konnte bei Konfiguration *C4* keine weitere Auswirkung auf die erzielte Überdeckung festgestellt werden. In *C5* waren alle statischen Voranalysen aktiv. Wie schon bei Aktivierung der SDA-Technik in *C3* konnte auch *C5* einen Überdeckungsgrad von 98,2% erzielen. Dieser blieb auch in *C6* unverändert. Es ist allerdings zu erkennen, dass mit 379 ein großer Teil der erreichten CGs durch über SMT-Solving erzeugte Testdaten erfüllt wurden – Testdatensuchen überdeckten weitere 71 CGs.

Modell B. Die Ergebnisse bezüglich des erzielten Überdeckungsgrads fallen für Modell *B* recht ähnlich aus. Die Konfiguration *C1* konnte 1055 von insgesamt 1113 CGs überdecken (Überdeckungsgrad 94,7%). SIA identifizierte in *C2* 52 unerreichbare und 96 stets erreichte CGs. Die Testdatensuchen überdeckten weitere 959 CGs, so dass der Überdeckungsgrad im Fall von *C2* 99,4% beträgt. Der Einsatz der SDA-Technik in *C3* hatte im

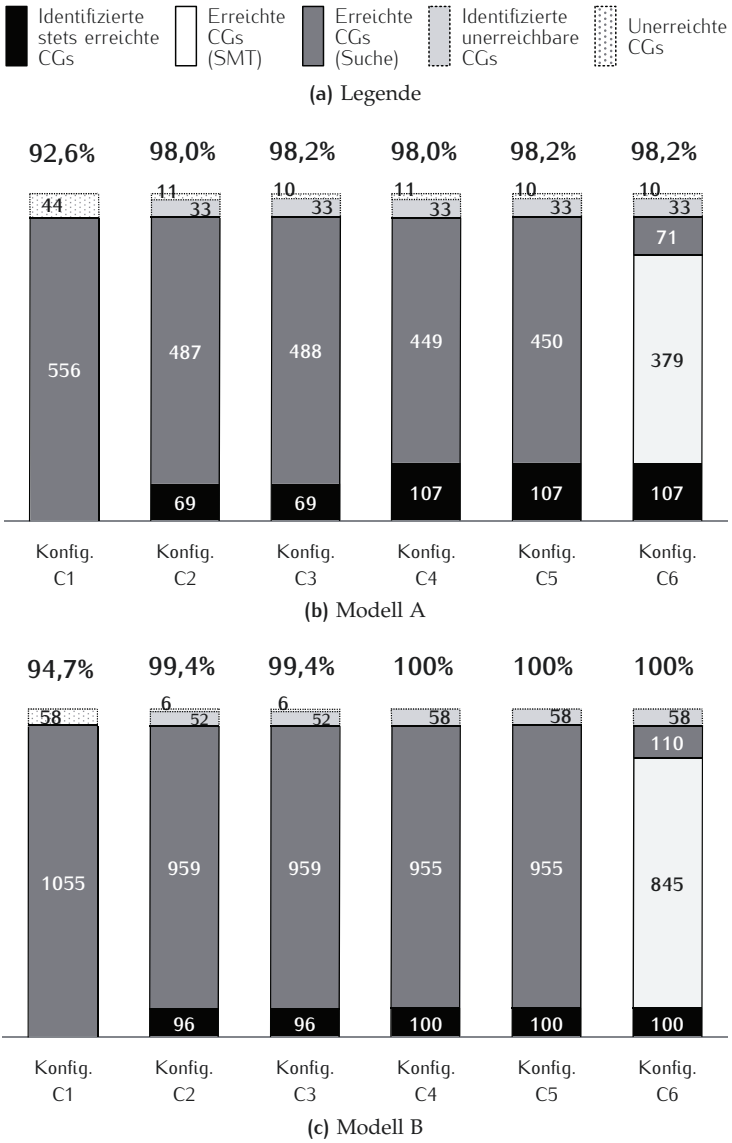
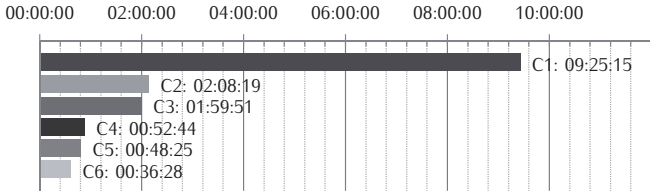


Abbildung 34: Effektivität der Konfigurationen C1 bis C6 im Vergleich (jeweils für die Modelle A und B): Dargestellt ist die Überdeckung der CGs (∅ 30 Durchläufe) sowie der sich hieraus ergebende Überdeckungsgrad.

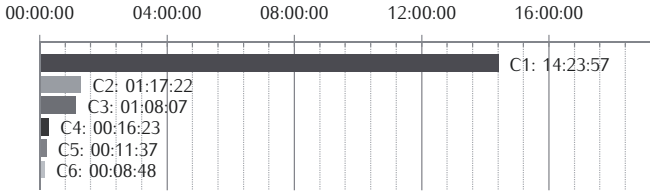
Vergleich hierzu keine Auswirkung auf die erzielte Überdeckung. Konfiguration *C4* identifizierte durch den Einsatz der CGSeq-Technik 4 weitere stets erreichte und 6 weitere unerreichbare CGs. Damit blieben keine CGs übrig, welche nicht durch Testdatensuchen erfüllt werden konnten und es ergibt sich ein Überdeckungsgrad von 100%. Der Einsatz aller Voranalysen in *C5* erzielte ebenfalls einen Überdeckungsgrad von 100%. Bei der hybriden Testdatengenerierung über Suche und SMT-Solving in Konfiguration *C6* wurde ebenfalls eine vollständige Überdeckung der erreichbaren CGs erzielt. Auch bei Modell *B* war hierbei das gegenüber den Testdatensuchen priorisierte SMT-Solving für einen Großteil der CG-Erfüllungen verantwortlich (845 CGs gegenüber 110 CGs).

AUFWAND

Als nächstes werden die Ergebnisse der Konfigurationen *C1* bis *C6* bezüglich des Aufwands der Testdatengenerierung vorgestellt. Als wesentliches Maß dient hierbei die Laufzeit. Diese ist als Summe der Metriken *D3d* (Laufzeit Testdatensuche) und *D3e* (Laufzeit symbolische Ausführung und SMT-Solving) in Abbildung 35 für die Modelle *A* und *B* dargestellt. Die in den Diagrammen enthaltenen Werte sind auch hier über die jeweils 30 Durchläufe gemittelt.



(a) Modell A



(b) Modell B

Abbildung 35: Effizienz der Konfigurationen C1 bis C6 im Vergleich (jeweils für die Modelle A und B): Laufzeit der Testdatengenerierung im Format *Stunden:Minuten:Sekunden* (∅ 30 Durchläufe)

Modell A. Die Testdatensuchen von Konfiguration *C1* benötigen insgesamt rund 9 h und 25 min, um die aus den CGs gebildeten singulären SGs abzuarbeiten. Der Ausschluss unerreichbarer und stets erreichter CGs – beziehungsweise der jeweils entsprechenden SGs – durch den Einsatz der SIA-Technik bewirkt in *C2*, dass die Laufzeit der Testdatensuchen nur noch rund 2 h und 8 min beträgt. In dieser Zeit nicht enthalten ist, dass SIA selbst im Schnitt rund 48 s benötigte (*D3a*). Die zusätzliche Aktivierung der SDA-Technik in *C3* reduziert die Laufzeit der Testdatensuchen auf rund 1 h 59 min. SDA selbst benötigte nur circa eine Sekunde (*D3b*). Eine im Vergleich zum Einsatz von SDA etwas größere Reduzierung der Laufzeit bewirkt der Einsatz der CGSeq-Technik. In Konfiguration *C4* dauern die Testdatensuchen nur noch rund 52 min. Der zeitliche Aufwand von CGSeq selbst fiel mit im Schnitt 15 s vergleichsweise gering aus (*D3c*). Bei Einsatz aller drei Voranalysen in Kombination kommen die Wechselwirkungen zwischen diesen Analysen zum Tragen. Im Falle von *C5* bewirkte dies eine weitere Reduzierung der Laufzeit auf rund 48 min. Die hybride Testdatengenerierung über Suche und SMT-Solving in *C6* dauerte im Vergleich hierzu sogar nur rund 36 min. Von dieser Laufzeit entfielen 56 s auf die symbolische Ausführung (inklusive SMT-Solving) und 35 min 32 s auf Testdatensuchen (*D3d* und *D3e*).

Modell B. Die Ergebnisse der Anwendung der Konfigurationen *C1* bis *C6* für Modell *B* fallen bezüglich der Aufwandseinsparungen ähnlich aus. Der Einsatz von SIA (Laufzeit 15 s, *D3a*) verringert die Laufzeit von über 14 h (*C1*) auf 1 h 17 min (*C2*). Ist zusätzlich SDA (Laufzeit 1 s, *D3b*) aktiv, beträgt die Laufzeit nur noch 1 h 8 min (*C3*). Durch den Einsatz von CGSeq (Laufzeit 20 s, *D3c*) sinkt die Laufzeit auf nur rund 16 min (*C4*) und bei Einsatz aller Voranalysen in Kombination beträgt diese sogar nur rund 11 min (*C5*). Wird zusätzlich noch das SMT-Solving verwendet (*C6*), erfolgt die gesamte Testdatengenerierung in etwas weniger als 9 min. Der Anteil von symbolischer Ausführung und SMT-Solving beträgt hierbei 1 min 12 s (*D3d*) – die Testdatensuchen dauern 7 min 36 s (*D3e*).

Um die anhand der Laufzeiten hinsichtlich des Aufwands gemachten Beobachtungen zu untermauern, wurde zusätzlich ausgewertet, wie viele Testdaten von den Konfigurationen jeweils generiert und betrachtet wurden (Metrik *D4*). Die hierbei protokollierten Anzahlen bestätigen im Falle beider Modelle die anhand der Laufzeiten gemachten Beobachtungen. Auf eine zusätzliche Auflistung oder Darstellung wird daher an dieser Stelle verzichtet.

SUCHZIELE, ERFOLGSQUOTE UND TESTSUITE

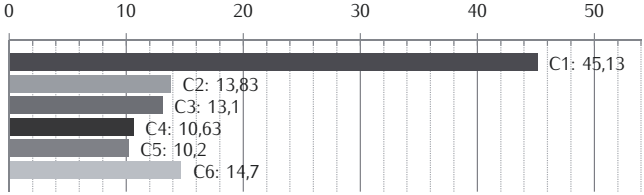
Um weitere Aspekte der CGSeq-Technik und auch der automatisierten Testdatengenerierung insgesamt bewerten zu können, wurden eine Reihe weiterer Messungen vorgenommen. So wurde bei Ausführung der einzelnen Konfigurationen festgehalten, wie viele Suchziele überhaupt

durch Testdatensuchen oder SMT-Solving anvisiert wurden (*D6*). Zudem wurde protokolliert, in wie vielen Fällen eine solche Anvisierung prozentual gesehen erfolgreich, das heißt mit der Erfüllung des jeweiligen SG, endet (*D7b*). Außerdem wurde registriert, wie umfangreich die erzeugten Testsuiten sind (*D5*). Die Ergebnisse dieser drei Messungen bei Ausführung der Konfigurationen *C1* bis *C6* sind in den Abbildungen 36, 37 und 38, jeweils für Modell *A* und *B*, dargestellt. Nachfolgend werden die in den Diagrammen enthaltenen Werte, welche über die je 30 Konfigurationsausführungen gemittelt sind, erläutert.

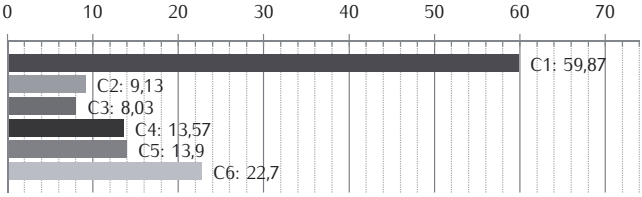
Suchziele. Im Fall von Modell *A* waren bei Konfiguration *C1* im Schnitt circa 45 SGs Gegenstand von Testdatensuchen. Durch den Ausschluss unerreichbarer SGs mittels SIA wurden in *C2* nur rund 14 SGs anvisiert. Bei Nutzung der SDA-Technik (*C3*) waren es nur unwesentlich weniger. Die Verwendung von CGSeq in *C4* und *C5* führte allerdings zu einer merklichen Verringerung dieser Anzahl auf rund 10 SGs. Da in *C6* SGs, welche SMT-Solving-geeignet sind, gegenüber SGs mit einer potenziell hohen Kollateralüberdeckung priorisiert werden, liegt die Anzahl anvisierter SGs mit rund 15 wieder etwas höher. Mit einer Ausnahme sind alle diese Beobachtungen auch im Fall von Modell *B* auszumachen. Die Ausnahme: Die Nutzung von CGSeq (beispielsweise in *C4*) führt bei Modell *B* zu einer höheren Anzahl anvisierter SGs. Die Analyse der Ergebnisse im nächsten Abschnitt erörtert, warum.

Erfolgsquote. Die Quote der anvisierten SGs, für welche mit Erfolg erfüllende Testdaten gefunden werden konnten, erhöht sich erwartungsgemäß, wenn unerreichbare SGs zuvor aussortiert werden. Im Fall von Modell *A* steigt die Erfolgsquote hierdurch von rund 3,9% (*C1*) auf rund 16,8% (*C2*), im Fall von Modell *B* von rund 3,1% auf rund 25%. Die Fokussierung der Testdatensuchen auf relevante Modelleingänge mittels SDA lässt die Erfolgsquoten in *C3* auf rund 20,5% (Modell *A*) und rund 32,1% (Modell *B*) ansteigen. Die Vorsortierung der SGs durch CGSeq bewirkte in den Fallstudien eine deutliche Erhöhung der Erfolgsquoten. Bei *C4* betrug diese rund 70,2% für Modell *A* und circa 98% für Modell *B*. Die gemeinsame Nutzung aller drei Voranalysen (*C5*) sowie die partielle Testdatengenerierung mittels SMT-Solving (*C6*) erhöhte diese Erfolgsquoten in den Fallstudien nochmals geringfügig.

Testsuite. Die Größe der Testsuite sank in den Fallstudien bei Einsatz der CGSeq-Technik (*C4* und *C5*), allerdings nur marginal. Im Fall von Modell *A* enthalten die durch die Konfigurationen *C1* bis *C3* – also diejenigen Konfigurationen, die CGSeq nicht einsetzen – erzeugten Testsuiten mit im Schnitt an die 7 Testdaten. Bei den Konfigurationen *C4* und *C5* enthalten die Testsuiten im Schnitt nur ein Testdatum weniger. Auffälliger ist im Vergleich hierzu, dass der Einsatz des SMT-Solving, beziehungsweise die Art des Einsatzes, in den Fallstudien zu einer recht deutlichen Vergrößerung der Testsuiten führte (*C6*). Für Modell *A* enthält eine Testsuite,

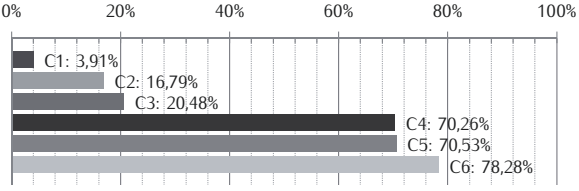


(a) Modell A

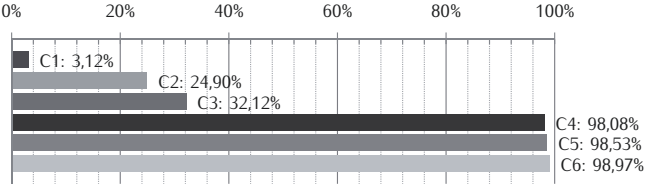


(b) Modell B

Abbildung 36: Anzahl der durch die Konfigurationen C1 bis C6 anvisierten SGs im Vergleich (∅ 30 Durchläufe, jeweils für die Modelle A und B)



(a) Modell A



(b) Modell B

Abbildung 37: Erfolgsquote der Konfigurationen C1 bis C6 für die anvisierten Suchziele im Vergleich (∅ 30 Durchläufe, jeweils für die Modelle A und B)

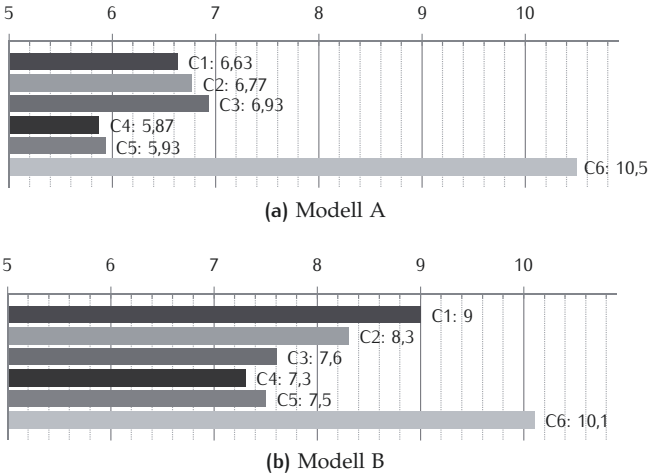


Abbildung 38: Größe der durch die Konfigurationen C1 bis C6 erzeugten Testsuiten im Vergleich (∅ 30 Durchläufe, jeweils für die Modelle A und B)

die aus einer Ausführung von Konfiguration *C6* hervorgegangen ist, im Schnitt 10-11 Testdaten. Alle diese Beobachtungen wurden auch bei Modell *B* gemacht. Die Verringerung der Testsuite-Größe bei Verwendung von CGSeq fällt hier allerdings noch geringer aus.

LOKALE SUCHE

Zur Evaluierung der lokalen Suche (LS) wurden, wie im Zuge des Versuchsaufbaus beschrieben, 12 ausgewählte CGs aus Modell *A* jeweils einzeln 30-mal mit der globalen Suche (GS), also dem LGA (*C5*), und 30-mal mit der LS (ASPO, *C7*) anvisiert. Hierbei wurde gemessen, in wie vielen der 30 Durchläufe jeweils erfüllende Testdaten gefunden werden konnten. Diese Erfolgsquote (*D7a*) je CG und Suchverfahren ist in Abbildung 39 dargestellt. Für 11 der 12 betrachteten CGs konnten beide Suchverfahren in mindestens einem der je 30 Durchläufe ein erfüllendes Testdatum finden. Bei CG 1 bis 9 war die GS stets erfolgreich, im Fall von CG 11 zudem in 93% der Fälle. Die LS konnte nur für 3 der 12 CGs (CG 1 bis 3) in jedem Durchlauf erfüllende Testdaten finden – im Falle eines weiteren CG war die LS zumindest bei 67% der Durchläufe erfolgreich (CG 5). Nur im Fall eines einzigen CG, nämlich CG 10, konnten beide Suchverfahren in keinem Durchlauf passende Testdaten finden.

Zum Vergleich des Aufwands beider Verfahren wird die Anzahl der bei einer Suche betrachteten Testdaten (*D4*) herangezogen. Diese ist für die

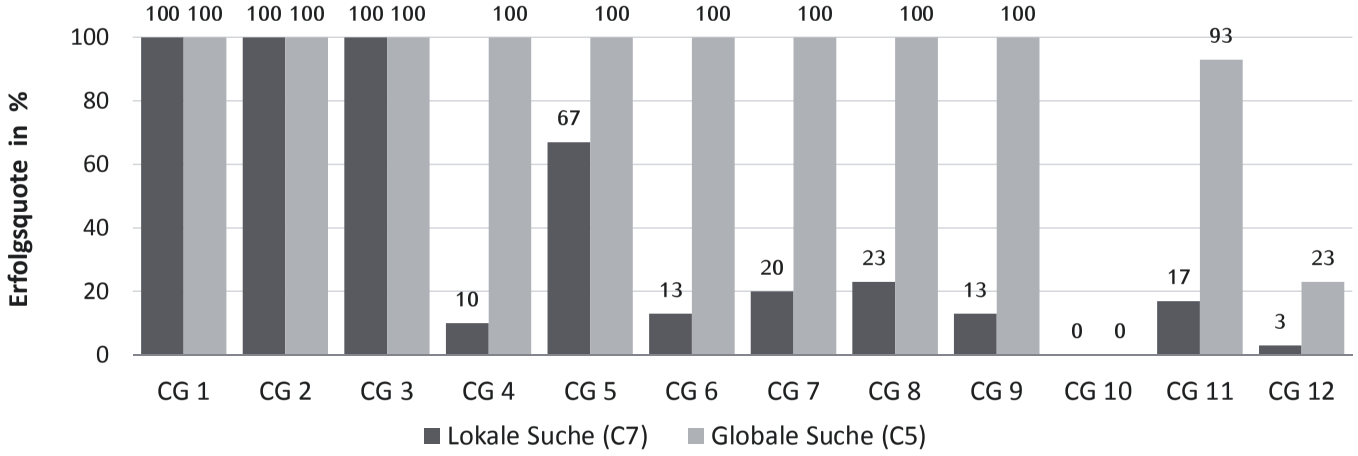


Abbildung 39: Vergleich der lokalen Suche (Konfiguration C7) mit der globalen Suche (Konfiguration C5) bezüglich ihrer Effektivität anhand ausgewählter CGs: Wie häufig wurde jedes CG in 30 Durchläufen erreicht?

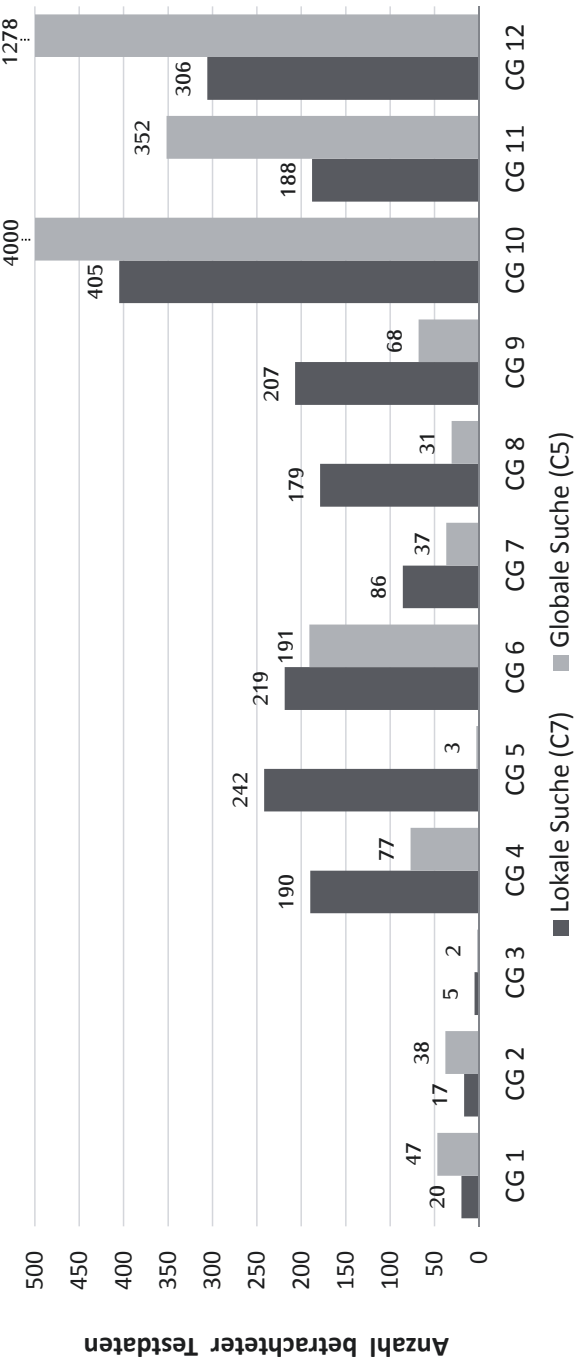


Abbildung 40: Vergleich der lokalen Suche (Konfiguration C7) mit der globalen Suche (Konfiguration C5) bezüglich ihrer Effizienz anhand ausgewählter CGs: Wie viele Testdaten wurden bei einer Suche für ein CG (∅ 30 Durchläufe) untersucht?

12 CGs und beide Suchverfahren in Abbildung 40 veranschaulicht. Die dort dargestellten Werte sind ebenfalls über die jeweils 30 Durchläufe gemittelt. Für zwei der drei CGs, welche sowohl die LS als auch die GS in jedem der Durchläufe erreichte (CG 1 bis 3), konnte die LS mit geringerem Aufwand passende Testdaten finden (CG 1 und 2). Im Falle der CGs 4 bis 9, welche die GS stets und die LS vereinzelt erfüllen konnte, betrachtete die GS stets weniger Testdaten als die LS. Bei CG 10 waren LS und GS beide in allen 30 Durchläufen erfolglos. Die GS stieß daher an ihr Abbruchkriterium von 4000 generierten Testdaten. Die LS endete bereits nach 405 betrachteten Testdaten, da einzelne Modifikationen aller Testdatenparameter zu keinem besseren Testdatum mehr führten (siehe Abschnitt 4.1.3). Dieses Abbruchkriterium ist auch für die relativ geringe Anzahl betrachteter Testdaten im Falle der Anwendung der LS für CG 11 und 12 verantwortlich. Im Falle dieser beiden CGs war die LS auch eher selten erfolgreich. Die GS war bei CG 11 und 12 häufiger erfolgreich, die Suchvorgänge waren jedoch mit einem relativ hohen Aufwand verbunden.

6.2.2 ANALYSE DER STATISCHEN VORANALYSEN

Die Ergebnisse werden im Folgenden vertiefend betrachtet und analysiert, vor allem hinsichtlich der aufgestellten Thesen. Dieser Abschnitt behandelt hierbei die statischen Voranalysen SIA, SDA und CGSeq.

6.2.2.1 *Analyse der Überdeckungsziel-Erreichbarkeitsprüfung*

These T1a. *Existieren in einem Modell unerreichbare CGs, so steigert der Einsatz von SIA die Effizienz der Testdatengenerierungsvorgänge insgesamt.*

In den zwei verwendeten Modellen existieren unerreichbare CGs (Metrik *D1*). Dies wurde durch SIA festgestellt und im Anschluss an die Fallstudien zur Bestätigung durch eine manuelle Analyse der Modelle nachvollzogen. Die Anzahl erreichter CGs (*D2*) ist in den Fallbeispielen bei Nutzung von SIA genauso hoch wie ohne SIA. Die Effektivität der gesamten Testdatengenerierung wurde durch SIA also nicht negativ beeinflusst. Der Aufwand der Testdatensuchen war durch den Ausschluss unerreichbarer und stets erreichter CGs allerdings deutlich geringer. Bei Modell *A* verringerte SIA die Laufzeit (*D3d*) um 77% und bei Modell *B* sogar um 91%. Die Fallbeispiele untermauern somit die Gültigkeit der These.

These T1b. *Der zeitliche Aufwand von SIA ist geringer als der Zeitgewinn, den der Einsatz von SIA für die Testdatengenerierungsvorgänge insgesamt bewirkt.*

Bei Modell *A* betrug die Laufzeit von SIA (*D3a*) 48 s, bei Modell *B* lediglich 15 s. Die durch den Einsatz von SIA eingesparte Laufzeit bei den Testdatensuchen beläuft sich hingegen auf über 7 h (Modell *A*) beziehungsweise auf über 13 h (Modell *B*). Diese Zeitgewinne ergeben sich durch Subtraktion der Messungen für die Metrik *D3d* aus den Konfigurationen *C1* und *C2*. Im Fall beider Modelle war die Laufzeit von SIA, verglichen mit dem für die Suchen bewirkten Zeitgewinn, verschwindend gering.

These T1c. *Der durch Testdatensuchen erzielte Überdeckungsgrad ist bei Einsatz von SIA mindestens genauso hoch wie ohne diesen.*

Ohne Einsatz von SIA (*C1*) erzielt die Testdatensuche einen Überdeckungsgrad von 92,6% (Modell *A*) beziehungsweise 94,7% (Modell *B*). Mit SIA (*C2*) beträgt er 98% beziehungsweise 99,4%. Es wurden allerdings keine zusätzlichen CGs erreicht (*D2*) – die Steigerung ergibt sich aus dem Ausschluss der identifizierten unerreichbaren CGs (*D1*) bei der Berechnung des Überdeckungsgrads. Würden diese nicht herausgerechnet werden, wären die Überdeckungsgrade von *C1* und *C2* im Fall beider Modelle identisch. Dies erfüllt allerdings ebenso die aufgestellte These.

Ergänzendes. Für Modell *B* konnten durch SIA und ergänzend durch CGSeq alle unerreichbaren CGs identifiziert werden. Im Fall von Modell *A* blieben bei allen Konfigurationen mindestens 10 CGs unerreicht. Im Anschluss an die Durchführung der Fallstudien wurde daher untersucht, warum für diese CGs keine Testdaten gefunden werden konnten. Es stellte sich heraus, dass 8 der 10 unerreichten CGs unerreichbar sind. Diese wurden durch SIA jedoch nicht erkannt, da die Block-spezifischen Wertebereichsberechnungen von SIA innerhalb von TASMO nicht für alle in Modell *A* vorkommenden Blocktypen implementiert wurden – beispielsweise für *Lookup-Table*-Blöcke. Bei einer spezifischen Unterstützung aller relevanten Blocktypen hätte SIA also 8 weitere unerreichbare CGs erkannt und die Laufzeit der Testdatengenerierungsvorgänge wäre somit nochmals deutlich geringer ausgefallen.

6.2.2.2 *Analyse der Modelleingang-Abhängigkeitsermittlung*

These T2a. *Der Einsatz von SDA steigert die Effizienz der Testdatengenerierungsvorgänge insgesamt.*

Die Anzahl erreichter CGs (*D2*) ist bei Nutzung von SDA im Fall von Modell *B* genauso hoch wie ohne SDA (Vergleich *C3* mit *C2*) und im Fall von Modell *A* sogar um im Schnitt rund ein CG höher. Die Effektivität der Testdatengenerierung nimmt durch die Einführung der SDA-Technik also nicht ab, sondern kann durch sie unter Umständen gesteigert werden. Durch die Unterstützung der Suchen über eine Fokussierung auf

relevante Modelleingänge reduziert sich zudem die Laufzeit der Testdatensuchen (*D3d*) – wie auch die Anzahl der betrachteten Testdaten (*D4*) – geringfügig. Für Modell *A* reduzierte sich die Laufzeit um 7% und für Modell *B* um 12%. Die effizienzsteigernde Wirkung der SDA-Technik konnten die Fallstudien demnach bestätigen.

These T2b. *Der zeitliche Aufwand von SDA ist geringer als der Zeitgewinn, den der Einsatz von SDA für die Testdatengenerierungsvorgänge insgesamt bewirkt.*

Die Ausführung der SDA-Technik dauerte bei beiden Modellen stets rund eine Sekunde (*D3b*). Da durch SDA im Fall von Modell *A* im Schnitt rund 8 min und im Fall von Modell *B* im Schnitt rund 9 min an Laufzeit für die Testdatensuchen (*D3d*) eingespart werden konnte, deuten die Fallstudien auf die Gültigkeit dieser These hin.

These T2c. *Der durch Testdatensuchen erzielte Überdeckungsgrad ist bei Einsatz von SDA mindestens genauso hoch wie ohne diesen.*

Während der Überdeckungsgrad ohne Einsatz von SDA (*C2*) für Modell *A* bei 98% lag, konnte er durch die Einbindung von SDA (*C3*) mit 98,2% geringfügig erhöht werden. Bei Modell *B* änderte sich der Überdeckungsgrad nicht, er blieb bei 99,4%. Für die betrachteten Fallbeispiele erweist sich diese These demnach als gültig.

Ergänzendes. Für alle CGs, welche durch SIA nicht als unerreichbar oder stets erreicht identifiziert wurden und die daher nicht vor Stattfinden von SDA aussortiert werden, ermittelte SDA, dass im Schnitt 35% der Eingänge von Modell *A*, beziehungsweise 41% der Eingänge von Modell *B*, zur CG-Erfüllung relevant sind. Beschränkt man die Menge der hierbei betrachteten CGs auf diejenigen CGs, welche in Konfiguration *C3* als Teil eines SG auch tatsächlich anvisiert wurden, so handelt es sich um im Schnitt 54% (Modell *A*), beziehungsweise 75% (Modell *B*) relevante Eingänge. Diese Werte verdeutlichen, wie stark die durch die SDA-Technik erzielte Suchraumreduktion für Modelle aus der Praxis sein kann. Die Auswirkung des SDA-Einsatzes spiegelt sich im Übrigen auch in der Erfolgsquote bezüglich der Erfüllung anvisierter SGs (*D7*) wieder. Im Falle der Aktivierung von SDA stieg in den Fallstudien beider Modelle die Erfolgsquote (Vergleich von *C3* mit *C2*). Dies deutet darauf hin, dass die Suchraumreduktion von SDA eine zielgerichtetere Suche zur Folge haben kann. Durch die Belegung von irrelevanten Modelleingängen mit zufällig generierten Signalen scheint sich zudem die von den Suchvorgängen ausgehende Kollateralüberdeckung leicht zu erhöhen. Dies bedeutet, mehr SGs/CGs wurden bei der Anvisierung anderer SGs/CGs zufällig mit erreicht. Dies geht daraus hervor, dass im Falle beider Modelle in *C3*, im Vergleich zu *C2*, weniger SGs anvisiert werden mussten.

6.2.2.3 *Analyse der Suchzielpriorisierung*

These T3a. *Der Einsatz von CGSeq steigert die Effizienz der Testdatengenerierungsvorgänge insgesamt.*

Im Falle beider Modelle erreichten die Testdatensuchen nach Durchführung von CGSeq (*C4*) eine ebenso hohe Anzahl an CGs (*D2*) wie ohne eine vorherige Durchführung von CGSeq (*C2*). Die Nutzung von CGSeq wirkt sich also nicht negativ auf die Effektivität der Testdatengenerierung aus. Die Effizienz dieser steigt allerdings deutlich. Die Laufzeit der Testdatensuchen (*D3d*) in Konfiguration *C4* verkürzte sich im Vergleich zu *C2* um circa 1 h 15 min (Modell *A*), beziehungsweise circa 1 h (Modell *B*). Dies entspricht einer Reduktion der Laufzeit um 59%, beziehungsweise 79%. Die höhere Laufzeitverringerung bei Modell *B* erklärt sich dadurch, dass die Erreichbarkeitsprüfung von CGSeq, welche SIA bezüglich der Erkennung unerreichbarer sowie stets erreichter CGs ergänzt, für Modell *B* 6 weitere unerreichbare CGs erkannte. Es bleibt festzustellen, dass die These durch beide Fallbeispiele nachdrücklich gestützt wird.

These T3b. *Der zeitliche Aufwand von CGSeq ist geringer als der Zeitgewinn, den der Einsatz von CGSeq für den gesamten Testdatengenerierungsvorgang bewirkt.*

Den verkürzten Laufzeiten der Testdatensuchen (*D3d*) bei Nutzung von CGSeq steht eine vergleichsweise geringe Ausführungsdauer von CGSeq selbst gegenüber (*D3c*). Während der Zeitgewinn, wie bei der Analyse der vorherigen These herausgestellt, bei beiden Modellen bei mindestens einer Stunde liegt, benötigt CGSeq vorab im Schnitt lediglich 16 s (Modell *A*) beziehungsweise 20 s (Modell *B*). Da sich der Aufwand von CGSeq also in Grenzen hält, ist der Einsatz dieser Technik unter diesem Gesichtspunkt nicht bedenklich.

These T3c. *Der durch Testdatensuchen erzielte Überdeckungsgrad ist bei Einsatz von CGSeq mindestens genauso hoch wie ohne diesen.*

Die Aktivierung von CGSeq hatte keine Auswirkung auf die Anzahl erreichter CGs (*D2*), sowohl bei Modell *A* als auch bei Modell *B*. Wie zuvor festgestellt, erkannte CGSeq bei Modell *B* allerdings 6 weitere unerreichbare CGs (*D1*). Während der Überdeckungsgrad also bei Modell *A* unabhängig vom Einsatz der CGSeq-Technik gleichbleibend bei 98% liegt, steigert der Einsatz von CGSeq den Überdeckungsgrad für Modell *B* von 99,4% auf 100%. In beiden Fällen gilt jedoch die These.

These T3d. *Die Bildung einer Abarbeitungsreihenfolge der SGs durch CGSeq führt bei den Testdatensuchen zu einer höheren Kollateralüberdeckung oder einer erhöhten Anzahl erfolgreich abgeschlossener Suchen.*

Die durch CGSeq durchgeführte Priorisierung der SGs erfolgt unter Zuhilfenahme verschiedener Metriken. Diese adressieren sowohl die

Schwierigkeit der Testdatengenerierung für ein SG als auch die Höhe der Kollateralüberdeckung, welche das Erreichen eines SG mit sich bringt. Daher war vom Einsatz von CGSeq in den Fallstudien eine Auswirkung auf folgende zwei Fallstudienmetriken zu erwarten: die Anzahl anvisierter SGs (*D6*) sowie die Erfolgsquote aller durchgeführten Testdatensuchen (*D7*). Werden leichter zu erreichende SGs den schwer zu erreichenden SGs vorgezogen, so sollte die Erfolgsquote steigen. Ist die Kollateralüberdeckung anvisierter SGs höher als diejenige von SGs, welche ohne eine vorherige Durchführung von CGSeq durch Testdatensuchen anvisiert werden, so sollten insgesamt weniger SGs überhaupt anvisiert werden.

Die Frage, welche dieser beiden Einflüsse bei der Suchzielpriorisierung stärker zum Tragen kommt, ist stark von der Beschaffenheit des betrachteten Modells und der sich aus diesem ableitenden CGs abhängig. Im Fall von Modell *B* existieren im Vergleich zu Modell *A* anteilig gesehen deutlich mehr CGs, welche Bedingungen über Signale mit einem Booleschen Wertebereich beschreiben. Auch wenn vom Erreichen dieser unter Umständen eine hohe Kollateralüberdeckung zu erwarten ist, so stellt CGSeq die dazugehörigen SGs in der Abarbeitungsreihenfolge hinten an. Die zu erwartende Schwierigkeit des Erreichens eines SGs hat bei Modell *B* also, so ist zu vermuten, einen größeren Einfluss bei der Reihenfolgenbildung als die abgeschätzte Kollateralüberdeckung.

Die Fallstudien bestätigen diese Vermutung. Sowohl bei Modell *A* als auch bei Modell *B* steigt zwar die Erfolgsquote der Testdatensuchen deutlich – von circa 17% auf rund 70% (Modell *A*), beziehungsweise von circa 25% auf rund 98% (Modell *B*) –, jedoch verringert sich die Anzahl anvisierter SGs nur bei Modell *A*. Während bei Modell *A* ohne Nutzung von CGSeq (*C2*) im Schnitt 13,8 SGs anvisiert werden, sind es mit CGSeq (*C4*) durchschnittlich 10,6 SGs. Bei Modell *B* steigt dieser Wert hingegen von 9,1 auf 13,6. Beide Fälle entsprechen der aufgestellten These und es wird deutlich, wie CGSeq bei der SG-Reihenfolgenbildung mehrere Aspekte vereint – zum Vorteil der Effizienz der Testdatengenerierung.

These T3e. *Der Einsatz von CGSeq bewirkt, dass die durch Testdatensuchen erzeugte Testsuite weniger Testdaten enthält.*

Die Größe der für einen Strukturtest generierten Testsuiten ist in der Praxis von Relevanz, da meist kein Testorakel existiert, welches die funktionale Korrektheit des Testobjekts für die erzeugten Testdaten automatisch prüft oder bewertet. Da dieser Umstand eine manuelle Durchsicht der generierten Testdaten notwendig macht, steht und fällt der Aufwand dieses manuellen, der automatischen Testdatengenerierung nachgeordneten Vorgangs mit der Größe der Testsuite.

Sowohl bei Modell *A* als auch bei Modell *B* enthält die durch Konfiguration *C4* erzeugte Testsuite weniger Testdaten als die von *C2* erzeugte

Testsuite. Auch wenn die Unterschiede jeweils nicht sonderlich groß sind, stützen diese Ergebnisse die durch die These adressierte Annahme.

6.2.2.4 *Analyse der Voranalysen-Kombination*

These T4a. *Der gemeinsame Einsatz von SIA, SDA und CGSeq bewirkt, im Vergleich zum Einsatz nur eines Teils dieser Techniken, eine zusätzliche Effizienzsteigerung der Testdatengenerierungsvorgänge insgesamt.*

In den Konfigurationen *C2* (SIA), *C3* (SIA und SDA) und *C4* (SIA und CGSeq) wurden die einzelnen Voranalyse-Techniken eingesetzt. Im Vergleich mit *C1*, wo keine der Voranalysen zum Einsatz kam, erzielte jede dieser drei Konfigurationen eine Effizienzsteigerung - bei gleichbleibender oder verbesserter Effektivität. Da die drei Voranalyse-Techniken untereinander auf ihre Ergebnisse zurückgreifen (siehe Abschnitt 3.4), sofern diese verfügbar sind, ist zu vermuten, dass die gemeinsame Nutzung aller drei Techniken in Konfiguration *C5* die effizienteste Konfiguration aus *C1* bis *C5* darstellt. Die Fallstudien bestätigen diese These. Sowohl bei Modell *A* als auch Modell *B* entsprechen die Anzahl der durch *C5* erreichten CGs (Metrik *D2*) und die Anzahl der identifizierten unerreichbaren CGs (*D1*) den diesbezüglich jeweils höchsten Werten aus *C1* bis *C4*. Während die gemeinsame Nutzung von SIA, SDA und CGSeq keinen negativen Einfluss auf die Effektivität zeigt, reduziert sich die Laufzeit der durchgeführten Testdatensuchen (*D3d*) im Vergleich zur diesbezüglich zuvor besten Konfiguration *C4* geringfügig – und zwar bei beiden Modellen um rund 4 min.

These T4b. *Der zeitliche Aufwand von SIA, SDA und CGSeq insgesamt ist geringer als der Zeitgewinn, den der Einsatz dieser Techniken für die Testdatengenerierungsvorgänge insgesamt bewirkt.*

Insgesamt, das heißt bei gemeinsamer Nutzung (C5), reduzierten die Voranalysen die Laufzeit der Testdatensuchen (*D3d*) im Vergleich zu *C1* um über 8 h (Modell *A*) beziehungsweise um über 14 h (Modell *B*). Rechnet man die Laufzeit der drei Analysen zusammen (*D3a*, *D3b*, *D3c*), so ist deren Aufwand mit 1 min 5 s (Modell *A*) beziehungsweise 36 s (Modell *B*) zu bemessen. Gegenüber der erzielten Laufzeiteinsparung fällt dieser Zusatzaufwand nicht ins Gewicht. Der höhere zeitliche Aufwand der Analysen für Modell *A* gegenüber denen für Modell *B* ergibt sich im Übrigen hauptsächlich aus der höheren Zahl von Schleifen (Rückkopplungen) in Modell *A*. Diese machen insbesondere die Wertebereichsanalyse durch SIA etwas aufwändiger.

6.2.3 ANALYSE DES SMT-SOLVER-EINSATZES

These T5a. *Die Kriterien-basierte Testdatengenerierung mittels Suche und symbolischer Ausführung steigert die Effizienz der Testdatengenerierungsvorgänge insgesamt.*

In Konfiguration *C6* wurden für ein anvisiertes SG bevorzugt Testdaten mittels symbolischer Ausführung und SMT-Solving erzeugt, sofern die aufgestellten Einsatzkriterien dies zuließen. Andernfalls fand eine Suche durch den LGA statt. Die von CGSeq bestimmte SG-Abarbeitungsreihenfolge zog zudem SMT-Solving-geeignete SGs nach vorne. Dieser Konfiguration stand die Konfiguration *C5* gegenüber, bei der alle SG-Anvisierungen mittels Testdatensuchen stattfanden. Insgesamt wurden durch *C6* im Schnitt genauso viele CGs erreicht (Metrik *D2*) wie durch *C5*. Die Hybridisierung der Suche mit einer symbolischen Ausführung hatte im Falle beider Modelle also keine negativen Folgen für die Effektivität der Testdatengenerierungsvorgänge insgesamt. Die Laufzeit dieser (Summe aus *D3d* und *D3e*) konnte allerdings gesenkt werden. Bei Modell *A* fiel die Laufzeit im Schnitt um rund 12 min geringer aus und bei Modell *B* um rund 3 min. Die Fallstudien stützen daher die These.

These T5b. *Der durch den Hybrid aus Suche und symbolischer Ausführung erzielte Überdeckungsgrad ist mindestens genauso hoch wie der durch ausschließliche Suchen erreichte.*

Wie erwähnt erreichte die Konfiguration *C6* bei beiden Modellen, im Vergleich zu *C5*, eine gleich hohe Anzahl von CGs (*D2*). Die Anzahl der jeweils durch SIA und CGSeq identifizierten CGs ist ebenfalls identisch. Daher ist auch der jeweils von *C5* und *C6* erzielte Überdeckungsgrad gleich – 98,2% für Modell *A* und 100% für Modell *B*. Die Fallbeispiele bestätigen die These somit.

Ergänzendes. Im Zuge der Präsentation der protokollierten Daten wurde dargelegt, dass die Testdatengenerierungen mittels symbolischer Ausführung und SMT-Solving im Vergleich zu den Testdatensuchen nur einen geringen Anteil der gesamten Laufzeit bei *C6* einnahmen. Hierzu ergänzend sei erwähnt, dass im Fall von Modell *A* rund 37% der SG-Anvisierungen in *C6* durch symbolische Ausführung und SMT-Solving stattfanden – die übrigen circa 63% wurden mittels Testdatensuchen anvisiert. Bei Modell *B* waren es rund 42% durch symbolische Ausführung/SMT-Solving und circa 58% durch Testdatensuchen. Diese Zahlen liefern einen Eindruck, wie groß der Anteil der SGs ist, für welche das SMT-Solving überhaupt eingesetzt wurde, und wie stark sich die SMT-Solving-Einsatzkriterien innerhalb dieser Hybridisierung auswirken.

Bei den Ausführungen der Konfiguration *C6* fiel zudem auf, das sowohl die Anzahl der anvisierten SGs (*D6*) als auch die Größe der erzeugten Testsuiten (*D5*) durch die Kombination der Suche mit dem SMT-Solving

höher ausfallen. Beide diese Begebenheiten sind Folge der Priorisierung SMT-Solving-geeigneter SGs in der Abarbeitungsreihenfolge. Das Kriterium der Kollateralüberdeckung eines SG rückt durch diese Priorisierung etwas weiter in den Hintergrund. Da die SMT-Solving-geeigneten SGs im Schnitt also eine geringere Kollateralüberdeckung auslösen, müssen mehr SGs anvisiert werden und im Erfolgsfall daher auch mehr Testdaten in die Testsuite aufgenommen werden. Um diesem Nebeneffekt entgegenzuwirken, könnten Techniken zur Testsuite-Minimierung helfen.

6.2.4 ANALYSE DES LOKALEN SUCHVERFAHRENS

These T6. *Das lokale Suchverfahren ASPO ist für zumindest einen Teil der CGs in der Lage, die jeweils gesuchten Testdaten effizienter zu finden als das globale Suchverfahren LGA.*

Die Fähigkeiten und die Leistung des lokalen Suchverfahrens ASPO wurde durch Ausführungen der Konfigurationen *C6* (lokale Testdatensuche mit ASPO) und *C5* (globale Testdatensuche mit LGA) für 12 ausgewählte CGs aus Modell *A* untersucht. Wie bei der Vorstellung der Ergebnisse dargelegt, gab es hierbei drei CGs, welche sowohl von *C5* als auch *C6* in jeder der 30 Ausführungen erfüllt werden konnten. Bei zwei dieser drei CGs gelangte ASPO effizienter zu erfüllenden Testdaten als LGA – die lokale Suche betrachtete hierbei höchstens nur halb so viele Testdaten wie die globale Suche.

Ergänzendes. Betrachtet man die Charakteristik dieser drei CGs, so fällt auf, dass sich die an ihnen beteiligten Signale innerhalb von Modell *A* relativ nah an den Eingängen befinden. Zudem hängt ihre Erfüllung jeweils auch nur von wenigen dieser Eingänge ab. Dies bedeutet, der Suchraum ist vergleichsweise klein und die Fitnesslandschaft weist womöglich wenige Eigenschaften auf, die für eine lokale Suche hinderlich sind – beispielsweise Plateaus oder lokale Optima.

Bei den übrigen neun der insgesamt 12 CGs konnte das lokale Suchverfahren ASPO nicht überzeugen. ASPO fand nur in wenigen Fällen erfüllende Testdaten und konnte, auf alle 30 Ausführungen bezogen, auch hinsichtlich der Effizienz den LGA nicht übertrumpfen. In den Einzelfällen, wo die lokale Suche mit Erfolg erfüllende Testdaten fand, war die Anzahl der insgesamt betrachteten Testdaten jedoch oftmals geringer. Dies zeigt wohlgemerkt, dass die lokale Suche durch ihre systematische Vorgehensweise effizienter als eine globale Suche sein kann. Die neun angesprochen CGs sind tiefer innerhalb des Modells anzusiedeln als die drei zuvor behandelten CGs. Ihre Erfüllung ist zumeist auch von mehr Modelleingängen abhängig. Die globale Suche war in den Fallstudien in der Lage, mit der größeren Komplexität der Suchräume und Fitnesslandschaften dieser CGs umzugehen.

Einzig bei drei dieser neun CGs schaffte es auch der LGA nicht immer, erfüllende Testdaten zu finden. Bei diesen für eine Testdatensuche recht herausfordernden CGs muss der lokalen Suche allerdings zugutegehalten werden, dass sie eine erfolglose Suche deutlich frühzeitiger beendete als es die globale Suche tat. Grund hierfür ist das Vorgehen von ASPO: Sobald die individuelle Änderung eines jeden Parameters zu keinem besseren Testdatum mehr führt, wird die Suche beendet. Die globale Suche geht an dieser Stelle weniger systematisch vor und kannte nur den Abbruch bei Erfolg oder bei Erreichen einer maximalen Zahl an Optimierungsiterationen. Würde allerdings zusätzlich das ansonsten in den Fallstudien dieser Arbeit verwendete Abbruchkriterium der Fitness-Stagnation Anwendung finden, so wäre auch diese Schwäche der globalen Suche zumindest gelindert.

Eine weitergehende Analyse der Fallstudienergebnisse zeigte, dass neben der Suchraum- und Fitnesslandschaftskomplexität ein weiterer Faktor die Überlegenheit der globalen Suche begünstigte. Da die globale Suche grundsätzlich mit einer Vielzahl initial generierter Testdaten startet, ist die Wahrscheinlichkeit im Vergleich zur lokalen Suche deutlich höher, dass das jeweils betrachtete CG bereits zufällig durch eines dieser Testdaten erfüllt wird. Die lokale Suche hingegen startet mit nur einem initialen Testdatum. Erfüllt dieses das CG nicht, so bewegt sich die Suche mit hohem Aufwand Stück für Stück durch den Suchraum, um – falls überhaupt möglich – zu einem erfüllenden Testdatum zu gelangen, welches die globale Suche möglicherweise schon in ihrer ersten Iteration findet.

6.3 BEWERTUNG

Das hybride Testverfahren, bestehend aus den statischen Voranalysen SIA, SDA und CGSeq, dem globalen Suchverfahren LGA und der symbolischen Ausführung über SMT-Solving, konnte die Gültigkeit der Arbeitsthese in den durchgeführten Fallstudien unterstreichen. Diese lautet:

Die Effizienz des Ansatzes von Windisch zur suchbasierten Testdatengenerierung für SL-Modelle lässt sich durch eine Hybridisierung derselben mit geeigneten statischen oder dynamischen Techniken erhöhen.

Die Auswertung der einzelnen Thesen, welche für die Bausteine des hybriden Testverfahrens aufgestellt wurden, bestätigt diese Annahme. So wirkt sich das Hinzufügen der zusätzlichen Techniken SIA, SDA, CGSeq und SMT-Solving zum Ansatz der suchbasierten Testdatengenerierung von Windisch zum einen positiv auf den erreichten Überdeckungsgrad auf. Zum Anderen sinkt der Aufwand der Testdatengenerierung im Rahmen eines Strukturtests drastisch. Dies ließ sich in den Fallstudien nicht nur an den gemessenen Laufzeiten festmachen, sondern auch an der

Zahl der Testdaten, welche betrachtet werden mussten, um zu der letzten Endes gesuchten Testsuite zu gelangen: Ohne die zusätzlichen Techniken (Konfiguration *C1*) wurden im Schnitt 6094 (Modell *A*) und 8126 (Modell *B*) Testdaten auf dem Weg zu den in die Testsuiten aufgenommenen Testdaten „besucht" – mit den Erweiterungen dieser Arbeit (*C6*) waren es im Schnitt nur 485 (Modell *A*) und 91 (Modell *B*) Testdaten. Die Techniken SIA und CGSeq sorgten hierbei für die höchsten Effizienzsteigerungen.

Die Größe der erzeugten Testsuiten ist darüber hinaus als positiv zu bewerten. Teil eines Strukturtests ist es nämlich auch, die korrekte Funktionalität des Testobjekts für die aus ihm abgeleiteten Testdaten sicherzustellen. Werden Testdaten automatisch erzeugt und existiert kein Testorakel, welches die Frage der korrekten Funktionalität automatisiert beantwortet, so muss das Testobjekt-Verhalten für die einzelnen Testdaten der generierten Testsuite manuell untersucht werden. Je kleiner die Testsuite, desto weniger aufwändig ist dies. Die von dem hybriden Testverfahren in den Fallstudien erzeugten Testsuiten sind von diesbezüglich praktikabler Größe. Dennoch sind Bemühungen hinsichtlich einer Minimierung der Testsuite-Größen und einer automatisierten Verwertung der generierten Testdaten (Testorakel) natürlich wünschenswert.

Neben dem hybriden Testverfahren untersuchten die Fallstudien auch die Eignung des im Rahmen dieser Arbeit geschaffenen lokalen Suchverfahrens ASPO zur Testdatengenerierung für SL/TL-Modelle. Auch wenn die Ergebnisse zeigen, dass das lokale Suchverfahren ASPO nicht geeignet ist, um das globale Suchverfahren LGA in seiner Funktion innerhalb des in dieser Arbeit behandelten hybriden Testverfahrens abzulösen, so stellte sich dennoch (erwartungsgemäß) heraus, dass die Vorzüge von ASPO in Einzelfällen zum Tragen kommen. Es stellt sich daher die Frage, wie und in welchen Situationen ASPO innerhalb einer Testdatensuche eingesetzt werden könnte, um diese noch effizienter oder effektiver zu gestalten. Mischformen aus lokaler und globaler Suche, wie die in Abschnitt 2.4.1 behandelten memetischen Algorithmen, sind ein denkbarer Ansatz. Derartige Überlegungen und Untersuchungen, ausgerichtet auf eine Testdatengenerierung für SL/TL-Modelle, könnten Gegenstand zukünftiger Arbeiten sein.

Abschließend stellt sich die Frage, wie allgemeingültig die Resultate dieser Fallstudien sind und welche Faktoren dies maßgeblich beeinflussen. Als Fallbeispiele dienten zwei SL/TL-Modelle aus der Automobilindustrie. Bei Verwendung anderer Modelle könnte beispielsweise die Effizienzsteigerung deutlich geringer ausfallen, beispielsweise wenn nur wenige oder keine unerreichbaren CGs existieren. Die zwei Modelle dieser Studien wurden jedoch bewusst aus zwei unterschiedlichen Fahrzeugdomänen gewählt. Sie sind unterschiedlich groß und weisen verschiedene Charakteristiken auf. Daher sind sie durchaus als repräsentativ anzusehen. Natürlich wäre eine Bestätigung der gemachten

Beobachtungen für Modelle aus anderen Industrien hilfreich. Weitere Faktoren, welche die Allgemeingültigkeit der Studien beeinflussen, sind der gewählte Anwendungsfall des Strukturtests sowie die gewählten Überdeckungskriterien. Die verwendeten Überdeckungskriterien sind in der Praxis weit verbreitet. Der Strukturtest stand im Fokus der Arbeit – Fallstudien zur Anwendung des hybriden Testverfahrens für beispielsweise Funktionstests wären der nächste logische Schritt. Zuletzt ist darauf hinzuweisen, dass Fehler in der prototypischen Umsetzung des hybriden Testverfahrens nicht ausgeschlossen werden können. Die Wahrscheinlichkeit eines Fehlverhaltens wurde allerdings durch ausgiebige Tests minimiert. Die Fallstudienergebnisse (wie die Unerreichbarkeit von CGs und die durch eine Testsuite erzielte Modellüberdeckung) wurden zudem durch manuelle Prüfungen nachvollzogen.

7 ZUSAMMENFASSUNG UND AUSBLICK

Die modellbasierte Entwicklung der Software eingebetteter Systeme mittels SL und die automatische Codegenerierung mittels TL ist in der Automobilindustrie weit verbreitet. Da TL-konforme SL-Modelle im Entwicklungsprozess in der Regel die ersten ausführbaren Artefakte darstellen, ist die Qualitätssicherung und insbesondere der Test dieser Modelle im Sinne einer möglichst frühzeitigen Fehlerfindung besonders attraktiv. Aufgrund des steigenden Umfangs und der wachsenden Komplexität von Software in modernen Fahrzeugen bedarf es einer Systematisierung und Automatisierung der Testprozesse und Testaktivitäten.

Diese Arbeit adressiert in diesem Kontext die automatische Erzeugung von Testdaten zum Test von SL/TL-Modellen. Der Fokus liegt hierbei auf dem Strukturtest, bei welchem Testdaten zur Überdeckung der Struktur eines Modells hinsichtlich zustandsbezogener Kriterien gesucht sind. Die vorliegende Arbeit ist die Fortsetzung einer Vorarbeit von Windisch [132], welche die strukturorientierte Testdatengenerierung für SL-Modelle durch einen suchbasierten Ansatz realisierte. Die Anwendung dieses Ansatzes zur automatischen Testdatengenerierung für komplexe Modelle aus der industriellen Praxis zeigte, dass dieser Ansatz im Vergleich mit dem Zufallstest und einem kommerziellen Werkzeug zu einer höheren Modellüberdeckung führt. Die Laufzeit dieser Automatisierung fiel allerdings, zu Lasten der Anwendbarkeit des Ansatzes in der industriellen Praxis, sehr hoch aus. Das bestehende Potenzial einer Laufzeitverringerung stellte die Hauptmotivation der vorliegenden Arbeit dar.

7.1 ZUSAMMENFASSUNG

Die Grundidee dieser Arbeit besteht in der geeigneten Hybridisierung des suchbasierten Ansatzes mit zusätzlichen Techniken. Mit dem Ziel, eine effiziente Testdatengenerierung zu ermöglichen, wurden fünf Techniken entwickelt. Hierbei handelt es sich einerseits um drei statisch operierende Analysetechniken, welche vor der Durchführung automatischer Testdatensuchen verschiedene Aufgaben übernehmen, um die Effizienz des suchbasierten Ansatzes insgesamt zu steigern. Andererseits wird der suchbasierte Ansatz um eine Testdatengenerierung durch eine symbolische Ausführung, welche auf SMT-Solving zurückgreift, ergänzt.

Hinzu kommt ein lokales Suchverfahren, welches allerdings nicht Teil des gebildeten Hybrids wurde. Zweck und Grundzüge dieser fünf Beiträge werden im Folgenden zusammengefasst.

HYBRIDISIERUNG DER SUCHE MIT VORANALYSEN

Überdeckungsziel-Erreichbarkeitsprüfung. Die mit dem Kürzel SIA bezeichnete Technik hat das Ziel, die Erreichbarkeit der zustandsbezogenen Ziele (Überdeckungsziele, CGs) innerhalb des zu testenden SL/TL-Modells zu prüfen. Unerreichbare CGs werden herausgefiltert, so dass der suchbasierte Ansatz nicht mit hohem Aufwand nach nicht auffindbaren Testdaten fahndet. SIA führt zu diesem Zweck eine abstrakte Interpretation des SL/TL-Modells unter Verwendung einer eigens definierten Intervallsemantik durch. Diese Interpretation berücksichtigt die zeitliche Dynamik von SL/TL-Modellen und führt Modelltransformationen durch, welche den anderen der in dieser Arbeit vorgestellten Techniken zugute kommen. Resultat dieser Analyse sind Wertebereichsangaben für die Signale des zu testenden SL/TL-Modells. Aus diesen lässt sich ableiten, ob ein CG unerreichbar ist.

Modelleingang-Abhängigkeitsermittlung. Die mit der Abkürzung SDA vorgestellte Technik hat das Ziel, den bei einer Testdatensuche betrachteten Raum aller in Frage kommenden Testdaten (Suchraum) einzugrenzen und somit die Suche zu fokussieren. SDA analysiert unter Berücksichtigung der Funktionalitäten der im Modell enthaltenen Blöcke die Abhängigkeiten zwischen allen Signalen im Modell. Ein Abhängigkeitsverhältnis drückt dabei aus, dass sich die Wertebelegung innerhalb eines Signals auf Wertebelegungen in einem anderen Signal auswirken können. Die ermittelten Abhängigkeiten nutzt SDA, um Modelleingänge, die irrelevant für das Erreichen eines CG sind, von einer Testdatensuche für dieses CG auszuschließen. Die Suche kann sich somit auf das Finden passender Testdaten für relevante Modelleingänge konzentrieren.

Suchzielpriorisierung. Bei einem Strukturtest sind in der Regel für eine Vielzahl von CGs Testdaten zu finden. Der suchbasierte Ansatz beruht darauf, dass grundsätzlich für jedes CG eine Testdatensuche durchzuführen ist. Bei einer Testdatensuche kann es vorkommen, dass andere als das betrachtete CG zufällig mit erreicht werden (Kollateralüberdeckung). Um die Effizienz der gesamten Automatisierung zu steigern, ist es zum einen das Ziel der mit CGSeq abgekürzten Suchzielpriorisierung, die Kollateralüberdeckung der Testdatensuchen zu erhöhen. Zum anderen versucht CGSeq, Testdatensuchen für einfacher zu bearbeitende CGs vorzuziehen. CGSeq analysiert hierzu die Abhängigkeiten zwischen allen betrachteten CGs. Die Abhängigkeiten geben unter anderem wieder, dass die Erfüllung eines CG zur Erfüllung eines anderen CG führt. CGSeq

berücksichtigt hierbei auch Abhängigkeiten, die sich aus der Funktionalität des Modells ergeben. Basierend auf den ermittelten Abhängigkeiten überführt CGSeq einzelne CGs oder Kombinationen von CGs in sogenannte SGs (Suchziele). Diese priorisiert CGSeq durch Anwendung verschiedener Metriken für die darauffolgenden Testdatensuchen.

HYBRIDISIERUNG DER SUCHE MIT SMT-SOLVING

Eine im Unterschied zum suchbasierten Ansatz rein statisch operierende Testdatengenerierungstechnik ist die symbolische Ausführung. Diese sammelt im Zuge einer symbolischen Ausführung Bedingungen, deren Erfüllung die Erfüllung eines CG zur Folge haben. Aus der Lösung dieser Bedingungen mit beispielsweise einem SMT-Solver lassen sich Testdaten ableiten. Dieser Ansatz ist in vielen Fällen in der Lage, mit hoher Geschwindigkeit Testdaten zu generieren, weist aber in der Anwendung für SL/TL-Modelle zweifelsohne Einschränkungen auf. Die vorliegende Arbeit erarbeitete daher einen Ansatz zur Testdatengenerierung mittels SMT-Solving, welcher das SMT-Solving, bezogen auf ein einzelnes CG oder SG, nur dann einsetzt, wenn spezielle Einsatzkriterien zutreffen. Andernfalls wird eine Testdatensuche durchgeführt. Der Ansatz ist zudem in der Lage, plausible, soll heißen realistische Testdaten zu erzeugen.

LOKALES SUCHVERFAHREN

Die Vorarbeit von Windisch setzte zur Durchführung der Testdatensuchen primär einen genetischen Algorithmus ein. Dieser gehört zur Klasse der globalen Suchen (GS). Daneben existiert die Klasse lokaler Suchen (LS). Verfahren zur LS suchen im Suchraum im Unterschied zu einer GS meist nur an einer anstatt an mehreren Stellen gleichzeitig. Dem geringeren Grundaufwand und der äußerst systematischen sowie gründlichen Vorgehensweise einer LS steht das Risiko entgegen, nicht in wichtige Bereiche des Suchraums gelangen zu können. Motiviert durch die schwierige Abwägbarkeit dieser Vor- und Nachteile untersuchte die vorliegende Arbeit, ob eine LS im betrachteten Anwendungskontext erfolgreich und effizient Testdaten generieren kann. Hierzu wurde eine auf der AVM basierende LS namens ASPO entwickelt. Diese ist ausgerichtet auf die Suche plausibler Testdaten für SL/TL-Modelle.

VALIDIERUNG

Die in dieser Arbeit vorgestellten Techniken wurden prototypisch in einem Werkzeug implementiert. Mit diesem Werkzeug wurden zur Evaluierung der Beiträge dieser Arbeit Fallstudien durchgeführt. Zwei reale, aus der Automobilindustrie stammende SL/TL-Modelle dienten hierbei als Fallbeispiele. Es wurden verschiedene Konfigurationen des hybriden

Testverfahrens aufgestellt, in welchen die einzelnen Techniken dieser Arbeit jeweils aktiviert oder deaktiviert waren. Auf diese Weise konnten die Auswirkungen der einzelnen Beiträge auf die strukturorientierte Testdatengenerierung gesondert analysiert und bewertet werden.

Die vier Techniken, welche in das hybride Testverfahren eingeflossen sind – namentlich SDA, SIA, CGSeq und das SMT-Solving – konnten in den Fallstudien allesamt ihren Einsatz rechtfertigen. Durch den gemeinsamen Einsatz all dieser Techniken in Kombination mit einer globalen Testdatensuche konnte die Laufzeit der Strukturtest-Automatisierung im Falle beider Modelle um über 90% reduziert werden. Die größten Anteile an diesen Laufzeitverringerungen sind hierbei den statischen Voranalysen SIA und CGSeq zuzuschreiben. Die zusätzliche Laufzeit durch Anwendung dieser Techniken war, gemessen an der erzielten Laufzeitverringerung, verschwindend gering. Darüber hinaus wirkte sich der Einsatz jeder zusätzlichen Technik nicht negativ auf die erzielte Modellüberdeckung aus. Ganz im Gegenteil: Die SDA-Technik schaffte es bei einem der Modelle sogar, die Überdeckung zu erhöhen.

Die Evaluierung des lokalen Suchverfahrens ASPO ergab zum einen, dass der Einsatz der globalen Suche für die Breite der Testdatengenerierungsprobleme, die sich durch die aus einem SL/TL-Modell abgeleiteten CGs stellen, erfolgversprechender ist. Im Falle mancher CGs konnte die LS allerdings mit geringerem Aufwand erfüllende Testdaten finden. Daher steht die Überlegung im Raum, globale und lokale Suche in geeigneter Weise miteinander zu kombinieren. Dieser Schritt wurde allerdings im Rahmen dieser Arbeit nicht gegangen.

Der entwickelte Werkzeug-Prototyp und die durchgeführten Fallstudien zeigen, dass die Hybridisierung des suchbasierten Ansatzes, wie in dieser Arbeit realisiert, in einem Testdatengenerierungsverfahren für SL/TL-Modelle mündet, welches sowohl praxistauglich als auch anwenderfreundlich ist. Angewendet für einen Strukturtest stellte sich dabei auch heraus, dass das hybride Testverfahren über seine Testdatengenerierung relativ kleine Testsuiten zusammenstellt, so dass deren manuelle Weiterverwendung zu funktionalen Tests auch ohne Vorhandensein eines automatischen Testorakels praktikabel ist.

7.2 AUSBLICK

Auch wenn die vorgestellte Arbeit im betrachteten Themengebiet deutliche Fortschritte erzielen konnte, gibt es dennoch Anknüpfungspunkte und weitere Ideen, welche Grundlage zukünftiger Arbeiten sein könnten. Derlei Aspekte werden zum Abschluss dieser Arbeit in diesem Abschnitt adressiert.

WEITERENTWICKLUNG DER BEITRÄGE DIESER ARBEIT

Bei der Ausarbeitung der fünf in dieser Arbeit vorgestellten Beiträge wurden an bestimmten Stellen Einschränkungen gemacht und Festlegungen getroffen, um den Aufwand der Entwicklung dieser Techniken insgesamt in einem durchführbaren Rahmen zu halten. Die Erarbeitung zusätzlicher Lösungen könnte die Qualität dieser Techniken steigern.

Zeitliche Dynamik. So berücksichtigen die Voranalysen SDA und CGSeq bei Bildung ihrer Abhängigkeitsgraphen nicht die zeitliche Dynamik in einem SL/TL-Modell. Eine entsprechende Berücksichtigung erhöht die Komplexität der Analysen und ist daher nicht immer trivial zu lösen. Lässt sich dies jedoch bewerkstelligen, so können der Testdatensuche zum Beispiel Hinweise geliefert werden, zu welchen Zeitschritten oder Zeitphasen eine passende Stimulation bestimmter Modelleingänge notwendig ist, um ein SG zu erreichen (Erweiterung von SDA), oder welche CGs in Kombination zu verschiedenen Zeitschritten zu erfüllen sind, um bestimmte andere CGs zu erreichen (Erweiterung von CGSeq).

Stateflow und iterative Subsysteme. Darüber hinaus wäre es sinnvoll, die statischen Techniken dieser Arbeit um spezielle Behandlungen für SF-Blöcke zu erweitern. Diese Blöcke, welche ihre Funktionalität über Zustandsdiagramme abbilden, werden von den Voranalysen und bei der symbolischen Ausführung als unbekannte Blöcke aufgefasst. Zudem wurde in dieser Arbeit das SL-Konstrukt der Programmschleifen nicht betrachtet. Über spezielle Blöcke lassen sich beispielsweise *while*-Schleifen bilden. Der Schleifenkörper befindet sich dabei innerhalb eines speziellen Subsystems und die dort enthaltenen Blöcke werden je nach Schleifenbedingung pro Zeitschritt gegebenenfalls mehrfach ausgeführt.

Testdaten-Bedingungen. Die vorgestellte Arbeit berücksichtigt zudem die in der Arbeit von Windisch [132] vorgestellte Einbeziehung zusätzlicher Testdaten-Bedingungen (siehe Abschnitt 2.4.2) nicht explizit. Während sowohl globale als auch lokale Suche mit dem diesbezüglichen Ansatz von Windisch kompatibel sind, beachtet das SMT-Solving die zusätzlichen Bedingungen in dieser Arbeit nicht. Idealerweise sollten auch die Voranalysen SIA und CGSeq etwaige Testdaten-Bedingungen berücksichtigen.

Grenzen der symbolischen Ausführung. Der in dieser Arbeit verfolgte Ansatz einer Testdatengenerierung durch SMT-Solving setzt auf eine vorherige Prüfung von Einsatzkriterien. Da diese Kriterien unter Umständen eine Testdatengenerierung durch SMT-Solving für ein SG ausschließen, obwohl diese durchführbar wäre, ist eine weitergehende Untersuchung der Grenzen des SMT-Solving in dem betrachteten Kontext ratsam. Insbesondere die Aufnahme von Modellfunktionalitäten mit zeitlicher Dynamik in das SMT-Solving könnte die Effizienz der gesamten Testdatengenerierung für ein Modell nochmals steigern.

Lokale Suche. Wie in der Zusammenfassung der Arbeit bereits angedeutet, könnte die Intelligenz des suchbasierten Ansatzes darüber hinaus durch eine geschickte Kombination von lokaler und globaler Suche erhöht werden. Für das lokale Suchverfahren selbst existieren zudem potenzielle Verbesserungsmöglichkeiten, wie die Wiederaufnahme von erfolglosen Suchen mit anderen initialen Testdaten oder die Generierung von Optimierungssequenzen mit variabler Segmentanzahl.

WEITERVERWENDUNG GENERIERTER TESTDATEN

Das hybride Testverfahren dieser Arbeit automatisiert, wie in den Fallstudien demonstriert, das Auffinden von Testdaten für einen Strukturtest. Eingesetzt in einem praktischen Umfeld stellt sich jedoch die Frage, wie mit den erzeugten Testdaten umzugehen ist – denn zur Identifikation möglicher Fehler im Modell ist eine Überprüfung des Testobjektverhaltens bei Ausführung mit diesen Testdaten notwendig. Existieren beispielsweise Auswertungsskripte aus funktionalen Tests, so wäre eine Wiederverwendung dieser denkbar. Sind keine solchen Skripte oder anderweitige Testorakel verfügbar, wäre es hilfreich, zumindest eingrenzen zu können, welche Anforderungen aus der Spezifikation bei der Überprüfung der Testdaten jeweils von Relevanz sind. Existieren (transitive) Verlinkungen zwischen Anforderungen und Subsystemen (oder sogar einzelnen Blöcken) in Modellen, so könnte ein Mapping zwischen den durch ein Testdatum überdeckten CGs und, über deren Zuordnung innerhalb des Modells, den relevanten Anforderungen erfolgen. Da automatisch generierte Testdaten meist funktionale Tests ergänzen, stellt sich zudem die Frage, ob die funktionalen Testfälle oder sogar die Spezifikation möglicherweise Lücken aufweisen. Eine automatische Analyse der Nähe automatisch generierter Testdaten zu den Testfällen vorheriger Funktionstests wäre daher wünschenswert.

WIEDERVERWENDUNG GENERIERTER TESTDATEN

Unter zweierlei Gesichtspunkten könnten Überlegungen zur Wiederverwendung automatisch generierter Testdaten angestellt werden. Zum einen innerhalb der Testdatengenerierung für den Strukturtest eines Modells und zum anderen bei Testdatengenerierungen für Strukturtests verschiedener Modellversionen. Die Testdatensuchen, welche das hybride Testverfahren durchführt, starten initial mit einem Satz zufällig generierter Testdaten. Möglicherweise lassen sich Zusammenhänge innerhalb eines Modells oder zwischen CGs ausnutzen, um den Satz initialer Testdaten konstruktiver zu gestalten. Wird ein überarbeitetes oder verändertes Modell zudem einem erneuten Strukturtest unterzogen, stellt sich die Frage, inwiefern zuvor generierte Testdaten wiederverwendbar sind.

FUNKTIONSTESTS

In Abschnitt 2.2 wurden verschiedene Szenarien und Testarten behandelt, in denen eine automatisierte Testdatengenerierung für SL/TL-Modelle sinnvoll erscheint. Der Strukturtest wurde dabei als Repräsentant der verschiedenen Szenarien ausgewählt und bildete fortan den Fokus bei der Ausgestaltung der Techniken dieser Arbeit – sowie in den Fallstudien. Es empfiehlt sich eine Evaluierung des entstandenen hybriden Testverfahrens in einem anderen Kontext, insbesondere zu Zwecken von Funktionstests (siehe Abschnitt 2.2.2). In diesem Anwendungsfall wäre zudem zu klären, ob und inwieweit die SL- und TL-Sprachmittel zur Definition funktionaler Modelleigenschaften genügen – und welche neuen, speziellen Blocktypen gegebenenfalls ergänzend benötigt werden.

WEITERE IDEEN UND ANSÄTZE

Beim Studium verwandter Arbeiten sowie der Gestaltung und Entwicklung der Beiträge dieser Arbeit sind weitere Ideen und Überlegungen entstanden, welche an dieser Stelle nicht unerwähnt bleiben sollen.

Signallänge. Die Größe des Suchraums hängt neben der Anzahl relevanter Modelleingänge wesentlich von der Länge der zu generierenden Signale ab. Diese wird vom Anwender vorab spezifiziert. Eine statische Analyse des Modells könnte die zum Erreichen eines CG erforderliche Mindestlänge der Signale möglicherweise automatisch bestimmen.

Adaptive Abbruchkriterien. Wie im Zuge der Vorstellung der CGSeq-Technik betont, sind die bei einem Strukturtest auftretenden CGs üblicherweise unterschiedlich einfach oder schwer durch eine Testdatensuche zu erfüllen. Je nach Schwierigkeitsgrad könnte man einer Testdatensuche einen geringeren oder höheren Suchaufwand zugestehen. So könnte für die maximal erlaubte Anzahl an Optimierungsiterationen im Falle eines CG mit einer geringen Modelltiefe ein niedriger Wert gewählt werden als für ein CG mit einer hohen Modelltiefe.

Gezielte Testdatenmodifikation. Die Fitness eines Testdatums bestimmt sich im betrachteten Kontext, wie in Abschnitt 2.4.2 dargestellt, aus einer Sequenz von Distanzwerten - oder anders ausgedrückt: aus einem Fitness-Signal. Zwischen Fitnessbestimmung und Testdatenmodifikation kommt es bei einer Testdatensuche zu einem Informationsverlust. Die Information, bei welchem Zeitschritt das Fitness-Signal den geringsten Distanzwert aufweist, spielt nämlich bei den Änderungsoperatoren des LGA und auch der ASPO keine Rolle. Diesbezüglich gezieltere Testdatenmodifikationen könnten die Suche beschleunigen.

Testdatenregionen. Verschiedene andere Arbeiten im Bereich der automatisierten Testdatengenerierung [2, 70] beschäftigen sich mit der Identifikation von Bereichen ähnlicher oder sogar redundanter Testdaten

im Suchraum. Eine Technik, welche bei der Testdatensuche für ein CG Regionen im Suchraum erkennt, welche hinsichtlich der CG-Erfüllung äquivalent sind, könnte dem Erfolg und der Effizienz einer Suche überaus zuträglich sein.

Modelltransformationen. Bei der automatisierten Testdatengenerierung für Programmcode werden Teile des Codes oder der abgeleitete Kontrollflussgraph oftmals transformiert, um die Testdatengenerierung zu erleichtern. Bei solchen Transformationen handelt es sich in der Regel um semantikbewahrende Transformationen. Das hybride Testverfahren dieser Arbeit transformiert das zu testende Modell in dieser Hinsicht bereits zum Teil. Allerdings wurde dieser Aspekt nur im Ansatz untersucht. Die Arbeit von Tran et al. [119], welche sich ausgiebig mit der Transformation von SL-Modellen beschäftigt, könnte als Ausgangspunkt einer systematischen Betrachtung von Modelltransformationen zum Zwecke einer verbesserten Testbarkeit dienen.

Werkzeug-Features. Neben derlei technischen Überlegungen sind im Zuge dieser Arbeit auch Aspekte aufgekommen, welche die Akzeptanz und Anwendbarkeit eines Testdatengenerierungswerkzeugs für SL/TL-Modelle in der Praxis adressieren. Einer dieser Aspekte betrifft die Existenz von individuellen Modellböcken, deren Funktionalität der Modellierer durch Programmcode beschreibt. Auch dieser Programmcode könnte bei einem Strukturtest miteinbezogen und durch Code-Überdeckungskriterien adressiert werden. Des Weiteren sollten Techniken zur Testsuite-Minimierung zum Einsatz kommen, um nach der Testdatengenerierung eine möglichst kleine Testsuite ausgeben zu können. Das Werkzeug sollte zudem Überdeckungskriterien wie MC/DC unterstützen. In dieser Arbeit wurde dieses Kriterium nicht behandelt. Es existieren allerdings Arbeiten [8, 46], welche zeigen, dass mit einem suchbasierten Ansatz auch dieses Kriterium behandelt werden kann. Zuletzt sei darauf hingewiesen, dass aktuelle Versionen des SL-Werkzeugs eine parallelisierte Ausführung von Modellsimulationen ermöglichen. In der Version, welche im Rahmen dieser Arbeit genutzt wurde, ist dies nicht möglich. Eine Parallelisierung der Modellausführungen könnte die Laufzeit des hybriden Testverfahrens nochmals deutlich reduzieren.

LITERATURVERZEICHNIS

[1] R. K. Ahuja, Özlem Ergun, J. B. Orlin und A. P. Punnen (2002). *A survey of very large-scale neighborhood search techniques*. Discrete Applied Mathematics, **123**(1–3), 75–102. ISSN 0166-218X. (Zitiert auf Seite 55.)

[2] R. Alur, A. Kanade, S. Ramesh und K. C. Shashidhar (2008). *Symbolic Analysis for Improving Simulation Coverage of Simulink/Stateflow Models*. In Proceedings of the 8th ACM International Conference on Embedded software, Seiten 89–98. EMSOFT '08, ACM, New York, NY, USA. ISBN 978-1-60558-468-3. (Zitiert auf Seite 227.)

[3] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold und P. McMinn (2013). *An Orchestrated Survey of Methodologies for Automated Software Test Case Generation*. Journal of Systems and Software, **86**(8), 1978–2001. ISSN 0164-1212. (Zitiert auf den Seiten 40 und 41.)

[4] A. Arcuri und L. C. Briand (2011). *Adaptive Random Testing: An Illusion of Effectiveness*. In M. B. Dwyer und F. Tip, Herausgeber, Proceedings of the 20th International Symposium on Software Testing and Analysis, Seiten 265–275. ISBN 978-1-4503-0562-4. (Zitiert auf den Seiten 45 und 46.)

[5] A. Arcuri, M. Z. Iqbal und L. Briand (2010). *Formal Analysis of the Effectiveness and Predictability of Random Testing*. In Proceedings of the 19th International Symposium on Software Testing and Analysis, Seiten 219–230. ISSTA '10, ACM. ISBN 978-1-60558-823-0. (Zitiert auf Seite 45.)

[6] A. Arcuri, M. Iqbal und L. Briand (2012). *Random Testing: Theoretical Results and Practical Implications*. IEEE Transactions on Software Engineering, **38**(2), 258–277. ISSN 0098-5589. (Zitiert auf Seite 45.)

[7] M. Association (2014). *Modelica*. URL `https://www.modelica.org`. Letzter Zugriff: 05/2014. (Zitiert auf Seite 13.)

[8] Z. Awedikian, K. Ayari und G. Antoniol (2009). *MC/DC automatic test input data generation*. In Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, Seiten 1657–1664. GECCO '09, ACM, New York, NY, USA. ISBN 978-1-60558-325-9. (Zitiert auf Seite 228.)

[9] M. Baluda, P. Braione, G. Denaro und M. Pezzè (2010). *Structural Coverage of Feasible Code*. In Proceedings of the 5th Workshop on Automation of Software Test, Seiten 59–66. AST '10, ACM, New York, NY, USA. ISBN 978-1-60558-970-1. (Zitiert auf Seite 33.)

[10] A. Baresel, M. Conrad, S. Sadeghipour und J. Wegener (2003). *The Interplay between Model Coverage and Code Coverage*. In Conference On Computer Aided Systems Theory. (Zitiert auf den Seiten 30, 31 und 37.)

[11] A. Baresel, H. Pohlheim und S. Sadeghipour (2003). Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In E. Cantú-Paz, J. A. Foster, K. Deb, L. D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. A. Dowsland, N. Jonoska, und J. Miller, Herausgeber, Genetic and Evolutionary Computation — GECCO 2003, Band 2724 von *Lecture Notes in Computer Science*, Seiten 2428–2441. Springer Berlin Heidelberg. ISBN 978-3-540-40603-7. (Zitiert auf Seite 59.)

[12] M. Barr (2013). *Toyota and Unintended Acceleration*. URL `http://embeddedgurus.com/barr-code/2013/10/an-update-on-toyota-and-unintended-acceleration`. Letzter Zugriff: 05/2014. (Zitiert auf Seite 4.)

[13] C. Barrett, R. Sebastiani, S. A. Seshia und C. Tinelli (2009). *Satisfiability Modulo Theories*, Kapitel 26, Seiten 825–885. Band 185 von [18]. (Zitiert auf Seite 163.)

[14] C. Barrett, A. Stump und C. Tinelli (2010). *The SMT-LIB Standard: Version 2.0*. In A. Gupta und D. Kroening, Herausgeber, Proceedings of the 8th International Workshop on Satisfiability Modulo Theories. (Zitiert auf Seite 164.)

[15] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovi?, T. King, A. Reynolds und C. Tinelli (2011). Cvc4. In G. Gopalakrishnan und S. Qadeer, Herausgeber, Computer Aided Verification, Band 6806 von *Lecture Notes in Computer Science*, Seiten 171–177. Springer Berlin Heidelberg. ISBN 978-3-642-22109-5. (Zitiert auf Seite 163.)

[16] C. Barrett, M. Deters, L. M. de Moura, A. Oliveras und A. Stump (2013). *6 Years of SMT-COMP*. J. Autom. Reasoning, **50**(3), 243–277. (Zitiert auf Seite 164.)

[17] K. Berns, B. Schürmann und M. Trapp (2010). *Eingebettete Systeme - Systemgrundlagen und Entwicklung eingebetteter Software.* Vieweg+Teubner. ISBN 978-3-8348-0422-8. (Zitiert auf Seite 12.)

[18] A. Biere, M. J. H. Heule, H. van Maaren und T. Walsh, Herausgeber (2009). *Handbook of Satisfiability*, Band 185 von *Frontiers in Artificial Intelligence and Applications*. IOS Press. ISBN 978-1-58603-929-5, 980 Seiten. (Zitiert auf den Seiten 163 und 230.)

[19] N. Bjørner (2012). Taking satisfiability to the next level with z3. In B. Gramlich, D. Miller, und U. Sattler, Herausgeber, Automated Reasoning, Band 7364 von *Lecture Notes in Computer Science*, Seiten 1–8. Springer Berlin Heidelberg. ISBN 978-3-642-31364-6. (Zitiert auf Seite 163.)

[20] P. Boström und J. Björkqvist (2005). Optimisation-based black-box testing of assertions in Simulink models. Technischer Bericht 711, Åbo Akademi University. (Zitiert auf Seite 47.)

[21] A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rümmer und G. Weissenbacher (2010). Mutation-based test case generation for simulink models. In F. Boer, M. Bonsangue, S. Hallerstede, und M. Leuschel, Herausgeber, Formal Methods for Components and Objects, Band 6286 von *Lecture Notes in Computer Science*, Seiten 208–227. Springer Berlin Heidelberg. ISBN 978-3-642-17070-6. (Zitiert auf Seite 43.)

[22] E. Bringmann und A. Krämer (2008). *Model-Based Testing of Automotive Systems*. In Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, Seiten 485–493. ICST '08, IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-3127-4. (Zitiert auf Seite 12.)

[23] V. H. S. Campos, R. E. Rodrigues, I. R. de Assis Costa und F. M. Q. a. Pereira (2012). *Speed and Precision in Range Analysis*. In Proceedings of the 16th Brazilian Conference on Programming Languages, Seiten 42–56. SBLP'12, Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-642-33181-7. (Zitiert auf Seite 94.)

[24] T. Y. Chen, F.-C. Kuo, R. G. Merkel und T. H. Tse (2010). *Adaptive Random Testing: The ART of Test Case Diversity*. Journal of Systems and Software, **83**(1), 60–66. ISSN 0164-1212. (Zitiert auf Seite 45.)

[25] A. Cimatti, A. Griggio, B. Schaafsma und R. Sebastiani (2013). *The MathSAT5 SMT Solver*. In N. Piterman und S. Smolka, Herausgeber, Proceedings of TACAS, Band 7795 von *LNCS*. Springer. (Zitiert auf Seite 163.)

[26] D. Cok, A. Griggio, R. Bruttomesso und M. Deters (2013). *The 2012 SMT Competition*. In P. Fontaine und A. Goel, Herausgeber, SMT 2012, Band 20 von *EPiC Series*, Seiten 131–142. EasyChair. ISSN 2040-557X. (Zitiert auf Seite 164.)

[27] D. R. Cok (2013). *The SMT-LIBv2 Language and Tools: A Tutorial*. Online. URL https://www.lri.fr/~conchon/TER/2013/2/SMTLIB2.pdf. Letzter Zugriff: 05/2014. (Zitiert auf Seite 164.)

[28] M. Conrad (2004). *Modell-basierter Test eingebetteter Software im Automobil: Auswahl und Beschreibung von Testszenarien*. Dissertation, Technische Universität Berlin. (Zitiert auf Seite 29.)

[29] M. Conrad und S. Sadeghipour (2002). *Einsatz von Überdeckungskriterien auf Modellebene - Erfahrungsbericht und experimentelle Ergebnisse*. Softwaretechnik-Trends, **22**(2). (Zitiert auf Seite 34.)

[30] P. Cousot und R. Cousot (1976). *Static Determination of Dynamic Properties of Programs*. In Proceedings of the Second International Symposium on Programming, Seiten 106–130. (Zitiert auf den Seiten 71 und 93.)

[31] P. Cousot und R. Cousot (1977). *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Seiten 238–252. POPL '77, ACM, New York, NY, USA. (Zitiert auf Seite 71.)

[32] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux und X. Rival (2005). *The ASTRÉE analyzer*. In Programming Languages and Systems, Proceedings of the 14th European Symposium on Programming, volume 3444 of Lecture Notes in Computer Science, Seiten 21–30. Springer. (Zitiert auf Seite 94.)

[33] L. De Moura und N. Bjørner (2008). *Z3: An Efficient SMT Solver*. In Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Seiten 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg. ISBN 3-540-78799-2, 978-3-540-78799-0. (Zitiert auf Seite 163.)

[34] L. De Moura und N. Bjørner (2011). *Satisfiability Modulo Theories: Introduction and Applications*. Commun. ACM, **54**(9), 69–77. ISSN 0001-0782. (Zitiert auf Seite 162.)

[35] D. do Couto Teixeira und F. M. Q. Pereira (2011). *The Design and Implementation of a Non-Iterative Range Analysis Algorithm on a Production Compiler*. SBLP. SBC, Seiten 45–59. (Zitiert auf Seite 94.)

[36] dSpace (2014). *TargetLink*. URL http://www.dspace.com. Letzter Zugriff: 05/2014. (Zitiert auf Seite 5.)

[37] F. Elberzhager, A. Rosbach und T. Bauer (2013). *Analysis and Testing of Matlab Simulink Models: A Systematic Mapping Study*. In Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation, Seiten 29–34. JAMAICA 2013, ACM, New York, NY, USA. ISBN 978-1-4503-2161-7. (Zitiert auf Seite 30.)

[38] M. D. Ernst (2003). *Static and Dynamic Analysis: Synergy and Duality*. In WODA 2003: Workshop on Dynamic Analysis, Seiten 24–27. Portland, Oregon. (Zitiert auf den Seiten 7 und 70.)

[39] I. O. for Standardization / Technical Committee 22 (ISO/TC 22) (2009). *ISO/DIS 26262 - Road vehicles — Functional safety*. (Zitiert auf den Seiten 13 und 16.)

[40] G. Fraser und A. Arcuri (2011). *Evolutionary Generation of Whole Test Suites*. In 11th International Conference on Quality Software (QSIC), Seiten 31–40. ISSN 1550-6002. (Zitiert auf Seite 136.)

[41] G. Fraser und F. Wotawa (2007). *Improving Model-Checkers for Software Testing*. In International Conference on Quality Software, Seiten 25–31. IEEE Computer Society, Los Alamitos, CA, USA. ISSN 1550-6002. (Zitiert auf den Seiten 5 und 42.)

[42] G. Fraser, A. Arcuri und P. McMinn (2013). *Test Suite Generation with Memetic Algorithms*. In Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference, Seiten 1437–1444. GECCO '13, ACM, New York, NY, USA. ISBN 978-1-4503-1963-8. (Zitiert auf Seite 56.)

[43] A. A. Gadkari, S. Mohalik, K. Shashidhar, A. Yeolekar, J. Suresh und S. Ramesh (2007). *Automatic Generation of Test-Cases Using Model Checking for SL/SF Models*. In Proceedings of the 4th Model-Driven Engineering, Verification and Validation Workshop, Seiten 33–46. (Zitiert auf Seite 43.)

[44] A. A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik und K. C. Shashidhar (2008). *AutoMOTGen: Automatic Model Oriented Test Generator for Embedded Control Systems*. In Proceedings of the 20th International Conference on Computer Aided Verification, Seiten 204—-208. (Zitiert auf Seite 43.)

[45] T. Gawlitza, J. Leroux, J. Reineke, H. Seidl, G. Sutre und R. Wilhelm (2009). Polynomial precise interval analysis revisited. In S. Albers, H. Alt, und S. Näher, Herausgeber, Efficient Algorithms, Band 5760 von *Lecture Notes in Computer Science*, Seiten 422–437. Springer Berlin Heidelberg. ISBN 978-3-642-03455-8. (Zitiert auf Seite 93.)

[46] K. Ghani und J. A. Clark (2009). *Automatic Test Data Generation for Multiple Condition and MCDC Coverage*. In Proceedings of the 2009 4th International Conference on Software Engineering Advances, Seiten 152–157. ICSEA '09, Washington, DC, USA. ISBN 978-0-7695-3777-1. (Zitiert auf Seite 228.)

[47] K. Ghani und J. A. Clark (2009). *Widening the Goal Posts: Program Stretching to Aid Search Based Software Testing*. In 1st International Symposium on Search Based Software Engineering, Seiten 122–131. (Zitiert auf Seite 47.)

[48] K. Ghani, J. A. Clark und Y. Zhan (2009). *Comparing Algorithms for Search-Based Test Data Generation of Matlab Simulink Models*, Seiten 2940–2947. IEEE. ISBN 978-1-4244-2958-5. (Zitiert auf Seite 160.)

[49] P. Godefroid, N. Klarlund und K. Sen (2005). *DART: Directed Automated Random Testing*. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Seiten 213–223. PLDI '05, ACM, New York, NY, USA. ISBN 1-59593-056-6. (Zitiert auf den Seiten 5, 48 und 49.)

[50] A. Goldberg, T. C. Wang und D. Zimmerman (1994). *Applications of Feasible Path Analysis to Program Testing*. In Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis, Seiten 80–94. ISSTA '94, ACM, New York, NY, USA. ISBN 0-89791-683-2. (Zitiert auf den Seiten 70 und 94.)

[51] K. Grimm (1995). *Systematisches Testen von Software: eine neue Methode und eine effektive Teststrategie*. Dissertation, Technische Universität Berlin. (Zitiert auf den Seiten 29 und 31.)

[52] S. Gulwani und S. Juvekar (2009). Bound analysis using backward symbolic execution. Technischer Bericht, Microsoft Research. (Zitiert auf Seite 167.)

[53] G. Hamon (2005). *A Denotational Semantics for Stateflow*. In Proceedings of the 5th ACM International Conference on Embedded Software, Seiten 164–172. EMSOFT '05, ACM, New York, NY, USA. ISBN 1-59593-091-4. (Zitiert auf Seite 17.)

[54] G. Hamon (2008). *Simulink Design Verifier - Applying Automated Formal Methods to Simulink and Stateflow*. In AFM'08: Third Workshop on Automated Formal Methods. (Zitiert auf Seite 43.)

[55] G. Hamon und J. Rushby (2007). *An Operational Semantics for Stateflow*. International Journal on Software Tools for Technology Transfer, **9**(5-6), 447–456. ISSN 1433-2779. (Zitiert auf Seite 17.)

[56] G. Hamon, L. de Moura und J. Rushby (2004). *Generating Efficient Test Sets with a Model Checker*. In 2nd International Conference on Software Engineering and Formal Methods, Seiten 261–270. IEEE Computer Society, Beijing, China. (Zitiert auf Seite 43.)

[57] M. Harman (2007). *The Current State and Future of Search Based Software Engineering*. In 2007 Future of Software Engineering, Seiten 342–357. FOSE '07, IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2829-5. (Zitiert auf Seite 55.)

[58] M. Harman und R. Hierons (2001). *An Overview of Program Slicing*. Software Focus, **2**(3), 85–92. ISSN 1529-7950. (Zitiert auf Seite 106.)

[59] M. Harman und P. McMinn (2010). *A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search*. IEEE Transactions on Software Engineering, **36**(2), 226–247. ISSN 0098-5589. (Zitiert auf den Seiten 46, 52, 54 und 56.)

[60] M. Harman, C. Fox, R. Hierons, L. Hu, S. Danicic und J. Wegener (2002). *VADA: A Transformation-Based System for Variable Dependence Analysis*. In Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation, Seiten 55–64. (Zitiert auf Seite 105.)

[61] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn und J. Wegener (2007). *The Impact of Input Domain Reduction on Search-based Test Data Generation*. In Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Seiten 155–164. ESEC-FSE '07, New York, NY, USA. ISBN 978-1-59593-811-4. (Zitiert auf Seite 105.)

[62] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn und S. Yoo (2010). *Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem*. In 3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), Seiten 182–191. (Zitiert auf Seite 136.)

[63] K. Hartig (2013). *Einsatz lokaler Suchverfahren zur strukturorientierten Testdatengenerierung für Simulink-/TargetLink-Modelle*. Diplomarbeit, TU Berlin. (Zitiert auf den Seiten 141, 159 und 198.)

[64] N. He, P. Rümmer und D. Kroening (2011). *Test-Case Generation for Embedded Simulink via Formal Concept Analysis*. In Proceedings of the 48th Design Automation Conference, Seiten 224–229. DAC '11, ACM, New York, NY, USA. ISBN 978-1-4503-0636-2. (Zitiert auf Seite 43.)

[65] P. Herber, R. Reicherdt und P. Bittner (2013). *Bit-precise formal verification of discrete-time MATLAB/Simulink Models using SMT Solving*. In 2013 Proceedings of the International Conference on Embedded Software (EMSOFT), Seiten 1–10. (Zitiert auf Seite 176.)

[66] R. Höhn und S. Höppner, Herausgeber (2008). *Das V-Modell XT: Grundlagen, Methodik und Anwendungen*. Springer, Berlin. ISBN 978-3-540-30249-0. (Zitiert auf Seite 28.)

[67] S. Horwitz, T. Reps und D. Binkley (1988). *Interprocedural Slicing Using Dependence Graphs*. In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, Seiten 35–46. PLDI '88, ACM. ISBN 0-89791-269-1. (Zitiert auf Seite 98.)

[68] K. Inkumsah und T. Xie (2007). *Evacon: A Framework for Integrating Evolutionary and Concolic Testing for Object-Oriented Programs*. In Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, Seiten 425–428. ASE '07, ACM, New York, NY, USA. ISBN 978-1-59593-882-4. (Zitiert auf Seite 50.)

[69] N. Instruments (2014). *LabView*. URL `http://www.ni.com/labview/d`. Letzter Zugriff: 05/2014. (Zitiert auf Seite 13.)

[70] A. Kanade, R. Alur, F. Ivančić, S. Ramesh, S. Sankaranarayanan und K. C. Shashidhar (2009). *Generating and Analyzing Symbolic Traces of Simulink/Stateflow Models*. In Proceedings of the 21st International Conference on Computer Aided Verification, Seiten 430–445. CAV '09, Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-642-02657-7. (Zitiert auf den Seiten 44 und 227.)

[71] O. Kindel und M. Friedrich (2009). *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. dpunkt, Heidelberg. ISBN 978-3-898-64563-8. (Zitiert auf Seite 13.)

[72] J. C. King (1975). *A New Approach to Program Testing*. SIGPLAN Not., **10**(6), 228–233. ISSN 0362-1340. (Zitiert auf den Seiten 5 und 41.)

[73] S. Kirkpatrick, C. D. Gelatt und M. P. Vecchi (1984). *Optimization by Simulated Annealing: Quantitative Studies*. Journal of Statistical Physics, **34**(5-6), 975–986. ISSN 0022-4715. (Zitiert auf Seite 160.)

[74] R. Kirner (2009). *Towards Preserving Model Coverage and Structural Code Coverage*. EURASIP Journal on Embedded Systems, **2009**, 6:1–6:16. ISSN 1687-3955. (Zitiert auf Seite 37.)

[75] S. Kirstan (2011). *Kosten und Nutzen modellbasierter Entwicklung eingebetteter Softwaresysteme im Automobil*. Dissertation, Technische Universität München. (Zitiert auf Seite 12.)

[76] B. Korel (1990). *Automated Software Test Data Generation*. IEEE Transactions on Software Engineering, **16**(8), 870–879. (Zitiert auf den Seiten 46, 54, 105, 142 und 156.)

[77] K. Lakhotia, P. McMinn und M. Harman (2009). *Automated Test Data Generation for Coverage: Haven't We Solved This Problem Yet?* In Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques, Seiten 95–104. TAIC-PART '09, IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-3820-4. (Zitiert auf Seite 49.)

[78] K. Lakhotia, P. McMinn und M. Harman (2010). *An Empirical Investigation Into Branch Coverage for C Programs Using CUTE and AUSTIN*. Journal of Systems and Software, **83**(12), 2379–2391. ISSN 0164-1212. (Zitiert auf den Seiten 49 und 70.)

[79] K. Lakhotia, N. Tillmann, M. Harman und J. De Halleux (2010). *FloPSy: Search-Based Floating Point Constraint Solving for Symbolic Execution*. In Proceedings of the 22nd IFIP WG 6.1 International Conference on Testing Software and Systems, Seiten 142–157. ICTSS'10, Springer-Verlag, Berlin, Heidelberg. ISBN 3-642-16572-9, 978-3-642-16572-6. (Zitiert auf Seite 49.)

[80] E. Legros, W. Schäfer, A. Schürr und I. Stürmer (2011). Mate - a model analysis and transformation environment for matlab simulink. In H. Giese, G. Karsai, E. Lee, B. Rumpe, und B. Schätz, Herausgeber, Model-Based Engineering of Embedded Real-Time Systems, Band 6100 von *Lecture Notes in Computer Science*, Seiten 323–328. Springer Berlin Heidelberg. ISBN 978-3-642-16276-3. (Zitiert auf Seite 17.)

[81] J. J. Li und H. Yee (2004). *Code-coverage guided prioritized test generation*. In Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC), Band 2, Seiten 178–181. ISSN 0730-3157. (Zitiert auf Seite 137.)

[82] P. Liggesmeyer (2009). *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag. ISBN 9783827420565. (Zitiert auf Seite 31.)

[83] P. McMinn (2011). *Search-Based Software Testing: Past, Present and Future*. In IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Seiten 153–163. (Zitiert auf den Seiten 5, 46 und 52.)

[84] P. McMinn und M. Holcombe (2004). Hybridizing evolutionary testing with the chaining approach. In K. Deb, Herausgeber, Genetic and Evolutionary Computation – GECCO 2004, Band 3103 von *Lecture Notes in Computer Science*, Seiten 1363–1374. Springer Berlin Heidelberg. ISBN 978-3-540-22343-6. (Zitiert auf den Seiten 64 und 136.)

[85] P. McMinn und M. Holcombe (2005). *Evolutionary Testing of State-Based Programs*. In Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, Seiten 1013–1020. GECCO '05. ISBN 1-59593-010-8. (Zitiert auf Seite 57.)

[86] P. McMinn, M. Harman, D. Binkley und P. Tonella (2006). *The Species per Path Approach to Search-Based Test Data Generation*. In Proceedings of the 2006 International Symposium on Software Testing and Analysis, Seiten 13–24. ISSTA '06, ACM, New York, NY, USA. ISBN 1-59593-263-1. (Zitiert auf Seite 64.)

[87] P. McMinn, M. Harman, K. Lakhotia, Y. Hassoun und J. Wegener (2012). *Input Domain Reduction through Irrelevant Variable Removal and Its Effect on Local, Global, and Hybrid Search-Based Structural Test Data Generation*. IEEE Transactions on Software Engineering, **38**(2), 453–477. ISSN 0098-5589. (Zitiert auf Seite 105.)

[88] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller und E. Teller (1953). *Equation of State Calculations by Fast Computing Machines*. The Journal of Chemical Physics, **21**(6), 1087–1092. (Zitiert auf Seite 160.)

[89] W. Miller und D. Spooner (1976). *Automatic Generation of Floating-Point Test Data*. IEEE Transactions on Software Engineering, **SE-2**(3), 223–226. (Zitiert auf Seite 46.)

[90] S. Mohalik, A. A. Gadkari, A. Yeolekar, K. Shashidhar und S. Ramesh (2013). *Automatic Test Case Generation from Simulink/Stateflow Models Using Model Checking*. Software Testing, Verification and Reliability. ISSN 1099-1689. (Zitiert auf Seite 43.)

[91] R. E. Moore (1966). *Interval Analysis*. Prentice-Hall. (Zitiert auf Seite 73.)

[92] L. Moura und N. Bjørner (2009). Satisfiability modulo theories: An appetizer. In M. V. Oliveira und J. Woodcock, Herausgeber, Formal Methods: Foundations and Applications, Seiten 23–36. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-642-10451-0. (Zitiert auf Seite 163.)

[93] F. Neri, C. Cotta und P. Moscato (2011). *Handbook of Memetic Algorithms*. Studies in Computational Intelligence, Springer. ISBN 9783642232466. (Zitiert auf den Seiten 55 und 143.)

[94] J. Oh, M. Harman und S. Yoo (2011). *Transition Coverage Testing for Simulink/Stateflow Models Using Messy Genetic Algorithms*. In Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO), Seiten 1851–1858. ACM. (Zitiert auf Seite 47.)

[95] P. Peranandam, S. Raviram, M. Satpathy, A. Yeolekar, A. Gadkari und S. Ramesh (2012). *An integrated test generation tool for enhanced coverage of Simulink/Stateflow models*. In Proceedings of the Conference on Design, Automation and Test in Europe, Seiten 308–311. (Zitiert auf den Seiten 50 und 176.)

[96] C. S. Păsăreanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karsai, H. Nine und S. Neema (2009). *Model Based Analysis and Test Generation for Flight Software*. In Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology, Seiten 83–90. (Zitiert auf den Seiten 43 und 176.)

[97] Reactive Systems (2014). *Reactis*. URL `http://www.reactive-systems.com`. Letzter Zugriff: 05/2014. (Zitiert auf den Seiten 33 und 51.)

[98] R. Reicherdt und S. Glesner (2012). *Slicing MATLAB Simulink models*. In 34th International Conference on Software Engineering (ICSE), Seiten 551–561. ISSN 0270-5257. (Zitiert auf Seite 106.)

[99] G. F. Renfer (1962). *Automatic Program Testing*. In Proceedings of 3rd Conference of the Computing and Data Processing Society of Canada. University of Toronto Press. (Zitiert auf den Seiten 5 und 45.)

[100] M. Research (2014). *Pex – White box Unit Testing for .NET*. URL `http://research.microsoft.com/en-us/projects/Pex`. Letzter Zugriff: 05/2014. (Zitiert auf den Seiten 49 und 164.)

[101] P. Roy und N. Shankar (2010). *SimCheck: An Expressive Type System for Simulink*. In C. Muñoz, Herausgeber, Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215, Seiten 149–160. NASA, Langley Research Center, Hampton VA 23681-2199, USA. (Zitiert auf den Seiten 43 und 176.)

[102] P. Roy und N. Shankar (2011). *SimCheck: A Contract Type System for Simulink*. Innovations in Systems and Software Engineering, **7**(2), 73–83. ISSN 1614-5046. (Zitiert auf den Seiten 43 und 176.)

[103] J. Rumbaugh, I. Jacobson und G. Booch (2004). *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education. ISBN 0321245628. (Zitiert auf Seite 12.)

[104] S. Sadeghipour und M. Lim (2005). *Einsatz automatischer Testvektorgenerierung im modellbasierten Test*. In A. B. Cremers, R. Manthey, P. Martini, und V. Steinhage, Herausgeber, GI Jahrestagung (2), Band 68 von *LNI*, Seiten 475–479. GI. ISBN 3-88579-397-0. (Zitiert auf den Seiten 30 und 39.)

[105] M. Satpathy, A. Yeolekar und S. Ramesh (2008). *Randomized directed testing (REDIRECT) for Simulink/Stateflow models*. In Proceedings of the 8th ACM international conference on Embedded software, Seiten 217–226. (Zitiert auf Seite 50.)

[106] J. Schäuffele und T. Zurawka (2010). *Automotive Software Engineering*. ATZ-MTZ-Fachbuch, Westdeutscher Verlag GmbH, 4. Auflage. ISBN 9783834893680. (Zitiert auf den Seiten 4 und 13.)

[107] J. Scheible (2012). *Automatisierte Qualitätsbewertung von MATLAB Simulink-Modellen in der Automobil-Domäne*. Dissertation, Universität Tübingen. (Zitiert auf den Seiten 17 und 181.)

[108] K. Sen und G. Agha (2006). *CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools*. In Proceedings of the 18th International Conference on Computer Aided Verification, Seiten 419–423. CAV'06, Springer-Verlag, Berlin, Heidelberg. (Zitiert auf Seite 50.)

[109] K. Sen, D. Marinov und G. Agha (2005). *CUTE: A Concolic Unit Testing Engine for C*. In Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seiten 263–272. ESEC/FSE-13, ACM, New York, NY, USA. ISBN 1-59593-014-0. (Zitiert auf den Seiten 5, 48 und 49.)

[110] A. Spillner und T. Linz (2010). *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard*. dpunkt, Heidelberg, 4. Auflage. ISBN 978-3-89864-358-0. (Zitiert auf Seite 4.)

[111] E. Technologies (2014). *SCADE Suite*. URL `http://www.esterel-technologies.com/products/scade-suite?` Letzter Zugriff: 05/2014. (Zitiert auf Seite 13.)

[112] The Mathworks (2013). *MATLAB and Simulink Deployed to International Space Station for NASA SPHERES Project*. URL `http://www.mathworks.de/company/newsroom/MATLAB-and-Simulink-Deployed-to-International-Space-Station-for-NASA-SPHERES-Project.html`. Letzter Zugriff: 05/2014. (Zitiert auf Seite 5.)

[113] The Mathworks (2014). *Matlab*. URL `http://www.mathworks.de/products/matlab`. Letzter Zugriff: 05/2014. (Zitiert auf Seite 5.)

[114] The Mathworks (2014). *Matlab Simulink*. URL http://www.mathworks.de/products/simulink. Letzter Zugriff: 05/2014. (Zitiert auf den Seiten 5 und 33.)

[115] The Mathworks (2014). *Polyspace*. URL http://www.mathworks.de/products/polyspace. Letzter Zugriff: 05/2014. (Zitiert auf Seite 94.)

[116] The Mathworks (2014). *Simulink Design Verifier*. URL http://www.mathworks.de/products/sldesignverifier. Letzter Zugriff: 05/2014. (Zitiert auf Seite 51.)

[117] F. Tip (1995). *A Survey of Program Slicing Techniques*. Journal of Programming Languages, **3**, 121–189. (Zitiert auf Seite 105.)

[118] P. Tonella (2004). *Evolutionary Testing of Classes*. In Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, Seiten 119–128. ISSTA '04, ACM, New York, NY, USA. ISBN 1-58113-820-2. (Zitiert auf Seite 50.)

[119] Q. M. Tran, B. Wilmes und C. Dziobek (2013). *Refactoring of Simulink Diagrams via Composition of Transformation Steps*. In 8th International Conference on Software Engineering Advances (ICSEA). (Zitiert auf den Seiten 17, 95 und 228.)

[120] S. Varshney und M. Mehrotra (2013). *Search Based Software Test Data Generation for Structural Testing: A Perspective*. SIGSOFT Software Engineering Notes, **38**(4), 1–6. ISSN 0163-5948. (Zitiert auf Seite 46.)

[121] T. E. J. Vos, A. I. Baars, F. F. Lindlar, P. M. Kruse, A. Windisch und J. Wegener (2010). *Industrial Scaled Automated Structural Testing with the Evolutionary Testing Tool*. In Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, Seiten 175–184. ICST '10, IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-3990-4. (Zitiert auf Seite 62.)

[122] T. E. J. Vos, A. I. Baars, F. F. Lindlar, A. Windisch, B. Wilmes, H. Gross, P. M. Kruse und J. Wegener (2012). *Industrial Case Studies For Evaluating Search Based Structural Testing*. International Journal of Software Engineering and Knowledge Engineering, **22**(08), 1123–1149. (Zitiert auf Seite 46.)

[123] T. E. J. Vos, F. F. Lindlar, B. Wilmes, A. Windisch, A. I. Baars, P. M. Kruse, H. Gross und J. Wegener (2013). *Evolutionary Functional Black-Box Testing in an Industrial Setting*. Software Quality Journal, **21**(2), 259–288. ISSN 0963-9314. (Zitiert auf Seite 47.)

[124] M. A. Vouk (1990). *Back-to-back testing*. Information and Software Technology, **32**(1), 34–45. ISSN 0950-5849. Special Issue on Software Quality Assurance. (Zitiert auf Seite 39.)

[125] Y. Wang, Y. Gong, J. Chen, Q. Xiao und Z. Yang (2008). *An Application of Interval Analysis in Software Static Analysis*. In Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing - Volume 02, Seiten 367–372. EUC '08, IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-3492-3. (Zitiert auf den Seiten 73 und 94.)

[126] J. Wegener, A. Baresel und H. Sthamer (2001). *Evolutionary Test Environment for Automatic Structural Testing*. Information and Software Technology, **43**(14), 841–854. ISSN 0950-5849. (Zitiert auf Seite 46.)

[127] J. Wegener, K. Buhr und H. Pohlheim (2002). *Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing*. In Proceedings of the Genetic and Evolutionary Computation Conference, Seiten 1233–1240. GECCO '02, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-878-8. (Zitiert auf den Seiten 46 und 48.)

[128] K. Weicker (2007). *Evolutionäre Algorithmen*. Leitfäden der Informatik, Vieweg+Teubner Verlag. ISBN 9783835102194. (Zitiert auf Seite 54.)

[129] M. Weiser (1981). *Program Slicing*. In Proceedings of the 5th International Conference on Software Engineering, Seiten 439–449. ICSE '81, IEEE Press, Piscataway, NJ, USA. ISBN 0-89791-146-6. (Zitiert auf Seite 105.)

[130] B. Wilmes und A. Windisch (2010). *Considering Signal Constraints in Search-Based Testing of Continuous Systems*. In 2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), Seiten 202–211. (Zitiert auf den Seiten 47 und 51.)

[131] A. Windisch (2009). *Search-based Testing of Complex Simulink Models Containing Stateflow Diagrams*. In 31st International Conference on Software Engineering, Seiten 395–398. (Zitiert auf den Seiten 47 und 62.)

[132] A. Windisch (2011). *Suchbasierter Strukturtest für Simulink Modelle*. Dissertation, TU Berlin. (Zitiert auf den Seiten 6, 33, 47, 48, 51, 53, 56, 58, 60, 61, 63, 64, 70, 141, 179, 182, 183, 193, 198, 221 und 225.)

[133] A. Windisch und N. Al Moubayed (2009). *Signal Generation for Search-Based Testing of Continuous Systems*. In International Conference on Software Testing, Verification and Validation Workshops, Seiten 121–130. ICSTW '09. (Zitiert auf den Seiten 47, 51, 59, 62 und 143.)

[134] T. Xie, N. Tillmann, P. de Halleux und W. Schulte (2009). *Fitness-Guided Path Exploration in Dynamic Symbolic Execution*. In Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009), Seiten 359–368. (Zitiert auf Seite 70.)

[135] Y. Zhan und J. Clark (2004). Search based automatic test-data generation at an architectural level. In K. Deb, Herausgeber, Genetic and Evolutionary Computation – GECCO 2004, Band 3103 von *Lecture Notes in Computer Science*, Seiten 1413–1424. Springer Berlin Heidelberg. ISBN 978-3-540-22343-6. (Zitiert auf den Seiten 47, 48 und 62.)

[136] Y. Zhan und J. A. Clark (2006). *The State Problem for Test Generation in Simulink*. In Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, Seiten 1941–1948. GECCO '06, ACM, New York, NY, USA. ISBN 1-59593-186-4. (Zitiert auf den Seiten 48 und 62.)

[137] Y. Zhan und J. A. Clark (2008). *A Search-Based Framework for Automatic Testing of Matlab/Simulink Models*. Journal of Systems and Software, **81**(2), 262–285. ISSN 0164-1212. (Zitiert auf den Seiten 47, 48 und 160.)

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

ABKÜRZUNGSVERZEICHNIS

In diesem Verzeichnis sind die verwendeten Abkürzungen und Kurzschreibweisen, gegebenenfalls mit Pluralform, aufgeführt. Zu jedem Eintrag ist hierbei die Seite gelistet, auf welcher der dazugehörige Begriff erstmals genannt oder erklärt ist.

ASPO		Alternierende Signalparameteroptimierung	141
AVM		Alternating Variable Method	54
CC		Bedingungsüberdeckung	31
CG	CGs	Überdeckungsziel	32
CGSeq		Suchzielpriorisierung	107
CSC		Subsystem-Bedingungsüberdeckung	33
CSP		Constraint-Satisfaction-Problem	162
DC		Entscheidungsüberdeckung	31
GS		Globale Suche	52
HC		Hill Climbing	54
KFG		Kontrollflussgraph	31
LGA		Längenvariabler genetischer Algorithmus	60
LS		Lokale Suche	52
MC/DC		Modified Condition/Decision Coverage	32
ML		Matlab	5
PAP		Programmablaufplan	31
SAT		Erfüllbarkeitsproblem der Aussagenlogik	162
SDA		Modelleingang-Abhängigkeitsermittlung	96
SF		Stateflow	16
SG	SGs	Suchziel	130
SIA		Überdeckungsziel-Erreichbarkeitsprüfung	69
SL		Simulink	14
SMT		Satisfiability Modulo Theories	162
TASMO		Testing via Automated Search for Models	179
TDGP	TDGPs	Testdatengenerierungsproblem	166
TL		TargetLink	16
VCG	VCGs	Virtuelles Überdeckungsziel	122

EIDESSTATTLICHE ERKLÄRUNG

Die selbständige und eigenhändige Anfertigung dieser Arbeit versichere ich an Eides statt.

Berlin, März 2015

_____

Benjamin Wilmes