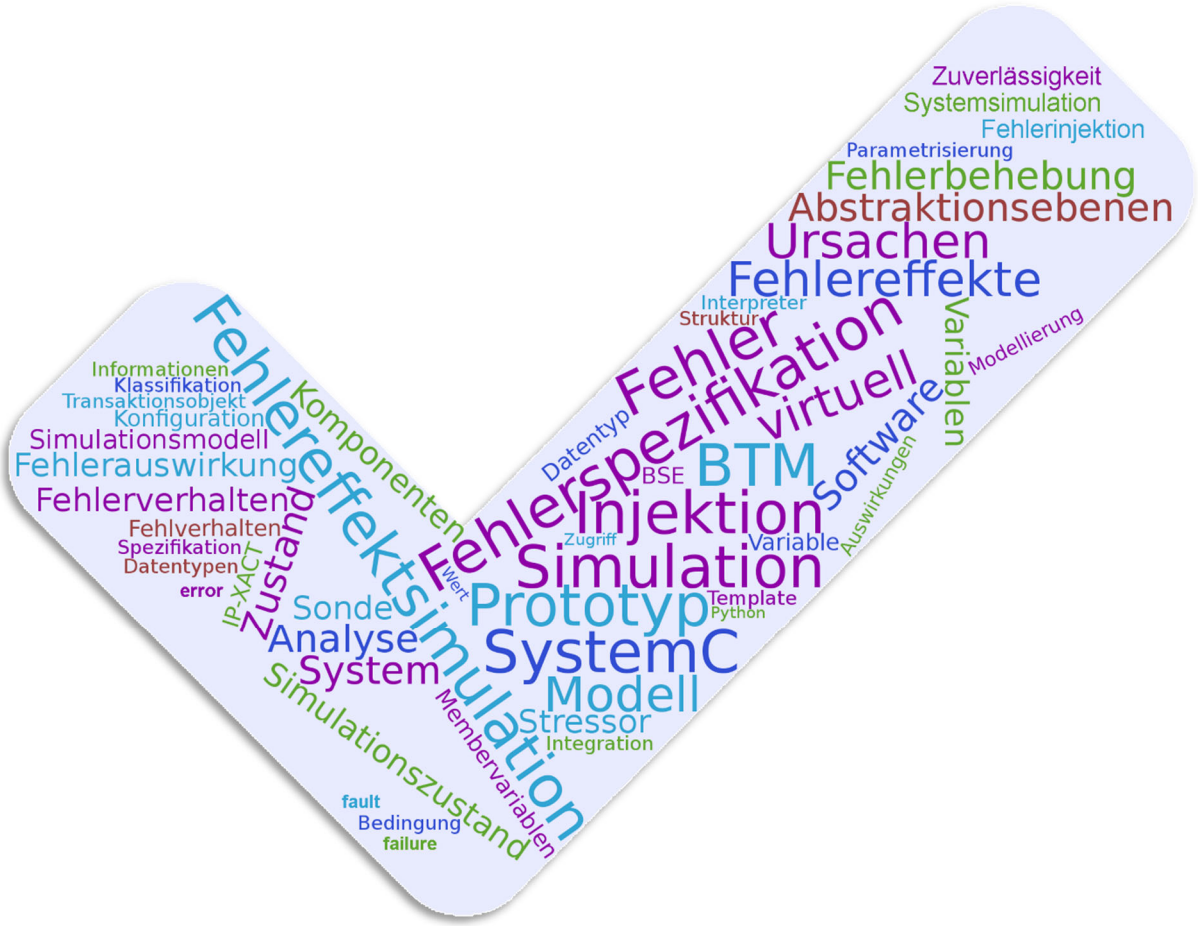


Zuverlässigkeitsbewertung von vernetzten eingebetteten Systemen mittels Fehlereffektsimulation

Sebastian Reiter

Zuverlässigkeitsbewertung von vernetzten eingebetteten Systemen mittels Fehlereffektsimulation

Dissertation
der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Sebastian Reiter
aus Malsch

Tübingen
2018

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	23.11.2018
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Oliver Bringmann
2. Berichterstatter:	Prof. Dr. Wolfgang Rosenstiel

Danksagung

Die vorliegende Arbeit entstand neben meiner Tätigkeit als wissenschaftlicher Mitarbeiter in der Abteilung Systementwurf in der Mikroelektronik (SiM) des Forschungsbereichs Intelligent Systems and Production Engineering (ISPE) am FZI Forschungszentrum Informatik.

Ich möchte an dieser Stelle allen danken, die mich während dieser Zeit unterstützt und so die Entstehung dieser Arbeit ermöglicht haben.

Mein besonderer Dank gilt Herrn Prof. Dr. Oliver Bringmann für die Betreuung und Unterstützung meiner Arbeit, sowie für die Übernahme des Referats dieser Dissertation. Herrn Prof. Dr. Wolfgang Rosenstiel danke ich für die Übernahme des Korreferats und seine sorgfältige Begutachtung der Arbeit.

Weiterhin danke ich allen Kolleginnen und Kollegen am FZI und der Universität Tübingen, sowie allen Studenten, die direkt oder indirekt an dieser Arbeit beteiligt waren. Insbesondere danke ich Herrn Dr. Alexander Viehl für die langjährige Zusammenarbeit sowie für die gewinnbringenden Diskussionen. Nicht zuletzt durch diese zahlreichen Gespräche und sein großes Fachwissen bekam ich wertvolle Hinweise und Anregungen, die mir bei der Erarbeitung und Durchführung meines Themas hilfreich waren.

Schließlich danke ich meiner Familie, die mich während meines Studiums und meiner gesamten Doktorandenzeit stets darin bestärkt und unterstützt hat, meinen Weg zu gehen.

Karlsruhe, Dezember 2018

Sebastian Reiter

Kurzfassung

Die Bedeutsamkeit von eingebetteten Systemen steigt kontinuierlich. Dies zeigt sich bereits anhand ihrer hohen Verbreitung. Neben der reinen Anzahl steigt zusätzlich die Komplexität der einzelnen Systeme. Dies resultiert nicht nur in einem steigenden Entwurfsaufwand, sondern betrifft zusätzlich den Analyseaufwand. Hierbei ist zu beachten, dass die Systeme vermehrt sicherheitsrelevante Aufgaben übernehmen. Ein anschauliches Beispiel stellen Systeme zur Fahrerassistenz bzw. Fahrzeugautomatisierung dar. Durch den rasanten Fortschritt in den letzten Jahren, wird erwartet, dass diese Systeme in den nächsten Jahren bereits hochautomatisiertes Fahren ermöglichen. Für solche Systeme bedeutet ein Ausfall bzw. falsch erbrachter Dienst schwerwiegende Folgen für die Umwelt und Personen im Umfeld. Eine Sicherheitsbewertung ist zwingend vorgeschrieben. Die hohe Vernetzung der einzelnen Systeme bedingt, dass eine isolierte Betrachtung nicht mehr ausreichend ist. Deshalb muss die Analyse neben der gestiegenen Komplexität der einzelnen Systeme zusätzlich die Interaktionen mit weiteren Systemen beachten. Aktuelle Standards empfehlen zur Sicherheitsbewertung häufig Verfahren wie Brainstorming, Fehlermöglichkeits- und Fehlereinflussanalysen oder Fehlerbaumanalysen. Der Erfolg dieser Verfahren ist meist sehr stark von den beteiligten Personen geprägt und fordert ein umfassendes Systemwissen. Die beteiligten Personen müssen die zuvor beschriebene erhöhte Komplexität und Vernetzung beachten und analysieren.

Diese Arbeit stellt einen Ansatz zur Unterstützung der Sicherheitsbewertung vor. Ziel ist, das benötigte Systemwissen von den beteiligten Personen, auf ein Simulationsmodell zu übertragen. Der Anwender ermittelt anhand des Simulationsmodells die systemweiten Fehlereffekte. Die Analyse der Fehlerpropagierung bildet die Grundlage der traditionellen Sicherheitsanalysen. Da das Simulationsmodell die Systemkomplexität und die Systemabhängigkeiten beinhaltet, reduzieren sich die Anforderungen an die beteiligten Personen und folglich der Analyseaufwand. Um solch ein Vorgehen zu ermöglichen, wird eine Methode zur Fehlerinjektion in Simulationsmodelle vorgestellt. Hierbei ist vor allem die Unterstützung unterschiedlicher Abstraktionsgrade, insbesondere von sehr abstrakten System-Level-Modellen, wichtig. Des Weiteren wird ein Ansatz zur umfassenden Fehlerspezifikation vorgestellt. Der Ansatz ermöglicht die Spezifikation von Fehlerursachen auf unterschiedlichen Abstraktionsebenen sowie die automatisierte Einbringung der Fehler in die Simulation. Neben der Einbringung der Fehler bildet die Beobachtung und Analyse der Fehlereffekte weitere wichtige Aspekte. Eine modellbasierte Spezifikation rundet den Ansatz ab und vereinfacht die Integration in einen modellgetriebenen Entwurf.

Abstract

The acceptance and therefore the amount of embedded systems are continuously increasing. Additionally, an increased complexity and interconnection of the individual systems can be observed. These developments do not only affect the effort for system development but also the effort for system analysis. Taking into account that an increasing amount of these systems realizes safety critical tasks, the analysis is an important factor. An illustrative example are the advanced driver assistance systems. These systems made considerable progress over the past years and already enable highly automated driving. With these systems, a failure or a wrongly delivered service can have significant negative effects on people or the environment. A safety assessment is therefore a mandatory task. However, because of the highly interconnection of the different systems, an isolated analysis of a single system is not sufficient. Therefore, not only the increased complexity of the single system, but also the interconnection with other systems have to be taken into account during analysis. Current standards often recommend techniques such as brainstorming, failure mode and effects analysis or fault tree analysis for safety assessment. The success of these techniques depends strongly on the involved people and their knowledge about the system. They have to assess the increased complexity and interconnections during the safety assessment.

This thesis presents an approach to support current safety assessments. The required system knowledge is transferred from the involved people to a simulation model. Basis for the safety assessment is the determination of the effects of a fault in the system. By the provision of a simulation model, the error effects are automatically determined. Meaning the system complexity and the system dependencies are described automatically by the system model. This work presents an approach for fault injection into simulation models to enable such an assessment. One important aspect is the support of different abstraction levels, especially the support of abstract system-level models. Additionally, a general approach for fault specification is presented. This approach enables the specification of faults, with regard to different abstraction levels, the automatic interpretation and control of the fault injectors. Besides the injection of faults, the effect monitoring and classification is an important part of the work. A model-driven specification that easies the integration of the approach into a model-driven design flow completes the overall approach. Provided approaches for code generation reduces the effort to execute the error effect simulation.

Inhaltsverzeichnis

Kurzfassung iii

Abstract v

Abkürzungsverzeichnis xi

1 Einleitung 3
- 1.1 Motivation . . . 3
- 1.2 Ziel der Arbeit . . . 6
- 1.3 Aufbau der Arbeit . . . 8

2 Entwurf verlässlicher Systeme 9
- 2.1 Aspekte verlässlicher Systeme . . . 9
  - 2.1.1 Terminologie verlässlicher Systeme . . . 9
  - 2.1.2 Fehlerbeschreibung . . . 12
- 2.2 Systeme und Modelle . . . 13
  - 2.2.1 Ereignisdiskrete Systeme . . . 13
  - 2.2.2 Systembestandteile . . . 16
  - 2.2.3 Timed Automata . . . 18
  - 2.2.4 Computation Tree Logic . . . 19
- 2.3 Systemsimulation . . . 20
  - 2.3.1 Simulationssprache SystemC . . . 20
  - 2.3.2 Ergänzende Simulationsumgebungen . . . 24
- 2.4 Abstraktionsebenen . . . 24
- 2.5 Modellgetriebener Entwurf . . . 28
  - 2.5.1 Unified Modeling Language . . . 29
  - 2.5.2 IP-XACT . . . 32

3 Existierende Methoden zur Zuverlässigkeitsbewertung 33
- 3.1 Normen und Methoden . . . 33
- 3.2 Testmethode: Fehlerinjektion . . . 39
- 3.3 Modelbasierte Sicherheitsbewertung . . . 45
- 3.4 Fehlerspezifikation . . . 47
- 3.5 Defizite des Stands der Wissenschaft und Technik . . . 47

4 Zuverlässigkeitsbewertung mittels Fehlereffektsimulation 51
4.1 Bestandteile der Fehlereffektsimulation 54

5 Fehlereffektsimulation 57
5.1 Systemsimulation 58
5.1.1 Transaktionsbasierte Makroarchitektur 59
5.1.2 Modulare, parametrisierbare Mikro-Architektur 63
5.1.3 Spezifikation der Systemsimulation 64
5.2 Fehlerinjektion 68
5.2.1 Fehlerinjektor 69
5.2.2 Ergänzende Verwendung von Fehlerinjektoren 79
5.2.3 Wertesonde 80
5.2.4 Fehlerinjektionsablauf 81
5.3 Injektionssteuerung 82
5.3.1 Fehlerverhaltensspezifikation 83
5.3.2 Interpretation und Steuerung 89
5.4 Fehlereffektbeobachtung 94
5.4.1 Erweiterung zur Fehlerdetektion 95

6 Grafische Notation der Fehlereffektsimulation 103
6.1 Spezifikation der Simulationseinheitenbibliothek 105
6.2 Spezifikation der Fehlereffektsimulation 108
6.3 Fehlerspezifikation 110

7 Systemanalyse mittels Fehlereffektsimulation 113
7.1 Analysemethodik 113
7.1.1 Single Fault – Single System 114
7.1.2 Multiple Fault – Single System 116
7.1.3 Single/Multiple Fault – Multiple System 117
7.2 Fallbeispiele 118
7.2.1 Untersuchte Systemarchitekturen 120
7.2.2 Performanzbewertung am FB1 124
7.2.3 Softwarefehlereffekte am FB2 129
7.2.4 Zuverlässigkeitsbewertung am FB3 136
7.2.5 Bewertung der Fallstudien 154

8 Zusammenfassung und Ausblick 157
8.1 Ausblick 159

Literaturverzeichnis 161

A Übersicht Modultests 171
A.1 Injektion via C++ Modellierungsprimitiven 171
A.1.1 [MT1.1] Zeitbedingter, transienter Fehler 172
A.1.2 [MT1.2] Zeitbedingter, intermittierender Fehler 173

A.1.3 [MT1.3] Synchronisierte Fehler 174
A.1.4 [MT1.4] Wertebedingter, transienter Fehler 175
A.1.5 [MT1.5] Zeitbedingter, permanenter Fehler 176
A.1.6 [MT1.6] Permanenter, adaptiver Fehler 177
A.1.7 [MT1.7] Rücksetzverhalten: Original Wert 178
A.1.8 [MT1.8] Rücksetzverhalten: Freigabe der Injektion 179
A.2 Injektion via aggregierte C++-Modellierungsprimitiven 180
A.2.1 [MT2.1] Fehlerinjektion in eine Array-Datenstruktur 181
A.2.2 [MT2.2] Fehlerinjektion auf einzelne Elemente eines Arrays . 182
A.2.3 [MT2.3] Randomisierte Fehlerinjektion in Arrays 183
A.3 Injektion via SystemC Modellierungsprimitiven 185
A.3.1 [MT3.1] Injektion in SystemC Signale 185
A.3.2 [MT3.2] Injektion in Transaktionsobjekte 186

Abkürzungsverzeichnis

ASIL	Automotive Safety Integrity Level
BSE	Basis-Simulationseinheit
BTM	Behavioral Threat Model
CAN	Controller Area Network
CFT	Component Fault Tree
COTS	Commercial off-the-shelf
CPS	Cyber-physisches System
CRC	Cyclic Redundancy Check
CTL	Computation Tree Logic
DEDS	Discrete Event Dynamic Systems
DUT	Design under Test
E/E/PE-System	elektrische, elektronische oder programmierbare elektronische Systeme
EAST-ADL	Electronics Architecture and Software Technology - Architecture Description Language
EMF	Eclipse Modeling Framework
Euro NCAP	European New Car Assessment Programme
FIM	Fault Injection Module
FIT	Failure-In-Time
FMEA	Failure Mode and Effects Analysis
FMECA	Failure Mode, Effects and Criticality Analysis
FMEDA	Failure Mode, Effects and Diagnostic Analysis
FPGA	Field Programmable Gate Array
FTA	Fault Tree Analysis
HFT	Hardware Fault Tolerance
HMI	Human-Machine-Interface
IP	Internet Protocol
ISS	Instruction Set Simulator
LAA	Logischen Anwendungsadresse
LTL	Linear Temporal Logic
MARTE	Modeling and Analysis of Real-time and Embedded Systems
MBSA	Model-Based Safety Assessment
MHP	MOST High Protocol
MIPS	Microprocessor without interlocked pipeline stages

MOF	Meta-Object-Facility
MOST	Media Oriented Systems Transport
MTTF	Mean Time To Failure
MTTR	Mean Time To Repair
NEFZ	Neue Europäische Fahrzyklus
OCL	Object Constraint Language
OEM	Original Equipment Manufacturer
OMG	Object Management Group
OVCL	Object Variant Constraint Language
PIM	plattformunabhängigen Modell
PSM	plattformspezifischen Modell
QVT	Query/View/Transformation
RBD	Reliability Block Diagram
RPN	Risk Priority Number
RTL	Register-Transfer-Level
SE	Software Engineering
SEB	Simulationseinheitenbibliothek
SER	Simulationseinheitenregister
SEU	Single Event Upset
SFF	Safe Fail Fraction
SFI	Simulation Fault Injection
SG	Steuergerät
SIL	Safety Integrity Level
SMT	Satisfiability Modulo Theories
SoC	System-on-a-Chip
SVM	Support Vector Machine
SysML	Systems Modeling Language
TA	Timed Automata
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
TLM	Transaction Level Modeling
UDP	User Datagram Protocol
UML	Unified Modeling Language
UVM	Universal Verification Methodology
VCD	Value Change Dump
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VoIP	Voice over IP
XML	Extensible Markup Language

Kapitel 1

Einleitung

1.1 Motivation

Eingebettete Systeme stellen eine Schlüsseltechnologie zur Bewältigung gesellschaftlicher und ökonomischer Herausforderungen dar, wie die Studie [42] des ZVEI zeigt. Die Verbreitung und Akzeptanz eingebetteter Systeme stieg über die letzten Jahre kontinuierlich. Sowohl das alltägliche Leben als auch industrielle Produktionsprozesse sind ohne eingebettete Systeme kaum mehr vorstellbar.

Die Automobilindustrie bietet hierfür viele anschauliche Beispiele. An ihr lässt sich sehr eindrucksvoll die Bedeutung aber auch die Herausforderungen dieser Entwicklungen darlegen. In heutigen Fahrzeugen übernehmen elektronische Systeme vermehrt Aufgaben, die erst durch Entwicklungen im Bereich der Elektronik möglich sind bzw. zuvor durch mechanische Systeme realisiert wurden. Charette beleuchtet in [27], dass Computersysteme ungefähr 80 % der Innovationen im Fahrzeug ermöglichen. Im Jahr 2016 verfügten Fahrzeuge über durchschnittlich 30 unterschiedliche programmierbare, elektronische Systeme wie unterschiedliche Quellen [27, 36, 121] dokumentieren. Premiummodelle verfügen bereits über 100 ECUs wie in [72] dargelegt ist. Sie realisieren Funktionen zur Steigerung des Komforts, der Energieeffizienz aber auch der Sicherheit. Um die höchste Bewertung mit fünf Sternen bei dem European New Car Assessment Programme (Euro NCAP) zu bekommen, müssen Neufahrzeuge mit praxisgerechter Unfallvermeidungstechnologie ausgestattet sein. Bei dem Euro NCAP handelt es sich um eine Gesellschaft, die anhand von Crashtests als auch anhand der vorhandenen Sicherheitssysteme die Sicherheit neuer Fahrzeuge bewertet. Zu den geforderte Unfallvermeidungstechnologie zählt seit 2014 ein Notbremsassistent, genauer ein autonomer Notbremsassistent für außerhalb der Ortschaft, und seit 2016 ein Notbremsassistent mit Fußgängererkennung [104]. Dies verdeutlicht den rasanten Einzug und die Akzeptanz von Assistenzsystemen, die nur mit Hilfe von programmierbaren, elektronischen Systemen realisiert werden können.

Neben der reinen Anzahl steigt, getrieben von Moore's Law, zusätzlich die Leistungsfähigkeit einzelner Verarbeitungs- und Kommunikationssysteme kontinuierlich. So auch in der Automotive-Domäne. Viele Funktionen, von z. B. Infotainment- aber auch Assistenzsysteme, sind lediglich durch die gestiegene Leistungsfähigkeit

der programmierbaren, elektronischen Systeme zu realisieren. Hierunter fallen z. B. Assistenzsystem wie adaptive Geschwindigkeitsregelung, Fahrspurerkennung, intelligente Einparkhilfeassistenz oder Kollisionsvermeidung. Der Softwareanteil im aktuellen Fahrzeug übersteigt den einer Boeing 787 um das Fünfzehnfache wie Charette in [27] zeigt. Einhergehend mit dem gestiegenen Funktionsumfang elektronischer Systeme entwickelt sich auch die Komplexität. Die reduzierten Strukturgrößen ermöglichen die Steigerung der Komplexität einzelner Systeme bzw. Bauteile. Als Folge steigt die Integrationsdichte der Systeme. Einzelne Chips bzw. Bauteile realisieren komplexe Systemarchitekturen mit On-Chip-Bussen, mehreren Prozessoren sowie unterschiedlicher Peripherie.

Ein weiterer Aspekt ist die gestiegene Vernetzung der Systeme. Die Funktionalität eines Systems ist von einer hohen Interaktion mit anderen Systemen geprägt. Die aktuelle Fahrgeschwindigkeit regelt z. B. die Lautstärke des Infotainment- und Navigationssystems und bedingt damit eine Vernetzung der unterschiedlichen Systeme. Durch eine schrittweise Einführung zusätzlicher Assistenz- bzw. Fahrzeugautomatisierungssysteme sowie einer stärkeren Vernetzung der Systeme, soll in naher Zukunft hochautomatisiertes Fahren möglich sein. Laut einer Studie des Bundeswirtschaftsministeriums [22] soll bereits 2020 hochautomatisierte Fahrfunktionen auf Autobahnen Marktreife erreicht haben. Nach der Definition der Bundesanstalt für Straßenwesen [39] übernimmt bei hochautomatisiertem Fahren das System die komplette Quer- und Längsführung, wobei der Fahrer jederzeit zur vollständigen Fahrzeugübernahme bereit sein muss. Eine weitere Studie von Frost & Sullivan [37] erwartet für 2025 allein 3,5 Millionen Fahrzeuge weltweit mit hochautomatisierten Fahrfunktionen. Eine hierfür wichtige Entwicklung basiert auf Algorithmen zur Sensorfusion. Hierbei beruht eine Berechnung, z. B. die Fahrsituationserkennung oder die Fahrstrategieentwicklung, auf einer Vielzahl von unterschiedlichen Sensordaten. Für die Zukunft kann angenommen werden, dass die Anzahl elektronischer Systeme, die Komplexität der einzelnen Systeme als auch ihre Vernetzung weiter steigen wird.

Diese Entwicklungen haben nicht nur schwerwiegende Auswirkungen auf den Entwurf der Systeme, sondern auch auf die Qualifikation der einzelnen Systeme bzw. des Gesamtsystems. Laut einer Studie des Bundeswirtschaftsministeriums [22] wird das Validieren und Testen mit 16% an den Gesamtkosten, eines Systems für hochautonomes Fahren, einen signifikanten Anteil in der Wertschöpfung im Jahr 2020 haben. Dies stellt den drittgrößten Anteil nach der Entwicklung und Herstellung der Umfeldsensorik und der Steuergeräte dar. Wie die genannten Beispiele zeigen, sind viele der durch elektronische Systeme übernommenen Aufgaben sicherheitskritisch. Ein Ausfall bzw. falsch erbrachter Dienst kann schwerwiegende Folgen für die Umwelt und Personen im Umfeld des Systems haben. Dies bedeutet, dass neben der reinen funktionalen Qualifikation, die Sicherheitsbewertung eine wichtige Rolle spielt.

Die gestiegene Komplexität und Vernetzung hat gravierende Auswirkungen auf die Entwicklung sicherheitskritische Systeme. Ein Aspekt ist die gestiegene Fehleranfälligkeit der Systeme. Eine kleinere Strukturgröße steigert die Anfälligkeit für Single Event Upsets (SEUs), hervorgerufen z. B. durch ionisierte Teilchen, wie Wang et al. in [116] beleuchtet haben. Die Interaktion der Systeme bedingt, dass ein Aus-

fall eines beteiligten Systems zu einem kritischen Zustand des sicherheitsrelevanten Gesamtsystems führen kann. Zusätzlich kann es bei dem Datenaustausch durch Übersprechen und elektromagnetische Strahlung zu einer Korruption der Daten kommen, was die Fehleranfälligkeit weiter steigert. Die Sicherheitsbewertung muss diese Zusammenhänge analysiert und alle kritischen Zustände erkennen. Eine isolierte Betrachtung der Systeme ist meist nicht ausreichend. Zur Sicherheitsbewertung ist eine ganzheitliche Systembetrachtung notwendig. Die steigende Fehleranfälligkeit sowie die Anforderung zur Betrachtung des komplexen Gesamtsystems erhöhen den Aufwand der Sicherheitsanalyse erheblich. Winner und Wachenfeld schätzen in [119] dass ungefähr 100 Millionen Testkilometer zurückgelegt werden müssen, um nachzuweisen, dass hochautomatisierte Fahrzeuge so sicher sind wie manuell gesteuerte Fahrzeuge.

Ein weiterer Aspekt bei der Betrachtung des Gesamtsystems ist, dass durch die Komplexität der Teilsysteme die Entwicklung häufig auf mehrere Zulieferer verteilt wird. Die Zulieferer ihrerseits kaufen wiederum Systemteile zu. Das heißt, es entsteht eine Kette von Systemteileentwicklern, die eine sogenannte Wertschöpfungskette bilden. Der Produkthersteller (engl. Original Equipment Manufacturer (OEM)) agiert häufig vorrangig als Systemintegrator. Dieser Systemintegrator muss aber die Sicherheitsbewertung des Gesamtsystems durchführen. Wohingegen die Entwickler der Teilsysteme und demzufolge die Experten für diese Teilsysteme die Zulieferer sind.

Gängige Techniken zur Sicherheitsbewertung basieren heutzutage auf manuellen Prozessen. Hierbei ermitteln Experten alle erdenklichen Fehler und deren Ursachen. Anschließend bewerten Sie die Auswirkungen dieser Fehler und die Wirksamkeit von Absicherungsmaßnahmen. Das Ziel ist es die funktionale Sicherheit eines Systems zu bewerten. Der Begriff funktionale Sicherheit drückt aus, dass Systeme ihre sicherheitsrelevanten Funktionen zuverlässig bzw. mit vertretbarem Ausfallrisiko erbringen. Er wird zur Abgrenzung von weiteren Sicherheitsaspekten wie elektrische Sicherheit, Brandschutz oder Strahlenschutz verwendet. Die Bewertung findet meistens in den finalen Qualifizierungsprozessen statt. Die eingesetzten Methoden sind sehr stark von den Fähigkeiten und Erfahrungen der beteiligten Personen abhängig. Die Identifikation und Verknüpfung von Fehlerauswirkungen und Fehlerursachen beruhen häufig auf einer systematischen, aber nicht wohldefinierten, Sammlung von Erfahrungswissen. Das Vorgehen wird erschwert, da die Bewertung unterschiedliche Personengruppen wie Sicherheitsingenieure, Software- oder Hardwareentwickler miteinbeziehen muss. Zusätzlich kommen die benötigten Systemexperten häufig nicht aus dem eigenen Unternehmen, sondern von Drittanbietern wie z. B. Zulieferern oder externe Dienstleistern. Vor allem in der Automobilindustrie tritt dies häufig auf, wenn der Fahrzeughersteller vorrangig als Systemintegrator agiert, anstelle als Systementwickler. Um die steigende Komplexität handhaben zu können, ist eine Methodik zum Erfassen und Analysieren der komplexen Systemabhängigkeiten, zur Reduktion der subjektiven Einschätzungen sowie zur Beteiligung unterschiedlicher Experten notwendig. Eine weitere Herausforderung ist es, die Validierung der Systemzuverlässigkeit so früh wie möglich in den Entwurfsphasen durchzuführen. Hierdurch ist es möglich kostenintensive Entwurfsänderungen zu vermeiden.

1.2 Ziel der Arbeit

Das Ziel der Arbeit ist die Erforschung und Umsetzung einer Methodik, welche die Analyse komplexer, vernetzter, elektronischer Systeme, unter dem Gesichtspunkt der funktionalen Sicherheit, ermöglicht. Zur Bewertung ob ein System seine Sicherheitsfunktionen zuverlässig erbringt, wird eine Systemsimulation des zu analysierenden Systems bereitgestellt. Mit diesem sogenannten *Simulationsmodell* können die Auswirkungen von Fehlern automatisiert bestimmt werden. Ziel ist es entlang des Entwurfsprozesses Fehler im aktuellen Abstraktionsgrad des Systems zu simulieren und die Fehlereffekte zu analysieren. Dies ermöglicht bereits in frühen Entwicklungsphasen, Entwurfsentscheidungen in Bezug auf die funktionale Sicherheit zu bewerten. Abbildung 1.1 verdeutlicht das angedachte Vorgehen am klassischen V-Modell. Beim V-Modell [41] handelt es sich um ein Vorgehensmodell in der Softwareentwicklung, bei welchem die Entwicklung und Qualitätssicherung in Phasen aufgeteilt sind. Viele aktuelle Entwicklungsmodelle basieren auf diesem. Aktuelle Standards wie die ISO 26262 [51] oder die IEC 61508 [46] greifen diese Vorgehensmodelle auf. Die jeweiligen Simulationen stellen die Fehlerauswirkungen in Relation zu den Fehlerursachen. Zu den unterschiedlichen Phasen im V-Modell werden die, für den jeweiligen Detaillierungsgrad passenden, Systemsimulationen angewandt. Wie in Abbildung 1.1 zu erkennen, verfeinern sich die Simulationsmodelle mit dem Voranschreiten der Entwicklung und ermöglichen eine genauere Abschätzung der Fehlerauswirkung.

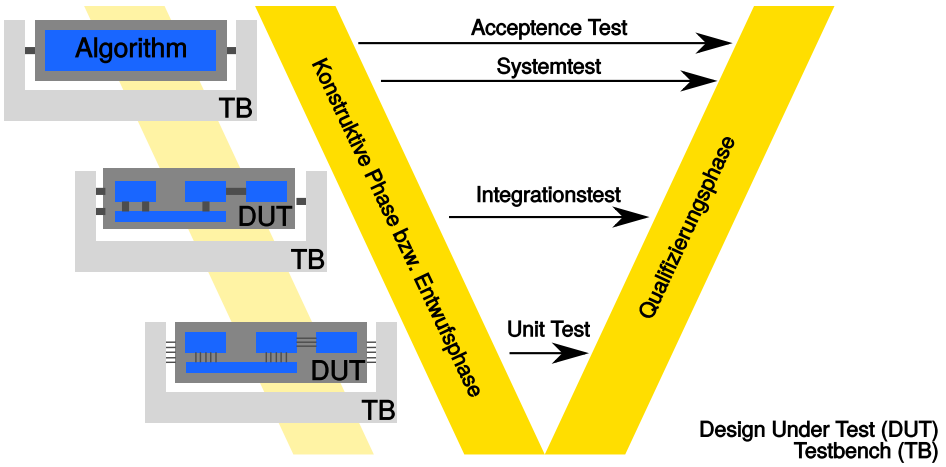


Abbildung 1.1: Durchgängige Fehlereffektbewertung

Während der Entwurfsphase werden Entwurfsentscheidungen getroffen, welche die Fehlerauswirkungen reduzieren oder sogar vermeiden. Im Rahmen des Entwurfs können z. B. redundante Funktionseinheiten hinzugefügt werden, die den Ausfall einzelner Einheiten kompensieren. Ein weiterer Ansatz bilden Fehlerkorrekturmechanismen, z. B. die Vorwärtsfehlerkorrektur [78], welche die Fehlerauswirkungen auf Systemebene abschwächen. Die Auswahl bzw. die Abschätzung der Wirksamkeit werden mithilfe des in dieser Arbeit vorgestellten Ansatzes bewertet. Die Bewertung wird zur Unterstützung sowohl des Systementwurfs als auch der klassischen Quali-

fizierungsprozesse herangezogen. Der Anwender hat ein Simulationsmodell des zu analysierenden Systems, mit dem er die Auswirkungen von Fehlern automatisiert bestimmt werden können. Der Anwender muss die Komplexität des Systems zur Bestimmung der Fehlerauswirkungen nicht mehr manuell beherrschen, sondern kann sich auf das Simulationsmodell stützen. Die Systemkomplexität ist somit im Simulationsmodell hinterlegt. Der Anwender kann sich auf die Identifikation von möglichen Fehlerursachen konzentrieren, anstelle der Herleitung Fehlerauswirkung. Ähnlich verhält es sich mit den Systemabhängigkeiten. Da diese im Simulationsmodell spezifiziert sind, werden sie bei der Fehlereffektbestimmung automatisch berücksichtigt. Ein weiterer Vorteil ist, dass der Systemintegrator die Systemmodelle der Zulieferer in die Analysen einbeziehen kann. Hierdurch ist es möglich die Experten der Zulieferer indirekt an der Sicherheitsanalyse, beim Systemintegrator, zu beteiligen, ohne sie in den eigentlichen Qualifizierungsprozess einzubeziehen. Sie müssen lediglich die benötigten Modelle zur Verfügung stellen. Abbildung 1.2 zeigt die angestrebte Kooperation über die Simulationsmodelle. Die unterschiedlichen Domänenexperten,

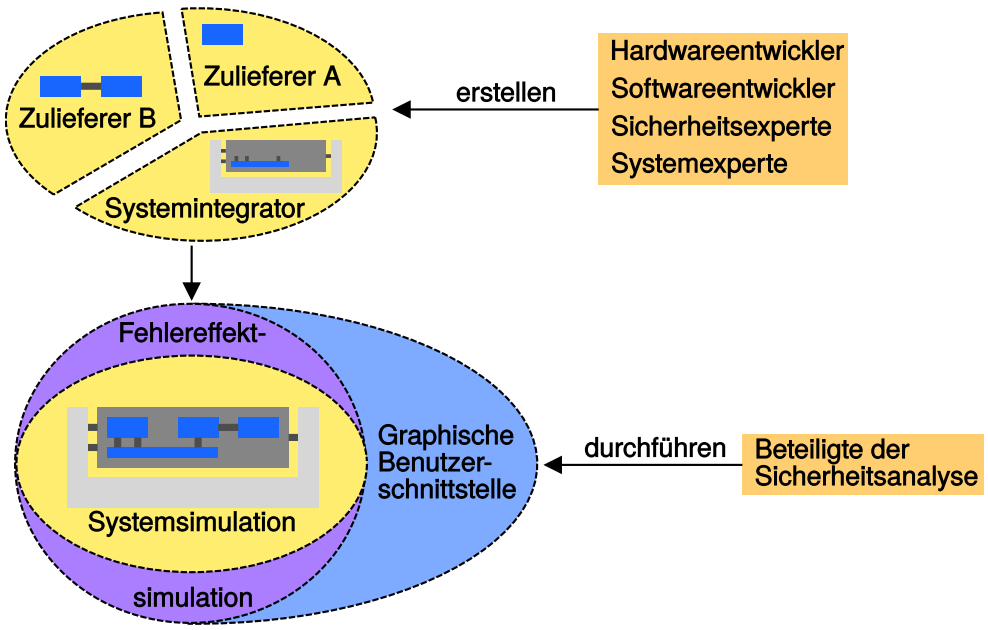


Abbildung 1.2: Angestrebter Analyseprozess

wie z. B. Hardware-, Softwareentwickler, erstellen die Simulationsmodelle bzw. Teile der Simulationsmodelle. Hierbei wird unterstützt, dass die Modelle von unterschiedlichen Zulieferern kommen bzw. sich an der Wertschöpfungskette entlang propagieren. Die in dieser Arbeit vorgestellte Fehlereffektsimulationsumgebung kombiniert die gelieferten Simulationsmodelle und erweitert sie für die Zuverlässigkeitsbewertung. Eine grafische Benutzerschnittstelle dient den beteiligten Sicherheitsexperten als Schnittstelle während des Qualifizierungsprozesses.

In dieser Arbeit wird der Ansatz im Rahmen der Sicherheitsbewertung (engl. safety assessment) motiviert, d. h. im Rahmen der Bewertung ob Sicherheitsziele (engl. safety goals) ausreichend spezifiziert bzw. erreicht werden. Bei der Entwicklung sicher-

heitsrelevanter Systeme wird meist in der Anforderungsphase bereits eine Gefahren- und Risikoanalyse (engl. hazard analysis and risk assessment) durchgeführt. Ziel ist es die sicherheitsrelevanten Teile des Systems zu bestimmen und den Grad der Sicherheitsrelevanz festzulegen. In diesem Prozess werden gefährliche Situationen identifiziert und Sicherheitsziel sowie das zugehörige Sicherheitsintegritätslevel festgelegt. Diese Anforderungen werden im Rahmen der Sicherheitsbewertung verifiziert. Auch im Rahmen der Gefahrenanalyse müssen Fehlerauswirkungen in Relation zu den Fehlerursachen gestellt werden. Der in dieser Arbeit entwickelte Ansatz ist somit auch in dieser Analyse anwendbar. Der Fokus in dieser Arbeit soll aber auf der Sicherheitsbewertung sein.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist wie folgt aufgebaut. Kapitel 2 stellt grundlegende Konzepte zum Entwurf verlässlicher Systeme vor. Neben der allgemeinen Vorstellung der verwendeten Terminologie stellt es Aspekte der in dieser Arbeit angewendeten Konzepte, wie SystemC, die Unified Modeling Language (UML) oder IP-XACT vor. In Kapitel 3 werden aktuelle Ansätze zur Zuverlässigkeitsbewertung dargelegt. Hierbei werden aktuell angewandte Standards und Verfahren sowie aktuelle Forschungsansätze vorgestellt. Bei den Forschungsansätzen liegt der Fokus auf Techniken zur Fehlerinjektion und Fehlerspezifikation. Kapitel 4 fasst den angestrebten Lösungsansatz zusammen, wohingegen Kapitel 5 und Kapitel 6 die einzelnen Bestandteile im Detail vorstellen. Der erste Teil von Kapitel 5 stellt die Simulationsinfrastruktur und die Methodik der Systemsimulation vor. Die weiteren Teile stellen die notwendigen Erweiterungen zur Fehlerinjektion vor. Die optionale Integration in einen modellgetriebenen Entwurfsablauf sowie das entwickelte Spezifikationsmodell werden in Kapitel 6 vorgestellt. Kapitel 7 legt unterschiedliche Vorgehensweisen bei der Fehlereffektsimulation dar. Drei Fallstudien demonstrieren die entwickelten Ansätze. Abgeschlossen wird die Arbeit durch elementare Testfälle, die einzelne Aspekte der Fehlerinjektionsplattform darlegen. Diese elementaren Tests sind im Anhang A vorgestellt.

Kapitel 2

Entwurf verlässlicher Systeme

Im folgenden Kapitel werden die in dieser Arbeit verwendeten Begriffe und gängige Verfahren vorgestellt. Ferner stellt es Ansätze und Techniken vor, welche die Grundlage dieser Arbeit bilden. Abschnitt 2.1 stellt allgemeine Begriffe und Ziele der Zuverlässigkeitsbewertung vor. Der folgende Abschnitt 2.2 zeigt Besonderheiten bei der Analyse ereignisdiskreter Systeme, die den Fokus dieser Arbeit bilden, auf. Ein weiterer Aspekt des Abschnitts sind unterschiedliche Ansätze Systeme zu beschreiben. Im anschließenden Abschnitt 2.3 wird das Konzept der Simulation zur Qualifikation von Systemen vorgestellt. Hierbei spielt die Modellierungssprache SystemC die zentrale Rolle. Nachfolgend werden unterschiedliche Abstraktionsebenen zur Modellierung vorgestellt sowie deren Realisierung in SystemC dargelegt. Abgeschlossen wird dieses Kapitel mit der Vorstellung von Aspekten des modellgetriebenen Softwareentwurfs.

2.1 Aspekte verlässlicher Systeme

Der folgende Abschnitt zeigt die Grundbegriffe verlässlicher Systeme auf. Neben der Schaffung eines einheitlichen Begriffsverständnisses, helfen die eingeführten Begrifflichkeiten das Analyseziel dieser Arbeit zu verdeutlichen. Die in dieser Arbeit verwendeten deutschen Bezeichnungen orientieren sich an den Übersetzungen von Laprie et al. in [65].

2.1.1 Terminologie verlässlicher Systeme

Die Verlässlichkeit (engl. dependability) ist eine nichtfunktionale Eigenschaft eines Systems. Avizienis et al. definieren, in [3], die Verlässlichkeit wie folgt: „Dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable“. Somit ist die Verlässlichkeit ein Maß für die Fähigkeit eines Systems Fehlerauswirkungen bzw. Ausfälle zu vermeiden, die durch ihre Häufigkeit oder ihre Schwere nicht akzeptabel sind. Um eine hohe Verlässlichkeit zu gewährleisten, kann nach dieser Definition, somit entweder die Häufigkeit des Auftretens oder die Schwere der Auswirkung reduziert werden. Ein wichtiger Aspekt

bei der Bewertung der Verlässlichkeit von Systemen ist somit die Abschätzung der Auftrittshäufigkeit.

Nach den Definitionen von Verlässlichkeit in [3, 31], kann der Begriff weiter unterteilt werden. Hierbei wird die Verlässlichkeit eines Systems durch die folgenden Attribute bestimmt:

- Überlebenswahrscheinlichkeit (engl. Reliability):
  Fähigkeit eines Systems, für eine gegebene Zeit korrekt zu arbeiten (engl. „continuity of correct service“ [3]). Häufig verwendete Synonyme sind die Zuverlässigkeit bzw. Funktionszuverlässigkeit.

- Verfügbarkeit (engl. Availability):
  Wahrscheinlichkeit für die korrekte Funktion eines Systems im Sinne der Zuverlässigkeit (engl. „readiness for correct service“ [3]).

- Sicherheit (engl. Safety):
  Die Abwesenheit von katastrophalen Konsequenzen für den Anwender und die Umwelt (engl. „absence of catastrophic consequences on the user(s) and the environment“ [3]).

- Integrität (engl. Integrity):
  Abwesenheit von unzulässigen Systemänderungen (Updates) (engl. „absence of improper system alterations“ [3]). Ein häufig verwendetes Synonym ist die Unversehrtheit.

- Instandhaltbarkeit (engl. Maintainability):
  Fähigkeit einer Einheit, in einem Zustand erhalten bzw. in ihn zurückversetzt zu werden, indem sie eine geforderte Funktion erfüllen kann (engl. „ability to undergo modifications and repairs“ [3]).

Die ISO 26262 [51] konzentriert sich auf den Begriff funktionale Sicherheit (engl. functional safety). Wie der Begriff Verlässlichkeit bezieht sich dieser auf die Abwesenheit eines unvertretbaren Risikos von Gefahren durch ein Fehlverhalten (engl. malfunctioning behavior) des Systems.

Aus diesen Definitionen resultiert, dass für die Verlässlichkeitsanalyse die Detektion möglichen Fehlverhaltens und die Abschätzung der Auftrittswahrscheinlichkeit ein wichtiger Aspekt ist. Das mögliche Fehlverhalten des Systems wird durch die Fehlerauswirkung (engl. failure) beschrieben.

Begriff 1. *Fehlerauswirkung*
Verlust der Fähigkeit eines Systems, die geforderte Funktion (Dienst) zu erfüllen.

Um eine Fehlerauswirkung zu beschreiben, muss immer ein Bezug zu einem System vorhanden sein, da sich eine Fehlerauswirkung immer am externen Zustand (Dienstschnittstelle) des Systems manifestiert. Die Fehlerauswirkung muss nicht mit dem kompletten Verlust der Funktionalität einhergehen, eine Abweichung von der erwarteten bzw. spezifizierten Funktionalität ist ausreichend. Hierbei kann zwischen

einer Veränderung des Werts der Systemausgabe, z. B. der Verfälschung eines Steuersignals oder der zeitlichen Verfälschung der Systemausgabe unterschieden werden. Mögliche Synonyme sind Ausfall (vgl. [65]), Versagen oder Betriebsfehler. Die Fehlerauswirkung wird durch einen im System vorliegenden Fehlerzustand (engl. error) hervorgerufen.

Begriff 2. *Fehlerzustand*
Verfälschung des internen Systemzustands, die sich als Fehlerauswirkung an der Systemschnittstelle manifestieren kann.

Der Fehlerzustand gehört zum internen Zustand $X(t)$ des Systems. Nicht jeder Fehlerzustand hat eine Fehlerauswirkung an den Systemschnittstellen zur Folge. Ein verborgener Fehlerzustand (engl. latent error) wird nicht an die Systemschnittstelle $U(t)$ propagiert und ist somit, von extern, unentdeckt. Der letzte zu untersuchende Baustein ist die Fehlerursache (engl. fault).

Begriff 3. *Fehlerursache*
Erkannte oder vermutete Ursache eines Fehlerzustands [51].

Eine Fehlerursache wird als aktiv bezeichnet, wenn sie zu einem internen Fehlerzustand führt, andernfalls handelt es sich um eine ruhende (engl. dormant) Fehlerursache. Bezogen auf die Dauer wird zwischen einer transienten und permanenten Fehlerursache unterschieden. Eine transiente Fehlerursache tritt für eine gewisse Zeit auf und behebt sich ohne externes Eingreifen ins System. Die Zeitspanne zwischen Auftreten und Beheben wird Erholungszeit (engl. recovery time) genannt. Die permanente Fehlerursache hingegen bleibt bestehen, bis sie durch eine externe Korrekturmaßnahme beseitigt wird. Hierbei spielt die Instandhaltbarkeit eine wichtige Rolle.

Die Dreiteilung in Fehlerursache, Fehlerzustand und Fehlerauswirkung ermöglicht es, eine Fehlerwirkkette zu erstellen. Durch die Aktivierung resultiert die Fehlerursache in einem Fehlerzustand. Der Fehlerzustand kann sich zu einer Fehlerauswirkung an der Systemschnittstelle ausbreiten. Hängen Systeme voneinander ab, z. B. System A_2 benötigt einen Dienst von System A_1, führt dies dazu, dass die Fehlerauswirkung von System A_1 als Fehlerursache in System A_2 angesehen wird. Die Fehlerauswirkung von System A_2 liegt als ruhende Fehlerursache im System A_2 vor, bis System A_2 den bereitgestellten Dienst in Anspruch nimmt, um einen internen Zustand zu ändern (siehe Abbildung 2.1).

Zur detaillierteren Beschreibung unterschiedlicher Fehlerauswirkungen innerhalb eines komplexen Systems ist es ratsam das System in Teilsysteme zu unterteilen, z. B. in die Teilschritte einer Verarbeitungskette. Jedes Teilsystem erbringt einen Dienst und benötigt möglicherweise den Dienst eines anderen Teilsystems. In Abbildung 2.1 sind die Fehlerursache ($FU1_1$), der Fehlerzustand (FZ_1) und die Fehlerauswirkung (FA_1) in Bezug auf System A_1 dargestellt. In Bezug auf das Gesamtsystem, d. h. ohne die Unterteilung in Teilsysteme besteht lediglich die Fehlerursache ($FU1_1$), der interne Fehlerzustand (FZ_1, FZ_2) und die Fehlerauswirkung (FA_2). Somit orientiert sich die Fehlerwirkkette an der Verkettung der (Teil-)Systeme.

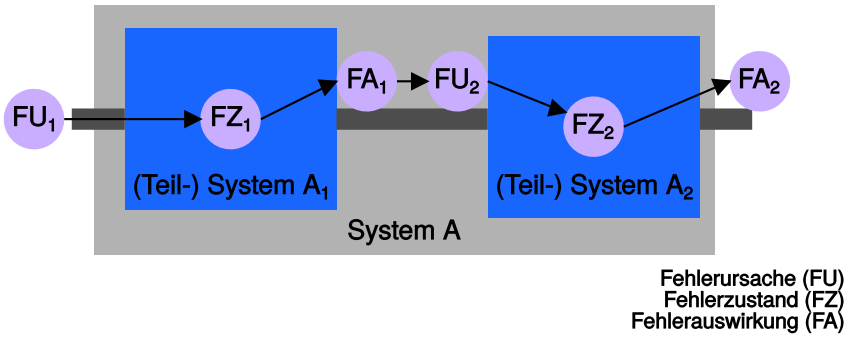


Abbildung 2.1: Struktur der Fehlerwirkkette

2.1.2 Fehlerbeschreibung

Zur Beschreibung von Fehlern (Fehlerursache, Fehlerzustand, Fehlerauswirkung) sind vier wichtige Eigenschaften zu spezifizieren.

1. Fehlerlokation: Welche Komponenten bzw. welche Information ist vom Fehler betroffen.
2. Auftrittszeitpunkt: Wann tritt der Fehler auf.
3. Fehlerwirkung: Welche Wirkung hat der Fehler.
4. Fehlererholungszeit: Wie lange ist der Fehler aktiv.

Fehlerlokation und Fehlerwirkung können auch zusammengefasst werden. Die Spezifikation dieser Eigenschaften erfolgt über unterschiedliche Metriken. Besonders zur Spezifikation des Auftrittszeitpunkts bzw. der Auftrittswahrscheinlichkeit existieren unterschiedliche Metriken. Birolini stellt in [11], im Speziellen auf die Kapitel 1.2.3–1.2.6, unterschiedliche Metriken im Detail vor.

Die Metrik Failure-In-Time (FIT) ist eine Spezifikation der Ausfallrate. Die Ausfallrate wird über die Anzahl der Ausfälle innerhalb eines Zeitintervalls bestimmt. Häufig kommt eine Zeitspanne von 10^9 Stunden zum Einsatz. Aber auch abweichende Zeitintervalle sind möglich. In diesen Fällen wird das Zeitintervall explizit mit angegeben.

Die allgemeinere Angabe der Ausfallrate erfolgt über die Angabe der Ausfälle pro Zeiteinheit. Diese Rate wird meistens mit λ bezeichnet.

Der Kehrwert der Ausfallrate gibt die Zeitspanne des korrekten Betriebs an und wird als mittlere Betriebsdauer bis zum Ausfall (engl. Mean Time To Failure (MTTF)) bezeichnet.

$$\lambda = 1/MTTF$$

Eine weitere häufig vorkommende Metrik ist die Nichtverfügbarkeit U (engl. unavailability). Sie berücksichtigt zusätzlich die mittlere Reparaturzeit (engl. Mean Time To Repair (MTTR)). Bei Auftreten eines permanenten Fehlers kann lediglich ein externes Eingreifen diesen beheben. Diese Dauer wird als MTTR bezeichnet.

$$U = MTTR/(MTTR + MTTF)$$

Die Spezifikation der Auswirkungen eines Fehlers erfolgt häufig über vordefinierte Fehlermodelle wie z. B. das Haftfehler- oder das Brückenfehlermodell [71, 87]. Diese Modelle sind häufig auf den unteren Abstraktionsebenen (siehe Abschnitt 2.4) angesiedelt. Sie entsprechen abstrakten Beschreibungen, welche die Auswirkungen physikalischer Effekte beschreiben. Ihren Ursprung bzw. ihre häufigste Anwendung haben diese Modelle in der Spezifikation von Fehlern in elektronischen Schaltungen. Sie spezifizieren den auftretenden Fehler und müssen lediglich den entsprechenden Stellen im System zugewiesen werden. Die Spezifikation der Dauer eines transienten Fehlers wird selten spezifiziert bzw. in informeller Art und Weise annotiert.

2.2 Systeme und Modelle

Die im Folgenden vorgestellten Grundlagen sind in [24, 102] vertiefend behandelt. Im Zentrum dieser Arbeit steht das zu analysierende System, genauer gesagt dessen Modell. Bei Erreichen eines aussagekräftigen Modells können die Begriffe System und Modell gleichbedeutend verwendet werden, sodass in dieser Arbeit die Begriffe meist synonym verwendet werden. Der Abschnitt motiviert zuerst die Betrachtung von Systeme als ereignisdiskrete Systeme bzw. geht auf eine mögliche Gliederung von Systemen ein. Danach werden unterschiedliche Modelle zur Spezifikation von Systemen vorgestellt.

2.2.1 Ereignisdiskrete Systeme

Die ISO 26262 [51] definiert ein System als ein „set of elements, at least sensor, controller, and actuator, in relation with each other in accordance with a design“. Diese Definition stellt die Betrachtung des Systems als Eingabe-Ausgabe-Modell in den Fokus. Hierbei wird die Systemeingabe, also z. B. die durch die Sensoren erfassten Daten, als Funktionen über die Zeit beschrieben.

$$u(t) = [u_1(t), \ldots, u_p(t)]^T$$

Analog werden auch die Systemausgaben als Funktionen über die Zeit definiert. Es ist naheliegend anzunehmen, dass die Systemausgaben von den Systemeingaben abhängen.

$$y(t) = [g_1(u_1(t), \ldots, u_p(t)), \ldots, g_m(u_1(t), \ldots, u_p(t))]^T$$

Dieses mathematische Modell beschreibt *statische Systeme*. Bei statischen Systemen ist die Systemausgabe zum Zeitpunkt t unabhängig von den vorhergehenden Systemeingaben. Die Systemausgaben von *dynamischen Systemen* hingegen hängen von den vorhergehenden Systemeingaben ab. Um die Vorhersagbarkeit bei dynamischen Systemen zu gewährleisten, wird der Begriff des Systemzustands verwendet. Durch Kenntnis des Systemzustands zum Zeitpunkt t_0 sowie der folgenden Systemeingaben $u(t), t \geq t_0$ kann die Systemausgabe $y(t), t \geq t_0$ eindeutig bestimmt werden. Dies führt zu folgenden Zustandsgleichungen:

$$\dot{x}_1(t) = f_1(x_1(t), \ldots, x_n(t), u_1(t), \ldots, u_p(t), t), \quad x_1(t_0) = x_{10}$$
$$\vdots$$
$$\dot{x}_n(t) = f_n(x_1(t), \ldots, x_n(t), u_1(t), \ldots, u_p(t), t), \quad x_n(t_0) = x_{n0}$$

Neben den Zustandsgleichungen ergeben sich folgenden Gleichungen für die Systemausgaben.

$$y_1(t) = g_1(x_1(t), \ldots, x_n(t), u_1(t), \ldots, u_p(t), t)$$
$$\vdots$$
$$y_m(t) = g_m(x_1(t), \ldots, x_n(t), u_1(t), \ldots, u_p(t), t)$$

Der Zustandsraum wird meist mit X bezeichnet und beinhaltet alle möglichen Werte des Systemzustands.

Historischer Ursprung der Systemtheorie bildete die Beschreibung physikalischer Systeme. Vorrangig beschrieben die Modelle Eigenschaften wie die Geschwindigkeit, Masse, Temperatur oder der Druck von Festkörpern, Flüssigkeiten oder Gasen. Bei der Beschreibung der physikalischen Vorgänge kommen vorrangig kontinuierliche Spezifikationen, wie Differenzialgleichungen, zum Einsatz. Man spricht hier dann von *kontinuierlichen Systemen*. Abbildung 2.2 zeigt die Struktur einer Füllstandsregelung eines Wassertanks. In diesem Beispiel ist das herausfließende Wasser $y(t)$ durch die Systemeingangsgröße des zufließenden Wassers $u(t)$ sowie des aktuellen Wasserstands im Behälter $X(t)$ beschrieben. Der Schwerpunkt der Analysen in dieser Arbeit

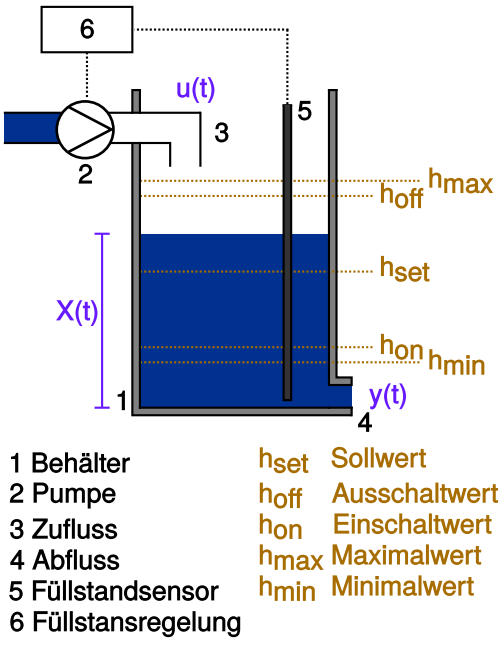


Abbildung 2.2: Füllstandsregelung eines Wassertanks

liegt auf cyber-physischen Systemen (CPSs) und hierbei insbesondere auf den 'informationsverarbeitenden Prozessen'. Aufgrund dieser Betrachtung kann der Begriff

des zu analysierenden Systems enger gefasst werden. Bei durch Software gesteuerten oder überwachten Prozessen rückt die Spezifikation und Analyse in den Fokus von *diskreten Systemen*. Hierbei können zwei Aspekte des Systems diskretisiert werden. Bei einem System mit diskretem Zustandsraum besteht dieser aus einer diskreten Menge von Zuständen. Wohingegen bei kontinuierlichen Systemen der Zustandsraum aus einem Vektor kontinuierlicher Werte besteht. Bei zustandsdiskreten Systemen springt der Systemzustand von einem diskreten Zustand zu einem anderen. Ein weiterer Aspekt ist die Diskretisierung der Zeit. Anstelle Systeme mit einer kontinuierlichen Zeitbasis zu untersuchen, werden Systeme mit Hilfe einer Sequenz von Zeitpunkten in einem festen Intervall untersucht. Bei diesen zeitdiskreten Systemen ist somit nur an bestimmten, äquidistanten Zeitpunkten ein Zustandswechsel des Systems möglich. Bei zeit- und zustandsdiskreten Systemen finden somit diskrete Zustandsübergänge nur an diskreten Zeitpunkten statt, was als sogenanntes Ereignis bezeichnet wird. Diese Klasse von Systemen werden als *ereignisdiskrete Systeme* bezeichnet. Zur Verdeutlichung des dynamischen Aspekts dieser Systeme wird häufig die Bezeichnung Discrete Event Dynamic Systems (DEDS) verwendet.

Softwarebasierte Systeme haben üblicherweise eine ereignisdiskrete Sicht auf die physikalischen, kontinuierlichen Vorgänge. Sensoren erfassen den Zustand eines physikalischen Systems in periodischen Abständen. Die Speicherung und Verarbeitung basieren auf wertediskreten Zuständen.

Begriff 4. *Discrete Event Dynamic Systems (DEDS)*
Zustandsdiskrete, ereignisgetriebene Systeme, bei welchen der Zustandswechsel über die Zeit alleinig vom Eintreten asynchroner, diskreter Ereignisse abhängig ist [24].

Zwischen den Ereignissen eines DEDS bleibt das System in einem stabilen Zustand, d. h. es erfolgen keine Zustandswechsel. Diese Systembetrachtung ist weitverbreitet bei der Beschreibung technischer Systeme, die zur Steuerung oder Regelung von kontinuierlichen Prozessen dienen. Am Beispiel der Füllstandsregelung sind weniger die kontinuierlichen, physikalischen Prozesse im Wassertank zu analysieren, sondern die Ansteuerung der Wasserzufuhr anhand der gelieferten Sensordaten. Abbildung 2.3 zeigt den kontinuierlichen Verlauf der Füllstandshöhe sowie die vom Sensor erfasste Füllstandshöhe. Basierend auf der periodischen Abfrage der Sensorwerte, entsteht eine diskrete Abbildung des zeitlichen Verlaufs des Füllstands. Hierbei findet sowohl eine Diskretisierung auf der Zeitachse, durch periodisches Abfragen des Sensors als auch eine Quantisierung bezüglich der gelesenen Werte statt. Die Quantisierung der Werte liegt an der binären Darstellung der Messgrößen mit endlicher Datenwortbreite.

Im Rahmen der Arbeit liegt der Fokus auf der Analyse von DEDS. Cassandras und Lafortune benennen in [24] als relevante Gebiete für die ereignisdiskrete Modellierung: Kommunikationsnetzwerke, Produktionsanlagen oder allgemein die Ausführung von Computerprogrammen. Sie stellen heraus, dass insbesondere Anwendungsfälle, die digitale Computersysteme beinhalten, relevant sind. Dingel et al. zeigen in [32] wie viele Probleme im Software Engineering (SE) auf diskrete Ereignisse abgebildet werden können. Im speziellen auch im Rahmen der Verifikation von Kommunikationsprotokollen. Mit dem Fokus auf vernetzte, eingebetteten Systemen, mit

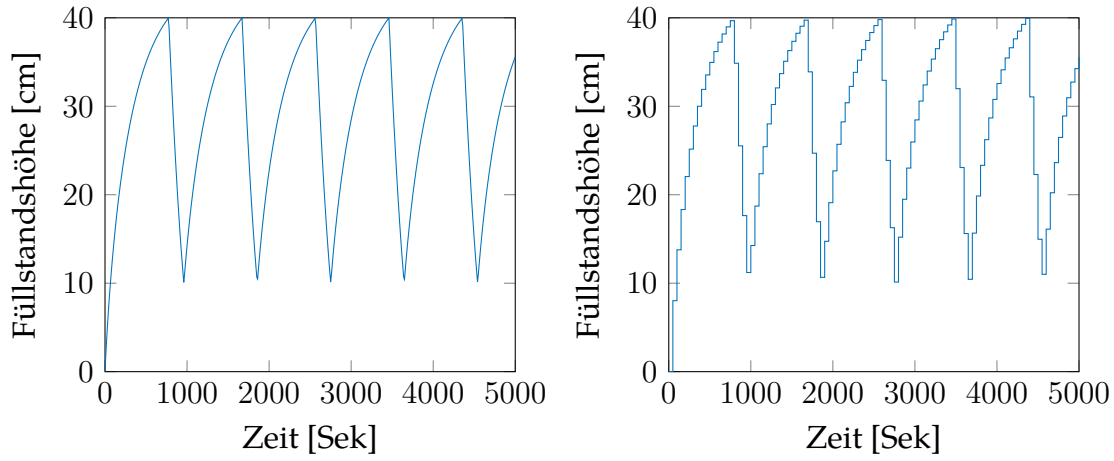


Abbildung 2.3: Vergleich einer kontinuierlichen und diskreten Zustandsbeobachtung

informationsverarbeitende Prozessen, stellt der Fokus auf DEDS somit keine relevante Einschränkung dar. Auch im Zusammenhang mit dieser Modellbildung können die Begriffe Modell und System, bei Erreichen einer aussagekräftigen Genauigkeit, synonym verwendet werden. An dieser Stelle sei darauf hingewiesen, dass die Fehlereffektbeobachtung weiterhin im Kontinuierlichen stattfinden kann. Am Beispiel des Wassertanks ergibt die Analyse ein ereignisdiskretes Ansteuerverhalten für den Wasserzufluss, das einer kontinuierlichen Füllstandshöhe entspricht.

Aufbauend auf diesen Überlegungen und angelehnt an das Konzept in [118] wird ein System als folgendes 3-Tupel beschrieben:

$$\Sigma = (\mathbb{T}, \mathbb{X}, B)$$

Hierbei wird das System über den Zustandsraum $\mathbb{X}$ spezifiziert. Das Verhalten des Systems ist somit eine Abfolge von Belegungen des Zustandsraums über die Zeit ($\mathbb{X}^{\mathbb{T}}$). B repräsentiert die Verhaltensbeschreibung des Systems und ordnet jedem Zeitpunkt $t \in \mathbb{T}$ eine Belegung des Zustandsraums $\mathbb{X}$ zu. Bei der mathematischen Beschreibung kontinuierlicher Systeme würde diese Abbildung durch Differenzialgleichungen spezifiziert. Bei ereignisdiskreten Systemen kann das Verhalten als Zustandsübergänge, abhängig von asynchronen Ereignissen, aufgefasst werden. Wie motiviert, liegt der Fokus dieser Arbeit auf ereignisdiskreten Modellen zur Beschreibung des Systems. Hierbei wird das Verhalten des Systems mittels diskreten Prozessen modelliert. Eine detaillierte Vorstellung der verwendeten Simulationssprache erfolgt in Abschnitt 2.3. Zuvor wird der Begriff des Systems bzw. dessen Modell nochmals genauer untergliedert.

2.2.2 Systembestandteile

Bei der Betrachtung der mannigfaltigen Definitionen eines Systems – im vorherigen Abschnitt wird ein System z. B. als Eingabe-Ausgabe-Modell beschrieben – sind meist die folgenden Eigenschaften zu erkennen:

1. Ein System ist etwas ‚Zusammengesetztes' und kann aus einer Vielzahl von interagierenden Komponenten bestehen.

2. Ein System ‚realisiert eine Funktion' bzw. erbringt einen geforderten Dienst.

Den Sachverhalt des ‚Zusammengesetzten' spiegelt sich in dieser Arbeit durch die Untergliederung eines Systems wider. Die Begrifflichkeiten werden nachfolgend definiert.

Begriff 5. *System*
Als System wird immer die Gesamtheit des zu analysierenden Gegenstands gesehen.

Es entspricht somit dem Design under Test (DUT). Zur reproduzierbaren Überprüfung von Eigenschaften des DUTs wird häufig eine Testbench bzw. ein Prüfstand herangezogen. Dieser stimuliert das DUT mittels einer Eingabe ($u(t)$) und überprüft dessen Ausgaben ($y(t)$). In dieser Betrachtung spiegelt sich auch der zweite Punkt ‚die realisierte Funktion' wider. Im Rahmen der Arbeit werden vorrangig Systemsimulationen eingesetzte. Eine ausführliche Vorstellung der Interaktion zwischen Testbench und DUT, im Kontext von Systemsimulationen, ist Standard zur Universal Verification Methodology (UVM) [1] vorgestellt. Ein System untergliedert sich in Komponenten, die als Hardware oder Software realisiert werden können.

Begriff 6. *Komponente*
Bestandteile des Systems, die zur Erbringung der Systemfunktionalität miteinander interagieren.

Im Zusammenhang mit der Systemsimulation, vergleiche Abschnitt 2.3, wird der Begriff der Einheit verwendet.

Begriff 7. *(Basis-Simulations)einheit*
In Bezug auf die Systemsimulation stellt die Basis-Simulationseinheit (BSE) den kleinsten atomaren Grundbaustein dar, der zur Zusammenstellung einer Komponente verwendet wird.

Abbildung 2.4 zeigt die Bestandteile des Systems und verdeutlicht deren Zusammenhänge. Beim Aufbau der Systemsimulation wird eine flache Hierarchie gewählt.

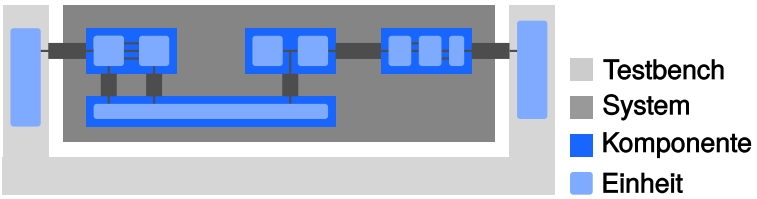


Abbildung 2.4: Bestandteile einer Systemsimulation

Somit ist die Komponente, im Zusammenhang mit der Systemsimulation, lediglich eine logische Einheit, die sich nicht unmittelbar in der Simulation manifestiert. Die Testbench wird in dieser Arbeit als Teil des Systems betrachtet. Somit wird der Begriff des Systemzustands auf den Simulationszustand ausgeweitet. Sollte eine dedizierte

Unterscheidung notwendig sein, wird explizit darauf hingewiesen, ob es sich um den System- oder Simulationszustand handelt. Zur Unterscheidung wird der Simulationszustand $\mathbb{X}$ unterteilen in $\mathbb{X}_{tb}$, also alle Zustände, die der Testbench zugeordnet werden und $\mathbb{X}_{dut}$, als der Zustandsraum des eigentlichen DUTs.

2.2.3 Timed Automata

Ein formaler Ansatz DEDS zu Beschreiben bedient sich der Theorie von Sprachen und Automaten. Hierbei wird die Menge der Ereignisse des DEDS als Alphabet einer Sprache betrachtet. Eine Ereignissequenz ist somit ein Wort in dieser Sprache. Zur Erinnerung: Ereignisse lösen einen Zustandsübergang im System aus. Somit ist die Ereignissequenz nichts anderes als die Zustandswechsel des Systems über die Zeit. Um alle möglichen Wörter, d. h. Ereignissequenzen des Systems zu spezifizieren, können Automaten verwendet werden.

Begriff 8. *Deterministischer Automat*
Formell wird ein Automat als Sechstupel $< X, E, f, \Gamma, x_0, X_m >$ *beschrieben.*

Wobei X *eine endliche Menge von Zuständen ist,* E *eine endliche Menge von Ereignissen,* $f : X \times E \to X$ *die Übergangsfunktion, die einem Zustand und Ereignis einen Folgezustand zuordnet,* $\Gamma : X \to 2^E$ *ist die Aktive-Ereignis-Funktion, die angibt ob* $f(x, e)$ *definiert ist,* x_0 *ist der Startzustand und* $X_m \subseteq X$ *die Menge markierter Zustände.*

Eine Erweiterung von Automaten stellt die Timed Automata (TA) dar, wie in Alur und Dill in [2] vorgestellt.

Begriff 9. *Timed Automata*
Formell wird ein TA als Sechstupel $< X, x_0, \Sigma, C, I, E >$ *beschrieben.*

Wobei X *eine endliche Menge von Zuständen ist,* Σ *wird als Ausgabealphabet bezeichnet und* C *ist eine endliche Menge von Zeitgebern.* I *bezeichnet Invarianten von Zuständen und* $E \subseteq (X \times \sigma \times g(C) \times r(C) \times X)$ *ist die Menge von Zustandsübergängen.*
x_0 *ist der initiale Zustand, g(C) beschreibt Übergangsbedingungen abhängig von Zeitgebern und r(C) deren Rücksetzen.*

TAs werden häufig zur Modellierung und Analyse von Echtzeitsystemen verwendet. Sie bieten neben der Modellierung des zustandsbasierten Verhaltens, Spezifikationsmöglichkeiten zur Berücksichtigung von Zeitverhalten [115]. Insbesondere durch die Modellierung asynchron rücksetzbarer Zeitgeber.

Ein TA besteht aus einer endlichen Menge von Zuständen X, Aktionen Σ, Zeitgebern C, die einzeln zurückgesetzte werden können und einer Menge von Invarianten I. Eine Kante $(x_i, \sigma, g(C), r(C), x_j) \in E$ beschreibt einen Übergang von Zustand x_i zu Zustand x_j unter der Bedingung $g(C)$. Alle Anweisungen im Zustandsautomaten sind an den Zustandsübergängen annotiert, ähnlich eines Mealy-Automaten. Beim Übergang kann eine Aktion a ausgeführt sowie Zeitgeber zurückgesetzt ($r(C)$) werden.

UPPAAL

UPPAAL ist eine Werkzeugumgebung zur Qualifikation von Echtzeitsystemen, welche in [7] im Detail vorgestellt wird. Der Kern der Umgebung bildet ein Model Checker für TA, wobei Erweiterungen für Invariante, Signalisierung sowie begrenze Datenbereiche beinhaltet sind. Ziel ist es das zu untersuchende System als TA zu modellieren, simulieren und über Zusicherungen bestimmte Systemeigenschaften zu überprüfen. Hierbei kann mittels eines Simulators der TA schrittweise simuliert werden. In jedem Schritt werden die benötigten Eingaben, interaktiv eingegeben. Zu überprüfende Systemeigenschaften können als Zusicherungen mittels einer Untermenge der Computation Tree Logic (CTL) formuliert werden und automatisch durch den beinhalteten Model Checker überprüft werden. Hierbei wir abgefragt ob ein bestimmter Zustand des TAs erreicht werden kann (Reachability Property), ob gewisse Systemeigenschaften nie auftreten (Safety Property) oder ob ein gewisses Ereignis eintritt unter gegebenen Vorbedingungen (Liveness Properties). Der Model Checker führt dann eine vollständige Suche durch, die alle möglichen Zustandsfolgen des Systems beinhaltet, um die Eigenschaften zu überprüfen. Sollte eine Verletzung der Zusicherung möglich sein, generiert der Model Checker ein Gegenbeispiel, welches im Simulator nachverfolgt werden kann.

2.2.4 Computation Tree Logic

Zur formalen Spezifikation von Systemeigenschaften erfolgt häufig in der Linear Temporal Logic (LTL) bzw. deren Erweiterung der CTL. Ein Beispiel hierfür ist der im vorherigen Abschnitt vorgestellte Model Checker der UPPAAL-Werkzeugumgebung. Beide Sprachen werden ausführlich von Kropf in [63] vorgestellt. Bei beiden Sprachen handelt es sich Sprachen der temporalen Logik, in welcher zeitliche Abläufe spezifiziert werden.

In er LTL werden Aussagen über zeitliche Abfolge von Ereignissen auf einem Pfad getroffen. Für einstellige Verknüpfungen sind folgende Zusicherungen möglich:

Xϕ: Die Bedingung ϕ gilt im nächsten Zustand.

Gϕ: Die Bedingung ϕ gilt in allen nachfolgenden Zuständen.

Fϕ: Die Bedingung ϕ ist irgendwann in den nachfolgenden Zuständen erfüllt.

Durch zweistellige Verknüpfungen können zwei Bedingungen in Relation zueinander gesetzte werden. Ein Beispiel ist eine Bedingung ψ, die mindestens solange gilt, bis die Bedingung phi gilt (ψ**U**phi).

Die CTL beinhaltet die LTL, erweitert diese aber um Pfadquantoren. Hierbei wird die Zustandsabfolge nicht rein linear modelliert, wie bei der LTL. Durch die Verzweigung ist die CTL besonders geeignet, Zustandsabfolgen zu modellieren, die unterschiedliche Ausführungspfade besitzen. Die Abfolge von Ereignissen wird als Baumstruktur modelliert. Die für diese Arbeit relevanten Konstrukte sind die folgenden:

EXϕ: Die Bedingung ϕ gilt in einem nächsten Zustand.

EGϕ: Die Bedingung ϕ gilt für alle Zustände auf mindesten einem Pfad.

EFϕ: Die Bedingung ϕ gilt in einem der folgenden Zustände.

AXϕ: Die Bedingung ϕ gilt in jedem nächsten Zustand.

AGϕ: Die Bedingung ϕ gilt in allen Zuständen aller Pfade.

AFϕ: Auf allen Pfaden gilt die Bedingung ϕ mindestens einmal.

Mit Hilfe dieser Ausdrücke können Zusicherungen nun für unterschiedliche Zustandsfolgen eines Systems spezifiziert werden.

2.3 Systemsimulation

Die Verwendung von Simulationen zur Systemanalyse und Effektbeobachtung ist weitverbreitet. Sie werden z. B. in der Wettervorhersage [113], bei virtuellen Auto-Crashtests [14] aber auch beim Entwurf von elektronischen Systemen eingesetzt. Simulationen kommen immer dann zum Einsatz, wenn es umständlich, kostenintensiv oder unmöglich ist die Analysen physisch durchzuführen. Simulationen erlauben eine höhere Kontrolle über Parameter und eine detailliertere Einsicht in das Experiment. Die in dieser Arbeit durchgeführten Analysen basieren auf Verhaltensmodellen des zu untersuchenden eingebetteten Systems. Diese so genannten *virtuellen Prototypen* erlauben die Analyse von Hardware- und Softwaresystemen mittels Simulationen. Ein softwarebasierter Simulationskern simuliert die benötigten Systemeigenschaften.

2.3.1 Simulationssprache SystemC

Die ereignisbasierte Simulationssprache SystemC [12, 49] erlaubt das Modellieren von komplexen Systemen auf unterschiedlichen Abstraktionsebenen (vgl. hierzu Abschnitt 2.4).

SystemC ist eine C++-Klassenbibliothek, die das Modellieren nebenläufiger Hardware und Software erlaubt. C++ wird um Mechanismen wie Synchronisation, Parallelität und Interprozesskommunikation erweitert. Anfangs wurde SystemC als Hardwarebeschreibungssprache mit bereitgestelltem Simulator entwickelt, ähnlich zu Sprachen wie Very High Speed Integrated Circuit Hardware Description Language (VHDL) [48] und Verilog [47]. Mechanismen wie strukturelle Hierarchie, mehrwertige Logik oder taktgesteuerte Prozesse erlauben das Simulieren von Registern oder logische Funktionen. Aufgrund der Transaction Level Modeling (TLM)-Erweiterung entwickelte sich SystemC zur umfassenden Sprache zur Modellierung auf Systemebene. TLM-2.0 erlaubt die Modellierung von Kommunikation ohne Kenntnis der spezifischen Hardwareimplementierung. Hierbei wird die Kommunikation, d. h. das eigentliche Protokoll, durch eine logische Transaktion abstrahiert. Neben der Abstraktion bietet das Vorgehen eine wohldefinierte Schnittstelle zur Interaktion zwischen

Simulationskomponenten. Die vereinheitlichten Schnittstellen erhöhen die Interoperabilität von Modellen. So bietet z. B. Synopsys eine SystemC-TLM-Bibliothek mit einer Vielzahl von Simulationsmodellen [110] an.

Der Aufbau von SystemC und der TLM-2.0-Erweiterung ist in Abbildung 2.5 skizziert. Die SystemC-Kernsprache bietet Mechanismen wie Module, Prozesse, Ports

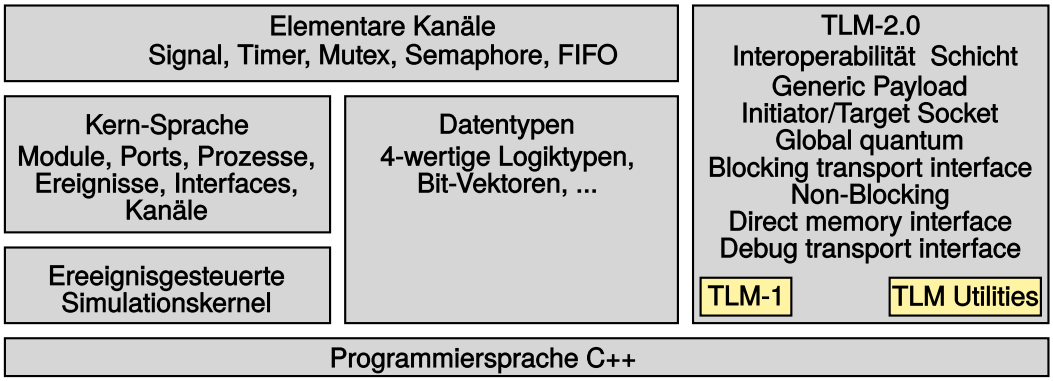


Abbildung 2.5: Sprachaufbau von SystemC

und Signale. Eines der wichtigsten Sprachkonstrukte ist das Modul (`sc_module`) zur Modellierung von abgeschlossenen Simulationseinheiten und somit dem Aufbau einer hierarchischen Struktur. Ein Modul kann aus weiteren Modulen bestehen (hierarchische Sicht) und/oder unterschiedliche Prozesse beinhalten (funktionale Sicht). Per Definition können Module über Kanäle/Signale und Ports miteinander kommunizieren. Ports bilden die externe Schnittstelle eines Moduls. Kanäle/Signale verbinden Ports in der übergeordneten Hierarchieebene.

Signale (`sc_signal`) sind elementare Kanäle mit wohldefinierten Zugriffsmethoden. Kanäle reichen von einfachen Kanälen, die eine einzige Leitung modellieren zu komplexen hierarchischen Modellen.

TLM-2.0 erweitert SystemC um wohldefinierte transaktionsorientierte Kommunikationsschnittstellen. Das ursprüngliche Anwendungsgebiet war die Modellierung von Memory-Mapped On-Chip Bussen. TLM-2.0 definiert einen Initiator-Socket, der eine Transaktion initiiert und ein Ziel-Socket, der die Transaktion verarbeitet. Der Ziel-Socket kann sowohl eine blockierende als auch eine nicht blockierende Verarbeitung der Transaktion bereitstellen. Bei blockierenden Aufrufen erfolgt die Abarbeitung in einer einzigen Phase, bei nicht blockierenden Aufrufen wird eine Interaktion in mehrere Phasen aufgeteilt. Das Transaktionsobjekt (`tlm_generic_payload`) spezifiziert die bei einer Transaktion ausgetauschten Daten. Das Transaktionsobjekt beinhaltet Felder wie die Zieladresse, ein Längenfeld, einen Zeiger auf die eigentlichen Nutzdaten sowie ein Antwortfeld, in dem der Ziel-Socket den Empfangsstatus zurückmeldet.

Zur Modellierung von Nebenläufigkeit stellt SystemC unterschiedliche Prozesstypen (`SC_THREAD`, `SC_CTHREAD` und `SC_METHOD`) zur Verfügung. Sie sind die grundlegende Methode zur Modellierung von Nebenläufigkeit und dienen hauptsächlich der Kapselung von Funktionalität. Prozesse bestehen aus sequenziellen Anweisungen, die an vorgegebenen Stellen suspendiert und reaktiviert werden können. Die

Prozesstypen unterscheiden sich durch ihre Suspendierungs- und Reaktivierungseigenschaften. Ein Prozess vom Typ `SC_THREAD` wird einmalig zu Simulationsbeginn gestartet und mittels Warteanweisungen suspendiert. Eine Dauerschleife unterbindet die Beendigung eines `SC_THREAD`s. Der Prozesstyp `SC_METHOD` hingegen kann mehrmals ausgeführt werden und kann nicht durch Warteanweisungen suspendiert werden. Bei der Deklaration wird eine Sensitivität auf Ereignisse angegeben. Tritt eines der Ereignisse ein, wird die Methode aktiviert und danach komplett ausgeführt. Der `SC_CTHREAD` ist ein Hybrid. Er kann durch Warteanweisungen suspendiert werden, wird aber mittels einer Sensitivität erneut reaktiviert. Häufiges Anwendungsszenario sind Prozesse, die über ein Taktsignal getriggert werden. Zwischen der Reaktivierung und Suspendierung wird eine Sequenz von Anweisungen durchlaufen. Diese autarken Anweisungsblöcke werden somit innerhalb eines Zeitpunktes (Simulationszeit) abgearbeitet. Warteanweisungen existieren als zeitbedingte oder ereignisbedingte Anweisungen. Der Simulationskern reaktiviert zeitbedingte Anweisungen, wenn die spezifizierte Zeit abgelaufen ist, wohingegen er ereignisbedingte Anweisungen reaktiviert, wenn das spezifizierte Ereignis eintritt. Hierbei reichen die Ereignisse von dem Ändern eines Signals (`value_change_event()`) über das Auftreten einer Taktflanke (`posedge()`) bis hin zu dem dedizierten Auslösen eines Ereignisses (`sc_event`).

Zum Scheduling der einzelnen Prozesse verwendet SystemC einen zeitdiskreten, deltazyklenbasierten Simulationskern. Das Konzept der Deltazyklen erlaubt die quasi-parallele Ausführung der beschriebenen Prozesstypen. Der Simulationskern verwaltet eine Datenstruktur, die alle im aktuellen Zeitschritt lauffähigen Prozesse beinhaltet. Der Simulationskern startet diese Prozesse sequenziell, wobei zwischen zwei Prozessen die Simulationszeit nicht voranschreitet. Sollte ein Prozess ein Ereignis auslösen, das einen weiteren Prozess ausführbar macht, wird dieser Prozess in die Datenstruktur eingefügt und ohne Voranschreiten der Simulationszeit abgearbeitet. Ein Durchlauf durch diese Prozessabarbeitung, bis alle lauffähigen Prozesse zu einem Zeitpunkt abgearbeitet, sind, wird als *Deltazyklus* bezeichnet. Die Ausnahme bilden Ereignisse welche mit einem expliziten Zero-Time Argument ausgeführt werden. In diesem Fall wird Abarbeitung erst im nächsten Deltazyklus, aber immer noch zum selben Simulationszeitpunkt, ausgeführt. Sind alle Deltazyklen und somit alle lauffähigen Prozesse abgearbeitet, wird die Simulationszeit auf den Zeitpunkt des nächsten abzuarbeitenden Prozesses gesetzt. Abbildung 2.6 verdeutlicht das Vorgehen des Simulationskerns. Zum Zeitpunkt t_n sind die Prozesse b_1, b_2, b_3 lauffähig. Bei der Ausführung des Prozesses b_1 wird die Ausführung des Prozesses b_5 zum Zeitpunkt t_{n+1} festgelegt. Ähnlich verhält es sich mit dem Prozess b_2, außer, dass dieser Prozess sofort lauffähig ist. Der Prozesse b_3 verwendet ein Zero-Time Argument, um sich selbst lauffähig zu machen. Dies geschieht mit einem neuen Deltazyklus zum gleichen Simulationszeitpunkt. Sind zum aktuellen Zeitpunkt (t_n) alle Deltazyklen und somit alle lauffähigen Prozesse abgearbeitet, wird die Simulationszeit vorangeschritten. Diese Sichtweise verdeutlicht den Bezug zur Definition des Systems $\Sigma = (\mathbb{T}, \mathbb{X}, B)$ in Abschnitt 2.2.1. Hierbei ist $B = \{b_0, \ldots, b_n\}$ die Menge von Funktionen bzw. Prozessen, die den Systemzustand $\mathbb{X}$ modifizieren. Der SystemC-Kernel ruft die einzelnen Prozesse auf. Hierbei gilt anzumerken, dass die Aufteilung

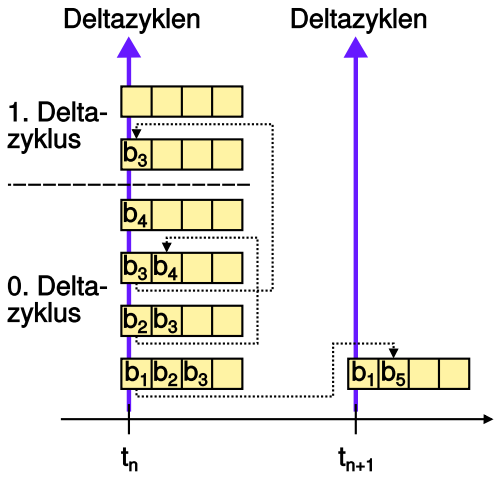


Abbildung 2.6: Scheduling Beispiel des SystemC-Kernels

der Prozesse $B = \{b_0, \ldots, b_n\}$ nur bedingt die Struktur des Quellcodes widerspiegelt. So ist es möglich, ein `SC_THREAD` durch Warteanweisungen zu unterbrechen. Dies wird logisch mit mehreren Prozessen $\{b_i, \ldots, b_k\}$ dargestellt. Mit dieser Sichtweise lässt sich der System- bzw. Simulationszustand $\mathbb{X}$ definieren.

Begriff 10. *System- bzw. Simulationszustand $\mathbb{X}$*
Der System- bzw. Simulationszustand $\mathbb{X}$ bezeichnet alle Variablen in der Simulation, die Informationen über Prozessaufrufe und somit über Delta- und Zeitschritte transportieren.

Basierend auf dem Vorgehen, sowohl die Hardware als auch die Software eines Systems in einer softwarebasierten Simulation zu beschreiben, wird die Hardware/Software-Partitionierung bestens unterstützt. Des Weiteren ist die Verfeinerung der Modelle und damit verbunden z. B. die Konkretisierung des Zeitverhaltens gegeben. Wie bei allen Simulationen muss immer ein Trade-Off zwischen Simulationsgenauigkeit und Simulationsaufwand vorgenommen werden. Bei dieser Betrachtung ist die Einbeziehung des Analyseziels entscheidend. Durch Erhöhung des Simulationsaufwands kann in der Regel die Simulationsgenauigkeit gesteigert werden. Somit ist die Wahl des Modellierungsdetailgrades, zum Erreichen des gewünschten Analyseziels, entscheidend. Unter Simulationsaufwand ist sowohl die Rechenzeit zur Durchführung der Simulation als auch die Entwicklungszeit des Simulationsmodells zu verstehen. Die stetig fallenden Kosten für Rechenressourcen erlauben es, die relevanten Simulationen in annehmbarer Zeit durchzuführen. Herkömmliche Arbeitsplatzrechner können, die in dieser Arbeit betrachteten Simulationen, in akzeptabler Zeit durchführen. Techniken wie das Pipelining oder die parallele Ausführung mehrerer Simulationen bieten eine ausreichende Simulationsbeschleunigung zur Durchführung der vorgestellten Fallstudien. Die Optimierung einzelner Simulationsmodelle unter dem Gesichtspunkt der Ausführungsdauer wird in dieser Arbeit nicht behandelt.

Das Simulationsmodell der Füllstandsregelung verdeutlicht die Systembeschreibung mit SystemC. Der Wasserbehälter und die Ansteuerung sind als separate Module implementiert. Der Wasserbehälter enthält mehrere Prozesse. Diese modellieren

den Füllstand des Behälters sowie die Verarbeitung der Sensor- und Aktorsignale. Das Sensorsignal wird als `SC_THREAD` mit festen Wartezeiten modelliert. Dies entspricht der periodischen Abfrage des Füllstands durch einen elektronischen Sensor. Die Ansteuerung wird je nach Implementierung (im Detail in Kapitel 7.2.3 vorgestellt) in mehrere Module untergliedert. Die Ansteuerung der Aktoren sowie das Lesen der Sensoren des Wasserbehälters erfolgen über Signale, die der jeweiligen Ventilöffnung oder dem Füllstand entsprechen.

2.3.2 Ergänzende Simulationsumgebungen

Neben SystemC existiert eine Vielzahl von Simulationswerkzeugen. Die unterschiedlichen Werkzeuge konzentrieren sich häufig auf einen dedizierten Analysefokus bzw. Anwendungsgebiet. Das Simulationswerkzeug OMNeT++ wurde speziell zur Simulation und Analyse von Netzwerken entwickelt. Die Bibliothek INET/OMNeT++ [13] beinhaltet Modelle zur Simulation von Internetprotokollen. Trotz der beiden unterschiedlichen Anwendungsgebiete ist es möglich, OMNeT++-Modelle in SystemC zu überführen und in den hier vorgestellten Ansatz einzubeziehen.

Bei anderen Simulationsumgebungen ist eine einfache Überführung in die Simulationssprache SystemC nicht unbedingt ohne Mehraufwand möglich. Die Simulationsumgebung IPG CarMaker [50] zum Beispiel richtet sich vor allem an die Simulation des Fahrzeugumfelds. Sie modelliert Aspekte der Fahrdynamik sowie der Interaktion des Automobils mit der Umgebung. Dies ermöglicht die Entwicklung von Algorithmen im virtuellen Fahrversuch. Eine Überführung der Simulationsumgebung nach SystemC würde sich hier nicht anbieten, da die simulationsinternen Funktionen zur Fahrdynamikberechnung nachempfunden werden müssten. Aus diesem Grund wird eine Co-Simulation angestrebt. Die Co-Simulation synchronisiert die zeitliche Abfolge des SystemC-Simulationskerns und der CarMaker-Simulation. Zudem ermöglicht sie einen Datenaustausch zwischen beiden Simulationen. Mit diesem Vorgehen ist es möglich die in dieser Arbeit erstellten Werkzeuge auf die Co-Simulationsumgebung anzuwenden und somit weitere Simulationswerkezuge indirekt zu unterstützen. Die Fallstudie in Abschnitt 7.2.1.3 ist in einer solchen Co-Simulationsumgebung angesiedelt.

2.4 Abstraktionsebenen

Wie bereits motiviert, bildet ein Modell des zu untersuchenden Systems die Grundlage der Sicherheitsanalyse. In diesem Zusammenhang wird, je nach Analyseziel, unterschiedliche Detaillierungsgrade des Modells verwendet. Wie im vorherigen Abschnitt vorgestellt, ermöglicht SystemC eine Verfeinerung der Simulationsmodelle. Hierbei können detaillierte Modelle zur akkuraten und häufig zur lokalen Analyse zum Einsatz kommen. Abstrakte, globale Simulationsmodelle werden hingegen zur Beschleunigung der Simulation und somit zur Exploration eingesetzt. Der folgende Abschnitt stellt typische Abstraktionsebenen vor und setzt sie in Bezug zu einem

SystemC-Simulationsmodell. Zur Bereitstellung eines durchgängig anwendbaren Ansatzes ist es wichtig, die unterschiedlichen Abstraktionsebenen zu unterstützen.

Beim Entwurf integrierter Schaltkreisen hat sich zur Darstellung der unterschiedlichen Sichtweisen das Y-Diagramm in Abbildung 2.7, mit der typischen Aufteilung in die drei Domänen: Verhalten, Struktur und Geometrie, etabliert [38]. Erfolgt ein

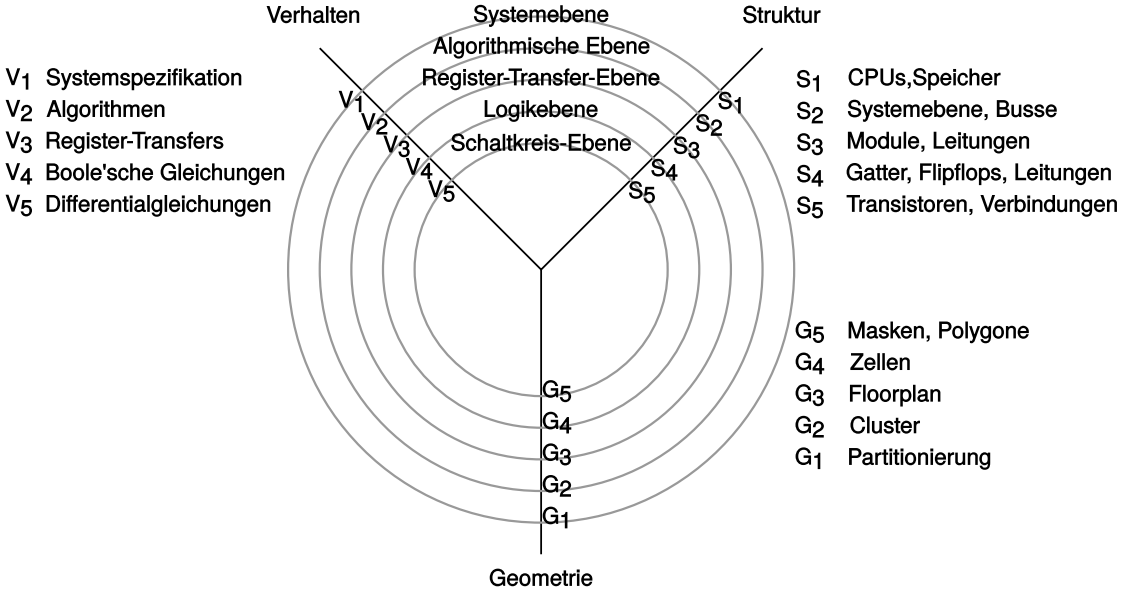


Abbildung 2.7: Y-Diagramm nach Gajski-Kuhn

Schritt auf der Verhaltensachse, z. B. von V_1 nach V_2, wird dies als *Verfeinerung* bezeichnet. Der Übergang von der Verhaltens- auf die Strukturachse (V_4 nach S_4), wird als *Synthese* bezeichnet. Automatisierte Syntheseschritte realisieren den Übergang auf den untersten Ebenen. Hierdurch ist gewährleistet, dass die Implementierung, in Bezug auf das darüberliegende Modell, korrekt ist.

Eine weitere Darstellungsform unterteilt die Abstraktionsebenen in Berechnung und Kommunikation, da die Verfeinerung der Kommunikation und des Verhaltens meistens orthogonal erfolgen kann. Diese Aufteilung wird zur transaktionsbasierten Modellierung mit SystemC verwendet [40] und ist in Abbildung 2.8 dargestellt. Eine ähnliche Unterteilung wird in der SystemC-TLM-2.0-Spezifikation [4] aufgegriffen. Hierbei sind auf der horizontalen Achse die Verfeinerung des Verhaltens und auf der vertikalen Achse die Verfeinerungsgrade der Kommunikation aufgetragen. Beim Verhalten wird z. B. unterschieden, ob ein Zeitverhalten modelliert wird und wenn ja mit welcher Genauigkeit. Die Grenzen zwischen den Abstraktionsebenen sind meist fließend. Im Folgenden werden drei Klassen von Abstraktionsebenen vorgestellt, die unterschiedliche Anforderungen an den entwickelten Ansatz stellen. Der Übergang zwischen den Klassen ist fließend und dient hier lediglich der strukturierten Darstellung. Für jede Klasse werden typische Vertreter aufgezeigt.

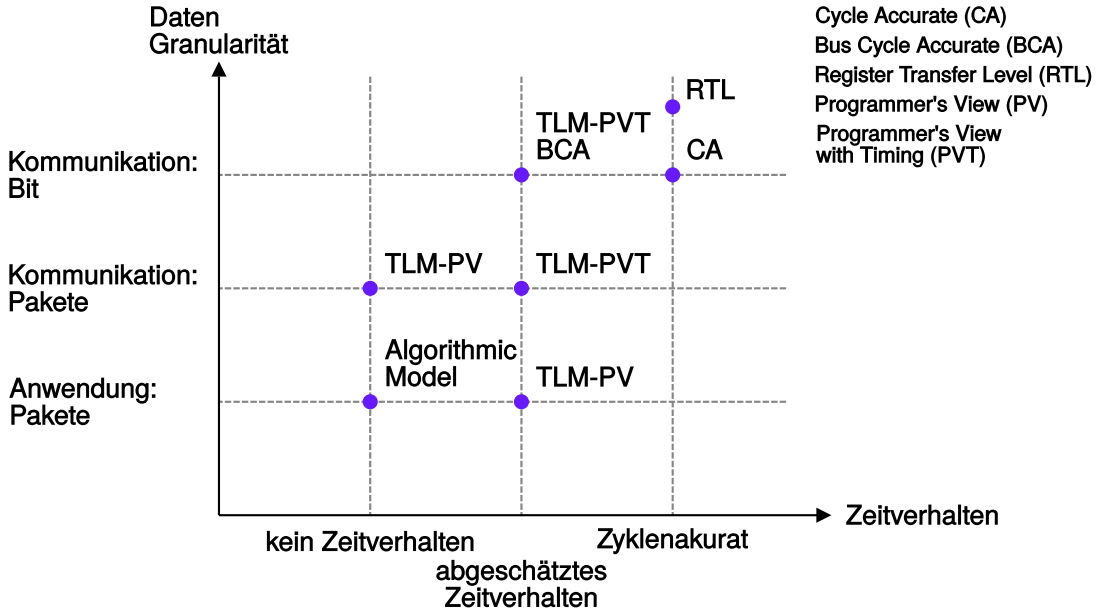


Abbildung 2.8: TLM-Abstraktionsebenen nach [40]

Funktionale Modelle

Funktionale Modelle sind meist auf Systemebene beschrieben. Sie kommen häufig in frühen Entwicklungsphasen zum Einsatz. Zur Abbildung der Systeminformationen kommen häufig komplexe Datenstrukturen zum Einsatz.

Begriff 11. *Verhaltensbeschreibung auf Systemebene*
Funktionale Beschreibung des Systemverhaltens.

Die Verhaltensbeschreibung auf Systemebene erfolgt meist ohne Zeitverhalten oder mit einer sehr vereinfachten Berechnungsabfolge, die auf dem Empfang von Nachrichten bzw. Funktionsaufrufen basiert. Die Modelle weisen häufig eine geringe Partitionierung in unterschiedliche Module bzw. C++-Klassen auf. Häufig finden sich monolithische Modelle wieder, die alle komplexen Systemzusammenhänge modellieren. Bezug nehmend zum Y-Diagramm in Abbildung 2.7, siedeln sich die Modelle auf dem äußersten Ring an. Die Kommunikation zwischen den Modulen ist häufig durch reine C++-Funktionsaufrufen oder über nachrichtenbasierte Transaktionen modelliert. In [4] werden diese Modelle auch als *algorithmic model* bezeichnet. Cai und Gajki verwenden in [23] die Begriffe *specification model* oder aber *untimed functional model*.

Architekturmodelle

Ausgehend von den funktionalen Modellen findet meist eine Verfeinerung der Struktur statt. Separate, kommunizierende SystemC-Module modellieren die Systemfunktionen.

Begriff 12. *Kommunikation-Arbitrierungs-Ebene*
Funktionale Beschreibung der Kommunikation ohne zeitliches Verhalten, häufig mit komplexen Datenstrukturen, sogenannten Nachrichten.

In Modellen der Kommunikation-Arbitrierungs-Ebene kommen TLM-Transaktionen ohne Zeitverhalten zum Einsatz. Lediglich die ausgetauschten Daten stehen im Vordergrund. Die Modellierung erfolgt vorrangig durch ausgetauschte Nachrichten. In den Modellen kommen häufig TLM-Transaktionsobjekte zum Einsatz. In [23] wird der Begriff *component assembly model* für Modelle dieser Abstraktionsebene verwendet. Wobei die Autoren hierunter auch die Verfeinerung der Berechnung sehen, welche hier getrennt betrachtet wird.

Begriff 13. *Verhaltensbeschreibung auf Komponentenebene*
Funktionale Beschreibung des Verhaltens von Komponenten. Die Systembeschreibung wird durch die Aggregation von Komponenten modelliert.

Hierbei wird meist eine erste Hardware/Software-Partitionierung berücksichtigt. Durch die Zuordnung zu Hardware bzw. Software besitzen diese Modelle meist ein abstraktes Zeitverhalten, das häufig mittels einer Abfolge von Ereignissen modelliert wird. Einzelne Ereignisse fassen meist eine Menge von Taktschritten zusammen. Die nachrichtenbasierte Kommunikation wird mit aggregiertem Zeitverhalten modelliert. Diese Gruppe der Modelle befindet sich meist auf der algorithmischen Ebene in Abbildung 2.7. Cai und Gajki verwenden in [23] die Begriffe *bus-transaction model*

Mikroarchitekturmodelle

Bei der letzten Klasse von Modellen wird das Systemverhalten detailliert modelliert. Vertreter dieser Klasse modellieren meist ein spezifisches Kommunikationsprotokoll und ein taktgenaues Zeitverhalten.

Begriff 14. *Funktionale Kommunikationsebene*
Funktionale Beschreibung des zyklenakkurat modellierten Nachrichtenaustauschs.

Kommunikationsmodule mit dieser Abstraktion können meist ohne großen Aufwand mit Register-Transfer-Level (RTL)-Modulen gekoppelt werden. Die Kommunikation wird häufig zyklengenau bzw. die einzelnen Protokollschritte detailliert modelliert. In [23] wird der Begriff *bus functional model* verwendet. Bei der Modellierung dieser Systeme mit SystemC sind häufig anwendungsspezifische Kanalmodelle verwendet, wobei auch detaillierte TLM-Kommunikationsmodelle, aus Gründen der Austauschbarkeit, verwendet werden.

Begriff 15. *Register-Transfer-Level*
Das Verhalten wird in kombinatorische Logik und Speicherelemente aufgeteilt. Ein Taktsignal synchronisiert die Speicherbauteile (Flip-Flops, Latches).

Dieser Verfeinerungsgrad setzt taktgenaue, fein partitionierte RTL-Modelle ein. In Bezug auf SystemC verwenden die Modelle häufig SystemC-Signale zur Modellierung der internen Kommunikation sowie getaktete oder ereignissensitive Prozesse. Die Begriffe *cycle-accurate computational model* und *implementation model* welche in [23] verwendet werden, können dieser Klasse zugeordnet werden.

Im Rahmen dieser Arbeit ist dies der niedrigste betrachtete Verfeinerungsgrad von Modellen. Die weiteren Verfeinerungen werden meist über Synthesewerkzeuge automatisiert ineinander überführt.

Begriff 16. *Gatterebene*
Das Verhalten wird Netzliste aus einzelnen Gattern (AND, OR, NOT,...) beschrieben.

In dieser Verfeinerungsstufe wird die Funktionalität durch Logikgatter, die einfache Logikfunktionen realisieren, modelliert. Eine Modellierung mittels SystemC ist möglich, wobei hier vorrangig SystemC-Signale sowie auf Signalwechsel sensitive Prozesse verwendet werden.

Begriff 17. *Layout-Ebene*
Unterschiedliche Zellen der Zieltechnologie, die platziert und geroutet sind.

Die meist finale Modellierungsebene, betrachtet die geometrische Anordnung von Zellen und deren Verbindung. Eine Modellierung mittels SystemC ist in diesem Detailgrad nicht mehr möglich.

Ziel des hier vorgestellten Ansatzes ist es das Abstraktionsniveau der Fehlerinjektion anzuheben und somit die Grundlage der Synthese zu verifizieren anstelle des Ergebnisses. Aus diesem Grund berücksichtigt diese Arbeit Netzlisten auf Gatterebene lediglich durch kleinere Anwendbarkeitsstudien.

2.5 Modellgetriebener Entwurf

Bei der modellgetriebenen Entwicklung, bzw. der modellzentrischen Entwicklung, handelt es sich um ein Vorgehensmodell [35, 103]. Im Vergleich zur quellcodezentrischen Entwicklung steht nicht der finale Quellcode im Mittelpunkt, sondern ein Modell. Die Entwicklungsingenieure verwenden Modelle, um die Struktur oder das Verhalten des Systems zu beschreiben. Grund für dieses Vorgehen sind die kontinuierlich steigenden Anforderungen der immer komplexeren Systeme. Wie beim Schritt von der Assemblersprache zu Hochsprachen, wie C++, verspricht die Erhöhung der Abstraktion die Handhabung von komplexeren Systemen. Die Handhabbarkeit der Komplexität beruht auf der Tatsache, dass durch die Modelle eine Abstraktion des komplexen Systems eingeführt wird, welche die Komplexität für den Anwender verringert. Durch automatisierte Abbildungsschritte, wie z. B. der Kompilierung oder Codegenerierung, wird von der abstrahierten Betrachtung auf die komplexere Realisierung automatisiert gewechselt. Bei den zugrunde liegenden Modellen kann zwischen dem plattformunabhängigen Modell (PIM) und dem plattformspezifischen Modell (PSM) unterschieden werden. Bei dem PIM existiert kein Bezug zur finalen

Plattform. Das Modell beschreibt vorrangig die Funktionalität des Systems. Bei dem PSM findet eine Abbildung der Software auf die finale Plattform statt, d. h. es wird z. B. eine Abbildung auf Prozessorkomponenten vorgenommen oder aber auf Kommunikationsbusse. Kerngedanke der modellzentrischen Entwicklung ist die Wiederverwendung, Überführung oder Generierung der verfeinerten Modelle aus den vorherigen Modellen. Häufig wird ein PIM zur Spezifikation der Funktionalität des zu entwickelnden Systems verwendet, das verfeinert wird, bis eine Abbildung auf die Plattform erfolgen kann. Hierbei werden oft die bestehenden Modelle mit Abbildungsinformationen erweitert sowie hardwarespezifische Erweiterungen vorgenommen. Im letzten Schritt wird aus dem PSM der finale Quellcode des Systems generiert. Durch die Verknüpfung der Modelle ist es möglich, am ursprünglichen Modell Änderungen vorzunehmen und durch die Verfeinerungs- und Generierungsprozesse diese Änderung zum finalen Quellcode zu propagieren.

2.5.1 Unified Modeling Language

Eine der bekanntesten Modellierungssprachen im Bereich der Softwareentwicklung ist sicherlich die Unified Modeling Language (UML) [83], die von der Object Management Group (OMG) standardisiert wurde. Hierbei spezifiziert die OMG die Metamodelle, welche die UML und deren Modelle definieren. Ein Metamodell definiert ein Modell, bzw. umgekehrt gilt ein Modell als eine Instanz des Metamodells. Die OMG verwendet hierzu das Konzept der Meta-Object-Facility (MOF) [81], die eine Metamodellierungsarchitektur darstellt. Sie besteht aus 4 Schichten. Die oberste Schicht (M3) ist die sogenannte Metametamodellebene und definiert die Sprache, die verwendet wird, um die eigentlichen Metamodelle, aus der Schicht M2, zu definieren. Die MOF-Schicht M1 beinhaltet die eigentlichen Modelle, die Instanzen der Metamodelle der Schicht M2 darstellen. Die unterste Schicht (M0), häufig auch als Datenschicht bezeichnet, sind Instanzen der Modelle, welche die realen Systeme beschreiben. Abbildung 2.9 zeigt die MOF in Bezug auf die UML.

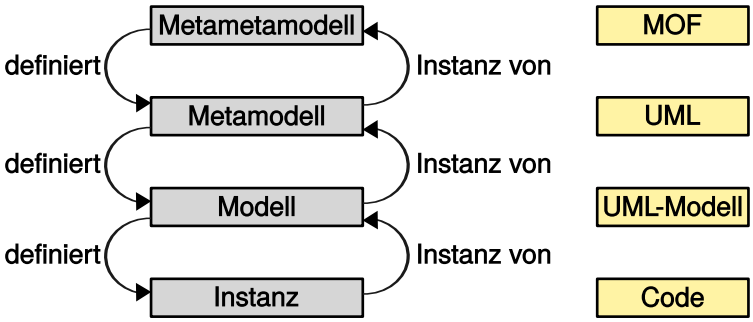


Abbildung 2.9: Struktur der MOF in Bezug auf die UML

Die UML bietet Modelle zur Spezifikation unterschiedlicher Softwareaspekte. Insgesamt 14 Diagrammtypen sind spezifiziert, um die Modellinformationen in einer grafischen Repräsentation und somit besser für den Anwender handhabbar, darzustellen. Die Diagramme lassen sich in die statische und dynamische Systemsicht

klassifizieren. Die statische Sicht beinhaltet das Klassendiagramm sowie das Kompositionsstrukturdiagramm. In [89] untersucht Petre die Verwendung unterschiedlicher Diagramme und stellt heraus, dass es sich beim Klassendiagramm um das am häufigsten verwendete Diagramm handelt. Sequenzdiagramme, Aktivitäts- oder Zustandsdiagramme modellieren die dynamischen Systemaspekte.

Durch die grafische Spezifikation hat sich die UML zur Dokumentation von Systemen etabliert. Viele Werkzeuge bieten explizite Erweiterungen zur begleitenden Dokumentation mit UML-Diagrammen, wie Sutherland in [109] am Beispiel DOORS zeigt.

Klassendiagramm

Das Klassendiagramm spezifiziert die Klassen einer objektorientierten Spezifikation sowie die Beziehungen zwischen den Klassen. Das Klassendiagramm berücksichtigt folgende Beziehungen:

- Schnittstelle,
- Generalisierung,
- Assoziation sowie
- Komposition und Aggregation.

Eine Klasse wird durch einen Klassennamen sowie ihre Attribute und Operationen beschrieben. Unterschiedliche Schlüsselwörter erlauben die detailliertere Spezifikation. Die häufig verwendeten Schlüsselwörter, `private`, `protected` und `public`, spezifizieren z. B. die Sichtbarkeit von Attributen oder Operationen.

Kompositionsstrukturdiagramm

Das Kompositionsstrukturdiagramm modelliert die interne Struktur einer Klasse. Eine Klasse kann sich aus unterschiedlichen Teilen (engl. Parts) zusammensetzen, die über Ports miteinander interagieren.

Zustandsdiagramm

Beim Zustandsdiagramm handelt es sich um ein dynamisches Modell, das die Zustände einer Klasse modelliert. Es spezifiziert die Ereignisse, die den Zustand einer Klasse ändern. Das Diagramm spezifiziert neben dem Zustandswechsel zusätzlich die durch die Klasse ausgelöste Aktion. Unterschiedliche Pseudozustände, wie z. B. der Start- und Endzustand, dienen als Steuerungselemente innerhalb der Spezifikation. Für diese Zustände existiert keine reale Belegung in der Klasse.

2.5.1.1 Erweiterung der UML

Bei der UML handelt es sich um eine generische Beschreibungssprache. Die UML bietet Erweiterungsmöglichkeiten, um eine Anpassung auf Anwendungsbereiche vorzunehmen. Mittels UML-Profilen ist es möglich, die UML auf eine spezielle Anwendungsdomäne anzupassen. Über Stereotypen ist eine Erweiterung des UML-

Metamodells möglich. Hierbei spezifiziert der Anwender Stereotypen, welche die bestehenden Modellierungsprimitiven, d. h. Elemente des Metamodells, erweitern. Die Stereotypen ermöglichen es, zusätzliche Informationen an die UML-Elemente zu annotieren.

In den letzten Jahren wurden unterschiedlichste Profile und Ableitungen entwickelt. Im Folgenden sind drei dieser Erweiterungen vorgestellt.

MARTE

Das UML-Profil mit der Bezeichnung Modeling and Analysis of Real-time and Embedded Systems (MARTE) [80] wird zur Modellierung von eingebetteten Echtzeitsystemen verwendet. Es beinhaltet unterschiedliche Stereotype, um Eigenschaften von Echtzeitsystemen zu spezifizieren. Das Profil beinhaltet Erweiterungen zur Modellierung von nichtfunktionalen Eigenschaften wie z. B. Zeitverhalten oder Ressourcenbedarf sowie eine daraus resultierende Ressourcenallokierung. So ist es möglich, den Ressourcenbedarf von Softwarekomponenten und die bereitgestellten Hardwareressourcen zu spezifizieren. Diese Informationen ermöglichen es, z. B. Schedulinganalysen durchzuführen.

SysML

Die Systems Modeling Language (SysML) [82] besteht nicht nur aus einer Erweiterung der UML mittels Profilen, sondern ist eine eigenständige Modellierungssprache, die auf der UML basiert. Sie besteht aus einer Untermenge der Diagramme der UML. Des Weiteren definiert sie zusätzliche Diagramme zur Modellierung von Systemanforderungen und Parametrisierungen. SysML findet vorrangig in der Domäne des Systems Engineering Anwendung. Sie unterstützt den Entwurf und Qualifikation von komplexen Systemen. Hierzu bietet sie Möglichkeiten zur Spezifikation von Systemanforderungen, der Systemarchitektur und insbesondere der physikalischen Aspekte von Systemen. Mittels des Stereotypen `blockProperty` ist es z. B. möglich, physikalische Eigenschaften von Komponenten zu spezifizieren.

EAST-ADL2

Eine weitere, aus der UML abgeleitete Modellierungssprache ist die Electronics Architecture and Software Technology - Architecture Description Language (EAST-ADL) [29]. EAST-ADL wurde speziell zur Spezifikation von Systemen der Automotive-Domäne entwickelt. Wie SysML geht sie über die reine Softwaremodellierung hinaus und bezieht die physikalischen Systemeigenschaften mit ein. Das EAST-ADL-Metamodell ist in die folgenden vier Abstraktionsebenen unterteilt:

- Feature (Vehicle)-Ebene,
- Analyseebene,
- Design-Ebene und
- Implementierungsebene.

2.5.2 IP-XACT

Bei IP-XACT [107] handelt es sich um ein standardisiertes Format, das auf der Extensible Markup Language (XML) basiert. Es wird vorrangig zur Spezifikation von elektronischen Komponenten und den auf ihnen basierten Entwurf verwendet. IP-XACT liegt als XML-Schema vor. Das standardisierte Schema erleichtert die Erstellung und Validierung konformer XML-Dateien und bietet somit die Grundlage zur automatisierten Konfiguration und Integration von Commercial off-the-shelf (COTS)-Komponenten. Bei COTS-Komponenten handelt es sich um nicht kundenspezifische Komponenten, häufig in Serienfertigung für ein gesamtes Marktsegment produziert. Anpassungen an Kundenanforderungen, erfolgt maximal über bereitgestellte Konfigurations- bzw. Parametrisierungsmöglichkeiten.

Bei der Verwendung existierender COTS-Komponenten besteht die Herausforderung, dass diese keine standardisierten Schnittstellen besitzen. Aus diesem Grund wird das Format IP-XACT verwendet, um die Schnittstellen dieser COTS-Komponenten zu beschreiben. Durch die wohldefinierte Struktur, die als XML-Schema beschrieben ist, ist eine automatisierte Verarbeitung möglich. Dies bietet das Potenzial für eine automatisierte Integration der Komponente und vereinfacht somit die manuelle Integration solcher COTS-Komponenten. IP-XACT spezifiziert nicht die funktionalen Eigenschaften der Komponenten. Stattdessen ist es möglich, auf existierende Modelle z. B. in VHDL oder SystemC zu verweisen.

Der IP-XACT Standard definiert unterschiedliche Modelle. Die Komponentenspezifikation beschreibt die Schnittstelle einer Komponente. Dies umfasst z. B. die physikalischen Signale, ihre Abbildung auf logische Schnittstellen, Adressinformationen, Registerbeschreibungen sowie deren Dokumentation.

Zur Beschreibung eines Entwurfs, aus spezifizierten Komponenten, wird die Design-Spezifikation verwendet. Dieses Modell ermöglicht, Komponenten, mit zuvor spezifizierten Schnittstellen, zu einem System zu verbinden. Des Weiteren ist die Parametrisierung der Komponenten vorgesehen. Der spezifizierte Entwurf des Systems dient dem Anwender als Datenbasis zur Systemverifikation und zur Überprüfung der Schnittstellenkompatibilität.

Kapitel 3

Existierende Methoden zur Zuverlässigkeitsbewertung

Der Abschnitt 3.1 dieses Kapitels stellt den aktuellen Stand der Technik und Forschung vor. Hierbei nimmt er Bezug auf gängige Praxis sowie vorherrschende Standards bei der Entwicklung sicherheitsrelevanter Systeme. Die vorgestellten Verfahren beziehen sich vorrangig auf die Entwicklung des Systems in der Gesamtheit. Abschnitt 3.2 stellt unterschiedliche Forschungsansätze vor. Sie beziehen sich auf Teilaspekte des hier vorgestellten Ansatzes, vorrangig um Ansätze zur Fehlerinjektion. Im nachfolgenden Abschnitt 3.3, werden ergänzend Ansätze zur modellbasierte Sicherheitsbewertung vorgestellt. Ansätze zur Fehlerspezifikation, die in den genannten Arbeiten verwendet werden, betrachtet Abschnitt 3.4. Hierzu werden die zuvor betrachteten Fehlerinjektionsansätze unter dem Gesichtspunkt des Spezifikationsformats analysiert sowie ergänzende Ansätze betrachtet. Abschnitt 3.5 fasst die Defizite zusammen. Weitere Herausforderungen, bei der Anhebung der Fehlerinjektion auf Systemebene, wurden in [79] publiziert.

3.1 Normen und Methoden

Dieser Abschnitt präsentiert zwei Normen zur Entwicklung sicherheitsrelevanter Systeme sowie daraus resultierende Herausforderungen. Insbesondere liefern die Normen Vorgehensmodelle zum Nachweis der Sicherheit. Der in dieser Arbeit entwickelte Ansatz zur Sicherheitsbewertung leistet Unterstützung bei diesen Prozessen, weswegen sie vorgestellt werden um die entwickelte Lösungen in den Kontext der Standards zu setzen. Zusätzlich werden zwei in dem Standard vorgeschlagene Techniken zur Analyse sicherheitskritischer Systeme im Detail betrachtet. Ziel dieser Ansätze ist das systematische Vorgehen zur Identifikation von Ursachen möglicher Ausfälle. Eines der beiden Verfahren wird zur Demonstration der Anwendbarkeit der in dieser Arbeit entwickelten Lösung aufgegriffen.

ISO 26262

Die ISO 26262 [51] ist eine Adaption der Norm IEC 61508 [46] zur funktionalen Sicherheit in der Automobildomäne. Die speziellen Anforderungen automobiler, elektrischer, elektronischer Systeme erforderten eine Adaption des allgemeinen Standards. Adressiert wird der komplette Lebenszyklus aller sicherheitsrelevanter elektrischen, elektronischen oder programmierbaren elektronischen Systeme (E/E/PE-Systems) im Automobil. Schwerpunkt bilden Anforderungen und Vorgehensempfehlungen für die Produktentwicklung. Hierbei behandelt die ISO 26262 die Phasen Entwurfsspezifikation, Konzeptionierung, Implementierung, Integration und Qualifikation. Der Standard bezieht sowohl die Hardware als auch die Software mit ein. Der Standard ist in zehn Teile gegliedert, die sich an den Phasen bzw. Aufgabenfelder der Produktentwicklung orientieren. Abbildung 3.1 zeigt eine Übersicht der einzelnen Bestandteile.

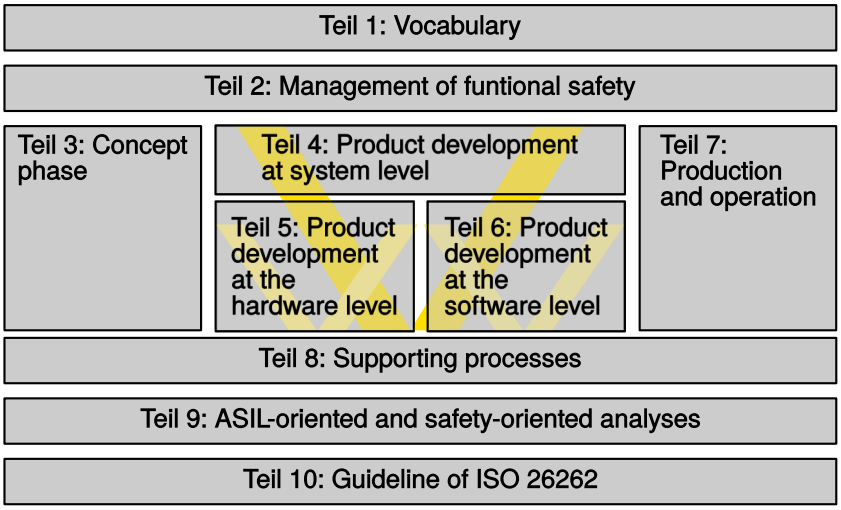


Abbildung 3.1: Struktur der ISO 262622

Die ISO 26262 beschreibt einen automobil-spezifischen Ansatz zur Risikobewertung. Dass sogenannte Automotive Safety Integrity Level (ASIL), bestimmt Anforderungen, an die Entwicklung der Systeme, abhängig von der Sicherheitskritikalität des Systems. Die Norm unterscheidet vier Stufen: ASIL-A, ASIL-B, ASIL-C und ASIL-D. Je höher die Stufe, desto restriktiver und umfangreicher sind die Anforderung an die Entwicklung der Komponente. Hierbei stellt ASIL-A die höchste Stufe dar. Zur Bestimmung der benötigten Stufe werden für jeden Fehler die Wahrscheinlichkeit des Auftretens, die Handhabbarkeit und der Schweregrad der Fehlerauswirkung ermittelt. Hierbei erfolgt die Quantifizierung in generellen Klassen. Tabelle 3.1 zeigt die definierten Klassen für die Auftrittswahrscheinlichkeiten. Die Klasse E1 wird definiert als: Ereignis, das weniger als einmal pro Jahr, für die Mehrheit der Fahrer, auftritt. Die ISO 26262 sieht auch zur Beurteilung der Schwere eines Fehlers unterschiedliche Klassen vor. Die Klasse S0 beschreibt Fehler, die nur materiellen Schaden aber keine Verletzung von Personen hervorruft. Die Klasse S3 hingegen kann zu tödlichen Verletzungen bei den involvierten Personen hervorrufen. In der Norm enthaltene Tabellen,

E0	E1	E2	E3	E4
Incredible	Very low probability	Low probability	Medium probability	High probability

Tabelle 3.1: Klassen der Auftrittswahrscheinlichkeiten ([51], Teil 3, Seite 9)

für Schwere (S), Auftrittswahrscheinlichkeit (E) und Handhabbarkeit (C), erlauben, anhand der zugewiesenen Klassen das benötigte ASIL zu bestimmen. Das angestrebte ASIL legt die Maßnahmen zur Bewertung des Systems fest, wie z. B. Reviews durch externe Personen oder funktionale Sicherheitsbeurteilungen. Zur Bestimmung der Auftrittswahrscheinlichkeit empfiehlt die ISO 26262 ein vereinfachtes Vorgehen. Die Häufigkeit des Fahrszenarios, in dem der Fehler auftritt, wird mit der Ausfallrate des Systems multipliziert.

Der Standard empfiehlt unterschiedliche Verfahren zur Bestimmung der Ausfallraten. Einer der ersten Schritte ist die Definition des zu untersuchenden Elements sowie dessen Interaktionen mit anderen Elementen vor allem dessen Abhängigkeiten. Die Analyse soll auf Methoden wie: Brainstorming, Checklisten, Failure Mode and Effects Analysis (FMEA), Fault Tree Analysis (FTA) oder Feldstudien zurückgreifen. Zur Verifikation von Sicherheitsmechanismen auf Systemebene empfiehlt der Standard z. B.

- Fehlerinjektionstests,
- die Nutzung von Expertenwissen zur Vorhersage von Fehlern oder
- das Heranziehen von Fahrversuchen im Feldversuch.

Zur Verifikation der Softwarearchitektur, im Speziellen zur Analyse der Stufe ASIL-D, empfiehlt der Standard:

- Code- bzw. Modellreviews,
- Simulation der dynamischen Aspekte des Entwurfs,
- Generierung von Prototypen,
- Kontrollflussanalysen sowie
- Datenflussanalysen.

Das heißt, für die Verifikation von Systemen mit den höchsten Anforderungen sieht die ISO Simulationen sowie Tests mit Prototypen vor. Virtuelle Prototypen stellen eine Erweiterung der Simulation dar und schießen die Lücke zum Test mit physikalischen Prototypen. Die reine Simulation von dynamischen Softwareaspekten stellt einen Spezialfall der virtuellen Prototypen dar, in dem auf die Modellierung von Hardwareaspekten verzichtet wird. Die Virtualisierung von physikalischen Hardwareprototypen ermöglicht, deren Vorteile in früheren Entwurfsphasen zu nutzen.

Die Nutzung von physikalischen Hardwareprototypen kann durch die Virtualisierung in frühere Entwurfsphasen verschoben werden. Wie bereits erwähnt, wird die Fehlerinjektion zur Sicherheitsbewertung vorgeschlagen, d. h. auch hierbei bildet die Fehlerinjektion in virtuelle Prototypen die Erweiterung der Tests mit physikalischen Prototypen. Damit inkludieren virtuelle Prototypen die von der ISO 26262 geforderten Methoden, erweitern diese aber gleichzeitig. Die Erweiterung unterstützen bei der Beherrschung der Komplexität zukünftiger Systeme.

IEC 61508

Die Norm IEC 61508 beschreibt ein zum ASIL ähnliches Konzept, das sogenannte Safety Integrity Level (SIL). Für die SIL-Bestimmung liegt die Wahrscheinlichkeit der gefährlichen Fehler (engl. dangerous failures) zu Grunde. Für ein SIL der Stufe 4 muss die Wahrscheinlichkeit zwischen 10^{-5} und 10^{-4} liegen. Zur Bestimmung des SIL müssen die Safe Fail Fraction (SFF) und die Hardware Fault Tolerance (HFT) bestimmt werden. Hierbei wird eine Reihe von Fehlerkategorien unterschieden:

- Sichere, erkannte Fehler (engl. safe detected (sd))
- Sichere, unerkannte Fehler (engl. safe undetected (su))
- Gefährliche, erkannte Fehler (engl. dangerous detected (dd))
- Gefährliche, unerkannte Fehler (engl. dangerous undetected (du))

Mit dem Bezeichner λ_{xx} wird die jeweilige Auftrittsrate bezeichnet, z. B. mit ISD die Auftrittsrate der sicheren, erkannten Fehler. Zur Berechnung der SFF wird die Summe der sicheren Fehler ($\lambda_{sd} + \lambda_{su}$) und der gefährlichen, erkannten Fehler (λ_{dd}) gebildet. Zusätzlich wird die Summe der sicheren Fehler ($\lambda_{sd} + \lambda_{su}$) und der gefährlichen Fehler ($\lambda_{dd} + \lambda_{du}$) errechnet. Die SFF bildet sich aus dem Quotienten der beiden Summen.

$$SFF = \frac{\lambda_{sd} + \lambda_{su} + \lambda_{dd}}{\lambda_{sd} + \lambda_{su} + \lambda_{dd} + \lambda_{du}}$$

Nachdem die SFF bestimmt wurde, muss der Anwender die HFT, also die Zahl der Hardwarefehler, die das System toleriert, bevor die Sicherheitsfunktion beeinflusst wird, bestimmen. Eine Tabelle legt basierend auf der berechneten HFT und SFF das resultierende SIL fest. Es existieren Abbildungen zwischen ASIL und SIL.

Wie aus dieser Betrachtung ersichtlich, benötigen viele der Teilschritte eine Abschätzung der Fehlerpropagierung. Bei der Verwendung der Tabellen zur ASIL-Einstufung, muss der Anwender die Auftrittswahrscheinlichkeit abschätzen. Bei der Berechnung des SIL muss die Aufteilung in detektierte und nicht detektierte Fehler vorgenommen werden. Diese Einteilung erfolgt meist durch die beteiligten Personen, weswegen die Standards Techniken wie Brainstorming empfehlen. Nichtsdestotrotz bleiben viele Entscheidungen subjektiv. Gerade bei komplexen, interagierenden Systemen ist es, für einzelne Personen, nicht ohne detailliertes Systemwissen möglich die Fehlerpropagation zu bestimmen.

FMEA

Die Fehlermöglichkeits- und -einflussanalyse (engl. Failure Mode and Effects Analysis (FMEA)) bzw. Erweiterungen dieser werden in der ISO 26262 als auch in der IEC 61508 als ein Ansatz zur Sicherheitsbewertung empfohlen. Die FMEA entstand bereits in den 1950er zur Beurteilung von Fehlfunktionen militärischer Systeme [30]. Heutzutage wird sie in vielen Domänen zum Zuverlässigkeitsnachweis verwendet.

Die FMEA beschreibt ein Vorgehen, um systematisch Fehlerursachen und die resultierenden Fehlereffekte aufzuzeigen. Aus diesem Grund ist sie eine präventive Methode zur Vermeidung und Erkennung von Fehlern in frühen Stadien der Entwicklung. Spezielle Formblätter dokumentieren für unterschiedliche Komponenten die Fehlermodi und ihre Effekte auf das Gesamtsystem.

Das Konzept der FMEA ist so generell, dass unterschiedlichste Ausprägungen existieren: Es existiert die System-FMEA, die Konstruktions-FMEA, die Prozess-FMEA und die Design-FMEA. Die unterschiedlichen FMEAs beziehen sich auf unterschiedliche Objekte oder Prozesse des Systementwurfs. Die Analysen bauen zum Teil aufeinander auf. Die Ergebnisse der System-FMEA bilden die Grundlage der Konstruktions-FMEA, die wiederum Grundlage für eine Prozess-FMEA sein kann. In der Praxis ist eine eindeutige Trennung, im Speziellen zwischen System- und Konstruktions-FMEA nicht notwendig. In dieser Arbeit wird implizit die System-FMEA angenommen, wenn nicht anderweitig darauf hingewiesen wird.

Wird das FMEA-Formblatt zur Beschreibung des Ausfallrisikos von Komponenten erweitert, wird die Analyse als FMECA (engl. Failure Mode, Effects and Criticality Analysis (FMECA)) bezeichnet. Das Ausfallrisiko wir häufig über die Risiko-Prioritäts-Zahl (engl. Risk Priority Number (RPN)) beschrieben. Die RPN ist von Faktoren wie der Auftrittswahrscheinlichkeit, der Schwere und der Entdeckungswahrscheinlichkeit des Fehlerzustands abhängig. Die letzte hier betrachtete Erweiterung, die auch vorrangig in den Prozessen der IEC 6150 und der ISO 26262 vorgesehen wird, ist die FMEDA (engl. Failure Mode, Effects and Diagnostic Analysis (FMEDA)) Sie erweitert die FMEA um quantifizierbare Fehlerdaten. Dies umfasst ähnlich zur FMECA die Ausfallwahrscheinlichkeit aber auch die Verteilung der Fehlerzustände. Die FMEDA unterteilt die Ausfallwahrscheinlichkeiten in unterschiedliche Fehlerkategorien ($\lambda_{sd}, \lambda_{su}, \lambda_{dd}$ und λ_{du}). Die Kategorien und die darauf aufbauende Berechnung der SFF sind im Abschnitt zur IEC 61508 detailliert vorgestellt.

Die FMEA ist meist eine qualitative Methode, bei der unterschiedliche Fachleute in einem gerichteten Prozess ihr Systemwissen einbringen. Dies beinhaltet Abschätzungen wie zum Beispiel die Wahrscheinlichkeit des Auftretens oder des Entdeckens von Fehlern. Diese Abschätzungen beruhen häufig auf Erfahrungen mit ähnlichen Systemen oder aber auf Datenblättern von Systemteilen. Zur Erleichterung der Abschätzung kommen vordefinierte Quantifizierungsstufen, wie *häufig* oder *selten*, zum Einsatz. Nichtsdestotrotz bleiben viele der Abschätzungen stark durch die involvierten Personen geprägt, weswegen eine breite Beteiligung bei der Durchführung gefordert wird.

Bei der FMEA handelt es sich um eine Bottom-Up-Analyse, d. h. ausgehend von den Fehlern in den Komponenten wird auf die Fehler des Gesamtsystems geschlossen.

Hierfür existieren standardisierte Vorgehensmodelle, nichtsdestotrotz handelt es sich um einen sehr zeitaufwendigen Prozess, insbesondere bei komplexen Systemen. Das Vorgehen ist meist sehr ähnlich.

1. Bestimmung der Komponenten des Gesamtsystems sowie deren Funktion.
2. Festlegung aller potenzielle Fehlerursachen und Fehlerraten für die Komponenten. Hierbei wird meist auf Datenblätter, Erfahrungen mit bestehenden Systemen oder Zuverlässigkeitshandbücher zurückgegriffen.
3. Beschreibung der Fehlerauswirkungen, der zuvor identifizierten Fehlerursachen, auf das Gesamtsystem.
4. Untersuchung des Systems nach Fehlererkennungs- und Fehlerkorrekturmechanismen.
5. Untersuchung ob externe Randbedingungen die Auftrittswahrscheinlichkeiten, der identifizieren Fehlerursachen, reduzieren.
6. Abschätzung der Schwere der identifizierten Fehlerauswirkungen.
7. Abschätzung der Auftrittswahrscheinlichkeit von Fehlerursachen.
8. Formulierung von Handlungsempfehlungen zur Reduktion der vom System ausgehenden Gefahren.

Wie die skizzierte Abfolge nahelegt, beinhalten viele Schritte Abschätzungen oder beziehen sich auf allgemeine Erfahrungen der beteiligten Personen.

FTA

Neben der FMEA ist die sogenannte Fehlerbaumanalyse (engl. Fault Tree Analysis (FTA)), ein weiterer Ansatz zur Spezifikation und Analyse der Fehlerpropagierung. Hierbei wird das System in Komponenten unterteilt und die Fehlerpropagierung mittels Und- und Oder-Relationen spezifiziert. Die Fehlerursachen werden als Blattkonten dargestellt und als Basic Event bezeichnet. Abbildung 3.2 zeigt schematisch einen Fehlerbaum. Hierbei wirkt sich ein Fehler am Eingang des Service (S_{in}) auf den erbrachten Dienst (S_{out1}, S_{out2}) aus. Der Fehler im erbrachten Dienst propagiert sich wiederum zum Top-Event, der Fehlerauswirkung. Das Top Event stellt im Allgemeinen die Verletzung eines Safety Goals dar.

Im Gegensatz zur FMEA handelt es sich um einen Top-Down-Ansatz. Ausgehend von einer potenziellen Fehlerauswirkung wird innerhalb des Gesamtsystems nach möglichen Fehlerursachen gesucht. Hierbei wird die Fehlerpropagation durch das System analysiert und in der Baumstruktur dokumentiert. Das hierbei kritische Cut Set sind alle Blattknoten, welche die Fehlerauswirkung hervorruft. Das heißt, für die Analyse ist es kritisch, die Fehlerpropagation und die Fehlerbehebung innerhalb des Systems zu kennen. Gerade bei stark vernetzten, komplexen Systemen stellt dies eine große Herausforderung dar.

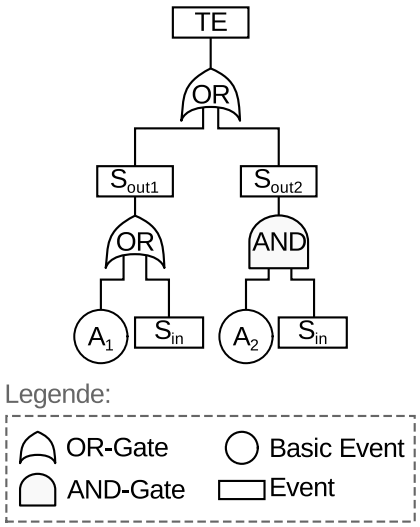


Abbildung 3.2: Schematische Darstellung eines Fehlerbaumes

Eine Erweiterung des klassischen Fehlerbaums ist der sogenannte Komponentenfehlerbaum (engl. Component Fault Tree (CFT)) [44, 57]. Hierbei wird kein systemweiter Fehlerbaum erstellt, sondern für jede Komponente wird ein separater Fehlerbaum spezifiziert. Hierbei wird für jede Fehlerauswirkung an den Ausgangsports der Komponente ein Fehlerbaum mit der Fehlerauswirkung als Wurzelelement erzeugt. Die Eingangsports und die, in der Komponente, entstehenden Fehler sind die Blattknoten. Eine zusätzliche Spezifikation gibt an, wie sich das Gesamtsystem, aus den einzelnen Komponenten, zusammensetzt. Die Spezifikationen ermöglicht die Generierung der systemweiten Fehlerbäume.

3.2 Testmethode: Fehlerinjektion

Im Folgenden wird der zum Zeitpunkt der Anfertigung der Arbeit vorherrschende Stand der Forschung vorgestellt. Der erste Abschnitt stellt unterschiedliche Ansätze zur Fehlerinjektion vor und nimmt Bezug auf ihre Vor- und Nachteile. Der zweite Abschnitt beschäftigt sich mit Ansätzen zur Fehlerspezifikation, also der Beschreibung des zu injizierenden Verhaltens.

Eine Möglichkeit, die Techniken zur Fehlerinjektion zu klassifizieren ist die Einteilung anhand des Zielsystems. Hierbei wird zwischen der Hardware-Fehlerinjektion, der Software-Fehlerinjektion und der Fehlerinjektion in Simulationsmodelle (engl. Simulation Fault Injection (SFI)) unterschieden. Abbildung 3.3 zeigt das Schema zur Klassifizierung der Fehlerinjektionsansätze. Bei der Hardware-Fehlerinjektion kann weiter unterschieden werden ob diese kontaktlos durch externe Quelle, z. B. durch Ionenstrahlung, oder kontaktbehaftet erfolgt. Unter die kontaktbehaftete Fehlerinjektion fällt das Setzen von Signalen mittels Sonden oder das Einbringen von Fehlerinjektions-Sockets zwischen Platine und Chip. In dieser Arbeit wird ein Simulationsmodell als Analysegrundlage verwendet, aus diesem Grund können die Techniken zur

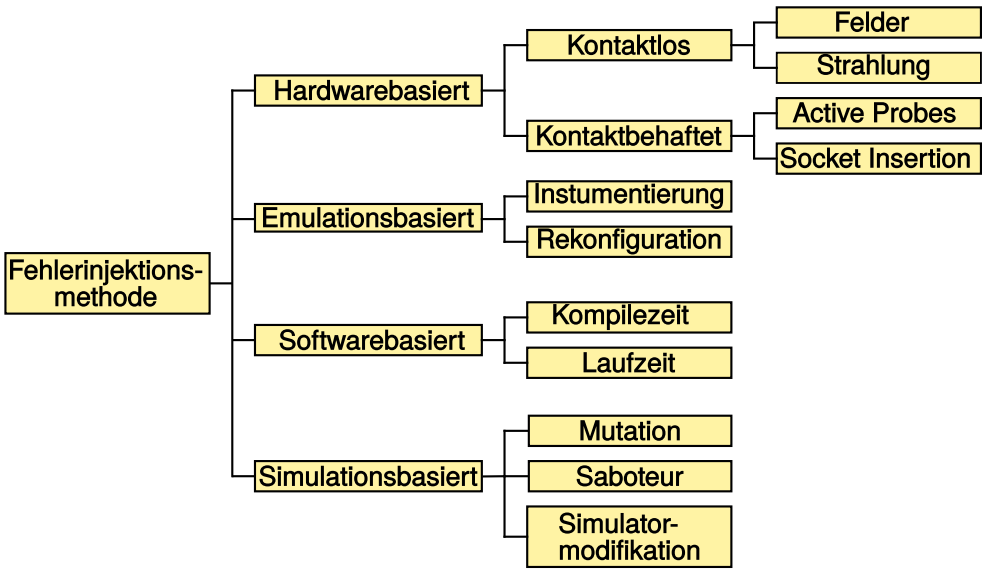


Abbildung 3.3: Klassifikation von Fehlerinjektionsansätzen nach [45] und [120]

Hardware-Fehlerinjektion nicht übertragen werden, lediglich im Kontext der Fehlerspezifikation müssen diese Ansätze beleuchtet werden. Eine Abwandlung verwendet anstelle des Zielsystems eine Emulation mit z. B. einem Field Programmable Gate Array (FPGA). Eine weitere Gruppe der Fehlerinjektion basiert auf der Software des Systems. Ein Nachteil hierbei ist, dass Fehler nur in Informationen auf welche von der Software zugegriffen wird, injiziert werden können. Für eine ausführliche Betrachtung der einzelnen Klasse sei auf die Veröffentlichungen [45] und [120] verwiesen.

Die SFI weist meist eine bessere Beobachtbarkeit, Steuerbarkeit und damit eine höhere Reproduzierbarkeit auf. Neben der eigentlichen Software können auch Hardwareaspekte berücksichtigt werden. Auf der anderen Seite sind die Analysen lediglich so aussagekräftig wie das zugrunde liegende Modell. Unter der Annahme eines aussagekräftigen Modells weist somit die SFI erhebliche Vorteile gegenüber den anderen Methoden auf. Des Weiteren kann sie als Host-Code-Ausführung einer Software gesehen werden und weist somit zu einem bestimmten Grad Parallele zur softwarebasierten Fehlerinjektion, die im Regelfall auf dem Zielsystem erfolgt, auf. Die im Folgenden betrachteten, aktuellen Forschungsansätze beschränken sich aus diesen Gründen auf die SFI. Wobei Techniken der reinen Software-Fehlerinjektion teilweise auf die Simulation, die eine reine Softwareausführung darstellt, angewandt werden können. Bei der Software-Fehlerinjektion kommen aber auch Ansätze wie eine virtuelle Ausführungsumgebung mit Injektionsschnittstelle [90] oder die Injektion in Bibliotheksaufrufe [70] zum Einsatz. Diese Ansätze sind grundsätzlich anwendbar, würden aber auf die komplette Simulation abzielen anstelle auf das simulierte System, weswegen sie hier nicht weiter betrachtet werden. Ansätze zur Hardware-Fehlerinjektion werden im Folgenden ausgeblendet.

Die unterschiedlichen Ansätze der SFI können nach einer Vielzahl von Kriterien gruppiert werden. Eine erste Gliederung der Ansätze erfolgt anhand der Simulati-

onssprache. Weitverbreitete Sprachen zur Simulation und vor allem zur Synthese von Hardware-Systemen sind VHDL und Verilog. Somit ist es naheliegend, dass eine Vielzahl von Ansätzen zur Fehlerinjektion [53, 58, 106] auf diesen beiden Sprachen basieren. Wie bereits vorgestellt besitzen SystemC und VHDL bei der RTL-Modellierung eine ähnliche Modellierungsmächtigkeit. SystemC bietet hingegen klare Vorteile bei der Modellierung auf System- und Transaktionsebene. Aus diesem Grund behandelt dieser Abschnitt vorrangig Ansätze, die auf SystemC basieren, da diese ein breiteres Spektrum an Abstraktionsebenen umfassen.

Ein weiterer Ansatz zur Klassifikation von Fehlerinjektionsansätzen ist anhand der verwendeten Technik. Über die Jahre haben sich drei unterschiedliche Techniken zur Fehlerinjektion herausgebildet:

- Saboteur-Ansätze,
- Mutations-Ansätze und
- Fehlerinjektion durch Simulatormodifikation

Die folgenden Abschnitte beziehen sich auf diese Unterteilung. Viele der in VHDL oder Verilog vorliegenden Ansätze lassen sich auf SystemC übertragen, weswegen vereinzelnd auf solche Ansätze eingegangen wird. Misera et al. stellt in [76] die drei grundlegenden Fehlerinjektionstechniken in SystemC vor, die von VHDL-Ansätzen inspiriert wurden.

Saboteur-Ansatz

Der Saboteur-Ansatz fügt dem Signal- bzw. Kommunikationspfad zusätzliche Komponenten hinzu. Diese Module erlauben die Verfälschung der ausgetauschten Informationen. In [53] wird die Werkzeugumgebung MEFISTO (Multi-level Error/Fault Injection Simulation Tool) für VHDL-Modelle vorgestellt. Einer der beiden verfolgten Ansätze beruht auf dem Hinzufügen von seriellen und parallelen Saboteur-Modulen. Bei den seriellen Saboteur-Modulen wird der Signalpfad zwischen Treiber und Empfänger aufgetrennt und hierdurch Fehler injiziert. Beim parallelen Saboteur-Modul wird das Signal durch einen zusätzlichen Treiber, dem Saboteur, geschrieben und abhängig der `Resolution`-Funktion des Signaltyps aufgelöst.

Auch Yung-Yuan Chen et al. haben, anhand mehrerer Beispiele [25, 26, 28], die Verwendung von Saboteur-Modulen zur Fehlerinjektion gezeigt. Ein sogenanntes Fault Injection Module (FIM) wird in [25] in den Kommunikationspfad zwischen einem Master-Device und einem hierarchischen Kanal eingefügt. Das Modul injiziert Fehler in die übertragenen Daten und Steuersignale. In [28] wurde diese Strategie angewendet, um Fehler in ein Modell eines System-on-a-Chips (SoCs) zu injizieren. Hierbei adressiert der Ansatz zyklengenaue und TLM-Modelle, indem er FIMs mit Schnittstellen für `sc_signal` und `sc_fifo` vorstellt. Im Falle von `sc_signal` wird das FIM parallel zum Kommunikationskanal geschaltet und per Multiplexer wird entweder das korrekte oder das fehlerhafte Signal gesetzt. Im `sc_fifo` Fall wird die Verbindung aufgetrennt und das FIM eingefügt. Vorteil ist, dass beide Schnittstellen durch

SystemC standardisiert sind und damit auf Modelle der gleichen Abstraktionsebene angewandt werden können, jedoch nicht über Abstraktionsebenen hinweg.

Ein weiterer Ansatz, der auf zusätzlichen Modulen basiert, wurde von K. Rothbart et al. in [101] vorgestellt. Hierbei wurde im Kontext einer Security-Analyse ein abstraktes SystemC-Modell einer Smart Card mit Fehlerinjektionsmodulen erweitert. Hierbei durchtrennen FIMs die Kommunikationspfade zwischen den unterschiedlichen Funktionsblöcken. Die FIMs injizieren mehrere Stuck-at-Fehler in die Variablen und Signale der Kommunikation zwischen Funktionsblöcken. Die unterschiedlichen FIMs müssen die gleichen Schnittstellen wie die Funktionsblöcke bereitstellen, damit die Funktionsblöcke unveränderten weiterverwendet werden können.

Der Ansatz des Saboteurs eignet sich vorrangig bei standardisierten Schnittstellen wie z. B. bei `sc_signal`. Misera et al. stellt in [73–75] einen Ansatz mit Saboteur-Modulen vor, der Stuck-at-Fehler in kombinatorische Schaltungen injiziert. Der Ansatz betrachtet Verhaltensbeschreibungen als Blackbox und injiziert Fehler lediglich in die Eingangssignale. Ein Saboteur-Modul leitet entweder den anliegenden Signalwert oder den Stuck-at-Fehlerwert weiter. Die Injektion wird über eine Steuerleitung aktiviert bzw. deaktiviert. Dies ermöglicht die Injektion von transienten Fehlern. Ein sogenannter *Fehlerbus* fasst die so entstehenden Steuerleitungen zusammen. Durch Setzen mehrere Steuerleitungen ist es möglich, mehrere Stuck-at-Fehler gleichzeitig zu injizieren. Die Steuerleitungen der Fehlerinjektoren werden als zusätzlicher Stimulus des Testpattern-Generators gesehen.

Vorteil der Saboteur-Technik ist, dass der Anwender die existierenden Simulationskomponenten nicht verändern muss. Lediglich die Struktur der Systemsimulation wird angepasst. Bei einer hierarchischen Struktur kann dies auch in der Strukturbeschreibung von Teilkomponenten erfolgen. Die Verhaltensbeschreibung der einzelnen Simulationseinheiten wird nicht verändert. Hierbei liegen aber genau die Nachteile des Ansatzes. Eine Fehlerinjektion erfolgt nur in den ausgetauschten Informationen. Die Modifikation von internen Informationen der Simulationseinheiten ist nicht möglich. Hierzu zählen vor allem nichtfunktionale Eigenschaften, wie z. B. das Zeitverhalten von Simulationseinheiten. Diese Eigenschaften sind häufig intern modelliert und werden nicht über Schnittstellen nach außen gegeben. Ein weiterer Nachteil ist die Bereitstellung einer einheitlichen Schnittstelle zwischen Komponenten. Um die gleiche Saboteur-Komponente in unterschiedlichen Simulationen, bzw. mit unterschiedlichen Simulationseinheiten, verwenden zu können, müssen die Komponenten über eine standardisierte Schnittstelle verfügen. Dies ist in SystemC-Simulationen vor allem mittels SystemC-Ports und TLM-Schnittstellen möglich. Bei abstrakten funktionalen Modellen, die meist über applikationsspezifische Schnittstellen kommunizieren, ist dieser Ansatz nicht tragfähig, da der Anwender für jede Schnittstelle ein angepasstes Adaptermodul bereitstellen müsste.

Mutationen

Mutationen verändern die Funktionalität einzelner Simulationskomponenten. Dies wird erreicht, indem die Verhaltensbeschreibung oder die Strukturbeschreibung einer

Komponente abgeändert wird. Es liegen demzufolge eine korrekte und eine fehlerhafte Beschreibung der Komponente vor. Ein Beispiel ist die Änderung der Funktionalität eines AND-Gatters in ein OR-Gatter. Häufig implementiert der Anwender sowohl das korrekte als auch das fehlerhafte Verhalten und über Steuereingänge wird das gewünschte Verhalten ausgewählt. Eine weitere Technik der bereits vorgestellten Werkzeugumgebung MEFISTO [53] ist die Injektion von Fehlern über die Mutation von VHDL-Modellen. Hierbei beinhaltet das Simulationsmodell zwei Implementierungen der Komponente und über die Konfigurationsmöglichkeiten der VHDL wird die korrekte oder die mutierte Funktionalität ausgewählt.

Franco Fummi et al. präsentieren mehrere Ansätze [16, 34] zur Fehlerinjektion basierend auf Mutationen. Der in [16] vorgestellte Ansatz ändert TLM-2.0-Schnittstellen ab, um Fehler in den Datenpfad von transaktionsbasierten Modellen zu injizieren. Ein weiterer vorgestellter Ansatz [34] fügt Fehlercode, sogenannte Codemutationen, bei der Übersetzung von VHDL-Simulationsmodellen nach SystemC ein. Die mutierte Systembeschreibung injiziert Fehler in `sc_bits` und ermöglicht die Injektion von transienten Bitfehlern. Der Ansatz verwendet für unterschiedliche Abstraktionsebenen, unterschiedliche Mutationsansätze. Vorteil ist, dass auch die internen Informationen durch die Fehlerinjektion verfälscht werden können. Ein Nachteil dieses Ansatzes ist, dass häufig mehrere Implementierungen vorgehalten werden, die eine Änderung der funktionalen Spezifikation des Systemmodells bedingen.

Ein Ansatz, der versucht diesen Nachteil zu minimieren, wurde von Lisherness et al. mit ihrem Werkzeug SCEMIT [68] vorgestellt. Die bisherigen Ansätze hatten gemein, dass sie Simulationsmodelle abändern. Entweder verändern sie die Struktur der Systemsimulation oder die Implementierung der Simulationseinheiten. Beide Fälle gehören zur Klasse der invasiven Ansätze. SCEMIT [68] verwendet den GCC Compiler, um das Zwischenformat des Compilers abzuändern. Der Ansatz unterstützt arithmetische Operatorenersetzungen, die Ersetzung von Variablen oder Konstanten durch konstante Werte sowie die Ersetzung von Relationsoperatoren. Am Beispiel eines C++/SystemC-Modells wird die Fehlerinjektion demonstriert. Um den Ansatz anzuwenden, wird GCC Version 4.5.0 oder höher benötigt, da frühere Versionen keine Plug-in-Schnittstelle anbieten. Durch die Änderung interner Variablen ist es z. B. möglich für eine Komponente unterschiedliches Zeitverhalten zu stimulieren, da dieses häufig als interne Variable vorgehalten wird. Der Ansatz hat den Nachteil des Vorhaltens unterschiedlicher Implementierungen im Modell umgangen, leider eröffnet sich dadurch ein neuer Nachteil. Es entsteht eine Bindung zu dem verwendeten Compiler und damit ist nicht sichergestellt, dass der Ansatz proprietären Entwicklungsumgebungen, wie z. B. das Synopsys Virtualizer Studio [111], unterstützt. Diese Entwicklungsumgebungen stellen häufig einen eigenen, optimierten Compiler bereit.

Ein weiterer nichtinvasiver Ansatz [112] verwendet den GNU Debugger GDB um Variablen während der Simulation zu mutieren. Hierbei wird die Systemsimulation mittels Breakpoints angehalten, die Veränderung am Systemzustand vorgenommen und die Simulation weiter ausgeführt. Durch die Modifikation reiner C++-Datentypen ist eine Injektion sowohl in SystemC-Ports, C++-Variablen als auch in Transakti-

onsobjekte möglich. Leider ist wie im vorherigen Ansatz eine Abhängigkeit zum GDB vorhanden. Des Weiteren ist es notwendig, das auszuführende Simulationsmodell mit Debugging-Informationen zu kompilieren. Ein weiterer Nachteil ist die Abhängigkeit von der Anzahl verfügbarer Hardware-Breakpoints. Die Autoren weisen darauf hin, dass bei der Verwendung von durch die Software emulierten Breakpoints die Performanz des Ansatzes stark zurückgeht. Die verwendete Architektur stellt vier Hardware-Breakpoints zur Verfügung. Dies ermöglicht die performante Injektion von bis zu vier Fehlern gleichzeitig.

Beltrame et al. präsentieren in [8, 9, 15] einen nichtinvasiven Ansatz, der es erlaubt SystemC/C++-Variablen in der Simulation zu mutieren. Hierzu werden über eine Werkzeugkette die Membervariablen der Simulationsklassen für Python zugreifbar gemacht. Dieser Ansatz funktioniert sowohl mit privaten als auch mit öffentlichen Membervariablen. Die Steuerung der Simulation als auch die Änderung von Variablenbelegungen wird durch die Python-Simulation vorgenommen. Dieser Ansatz ist sehr vielversprechend, da er viele, der in dieser Arbeit angestrebten Eigenschaften besitzt. Wird angenommen, dass die Fehlerinjektion eine Variable dauerhaft überschreibt, kann dieser Ansatz jedoch zu einem sehr starken Synchronisationsaufwand zwischen Python und der SystemC-Simulation führen. Dies begründet sich in der Tatsache, dass Python sicherstellen muss, dass eine durch die Injektion veränderte Variable nicht durch die Simulation zurückgesetzt wird. Somit muss Python bei jedem Delta-Schritt die Variable prüfen und gegebenenfalls erneut setzen. Bei abstrakten, nicht zeitbehafteten Modellen ist dies nicht möglich, da die Kontrolle an SystemC für mindestens einen Delta-Zyklus abgegeben wird.

Ansätze, die auf der Ersetzung bzw. Weiterleitung von Datentypen in den Simulationsmodellen beruhen sind [69, 105, 114]. Der Ansatz [69] führt einen erweiterten Datentyp ein, der zum korrekten Wert auch alle, durch die Injektion stimulierten Fehlwerte speichert. Dies wird verwendet, um unterschiedliche Fehlerinjektionen parallel durchzuführen. Die Fehlerinjektion verwendet Techniken der nebenläufigen, komparativen Simulation. Ein Ansatz, der auf Datentypersetzung beruht, wurde von Shafik et al. in [105] vorgestellt. Bei diesem Ansatz wird eine Auswahl an Datentypen angegeben, welche die ursprünglichen Datentypen ersetzen und ein Abändern des Zustands der Variable erlauben. Hierbei behandelt der Ansatz vor allem Datentypen, die vorrangig auf der RTL-Ebene vorzufinden sind. Dies umfasst z. B. `sc_logic`, `sc_int`, `sc_lv` oder `sc_bv`. Beide Ansätze verwenden C++-Templates, um eine weitestgehend generische Implementierung bereitzustellen. Der Ansatz [114] führt keine Ersetzung bzw. Kapselung der Variablen durch, sondern registriert die Speicheradresse unter einem Identifikator. Dies erfolgt innerhalb des Quellcodes durch Hinzufügen des Aufrufs der Registrierungsfunktion. Über den Speicherzugriff wird der Wert der Variable geändert. Vedder et al. machen in [114] keine Angaben, wie der injizierte Wert über mehrere Schreibzugriffe der Simulation aufrecht gehalten werden kann.

Wie bereits die reine Anzahl der unterschiedlichen Forschungsansätze zeigt, ist die Mutation eine vielversprechende Technik zur Fehlerinjektion. Einer der entscheidendsten Vorteile ist die Verfälschung interner Variablen. Die vorgestellten Techniken lassen sich in invasive und nichtinvasive Ansätze unterscheiden. Die invasiven

Ansätze halten häufig mehrere Implementierungen vor. Das Vorhalten alternativer Implementierungen führt zu einer erheblichen Modifikation des originalen Simulationsmodells. Die nichtinvasiven Ansätze hängen von dedizierten Werkzeugumgebungen ab und hierdurch ist nicht sichergestellt, dass der Anwender proprietäre Werkzeugumgebungen verwenden kann.

Simulatormodifikation

Eine rein nichtinvasive Technik ist die Änderung des Simulators bzw. dessen Simulationsablauf. Misera et al. stellen in ihrer Übersicht [76] einen Injektionsansatz mit dem abgeänderten `sc_logic`-Datentyp vor. Hierbei wird die Aufzählung um Werte für einen überschreibenden High- und Low-Pegel sowie einem speziellen Release-Zustand erweitert. Durch diese Erweiterung lassen sich Stuck-at-1 und Stuch-at-0 Fehler in die Simulation einbringen. Mit dem Abändern der Resolution-Tabelle wird sichergestellt, dass ein injizierter Fehler nicht durch ein erneutes Setzen des Signals überschrieben wird.

Entscheidender Vorteil dieses Ansatzes ist die unveränderte Verwendung der ursprünglichen Simulationsmodelle. Nachteil ist, dass der Ansatz häufig geänderte bzw. dedizierte Simulatoren oder Werkzeugumgebungen einsetzt. Aber gerade beim industriellen Einsatz von Simulationen kommen häufig spezielle, optimierte Simulatoren zum Einsatz, die nicht ausgetauscht bzw. abgeändert werden können.

3.3 Modelbasierte Sicherheitsbewertung

Im vorherigen Abschnitt wurden Ansätze zur Fehlerinjektion betrachtet, die auf einem ausführbaren, funktionalen Modell des Systems beruhen. Der Vollständigkeit halber werden in diesem Abschnitt Ansätze zur modellbasierte Sicherheitsbewertung (engl. Model-Based Safety Assessment (MBSA)) vorgestellt. Hierbei kommen vorrangig formalisierte Modell zum Einsatz, die einzelne Aspekte, wie z. B. die Architektur, des Systems abbilden. Bei der Sicherheitsbewertung eines Systems ist das Ziel das Ausfallrisiko von Systemdiensten abzuschätzen. Wie der Name Risiko impliziert handelt es sich um eine probabilistische Betrachtung, wie bereits bei der Betrachtung der Sicherheitsnormen vorgestellt. Aus diesem Grund werden häufig probabilistische Modelle zur Bestimmung der Ausfallwahrscheinlichkeit verwendet. Bei den Modellen steht die Modellierung der Auftritts-, Propagierung- und Behebungswahrscheinlichkeit von Fehlern im Vordergrund. Erweitert werden die Ansätze durch eine Systemarchitekturmodellierung, um die Fehlerpropagierung zu berücksichtigen. Bekannte Vertreter sind die bereits motivierten Fehlerbäume aber auch traditionelle Methoden wie Event Trees, Markov-Ketten oder stochastische Petri-Netze.

Hierbei bilden Fehlerbäume, Event Trees und Reliability Block Diagrams (RBDs) die Klasse der kombinatorischen Formalismen. Der Ansatz Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS)[86] verwendet ein Architekturmodell, das mit logischen Fehlerannotationen angereichert wird. Abhängig des Entwurfs der Architektur können FTA oder FMEA Analysen durchgeführt werden.

Durch die Annotation von quantitativen Fehlerinformationen stellt der Ansatz eine quantitative Fehlereffektanalyse bereit. Der Ansatz ist vergleichbar mit der weiter oben, bei der Fehlerbaumanalyse vorgestellten, Komponentenfehlerbaumanalyse (CFT) [44, 57].

Ein weiterer Ansatz stellt das Rahmenwerk AltaRica3.0 [6] dar. Der Kern des Ansatzes bildet das mathematische Modell: *Guarded Transition System*. Das Guarded Transition System ist eine zustandsbasierte Beschreibung, die eine Generalisierung traditioneller Ansätze wie RBD, Pertri-Netze oder Markov-Ketten darstellt. Es verwendet Elemente wie Zustände, Ereignisse, bedingte Übergänge und Variablenzuweisungen. AltaRica3.0 bietet eine Reihe von Analysewerkzeugen, die auf diesem Guarded Transition System aufbauen, z.B. die Generierung von Petri-Netze wie in [93] dargelegt oder Fehlerbäume wie in [92] präsentiert.

Weiter Ansätze setzen auf traditionellen RBDs [77] oder Markov-Ketten [59] auf. Ziel der Ansätze ist es die Systemarchitektur mit Informationen zur Fehlerausbreitung oder -behebung anzureichern und quantitative Aussagen über die Fehlerauswirkungen auf Systemebene zu treffen.

Bernardi et al. präsentieren in [10] eine Übersicht über Modellierungs- und Analyseansätze basierend auf der UML. Die Ansätze annotieren zusätzliche Informationen an bestehende UML- Modelle um eine Zuverlässigkeitsbewertung durchzuführen. Die Ansätze können in zwei grundlegenden Kategorien unterteilt werden:

1. Ansätze, die das annotierte Modell extrahieren, um anschließend traditionelle Ansätze auf den Daten durchzuführen und

2. Ansätze, die Analysen direkt auf dem UML-Modell ausführen.

Die UML wird vorrangig zur strukturellen Beschreibung des Systems verwendet. Komponenten werden mit zusätzlichen Informationen, wie z. B. der Ausfallhäufigkeit oder der Propagierungswahrscheinlichkeit, annotiert. Die erste Kategorie der Ansätze exportiert diese Informationen sowie die zugrundeliegende Struktur und verwendet im Anschluss Analysen wie bereits in diesem Kapitel vorgestellt, wie z. B. eine FTA. Die zweite Kategorie generiert interne Datenstrukturen, auf welchen die Analysen durchgeführt werden. Hier kommen z. B. Markov-Ketten oder Datenabhängigkeitsgraphen zum Einsatz.

Den Ansätzen gemein ist, dass der Anwender ein Modell der Systemabhängigkeiten und Fehlerausbreitung erstellen muss. Die Modelle beinhalten keine detaillierte Funktionalität, sondern maximal eine abstrakte Repräsentation, wie z. B. die Modellierung von Datenabhängigkeiten. Aus diesem Grund muss die Effektivität von Fehlerbehebungsmechanismen abgeschätzt werden. Hierbei kommen, aufgrund der Systemkomplexität, häufiger Überapproximationen zum Tragen, was die Genauigkeit der Analyse reduziert. Gerade bei komplexen Systeme, kann zudem die Abschätzung der Abhängigkeiten und somit die Modellbildung fehlerhaft sein, was wiederum zu inkorrekten Analysen führt.

3.4 Fehlerspezifikation

Während Modelle auf niedrigen Abstraktionsebenen, wie z. B. strukturelle Netzlisten, wohldefinierte und ausgiebig erforschte Fehlermodelle aufweisen, fehlen bei abstrakten Systemmodellen solche generellen Fehlermodelle. Viele der im vorherigen Abschnitt präsentierten Injektionsansätze verwenden vordefinierte bzw. bereits implementierte Fehlermuster, die lediglich zur Laufzeit durch ein Kontrollsignal aktiviert werden.

K. Rothbart [101] verwendet eine Fault Injection Control Unit (FIC) zum Aktivieren der Fehler. Die Bestimmung des Injektionszeitpunkts erfolgt abhängig von der Simulationszeit und das zu injizierende Muster beschränkt sich auf, von der Simulation unabhängige, Bitmuster. Ein ähnlicher Ansatz verwendet Perez et al. in [88]. Die Injektion wird abhängig von dem Wert eines Zeitgebers ausgelöst. Der Ansatz unterstützt Zeitgeber mit unterschiedlichen Auflösungen. Für die Injektion werden konstante Werte verwendet.

Der Ansatz [25] verwendet die Anzahl von Bustransaktionen, um die Fehlerinjektion auszulösen und zu beheben. Hierbei handelt es sich um einen ereignisorientierten Ansatz, da Bustransaktionen als interne Simulationsereignisse gesehen werden können. Dies bildet einen ersten Ansatz, der den aktuellen Systemzustand, für die Fehlerinjektion betrachtet. Das zu injizierende Verhalten wird ähnlich der anderen Ansätze aus einer Liste vorbestimmter Injektionsanweisungen ausgewählt.

Die Fehleremulationsplattform PARSIFAL [117] unterstützt die Fehlermodelle: Stuck-at, einfacher Brückenfehler und ein Verzögerungsfehler. Die Aktivierung erfolgt hierbei durch das Setzen eines FlipFlops. Auch dieser Ansatz bestimmt die möglichen Fehlermodelle durch die Wahl des Fehlerinjektors und die Fehlersimulation beschränkt sich auf die Aktivierung der Fehler.

3.5 Defizite des Stands der Wissenschaft und Technik

Die vorgestellten Injektionsansätze haben alle ihre Vor- und Nachteile und deswegen ihren Nutzen in den jeweiligen Anwendungsfällen. Der in dieser Arbeit vorgestellte Ansatz stellt eine ganzheitliche Methode zur Fehlerinjektion dar. Vorrangiges Augenmerk ist die durchgängige Anwendbarkeit des Ansatzes entlang des Entwurfsprozesses. Hierzu werden bestehende Konzepte erweitert, um die abstraktionsebenenübergreifende Anwendbarkeit zu gewährleisten. Ein weiterer adressierter Aspekt ist die Benutzerfreundlichkeit des Ansatzes. Hier werden Schnittstellen geschaffen, um unterschiedliche Stakeholder, wie Entwickler und Tester mit einzubeziehen.

Die zuvor vorgestellten Ansätze beschränken sich meist auf eine Abstraktionsebene, da sie typische Sprachkonstrukte der Abstraktionsebene verwenden. Viele Ansätze sind auf der RTL-Ebene oder der Gatter-Ebene angesiedelt, da sie Simulationsprimitiven wie `sc_lv` oder `sc_bit` zur Injektion verwenden. Nur wenige der Ansätze unterstützen abstrakte, rein funktionale Simulationsmodelle. Tabelle 3.2 gibt eine Übersicht über die bereits vorgestellten Ansätze und ihre Eigenschaften.

Hierbei werden die SFI-Ansätze nach der Art der Fehlerinjektion, der unterstützen Simulationsprimitiven zur Fehlerinjektion sowie der demonstrierten Abstraktionsebenen beschrieben. Des Weiteren wird aufgezeigt, ob Fehler es dem Fehlerinjektor möglich ist einen injizierten Wert aufrecht zu erhalten. Hiermit stark verknüpft, ob es dem Fehlerinjektor möglich ist den Fehler während der Simulation explizit zu beheben. Einige Ansätze adressieren diese Thematik explizit, andere Ansätze bieten das Verhalten inhärent, z. B. die Fehlerdominanz bei Saboteur-Modulen, bei einigen Ansätzen gibt es keine Angaben. Mit der Fehlerbehebung geht einher, ob der Ansatz Möglichkeiten bietet eine Reaktivierung der Simulation zu spezifizieren. Bei Ansätzen die hardwarenahe Simulationsprimitiven verwenden, wie z. B. `sc_signal`, ist das Verhalten inhärent. Alle Ansätze mit abstrakten Modellen, bieten keine explizite Modellierung des Reaktivierungsverhaltens. Die letzten beide Merkmale, die zur Klassifikation der bestehenden Fehlerinjektionsansätze, verwendet werden zeigen auf welche dynamische Kontrolle die Fehlerinjektoren bieten sowie die demonstrierten Fehlermodelle. Tabellenzellen, die Grün hervorgehoben sind weisen Eigenschaften auf, die durch diese Arbeit angestrebt werden. Wie zu erkenne, weist kein Ansatz alle angestrebten Eigenschaften auf.

Autor	**Lisherness [68]**	**Misera [73–75]**	**Fummi [16, 34]**	**Lu [69]**
Injektionstechnik	Mutation	Saboteur	Mutation	Mutation
Fehlerlokation	Intern	Verbindung	Intern	Intern
Simulations-primitiven	Operator, Variable mit Konstante	sc_logic, sc_signal	sc_bit, TLM-2.0-Primitiven	node<T>
Demonstrierte Abstraktions-ebenen	Systemebene, TLM, RTL, Gatter	RTL, Gatter	TLM, RTL, Gatter	Systemebene, RTL
Fehlerdominanz	Ja-Konstant	Inhärent	Ja	Ja-Konstant
Fehlerbehebung	Nein	Inhärent	Ja	Freigabe
Reaktivität der Simulation	Nein	Inhärent	Nein	Nein
Dynamische Fehlerspezifikation	Nein	Aktivierung Deaktivierung	Aktivierung Deaktivierung	-
Demonstrierte Fehlermodelle	-	Stuck-at	Stuck-at, Transition FaultModel	-

Autor	**Shafik [105]**	**Beltrame [8, 9, 15]**	**Tabacaru [112]**	**Chen [28]**
Injektionstechnik	Mutation	Mutation	Mutation	Saboteur
Fehlerlokation	Intern	Intern	Intern	Verbindung
Simulations-primitiven	reg<T>, sc_logic, sc_int, sc_bigint, sc_lv, sc_bv	Member-Variablen	Variablen, TLM-Payload, SystemC-Ports	TLM-Trans-aktoren, Multiplexer
Demonstrierte Abstraktions-ebenen	RTL	TLM, RTL	Systemebene, TLM, RTL, Gatter	TLM, RTL
Fehlerdominanz	-	Nein	Ja-Variabel	Inhärent
Fehlerbehebung	-	-	Freigabe	Inhärent
Reaktivität der Simulation	Nein	-	Nein	Inhärent
Dynamische Fehlerspezifikation	Zufällig	Zeitbedingt	Zeitbedingt	Bus-transaktions-bedingt
Demonstrierte Fehlermodelle	Stuck-at, Bit-Flip, Delay	-	-	-

Tabelle 3.2: Vergleich bestehender Ansätze

Bei der Analyse hat sich gezeigt, dass ein selten betrachtetes Problem, das bei abstrakten, funktionalen Modellen sehr stark ausgeprägt ist, die Spezifikation eines Reaktivierungsverhaltens bei der Fehlerinjektion ist. Keiner der betrachteten Ansätze spezifiziert das Reaktivierungsverhalten explizit. Die Saboteur-Ansätze sowie die Mutationsansätze über SystemC-Simulationsprimitiven stellen inhärente das Verhalten der Aktivierung der Simulation bereit. In Low-Level-Modellen, bei denen eine Fehlerinjektion in SystemC-Simulationsprimitiven erfolgt, erkennt der SystemC-Simulationskern den überschriebenen Wert bzw. den geänderten Wert und führt alle sensitiven Prozesse erneut aus. Bei abstrakten Modellen ist das Reaktivierungsverhalten nicht inhärent im Modell und der Anwender muss das Verhalten separat spezifizieren. Keiner der betrachteten Ansätze ging auf diese Problematik ein.

Eines der größten Defizite ist die Spezifikation des zu injizierenden Verhaltens. Der Fehler bzw. die Fehlerinjektion werden durch vier Informationen bestimmt:

- den Ort des Auftretens,
- den Zeitpunkt des Auftretens,
- die Art und Weise wie sich der Fehler manifestiert und
- ob und wann sich ein Fehler von selbst behebt.

Die betrachteten Ansätze verwenden zu Festlegung des Fehlerinjektionszeitpunkts Ereignisse in der Simulation, einfache Zähler [28] oder die Simulationszeit [112]. Bei der Spezifikation des Fehlerverhaltens greifen die meisten Ansätze auf vordefiniertes Fehlerverhalten zurück, das entweder direkt im Quellcode steht oder über Lookup-Tabellen ausgewählt wird. Keiner der Ansätze bietet eine umfassende dynamische Fehlerspezifikationsmethodik, die sich auf unterschiedliche Abstraktionsebenen anwenden lässt. Vor allem bei sehr abstrakten Modellen muss die Fehlerspezifikation die Lücke zwischen dem abstrakten Simulationsmodell und den meist detaillierten hardwarenahen Fehlermodellen überbrücken. Hierbei ist es notwendig, dass die Fehlerspezifikation den Systemzustand zur Bestimmung des Injektionszeitpunkts oder der Fehlerwirkung berücksichtigt.

Bei der Zielstellung einer ganzheitlichen Methode zur Fehlerinjektion, die durchgängige entlang des Entwurfsprozesses angewendet werden kann, darf der Aspekt der Integration nicht vernachlässigt werden. Gerade modellgetriebenen Entwurfsprozesse unterstützen den Anwender. Keiner der betrachteten Ansätze ging auf Möglichkeiten der Integration in eine modellgetriebene Entwicklungsumgebung ein. Da es sich um eine softwaregestützte Simulation handelt, können gängige modellgetriebenen Ansätze zur Softwareentwicklung angewandt werden. Durch den Bezug zur SFI entstehen neue Möglichkeiten, wie z. B. die Unterstützung bei der Fehlerspezifikation oder die automatisierte Integration von Fehlerinjektoren, die durch aktuelle Ansätze noch nicht unterstützt werden.

Kapitel 4

Zuverlässigkeitsbewertung mittels Fehlereffektsimulation

Das Ziel dieser Arbeit ist, eine Methode bzw. Werkzeugumgebung bereitzustellen, die eine simulationsgestützte Analyse der funktionalen Sicherheit entlang des Entwurfsprozesses ermöglicht. Abbildung 4.1 verdeutlicht das angedachte Vorgehen am klassischen V-Modell. Um dies zu erreichen, werden Methoden vorgestellt, welche die Defizite des Stands der Technik und Wissenschaft adressieren.

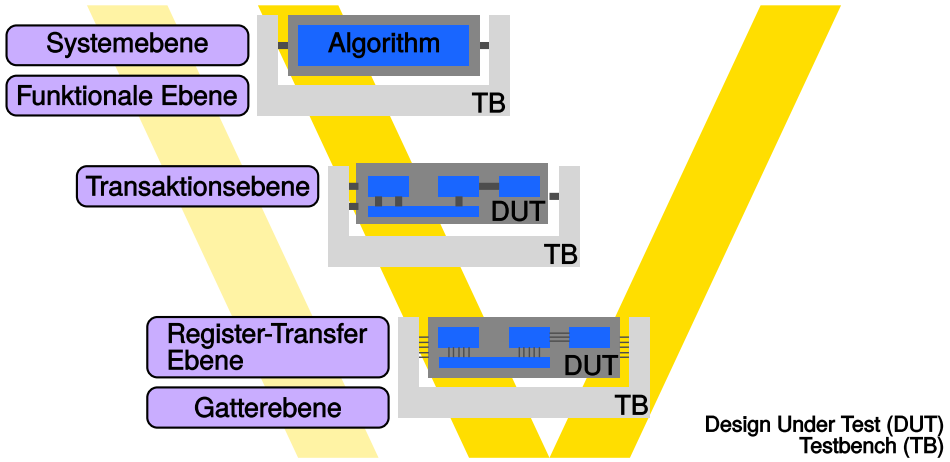


Abbildung 4.1: Durchgängige Fehlereffektbewertung

Dem Anwender wird eine Werkzeugumgebung zur Unterstützung der Zuverlässigkeitsbewertung komplexer, vernetzter Systeme zur Verfügung gestellt. Die Analysen beruhen auf Simulationsmodellen, die es ermöglichen die Effekte von Fehlerursachen zu analysieren. Um die Analyse ohne detailliertes Wissen über die Simulationsmodelle durchführen zu können, wird unterstützend ein grafischer Spezifikationsansatz, basierend auf der UML, bereitgestellt. Abbildung 4.2 zeigt die Eingliederung des Ansatzes in einen modellgetriebenen Entwurf. Hierbei wird ein paralleler Analyseablauf angestrebt, der auf Modellen basiert, die aus den Entwurfsmodellen hervorgehen. Der Anwender verwendet, zur Modellierung des finalen Systems, die

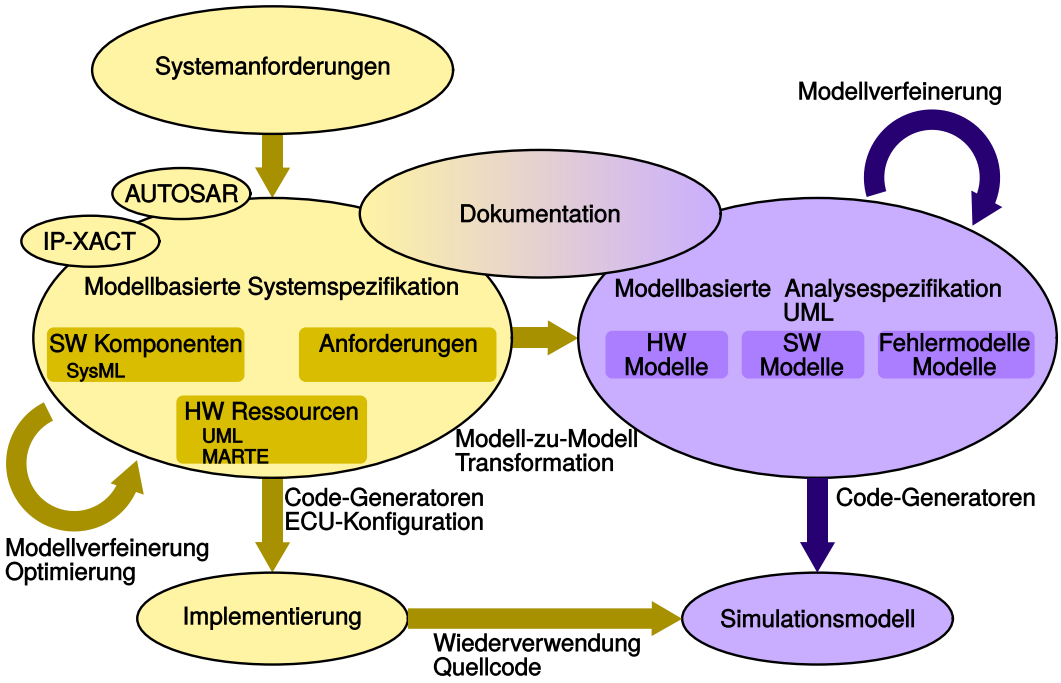


Abbildung 4.2: Eingliederung der Arbeit in einen modellgetriebenen Entwurfsablauf

gewohnte Modellierungsumgebung sowie domänenspezifische Modellierungsansätze wie AUTOSAR oder IP-XACT. Die Optimierungen und Verfeinerungen sowie die Ableitung der Implementierung bzw. Parametrisierungen bleiben unverändert. Aus dem zentralen Modell wird das Analysemodell, durch Modell-zu-Modell-Transformationen, erzeugt. Das Analysemodell entspricht größtenteils dem Systemmodell, beinhaltet aber Anpassungen zur Durchführung der Fehlereffektsimulation. Hierzu zählen vorrangig die Modellierung der Testbench wie z. B. Testpatterngeneratoren aber auch die in dieser Arbeit vorgestellte Fehlerspezifikation. Sind Systemteile für die Analyse irrelevant, können diese im Analysemodell durch Mock-Objekte abstrahiert werden. Das Spezifikationsmodell der Fehlereffektanalyse dient der automatischen Erzeugung der Fehlereffektsimulationen und unterstützt den Anwender allgemein bei der Durchführung der Sicherheitsanalysen.

Der UML-Modellierungsansatz ermöglicht die Spezifikation des zu analysierenden Systems, der zu untersuchenden Fehlerursachen sowie der kritischen Systembereiche. Die kritischen Systembereiche dienen zur Detektion der Fehlerauswirkungen. Der grafische Spezifikationsansatz gliedert sich in aktuelle Sicherheitsanalyseprozesse ein und unterstützt die Sicherheitsexperten bei der Analyse des Systems, z. B. bei der Erstellung einer FMEA. Eine detaillierte Kenntnis der zugrunde liegenden Simulationsmodelle bzw. der Simulationssprache wird von den beteiligten Sicherheitsexperten nicht benötigt. Die grafische Spezifikation dokumentiert, welches System mit welchem Fehler getestet wurde und wie die Systemreaktion ausgefallen ist.

Zur Spezifikation der zu untersuchenden Fehlerursachen wird eine UML-Erweiterung vorgestellt, um unabhängig von der untersuchten Abstraktionsebene, Fehlerursachen zu spezifizieren. Ein wohldefiniertes Spezifikationsformat für das zu injizie-

rende Fehlerverhalten, erleichtert die abstraktionsebenenübergreifende Anwendbarkeit. Besonders bei sehr abstrakten Modellen kommt der Fehlerspezifikation besondere Bedeutung zu, da sie die Lücke zwischen den häufig sehr hardwarenahen, erprobten Fehlermodellen und der abstrakten Systemsimulation überbrücken muss. Das in dieser Arbeit erweiterte Spezifikationsformat, wird vorrangig zur Spezifikation von Echtzeitsystemen verwendet. Das Spezifikationsformat umfasst die Interaktion mit der Systemsimulation und die kompakte Spezifikation des Fehlerverhaltens.

Ein wichtiges Konzept des Ansatzes ist, die Entkopplung der Erstellung und der Wartung der Simulationsmodelle von der Sicherheitsanalyse. Hierzu wird die Analyse über Spezifikationsmodelle konfiguriert. Der Sicherheitsexperte muss nicht die SystemC-basierten Simulationsmodelle kennen, sondern lediglich die grafischen Spezifikationsmodelle. Die Simulationsmodelle werden durch Entwickler, Drittanbieter oder aus dem Bereich der Systemqualifikation bereitgestellt. Um die Sicherheitsexperten von den eigentlichen Simulationsmodellen zu entkoppeln, wird eine auf IP-XACT basierte Konfigurationsdatei verwendet, die alle Informationen zur Durchführung der Fehlereffektsimulation enthält. Die Konfiguration wird zur Laufzeit eingelesen und konfiguriert die aktuell durchzuführende Simulationsinstanz. Die Konfiguration spezifiziert die zu instanziierenden Simulationseinheiten. Dies umfasst die Parametrisierung und Komposition der Simulationseinheiten. Zusätzlich wird die Platzierung der Fehlerinjektor, das zu injizierende Fehlerverhalten und die Platzierung von Beobachtungspunkten spezifiziert. Die Konfigurationsdatei wird vollständig aus der grafischen Spezifikation generiert und erfolgt transparent für den Anwender.

Um den Gesamtaufwand zur Erstellung der Simulationsmodelle zu reduzieren, werden die UML-basierten Spezifikationsmodelle wiederverwendet. Die grafische Benutzerschnittstelle beinhaltet verschiedene Codegenerierungsschritte, die den manuellen Aufwand für den Anwender reduzieren. Die grafische Umgebung spezifiziert die Simulationseinheiten, ihre Verknüpfungen sowie deren Attribute. Mittels Codegenerierung wird der Quellcode der Simulationseinheiten und die Konfigurationsdateien generiert.

Die so erzeugten Simulationseinheiten bilden den Kern der Analyse, den sogenannten virtuellen Prototypen. Um genauer zu sein, wird eine Simulationseinheitenbibliothek erzeugt und mittels eines dynamischen Konfigurationsansatzes werden Simulationsinstanzen generiert. Diese Instanzen sind die Grundlage für die durchgeführte Fehlereffektbestimmung. Die Simulationsmethodik berücksichtigt gängige Konzepte wie die Modularität der Simulationsmodelle oder standardisierte Schnittstellen von Simulationsmodellen. Herausforderungen, die speziell in frühen Entwicklungsphasen durch die hierbei existierende Variabilität des Systems hervorgerufen werden bzw. auf der generellen Anwendbarkeit der Methodik beruhen, erfordern eine Erweiterung bestehender Simulationsansätze. Aus diesen Gründen wird ein auf SystemC basiertes Simulationskonzept bereitgestellt, das den Stand der Technik um ein mehrschichtiges Architekturkonzept und ein Konzept zur Abstraktion von systemspezifischen Eigenschaften erweitert. Diese Maßnahmen erhöhen die Variabilität der Simulationsmodelle und somit

signifikant die Basis für die Fehlereffektsimulation.

Um eine höhere Flexibilität bei der Fehlerinjektion zu erzielen, wird die Trennung der Spezifikation des Fehlerverhaltens von der Realisierung der Fehlerinjektion vorgenommen. Das heißt, die Fehlerspezifikation wird zur Laufzeit durch die Fehlerinjektionsinfrastruktur interpretiert und in der Simulation enthaltene Fehlerinjektoren angesteuert. Um dies zu erreichen, wird eine maschinell verarbeitbare Fehlerspezifikation, das sogenannte Gefahrenverhaltensmodell (engl. Behavioral Threat Model (BTM)), bereitgestellt. Des Weiteren wird ein Konzept zur Injektion von Fehlern sowie zur Beobachtung und Klassifikation der Fehlereffekte eingeführt. Eine wichtige Anforderung der hier vorgestellten Lösung ist, dass es sich um einen Simulationskern unabhängigen Ansatz zur Fehlerinjektion handelt. Aus diesem Grund wird ein minimal invasiver Ansatz zu Fehlerinjektion, der auf der Mutation, bzw. der Ersetzung von Datentypen innerhalb der Simulationsmodelle beruht, bereitgestellt. Die Injektionserweiterung muss folgende Funktionalität bereitstellen:

- das Überschreiben des Simulationszustands durch ein globales Steuermodul,
- das Zurücksetzen auf den korrekten Simulationszustand,
- die Auswertung des Systemzustands durch die externe Steuerung und
- die erneute Aktivierung der Simulation nach der Injektion des Fehlers.

4.1 Bestandteile der Fehlereffektsimulation

Abbildung 4.3 zeigt die Hauptbestandteile der Fehlereffektsimulation. In der oberen Hälfte ist die grafische Spezifikation, die als Benutzerschnittstelle für die Sicherheitsanalyse und zur Unterstützung der Anwender dient, dargestellt. Sie ist mittels Eclipse Papyrus realisiert und wird detailliert in Kapitel 6 dargestellt. Codegenerierungsschritte generieren die Simulationsmodelle aus den spezifizierten UML-Klassendiagrammen. Die Erweiterungen zur Fehlereffektsimulation werden in Kapitel 5 vorgestellt. Die aus den Spezifikationsmodellen generierte Simulationseinheitenbibliothek (SEB) sowie die bereitgestellte Simulationsinfrastruktur, z. B. zum Einlesen der dynamischen Konfiguration, sind auf der linken Seite dargestellt. Die vorgeschlagene Struktur für Simulationsmodelle und die Erweiterungen zur dynamischen Konfiguration werden in Abschnitt 5.1 vorgestellt. Die Simulationsmodellerweiterungen zur Fehlerinjektion und Fehlereffektbeobachtung sind in Abschnitt 5.2 – 5.4 dargelegt. Neben den generierten Simulationsmodellen wird die bereitgestellte Infrastruktur mit in die SystemC/C++-Bibliothek kompiliert.

Ein Kompositionsstrukturdiagramm spezifiziert die Struktur der Systemsimulation sowie die Parametrisierung. Das Fehlverhalten wird über Zustandsdiagramme spezifiziert und in einem Kompositionsstrukturdiagramm annotiert. Zusammen dienen sie als Grundlage zur Generierung der dynamischen Simulationskonfiguration.

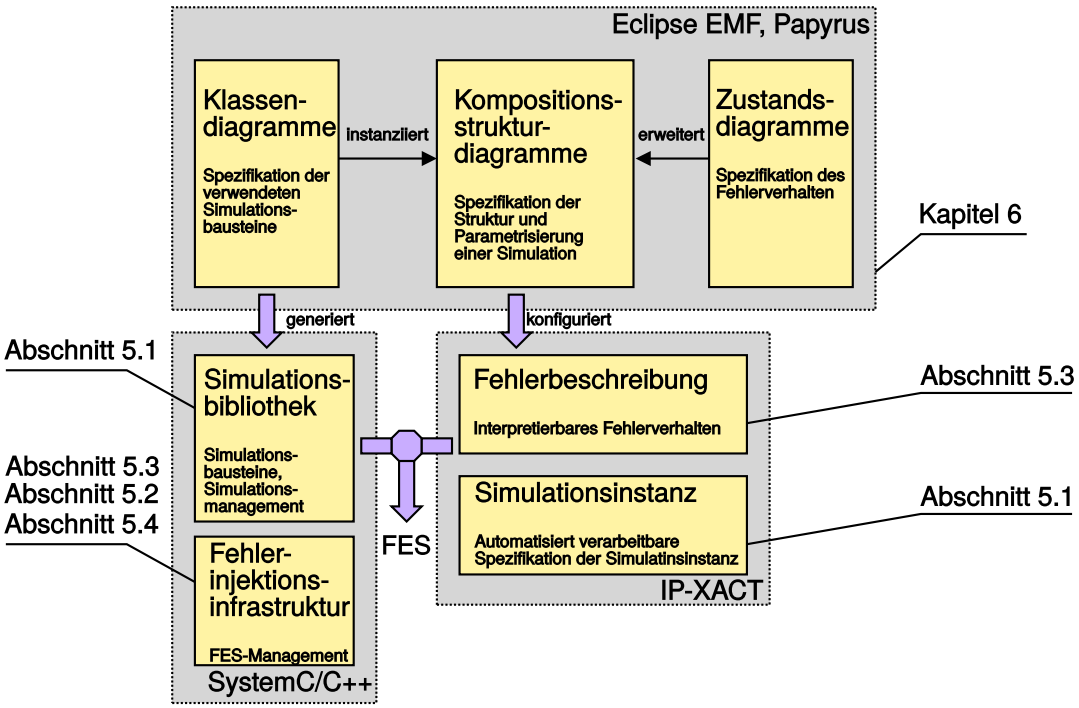


Abbildung 4.3: Bestandteile der Fehlereffektsimulation (FES)

Die kompilierte Systembibliothek sowie die dynamische Konfiguration ergeben eine Fehlereffektsimulationsinstanz. Durch Modifikation der Konfigurationsdatei ist es möglich unterschiedliche Simulationen auszuführen, ohne die SEB erneut erzeugen zu müssen. Insbesondere ermöglicht der Ansatz unterschiedliches Fehlverhalten, ohne erneute Kompilation, zu simulieren.

Kapitel 5

Fehlereffektsimulation

Im folgenden Abschnitt 5.1 wird das verwendete Simulationsrahmenwerk vorgestellt. Es bildet die Grundlage der Fehlereffektsimulation und ermöglicht die effiziente Durchführung unterschiedlicher Systemsimulationen. Die darauf aufbauenden Erweiterungen zur Fehlereffektsimulation werden in zwei Hauptbestandteile untergliedert. Zum einen in Erweiterungen der Simulationsmodelle für die Fehlerinjektion, die in Abschnitt 5.2 vorgestellt werden. Dies beinhaltet Fehlerinjektoren (I_1, I_2), die das Ändern des Systemzustands ermöglichen sowie Wertesonden (P_1, P_2), zum Erfassen des aktuellen Simulationszustands. Zum anderen wird eine Infrastruktur zur Steuerung der Fehlerinjektion und der dynamischen Interpretation des spezifizierten Fehlverhaltens benötigt. Diese Steuerung erfasst über Wertesonden den aktuellen Simulationszustand und steuert alle Fehlerinjektoren in der Systemsimulation. Neben Erweiterungen der Simulation wird in dieser Arbeit ein wohldefiniertes Spezifikationsformat, zur Beschreibung unterschiedlicher Fehlerfälle, verwendet. Ziel ist es unterschiedliche Fehlerfälle zu spezifizieren und automatisiert in die Simulation einzubringen. Die Steuerung wertet den aktuellen Simulationszustand aus und steuert die Fehlerinjektoren, anhand des zuvor spezifizierten Fehlerverhaltens, an. Abschnitt 5.3 stellt den entwickelten Spezifikationsansatz und die zentrale Steuerung der Fehlerinjektoren vor. Abbildung 5.1 verdeutlicht die Struktur. Aufbauen auf der Fehlerstimulation stellt Abschnitt 5.4 Ansätze zur Fehlereffektbeobachtung vor. Das Ziel ist es, automatisiert die Auswirkungen der Fehlerinjektion auf das System zu detektieren und zu klassifizieren. Teile der Erweiterungen zur Fehlerinjektion wurde vom Autor bereits in den Veröffentlichungen [94, 99, 100] publiziert.

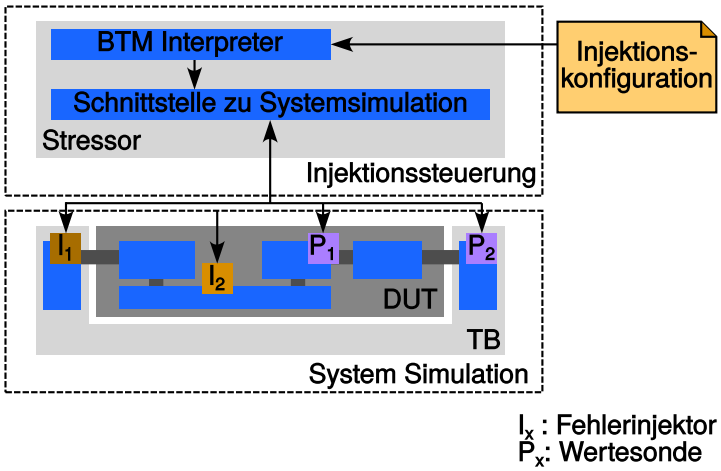


Abbildung 5.1: Struktur des Fehlerinjektionsansatzes

5.1 Systemsimulation

Der folgende Abschnitt stellt das Simulationsrahmenwerk vor. Besonderer Fokus wird auf die Wiederverwendbarkeit vorhandener Simulationskomponenten gelegt. Ähnlich der Grundidee komponentenbasierter Software [91] ist die Unterteilung der Simulation in wiederverwendbare Simulationseinheiten vorgesehen. Aus der entstehenden Simulationseinheitenbibliothek werden unterschiedliche Simulationsinstanzen erzeugt. Abbildung 5.2 zeigt den daraus resultierenden Simulationsablauf. Der

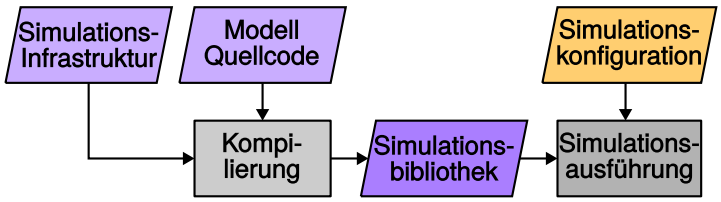


Abbildung 5.2: Schritte und Gegenstände des Simulationsablaufs

zentrale Modellquellcode wird mit der Simulationsinfrastruktur zu einer Simulationseinheitenbibliothek kompiliert. Die im Rahmen dieser Arbeit entwickelte Simulationsinfrastruktur ist modellunabhängig und somit wiederverwendbar. Über eine Konfigurationsdatei wird daraufhin die eigentliche Simulation aus den Bibliothekseinheiten zusammengestellt. Hierdurch lassen sich unterschiedliche Systeme sehr einfach simulieren. Teilaspekte wurden vom Autor bereits in [97, 100] publiziert.

Die folgenden Abschnitte stellen die benötigten Konzepte im Detail vor. Die Makroarchitektur der Systemsimulation ist in Abschnitt 5.1.1 dargelegt. Zur Gewährleistung der Wiederverwendbarkeit der Simulationskomponenten in unterschiedlichen Simulationsinstanzen definiert die Makroarchitektur eine mehrschichtige Architektur mit standardisierten Schnittstellen. Zusätzlich wird ein Konzept zur Abstraktion physikalischer Eigenschaften vorgestellt, um die Flexibilität des Bibliotheksansatzes zu gewährleisten. Die modulare, parametrisierbare Mikroarchitektur der Simulations-

komponenten ist in Abschnitt 5.1.2 vorgestellt. Die benötigte Infrastruktur und das Vorgehen zur effizienten Erzeugung unterschiedlicher Simulationsinstanzen werden in Abschnitt 5.1.3 dargelegt.

5.1.1 Transaktionsbasierte Makroarchitektur

Die in diesem Kapitel behandelten Beispiele dienen vorrangig der Einbeziehung unterschiedlicher Kommunikationssysteme, da dies häufig eine Limitation der Systemvariabilität darstellt. Die vorgestellten Konzepte sind allgemeingültig und stellen die Austauschbarkeit von Simulationsmodellen, trotz der Verwendung systemspezifischer Informationen, sicher. Anhand von Fallbeispielen wird die Anwendung der Modellierungsmethodik demonstriert und weitere Herausforderungen aufgezeigt. Die Simulationsmodelle verwenden eine transaktionsbasierte Makroarchitektur, um die Funktionalität der Systemsimulation zu unterteilen. Dieser Ansatz entspricht gängigen Ansätzen zur High-Level-Simulation. Hierbei wird die Kommunikation zwischen Komponenten von der Implementierung der einzelnen Komponenten getrennt. Die vorgestellte Makroarchitektur verwendet TLM-2.0-Schnittstellen, um den Datenaustausch zwischen Komponenten zu realisieren. TLM-2.0 steigert die Interoperabilität von Simulationskomponenten und entspricht dem heutigen Stand der Technik. Hierdurch erleichtert TLM-2.0 die Integration von Modellen von Drittanbietern. Abbildung 5.3 zeigt beispielhaft die Makroarchitektur eines Systems. Findet eine

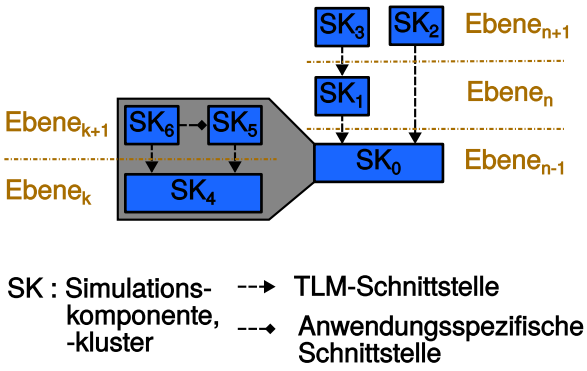


Abbildung 5.3: Makroarchitektur mit unterschiedlichen Ebenen

Kommunikation über Grenzen unterschiedlicher Ebenen hinweg statt, müssen die Module die Daten durch ein TLM-2.0-Transaktionsobjekt übertragen. Für Kommunikation innerhalb einer Ebene können die Module applikationsspezifische Schnittstellen, wie anwendungsspezifische Funktionsaufrufe verwenden. Durch die Kapselung ist es möglich, Simulationskomponenten unterschiedlicher Abstraktionsebenen oder verschiedene Implementierungen über die TLM-2.0-Schnittstellen zu verbinden. Zur Integration von Komponenten mit applikationsspezifischen Schnittstellen werden Adaptermodule verwendet, die einzelnen Transaktionen auf Signalansteuerungen abbilden und umgekehrt. Abbildung 5.4 zeigt die Kapselung eines Controller Area Network (CAN)-Controllers mit einem TLM-Adapter. Durch dieses Vorgehen ist es möglich, den CAN-Controller mit unterschiedlichen Anwendungen zu koppeln, die

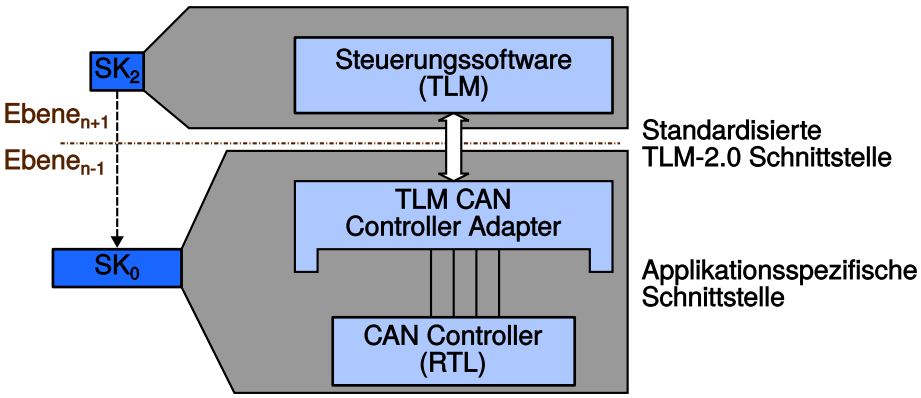


Abbildung 5.4: Generalisierung applikationsspezifischer Schnittstellen

über die TLM-Schnittstelle Daten übertragen. Der Adapter übersetzt die generellen Informationen des Transaktionsobjekts, wie Adresse, Nutzdaten oder ob es sich um einen Lese- oder Schreibzugriff handelt in die anwendungsspezifische Ansteuerung des CAN-Controllers. Im Zusammenhang mit dieser Funktionalität wäre auch der Begriff *Transaktor* passend. Aufgrund der eingeschränkten Verwendung im TLM-2.0 Language Reference Manual [4], wird in dieser Arbeit auch der Begriff des Adapters verwendet. Bei einem Schreibzugriff übersetzt der Adapter die Nutzdaten in die erwartete Datenstruktur und schreibt die Datenstruktur abhängig von der Adresse des Transaktionsobjekts in den vorgesehenen Puffer. Je nach Wahl des Sendepuffers versendet der CAN-Controller die Daten über den statischen oder dynamischen Slot.

Die Systemsimulation verwendet im Rahmen der vorgeschlagenen Methodik eine mehrschichtige Makroarchitektur. Am Beispiel einer dreischichtigen Makroarchitektur, die sich z. B. in die Anwendungs-, Protokoll- und Kommunikationsschicht unterteilt, wird das Konzept motiviert. Abbildung 5.5 zeigt die Untergliederung und ordnet die Ebenen in das OSI-Referenzmodell ein. Komponenten der unterschiedlichen Schichten sind über die TLM-2.0-Schnittstelle gekoppelt. Die standardisierte Schnittstelle gewährleistet die Interoperabilität von Komponenten unterschiedlicher Ebenen. Die Anwendungsschicht besteht aus Datenquellen und Datensenken, wobei einzelnen Komponenten gleichzeitig als Datensenke und -quelle agieren können. Alle Komponenten senden und empfangen Daten über eine TLM-2.0-Schnittstelle von der darunter liegenden Ebene. Die Protokollebene stellt unterschiedliche Übertragungsprotokolle zur Verfügung. Komponenten der Protokollebene stellen TLM-2.0-Schnittstellen zur Verfügung und benötigen gleichzeitig TLM-2.0-Schnittstellen von den Komponenten der darunter liegenden Ebene. Sie verarbeiten die eingehenden Daten und leiten diese an die nächste Ebene weiter. Repräsentatives Beispiel ist eine Transmission Control Protocol (TCP)-Implementierung, die den Daten einen TCP-Header voranstellt und sie danach weiterleitet. Nichtsdestotrotz können Komponenten der Ebene unabhängig von der darüber liegenden Ebene agieren, z. B. im Falle von Sendewiederholungen. Die unterste Ebene modelliert den technologiespezifischen Datenaustausch. Hier sind z. B. Modelle des Media Oriented Systems Transport (MOST)-Busses oder des CAN-Busses zu nennen. Diese Komponenten

stellen eine TLM-2.0-Schnittstelle zur Verfügung. Der technologiespezifische Datenaustausch zwischen den Simulationseinheiten erfolgt über modellspezifische Schnittstellen innerhalb der untersten Ebene. Durch die dreischichtige Architektur ist es möglich, eine Anwendung mit unterschiedlichen Kommunikationstechniken bzw. unterschiedlichen Protokollen zu analysieren, ohne die Simulationsmodelle der Anwendung ändern zu müssen. In Abbildung 5.5 sind anhand des Wassertankbeispiels

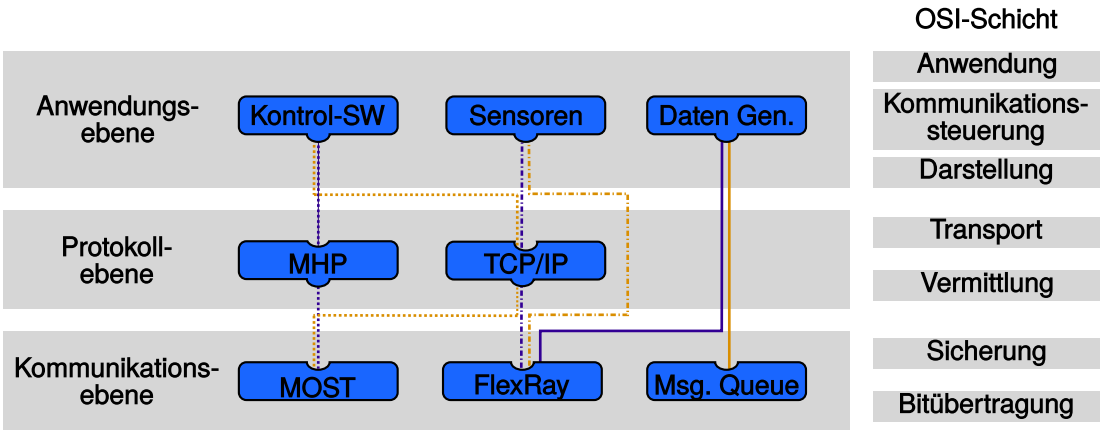


Abbildung 5.5: Beispiel einer dreischichtigen Makroarchitektur

exemplarische Abbildungen der Kommunikation auf unterschiedliche Bussysteme dargestellt. In diesem Szenario erfasst der Kontrollalgorithmus mit Sensoren Daten über den Zustand des Wassertanks. Zusätzlich ist ein generischer Datengenerator, zur Erzeugung einer Buslast dargestellt. Die Module sind auf der Anwendungsebene angesiedelt. Die dreischichtige Makroarchitektur und die jeweiligen Adapter ermöglichen, unterschiedliche Protokolle und Kommunikationsmedien zum Datenaustausch zu verwenden. So ist es möglich die unterschiedlichen Anwendungen auf Kommunikationsmedien, wie z. B. den MOST-Bus, den CAN-Bus oder einen einfachen TLM-Nachrichten-Puffer, abzubilden. Bei dem Kontrollalgorithmus ist beispielsweise dargestellt, dass es bei Verwendung des MOST-Busses möglich ist, unterschiedliche Protokolle zur Transportsteuerung zu verwenden. In der Abbildung stehen exemplarisch das MOST High Protocol (MHP) und das TCP/IP zur Verfügung. Auch das Umgehen des Kommunikationsprotokolls ist möglich, wie z. B. bei den Sensoren des Wassertanks dargestellt. Durch diese Kombinationsmöglichkeiten ergeben sich unterschiedliche Systemausprägungen. Abbildung 5.5 stellt lediglich die unterschiedlichen Realisierungsmöglichkeiten dar, in der finalen Systemausprägung ist es essenziell, dass alle Kommunikationspartner das gleiche Kommunikationsmedium und Protokoll verwenden. Des Weiteren ist es möglich, dass die Kommunikation direkt über applikationsspezifische Schnittstellen innerhalb einer Schicht erfolgt. Hierbei erfolgt der Datenaustausch zwischen Kontrollalgorithmus und Sensoren über Funktionsaufrufe oder applikationsspezifische Signale direkt in der Anwendungsebene. In diesem Fall ist es nicht notwendig, TLM-Adapter bereitzustellen.

Ein Problem, das sich an diesem Beispiel gut demonstrieren lässt, sind die heterogenen Adressierungsansätze bei der Abbildung einer Anwendung auf unterschied-

liche Kommunikationstechnologien. Das Problem ist nicht auf die Adressierung beschränkt, sondern tritt auf, wenn die Interoperabilität zwischen unterschiedlichen Technologien aber auch unterschiedlichen Abstraktionsebenen gewährleistet werden muss. Verwendet eine Anwendung die physikalische Adressierung, z. B. TCP/IP-Adressen, ist es nicht möglich, dass diese direkt mit anderen Protokollen oder Kommunikationstechnologien kombiniert werden kann. Um die Flexibilität zu erreichen, verwendet diese Arbeit das Konzept der *Logischen Anwendungsadresse (LAA)*. Eine LAA identifiziert eine Kommunikationsbeziehung, d. h., wenn zwei Komponenten an die gleiche Anwendung senden, werden zwei unterschiedliche LAAs verwendet. Benötigt eine Komponente die physikalische Adresse, z. B. bei der Erstellung der TCP-Header, stellt die Komponente eine Übersetzungstabelle bereit um die LAA in eine physikalische Adresse zu übersetzen. Die Transaktionen an den TLM-2.0-Schnittstellen dürfen lediglich die LAA verwenden. Die physikalischen Adressen werden nur innerhalb der Kommunikationsschicht oder eingebettet in den Nutzdaten verwendet. Verwenden zwei Kommunikationsassoziationen den gleichen physikalischen Kanal, werden die unterschiedlichen LAAs auf die gleiche physikalische Adresse abgebildet. Abbildung 5.6 zeigt ein Beispiel des Adressierungsansatzes. Hierbei senden zwei

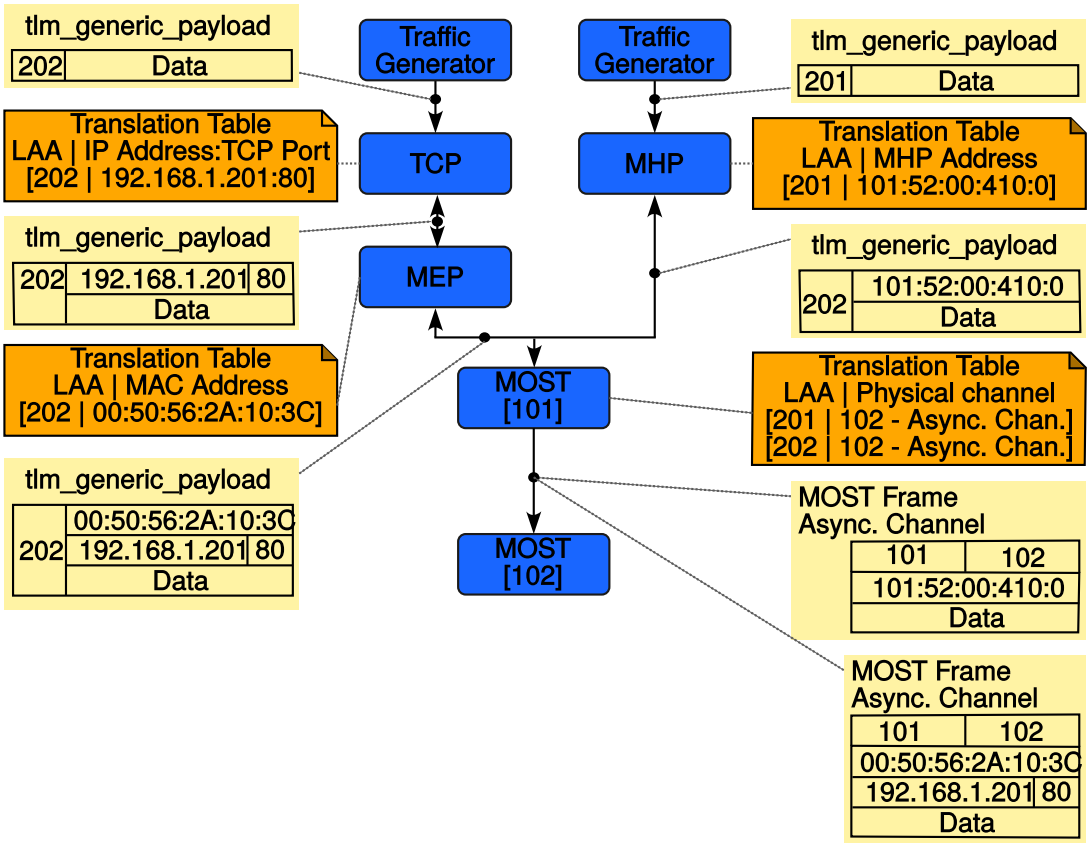


Abbildung 5.6: Beispiel der logischen und physikalischen Adressierung

Datengeneratoren Daten über einen MOST-Bus. Der MOST-Bus verwendet zwei unterschiedliche Protokolle zur Übermittlung der Daten. Jede Simulationseinheit der

Kommunikationsprotokolle besitzt eine Übersetzungstabelle um LAAs in physikalische Adressen zu übersetzen. Die TCP/IP-Protokollkomponente übersetzt die LAA in eine Internet Protocol (IP)-Adresse und TCP-Port. Die MHP-Simulationskomponenten hingegen übersetzt die LAA in ein MOST-eigenes Adressierungsformat.

5.1.2 Modulare, parametrisierbare Mikro-Architektur

Die Makroarchitektur mit den standardisierten TLM-Schnittstellen, in Kombination mit dem logischen Adressierungsansatz bietet einen ersten Freiheitsgrad zur Modularisierung des virtuellen Prototyps. Um die größtmögliche Wiederverwendbarkeit zu erreichen, werden die Komponenten in weitere Basis-Simulationseinheit (BSE) unterteilt. Mit der komponentenbasierten Mikroarchitektur entsteht eine hierarchische Struktur, bei der sich die übergeordnete Komponentenfunktionalität aus den atomaren Einheiten zusammensetzt. Es ist möglich, die Komponentenfunktionalität durch die alternative Wahl der BSEs zu verändern. Hierbei sind zwei Herangehensweisen zu unterscheiden. Die erste Variabilität entsteht durch Hinzufügen zusätzlicher Einheiten. Hierbei wird das Verhalten von Komponenten z. B. durch Hinzufügen zusätzlicher Ressourcen geändert. Ein weiterer Ansatz ist das Bereitstellen von Implementierungsalternativen, d. h. Einheiten, die Funktionen unterschiedlich implementieren. Hierbei ist darauf zu achten, dass die Einheiten eine gemeinsame Schnittstelle anbieten. Im Gegensatz zur Makroarchitektur sind die Schnittstellen der komponentenbasierten Mikroarchitektur nicht durch das Simulationsrahmenwerk vorgegeben. Ein weiterer Aspekt ist die Beschreibung der gleichen Funktionalität auf unterschiedlichen Abstraktionsebenen. Stand bei der vorherigen Betrachtung die Änderung des Verhaltens im Vordergrund, steht hier die Beschleunigung der Simulation im Fokus. Trägt das zeitliche und funktionale Verhalten einer BSE nicht wesentlich zum Analyseergebnis bei, wird diese BSE durch eine abstraktere BSE ersetzt. Mit einer höheren Abstraktion steigt meist auch die Simulationsperformanz. Ändert sich das Analyseziel, ist es jederzeit möglich zurück auf das detailliertere Simulationsmodell zu wechseln.

Am Anwendungsbeispiel des Simulationsmodells für Bussysteme ist z. B. die funktionale Beschreibung einer MOST-Kommunikationskomponente in 17 atomare Simulationseinheiten untergliedert. Es beinhaltet eine Einheit zum Einlesen des ausgetauschten MOST-Rahmens, BSEs zur Handhabung der unterschiedlichen Kanaltypen oder BSEs, die den internen Puffer modellieren. Das Bussystem MOST bietet mehrere logische Kanäle an. Ein Zeitmultiplexverfahren (engl. Time Division Multiple Access (TDMA)) überträgt die Daten über einen einzigen physikalischen Datenrahmen. Abbildung 5.6 zeigt ein Beispiel, das den asynchronen Kanal zu Datenübertragung verwendet. Daneben stellt MOST zusätzlich einen Kontrollkanal sowie mehrere synchrone Kanäle bereit. Verwendet die Simulation einen dieser Kanäle nicht, ist es möglich, die Verarbeitung der Daten aus dem Modell zu nehmen, um die Simulationsgeschwindigkeit zu steigern. Für jeden Kanaltyp existieren Module, welche die Bearbeitung der Daten übernehmen. Je nach Anwendungsfall ist es möglich,

durch einfaches Hinzufügen der Module die durch die Simulation erfassten MOST-Kanäle zu ändern.

Neben dem Hinzufügen oder Weglassen von Simulationseinheiten bieten einige Simulationseinheiten des MOST-Modells unterschiedliche Realisierungsalternativen. Es existieren z. B. unterschiedliche Pufferimplementierungen. Sie unterscheiden sich in der Allokierung von Speicher. Es existiert eine Implementierung, die eine bytegenaue Allokierung zulässt. Diese Simulationsmodelle reservieren, für empfangene und zu sendende Nachrichten, den Pufferspeicher bytegenau. Eine weitere Implementierung reserviert Speicher immer in der maximalen Nachrichtengröße. Durch Austausch der Einheiten ändert sich das funktionale Verhalten des Modells.

Neben den strukturellen Änderungen der Mikroarchitektur bieten die Simulationseinheiten Parameter zur Konfiguration einzelner Einheiten. Die Parameter dienen zum einen der Personalisierung der Einheiten, z. B. der Empfangsadresse, die sich von Instanz zu Instanz ändert, zum anderen aber auch der Variabilität der BSE. Hierunter fallen Parameter, die z. B. das Zeitverhalten der Einheit ändern oder das funktionale Verhalten. Das Simulationsmodell des MOST-Kommunikationscontrollers stellt Parameter zur Konfiguration des bereitgestellten Speicherplatzes bereit. Dieser Ansatz ermöglicht, die größtmögliche Variabilität zu erzielen.

5.1.3 Spezifikation der Systemsimulation

Die vorgestellten Mechanismen erhöhen die Flexibilität der Systemsimulation und erleichtern die Analyse unterschiedlicher Systeme. Auf der anderen Seite erschweren sie die Realisierung der Simulation, da der Anwender eine Vielzahl von Modulen verknüpfen und parametrisieren muss. Bei der manuellen Erstellung einer Systemsimulation wird dies häufig in einem Top-Modul umgesetzt. Um den Aufwand der Simulation zu reduzieren, wird zur Laufzeit eine Konfigurationsdatei eingelesen und die BSEs dynamisch anhand der Konfigurationsdatei parametrisiert und verknüpft. Mit diesem Ansatz ist es möglich unterschiedliche Systemalternativen zu simulieren, ohne jedes Mal ein neues Top-Modul zu implementieren und die Simulation neu zu kompilieren. Der Anwender muss lediglich die Konfigurationsdatei neu erstellen. Der Ansatz ist in Abbildung 5.7 dargestellt.

5.1.3.1 Simulationserweiterung

Für die dynamische Erzeugung einer Simulationsinstanz wird dem Simulationsrahmenwerk eine zusätzliche Konfigurationsverwaltung bereitgestellt. Aufgabe der Konfigurationsverwaltung ist das Einlesen der Konfigurationsdatei und das Erstellen der spezifizierten Simulationsinstanz. Die SEB fasst alle bereitgestellten Simulationseinheiten zusammen. Jede Einheit wird mittels einer Einheiten-ID eindeutig referenziert. Des Weiteren weist jede BSE eine Fabrikmethode zum Erzeugen einer Instanz der Einheit auf. Während der Instanziierung stellt das Konfigurationsmanagement der Fabrikmethode Informationen zur Parametrisierung der BSE zur Verfügung. Jeder Parameter ist mit einer, innerhalb der Einheit eindeutigen, Parameter-ID, einem Datentyp und dem eigentlichen Wert assoziiert. Die BSE liest diese Konfiguration und

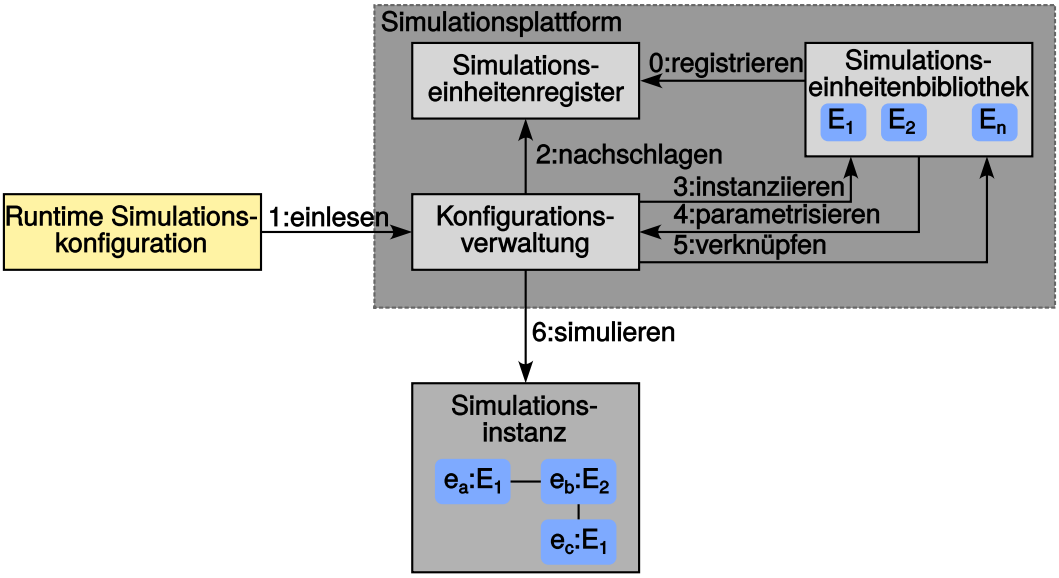


Abbildung 5.7: Aufbau und Konfiguration der Simulation

extrahiert die Parameterwerte. Die Fabrikmethode weist die gelesenen Werte den Membervariablen zu bzw. verarbeitet die gelesenen Informationen. Die Verantwortung über die korrekte Zuweisung bzw. Interpretation liegt bei der BSE, das Konfigurationsmanagement dient lediglich als Schnittstelle zur Konfigurationsdatei. Damit das Konfigurationsmanagement die korrekte Fabrikmethode aufrufen kann, registriert sich jede BSE im Simulationseinheitenregister (SER). Zu jeder Einheiten-ID wird die Fabrikmethode verlinkt, um eine Instanz der Einheit zu erstellen. Das Konfigurationsmanagement liest zur Laufzeit die Konfigurationsdatei ein und erstellt mithilfe des SERs eine parametrisierte Instanz der benötigten Einheiten. Jede instanziierte Einheit wird wiederum mittels einer eindeutigen Instanz-ID identifiziert. Nach dem alle benötigten Einheiten instanziiert und parametrisiert sind, werden sie zu einer gemeinsamen Simulation verbunden. Um zyklische Abhängigkeiten zwischen Einheiten zu erlauben, ist die Instanziierung und Verknüpfung in zwei Schritte gegliedert. Die Konfigurationsdatei beinhaltet neben der Zuweisung von Parametern, auch die Assoziationen zwischen BSEs. Mithilfe der Instanz-ID erhält das Konfigurationsmanagement eine Referenz auf die zu assoziierende Einheit. Diese Referenz wird der initiierenden BSE übergeben und diese stellt mit der Referenz die Assoziation her. Hierbei wird zwischen normalen Objektzeigern, TLM-2.0-Sockets und SystemC-Ports unterschieden. Die unterschiedlichen Typen benötigen eine unterschiedliche Handhabung: Zeiger werden einfach lokal gespeichert, TLM-Sockets werden gebunden und SystemC-Ports mit Kanälen verbunden. Nachdem alle Verknüpfungen zwischen den Einheiten hergestellt sind, kann die Simulation gestartet werden. Um einzelne Parameter oder Einheiten zwischen Simulationsläufen auszutauschen, muss lediglich die Konfigurationsdatei geändert werden.

5.1.3.2 Konfigurationsdatei

Die Konfigurationsdatei spezifiziert zwei wichtige Aspekte: Zum einen die Parametrisierung der verwendeten Simulationseinheiten, zum anderen die Assoziationen zwischen Simulationseinheiten. Zur Spezifikation wird ein IP-XACT-Design verwendet. Es ermöglicht die Spezifikation der in der Simulation enthaltenen Simulationseinheiten sowie ihre Parametrisierung. Abbildung 5.8 zeigt Ausschnitte des IP-XACT-Designs. Unter dem Element `<ipxact:componentInstances>` sind alle BSEs gelistet.

```
<?xml version="1.0" encoding="UTF-8"?>
<ipxact:design ...>
  <ipxact:componentInstances>
    <ipxact:componentInstance>
      <ipxact:instanceName>m_WaterTankPModel</ipxact:instanceName>
      <ipxact:componentRef library="Library" name="M_WaterTankPModel"
                           vendor="Library" version="0.1">
        <ipxact:configurableElementValues>
          <ipxact:configurableElementValue referenceId="m_tankHeight_cm">
             50.0 </ipxact:configurableElementValue>
          <ipxact:configurableElementValue referenceId="m_tankDiameter_cm">
             10.0 </ipxact:configurableElementValue>
          <ipxact:configurableElementValue referenceId="m_drainDiameter_cm">
            0.88 </ipxact:configurableElementValue>
        </ipxact:configurableElementValues>
      </ipxact:componentRef>
      <ipxact:vendorExtensions>
        <ipxact:file>
          <ipxact:name>
            src/application/watertank/M_WaterTankPModel.xml</ipxact:name>
          <ipxact:fileType user="xml">user</ipxact:fileType>
        </ipxact:file>
      </ipxact:vendorExtensions>
    </ipxact:componentInstance>
    ...
  </ipxact:componentInstances>
  ...
</ipxact:design>
```

Abbildung 5.8: Konfigurationsdatei zur Parametrisierung der BSE

Die Parameter der BSE sind als `<ipxact:configurableElementValue>` spezifiziert. Die IP-XACT-Design-Struktur ermöglicht nicht die Spezifikation des Parameterdatentyps. Der Parameterdatentyp ist notwendig, um eine grundlegende Überprüfung beim Einlesen der Parameter durch das Konfigurationsmanagement zu gewährleisten. Aus diesem Grund wird für jede BSE eine zusätzliche IP-XACT Komponentenspezifikation erstellt. Die Komponentenspezifikation ermöglicht die Beschreibung der Parameter mit ihren zugehörigen Datentypen. Die Komponenten im Design und in der Komponentenspezifikation sind über eindeutige IDs referenziert. Das IP-XACT-Design referenziert die Datei der IP-XACT-Komponentenspezifikation, um das Einlesen zu vereinfachen. Hierzu wird im IP-XACT-Design eine Nutzererweiterung verwendet, um den relativen Pfad zur Komponentenbeschreibung zu hinterlegen. In Abbildung 5.8 ist die Erweiterung als `<ipxact:vendorExtensions>` der Komponenten zu erkennen. Neben der Instanziierung von Komponenten ermöglicht das IP-XACT-Design die Spezifikation von Verbindungen unter den Komponenten, die sogenann-

ten `interconnections`. Dies dient der Verknüpfung der BSEs. Abbildung 5.9 zeigt den relevanten Ausschnitt. Unter dem Element `<ipxact:interconnections>` sind

```
<?xml version="1.0" encoding="UTF-8"?>
<ipxact:design ...>
  ...
  <ipxact:interconnections>
    <ipxact:interconnection>
      <ipxact:name>m_WaterTankPModel_m_controlValue</ipxact:name>
      <ipxact:activeInterface busRef="ControlValueIn"
                              componentRef="m_WaterTankPModel"/>
      <ipxact:activeInterface busRef="unkownBusRef"
                              componentRef="m_controlValue"/>
    </ipxact:interconnection>
    ...
  </ipxact:interconnections>
</ipxact:design>
```

Abbildung 5.9: Konfigurationsdatei zur Verbindung von Simulationseinheiten

die Verknüpfungen der BSE gelistet. Hierbei werden die zwei zu verknüpfenden BSEs sowie eine ID (`busRef`), welche die Verknüpfung identifiziert, spezifiziert. Die Identifikation wird benötigt, um innerhalb einer BSE, unterschiedliche Verknüpfungen zu unterscheiden. Die Elemente `<ipxact:componentInstances>` und `<ipxact:interconnections>` ermöglichen, eine komplette Simulationsinstanz zu spezifizieren.

In Abbildung 5.10 ist auszugsweise die Parameterspezifikation der zugehörigen Komponentenbeschreibung dargestellt. Die Eigenschaft `resolve=user` kennzeich-

```
<?xml version="1.0" encoding="UTF-8"?>
<ipxact:component ...>
  ...
  <ipxact:parameters>
    <ipxact:parameter parameterId="m_tankHeight_cm" type="real" resolve="user">
      <ipxact:name>m_tankHeight_cm</ipxact:name>
      <ipxact:displayName>TankHeight</ipxact:displayName>
      <ipxact:value>unsupported</ipxact:value>
    </ipxact:parameter>
    <ipxact:parameter parameterId="m_tankDiameter_cm" type="real" resolve="user">
      <ipxact:name>m_tankDiameter_cm</ipxact:name>
      <ipxact:displayName>TankDiameter</ipxact:displayName>
      <ipxact:value>unsupported</ipxact:value>
    </ipxact:parameter>
    ...
  </ipxact:parameters>
</ipxact:component>
```

Abbildung 5.10: IP-XACT Komponentenbeschreibung

net Parameter bzw. Membervariablen, welche die Konfigurationsdatei initialisiert. Für das Einlesen der Konfigurationsdatei ist vorrangig der Datentyp des Parameters wichtig, der hier spezifiziert wird. Ähnlich der Parameter spezifiziert die IP-XACT-Komponentenbeschreibung auch die Verknüpfungen der BSEs. Die IP-XACT-Notation stellt sie als Busschnittstellen dar und gibt für jede Schnittstelle eine ID und einen

Typ an. Zuletzt spezifiziert die Komponentenbeschreibung für jede Komponente die zugehörigen Header und Quelldateien mit dem Element `<ipxact:fileSets>`. Die Komponentenbeschreibung wird bei der Generierung des Quellcodes der SEB generiert und bleibt für alle Simulationsinstanzen gleich.

5.2 Fehlerinjektion

Wie im Abschnitt 3.5 motiviert, fehlt eine ganzheitliche Methode zur Fehlerinjektion. Die Grundvoraussetzung hierfür ist die Änderung des Systemzustands bzw. des Simulationszustands. Hierbei muss der Fehlerinjektor sowohl den Zustand der Testbench als auch des DUTs ändern. Neben den übergeordneten Anforderungen, wie die abstraktionsebenenübergreifende Anwendbarkeit, existieren weiterführende, detaillierte Anforderungen an den Entwurf des Injektors. Die Analyse mehrerer Fehlerfälle, auf unterschiedlichen Abstraktionsebenen, ergab folgende detaillierte Anforderungen an die Fehlerinjektoren:

A.1 Der Fehlerinjektor muss den Simulationszustand abändern. Hierbei muss er sowohl interne als auch externe Informationen abändern, d. h. er benötigt Zugriff auf Variablen vom Typ `public`, `protected` und `private`.

A.2 Der Fehlerinjektor muss unterschiedliche Datentypen unterstützen. Hierzu zählen z. B. aggregierter Datentypen und Simulationsprimitiven, als Ziel der Fehlerinjektion.

A.3 Der Fehlerinjektor muss unterschiedliche Strategien zur Fehlerbehebung bereitstellen, um eine umfassende Unterstützung von transienten Fehlern sicher zu stellen.

A.4 Der Fehlerinjektor muss sowohl die Aufrechterhaltung des Fehlers als auch eine Anpassung des Fehlers unterstützen.

A.5 Der Fehlerinjektor muss die Möglichkeit bieten, eine Reaktivierung der Simulation explizit zu spezifizieren und auszulösen. Bei einer ereignisorientierten Simulation ist es stellenweise erforderlich, nach dem Einbringen des Fehlers, explizit Ereignisse in der Simulation auszulösen, um die Ausführung von Simulationsprozessen zu starten.

A.6 Der Fehlerinjektor muss den Aufwand zur Einbringung des Fehlerinjektors in bestehende Simulationsmodelle gering halten.

A.7 Der Fehlerinjektor muss mit generischen SystemC-Simulationskernen kompatible sein und nicht auf Änderungen des Simulationskerns beruhen.

5.2.1 Fehlerinjektor

Im Folgenden werden die Anforderungen detaillierter motiviert und die Umsetzung der entwickelten Konzepte vorgestellt. Der entwickelte Ansatz beruht auf der Ersetzung von Datentypen, in den Simulationsmodellen, durch injizierbare Datentypen. Es bildet somit eine Abwandlung der mutationsbasierten Fehlerinjektion. Beim herkömmlichen Mutationsansatz stellen BSEs sowohl das korrekte als auch das fehlerbehaftete Verhalten bereit. Eine externe Steuerung aktiviert lediglich das jeweilige Verhalten. Bei dem entwickelten Ansatz wird der Simulationszustand von extern modifizierbar gehalten. Die Schnittstelle ermöglicht die Mutation des Simulationszustands und dadurch die Realisierung unterschiedlicher Fehler, durch eine externe Steuerung. Das heißt, anstelle des Vorhaltens eines fehlerbehafteten Verhaltens, werden Variablen durch eine externe Steuerung modifiziert, um das fehlerbehaftete Verhalten der BSE zu emulieren. Der ursprüngliche Datentyp wird mittels eines generischen Containers gekapselt. Der sogenannte *Fehlerinjektor* erweitert den ursprünglichen Datentyp um Schnittstellen für den externen Zugriff und ermöglicht somit die Änderung des Simulationszustands. Der ursprüngliche Datentyp bleibt im Kern erhalten, was die Unterstützung unterschiedlicher Datentypen und Simulationsprimitiven ermöglicht. Hierdurch wird die Anforderung A.2 adressiert.

Fehlerinjektoranwendung

Der injizierbare Datentyp implementiert die Schnittstelle eines intelligenten Zeigers (engl. Smartpointer). Intelligente Zeiger bzw. Zeiger im Allgemeinen sind weitverbreitet in der Softwaretechnik. „Since C, we know that pointers are important but are a source of trouble.[...]A usual approach to avoid these kinds of problems is to use ‚smart pointers'.“[56]. Somit stellt die Anforderung zur Verwendung eines intelligenten Zeigers keine praktische Einschränkung dar. Der überladene intelligente Zeiger verwendet die Aufrufkonvention und die Referenzzählung eines `shared_ptr` aus C++11 [52]. Die Modifikationen an den Simulationsmodellen beschränken sich auf die Ersetzung der Typdeklaration, des Aufrufs des Konstruktors sowie in einigen Fällen die Anpassung der Aufrufsemantik. Änderungen am funktionalen Verhalten oder an der Struktur der Simulationsmodelle sind nicht notwendig. Bereitgestellte Makro-Definitionen reduzieren die benötigten Änderungen und steigern die Benutzerfreundlichkeit. Abbildung 5.11 zeigt die benötigten Änderungen zur Erweiterung eines bestehenden Simulationsmodells für die Fehlerinjektion. Hierbei wird die Deklaration in Zeile 10 sowie die Initialisierung in Zeile 6 mit den bereitgestellten Makros ersetzt. Außerdem müssen alle Zugriffe auf den Datentyp durch Dereferenzierungsoperatoren wie in Zeile 14 ergänzt werden. Mithilfe von gängigen Werkzeugumgebungen sind die Ersetzungen ohne großen Aufwand durchzuführen. Eine weitere Möglichkeit, den Aufwand für die Ersetzungen zu reduzieren, ist die Erstellung eines benutzerspezifischen Makros, das den ursprünglichen Variablennamen durch die Dereferenzierung ersetzt. Hierdurch reduzieren sich die benötigten Ersetzungen und nur wenige Zeilen im ursprünglichen Quellcode ändern sich. Durch die Verwendung von intelligenten Zeigern, d. h. ein in der Softwaretechnik gängiges

```
1  class M_WaterTankPModel : public sc_core::sc_module {
2  public:
3    M_WaterTankPModel(sc_module_name name)
4      : sc_module(name),
5        m_waterVolume(0),
6        m_sensorWaterHight(0)
7    { ... };
8  private:
9    double m_waterVolume;
10   double m_sensorWaterHight;
11
12   void updateThread(); {
13     ...
14       m_sensorWaterHight = m_waterVolume/tankArea;
15     ...
16   };
17 ...
18 };
```

```
3    M_WaterTankPModel(sc_module_name name)
4      : sc_module(name),
5        m_waterVolume(0),
6        ees_inj_ctor("sensorValue", double, m_sensorWaterHight, 0)
7    { ... };
```

```
8  private:
9    double m_waterVolume;
10   ees_inj(M_WaterTankPModel, double) m_sensorWaterHight;
```

```
12   void updateThread(); {
13     ...
14       *m_sensorWaterHight = m_waterVolume/tankArea;
15     ...
16   };
```

Abbildung 5.11: Benötigte Änderungen zur Fehlerinjektion

Konstrukt und die Beschränkung von Änderungen auf wenige Stellen im Quellcode, wird die Anforderung A.6 erfüllt.

Injektionslokation

Im Rahmen dieser Arbeit beschränkt sich die Fehlereffektsimulation auf Membervariablen. Wie motiviert, ist das Ziel der Fehlerinjektion den Simulationszustand $X(t)$ zu modifizieren. Der persistente Systemzustand, der über Zeitschritte hinweg gilt, wird in der Regel in Membervariablen gespeichert. Die Begrenzung auf Membervariablen stellt somit keine erhebliche Einschränkung dar, bietet aber bei der Anbindung an den modellgetriebenen Entwurf erhebliche Vorteile. Die vorgestellten Fehlerinjektoren ermöglichen die Verwendung beliebiger Variablen. In dieser Arbeit wird der Fokus auf Membervariablen gelegt, um die Vorteile des modellgetriebenen Entwurfs zu nutzen. Membervariablen werden durch die Kapselung, um die Möglichkeit zur Injektion von Fehlern, d. h. die Veränderung des Systemzustands (Fehlerinjektor) sowie zum Beobachten des aktuellen Systemzustands (Wertesonde) erweitert. Die Fehlerinjektion muss unabhängig von der Sichtbarkeit der verwendeten Membervariable erfolgen, d. h. sowohl `public` als auch `protected` und `private` Variablen müssen ver-

fälscht werden. Basierend auf der Quellcodemodifikation, durch Austauschen des ursprünglichen Datentyps mit dem Fehlerinjektor, vom Datentyp `C_InjectorProbe<>`, wird dies gewährleistet.

Der jeweilige Fehlerinjektor registriert sich automatisch, während der Instanzierung, bei einem globalen Handler, der alle Fehlerinjektor verwaltet. Der Handler verwaltet eine interne Datenstruktur, in der die Zeiger auf alle Fehlerinjektoren im Simulationsmodell sowie deren ID hinterlegt sind. Dies dient der Fehlersteuerung als einheitliche Schnittstelle, um auf die Fehlerinjektoren zuzugreifen. Neben dem Zugriff auf die vom Fehlerinjektor eingenommene Variable ist es wichtig, auf die weiteren Informationen der Klasse Zugriff zu haben. Dies wird vor allem bei der Reaktivierung der Simulation, nach einer Fehlerinjektion, benötigt. In C++ existiert das Konzept von `friend`-Assoziationen, hierbei erhält eine externe Klasse Zugriff auf die internen Membervariablen. Bei der Verwendung des bereitgestellten Makros zur Deklaration des Fehlerinjektors (`C_InjectorProbe`) wird gleichzeitig die Klasse, zur Spezifikation eines Reaktivierungsverhaltens, als `friend`-Klasse definiert. Die Definition der `friend`-Assoziation erfolgt, bei Verwendung der bereitgestellten Makros, komplett transparent für den Nutzer. Abbildung 5.12 zeigt das bereitgestellte Makro in den Zeilen 1–5 sowie dessen Anwendung in Zeile 8. Die Ersetzungen mit den vor-

```
1 // macro provided by framework
2 #define ees_inj(cl, ty)
3   template<typename ContainingClass, int selector>     \
4   friend class ees_trigger_spec;                       \
5   C_InjectorProbe<cl, argument_type<void(ty)>::type, 0>
6
7 // declaration within the class
8 ees_inj(C_SimulatedClass, float) m_MemberVariableToChange;
```

Abbildung 5.12: Makro-Unterstützung zur Fehlerinjektor Deklaration

gestellten Makros und die benötigten Anpassungen des Zugriffs stellen die einzigen Modifikationen der Simulationsmodelle dar. Kapitel 6 stellt eine Erweiterung vor, welche die manuellen Anpassungen weiter reduziert.

Injektorzugriffskontrolle

Fehler manifestieren sich unterschiedlich im System. Permanente Fehler z. B. bleiben bestehen und überschreiben somit jede weitere Änderung durch die Simulation. Das heißt, nach der Fehlerinjektion muss der Fehlerinjektor jegliche Änderung durch die eigentliche Simulation verhindern. Um dies zu realisieren, wird die Dereferenzierung des Fehlerinjektors verwendet. Durch die Dereferenzierung detektiert der Fehlerinjektor Schreib- und Lesezugriffe. Bei jedem Zugriff wird überprüft, ob aktuell eine Injektion aktiv ist. Ist dies der Fall, wird der durch die Injektion gesetzte Wert zurückgeliefert. Überschreibt die Simulation diesen Wert, wird beim nächsten Lesezugriff bzw. bei der damit verbundenen Dereferenzierung, der injizierte Wert wiederhergestellt. Der Fehlerinjektor verwaltet hierzu mehrere interne Datenstrukturen. Im `access container` wird der aktuelle Wert des Fehlerinjektors abgelegt. Die Simulation greift lesend und schreibend auf diese Datenstruktur zu. Der `error container` speichert

den zu injizierenden Wert und der `backup container` den von der Simulation zuletzt geschriebenen Wert. Bei aktiver Injektion wird bei jedem Lese- oder Schreibzugriff der Wert des `access containers` mit dem Wert des `error containers` verglichen. Sind die Werte unterschiedlich bedeutet dies, dass die Simulation den Wert des Fehlerinjektors geändert hat. In diesem Fall wird der neue Wert in den `backup container` verschoben und der Wert des `error containers` erneut gesetzt. Durch dieses Vorgehen wird garantiert, dass immer der zu injizierende Wert im `access container` steht und gleichzeitig der Fehlerinjektor alle Schreibzugriffe aufgezeichnet. Hierdurch kann ein Fehler aufrechterhalten werden, trotz Änderungen des Simulationszustands. Dies setzt einen Teil der Anforderung A.4 um.

Fehlerbehebung

Das Aufzeichnen der Schreibzugriffe hat einen weiteren Grund. Transiente Fehlern weisen unterschiedliche Möglichkeiten zur Fehlerbehebung auf. Durch die Untersuchung verschiedener Anwendungsfälle haben sich drei Strategien zur Fehlerbehebung herausgebildet.

1. Wird ein Fehler z. B. durch ein Zurücksetzen des Systems behoben, nimmt das System den Zustand vor Auftreten des Fehlers ein.

2. Wird ein Fehler durch erneutes Setzen des Werts behoben, nimmt der Systemzustand den neu geschriebenen Wert an. Beispiel ist die Verfälschung einer Speicherzelle, die durch erneutes Beschreiben korrigiert wird.

3. Bei einem Short-To-Ground-Fehler wird häufig ein Signal auf den Wert null gesetzt, solange der Kurzschluss andauert. Wird der Kurzschluss behoben, wird automatisch der zuletzt gesetzte Wert, der während des Bestehens des Kurzschlusses gesetzt wurde, wiederhergestellt.

Um die unterschiedlichen Fehlerfälle zu unterstützen, implementiert der Fehlerinjektor drei unterschiedliche Strategien zur Fehlerbehebung. Abbildung 5.13 demonstriert die Strategien am Beispiel eines verfälschten Registerwertes. Die erste Werte-

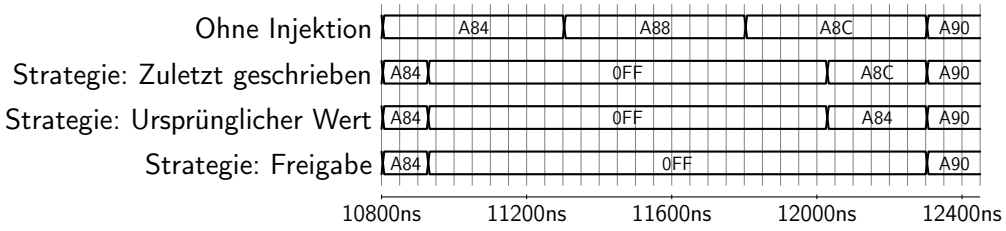


Abbildung 5.13: Unterstützte Rücksetzstrategien des Fehlerinjektors

folge zeigt, wie sich der Wert eines Registers über die Zeit ändert. Sie stellt somit das Referenzverhalten dar. Zum Zeitpunkt 10975 ns wird das Register durch den Fehlerwert 0xFF geändert. In der zweiten Zeile wird der Wert des zuletzt geschriebenen Registerinhalts sofort wiederhergestellt. In der dritten Zeile hingegen wird sofort der Wert, der zum Zeitpunkt der Injektion gültig war, wiederhergestellt. Im letzten

Fall wird lediglich die Fehlerinjektion zum Zeitpunkt 12300 ns freigegeben. Erst mit dem nächsten Schreibzugriff wird der gleiche Systemzustand wie in der Referenz hergestellt.

Um die unterschiedlichen Strategien zu realisieren, verwendet der Fehlerinjektor die bereits vorgestellten internen Kopien des Variablenwerts. Die vorherigen Beispiele stellten bereits einige vor. Beim Wiederherstellen des zuletzt geschriebenen Wertes wird die Kopie aus dem `backup container` wiederhergestellt. Wie weiter oben beschrieben, wird bei aktiver Injektion jeder durch die Simulation gesetzte Wert in den `backup container` kopiert, bevor das Injektionsartefakt wiederhergestellt wird. Somit beinhaltet der `backup container` den zuletzt gesetzten Wert. Um den ursprünglichen Wert wiederherzustellen, wird zusätzlich ein `original backup container` vorgehalten. Der Wert wird gesetzt, wenn die Injektion zum ersten Mal den Simulationswert überschreibt. Während der Injektion wird der `original backup container` nicht mehr überschrieben und ermöglicht damit die Wiederherstellung des ursprünglichen Werts. Bei der einfachen Freigabe wird kein Wert wiederhergestellt, sondern lediglich das erneute Überschreiben durch den Fehlerinjektor deaktiviert. Wie weiter oben beschrieben, stellt der Fehlerinjektor bei jeder Dereferenzierung den injizierten Wert wieder her, um somit eine dauerhafte Injektion zu ermöglichen (Anforderung A.4). Bei der einfachen Freigabe wird das Überschreiben deaktiviert und somit mit dem nächsten Schreibzugriff der Simulation der Fehler behoben. Durch die unterschiedlichen Strategien wird Anforderung A.3 adressiert. In Abbildung 5.14 ist

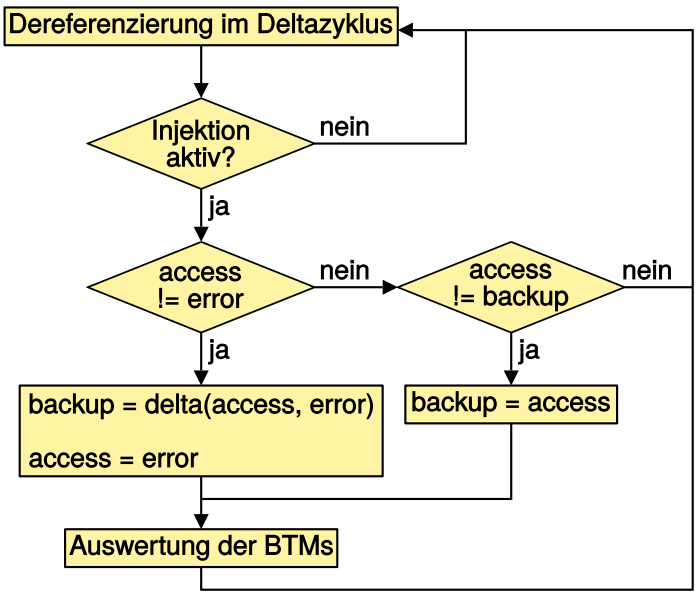


Abbildung 5.14: Überprüfungen und Aktionen bei Dereferenzierung

das Verhalten des Fehlerinjektors bei einer Dereferenzierung dargestellt. Bei einer aktiven Injektion und der Änderung des `access containers` ist zu beachten, dass beim Setzen des `backup containers` der `error container` mitberücksichtigt wird. Dies ist wichtig, wenn es sich um zusammengesetzte Datenstrukturen handelt. Wird der Fehlerinjektor auf ein Array von Daten, wie zum Beispiel ein Bild oder auf ein

Transaktionsobjekt mit mehreren Feldern angewandt, ist es nicht ausreichend, alleinig den `access container` zu sichern. Wird lediglich ein Teil der Daten im `access container` durch die Simulation geändert, aber nicht der injizierte Fehler, würde sich der Fehler in den `backup container` propagieren und ein Beheben des Fehlers wäre nicht mehr möglich. Aus diesem Grund werden nur die Unterschiede zwischen dem `error container` und dem `access container` in den `backup container` geschrieben. Hierdurch werden lediglich die Änderungen durch die Simulation berücksichtigt und nicht die Änderungen durch die Fehlerinjektion.

Fehlerausbreitung

Ähnlich der Möglichkeiten zur Fehlerbehebung unterscheiden sich Fehler im Verhalten bei der Injektion. Es existieren Fehler, die sich sofort durch das System propagieren, z. B. ein Kurzschluss in der Leitung einer kombinatorischen Schaltung. Auf der anderen Seite existieren Fehler, die sich erst durch den regulären Zugriff auf die verfälschte Information im System propagieren. Die verborgenen Fehler haben keine unmittelbare Auswirkung auf das System. Bei der Injektion der sofort propagierenden Fehler werden zwei Fälle unterschieden. In vielen Fällen ist die Simulation sensitiv auf die Werteänderung und der Simulationskern wird die jeweiligen Prozesse automatisch auslösen, wenn der Fehlerinjektor den Simulationswert ändert. Dies ist zum Beispiel bei allen Fehlerinjektoren, die auf `sc_signal` basieren, der Fall. Bei der Injektion in C++-Modellierungsprimitiven oder bei abstrakten TLM-2.0-Modellen findet häufig keine automatisierte Auslösung von Prozessen statt. In diesen Fällen muss der Fehlerinjektor selbst die Simulation reaktivieren. Für die Fehlersimulation ist es somit wichtig, Fehler *asynchron* zu injizieren. Hierdurch wird die Injektion von Fehlern außerhalb der originalen Simulationsereignisse bezeichnet.

Die Simulation wird um zusätzliche Ereignisse erweitert, die neue Reaktivierungen von Prozessen mit sich bringen. Setzt ein Prozess zu unterschiedlichen Zeitpunkten ein Signal, ist es wichtig, dass der Zeitpunkt, an dem der Fehlerinjektion das Signal ändern darf, unabhängig von den Zeitpunkten ist, an denen der Prozess das Signal ändert. Abbildung 5.15 zeigt die durch die Fehlerinjektion veränderten Simulationsereignisse. Der Prozess b_1 ändert den Wert des Signals x. Der Prozess b_2

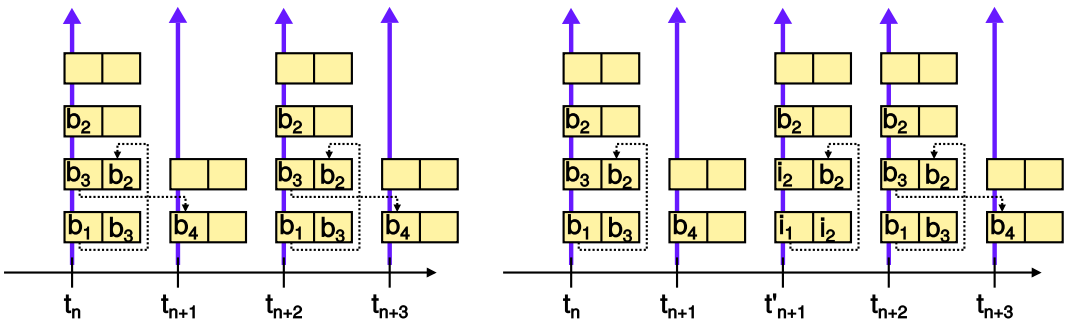


Abbildung 5.15: Änderung der Simulationsschritte durch die Injektion

verarbeitet das Signal und ist sensitiv zum Signal x. Die gleiche Struktur, lediglich mit Variablen, modellieren die Prozesse b_3 und b_4. Hierbei verändert Prozess b_3 die

Variable y und Prozess b_4 verarbeitet diese. Der Unterschied besteht im Auslöseverhalten der Prozesse. Der Prozess b_4 wird periodisch ausgelöst und ist nicht sensitiv auf Veränderungen der Variable. Der linke Teil der Abbildung 5.15 zeigt die regulären Schreib- und Lesezugriffe. Zum Zeitpunkt t_n wird sowohl das Signal (mittels b_1) als auch die Variable y (mittels b_3) beschrieben. Aufgrund der Sensitivität wird bereits im selben Deltazyklus das Signal x gelesen. Die Variable y hingegen erst im nächsten Zeitschritt, wenn der Prozess b_4 regulär ausgeführt wird.

Im rechten Teil der Abbildung ist die gleiche Abfolge mit einer Injektion in die Variable und in das Signal dargestellt. Die Fehlerinjektion i_1 verändert das Signal x und die Fehlerinjektion i_2 die Variable y. Es ist zu erkennen, dass sich die Fehlerinjektion zu einem sofortigen Auslösen des Prozesses b_2 führt und sich somit der Fehler sofort propagiert. Die Änderung der Variable hingegen wird erst mit dem regulären Zugriff in b_4 weiter propagiert.

Je nach Implementierung muss der Fehlerinjektor deswegen Aktionen in der Simulation auslösen, um eine sofortige Auswirkung, d. h. Aktivierung des Prozesses b_4, zu gewährleisten. Eine Möglichkeit dies zu bewerkstelligen, wäre es die betreffenden Prozesse sensitive für den Fehlerinjektor zu implementieren. Dies würde eine erhebliche Überarbeitung der Simulationsmodelle mit sich bringen und damit der Anforderung A.6 entgegenstehen. Deswegen wird ein separates Modul bereitgestellt, in dem der Benutzer die gewünschten Aktionen nach einer Fehlerinjektion spezifiziert. Im Kontext von SystemC sind wichtige auslösende Ereignisse:

- SystemC-Prozess-Trigger, hierzu zählen z. B. `sc_event` oder `sc_clock`,
- blockierende SystemC-Wait-Anweisungen, die durch `sc_event` aufgelöst werden,
- TLM-Funktionsaufrufe, wie z. B. `b_transport` oder
- Synchronisationsobjekte, wie z. B. `sc_semaphore` oder `sc_mutex`.

Vorsätzlich von der Liste ausgeschlossen ist das Abbrechen von zeitlich festgelegten Wait-Anweisungen. Zum einen bedingt dies eine Änderung am Simulationskern, da Ereignisse aus der internen Schedulingliste entfernt werden müssten. Zum anderen ist das Abbrechen einer zeitlich definierten Wait-Anweisung meist nicht in der Intention der Simulation. Ein Beispiel, das häufig zeitlich festgelegte Wait-Anweisungen verwendet, ist das Abtasten eines Signals. In solchen Fällen ist es unerwünscht, dass durch eine Verfälschung des Signals eine neue Evaluation, bzw. Abtastung stattfindet. Es ist gewünscht, dass sich kurzfristige Signaländerungen nicht über die Abtastung hinweg propagieren. An dieser Stelle sei nur darauf hingewiesen, dass die Fehlerinjektion die Abtastzeitpunkte jedoch verändern kann. Der Unterschied zwischen den beiden Fehlern liegt in der assoziierten Variablen. Um einen Jitter bei der Abtastung zu injizieren, wird der Fehlerinjektor der Variable mit der Abtastperiode zugewiesen und nicht dem Signal selbst.

Injektorimplementierung

Eine wichtige Anforderung ist die Anforderung A.2. Sie fordert vom Fehlerinjektor, dass die Handhabung unterschiedlicher Datentypen möglich ist. Der bereitgestellte Fehlerinjektor muss generisch einsetzbar sein. Der Anwender sollte nicht für jeden Anwendungsfall einen neuen Fehlerinjektor erstellen müssen. Die C++-Standardbibliothek bietet für eine von Datentypen unabhängige Implementierung das Konzept der Templates. Diese Methode ermöglicht das Bereitstellen einer generischen Implementierung. Der Fehlerinjektor ist als Template realisiert, das mit unterschiedlichen Datentypen spezialisiert wird. Neben dem Datentyp der ursprüngliche Variablen wird das Template mit dem Datentyp der beinhaltenden Klasse sowie einer optionalen ID spezialisiert. Der Datentyp der ursprünglichen Variable ist die wichtigste Eigenschaft zur Spezialisierung der Templates. Dieser wird benötigt, um die internen Container zu erstellen. Wie vorgestellt, ersetzt der Anwender den ursprünglichen Datentypen durch den Fehlerinjektor. Durch die Dereferenzierung erlangt die Simulation Zugriff auf den internen `access container`, der durch die Spezialisierung des Templates den gleichen Datentyp wie die ursprüngliche Variable aufweist. Wie in Abbildung 5.14 dargestellt, operiert der Fehlerinjektor auf den internen Containern um entweder den geänderten oder korrekten Wert zurückzuliefern. Um dies zu realisieren, müssen die Container folgende Operationen bereitstellen:

- die Konvertierung in den ursprünglichen Datentypen,
- die Änderung des Werts des Containers,
- die Feststellung der Gleichheit von Containerinhalten bzw. einzelner Unterschiede und
- das Kopieren von Containerinhalten.

Die Verwendung der benötigten Operationen ist in Abbildung 5.14 dargestellt. Bei der Fehlerinjektion wird das Injektionsartefakt dem `error container` zugewiesen. In der zweiten Bedingung wird überprüft, ob die beiden Container `access container` und `error container` ungleich sind. Bei der Bestimmung der Funktion `delta()`, zwischen zwei Containern, werden die Unterschiede der zusammengesetzten Daten ermittelt. In der letzten Anweisung wird der komplette Inhalt des `error containers` in den `access container` kopiert. Bei der Dereferenzierung muss der Container in den ursprünglichen Datentyp konvertiert werden. Um diese Funktionalität zu realisieren, müssen die Container bzw. die spezialisierten Templates die benötigten Operationen bereitstellen. Eine detaillierte Schnittstellendefinition ist in Abbildung 5.16 gegeben. Die interne Implementierung des Fehlerinjektors ist vollständig Datentyp unabhängig. Es wird die *Trait-Programmiermethodik* verwendet, um dies zu erreichen. Die Methode trennt die benötigten Zugriffsmethoden von der eigentlichen Funktionalität und realisiert diese über spezialisierte Templates. Das Konzept ermöglicht zum einen die Implementierung des Fehlerinjektors unabhängig vom eigentlichen Datentyp, da alle Anweisungen die vom Datentyp abhängen in der Trait-Klasse stehen. Zum anderen erlaubt das Konzept die einfache Erweiterung mit anwendungsspezifischen Datenstrukturen. In Rahmen der Arbeit werden Trait-Klassen für die Grunddatentypen

von C++, für den zusammengesetzten Datentyp `array`, für die Simulationsprimitiven von `sc_signal` und den `tlm_generic_payload` angeboten. Abbildung 5.17 stellt das verwendete Konzept zur Spezialisierung der Templates dar. Bei den Grunddatentypen von C++ werden lediglich die benötigten Funktionen auf die Funktion der Datentypen weitergeleitet. Aufgrund der Tatsache, dass alle nativen C++ Datentypen den Vergleichsoperator implementieren, werden alle nativen C++-Datentypen durch eine Trait-Klasse abgedeckt. Ähnlich verhält es sich mit der Gruppe von `sc_signal`-Datentypen. Auch hier kann auf die Funktionen der nativen Datentypen zugegriffen werden. Hierbei müssen lediglich die Funktionen `read()` und `write()` der SystemC-Signale verwendet werden, um auf die Inhalte zuzugreifen. Die Weitergabe des Datenzugriffs garantiert, dass die Spezialisierung für unterschiedliche Signal-Datentypen anwendbar ist. Anhand des `sc_signal`-Datentyps ist die Unterscheidung zwischen zusammengesetzten und einfachen Datentypen gut zu erkennen. Bei zusammengesetzten Datentypen wird zwischen dem aggregierenden Datentyp und dem Kerndatentyp unterschieden. Bei der Dereferenzierung erfolgt der Zugriff auf den Kerndatentyp. Der Zugriff bei `sc_signal`-Datentypen erfolgt durch die Funktionen `read()` und `write()`. Für den Zugriff auf eine zusammengesetzte Datenstruktur wie `array` ist hierfür eine Indizierung beim Zugriff auf einzelne Elemente notwendig. Im Falle von zusammengesetzten Datentypen kommt auch zum ersten Mal die Detektion von Unterschieden zwischen zwei Container zum Tragen. Die Fehlerinjektoren müssen im Falle von zusammengesetzten Datentypen Unterschiede bei den einzelnen Elementen detektieren. Die letzte bereitgestellte Trait-Spezialisierung behandelt den zusammengesetzten Datentypen `tlm_generic_payload`. Hierbei ist die Behandlung der unterschiedlichen Informationen wie Adresse oder Datenlänge wichtig, aber auch die Adressierung der einzelnen Bytes der Nutzdaten. Der Fehlerinjektor verwendet den Template-Parameter `CompositeType`, der im Falle von SystemC-Signalen mit dem Template-Parameter `sc_signal<BasicType, POL>` spezialisiert wird.

Neben den Template-Parametern definiert die Trait-Klasse die zwei internen Datentypen `basicType` sowie `containerType` zur Anpassung an die unterschiedlichen Datentypen zur Injektion. Die Spezialisierung der Trait-Klasse weist die entsprechenden Datentypen zu. Bei der Spezialisierung als `sc_signal<BasicType, POL>`

```
1  static CompositeType& convContainerToInjTypeRef( containerType* container );
2  static CompositeType* convContainerToInjTypePtr( containerType* container );
3  static const CompositeType& getValue(containerType* container,
4                                       int offset=-1);
5  static void setValue(containerType* container, const basicType& value,
6                       int offset=-1);
7  static void setValue(containerType* container, basicType* value, int offset=-1);
8  static void setValue(containerType* container,
9                       const basicType& potChanges, const basicType& original);
10 static bool equals(containerType* containerLHS, containerType* containerRHS,
11                    int offset=-1);
12 static void copyContainer(containerType* containerLHS,
13                           containerType* containerRHS);
```

Abbildung 5.16: Benötigte Funktionen zur Datentypenunterstützung

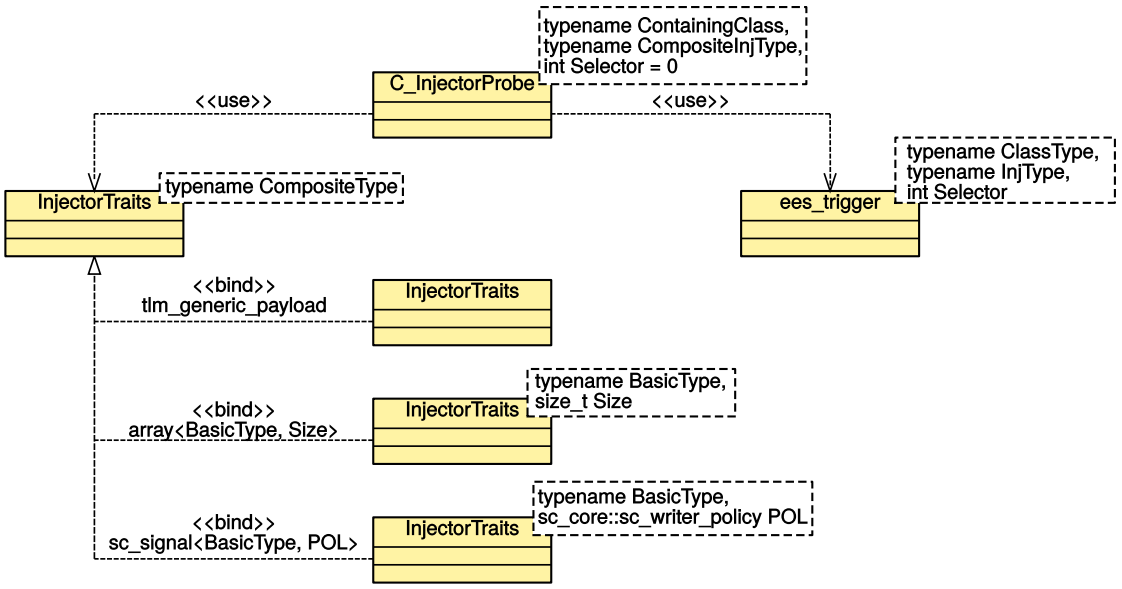


Abbildung 5.17: Struktur der Implementierung des Fehlerinjektors

entspricht der `containerType` dem `sc_signal`-Datentyp und der `basicType` dem elementaren Datentyp. Die generische Implementierung des Fehlerinjektors verwendet die beiden definierten Datentypen, um die internen Container zu erzeugen. Vor allem die Deklaration der Konvertierungsfunktionen verwendet die beiden Datentypen. Wird der Fehlerinjektor auf anwendungsspezifische Datentypen angewendet, im Besonderen, zusammengesetzte Strukturen, muss der Anwender die benötigte Trait-Klasse bereitstellen.

Neben der generischen Programmierung zur Behandlung unterschiedlicher Datentypen bietet der Ansatz der Spezialisierung von Templates einen weiteren Vorteil. Wie weiter oben motiviert, muss der Anwender in speziellen Fällen die Reaktivierungseigenschaften bei der Fehlerinjektion spezifizieren. Um die Spezifikation für den Anwender komfortable und flexible zu ermöglichen, wird die Spezialisierung des Injektor-Templates verwendet. Die Spezialisierung erfolgt über die Klasse `ees_trigger` wie in Abbildung 5.17 dargestellt. Das Fehlerinjektor-Template stellt zwei weitere Template-Parameter bereit. Der Erste gibt die Klasse, in welcher der Fehlerinjektor beinhaltet ist, an. Dieser Parameter legt, bei Fehlerinjektoren vom gleichen Injektionstyp, die Zugehörigkeit zu der jeweiligen Klasse fest. Der zweite Parameter ist eine optionale Selektions-ID, um Fehlerinjektoren vom gleichen Injektionstyp und in derselben Klasse zu unterscheiden. Mithilfe der drei Template-Parameter: Die auslösende Klasse, der Datentyp des Fehlerinjektors sowie einer fortlaufenden ID, ist es möglich unterschiedliche Reaktivierungsmethoden bereitzustellen und flexibel einer Klasse von Fehlerinjektoren zuzuweisen. Über die Spezialisierung aller Template-Parameter kann der Anwender jedem einzelnen Fehlerinjektor, eine dedizierte Reaktivierungsmethode zuweisen. Auf der anderen Seite kann der Anwender über die partielle Spezialisierung eine Reaktivierungsmethode für eine Menge von Klassen oder Datentypen angeben. Besitzt eine Klasse mehrere Fehlerinjektoren, aber nach

der Injektion wird immer die gleiche Routine gestartet, wird die Reaktivierungsmethode einem teilweise spezialisierten Template zugewiesen, bei dem der Fehlerinjektor-Datentyp nicht festgelegt ist. Existieren Fehlerinjektoren, die immer das gleiche Verhalten hervorrufen, ähnlich des Datentyps `sc_signal`, ist es möglich, eine Reaktivierungsmethode für alle Fehlerinjektoren dieses Typs anzugeben. Hierzu wird die beinhaltende Klasse sowie die fortlaufende ID als Template-Parameter nicht spezialisiert und damit das Reaktivierungsverhalten allen Fehlerinjektoren mit dem gleichen Datentyp zugewiesen. Auf der anderen Seite erlaubt die Selektions-ID, bei Fehlerinjektoren vom gleichen Datentyp in der gleichen Klasse, eine Unterscheidung bei dem Reaktivierungsverhalten.

5.2.2 Ergänzende Verwendung von Fehlerinjektoren

Im Folgenden werden zwei häufige Anwendungsfälle des Fehlerinjektors vorgestellt und gezeigt, dass die vorliegende Implementierung anwendbar ist.

Transaktionsbasierte Injektion

Die Systemverifikation entwickelt sich immer stärker zur transaktionsbasierten Modellierung [21, 23]. Eine Systembewertung entlang der Entwurfsphasen wird angestrebt. Der vorgestellte Fehlerinjektoransatz ermöglicht die Fehlerinjektion in die transaktionsbasierten Modelle. Um eine ähnliche Vorgehensweise wie beim Saboteur-Ansatz zu erhalten, wird ein zusätzliches Adaptermodul bereitgestellt. Das Adaptermodul fügt das ausgetauschte Transaktionsobjekt dem Systemzustand $X(t)$ hinzu, indem es dieses in einer Membervariable zwischengespeichert. Diese Variable wird mit einem Fehlerinjektor assoziiert, um die Fehlerinjektion in das Transaktionsobjekt zu ermöglichen. Hierdurch kann der Fehlerinjektor den Inhalt der Transaktion, z. B. den `tlm_generic_payload`, verfälschen. Die Entwicklung eigenständiger transaktionsbasierte Fehlerinjektoren ist nicht notwendig. Das Vorgehen beschränkt sich nicht auf Transaktionen, sondern ist auf weitere Datentypen anwendbar. Hierdurch wird ein allgemeines Vorgehensmodell dargestellt, um den in dieser Arbeit gezeigten Ansatz als Saboteur-Ansatz zu verwenden.

Blackbox Verifikation

Bei Anwendungsfällen, in denen Systemmodelle von Drittanbietern bereitgestellt werden, kann es vorkommen, dass die Simulation nicht im Quelltext vorliegt, bzw. der Quelltext verschlüsselt ist. Der vorgestellte Ansatz basiert auf Quellcodeänderungen und ist somit in diesen Fällen nicht direkt anwendbar. In diesen Fällen empfiehlt sich die Injektion an den Schnittstellen der Systemteile, die als Blackbox vorliegen. Mit den vorgestellten Adaptermodulen ist es einfach möglich, die Eingangssignale der Blackbox zu verfälschen. Auf Grund der Tatsache, dass die Eingangssignale indirekt den Systemzustand bedingen, stellt das angedachte Vorgehen keine schwerwiegende Einschränkung bei der Qualifikation von Systemteilen Dritter dar. Im Falle von verschlüsselten, bzw. vorkompilierten Systemmodellen, ist es möglich, dass der Zulieferer direkt die Fehlerinjektoren in das Systemmodell einfügt und mit verschlüsselt

bzw. kompiliert. Dadurch wird eine zusätzliche Schnittstelle geschaffen, die es dem Tester erlaubt, an den vorgesehenen Stellen Fehler zu injizieren. Die interne Struktur des Systems, bzw. dessen Implementierung, ist weiterhin geschützt.

5.2.3 Wertesonde

Neben dem Verändern des Simulationszustands ist auch das Erfassen des aktuellen Simulationszustands wichtig. Im Speziellen bei der Spezifikation der Fehler ist es wichtig, Fehler abhängig eines gewissen Systemzustands auszulösen bzw. den injizierten Wert abhängig vom aktuellen Systemzustand zu bestimmen. Weitere Details sind in Abschnitt 5.3.1 dargestellt. Um den lesenden Zugriff zu gewährleisten, verwendet der Ansatz das Konzept von Wertesonden. Es existieren zwei Versionen der Wertesonden-Implementierung, die *passive Wertesonde* und die *aktive Wertesonde*. Die passive Wertesonde ist in Bezug auf die Simulationsperformanz und die benötigten Änderungen von Vorteil. Für die Spezifikation von Zustandsübergangsbedingungen im BTM ist es nicht möglich passive Wertesonden zu verwenden. Lediglich aktive Wertesonde bieten die Eigenschaften zur Verwendung in Zustandsübergangsbedingungen. Dies liegt an der Tatsache, dass nur aktive Wertesonden die BTM-Auswertung bei Änderung des Werts der Wertesonden erneut auslösen. Abbildung 5.18 verdeutlicht das Konzept. Hierbei handelt es sich bei der Wertesonde aP_1 um eine

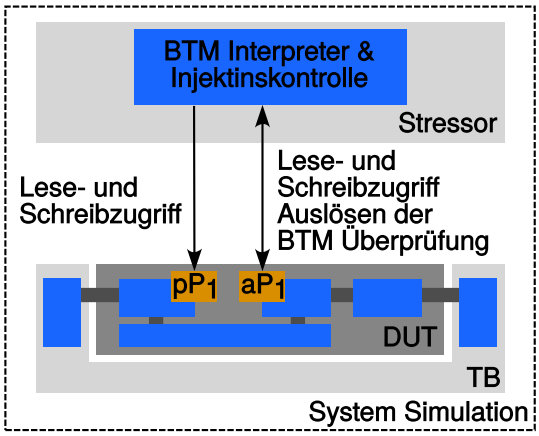


Abbildung 5.18: Skizze der ausgelagerten Ansteuerung des Fehlerinjektors

aktive Wertesonde und bei pP_1 um eine passive Wertesonde. In der Abbildung wird durch die Pfeilrichtung angezeigt, dass bei passiven Wertesonden der Zugriff nur durch den BTM-Interpreter erfolgt. Bei aktiven Wertesonden ist zusätzlich ein Auslösen der BTM-Überprüfung durch die Wertesonde möglich.

Die passive Wertesonde gewährt lediglich Lesezugriffe auf die assoziierte Variable. Realisiert wird der Zugriff mittels eines Templates, das mit dem ursprünglichen Datentyp assoziiert wird. Über Zeiger auf die ursprüngliche Variable ist es der Fehlersteuerung möglich, auf den Datenwert zu zugreifen. Bei der Realisierung handelt es sich um eine zusätzliche Klasse, die mit der Variablen verbunden ist und die Variable um die benötigten Schnittstellen erweitert. Die Erweiterung ist nur bei der

Instanziierung durchzuführen und die ursprüngliche Variable bleibt bestehen, d. h. die Variablenzugriffe müssen nicht über Zeigerdereferenzierungen erfolgen. Dieser Wertesondentyp kann nur verwendet werden, wenn die initiale Aktion vom BTM ausgelöst wird. Bei der Verwendung in Zustandsübergangsbedingungen, d. h. die Wertesonde löst eine Überprüfung des BTMs aus, muss der aktive Wertesondentyp verwendet werden.

Die Implementierung der aktiven Wertesonde besteht, ähnlich dem Fehlerinjektor, aus dem Konstrukt eines intelligenten Zeigers. Der ursprüngliche Datentyp wird durch die Wertesonde ersetzt. Durch die Natur des Zeigers werden alle Dereferenzierungen und somit alle Lese- und Schreibzugriffe detektiert. Bei einer Änderung kann die Wertesonde die Evaluation des BTMs auslösen. Hierdurch leitet eine aktive Wertesonde, Werteänderungen sofort an das BTM weiter. Die Implementierung ist hierdurch sehr ähnlich eines Fehlerinjektors, der auch Werteänderungen detektiert und die Evaluation des BTMs auslöst. Bei der weiteren Betrachtung wird deswegen, in Bezug auf den lesenden Zugriff auf die Systemsimulation, nicht zwischen Fehlerinjektor und aktiver Wertesonde unterschieden.

Sowohl die aktive als auch die passive Wertesonde registrieren sich bei einem übergeordneten Handler mittels einer eindeutigen ID. Die BTM-Instanzen verwenden einen übergeordneten Handler, um auf die Wertesonden zuzugreifen. Die Zugriffe auf die aktiven, passiven Wertesonden sowie die Fehlerinjektoren erfolgen über die gleiche Schnittstelle.

5.2.4 Fehlerinjektionsablauf

Abbildung 5.19 zeigt den resultierenden Analyseablauf. Ausgangspunkt bildet die

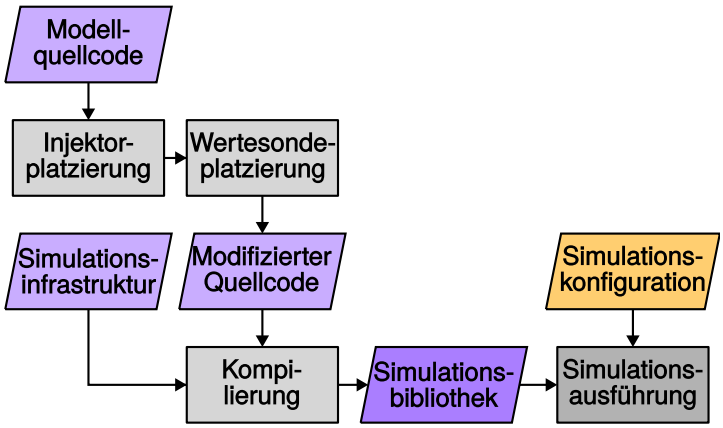


Abbildung 5.19: Schritte zur Durchführung der Fehlereffektsimulation

Simulationsmodellimplementierung. Die Implementierung wird mit den Fehlerinjektoren erweitert, d. h. Fehlerinjektoren ersetzen die bestehenden Variablendeklarationen an den gewünschten Injektionspunkten. Zusätzlich wird die Platzierung von Wertesonden vorgenommen, die der Abschnitt 5.4 vorstellt. Der modifizierte Quellcode wird mit der Simulationsinfrastruktur kompiliert. Diese Simulationsinfrastruktur

beinhaltet die Erweiterungen aus Abschnitt 5.1 als auch die Infrastruktur zur Steuerung der Fehlerinjektoren, die im folgenden Abschnitt vorgestellt wird. Aus der so gewonnenen Simulationseinheitenbibliothek werden einzelne Systemsimulationen erzeugt. Hierzu wird eine Konfigurationsdatei bereitgestellt, welche die Systemsimulation spezifiziert. Diese Simulation bildet die Grundlage für die Bewertung der funktionalen Sicherheit.

5.3 Injektionssteuerung

Die Spezifikation des zu injizierenden Verhaltens ist ein kritischer Punkt bei der Fehlerinjektion. Allgemein lässt sich jeder Fehler durch den Auftrittszeitpunkt, die durch den Fehler verursachte Abweichung sowie die Dauer des Fehlers beschreiben. Jeder Fehlerinjektor wird mit einer solchen Beschreibung konfiguriert. Abbildung 5.20 zeigt die Trennung von Fehlerinjektor und Fehlerbeschreibung. Bei der Spezifikation des

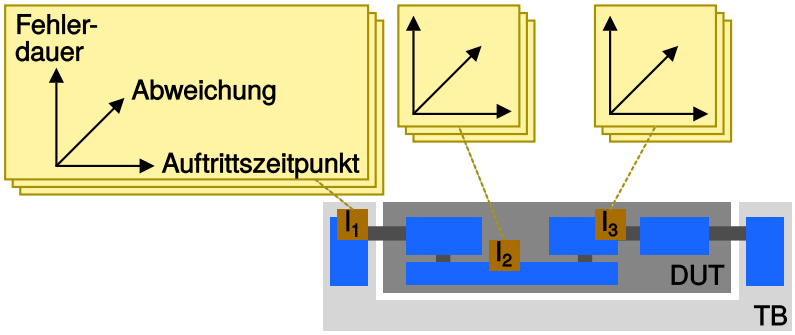


Abbildung 5.20: Trennung von Fehlerinjektor und Fehlerspezifikation

Fehlers ist zu beachten, dass das Fehlerverhalten auf unterschiedlichen Abstraktionsebenen beschrieben werden muss. Die reine Verwendung von zuvor spezifizierten Fehlerfällen, wie es vorrangig auf unteren Abstraktionsebenen erfolgt, ist nicht ausreichend. Während auf Gatterebene sehr ähnliche Fehler auftreten, z. B. der Stuck-at-Fehler, sind Modelle auf Systemebene so verschieden, dass kein einheitliches Fehlermodell bereitgestellt werden kann. Fehlerfälle bei abstrakten Modellen sind sehr stark vom modellierten System abhängig und somit sehr schwer allgemeingültig zu spezifizieren. Ein weiterer Punkt bei höheren Abstraktionsebenen ist, dass sich die Fehlerinjektion auf die in der Systemsimulation modellierten Systemteile bzw. ihrem Abstraktionsniveau beschränkt. Beeinflussen abstrahierte oder nicht modellierte Systemteile den Fehlerzustand, muss deren Einfluss auf die im Simulationsmodell enthaltenen Bestandteile in die Spezifikation des Fehlerverhaltens einfließen. Sind die Eigenschaften des realen Protokolls, z. B. die Rahmenstruktur, nicht modelliert, sondern lediglich ein abstrakter transaktionsbasierter Informationsaustausch, müssen Fehler auf die Ebene der Transaktion angehoben werden. Das heißt, die Beschreibung des Fehlerverhaltens beinhaltet Wissen über das modellierte Protokoll.

Des Weiteren muss die Fehlerinjektion das spezifizierte Fehlerverhalten automatisiert interpretieren können. Die Interpretation ermöglicht eine automatisierte An-

steuerung der Fehlerinjektoren durch die Fehlerspezifikation. Im Folgenden wird im ersten Abschnitt das Format der Fehlerspezifikation vorgestellt. Der zweite Abschnitt stellt die Infrastruktur zur automatisierten Interpretation des Fehlerverhaltens vor und zeigt die Interaktion mit den vorgestellten Fehlerinjektoren.

5.3.1 Fehlerverhaltensspezifikation

In dieser Arbeit wird ein Ansatz zur funktionalen Fehlerspezifikation basierend auf TA angewandt. Wie im Abschnitt 3.4 vorgestellt, verwenden heutige Ansätze meist vordefinierte Fehlermuster, die lediglich zur Laufzeit durch ein Kontrollsignal bzw. eine Zeitabhängigkeit aktiviert werden. Bei abstrakten Simulationsmodellen tritt häufig der Fall ein, dass das Auslöseereignis bzw. das injizierte Verhalten sich aus dem Systemzustand herleitet. Bei vordefinierten Fehlermustern müssten somit unterschiedliche Systemzustände berücksichtigt werden. Das Fehlerverhalten, in abstrakten Systemmodellen, ist stark vom Anwendungsfall geprägt und schwer generalisierbar, wie es z. B. auf Gatterebene mit dem Haftfehlermodell möglich ist. Dies führt dazu, dass die Anzahl an vorzuhaltenden Fehlermuster nicht mit den abstrakten Anwendungsfällen skaliert.

Aus diesem Grund verwendet der vorgestellt Ansatz eine eigens definierte zustandsbasierte Beschreibung, die zur Spezifikation des Fehlerverhaltens dient. Das sogenannte Gefahrenverhaltensmodell (engl. Behavioral Threat Model (BTM)) wird verwendet um das zu injizierende Fehlerverhalten zu spezifizieren.

Definition 5.1

Gefahrenverhaltensmodell (engl. Behavioral Threat Model)
Formell wird das BTM als Sechstupel $< L, l_0, \Sigma, C, S, E >$ *beschrieben, welches aus folgenden Teilen besteht:*

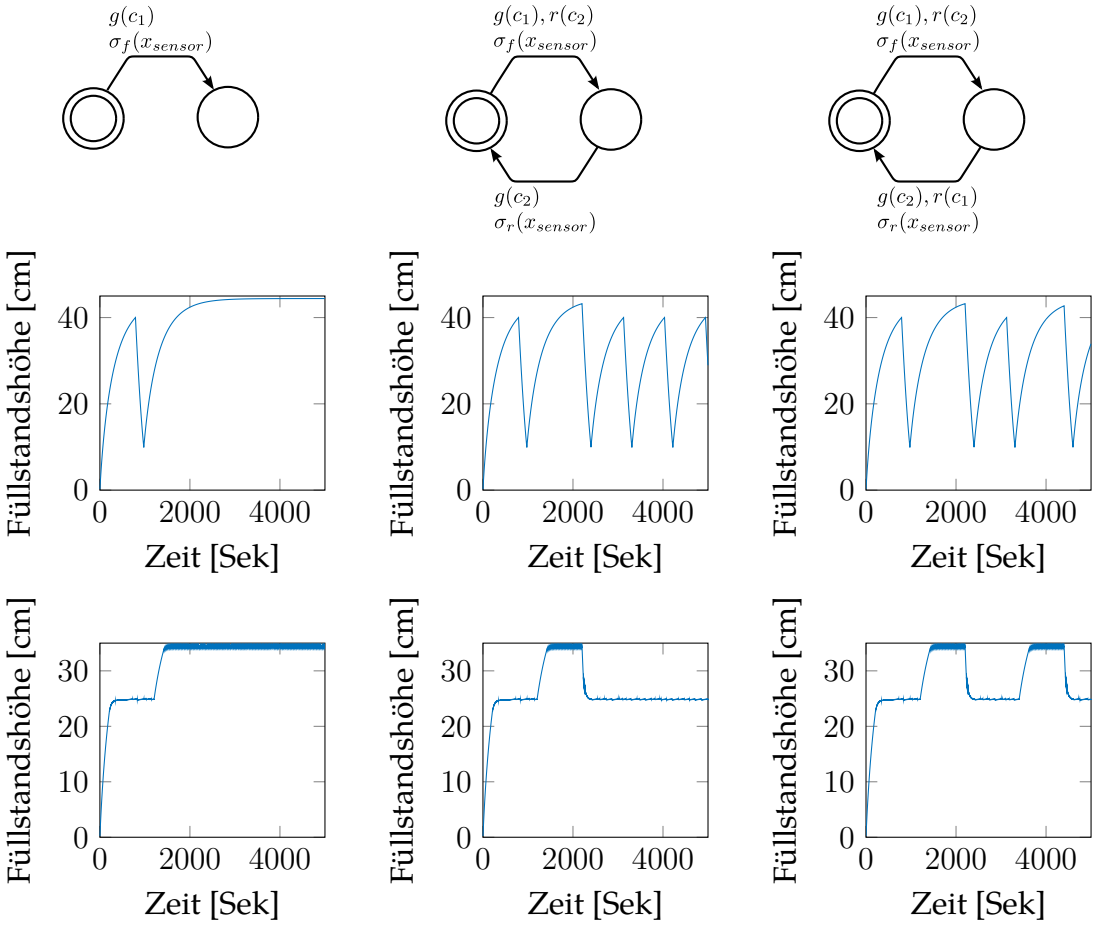
- L *ist eine endliche Menge von Zuständen.*
- l_0 *ist der initiale Zustand.*
- Σ *bezeichnet das Aktionsalphabet.*
- C *ist eine endliche Menge von Zeitgebern.*
- S *ist eine Menge von Signalen.*
- $E \subseteq (L \times \Sigma(\mathbb{X}, \mathbb{X}_{btm}) \times G(C, \mathbb{X}, \mathbb{X}_{btm}, P, S) \times R(C) \times N(S) \times L)$ *spezifiziert Übergänge. Hierbei ist:*
  - $G(C, \mathbb{X}, \mathbb{X}_{btm}, P, S)$ *eine Menge von boolesche Übergangsbedingungen,*
  - $R(C)$ *eine Menge von Funktionen zum Zurücksetzen von Zeitgebern und*
  - $N(S)$ *eine Menge von Signalisierungsfunktionen.*

Das Aktionsalphabet wird verwendet, um eine Fehlerinjektion auszulösen oder den Fehler zu beheben. Technisch gesehen bedingt eine Aktion somit die Ansteuerung der Fehlerinjektoren aus dem vorherigen Abschnitt. Formal erweitert sich eine Aktion somit zu $\sigma(\mathbb{X}) \in A$, da sie den Simulationszustand ändert. Bedingungen im TA sind alleinig auf der Menge von Zeitgebern definiert. Zur Beschreibung von Fehlern muss der Spezifikationsumfang von Bedingungen auf werteabhängige Bedingungen erweitert werden. Werte- und zeitabhängige Bedingungen $g(C, \mathbb{X})$ spezifizieren die Übergänge zwischen Fehlerzuständen, z. B. die schrittweise Verstärkung einer Fehlerursache über die Zeit. Hierbei können Bedingungen sowohl von den Zeitgebern als auch von dem Simulationszustand $\mathbb{X}$ abhängen. Hierbei wird mittels der Wertesonden aus dem vorherigen Abschnitt auf den Simulationszustand zugegriffen. Werteabhängige Bedingungen beschreiben Fehler, die z. B. von Ereignissen in der Simulation oder von dem Überschreiten eines Schwellwertes, in der Simulation, abhängen. Um ein Zeitverhalten bei der Fehlerinjektion zu berücksichtigen, verwendet das BTM die Zeitgeber des TAs, die synchron voranschreiten aber asynchron zurückgesetzt werden können. Durch die zurücksetzbaren Zeitgeber lassen sich komplexe Zeitbedingungen spezifizieren, die von absoluten oder relativen Zeitpunkten abhängen können.

Bei der Spezifikation des Fehlerverhaltens spielt die Modellierung von Wahrscheinlichkeiten eine entscheidende Rolle. Im Speziellen bei einer nicht vollständigen Verifikation basierend auf Simulationen. Dies führt zu einer weiteren Erweiterung der Bedingungen $g(C, \mathbb{X}, P)$. Beim Übergang können auch Wahrscheinlichkeiten sowie Wahrscheinlichkeitsverteilungen berücksichtigt werden.

Um die Komplexität einzelner BTMs gering zu halten, ist es möglich, das Fehlerverhalten mittels mehreren BTMs zu spezifizieren. Ein Signalisierungsmechanismus gewährleistet die synchronisierte Abarbeitung der einzelnen Automaten. Dies ist vor allem unter Berücksichtigung wahrscheinlichkeitsbedingter Übergänge wichtig, bei denen ein nichtdeterministisches Ereignis einen Zustandswechsel auslöst. Aus diesem Grund werden die Bedingungen für Zustandsübergänge um Signalisierungsereignisse erweitert $g(C, \mathbb{X}, P, S)$. Die Signalisierung von Ereignissen $n(S)$ erfolgt analog zu den Aktionen $\sigma(\mathbb{X})$ an den Zustandsübergängen. Zur Vereinfachung der BTMs werden eine Menge von lokalen Variablen $\mathbb{X}_{btm}$ verwendet, d. h. Variablen, die nicht Bestandteil des Simulationszustands $\mathbb{X}$ sind. Diese erlauben das Vorhalten von lokalen Informationen ohne das Aufblähen des Zustandsraums. Die Aktionen $\sigma(\mathbb{X}, \mathbb{X}_{btm})$ setzen die lokale Variablen. Zustandsübergangsbedingungen $g(C, \mathbb{X}, \mathbb{X}_{btm}, P, S)$ hingeben, greifen auf die lokalen Variablen zu.

Die zustandsbasierte Spezifikation ist bestens zur Beschreibung von Fehlern geeignet. Die Unterscheidung zwischen transienten, permanenten als auch intermittierenden Fehlern ist einfach zu spezifizieren. Die drei Fehlerarten, mit dem beobachteten Fehlerverhalten sowie die zugehörigen BTMs, sind in Abbildung 5.21 dargestellt. Hierbei werden die Effekte am Beispiel des Wassertanks, mit Steueralgorithmen die auf einer Hysterese-Regelung als auch auf einer PI-Regelung aufbauen, verdeutlicht. Die verwendete Fehlerinjektion ($\sigma_f(x_{sensor})$) führt dazu, dass die Regelung den Füllstand auf ein höheres Niveau regelt. Bei permanenten Fehlern wird nach der Fehler-



(a) Permanenter Fehler (b) Transienter Fehler (c) Intermittierender Fehler

Abbildung 5.21: BTMs und die zugehörigen Fehlereffekte auf eine Füllstandsregelung: Mit einer Hysterese-Regelung (obere Zeile) und einer PI-Regelung (untere Zeile)

injektion, spezifiziert durch die Aktion $\sigma_f(x_{sensor})$, in einem finalen Zustand verharrt. Das heißt, es findet keine Aufhebung des Fehlers statt. Bei transienten Fehlern existiert eine Transition aus dem Fehlerzustand. Die ausgehende Transition hebt die Fehlerinjektion auf. Die Fehlerbehebung in Abbildung 5.21 wird durch die Aktion $\sigma_r(x_{sensor})$ spezifiziert. Des Weiteren handelt es sich um einen transienten Fehler fester Dauer. Beim Übergang in den Fehlerzustand wird der Zeitgeber c_2 zurückgesetzt. Das Aufheben des Fehlers ist alleinig von den Zeitgebern abhängig. Bei intermittierenden Fehlern wird ein Zyklus erzeugt, indem der Ausgangszustand im fehlerfreien Zustand wiederhergestellt wird. Dieses Verhalten wird durch das Zurücksetzen der Zeitgeber und somit das erneute Auslösen von zeitbedingten Transitionen realisiert. Im Beispiel der Abbildung 5.21 sind die ausgeführten Aktionen mit der Systemzustandsvariable x_{sensor} verknüpft. Diese modelliert den vom Sensor erfassten Füllstand im Wassertank.

Um Aktionen und Bedingungen im BTM zu spezifizieren, wird die Interpreter-Sprache *Python* verwendet. Vorteil einer Interpretersprache ist, dass die spezifizier-

ten Ausdrücke dynamisch zur Laufzeit interpretiert werden, im Vergleich zu einer Compiler-Sprache wie C++. Der Interpreteransatz ergänzt sich somit sehr gut mit der dynamischen Konfiguration der Fehlereffektsimulation. Die Sprache Python ermöglicht es, komplexe, mathematische Bedingungen und Aktionen zu definieren. Die Python-Semantik wird verwendet, um Vergleiche oder arithmetische und logische Ausdrücke zu spezifizieren. Neben den rudimentären arithmetischen und binären Operationen wie $+, -, /, *, \&$, usw. sind auch komplexe Ausdrücke über temporäre Datenstrukturen oder Bibliotheksfunktionen möglich. Zum Beispiel bieten die Pseudo-Zufallszahlen-Generatoren von Python eine Alternative zu den Zufallszahlen des BTMs. Zu beachten ist lediglich, dass sich Aktionen zu einem einzigen Objekt auswerten lassen. Die Injektorsteuerung nutzt das erzeugte Objekt, das sogenannte Injektionsartefakt, zur Fehlerinjektion. Bedingungen sind Prädikate, die sich immer zu einem booleschen Objekt auswerten lassen. Das heißt, jede Bedingung lässt sich nach Wahr oder Falsch auswerten. Die genaue Auswertungslogik der Ausdrücke wird in Abschnitt 5.3.2 vorgestellt.

Um eine automatisierte Verarbeitung der BTMs sowie die Integrationsfähigkeit in den Konfigurationsansatz aus Abschnitt 5.1.3 zu gewährleisten, wird eine textuelle Repräsentation der BTMs verwendet. Hierzu wird eine an die XML angelegte Struktur verwendet, die lediglich auf schließende Tags verzichtet. Der Verzicht der schließenden Tags erhöht die Lesbarkeit der Ausdrücke. Die BTM-Sprache definiert auf der höchsten Abstraktionsebene entweder Zustände oder Übergänge. Ein Zustand besteht aus einem Namen mit dem vorangestellten Tag `<ESname>` sowie die Möglichkeit den Zustand als initialen Zustand zu spezifizieren. Dies erfolgt mit dem Tag `<EStype>`, gefolgt von dem Schlüsselwort `initial`, das den initialen Zustand spezifiziert. Ein Übergang wird immer mit dem Quell- (`<ETsrc>`) und Zielzustand (`<ETtgt>`) begonnen. Alle weiteren Tags sind optional und können in beliebiger Reihenfolge stehen. Der Tag `<ETguard>` leitet die Spezifikation der Übergangsbedingung ein. Wie motiviert, wird die Übergangsbedingung mithilfe eines Python-Ausdrucks spezifiziert. Analog erfolgt die Spezifikation der durchzuführenden Aktion mit dem Tag `<ETaction>`. Zur Spezifikation der verbleibenden Eigenschaften des BTMs, gibt es einen Tag zur Spezifikation der zurückzusetzenden Zeitgeber (`<ETupdate>`) sowie zur Synchronisation zwischen den BTMs (`<ETsync>`). Innerhalb von Aktionen und Bedingungen kann die Spezifikation über den Ausdruck `VPvar('varName').read()` auf den Wert der Variable `varName`, aus dem Simulationszustand $\mathbb{X}$, referenzieren. Mit der Anweisung `VPvar('varName').force(value)` werden die Fehlerinjektion getriggert und die Variable überschrieben. Diese Anweisung ist nur in Aktionen erlaubt. Die benötigte Funktionalität wird von den im vorherigen Abschnitt 5.2 vorgestellten Fehlerinjektoren bereitgestellt. Ähnlich verhält es sich mit der Auflösung der Fehlerinjektion über die Anweisung `VPvar('varName').release()`. Die Anweisung unterstützt die Freigabestrategien:

- `RESTORE_CV`: Wiederherstellung des zuletzt geschriebenen Werts,
- `RESTORE_OV`: Wiederherstellung des ursprünglichen Werts sowie
- `RELEASE`: Freigabe ohne Wiederherstellung eines Wertes.

Abbildung 5.13 zeigt das durch die Strategien realisierte Verhalten. Das Schlüsselwort `BTMvar('varName')` referenziert die lokalen Variablen $\mathbb{X}_{btm}$ des BTMs. Die Spezifikation bietet Funktionen zum Setzen `BTMvar('varName').set(value)` sowie zum Lesen `BTMvar('varName').read()` eines Wertes.

Um Fehlerverhalten auf Datenarrays zu unterstützen, wird ein Mechanismus zur Indizierung von Arrays in die Fehlerbeschreibung mit aufgenommen. Mit der Anweisung `VPvar('varName')[index].force(value)` wird ein Fehler in das Element mit dem Index `index` injiziert. Für den lesenden Zugriff auf die einzelnen Zeitgeber wird die Anweisung `BTMclock('timerID').read()` verwendet. Diese Anweisung wird verwendet, um zeitabhängige Übergänge im BTM zu modellieren. Die Anweisung `BTMclock('timerID').reset()` spezifiziert das Zurücksetzen eines Zeitgebers.

Im Folgenden wird anhand von Beispielen die Anwendung der BTMs vorgestellt. Abbildung 5.22 zeigt einen zeitgesteuerten, transienten Fehler. Die Fehlerbe-

```
1 <ESname> errFree <EStype> initial
2 <ESname> errState1
3 <ETsrc> errFree <ETtgt> errState1
4    <ETguard> BTMclock('OKTime').read() == sc_time(525, sc_time_unit.SC_SEC)
5    <ETaction> VPvar('sensorValue').force(20.0)
6    <ETupdate> BTMclock('ERRTime').reset()
7 <ETsrc> errState1 <ETtgt> errFree
8    <ETguard> BTMclock('ERRTime').read() >= sc_time(200, sc_time_unit.SC_SEC)
9    <ETaction> VPvar('sensorValue').release()
```

Abbildung 5.22: Zeitgesteuerter, transienter Fehler

schreibung besteht aus zwei Zuständen mit jeweils einem Übergang von Zustand `errFree` nach `errState1` und umgekehrt. Der erste Übergang wird genommen, wenn der Zeitgeber `OKTime` einen Wert von 525 ns aufweist. Der Übergang löst die Injektion eines konstanten Fehlers mit dem Wert 20.0 aus sowie das Zurücksetzen des weiteren Zeitgebers `ERRTime`. Dieser Zeitgeber wird verwendet, um eine konstante Dauer der Injektion, d. h. des transienten Fehlers, zu erreichen. Auf der Rückkante wird die Injektion mit dem Ausdruck `VPvar('sensorValue').release()` aufgehoben, sobald der Zeitgeber `ERRTime` den Wert von 200 ns überschreitet. Durch Weglassen der Rückkante und somit der Behebung des Fehlers wird ein permanenter Fehler modelliert. Auf der anderen Seite wird durch ein Zurücksetzen des Zeitgebers `OKTime` ein immer wiederkehrender Fehler, d. h. ein intermittierender Fehler, modelliert. Dieses Vorgehen ist in Abbildung 5.21 skizziert. Um Fehler wie das Übersprechen von Signalen zu modellieren, ist es möglich, neben der Injektion eines konstanten Wertes auch ein Wert aus dem virtuellen Prototyp zu lesen. Der Ausdruck

```
VPvar('Res').force(not(VPvar('InA').read() or VPvar('InB').read()))
```

liest zwei Werte aus dem virtuellen System und verbindet sie über eine logische Operation um den zu injizierenden Wert zu bestimmen. Diese Spezifikation entspricht der typischen Mutation von Operatoren, die häufig zur Fehlerinjektion in Gatter-Modellen verwendet wird. Operatorersetzungen sind eine gängige Mutationstechnik [54]. Im vorgestellten Beispiel wird ein NAND Gate durch ein NOR Gate ersetzt,

indem der Ausgang des simulierten NAND Gates durch Injektion auf einen Wert in Abhängigkeit der beiden Eingänge gesetzt wird.

```
<ESname> errFree <EStype> initial
<ESname> errState1
<ETsrc> errFree <ETtgt> errState1
   <ETsync> syncVar?
   <ETupdate> BTMclock('ERRTime2').reset()
   <ETaction> VPvar('InjProbeReg').force(99)
<ETsrc> errState1 <ETtgt> errFree
   <ETguard> BTMclock('ERRTime2').read() >= sc_time(10, sc_time_unit.SC_MS)

<ESname> timeState <EStype> initial
<ETsrc> timeState <ETtgt> timeState
   <ETguard> BTMclock('OKTime').read() >= sc_time(600, sc_time_unit.SC_MS)
   <ETsync> syncVar!
   <ETupdate> BTMclock('OKTime').reset()
```

Abbildung 5.23: Synchronisierte Fehlerspezifikation

Abbildung 5.23 zeigt eine Fehlerspezifikation, die in zwei BTMs aufgeteilt ist. Die untere Spezifikation besteht aus einem einzigen Zustand, der im Abstand von 600 ms ein Ereignis auslöst (`<ETsync> syncVar!`). Das zweite BTM wartet auf das Synchronisationsereignis (`<ETsync> syncVar?`), um die eigentliche Fehlerinjektion vorzunehmen. Die Synchronisation ermöglicht ein Aufteilen eines komplexen Verhaltens auf mehrere Automaten.

Fehler treten meist in Datenfeldern auf, z. B. beim Übertragen von Bildern oder aber in Speicherbereichen wie Registerarrays oder Datenmemory. Abbildung 5.24 zeigt in der Aktion die Injektion eines konstanten Wertes auf mehrere Elemente, von 0 bis 150, eines Arrays. Hierbei wird die kompakte Schreibweise von Schleifen aus Python verwendet. Der Grundgedanke ist die Erzeugung von Listen durch eine interne Schleife. Im Beispiel besteht die Liste aus dem Rückgabewert des `force()`-Aufrufs, der über den Index 0 bis 150 generiert wird. Der Rückgabewert des `force()`-Aufrufs ist nebensächlich, sondern hier steht der Aufruf der Funktion im Vordergrund. Das Beispiel zeigt, dass zur Spezifikation komplexer Fehlerszenarien ein rudimentäres Wissen über die Python-Sprache vorausgesetzt wird.

```
<ESname> errFree <EStype> initial
<ESname> errState1
<ETsrc> errFree <ETtgt> errState1
   <ETguard> BTMclock('OKTime').read() == sc_time(90, sc_time_unit.SC_NS)
   <ETaction> [VPvar('Hsin')[id].force(1.0) for id in range(0,150)]
```

Abbildung 5.24: Fehlerspezifikation auf Datenbereichen

5.3.2 Interpretation und Steuerung

Neben dem Fehlerinjektor bildet die Injektionssteuerung den zweiten Hauptteil der Erweiterungen der Simulationsinfrastruktur zur Fehlerinjektion. Hierbei handelt es sich vorrangig um eine zentrale Komponente, dem sogenannten *Stressor*, der von einzelnen Fehlerinjektoren losgelöst ist, und die Steuerung aller Fehlerinjektoren übernimmt. Vorteil ist hierbei, dass die Steuerung nicht die Fehlerinjektoren-Templates

beeinflusst und die Fehlerinjektoren deswegen eine fokussierte Implementierung aufweisen. Ein weiterer Vorteil ist, dass eine Fehlerbeschreibung mehrere Fehlerinjektoren ansprechen kann ohne eine zusätzliche Synchronisation zwischen den Fehlerinjektoren, wie es bei einer verteilten Steuerung notwendig wäre.

Aufgabe des Stressors ist die Interpretation der BTMs sowie die Bereitstellung aller hierzu benötigten Informationen. Kern der Interpretation bildet ein Python-Interpreter, der die in den BTMs enthaltenen Bedingungen und Aktionen interpretiert bzw. ausführt. Spezielle Python-Erweiterungen ermöglichen den Zugriff auf die Systemsimulation, auf die lokalen Ressourcen des BTMs oder die Verwendung von Simulationsprimitiven in der Fehlerspezifikation. Hierzu hat der Interpreter, über eine wohldefinierte Schnittstelle, Zugriff auf den Simulationszustand, der durch Wertesonden und Fehlerinjektoren bereitgestellt wird. Neben dem Zugriff auf die Systemsimulation verwaltet der Stressor interne Ressourcen, um die, im vorherigen Kapitel motivierten, lokalen Informationen des BTMs zu verwalten. Abbildung 5.25 gibt einen Überblick über die Struktur des Stressors.

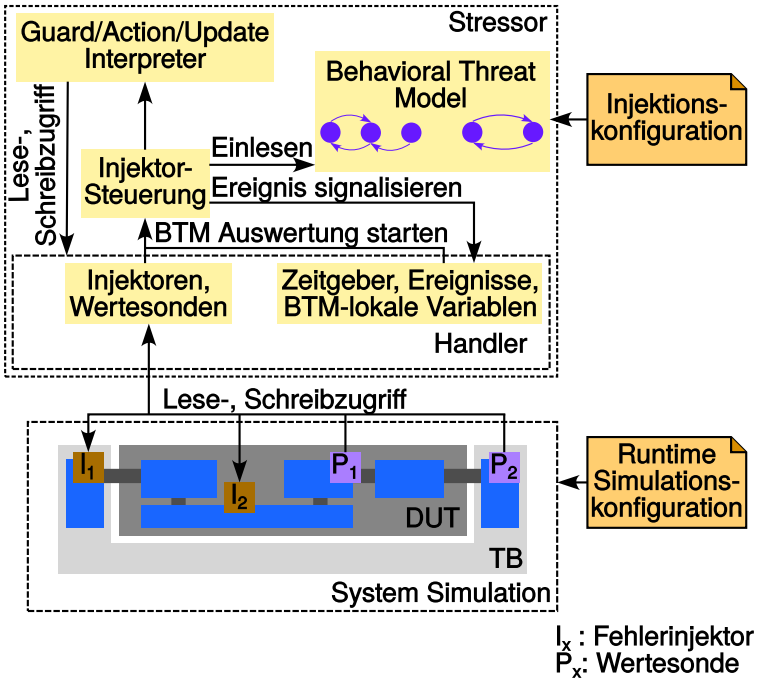


Abbildung 5.25: Detaillierte Struktur der Fehlerinjektionssteuerung

Bei Simulationsstart lädt der Stressor die BTM-Spezifikationen und überführt diese in eine interne Zustandstabelle. Die Tabelle beinhaltet Einträge für alle spezifizierten Zustandsübergänge, wobei die Zustandsnamen in ein numerisches Format aufgelöst werden. Der Stressor überführt alle Synchronisierungsanweisungen in SystemC-Events und fügt sie als `wait` und `notify` Listen zu jedem Übergang hinzu. Lediglich die Aktionen und Übergangsbedingungen übernimmt er unverändert in die Zustandstabelle.

Für den Zugriff auf die Fehlerinjektoren wird eine Lookup-Tabelle bereitgestellt, bei der jeder Fehlerinjektor mit einer eindeutigen ID hinterlegt ist. Hierzu wird bei der Instanziierung eines Fehlerinjektors eine Registrierungsroutine durchgeführt. Benötigt die Injektionssteuerung Zugriff auf den Systemzustand, wird die entsprechende Referenz, auf den Fehlerinjektor, aus der Lookup-Tabelle gelesen und anschließend auf den Fehlerinjektor zugegriffen. Um die Schnittstelle für die Injektionssteuerung homogen zu halten, stellt die Lookup-Tabelle sowohl Zugriffe auf Zeitgeber als auch auf lokale Variablen bereit. Für BTM-lokale Ressourcen werden beim erstmaligen Parsen der Aktionen und Zustandsübergänge Stellvertreter-Injektoren für `BTMvar` und Zeitgeber für `BTMclock` angelegt und registriert. Der Stressor verwaltet die lokalen Ressourcen intern. Für die Steuerung ist es demzufolge egal, ob sie auf lokale Ressourcen oder Simulationsressourcen zugreift.

Ein wichtiger Aspekt bei der Fehlerinjektionssteuerung ist, die Bestimmung des Zeitpunkts, wann eine Überprüfung der BTMs erfolgt. Durch die Überprüfung kann der Stressor eine neue Fehlerinjektion vornehmen bzw. eine bestehende Fehlerinjektion auflösen. Hierbei sind unterschiedliche Fälle zu unterscheiden. So muss der Stressor z. B. bei einer Werteänderung der Fehlerinjektoren oder Wertesonden die Zustandsübergangsbedingungen der BTMs auswerten. Das heißt, sobald ein Fehlerinjektor durch die Simulation geändert wird, wertet der Stressor die BTMs erneut aus. Die Überprüfung ist in Abbildung 5.14 in der letzten Anweisung dargestellt. Dies ist wichtig, da durch den geänderten Wert ein Zustandsübergang möglich wäre. Dieses Verhalten begründet auch die in Abschnitt 5.2.3 vorgestellten aktiven und passiven Wertesonden. Bei aktiven Wertesonden wird eine BTM-Auswertung ausgelöst. Sie sind somit geeignet, innerhalb von Übergangsbedingungen aufzutreten. Bei passiven Wertesonden wird keine BTM-Auswertung ausgelöst. Sie sind somit nur innerhalb von Aktionen sinnvoll. Mit dieser Unterscheidung ist es dem Anwender möglich, den Simulationsaufwand zu reduzieren, da passive Wertesonden weniger Vergleiche während der Simulation durchführen.

Eine weitere Auswertung ist notwendig, wenn die Simulationszeit voranschreitet, da zeitbedingte Zustandsübergänge ausgeführt werden könnten. Hierzu bietet die Injektionssteuerung einen eigenen Prozess, der auf dem kleinsten Time-out der Zeitgeber wartet. Hierzu wird bei jeder Überprüfung einer zeitbedingten Zustandsbedingung die zu wartende Zeit in eine sortierte Liste geschrieben. Der Timer-Prozess wartet nun auf die kleinste Warteperiode und garantiert somit, dass zu dem Zeitpunkt, an dem der Zustandsübergang möglich wäre, der Stressor die BTMs überprüft. Durch einen wertebedingten bzw. synchronen Zustandsübergang kann es vorkommen, dass dieser zeitbedingte Zustandsübergang nicht mehr möglich ist. Diese obsolete Überprüfung hat aber keine Auswirkungen auf das Fehlerverhalten.

Eine weitere Möglichkeit, einen Übergang im BTM auszulösen, sind Ereignisse. Hierbei wird durch einen Übergang in einem weiteren BTM ein Ereignis ausgelöst. Sobald ein Ereignis ausgelöst wird, wird die Überprüfung der restlichen BTMs ausgelöst. Alle Übergänge, die auf dieses Ereignis gewartet haben, werden genommen.

Bei der Auswertung der Aktionen und Übergangsbedingungen wird der integrierte Python-Interpreter verwendet. Wie vorgestellt, bestehen sowohl die Aktionen als

auch die Übergangsbedingungen aus Python-Ausdrücken. Die Python-Ausdrücke werden in eine dedizierte Struktur eingebunden und dem Python-Interpreter zur Ausführung übergeben. Die Struktur umfasst vorrangig die Einbindung unterschiedlicher Bibliotheken, wie z. B. die Erweiterungen zur Fehlerinjektion. Bei Übergangsbedingungen wird das Ergebnis des Ausdrucks einer Variable zugewiesen. Der Stressor verwendet den Wert der Variable, um auszuwerten, ob ein Zustandswechsel vorzunehmen ist.

Zur Bereitstellung einer Schnittstelle zwischen dem Python-Interpreter und der Fehlerinjektionssteuerung wird die Boost-Python-Schnittstelle verwendet. Diese ermöglicht es, C++-Objekte und -Klassen in Python verfügbar zu machen. Die Klasse `VPvar` ermöglicht den Zugriff auf die Fehlerinjektoren. Sie bietet Funktionen für den lesenden und schreibenden Zugriff auf die Fehlerinjektoren. Der Konstruktor der Klasse übernimmt die ID des Fehlerinjektors. Durch Aufruf der jeweiligen Funktion des C++-Fehlerinjektors, ist es dem Python-Interpreter möglich, auf Simulationsvariablen zuzugreifen. Hierbei bietet Python den Vorteil, dass es interne Daten als Python-Objekt vorhält. Dies bedeutet, dass die Problematik der unterschiedlichen Datentypen zur Fehlerinjektion an die Schnittstelle zwischen Stressor und Fehlerinjektor verlagert wird. Durch Bereitstellen der Konvertierungsfunktionen in den Fehlerinjektoren kann der Stressor ausschließlich die Datenstruktur des Python-Objekts (`PyObject`) verwenden.

Das gleiche Vorgehen wird bei den Zugriffen auf `BTMclock` und `BTMvar` angewandt. Die Implementierung berücksichtigt Sonderfälle wie verschachtelte Aufrufe. Zur Unterstützung von Array-Strukturen stellen die Klassen `VPvar` und `BTMvar` einen Indexierungsoperator bereit.

Neben dem Zugriff auf die lokalen Ressourcen und Simulationsressourcen stellt der Ansatz Erweiterungen zur Behandlung von SystemC spezifischen Primitiven bereit. Eine der wichtigsten Erweiterungen ist die Einbindung von `sc_time` in Python. Die Erweiterung stellt z. B. die SystemC-Aufzählung der Zeiteinheiten `SC_FS`, `...`, `SC_SEC` bereit. Somit ist es möglich, in Python direkt auf SystemC-Zeitprimitiven zuzugreifen. Abbildung 5.22 zeigt ein BTM mit zeitabhängigen Übergangsbedingungen. Eine weitere SystemC-spezifische Erweiterung ist das Einbinden der `tlm_generic_payload` Struktur. Durch die Bekanntgabe der Struktur ist es möglich, aus dem Python-Interpreter auf die einzelnen Strukturelemente zuzugreifen. Hierzu implementiert die Stellvertreterklasse die Zugriffsfunktionen auf die ursprüngliche Klasse `tlm_generic_payload`. Abbildung 5.26 zeigt die Bekanntgabe der Struktur für den Python-Interpreter. Hierbei wird jede Methode der C++-Klasse in Python bekannt gegeben. Die Bekanntgabe erfolgt unter Angabe einer ID bzw. des Funktionsnamens. Ein Python-Modul kapselt alle Methoden. Die entsprechende `import`-Anweisung lädt das Modul in den Python-Interpreter. Die C++-Schnittstelle, realisiert alle in dieser Arbeit beschriebenen Zugriffe von Python auf C++.

```
1  BOOST_PYTHON_MODULE(tlm)
2  {
3    boost::python::class_<tlm_generic_payload>("tlm_generic_payload")
4      .def("get_address", &tlm_generic_payload::get_address)
5      .def("set_address", &tlm_generic_payload::set_address)
6      .def("get_command", &tlm_generic_payload::get_command)
7      .def("set_command", &tlm_generic_payload::set_command)
8      .def("get_data", &tlm_generic_payload::get_data_structure)
9      .def("set_data", &tlm_generic_payload::set_data_structure)
10     .def("get_data_length", &tlm_generic_payload::get_data_length)
11     .def("set_data_length", &tlm_generic_payload::set_data_length)
12     .def("get_response_status", &tlm_generic_payload::get_response_status)
13     .def("set_response_status", &tlm_generic_payload::set_response_status);
14 }
```

Abbildung 5.26: Bekanntgabe von TLM-Primitiven in Python

Die Einbindung der `tlm_generic_payload` Struktur verdeutlicht die Wichtigkeit zur Einbindung zusammengesetzter Datenstrukturen. Bei der Verwendung von anwendungsspezifischen Datenstrukturen ist eine solche Bekanntgabe auch notwendig, um dem Python-Interpreter Zugriff auf die interne Struktur zu gewähren. Ein Beispiel ist die Bekanntgabe einer anwendungsspezifischen Struktur wie z. B. eines CAN-Rahmens. Um keine Änderungen am bestehenden Rahmenwerk vornehmen zu müssen, existiert eine Schnittstelle, die es dem Anwender ermöglicht, die Strukturdeklarationen zu registrieren. Das heißt, der Anwender stellt die Bekanntgabe wie in Abbildung 5.26 bereit und ruft lediglich die Registrierungsfunktion auf. Danach ist die Datenstruktur automatisch im Python-Interpreter und somit im BTM verfügbar.

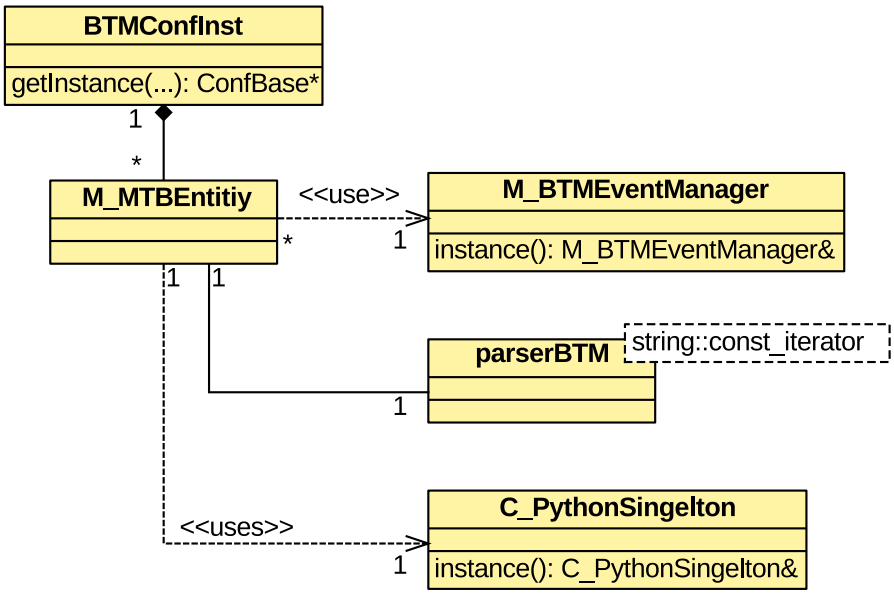


Abbildung 5.27: Klassendiagramm der BTM-Implementierung

Abbildung 5.27 zeigt die Struktur der Implementierung der Fehlersteuerung. Die Klasse `BTMConfInst` bildet die Wurzelklasse für die gesamte Steuerung der Fehlerin-

jektoren. Die Klasse ist als BSE spezifiziert. Sie wird der Simulationskonfiguration hinzugefügt und mit dem Fehlerverhalten parametrisiert. Die, durch den Konfigurationsparameter spezifizierten, Fehlerspezifikationen resultieren in je einem separaten BTM. Die Klasse `M_BTMEntity` repräsentiert die einzelnen BTMs. Aus diesem Grund besteht die `BTMConfInst` aus mehreren Instanzen vom Typ `M_BTMEntity`. Zur Speicherung des BTMs verwendet die Klasse `M_BTMEntity` eine Zustandstabelle. Um aus der textuellen Repräsentation des BTMs zu dieser Tabelle zu kommen, wird die Klasse `parseBTM` verwendet. Sie stellt hierfür die benötigte Parser-Funktionalität bereit. Die Klasse `BTMConfInst` übernimmt die Co-Simulation der BTMs mit der SystemC-Simulation. Hierzu ruft die Klasse die einzelnen Instanzen von `M_BTMEntity` auf, um den Status zu überprüfen und gegebenenfalls einen Zustandsübergang auszulösen. Für die Überprüfung der Bedingungen und der Ausführung der Aktionen nutzt die Klasse `M_BTMEntity` eine Referenz auf den Python-Interpreter. Für die Synchronisation zwischen den BTMs wird ein gemeinsamer Ereignis-Manager (`M_BTMEventManager`) verwendet. Der Ereignismanager ist als Singleton ausgelegt, was es den einzelnen Klassen vom Typ `M_BTMEntity` ermöglicht, sich einfach untereinander zu synchronisieren.

5.4 Fehlereffektbeobachtung

Für die Fehlereffektbeobachtung, bzw. für die Detektion von Fehlern bietet das Rahmenwerk zwei Erweiterungen. Bevor die folgenden Abschnitte die Ansätze im Detail vorstellen, werden die Möglichkeiten zur Verwendung der Fehlereffektsimulation in klassischen Verifikationsumgebungen betrachtet. Das vorgestellte Rahmenwerk versetzt den Anwender in die Lage, interne Zustände bestehender virtueller Prototypen mittels einer Fehlerspezifikation abzuändern. Durch Einsatz dieser Methode in der regulären Verifikationsumgebung ist es möglich, die Effekte von Fehlern auf das Gesamtsystem zu verifizieren. Regulären Verifikationsumgebungen stellen die Systemausgaben in Relation zu den Systemeingaben oder vergleichen sie gegen Erwartungswerte. Hier ist z. B. die UVM-Scoreboard-Technik zu nennen [1]. Durch die Erweiterung der Simulation durch die Fehlerinjektion wird der Eingaberaum der Simulation erweitert. Es ist möglich unterschiedliche Fehler zu parametrisieren. In diesem Szenario werden die Systemausgaben somit den Systemeingaben als auch der Fehlerparametrisierung gegenübergestellt. Durch dieses Vorgehen lassen sich z. B. Mechanismen wie *graceful degradation* auf ihre Wirksamkeit überprüfen. Eine weitere Methode ist die Anreicherung der Simulation mit Prüfbedingungen, sogenannten *Assertions*, welche die Verletzung bestimmter Bedingungen zur Simulationszeit melden [108]. Durch die Erweiterung mit der Fehlerinjektion, können diese Überprüfung nun im Kontext von Fehlereffektsimulationen angewandt werden und die Wirksamkeit der Fehlerkorrekturmechanismen überprüft werden. Da die Assertions meist vorkonfigurierte Systemeigenschaften überprüfen können diese nicht abhängig von der Fehlerinjektion konfiguriert werden. Dies bedeutet, dass durch den Ansatz nur nichtfehlerspezifische Invarianten überprüft werden können.

Bestehende SystemC-basierte Simulations- und Auswertewerkzeuge lassen sich mit diesen beiden Ansätzen verwenden. Durch die Kombination lässt sich das Systemverhalten, zusätzlich unter dem Einfluss von Fehlern verifizieren. Die Verifikation erfolgt bei beiden Ansätzen gegen zuvor bestimmtes Verhalten bzw. Invarianten. Das heißt, durch die Fehlerinjektion, ändert sich das Systemverhalten und stimmt somit nicht mehr mit der erwarteten Systemreaktion überein.

5.4.1 Erweiterung zur Fehlerdetektion

Im Folgenden sind zwei Analyseansätze vorgestellt, die sich speziell auf die Fehlereffektsimulation beziehen. Grundgedanke bei beiden Verfahren ist die Verwendung eines fehlerfreien Simulationslaufs als Referenzmodell für das erwartete Verhalten. Abbildung 5.28 verdeutlicht das Vorgehen. Durch dieses Vorgehen wird der Aspekt

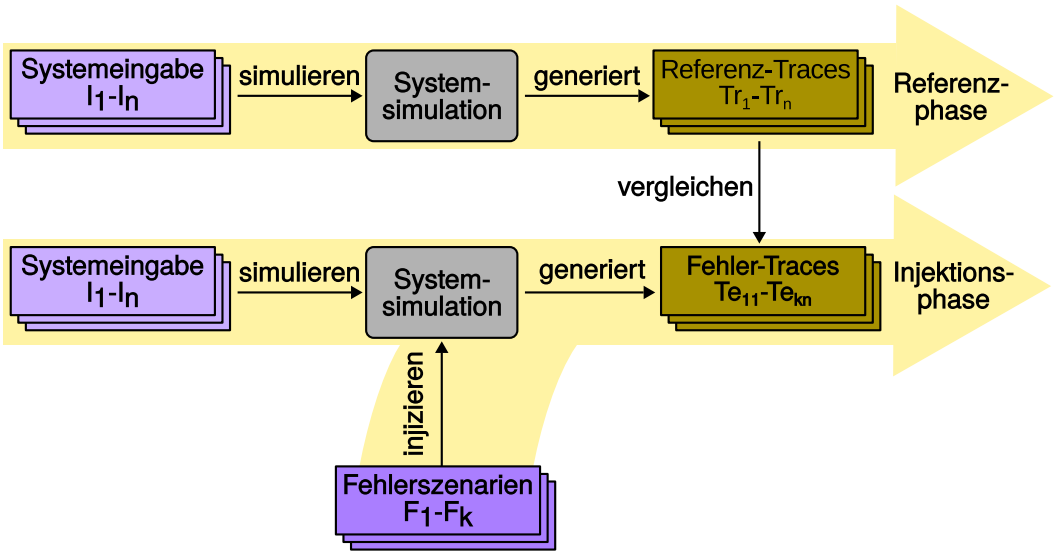


Abbildung 5.28: Analyseablauf mit Referenzvergleich

der Modellkorrektheit aus der Verifikation entfernt. Während bei Verwendung der herkömmlichen Verifikationsumgebung gegen ein spezifiziertes Verhalten oder ein Erwartungswert verifiziert wird, beschränken sich die beiden Ansätze auf den Vergleich mit einer vom Simulationsmodell generierten Referenz.

Bevor die Umsetzung der internen Fehlerdetektion und des werkzeuggestützten Detektionsverfahrens vorgestellt wird, eine Bemerkung zum Inhalt des Referenzverhaltens. Wie in Abschnitt 5.2.3 vorgestellt, bietet das Rahmenwerk zur Fehlereffektsimulation sogenannte Wertesonden, um den internen Zustand der Simulation zu beobachten. Die Auswertung der Fehlereffekte verwendet passive Wertesonden, die Mechanismen zur Trace-Erzeugung und Effektauswertung beinhalten. Die erweiterten Wertesonden werden als *Monitor* bezeichnet. Vorteil des Ansatzes ist, dass die Auswertung, im Vergleich zur alleinigen Beobachtung der Systemschnittstelle, auch interne Systemzustände betrachtet. Dies ist von Vorteil, wenn der Anwender die Ausbreitung des Fehlers analysiert. Durch die Betrachtung interner Zustände ist

es möglich, die Propagierung des Fehlers über die internen Komponenten hinweg zu detektieren. Ein weiterer Grund für die interne Fehlerdetektion ist die Maskierung von Fehlern. Bei der Maskierung kommt es zu keiner Auswirkung an den Systemgrenzen. Der Fehler wird innerhalb des Systems behoben bzw. hat keine relevante Auswirkung. Durch Einbeziehung des internen Zustands ist es möglich, Komponenten zu identifizieren, die den Fehler weiter propagieren oder den Fehler beheben. Ein weiterer Vorteil der Verwendung dedizierter Beobachtungspunkte ist, dass die Auswertung irrelevante Systemzustände nicht betrachtet. Dies ermöglicht die Konzentration auf die wesentlichen Systemzustände und führt dadurch zur Steigerung der Simulationsperformanz, im Vergleich zu einer ganzheitlichen Beobachtung des Simulationszustands. Hier liegt genauso ein Nachteil, da der Anwender vorab die Menge der wesentlichen Beobachtungspunkte bestimmen muss. Dieser Nachteil ist tolerierbar, da durch eine Überapproximation der Beobachtungspunkte die Auswertung immer die gewünschten Effekte beobachtet. Als Einstiegspunkt ist es möglich, die komplette Beobachtung nachzubilden. Bei Voranschreiten der Analysen kann der Anwender die Überapproximation schrittweise reduzieren und somit die Simulationsperformanz steigern.

Fehlerdetektion innerhalb SystemC/C++

Bei der Fehlerdetektion innerhalb von SystemC/C++ wird während der Fehlereffektsimulation das beobachtete Systemverhalten verifiziert. Hierbei wird ein zuvor durch die Monitore generiertes Referenzverhalten des Simulationsmodells geladen. Während der Simulation vergleichen die Monitore das aktuelle Systemverhalten gegen diese Referenz und protokollieren Abweichungen. Der Algorithmus bringt zuerst die Werteänderungen der Referenz und des beobachteten Systemverhaltens in eine zeitliche Ordnung. Bei einer Übereinstimmung der durch die Nominalsimulation generierten Referenz und dem Verhalten der fehlerbehafteten Simulation sind immer zwei Einträge für jeden Zeitpunkt vorhanden. Die geordnete Liste dient der Detektion von Abweichungen und damit zur Fehlerdetektion und -klassifikation. Abbildung 5.29 zeigt den verwendeten Algorithmus zur Fehlererkennung in Pseudocode. Die Klassifikation berücksichtigt nur die aktuell beobachteten Ereignisse (`entry.type == monitored`). Anhand des Vorgängerereignisses (`prev`) oder des nachfolgenden Ereignisses (`next`) werden die Abweichungen erkannt. Um eine zu frühe Werteänderung zu erkennen (Zeile 5), muss die nachfolgende Werteänderung im Referenz-Trace (Zeile 2) sich auf den gleichen Wert ändern (Zeile 3) aber zu einem späteren Zeitpunkt stattfinden (Zeile 4). Die Detektion von zu späten Wertänderungen oder falschen Wertänderungen erfolgt in ähnlicher Weise. Um die unregelmäßigen Fehler als sonstige Fälle erkennen zu können, detektiert der Algorithmus zusätzliche alle korrekten Werteänderungen (Zeile 14–17). Ein Sonderfall stellt das Ausbleiben von Werteänderungen in der aktuellen Simulation dar. Der unterstete `else`-Zweig des Algorithmus erkennt den Fall. Hierbei wird geprüft, dass keine zwei Referenzänderungen (`prev.type == reference`) direkt aufeinanderfolgen. Dies ist ein Zeichen dafür, dass im aktuellen Trace eine Werteänderung fehlt.

```
1  if( entry.type == monitored)
2    if( next.type == reference &&
3        next.value == entry.value &&
4        next.time > entry.time )
5      report(early)
6    else if( prev.type = reference &&
7             prev.value == entry.value &&
8             prev.time < entry.time )
9      report(late)
10   else if( prev.type = reference &&
11            prev.value != entry.value &&
12            prev.time == entry.time )
13     report(content)
14   else if( next.type == reference &&
15            next.value == entry.value &&
16            next.time == entry.time )
17     report(no_failure)
18   else
19     report(erratic_unintended)
20 else
21   if(prev.type = reference)
22     report(erratic_missing)
```

Abbildung 5.29: Algorithmus zur Fehlererkennung

Der Ansatz zum Laden eines beobachteten Verhaltens wird der Co-Simulation vorgezogen, da bei wiederholter Simulation von Fehlerinjektionen die Referenz nicht neu generiert werden muss. Der Anwender bestimmt, wann die Auswertung der aktuellen Werteänderungen erfolgt. Die Auswertung erfolgt mindestens am Ende der Simulation. Der Anwender kann über die Spezifikation einer Periode die Zeitpunkte, an denen eine Auswertung stattfindet, festlegen. Die Fehlerklassifikation wertet alle bis zu diesem Zeitpunkt angefallenen Werteänderungen aus. Hierdurch kann der Anwender regelmäßig die aktuelle Simulation überprüfen und bei Erkennen von Fehlern die Simulation beenden. Der Abbruch kann Simulationszeit sparen, wenn nur das Auftreten der ersten Abweichung interessant ist. Ein Problem bei diesem Vorgehen ist, dass zukünftige Werteänderungen die aktuelle Klassifikation verändern können. So kann es vorkommen, dass der Algorithmus eine zu frühe Werteänderung detektiert aber bei weiterer Ausführung der Simulation sich diese Fehlerklassifikation in einen unregelmäßigen Fehler abändern würde. Dies ist der Fall, wenn zwischen erster Änderung und Referenzänderung eine weitere Werteänderung auftritt. Abbildung 5.30 zeigt den Fall. Würde die Fehlerklassifikation zum Zeitpunkt t_3 erfolgen, würde eine zu frühe Werteänderung detektiert, da der Referenz-Trace komplett vorliegt. Durch die erneute Werteänderung zum Zeitpunkt t_4 müsste die Klassifikation aber einen unregelmäßigen Fehler erkennen. Dies kann sie aber erst zu einem späteren Zeitpunkt erkennen.

Ein Problem dieses Ansatzes ist, dass er keine anwendungsspezifischen Fehlerklassen detektiert. Hierzu müsste der Anwender den Algorithmus und somit des Simulationsrahmenwerks modifizieren. Um eine bessere Anpassbarkeit zu erzielen, wird im Folgenden eine erweiterbare Fehlerklassifikation vorgestellt.

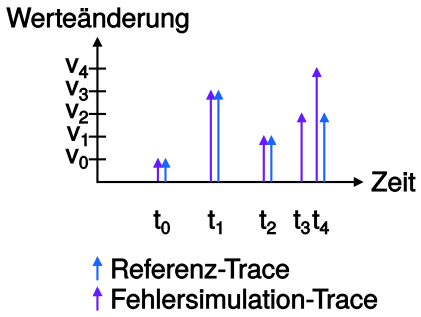


Abbildung 5.30: Kombinierte Werteänderungen

Fehlerdetektion mit UPPAAL

Bei der Fehlerdetektion mit UPPAAL erzeugt die Simulation eine Trace-Datei die sowohl das Verhalten der Nominalsimulation als auch der Fehlerinjektion beinhalten. Es existiert ähnlich zur Klassifikation mit SystemC/C++ eine Referenzphase, in der die Referenz generiert wird. Die Simulation speichert das Referenzverhalten zwischen und bei Durchführung der Fehlerinjektion wird die gemeinsame Trace-Datei erzeugt. Im Unterschied zur SystemC/C++-Klassifikation erfolgt der Vergleich durch ein separates Werkzeug. Zur Klassifikation der Abweichungen wird UPPAAL verwendet. Hierbei stellt die Simulation die Werteänderungen, die in Form eines Value Change Dump (VCD) Formats vorliegen, als eine Abfolge von Zuständen dar. Abbildung 5.31 zeigt auszugsweise die erzeugten Graphen mit den spezifizierten Werten. Es existiert ein Graph für das Voranschreiten der globalen Zeit sowie jeweils

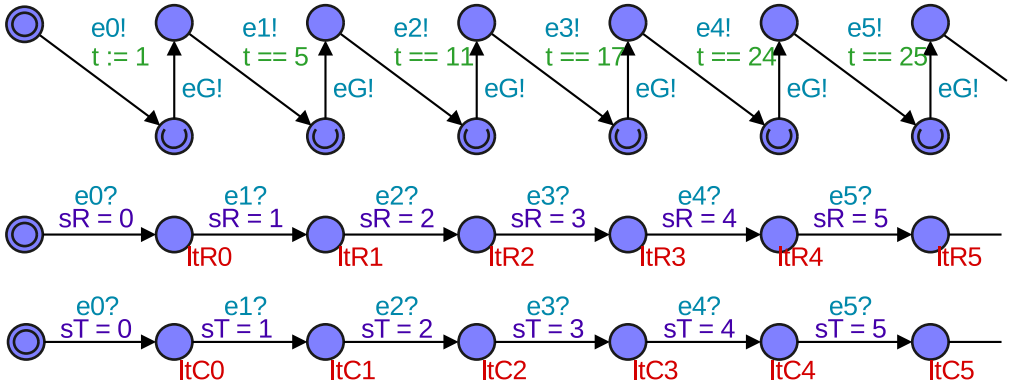


Abbildung 5.31: Darstellung der Traces in UPPAAL

ein Graph für die Werteänderungen der Referenz ($ltRi$) und des aktuellen Fehler-Traces ($ltCi$). Für jeden Zeitpunkt, an dem sich ein Wert ändert, wird ein Ereignis (ei) generiert. Dies betrifft sowohl den Referenz-Trace als auch den Trace der aktuellen Fehlersimulation. Das Ereignis wird verwendet, um einen Zustandswechsel in den Trace-Graphen hervorzurufen. Zum Beispiel löst das Ereignis $e1$ einen Zustandswechsel sowohl im Referenz-Trace als auch im aktuellen Fehler-Trace aus. Nach dem sich der Zustand in den Trace-Graphen geändert hat, wird zusätzlich ein globales Ereignis (eG) ausgelöst. Das Ereignis signalisiert, dass ein Zustandswechsel stattgefunden hat.

Lösen Ereignisse Transitionen aus, haben diese Vorrang vor anderen Transitionen. Transitionen, die durch ein Ereignis ausgelöst werden, haben Vorrang vor Transitionen ohne Bedingung. Die Vorrangregelung garantiert, dass das globale Ereignis erst nach den Wertewechseln ausgelöst wird. Um den temporären Zustand sofort zu verlassen und das globale Ereignis auszulösen, wird der Zustand als `urgent` spezifiziert. Der `urgent`-Zustand stellt sicher, dass sofort ein Zustandswechsel stattfindet. Vorteil des Vorgehens ist, dass Wertewechsel simultan in beiden Graphen erfolgen und erst, wenn beide Wechsel durchgeführt wurden, der globale Wertewechsel (eG) angezeigt wird. Das globale Wertewechselereignis wird verwendet, um den Klassifikationsgraphen zu triggern.

Mit dem in Abbildung 5.32 dargestellten Graphen werden beide Werteabfolgen verglichen und die auftretenden Fehler klassifiziert. Die Klassifikation mit UPPAAL

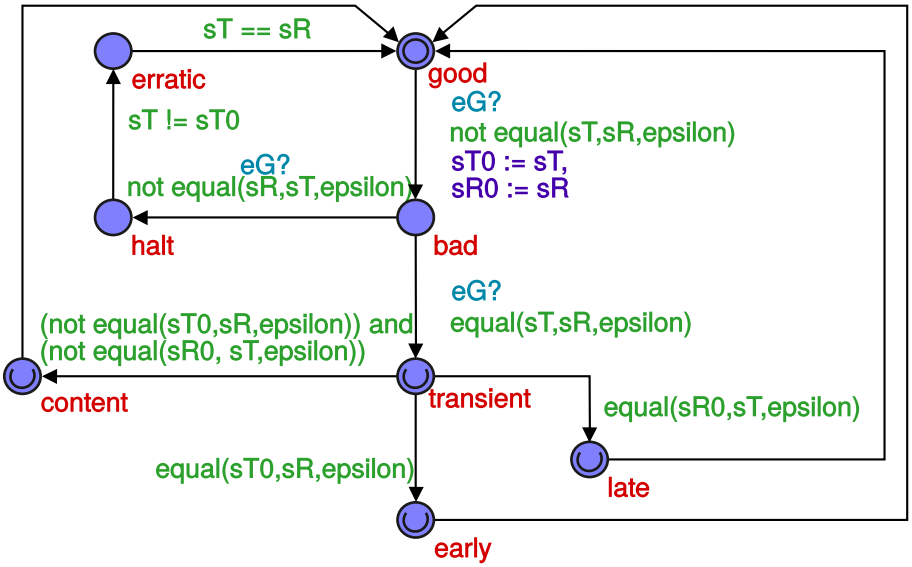


Abbildung 5.32: Fehlerklassifizierung in UPPAAL

verwendet die gleichen Fehlermodi wie die SystemC/C++-Klassifikation, die Abschnitt 5.4.1 vorstellt. Zum Wechsel der Zustände wird das globale Ereignis eines Zustandswechsels (eG) verwendet. Bei Auslösen des globalen Ereignisses wird der Klassifikationsautomat ausgewertet. Neben dem globalen Ereignis müssen für die Transitionen bestimmte Bedingungen vorliegen. Um aus dem fehlerfreien Zustand `good` in den fehlerbehafteten Zustand `bad` zu gelangen, müssen sich die Werte der beiden Traces unterscheiden. Dies wird durch die Bedingung `not equal(sT,sT, epsilon)` ausgedrückt. Die Funktion `equal()` vergleicht die beiden Traces. Hierbei ist `sT` der Wert des Fehler-Trace und `sR` der Wert des Referenz-Trace. Des Weiteren ist es möglich, einen Wert `epsilon` zu spezifizieren. Die Werteabweichung muss größer als der ϵ-Wert sein, damit die Klassifikation eine Abweichung detektiert. Dies ermöglicht das Ausblenden kleiner Schwankungen. In ähnlicher Weise lässt sich ein Zeitdelta angeben, das bei der Erstellung der Trace-Graphen berücksichtigt wird. Die Erstellung der Trace-Graphen filtert Änderungen, die kürzer als das angegebene Delta sind.

Wird im Fehlerklassifikationsgraphen die initiale Transition genommen, speichert dieser die aktuellen Werte der Traces in zwei lokalen Variablen. Dies ist für die spätere Fehlerklassifikation notwendig. Besitzen beide Traces mit der nachfolgenden Werteänderung erneut den gleichen Wert, wird ein transienter Fehler erkannt. Der Automat klassifiziert transiente Fehler entweder als Inhaltsfehler oder Zeitfehler. Bei Zeitfehlern wird zwischen zu späte bzw. zu frühe Wertwechsel unterschieden.

Der Ansatz verwendet CTL-Ausdrücke, um das Auftreten eines Fehlers zu detektieren. Hierbei prüft der CTL-Ausdruck, das Erreichen bzw. auch das nicht Erreichen eines Zustands im Fehlerklassifikationsgraphen. Abbildung 5.33 zeigt die verwendeten Anfragen zur Fehlerklassifikation. Die ersten vier Zeilen detektieren die Fehler-

```
1 E<> failureClassInst.late
2 E<> failureClassInst.early
3 E<> failureClassInst.erratic
4 E<> failureClassInst.content
5 A[] not failureClassInst.bad
```

Abbildung 5.33: UPPAAL-Queries zur Fehlerklassifikation

klassen der beobachteten Fehlerauswirkung. Die letzte Query gibt allgemein die Abweichung des Fehler-Trace vom Referenz-Trace an. Sobald der `Bad`-Zustand erreicht wurde, ist ein Fehler aufgetreten und der Zeitpunkt ist auch der Auftrittszeitpunkt des Fehlers. Da die Klassifikation erst später erfolgen kann, ist der Klassifikationszeitpunkt nur eine obere Schranke des Fehlerauftrittszeitpunkts. Zusätzlich wird eine Zustandsabfolge generiert, die es dem Anwender ermöglicht bis zu dem eigentlichen Fehlerzustand, durch die Wertewechsel zu laufen. Das Gegenbeispiel versetzen den Anwender in die Lage, den genauen Pfad zum Fehlerzustand nachzuverfolgen.

In der Spezifikation der Queries liegt der Vorteil der UPPAAL-Analyse. Der Anwender kann anwendungsspezifische Abfragen spezifizieren, um die korrekte Funktionalität des Systems nachzuweisen. Hierbei sind beliebige LTL-Ausdrücke, die sich auf die generierten Graphen beziehen, erlaubt. Wird bei der Regelung des Wassertanks ein Beobachtungspunk auf das aktuelle Wasserniveau gesetzt, kann der Anwender mit der Query `A[] sT <= 40` überprüfen, dass der Wasserspiegel nie über 40 steigt. Die Variable `sT` bezieht sich auf den Wert des aktuellen Traces und mit `A[]` wird überprüft, dass die Bedingung auf allen Werten des Trace gültig ist. Durch diesen Ansatz kann der Benutzer verschiedene anwendungsspezifische Anforderungen aus der Spezifikation überprüfen. Um komplexere Bedingungen zu überprüfen, ist es möglich, zusätzliche applikationsspezifische Klassifikationsgraphen zu spezifizieren. Ähnlich der generellen Fehlerklassifizierung stehen alle Eigenschaften der Traces zur Verfügung.

Die UPPAAL-Detektion ist in das Simulationsframework integriert. Das Simulationsframework erzeugt die benötigten Dateien automatisch und löst die UPPAAL-Analyse aus. Eine Zusammenfassung des Analyseergebnisses wird in das reguläre Simulationsergebnis aufgenommen. Die Zusammenfassung dokumentiert die Fehlerklasse sowie den Zeitpunkt des Fehlerauftritts. Alle erzeugten Dateien sowie die ge-

nerierten Gegenbeispiele stehen dem Anwender für eine nachträgliche anwendungsspezifische Analyse zur Verfügung.

Kapitel 6

Grafische Notation der Fehlereffektsimulation

In diesem Kapitel wird ein erweiternder Ansatz zur grafischen Darstellung der Fehlereffektsimulation vorgestellt. Der Ansatz ermöglicht zum einen eine vereinfachtere Integration in modellgetriebene Entwurfsabläufe zum anderen unterstützt er den Anwender und reduziert den Analyseaufwand. Durch die grafische Unterstützung wird der Spezifikationsprozess erleichtert und gleichzeitig die durchzuführenden Analysen dokumentiert. Wie in Abschnitt 2.5 dargelegt wird ein zentrales Modell angestrebt, in das alle Anforderungen und Entwurfsentscheidungen einfließen und das als Grundlage zur automatischen bzw. manuellen Ableitung der Implementierung dient. Abbildung 6.1 zeigt den angestrebten Ansatz. Grundlage des in dieser Arbeit

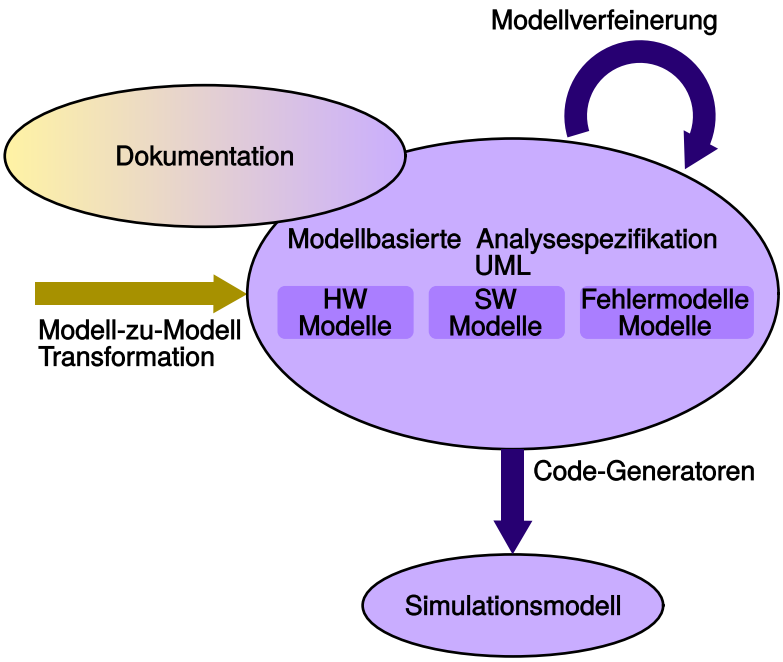


Abbildung 6.1: Modellzentrischer Ansatz zur Fehlereffektbewertung

verfolgten Ansatzes bildet die UML. Als Modellierungsumgebung wird das Eclipse Modeling Framework (EMF) mit dem Plug-in Papyrus [43] verwendet. Dieses stellt einen grafischen Editor für die UML-Diagramme zur Verfügung. Die im Rahmen dieser Arbeit entstandenen Eclipse-Plug-ins stellen die benötigten UML-Profilerweiterungen, Editor-Erweiterungen und Codegenerierungen bereit. Teilaspekte wurden vom Autor bereits in den Veröffentlichungen [18, 98] publiziert.

Modellgetriebener Analyseablauf

Bevor die Modellierung der einzelnen Bestandteile im Detail vorgestellt wird, erfolgt eine Eingliederung der Modellierung in den angestrebten Analyseablauf. Um den Aufwand der Fehlereffektsimulation zu reduzieren, wird der strukturelle Anteil der Simulationskomponentenbibliothek aus einer grafischen Beschreibung generiert. Mittels Techniken aus der Versionsverwaltung, im Speziellen die Merge-Operation, wird die gleichzeitige Entwicklung des strukturellen Anteils mittels grafischer Werkzeugumgebung und die funktionale Implementierung mit traditionellen Entwicklungsumgebungen unterstützt. Abbildung 6.2 verdeutlicht dies indem die SystemC/C++-Simulationsplattform sowohl von der grafischen Werkzeugumgebung, als auch durch eine funktionelle Spezifikation erzeugt wird. Die kooperative Entwicklung der BSEs

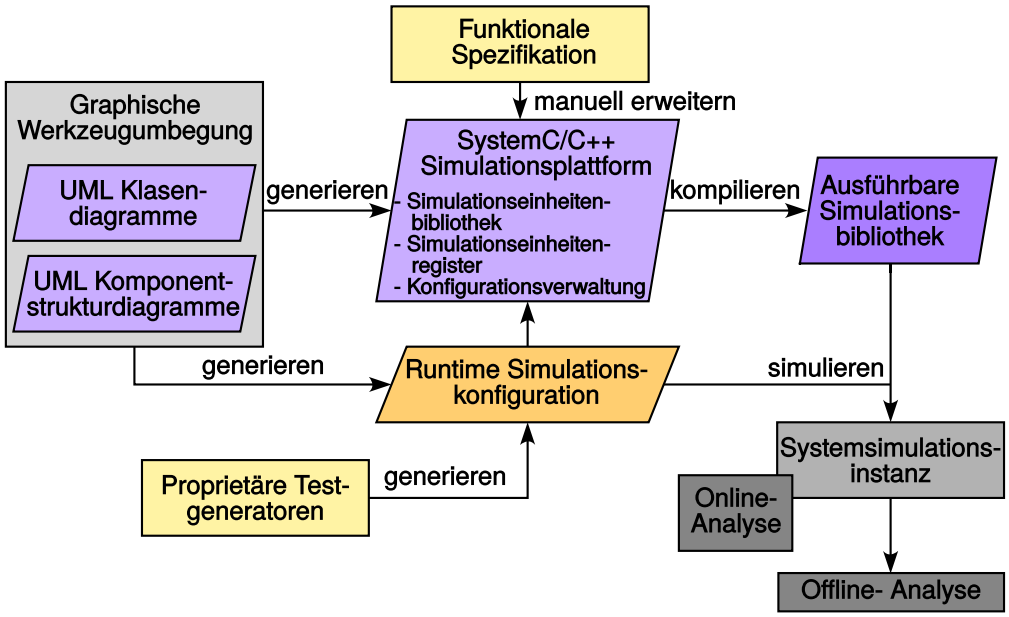


Abbildung 6.2: Analyseablauf im modellgetriebenen Entwurf

ist besonders zur Einbeziehung weitere Codegenerierungswerkzeuge wichtig. Vor allem bei der Entwicklung datenflussorientierter Algorithmen kommt häufig automatisch generierter Code aus Werkzeugen wie MATLAB/Simulink zum Einsatz, wie von Kuroki et al. in [64] motiviert. Die reine Fokussierung auf die UML-Spezifikation zur Erzeugung, der BSE, würde diese Ansätze ausschließen. Ein weiterer Aspekt bei der grafischen Spezifikation der Fehlereffektsimulation ist die Generierung der zur Laufzeit interpretierten Konfiguration, die mittels Kompositionsstruk-

turdiagrammen grafisch spezifiziert und automatisiert in die Konfigurationsdatei überführt wird. Zur Simulationslaufzeit wird diese Konfigurationsdatei vom Simulationsmodell interpretiert und eine Simulationsinstanz erzeugt. Abbildung 6.2 zeigt das Zusammenspiel der einzelnen Bestandteile. Die Kombination der kompilierten Simulationseinheitenbibliothek und die zur Laufzeit interpretierte Konfiguration erzeugt die Systemsimulation. Das Konzept ist ausführlich in Abschnitt 5.1 vorgestellt.

Neben der Reduzierung des Implementierungsaufwands bietet eine zentrale modellbasierte Spezifikation der Fehlereffektsimulation einen weiteren Vorteil. In heutigen Entwurfsprozessen finden modellbasierte Ansätze breite Anwendung. So existiert z. B. das UML-Profil MARTE, zur Modellierung von Echtzeitsystemen. Ein weiteres Profil, das vorrangig in der Automobilindustrie eingesetzte wird, ist EAST-ADL. Mittels Model-zu-Model-Transformationen ist es möglich, spezifizierte Systemteile in das Spezifikationsmodell der Fehlereffektsimulation zu überführen. Abbildung 6.3 verdeutlicht das angedachte Konzept. Dadurch, dass die genannten Spezifikations-

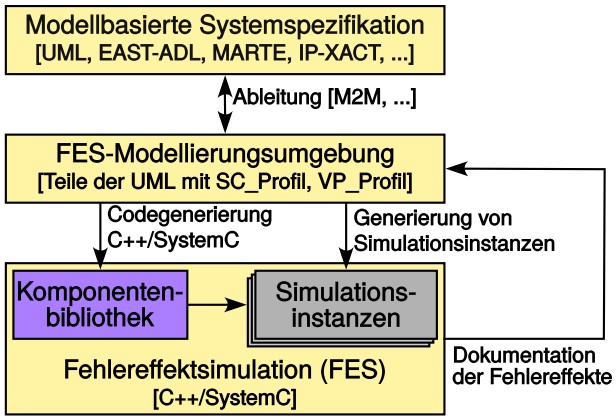


Abbildung 6.3: UML-Erweiterungen für die Spezifikationsmodelle

modelle allesamt UML-basiert sind, ist eine einfache Transformation zwischen den Modellen möglich. Hierzu kann z. B. die Query/View/Transformation (QVT)-Sprache verwendet werden [84].

6.1 Spezifikation der Simulationseinheitenbibliothek

Grundlegender Bestandteil der Fehlereffektsimulation ist die Simulationseinheitenbibliothek (SEB). Die grafische Spezifikation unterstützen den Anwender bei der Implementierung der SEB. Bevor die eigentliche Fehlereffektsimulation durchgeführt werden kann, muss die SEB spezifiziert und implementiert werden. Die in Kapitel 5.1 vorgestellten Mechanismen des Simulationsrahmenwerks erhöhen die Wiederverwendbarkeit bestehender Simulationsmodelle und erleichtern somit die Analyse unterschiedlicher Systeme. Zur Implementierung der einzelnen BSEs wird der Aufwand aber erhöht. Die Simulationseinheit muss z. B. eine Fabrikmethode zur dynamischen Erzeugung der Instanz bereitstellen. Zusätzlich muss der Anwender Methoden zum

Einlesen der Konfigurationsdatei und damit verbunden zum dynamischen Zuweisen von Parameterwerten und zum Verknüpfen der Instanzen implementieren. Der Mehraufwand liegt in der Implementierung von wiederkehrenden Codestrukturen. Codegenerierung ist bestens geeignet um solche sich wiederholende Muster automatisiert zu erzeugen. Zum einen wird die Codestruktur der Simulationseinheiten generiert. Dies umfasst die Klassen- bzw. Moduldeklaration einschließlich aller Membervariablen und Funktionsrümpfe. Zum anderen erzeugt die Codegenerierung die benötigten Fabrikmethoden und die Methoden zur Verknüpfung der BSEs. Die Codegenerierungsschritte reduzieren den Aufwand für den Anwender.

UML-Klassendiagramme spezifizieren die SEB. Jede BSE wird durch eine Klasse repräsentiert. Zur Spezifikation der Klassen stehen die bekannten Möglichkeiten der UML zur Verfügung, wie z. B. Attribute, Operationen, Generalisierung, Assoziationen oder deren Sichtbarkeit. Die folgende Betrachtung legt Schwerpunkt auf die UML-Konzepte, welche die Codegenerierung berücksichtigt. Jede Klasse spezifiziert unterschiedliche Attribute. Für jedes Attribut werden der Datentyp sowie die Sichtbarkeit (`private`, `protected`, `public`) angegeben. Des Weiteren spezifiziert die UML alle Operationen einer Klasse, mit ihren Parametern und dem Rückgabewert. Für Parameter und Rückgabewert kommen die gleichen Eigenschaften wie bei den Attributen zu trage. Zur nahtlosen Integration der Fehlereffektsimulation stellt die Arbeit drei Erweiterungen der UML bereit.

SystemC-Erweiterung

Ein entwickeltes Profil (SC_Profile) definiert Stereotypen, die sich auf Modellierungsprimitiven aus SystemC beziehen. Die UML bietet generelle Mechanismen zur Spezifikation des Quellcodes, wie z. B. die Schlüsselwörter `public` oder `private`. Für die dedizierten Modellierungsprimitiven von SystemC muss die UML erweitert werden.

Das Profil erlaubt eigene Klassen als `sc_module`, `sc_channel` zu spezifizieren. Die UML-Klassendiagramme spezifizieren die Operationen der Klasse. Hierbei wird über das UML-Profil SC_Profile die Möglichkeit geschaffen Operationen als `sc_thread`, `sc_method` oder `sc_cthread` zu spezifizieren. Des Weiteren wird zwischen normalen C++ Klassen und SystemC-Module unterschieden. Mit diesen Stereotypen ist es möglich, die korrekte Vererbungsbeziehung der Schnittstellen sowie die Thread-Deklarationen automatisch zu generieren.

Bei der Generierung der SEB wird lediglich der strukturelle Code generiert. Die Implementierung des applikationsspezifischen Verhaltens von Funktionen ist weiterhin Aufgabe des Anwenders. Die Codegenerierung erzeugt hingegen die funktionalen Implementierungen der Fabrikmethode und der Methode zur Verknüpfung von Simulationseinheiten.

Simulationsrahmenwerk-Erweiterung

Eine weitere Erweiterung stellt das UML-Profil VP_Profile dar. Es definiert Stereotypen für die dynamische Konfiguration des virtuellen Prototyps. Das Profil stellt Stereotypen bereit, um zwischen normalen Klassenattributen und durch die in der Konfigurationsdatei konfigurierten Attributen zu unterscheiden. Wird ein Attribut

als konfigurierbar markiert, wird es mit dem Stereotyp `vp_property` assoziiert. Dieser Stereotyp gibt einen global eindeutigen Bezeichner für das Attribut an. Der Bezeichner wird zur Identifikation innerhalb der Konfigurationsdatei verwendet. Die dynamische Konfiguration der Simulation bezieht sich neben den Attributen auch auf Assoziationen. Das Profil stellt für Assoziationen zwischen Klassen die Stereotypen `vp_link`, `vp_port` oder `vp_socket` bereit. Sie legen fest, ob Assoziationen eine direkte Klassenreferenz, einen SystemC-Port oder ein SystemC-TLM-Socket spezifizieren. Die Unterscheidung ist wichtig bei der Generierung der dynamischen Verlinkung, da die drei Klassen unterschiedliche Verknüpfungsmethoden benötigen. Die Verknüpfung von Sockets erfolgt durch Zuweisen, von Ports durch Binden und bei Links durch Vorhalten eines Zeigers auf die Klasse. Durch diese Informationen kann die Codegenerierung sowohl die Fabrikmethode als auch die Methode zur Verknüpfung der Simulationseinheiten automatisch erzeugen. In Abbildung 6.4 ist die Spezifikation der BSEs für das Wassertankbeispiel dargestellt. Die BSE `M_WaterTankPModel` wird

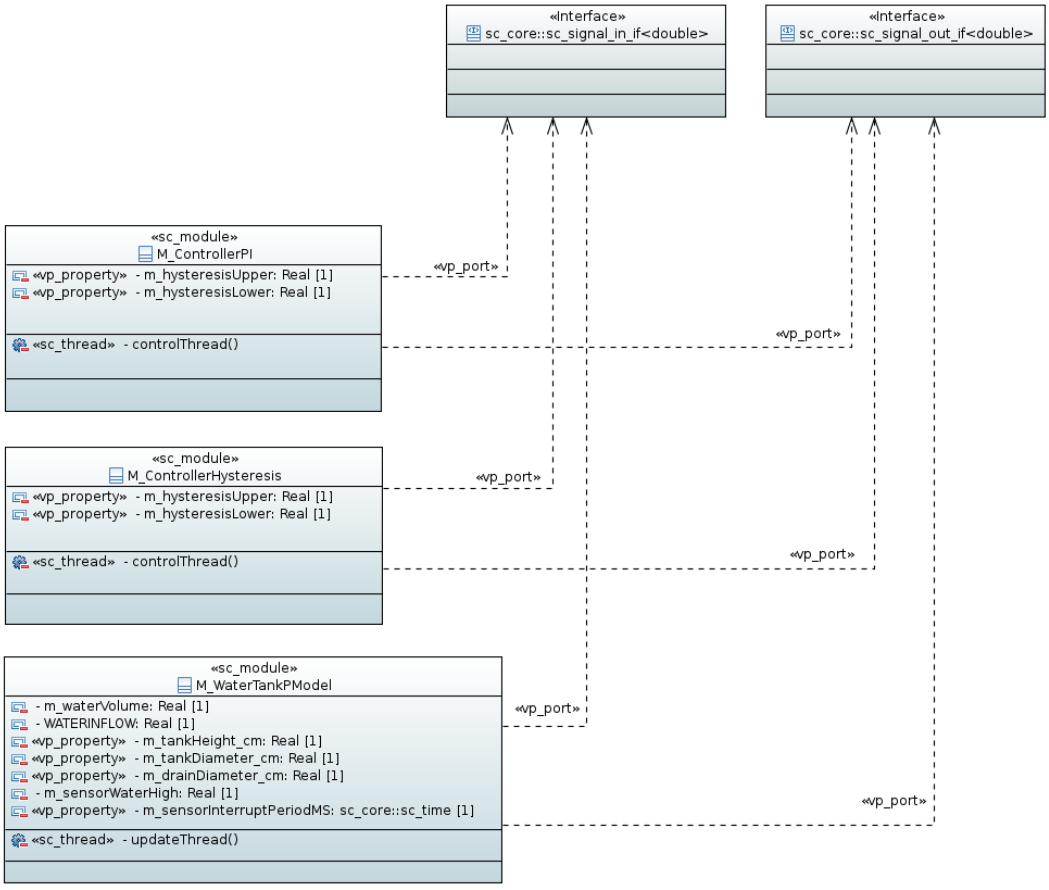


Abbildung 6.4: Spezifikation der BSEs für das Wassertankmodell

als `sc_module` gekennzeichnet und beinhaltet einen SystemC-Thread. Des Weiteren besitzt die BSE mehrere Membervariablen, wovon einige durch die Konfigurationsdatei parametrisierbar sind. Die parametrisierbaren Membervariablen besitzen den Stereotyp `vp_property`. Neben den direkten Membervariablen besitzt die Klasse zwei

Assoziationen, welche die Codegenerierung auch auf Membervariablen abbildet. Bei Assoziationen, die als `vp_port` spezifiziert sind, erzeugt die Codegenerierung automatisch einen lokalen Port sowie Methoden zum Binden des Ports. Der Datentyp der Ports entspricht dem Ziel der Assoziation.

Fehlerinjektor-Erweiterung
Ein weiteres Profil fügt einen Stereotyp zur Fehlerinjektion hinzu. Der Stereotyp `es_injector` assoziiert Membervariablen der BSEs mit einem Fehlerinjektor. Die Codegenerierung nimmt, für die assoziierten Membervariablen, automatisch die benötigten Ersetzungen, wie die Deklaration des Fehlerinjektors und der Aufruf des Konstruktors, vor. Der Anwender muss keinerlei Änderungen am Simulationsmodel vornehmen.

Durch dieses Vorgehen wird der Mehraufwand bei der Implementierung des Simulationsmodells reduziert. Zur Realisierung der dynamischen Konfiguration muss neben der eigentlichen Simulationscodegenerierung zusätzlich eine IP-XACT-Beschreibung der Komponente erfolgen. Die Generierung erfolgt parallel zu Quellcodegenerierung und verwendet die gleiche Datenbasis. Die Reduktion des Aufwands durch die Generierung der Konfigurationsdatei wird im Folgenden vorgestellt.

6.2 Spezifikation der Fehlereffektsimulation

Ein wichtiger Aspekt bei der Durchführung der Fehlereffektsimulation ist die Spezifikation und Dokumentation des analysierten Systems und der durchgeführten Simulationen. Wie bereits motiviert, besteht die Systemsimulation aus parametrisierbaren BSEs, die zu unterschiedlichen Systemsimulationen verbunden werden. Die vollständige Simulationsspezifikation beinhaltet somit alle verwendeten Module, ihre Parametrisierung sowie ihre Verbindungen untereinander. Die grafische Spezifikation und die Generierung der Konfigurationsdatei unterstützt den Anwender und vereinfacht die Dokumentation. Die vorgestellte Methode wird mittels eines Kompositionsstrukturdiagramms demonstriert. Diagramme dieses Typs spezifizieren die Verknüpfung der an der Simulation beteiligten BSEs sowie deren Parametrisierung. In dem Kompositionsstrukturdiagramm spezifiziert der Anwender Instanzen der BSE-Klassen und verknüpft diese. Somit bilden die spezifizierten Klassen die Grundlage zur Spezifikation der Simulationsinstanz, ähnlich wie die BSEs die Grundlage zur Simulation bilden.

Die oberste Instanz der Simulation wird mit dem Stereotyp `vp_system` gekennzeichnet, um die Spezifikation des virtuellen Prototyps zu betonen. Diese Komponente besteht aus mehreren mit einander verbundenen BSEs. Jeder BSE werden die benötigten Parameterwerte, mittels eines Tupels aus Parametername und Parameterwert zugewiesen. Diese Spezifikation stellt somit die Ausprägungsspezifikation der Klassen und Assoziationen aus der Spezifikation der Simulationseinheitenbibliothek dar.

Zur Steigerung der Übersichtlichkeit und Erhöhung der Wiederverwendbarkeit ist es möglich, eine hierarchische Struktur der BSEs zu spezifizieren. Hierzu stellt

der Ansatz unterschiedliche Stereotypen bereit. Neben dem Stereotyp `vp_system` gibt es den Stereotyp `vp_system_template`. Dieser ermöglicht die Modellierung einer Hierarchie. Innerhalb eines `vp_system_templates` werden, wie beim `vp_system`, unterschiedliche BSEs instanziiert, parametrisiert und verknüpft. Das Template kann Bestandteil der Spezifikation `vp_system` oder wiederum eines Templates sein. Hierdurch ist es möglich, wiederkehrende Strukturen in einem Template zu bündeln und mehrfach zu instanziieren.

Abbildung 6.5 zeigt die Spezifikation des Wassertankmodells ohne die Verwendung von Templates. Die Simulation besteht aus sechs BSEs. Die im Klassendia-

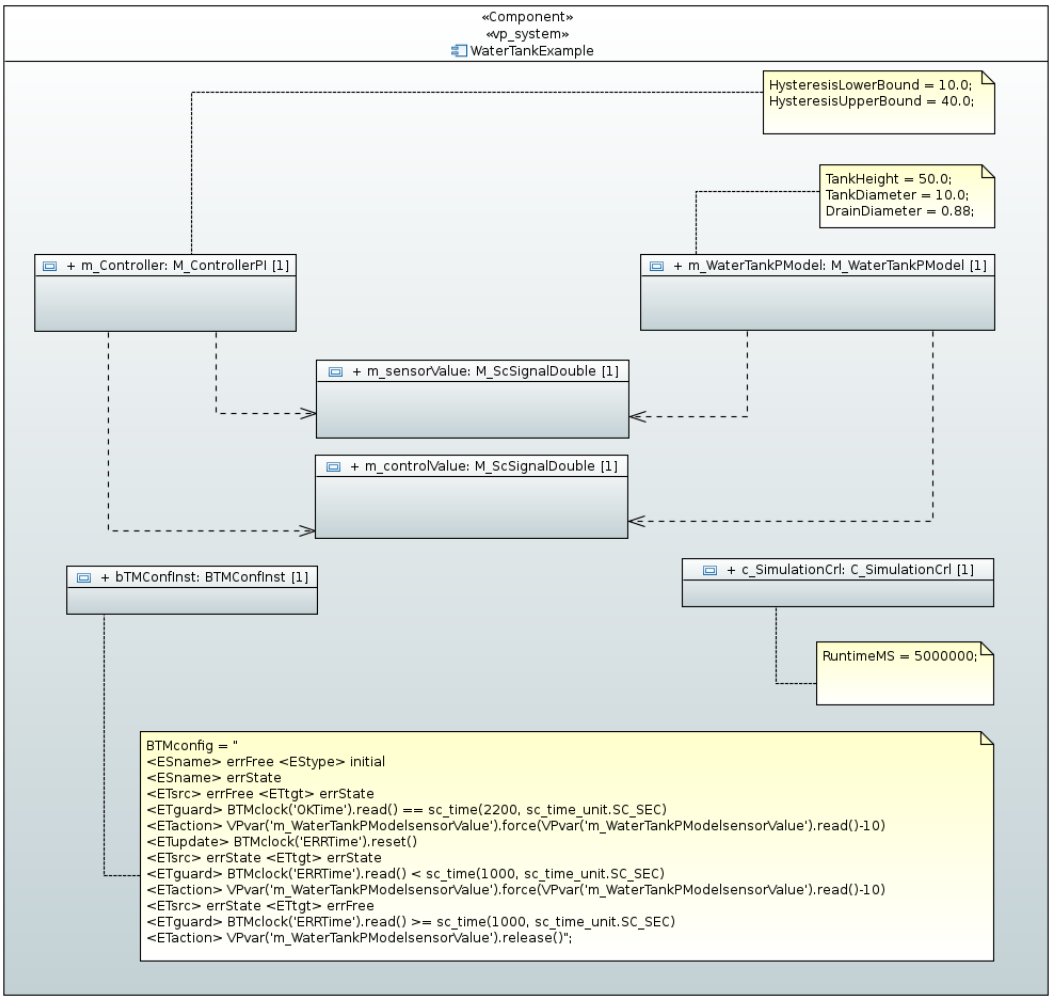


Abbildung 6.5: Spezifikation der Systemsimulation für die Analyse des Wassertanks

gramm spezifizierten Assoziationen sind in dem Diagramm mit Instanzen hinterlegt. Des Weiteren ist die Parametrisierung der einzelnen Instanzen dargestellt. Die Parametrisierung der BSE erfolgt über assoziierte Kommentare, die mit den Modulen verknüpft werden. Aufgrund der Tatsache, dass Papyrus kein Objektdiagramm vorsieht und innerhalb des Kompositionsstrukturdiagramms keine Parametrisierung der Komponenten möglich ist, wird die Lösung über die assoziierten Kommentare ver-

folgt. Neben dem eigentlichen Modell sind auch administrative Informationen, z. B. die Simulationsdauer sowie das zu injizierende Fehlerverhalten, in dem Diagramm spezifiziert.

Das Diagramm bildet die Grundlage zur Generierung der Konfigurationsdatei. Die IP-XACT-Datei beinhaltet alle Informationen des Diagramms und wird zur Ausführung der Simulation benötigt. Das erzeugte IP-XACT-Design beinhaltet alle Parameterbelegung und Assoziationen. Eine manuelle Erstellung oder Wartung der Konfigurationsdatei durch den Anwender ist nicht notwendig.

6.3 Fehlerspezifikation

Die Spezifikation der Fehlereffektsimulation beinhaltet auch die Spezifikation des zu injizierenden Fehlerverhaltens. Hierbei wird der gleiche Ansatz wie zur Parametrisierung von BSEs verwendet. Die Systemspezifikation wird um die Komponente `BTMConfInst` ergänzt. Diese Komponente entspricht dem BTM-Interpreter (Stressor) in der eigentlichen Simulation. Das Modul besitzt ein Parameter zur Angabe eines textuell beschriebenen BTMs. Das Modul `bTMConfInst` bzw. dessen Parametrisierung spezifiziert somit das zu injizierende Fehlerverhalten. Abbildung 6.5 verdeutlicht dieses Vorgehen.

Ein erweiternder Ansatz sieht die grafische Spezifikation des BTMs vor und die automatisierte Generierung der textuellen Repräsentation. Das BTM wird mit Zustandsdiagrammen der UML modelliert und in die Spezifikation der Simulation eingebunden. Basierend auf den Standard UML-Zustandsdiagrammen wird eine Erweiterung mit dem UML-Profil zur Fehlerinjektion bereitgestellt, welche die Spezifikation der BTM-Eigenschaften ermöglicht. Das Profil umfasst Stereotypen für <ETupdate>, <ETsync> sowie <ETaction>. Modellierungprimitiven der UML spezifizieren die Zustände und Transitionen des BTMs. Die entwickelten Stereotypen annotieren Bedingungen, Aktionen und Synchronisationsereignisse an die Transition. Abbildung 6.6 zeigt die grafische Spezifikation des BTMs aus Abbildung 6.5. Stereotypen annotieren die Eigenschaften des BTMs an den Transitionen. Der Übergang von Zustand `errFree` in den Zustand `errState` ist mit den Stereotypen `BTMguard`, `BTMaction` und `BTMupdate` spezifiziert.

In der Fehlereffektspezifikation wird dann lediglich auf das jeweilige Zustandsdiagramm verwiesen. Das heißt, anstelle einer textuellen Spezifikation des BTMs, verweist die `BTMConfInst` über eine Usage-Assoziation auf den Zustandsautomaten. Abbildung 6.7 verdeutlicht dies mit der Relation zwischen den Objekten `bTMConfInst` und `waterTankFault`, das die grafische Spezifikation des BTMs darstellt. Die grafische Notation wird automatisiert in die textuelle Repräsentation überführt und ergänzend der Fehlereffektspezifikation hinzugefügt. Vorteil dieses Ansatzes ist, dass die komplette Parametrisierung der Fehlereffektsimulation in einem Diagramm vorliegt. Würde dies nicht der Fall sein, können nachträgliche Änderungen an dem Zustandsdiagramm zu einer verfälschten Dokumentation der Fehlereffektsimulation führen. Sollte z. B. lediglich ein Link auf das Zustandsdiagramm verwendet werden, würde

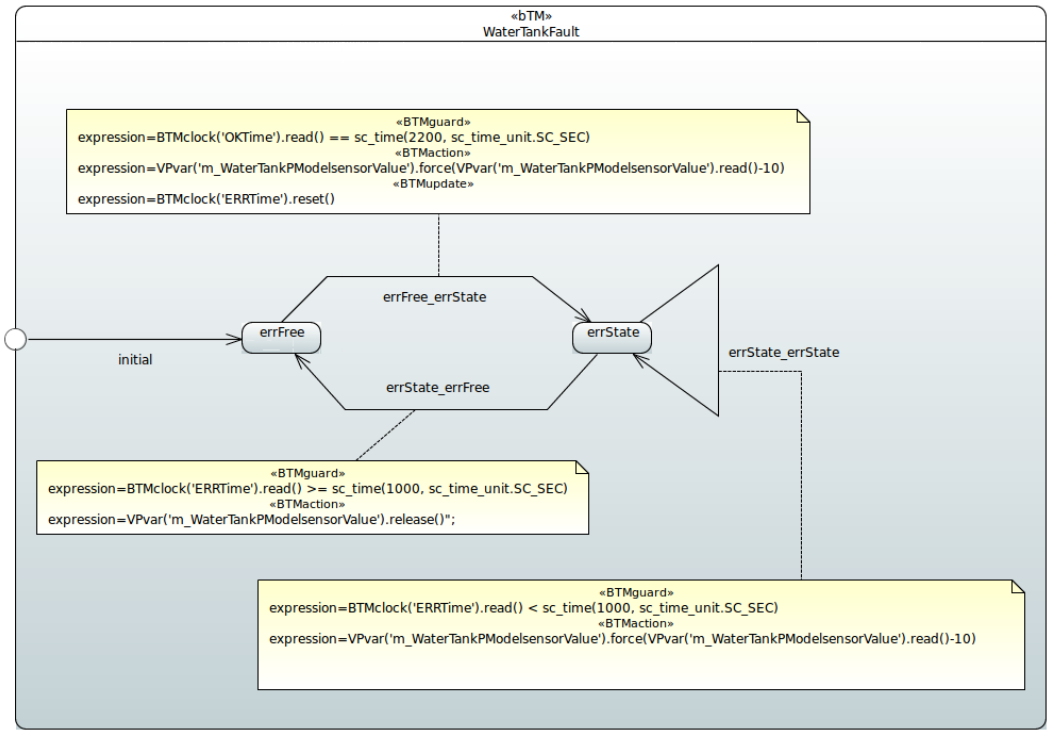


Abbildung 6.6: Grafische Spezifikation des BTMs aus Abbildung 6.5

eine Änderung des Zustandsautomaten nicht im übergeordneten Diagramm dokumentiert.

Ein weiterer Aspekt, der für die explizite Generierung des BTM-Kommentars spricht, anstelle der alleinigen Verlinkung des UML-Zustandsautomaten, ist die Möglichkeit zur Durchführung von Ersetzungen in der BTM-Spezifikation. Es ist möglich an der Assoziation zwischen Zustandsautomat und BTMConfInst, eine Ersetzungsvorschrift zu annotieren. Die Ersetzung wird auf den generierten Kommentar angewendet. Das Vorgehen ermöglicht die Anpassung des BTMs an den jeweiligen Anwendungsfall. Hierbei werden Identifikatoren bzw. eindeutige Werte im BTM bei der Generierung der textuellen Repräsentation ersetzt. Diese eindeutigen Schlüsselwörter in der BTM-Spezifikation können bei der Instanziierung mit Werten belegt werden. Durch das Vorgehen ist es möglich, Anwendungsfall unabhängige Meta-BTM-Beschreibungen vorzuhalten. Erst die Instanziierung im Anwendungsfall erzeugt das konkrete BTM. Als Beispiel ist es möglich, eine Spezifikation des klassischen, transienten Bitfehlermodells zu erstellen und den Auftrittszeitpunkt und die Dauer parametrisierbar zu gestalten. Erst die Instanziierung legt die applikationsspezifisch Daten fest. Der Anwender hat somit die Möglichkeit sich eine Fehlerbibliothek zu generieren, aus der er die unterschiedlichen Fehler auswählt und auf den Anwendungskontext anpasst.

Abbildung 6.7 zeigt die Instanziierung des in Abbildung 6.6 spezifizierten BTMs. Die Zuweisung des Zustandsautomaten zur BTMConfInst ist mit der Ersetzungsregel `2200;1516` annotiert. Dies bewirkt, dass in der generierten BTM-Spezifikation die

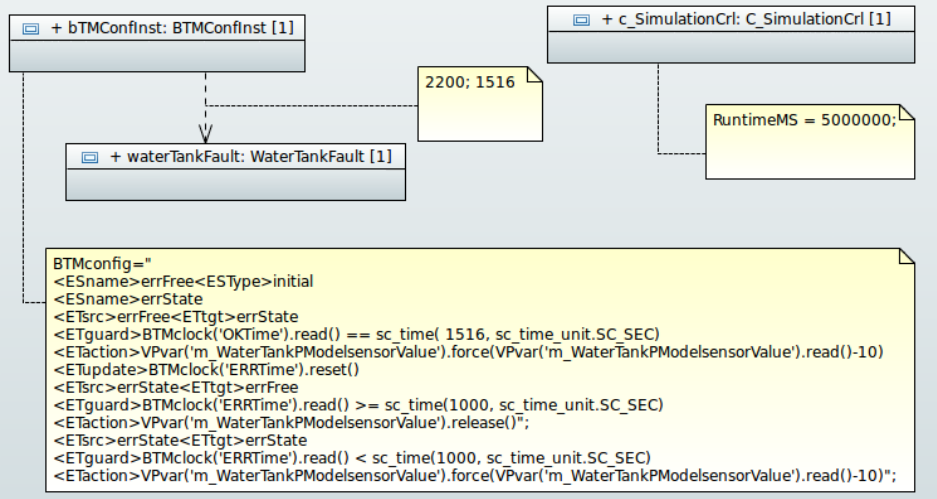


Abbildung 6.7: Instanziierung und Anpassung eines BTMs

2200 durch die 1516 ersetzt wird. Der ersetzte Zeitpunkt entspricht dem Auftrittszeitpunkt des Fehlers. Als Ersetzungsregel sind in der bereitgestellten Werkzeugumgebung reguläre Ausdrücke erlaubt.

Die vorgestellten grafischen Ansätze zielen vorrangig auf die Benutzerfreundlichkeit des Konzepts der BTMs ab. Hierunter fällt die allgemeine grafische Spezifikation als auch dedizierte Werkzeugunterstützungen. So wird der Anwender bei der Erstellung der textuellen Repräsentation des BTMs durch die Werkzeugumgebung unterstützt. Hierbei wird die Semantik aus Abschnitt 5.3.1 verwendet. Mittels hinterlegter Grammatik bietet der Editor dem Anwender die möglichen Tags als Autovervollständigung zur Auswahl an und unterstützt hierdurch den Anwender bei der Spezifikation. Die grafische Spezifikation des BTMs und die automatisierte Generierung erhöhen die Lesbarkeit sowie die Wiederverwendbarkeit der BTMs in der Modellierungsumgebung. Die eigentliche Übergabe an die Systemsimulation erfolgt weiterhin mit der IP-XACT-basierten Konfigurationsdatei. Das Konzept einer zustandsbasierten Beschreibung, die während der Simulation interpretiert wird, bleibt unverändert.

Kapitel 7

Systemanalyse mittels Fehlereffektsimulation

Das folgende Kapitel betrachtet generelle Analysemethoden in Abschnitt 7.1 sowie dedizierte Fallbeispiele in Abschnitt 7.2. Beide Aspekte verdeutlichen wie der in dieser Arbeit vorgestellte Analyseansatz in heutige Entwicklungsabläufe integriert werden könnte und an welchen Stellen Unterstützung geleistet wird. Anhand von einigen Fallbeispielen wird Anwendung des Ansatzes demonstriert.

7.1 Analysemethodik

Der Grundgedanke der Fehlereffektsimulation ist, dem Anwender eine Möglichkeit zur Durchführung von Was-Wäre-Wenn-Analysen bereitzustellen. Dadurch, dass das Systemverhalten im Simulationsmodell spezifiziert ist, kann der Anwender Fehler einstreuen und die Effekte beobachten. Prinzipiell lassen sich mehrere Phasen der Anwendung unterscheiden.

Während der Entwicklung bzw. Konzeptionierung des Systems dient die Fehlereffektsimulation vorrangig zur Bewertung von Entwurfsentscheidungen. Der Entwickler bzw. Systemarchitekt kann die getroffene Entwurfsentscheidung anhand der Systemsimulation unter Berücksichtigung der funktionalen Sicherheit bewerten. Auch die Integration von Teilsystemen in eine übergeordnete Struktur kann hierdurch überprüft und unterschiedliche Alternativen evaluiert werden.

Liegt die Implementierung des Gesamtsystems vor, kann die Systemsimulation bereits Testfälle für die Qualifikation mit den physikalischen Prototypen abschätzen. Die Exploration am Simulationsmodell identifiziert interessante Testfälle, die der physikalische Prototyp verifiziert.

Die Fehlereffektsimulation unterstützt damit die Qualifikation des Gesamtsystems bzw. der Systemteile in Kombination mit den physikalischen Prototypen. Hierbei ist von Vorteil, dass die Systemsimulation ein wiederholbares Systemverhalten aufweist und es ermöglicht, alle modellierten Systemteile zu beobachten. Dies bietet vor allem bei, mit physikalischen Prototypen, nur schwer wiederholbaren Fehlern einen Vorteil. In diesen Fällen kann die Simulation das Systemverhalten reprodu-

zierbar und im Detail beobachtbar untersuchen. Ein weiterer Vorteil zeigt sich bei Fehlern, die den physikalischen Prototypen zerstören bzw. beschädigen. Hier kann die Systemsimulation eine Voruntersuchung durchführen und die wichtigsten Systemtests identifizieren. Die physikalischen Tests beschränken sich auf die repräsentativen Testfälle. Das Vorgehen reduziert unnötige Systemtests und vermeidet somit eine unnötige Beschädigung physikalischer Prototypen.

Im Folgenden sind einzelne Unterstützungen für gängige Zuverlässigkeitsanalysen dargestellt. Hierbei wird anhand der injizierten Fehler und des untersuchten Systems unterschieden.

7.1.1 Single Fault – Single System

Eine wichtige Anwendung ist die Bewertung der Auswirkungen eines Fehlers. Wie in Abschnitt 3.1 vorgestellt, ist eine heutzutage weitverbreitete Analysemethode die Gruppe der FMEA. Diese Analysen ordnen unterschiedlichen Fehlerursachen die resultierenden Fehlereffekte zu. Diese Zuordnung erfolgt, wie beschrieben, meist durch eine Gruppe von Anwender, mit detailliertem Systemwissen. Der hier vorgestellte Ansatz versetzt die Anwender in die Lage, die Fehlereffekte automatisch anhand von Systemsimulationen zu ermitteln. Hierzu spezifiziert der Anwender alle potenziellen Fehlerursachen in je einem BTM. Zusätzlich fügt er die benötigten Fehlerinjektoren und Monitore in das Simulationsmodell hinzu. An den vordefinierten Beobachtungspunkten kann der Anwender dann die Auswirkungen des Fehlers beobachten. Durch diesen Ansatz wird das benötigte, detaillierte Systemwissen vom Anwender auf das Simulationsmodell übertragen. Der Anwender bleibt verantwortlich für die korrekte und vollständige Spezifikation der Fehlerursachen. Auf niedrigen Abstraktionsebenen sind die Fehlermodelle meist sehr gut erforscht und der Anwender kann auf bekannte Fehlermodelle wie dem Stuck-at-Fehlermodell zurückgreifen. Der Anwender muss lediglich entscheiden, welche Daten im Modell von den Fehlern betroffen sind. Das heißt, er assoziiert die entsprechenden Variablen mit einem Fehlerinjektor. Aber auch auf höheren Abstraktionsebenen existieren Richtlinien, die den Anwender bei der Spezifikation der Fehlerursachen leiten. Avizienis et al. definieren in [3] allgemeine Fehlerklassen, welche in dieser Arbeit zur Klassifikation der Fehlermodi an den Beobachtungspunkten verwendet werden. Der Ansatz zur Fehlereffektbeobachtung wird in Abschnitt 5.4.1 im Detail vorgestellt. Die Beobachtungspunkte unterscheiden folgende Fehlerklassen:

- Inhaltsfehler (engl. content failures),
- frühzeitige Verfügbarkeitsfehler (engl. early timing failures),
- zu späte Verfügbarkeitsfehler (engl. late timing failures),
- Haltefehler (engl. halt failures) und
- zufällige Fehler (engl. erratic failures).

Die allgemeine Klassifizierung gibt zusätzlich eine Richtlinie bei der Spezifikation von Fehlerursachen wieder. Hierbei wird grob in eine Änderung des Datenwerts sowie dessen Verfügbarkeit unterschieden. Ein Vorgehensmodell ist, dass der Anwender lediglich die zu verfälschenden Daten identifiziert und hierauf exemplarische Fehler aus den allgemeinen Klassen anwendet. Hierbei ist hervorzuheben, dass die Fehlerklassen kein detailliertes Fehlerverhalten spezifizieren, sondern lediglich eine Orientierungshilfe für den Anwender darstellen. Die detaillierte funktionale Spezifikation des Fehlers muss weiterhin der Anwender vornehmen. Hier liegt ein indirekter Vorteil des Ansatzes. Ist es bei der FMEA meist ausreichend sehr allgemeine, weniger konkrete Fehlerursachen zu spezifizieren, wie z. B. „die Verfälschung des Sensorwerts“, wird der Anwender bei der Fehlereffektsimulation gezwungen, die Auftrittsbedingungen sowie die Schwere des Fehlers zu bestimmen. Meist ist hierbei keine eindeutige Beziehung zwischen Fehlerbeschreibung in der FMEA und Fehlerbeschreibung in der Simulation herzustellen. Aus diesem Grund muss der Anwender Explorationsverfahren einsetzen, um aus einer abstrakten Fehlerbeschreibung unterschiedliche Injektionsbeschreibungen abzuleiten. Dies stellt eine Zwischenstufe zu dem in Abschnitt 7.1.2 vorgestellten Vorgehen dar.

Ein weiterer Punkt, der vom Anwender festgelegt wird, ist die Platzierung der Fehlerinjektionspunkte sowie der Beobachtungspunkte. Hier bietet sich die Unterteilung des Systems anhand der erbrachten Dienste an. So bietet ein Kommunikationscontroller oder eine Signalleitung den Dienst der Datenübertragung, Prozessoren bieten Dienste zur Ausführung von Berechnungsoperationen und Algorithmen erbringen anwendungsspezifische Dienste. Bei der dienstorientierten Betrachtung des Systems wird zwischen den eingehenden Daten sowie dem erbrachten Dienst unterschieden. Für die Fehlerinjektion ist eine Betrachtung der Dienstschnittstelle, an welcher der Dienst des Teilsystems erbracht wird, wichtig. An den hier ausgetauschten Daten sollten Beobachtungspunkte die Korrektheit des Dienstes überprüfen. Abbildung 7.1 zeigt die Dienstpartitionierung am Wassertankbeispiel. Ein Anwendungsfall ist die

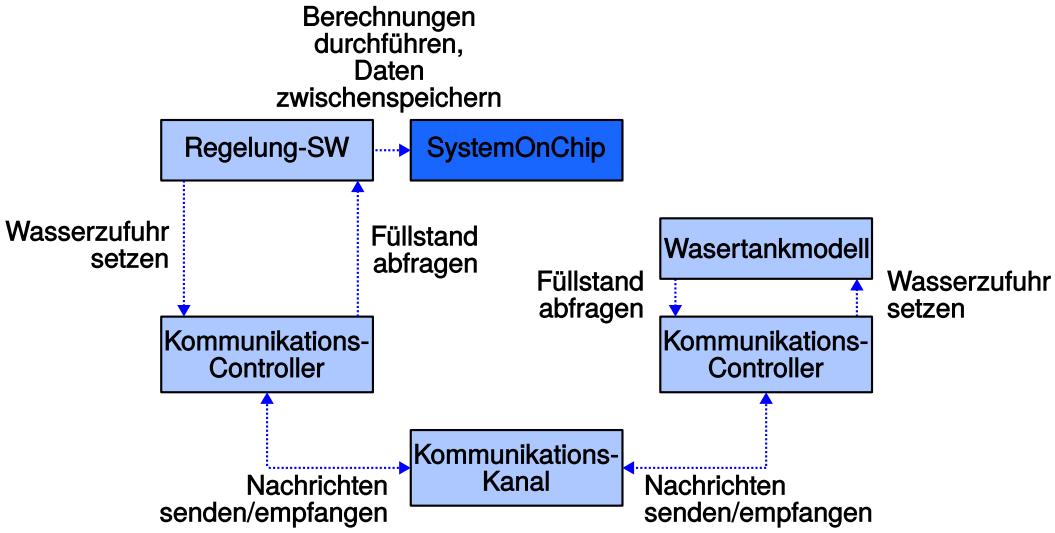


Abbildung 7.1: Dienstschnittstellen im Wassertankbeispiel

Überprüfung von Auswirkungen von Fehlern im Kommunikationskanal. In diesem Anwendungsfall injizieren Fehlerinjektoren Fehler in die übertragenen Nachrichten und Beobachtungspunkte an der Schnittstelle zur Anwendung detektieren die Auswirkungen. Eine weitere Analyse ist, ob der SoC die korrekten Daten zurückliefert. Die Daten können z. B. durch die Verfälschung von Registerinhalten abweichen. Bei der Analyse des Gesamtsystems ist es meist ausreichend nur die systemrelevanten Schnittstellen, in diesem Fall den Füllstand, zu beobachten und die internen Dienstschnittstellen zu vernachlässigen.

Durch die Abhängigkeiten der Dienste, z. B. benötigt ein Algorithmus die Berechnungsdienste des Prozessors, kommt es zur Propagierung des Fehlers über mehrere Beobachtungspunkte. Beobachtungspunkte an internen Dienstschnittstellen ermöglichen einen detaillierteren Einblick in das System. Hierdurch ist eine Analyse der Fehlerpropagierung möglich. Sind alle Fehlerursachen sowie die Menge der Beobachtungspunkte spezifiziert, kann der Anwender die Fehlereffektsimulation durchführen und allen Fehlerursachen eine detaillierte, quantifizierbare Fehlereffektbeschreibung zuweisen. Er wird somit in die Lage versetzt eine objektive, lediglich vom Modell abhängige, FMEA durchzuführen.

Durch die Definition mehrerer Beobachtungspunkte wird die Fehlerpropagierung über Komponenten erfasst. Die Erfassung der Fehlerpropagierung ist bei der Unterstützung einer FTA hilfreich. Hierbei ist es möglich, durch den Anwender erstellte Fehlerbäume, durch die Fehlereffektsimulation zu überprüfen. Hierzu werden die Events, im Speziellen die Basic Events, mit Fehlerinjektoren assoziiert. Das Fehlerverhalten der Basic-Events wird in die Simulation eingebracht und die Effekte auf das System beobachtet. Die Beobachtung erfolgt an dem resultierenden Top Event, bzw. bei der Analyse der Fehlerpropagierung bereits an den dazwischenliegenden Events. Das Systemmodell stellt eine Beziehung zwischen auslösendem Fehler und resultierendem Ereignis her. Eine korrespondierende Beziehung ist im Fehlerbaum enthalten. Hierdurch kann der Anwender den vorgestellten Ansatz zur Überprüfung des Fehlerbaums verwenden. Im dargestellten Beispiel, in Abbildung 3.2, wird z. B. nur durch die gemeinsame Injektion des Fehlers A_2 und S_{in} das Top Event TE ausgelöst. Wird kein Fehlereffekt detektiert, ist die Fehlereffektsimulation inkonsistent zum Fehlerbaum und der Anwender muss die betroffene Stelle überprüfen.

Auf die gleiche Weise ist es möglich zu überprüfen, ob Fehlereffekte durch die Kombination von Blattelementen ausgelöst werden, die der Fehlerbaum noch nicht spezifiziert. Hierbei ist anzumerken, dass bei der Überprüfung der Fehlerkombinationsraum sehr stark anwächst, was zu einem erhöhten Simulationsaufwand führt. Mit diesem Vorgehen überprüft der Ansatz nicht nur existierende Pfade im Fehlerbaum, sondern findet auch neue mögliche Pfade.

7.1.2 Multiple Fault – Single System

In dieser Analysekategorie lassen sich zwei Ansätze unterscheiden. Die erste Analysekategorie simuliert einen Fehler in unterschiedlichen Varianten. Die zweite Analysekategorie untersucht mehrere verschiedene Fehler gleichzeitig.

Wie bereits im vorherigen Abschnitt vorgestellt, spezifizieren herkömmliche Ansätze, wie die FMEA und die FTA, Fehler häufig sehr abstrakt. Hierbei ist eine eindeutige Abbildung auf ein zu injizierendes Fehlerszenario häufig nicht möglich. Aus diesem Grund wird ein Fehler in unterschiedliche BTMs transferiert und mehrere Simulationen durchgeführt. Abbildung 5.20 zeigt die Freiheitsgrade, die häufig variiert werden müssen. Hierzu zählen z. B. die Fehlerdauer und der Auftrittszeitpunkt des Fehlers. Ein abstrakter Fehler wird mit unterschiedlichen Dauern und Auftrittszeitpunkten simuliert. Der resultierende Fehlereffekt wird über die unterschiedlichen Simulationen ermittelt.

Bei den unvollständig spezifizierten Fehlern leistet die Simulation, durch die Möglichkeit zur wiederholten Ausführung, eine wichtige Unterstützung. Durch die Varianz der Freiheitsgrade der Fehlerursache ist es möglich, quantitativ die Auftrittswahrscheinlichkeit der Fehlerauswirkung zu ermitteln. Diese Information unterstützt die Durchführung der FTA und FMEA. Die Fehlereffektsimulation ermöglicht bei der FTA die Wahrscheinlichkeiten von Blattknoten (Basic Event) zu ermitteln. Bei der FMEA wird die Wahrscheinlichkeit direkt für die einzelnen Fehlermodi benötigt.

Eine weitere Kategorie ist die Untersuchung von Fehlereffekten, die aus der Kombination unterschiedlicher Fehlerursachen resultieren. Bei gängigen Fehlerbäumen bedingen häufig mehrere Fehlerursachen einen Fehlereffekt. In der FTA wird dies durch die `und`-Verknüpfung der Auftrittsereignisse spezifiziert. Bei diesen Fehlerszenarien bietet die Fehlereffektsimulation einen weiteren grundlegenden Vorteil. Durch die immer komplexeren Systeme mit unterschiedlichen Abhängigkeiten zwischen den Komponenten wird es für den Anwender schwer alle Abhängigkeiten zu identifizieren und die korrekte Fehlerpropagierung festzustellen. Vorteil des Simulationsmodells ist, dass die Abhängigkeiten inhärent im Modell spezifiziert sind. Somit ist es ohne Mehraufwand möglich, mehrere Fehler zu injizieren und die kombinierte Wirkung auf das System zu testen. Durch vorausgehende ‚Single Fault – Single System'-Analysen sind bereits alle Fehlerursachen mittels BTMs spezifiziert sowie alle interessanten Beobachtungspunkte in der Simulation eingefügt. Somit ist eine Kombination der unterschiedlichen Fehlerfälle ohne Mehraufwand möglich. Über die vorgestellten Synchronisationsmechanismen ist es sogar möglich, unterschiedliche Auslösereihenfolge der atomaren Fehlerursachen zu spezifizieren. Es ist ersichtlich, dass durch die Kombination der unterschiedlichen Fehlerursachen der Analyseraum sehr stark erhöht wird. Somit ist dieser Ansatz als nachgelagerte Analyse zur ‚Single Fault – Single System'-Analyse zu betrachten.

7.1.3 Single/Multiple Fault – Multiple System

Der letzte Analyseansatz tritt häufig in frühen Entwicklungsstufen mit abstrakten Modellen auf. Im Entwurfsablauf existieren unterschiedliche Realisierungsansätze für das finale System. Das vorgestellte Vorgehen unterstützt den Anwender bei dem Treffen von Entwurfsentscheidungen, unter dem Gesichtspunkt der Zuverlässigkeit. Hierzu spezifizieren Simulationsmodelle die unterschiedlichen Realisierungen. Diese Simulationsmodelle ermöglichen das Verhalten der Systemalternativen, auf einen

oder mehrere Fehler, zu analysieren.

Bei der Entwurfsraumexploration entsteht sehr schnell eine Vielzahl von Alternativen. So können die unterschiedlichen Simulationsmodelle komplett unterschiedliche Subsysteme, wie z. B. unterschiedliche Kommunikationstechnologien beinhalten. Aber auch die alleinige Parametrisierung von COTS, resultiert in eine Vielzahl von Systemalternativen. In den Veröffentlichungen [17, 19, 20] wird ein Ansatz zur Spezifikation von Plattformvarianten vorgestellt, der es erlaubt sowohl struktureller Variabilität als auch unterschiedliche Parameterräume zu spezifizieren. Zur Spezifikation wird die Object Variant Constraint Language (OVCL), eine Erweiterung der Object Constraint Language (OCL), verwendet. Dies bedeutet, dass die Spezifikation der Variabilität auf der UML basiert und hierdurch leicht in die Fehlereffektsimulation integrierbar ist. Mithilfe der OVCL ist es möglich, die Freiheitsgrade eines Systementwurfs zu beschreiben. Neben der einfachen Angabe der Freiheitsgrade ist es möglich, komplexe Abhängigkeiten zwischen den variablen Entwurfsparametern zu spezifizieren. Der vorgestellte Ansatz überführt die Freiheitsgrade und die einzelnen Abhängigkeiten in eine Problembeschreibung der Satisfiability Modulo Theories (SMT). Gängige Solver generieren valide Instanzen des Problems, die wiederum validen Systemvarianten entsprechen. Ein solcher Ansatz kombiniert mit der Fehlereffektsimulation ermöglicht die Generierung unterschiedlicher Systemalternativen mit integrierter Fehlerinjektion. Dies ermöglicht die Bewertung der einzelnen Systemalternativen unter dem Gesichtspunkt der funktionalen Zuverlässigkeit.

Eine Erweiterung stellt die Einbeziehung der Fehlerspezifikation in die Variantengenerierung dar. Bei diesem Vorgehen wird das Konzept der strukturellen Varianten auf die Fehlerinjektion erweitert. Die Spezifikation der Fehler erfolgt über die Komponente `BTMConfInst`. Mit der OVCL ist es möglich, diese Komponente als Alternative oder als Selektion zu deklarierten. Hierdurch selektiert der Variantengenerierungsprozess unterschiedliche Fehler bzw. kombiniert unterschiedliche Fehler, für die Fehlereffektsimulation.

Der in [17] vorgestellte Ansatz erzeugt, in Bezug auf die gesetzten Einschränkungen, alle validen System- bzw. Fehlervarianten. In [17] Abschnitt 4.3 wird dargelegt, wie der Ansatz „eine vollständige Generierung aller möglichen validen Systemvarianten garantiert“. Allein durch die Beschreibung der Systemalternative entsteht eine Vielzahl von Alternativen. Kombiniert mit den parametrisierbaren Fehlerbeschreibungen nimmt der zu testende Raum sehr stark zu. Erweiterungen, die den Testraum strukturiert erkunden [66, 67], anstelle alle Varianten zu generieren, können den Simulationsaufwand erheblich reduzieren.

7.2 Fallbeispiele

Neben dem in dieser Arbeit bereits diskutierten Anwendungsfall, einer Füllstandsregelung, präsentiert dieses Kapitel zwei weitere Anwendungsbeispiele aus der Automotive-Domäne. Hierbei handelt es sich um ein MOST-Kommunikationsszenario sowie um einen synthetischen Anwendungsfall, der wichtige Aspekte und Techniken heutiger Automotive-Systeme wiedergibt. Anhand selektiver Szenarien wird die An-

wendbarkeit sowie die Vorteile des entwickelten Ansatzes aber auch dessen Grenzen aufgezeigt.

Abschnitt 7.2.1 stellt die unterschiedlichen Fallstudien vor, bevor in den folgenden Abschnitten einzelnen Aspekte analysiert werden. In Abschnitt 7.2.2 wird die Übertragungsperformanz unterschiedlicher Konfigurationen einer Datenübertragung über den MOST-Bus untersucht. Diese Analysen finden häufig im Rahmen von Entwurfsraumexplorationen statt und dienen der Findung einer guten Parametrisierung bzw. Realisierung des Systems. Anhand der Fallstudie wird die Flexibilität des Simulationsrahmenwerks und der angedachten Struktur der Simulationsmodelle hervorgehoben. Des Weiteren wird das Potenzial von SystemC zur Integration realer Softwareprototypen vorgestellt. Aspekte der Analyse wurden vom Autor bereits in [60, 95] publiziert.

Die zweite Fallstudie greift die in dieser Arbeit bereits auszugsweise vorgestellten Analysen auf und zeigt die Fehlerinjektion anhand eines Regelungssystems. Fokus des Fallbeispiels bildet die eigentliche Fehlerinjektion sowie die benötigten Modifikationen an den Simulationsmodellen. Die Fallstudie verdeutlicht wie sich Hardwarefehler auf die Softwareausführung und somit auf den Regelalgorithmus auswirken.

Wie in den vorherigen Abschnitten dargelegt, ist eine Kernanforderung des in der Arbeit vorgestellten Ansatzes, die entwurfsbegleitende Anwendbarkeit. Diese entwurfsbegleitende Anwendbarkeit wird anhand des synthetischen Anwendungsfalls demonstriert. Abschnitt 7.2.4.1 präsentiert Untersuchungen an einem rein funktionalen Modell. Diese Art der Modelle liegt in frühen Entwicklungsphasen vor, wenn lediglich (Konzept-)Algorithmen der Funktionalität vorliegen. Die Modelle spezifizieren die Algorithmen ohne jeglichen Bezug zur finalen Hardware bzw. Systemarchitektur. Abschnitt 7.2.4.2 beschreibt denselben Anwendungsfall mittels eines transaktionsbasierten Modells. Hierbei umfassen die Modelle die grobe Systemarchitektur, wie z. B. die Gruppierung der Algorithmen auf Steuergeräte oder die Abbildung von Kommunikation auf ein gemeinsam genutztes Bussystem. Die letzte betrachtete Abstrahierung verwendet detaillierte Busmodelle zur Modellierung des Kommunikationsverhaltens und ist in Abschnitt 7.2.4.3 vorgestellt. Diese Art von Modellen liegt meist in späteren Entwicklungsphasen vor, wenn die Hardwareabbildung spezifiziert ist. Meist liegen bereits physikalische Prototypen vor. Die virtuelle Qualifikation ergänzt die Analyse mit den physikalischen Prototypen. Anhand von Fehlerbeispielen wird aufgezeigt, wie Anwender Fehlerauswirkungen in abstrakten Modellen analysieren können, bevor eine genauere Analyse am detaillierteren Modell erfolgt. Die gezeigten Beispiele analysieren die Auswirkungen von Fehlern bei der Kommunikation. Hierbei werden die Auswirkungen auf die Algorithmen analysiert, bevor der eigentliche Kommunikationsbus, der die Fehlerursache bildet, modelliert ist. Ähnlich verhält es sich mit abstrakten Zeitfehlern, die das Modell abschätzend analysiert, bevor das detaillierte Busmodell vorliegt, das die Injektion der eigentlichen Fehlerursache ermöglicht. Ergänzende Analysen mit fahrzeugtypischen Bussystemen, die zum Teil die vorgestellten Simulationsmodelle verwenden, sind in [19, 55, 96] vorgestellt.

7.2.1 Untersuchte Systemarchitekturen

Im Folgenden sind die unterschiedlichen Fallbeispiele vorgestellt, die zur Demonstration sowohl des Simulationsrahmenwerks als auch der Fehlerinjektion dienen. Das Fallbeispiel FB2 wird der Vollständigkeit hier nochmals vorgestellt, obwohl es als anschauliches Beispiel innerhalb dieser Arbeit bereits mehrfach verwendet wird.

7.2.1.1 FB1: MOST-Kommunikationsszenario

Das Fallbeispiel FB1 modelliert ein einfaches Kommunikationsszenario mit zwei Sendern und einem gemeinsamen Empfänger. Abbildung 7.2 zeigt die Struktur des Fallbeispiels. Im Fallbeispiel generieren zwei generische Datengeneratoren Daten

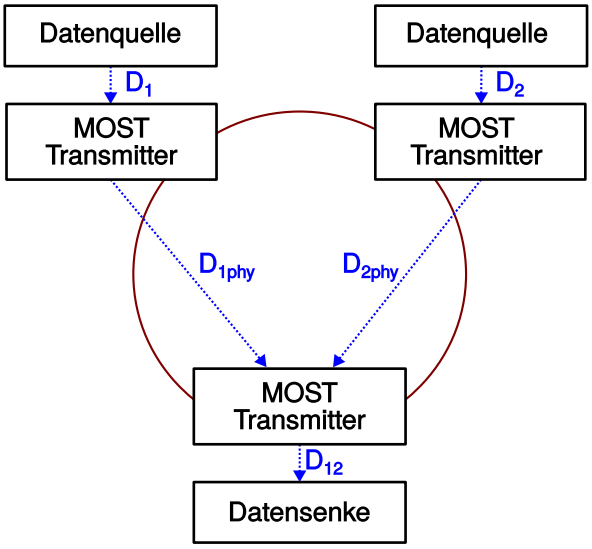


Abbildung 7.2: Struktur des Fallbeispiels FB1

(D_1, D_2) und senden diese zu einem gemeinsamen Empfänger. Hierbei kommen unterschiedliche Kommunikationsprotokolle zum Einsatz (D_{1phy}, D_{2phy}).

Das Ziel des Beispiels ist die Analyse und Optimierung der eingesetzten Kommunikationsprotokolle bzw. deren Parametrisierung. Aus diesem Grund ist es möglich Datengeneratoren anstelle konkreter Anwendungen zu verwenden. Diese Datengeneratoren ermöglichen es, unterschiedliche Datenmuster zu generieren. Als Gegenstelle wird eine Datensenke verwendet, die lediglich die Daten vom Netzwerk empfängt und Statistiken wie die Übertragungsdauer oder die Differenz zwischen Sendeanforderung und dem Empfang des letzten Pakets ausgibt. Dies wird erreicht, indem die Datengeneratoren zusätzliche Informationen in die generierten Daten codieren. Als Kommunikationsmedium wird der MOST-Bus beziehungsweise der asynchrone Kanal, verwendet. Der Kanal stellt einen geteilten Kommunikationskanal bereit, der unterschiedliche Netzwerkprotokolle unterstützt. Neben einem einfachen Telegrammprotokoll ohne jegliche Segmentierung der Daten gibt es das MOST-eigenen Protokoll MHP. Es stellt ein unidirektionales, verbindungsorientiertes Protokoll bereit. Dies ermöglicht die Segmentierung von Daten. Es ist sehr stark an das TCP

angelehnt, wobei es einen geringeren Datenmehraufwand hat. Neben dem MHP kann ebenfalls ein reguläres TCP verwendet werden. Aktuelle MOST-Entwicklungen präferieren die Verwendung generischer TCP-Implementierungen. Die MHP-Implementierung wird lediglich aus Kompatibilitätsgründen aufrechterhalten. Diese Entwicklung wird durch die Umbenennung des asynchronen Kanals in *MOST-Ethernet-Kanal* unterstrichen.

7.2.1.2 FB2: Füllstandsregelung

Das nächste in dieser Arbeit betrachtete Fallbeispiel wird mehrfach in dieser Arbeit als Beispiel zur Veranschaulichung herangezogen. Es untersucht die Füllstandsregelung eines Wassertanks. Das Fallbeispiel besteht wie in Abbildung 2.2 dargestellt aus dem Wassertank sowie einer Ansteuerung für den Wasserzufluss. Die Untersuchung fokussiert sich auf den Algorithmus zur Ansteuerung des Wasserzuflusses bzw. auf die Auswirkung von Fehlern bei der Berechnung und Übertragung von Steuersignalen. Hierzu führt ein RISC-Prozessor zwei unterschiedliche Algorithmen aus. Die Hysterese-Regelung definiert einen maximalen und minimalen Wasserstand. Bei Überschreiten bzw. Unterschreiten dieser Grenzen wird die Wasserzufuhr gestartet bzw. gestoppt. Der zweite Algorithmus verwendet einen PI-Regler, um den Wasserpegel auf einem vorgegebenen Niveau zu halten.

Sowohl das Signal zur Messung des aktuellen Füllstands als auch das Signal zur Steuerung der Pumpen wird über direkte Steuerleitungen realisiert. Zugriff auf die Steuersignale erfolgt durch die direkte Speicherzuweisung im Prozessor. Das heißt, über den Lese- bzw. Schreibzugriff auf dedizierte Speicherbereiche erfolgt die Kommunikation mit den Sensoren und Aktoren. Abbildung 7.3 zeigt die grobe Struktur der Fallstudie. Die Simulation besteht aus dem physikalischen Modell des Wassertranks sowie dem regelnden Prozessor. Der Prozessor modelliert einen einfachen MIPS-Prozessor, bestehend aus CPU, RAM sowie IO-Schnittstellen. In der Arbeit wird ein auf SystemC portiertes Simulationsmodell der Plasma CPU [85] verwendet.

7.2.1.3 FB3: Fahrerassistenzsystem

Diese Fallstudie bildet ein typisches Bordnetz sowie die beteiligten Steuergeräte (SGs) heutiger Automobile ab. Das System beinhaltet Instanzen eines MOST-Busses, eines CAN-Bussystems und eines FlexRay-Busses. Die realisierte Anwendung gliedert sich in eine Verkehrszeichenerkennung, die sich auf deutsche Geschwindigkeitsbegrenzungen bezieht sowie eine darauf aufbauende Berechnung von Fahrhinweisen. Die Fahrhinweise dienen der energieeffizienten Nutzung eines Elektrofahrzeuges [61, 62]. Der Algorithmus wird in dieser Arbeit lediglich bis zur Berechnung der detektierten Geschwindigkeitsänderung untersucht. Abbildung 7.4 stellt die Bordnetzarchitektur vor sowie die an den jeweiligen Bussen kommunizierende Steuergeräte. Das SG für das Human-Machine-Interface (HMI) implementiert die Benutzerschnittstelle und stellt dem Fahrer die aktuellen Geschwindigkeitsbegrenzungen bereit. Das Steuergerät wird über den MOST-Bus angesteuert, weil in bestimmten Konfigurationen nicht

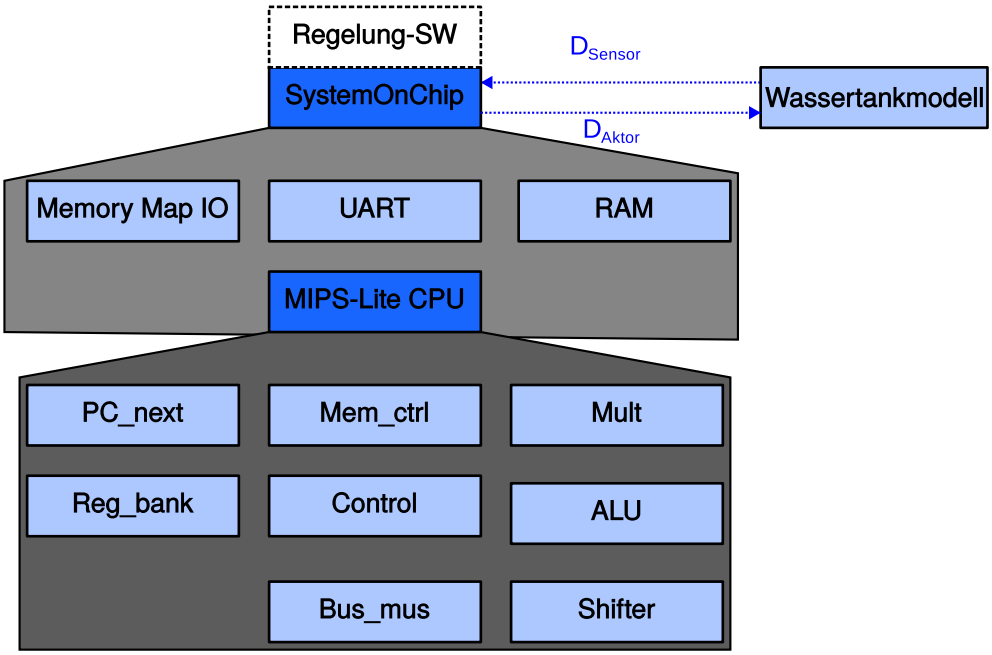


Abbildung 7.3: Architektur der Wassertankregelung

nur die Geschwindigkeitsbegrenzung, sondern auch das Kamerabild mit dem hervorgehobenen Verkehrszeichen angezeigt wird. Die Architektur des MOST-Busses ermöglicht eine 1:n Kommunikation. Die Frontkamera (*Kamera SG*) kommuniziert über den MOST-Bus da sich dieser bestens zur Übertragung des Videodatenstroms eignet. Der Bus bietet genügend Ressourcen zur Realisierung einer Stereo-Bild-Übertragung [55]. Die Verwendung von Stereobildern zur Schätzung der Entfernung eines Schilds ist im Allgemeinen genauer als die verwendete Abschätzung, die lediglich auf dem Durchmesser des Schilds beruht. Die Alternative mit der Stereokamera wird in der Fallstudie, jedoch nicht berücksichtigt. Das Steuergerät *Kreiserkennung* empfängt die Bilder der Frontkamera und führt eine Kreiserkennung durch. Das Steuergerät schneidet die erkannten Kreise aus, skaliert die Ausschnitte und sendet die skalierten Ausschnitte an das Steuergerät *Verkehrszeichenklassifikation*. Hierfür wird der FlexRay-Bus verwendet. Das Steuergerät *Verkehrszeichenklassifikation* empfängt die Bildausschnitte und versucht über eine Support Vector Machine (SVM) die Bildausschnitte einer Geschwindigkeitsbeschränkung zuzuordnen. Das Steuergerät versendet die klassifizierten Geschwindigkeitsbeschränkungen über das CAN. Das HMI-Steuergerät zeigt die Geschwindigkeitsbeschränkungen dem Fahrer an. Neben der Fahrerunterstützung durch Anzeige der aktuellen Geschwindigkeitsbegrenzung dienen die Informationen zusätzlich zur Bestimmung einer energieeffizienten Fahrstrategie. Hierzu versendet das Steuergerät *Kreiserkennung* den detektierten Kreisdurchmesser, kombiniert mit dem Zeitpunkt der Bildaufnahme. Das CAN leitet die Daten weiter und stellt sie dem *Coasting Assistant* zur Verfügung. Neben diesen Informationen empfängt der *Coasting Assistant* zusätzlich die klassifizierte Geschwindigkeitsbegrenzung, von dem Steuergerät *Verkehrszeichenklassifikation* sowie die aktuelle Fahrzeuggeschwindigkeit. Mit

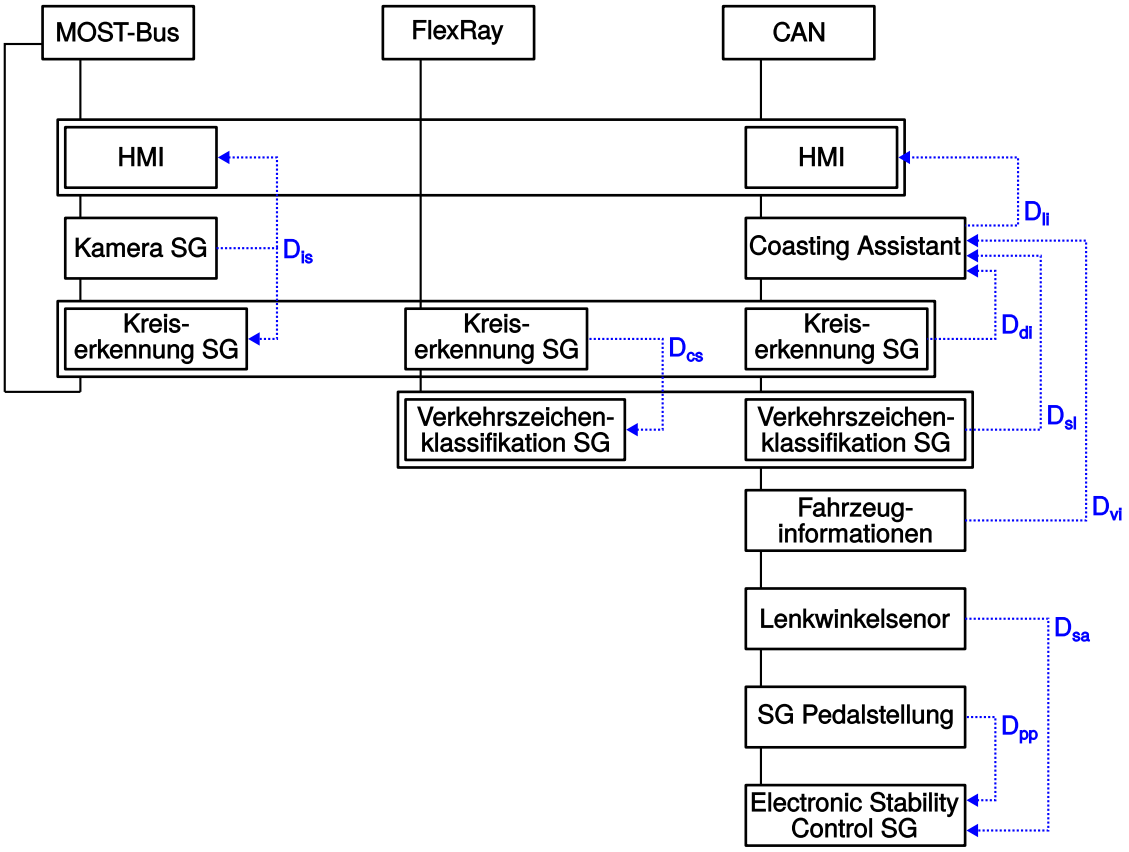


Abbildung 7.4: Busarchitektur des Fallbeispiels FB3

diesen Informationen ist es möglich, eine energieeffiziente Beschleunigungs- bzw. Verzögerungsstrategie zu wählen. Im Falle der Verzögerung existieren unterschiedliche Strategien zum regenerativen Bremsen. Der Anwendungsfall erfasst zusätzlich Drive-By-Wire-Informationen wie die Pedalstellung sowie den Lenkradeinschlag und sendet sie über das CAN. Das Steuergerät *Electronic Stability Control* empfängt die Informationen und leitet sie nach entsprechender Verarbeitung an den Antriebsstrang weiter. Im betrachteten Fallbeispiel wird die vom Fahrzeugführer getroffene Fahrstrategie nicht durch das System beeinflusst. Es werden lediglich dem Fahrer Strategieempfehlungen angezeigt. Durch zusammenführen der Steuersignale der Pedalstellung sowie der energieeffizienten Fahrempfehlungen, wäre aber auch ein aktiver Eingriff in die umgesetzte Fahrstrategie vorstellbar. Dieses Szenario wurde im betrachteten Fallbeispiel nicht ausgewertet, verdeutlicht aber den sicherheitsrelevanten Charakter des Fallbeispiels. Abbildung 7.1 gibt einen Überblick über alle ausgetauschten Informationen.

ID	Name	Beschreibung
D_{is}	ImageStream	Aufzeichnung der Straße vor dem Fahrzeug. Standardmäßig eine Rastergrafik (RGB) mit 720x576 Bildpunkten.
D_{sl}	SpeedLimit	Die klassifizierte Geschwindigkeitsbegrenzung als Datenwert (4 Byte)
D_{cs}	CircleSegment	Das ausgeschnittene Bildsegment, das den Kreis enthält. Standardmäßig eine Rastergrafik (RGB) mit 50x50 Bildpunkten.
D_{pp}	PedalPosition	Ein Tupel aus den Pedalstellungen für Bremse und Gas.
D_{sa}	SteeringAngle	Datenwert zur Stellung des Lenkrads.
D_{li}	LimtInformation	Nächste Geschwindigkeitsbegrenzung mit Abstand, Fahrempfehlung (brake, coast, accelerate) sowie das aktuelle Drehmoment.
D_{di}	DistanceInformation	Informationen zur Schätzung der Entfernung des Schilds. Beinhalten Schilddurchmesser und Zeitstempel.
D_{vi}	VehicleInformation	Fahrzeuginformationen wie die aktuelle Geschwindigkeit und Drehmoment.

Tabelle 7.1: Ausgetauschte Daten im ADAS-Anwendungsfall

7.2.2 Performanzbewertung am FB1

Am Fallbeispiel FB1 wird die Verwendung von virtuellen Prototypen zur Performanzanalyse präsentiert. Diese Art der Analysen braucht häufig eine umfassende Datenbasis um statistische Aussagen über die durchschnittliche, maximale oder minimale Leistungsfähigkeit des Systems treffen zu können. Virtuelle Prototypen bieten durch die Möglichkeit zur einfachen Wiederholung und Duplikation von Experimenten einen vielversprechenden Ansatz zur Durchführung der Analysen. Experimente bleiben auf der einen Seite durch das deterministische Verhalten wiederholbar, auf der anderen Seite erlauben Pseudozufallszahlengeneratoren mit unterschiedlichen Startwerten (engl. seed) die Einbeziehung von zufälligem Verhalten. Durch diese Zufälligkeit sind quantitative Analysen möglich. Der Konfigurationsansatz ermöglicht die effiziente Variation von Parametern und somit die einfache Erzeugung einer umfassenden Datenbasis.

Im ersten Szenario wird die durchschnittliche Übertragungsdauer eines Datenpakets mit unterschiedlichen Konfigurationen des MHPs untersucht. Eine Voruntersuchung zeigte, dass das physikalische Übertragungsmedium, bei regulärer Konfiguration, die Gesamtübertragungsdauer wenig beeinflusst. Um die Simulationsperformanz zu steigern, wird deswegen für die Untersuchung ein Simulationsaufbau bestehend aus einem Datengenerator, einer MHP-Implementierung und einem TLM-

Nachrichtenpuffer verwendet. Das heißt, es wird auf die Modellierung des physikalischen MOST-Rings verzichtet. Abbildung 7.5 zeigt die Struktur des Simulationsmodells. Das Szenario besteht aus zwei Busteilnehmern, die über das MHP Daten auf

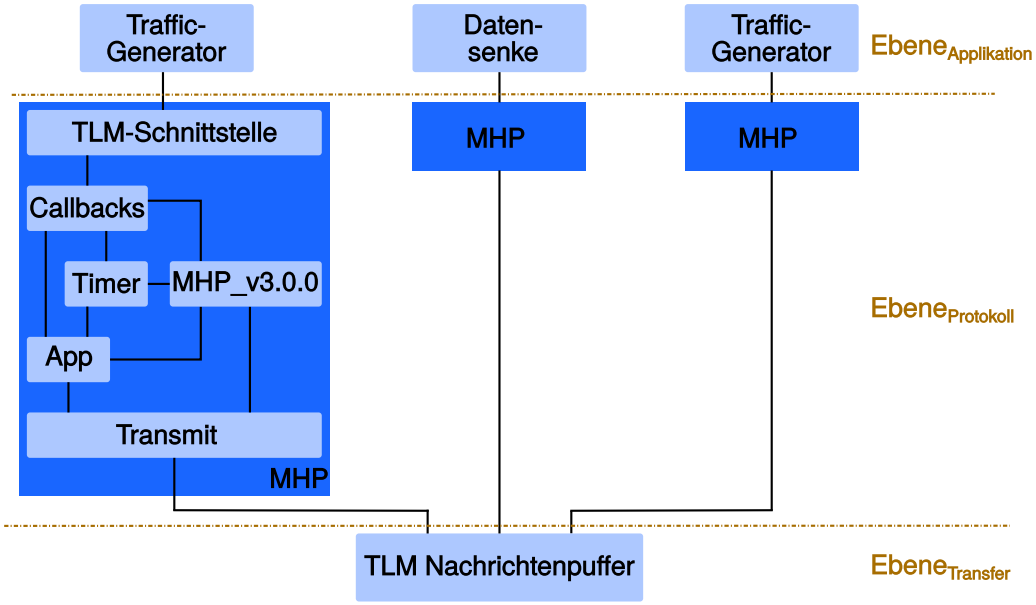


Abbildung 7.5: Struktur des MHP-Simulationsmodells

einen gemeinsamen Empfänger senden. Bei der MHP-Implementierung handelt es sich um die ursprüngliche ANSI-C-Implementierung, die mittels instanziierbaren Simulationseinheiten gekapselt wird. Die Wiederverwendung der ursprünglichen Implementierung reduziert zum einen den Modellierungsaufwand und steigert zum andern die Korrelation zwischen Modell und realem System. Sowohl die Datengeneratoren als auch die Datensenke bestehen aus einer einzigen BSE. Die Datengeneratoren bieten Parameter zur Konfiguration der Größe der zu sendenden Daten, der Segmentgröße sowie der Sendeperiode an. Abbildung 7.6 zeigt einen Auszug aus der Konfigurationsdatei. Aufgrund der Tatsache, dass die Konfiguration dynamisch geladen und interpretiert wird, ist es möglich unterschiedliche Szenarien zu simulieren, ohne die Simulationseinheitenbibliothek neu zu erstellen. Testgeneratoren, welche die IP-XACT-Datei erzeugen, reichen aus, um eine quantitative Untersuchung des Entwurfsraums durchzuführen.

Das MHP-Modell setzt sich aus mehreren Simulationseinheiten zusammen. Die Implementierung des MHPs besitzt applikationsspezifische Funktionen, die der Anwendung als Dienstschnittstelle zur Verfügung stehen. Die Schnittstelle bietet Funktionen zum Aufbau der Verbindung, zum Senden eines Datenarrays oder zum Empfangen an. Eine TLM-Schnittstelle kapselt diese Funktionen zur Integration in die Makroarchitektur. Der Anwendung wird anstelle von applikationsspezifischen Funktionen eine TLM-Schnittstelle bereitgestellt. Die TLM-Schnittstelle ist notwendig, da die Anwendung und das Protokoll über die Schichtengrenzen der Makroarchitektur ($Ebene_{Applikation}$ und $Ebene_{Protokoll}$) hinweg kommunizieren. Ähnliches trifft auf

```
...
  <ipxact:componentInstance>
    <ipxact:instanceName>TrafficGenOnceN00</ipxact:instanceName>
    <ipxact:componentRef library="Library" name="M_TrafficGenOnce"
                         vendor="Library" version="0.1">
      <ipxact:configurableElementValues>
        <ipxact:configurableElementValue referenceId="m_tMinStartTimeUS">
          15000</ipxact:configurableElementValue>
        <ipxact:configurableElementValue referenceId="m_tMaxStartTimeUS">
          15000</ipxact:configurableElementValue>
        <ipxact:configurableElementValue referenceId="m_unSegmentLength">
          65535</ipxact:configurableElementValue>
        <ipxact:configurableElementValue referenceId="m_tRetryDurationUS">
          100.0</ipxact:configurableElementValue>
        <ipxact:configurableElementValue referenceId="m_unDestDevAddress">
          701</ipxact:configurableElementValue>
        <ipxact:configurableElementValue referenceId="m_unMaxDataLength">
          10485760</ipxact:configurableElementValue>
        <ipxact:configurableElementValue referenceId="m_unMinDataLength">
          10485760</ipxact:configurableElementValue>
        <ipxact:configurableElementValue referenceId="m_tSendPeriodUS">
          8800</ipxact:configurableElementValue>
      </ipxact:configurableElementValues>
    </ipxact:componentRef>
    <ipxact:vendorExtensions>
      <ipxact:file>
        <ipxact:name>M_TrafficGenOnce.xml</ipxact:name>
        <ipxact:fileType user="xml">user</ipxact:fileType>
      </ipxact:file>
    </ipxact:vendorExtensions>
  </ipxact:componentInstance>
...
```

Abbildung 7.6: Konfiguration der Datenquelle mittels IP-XACT

die BSE `Transmit` zu, die eine TLM-Schnittstelle, zum Senden der Daten, benötigt. Durch diese Struktur ist es möglich unterschiedliche Protokolle zu verwenden, ohne die Datengeneratoren anpassen zu müssen. Im zweiten Teil der Fallstudie wird anstelle des MHPs ein TCP verwendet. Um die Austauschbarkeit zu ermöglichen, wird die logische Adresse anstelle der physikalischen Adresse verwendet. Abbildung 7.7 zeigt die Übersetzungstabelle der logischen auf die physikalischen Adressen. Hierbei wird eine logische Adresse in eine MHP-Adresse übersetzt. Der Eintrag `700 | 256.161.0.1040.0` bildet die logische Adresse 700 auf die physikalische MHP-Adresse ab. Die MHP-Adresse besteht aus der Geräteadresse (256), der Funktionsblock-ID (161), einer Instanz-ID (0), einer Funktions-ID (1040) sowie einer Operator-ID (0). Wird der Datengenerator mit einem TCP verwendet, muss die Protokollimplementierung eine Übersetzungstabelle mit IP-Adresse und Port bereitstellen. Im zweiten Teil wird gezeigt, wie das MHP durch das TCP ersetzt wird.

Die BSE `MHP_v3.0.0` kapselt die ursprüngliche Implementierung des Protokolls. Die anderen BSEs stellen die benötigten Hilfsfunktionen zur Verfügung. Das MHP bietet unterschiedliche Parameter, welche die Performanz beeinflussen. Hierzu zählen z. B. die maximale Anzahl simultaner Verbindungen, die Paketgröße oder die Timer-Einstellungen. Des Weiteren ermöglicht, die modulare Struktur der Simulationsmodelle unterschiedliche Versionen des ursprünglichen Quelltextes vorzuhalten

```
...
  <ipxact:componentInstance>
    <ipxact:instanceName>MhpTLMInterface00</ipxact:instanceName>
    <ipxact:componentRef library="Library" name="M_MhpTLMInterface"
                         vendor="Library" version="0.1">
      <ipxact:configurableElementValues>
        ...
        <ipxact:configurableElementValue referenceId="m_strAppTransTab">
          700 | 256.161.0.1040.0</ipxact:configurableElementValue>
        <ipxact:configurableElementValue referenceId="m_strAppTransTab">
          701 | 258.161.0.1040.0</ipxact:configurableElementValue>
        <ipxact:configurableElementValue referenceId="m_strAppTransTab">
          702 | 259.161.0.1040.0</ipxact:configurableElementValue>
      </ipxact:configurableElementValues>
    </ipxact:componentRef>
    ...
  </ipxact:componentInstance>
...
```

Abbildung 7.7: Logische Adresstabelle des MHPs

und einfach per Konfigurationsdatei auszuwählen. Die Infrastruktur zur Ausführung des Protokolls, wie z. B. Zeitgeber oder Callback-Funktionen, wird aufgrund der standardisierten Schnittstellen für unterschiedliche Protokollversionen verwendet.

Die exemplarische Analyse untersucht die resultierende Datenübertragungsdauer in Abhängigkeit der beiden Senderperioden. Hierbei ist zu erkennen, dass mit einer Verminderung der Periode die Übertragungsdauer sinkt. Ab einem gewissen Punkt tritt aber ein Überlastszenario ein, was zum Verwerfen von Nachrichten führt. Das MHP ist ein zuverlässiges Protokoll, das versucht, diesen Nachrichtenverlust durch Sendewiederholungen auszugleichen. In diesem Fall nimmt die gesamte Übertragungsdauer zu. Das heißt, die optimale Sendeperformanz befindet sich nicht an den Rändern der erlaubten Parameterbereiche. Die optimierte Einstellung lässt sich mithilfe von virtuellen Messungen und die Bildung von Durchschnittswerten annähern. Abbildung 7.8 zeigt das beobachtete Verhalten der durchgeführten Untersuchung. Auf der x- bzw. y-Achse sind die Werte für die unterschiedlichen Senderperioden aufgetragen. Auf der z-Achse die resultierende Übertragungsdauer. Es ist zu erkennen, dass durch ein Erhöhen der Periode die Übertragungsdauer zunimmt. Die maximale Übertragungsdauer ist im Punkt (9000,9000), dargestellt durch die gelbe Färbung in diesem Punkt. Aber auch an den unteren Rändern der Perioden ist ein Anstieg der Übertragungsdauer zu erkennen. Die Analyse zeigt, dass die niedrigste Übertragungsdauer bei einer Sendeperiode von ca. 2000 us erreicht wird.

In einem ähnlichen Anwendungsfall wird die Wechselwirkung mehrerer TCP/IP-Verbindungen über den MOST-Ethernet-Kanal untersucht. Auch hier sind unterschiedliche Parametrisierungen möglich, welche die Übertragungsperformanz beeinflussen. Im Analyseszenario werden zwei TCP-Verbindungen parallel zu zwei existierenden UDP-Verbindungen aufgebaut. Das Szenario simuliert die Auswirkungen von zwei Internetverbindungen (TCP) bei zwei bestehenden Voice over IP (VoIP)-Kanälen (User Datagram Protocol (UDP)). Die Analyse verwendet die Datengeneratoren aus dem vorherigen Beispiel, lediglich die Parameter werden auf die Verbin-

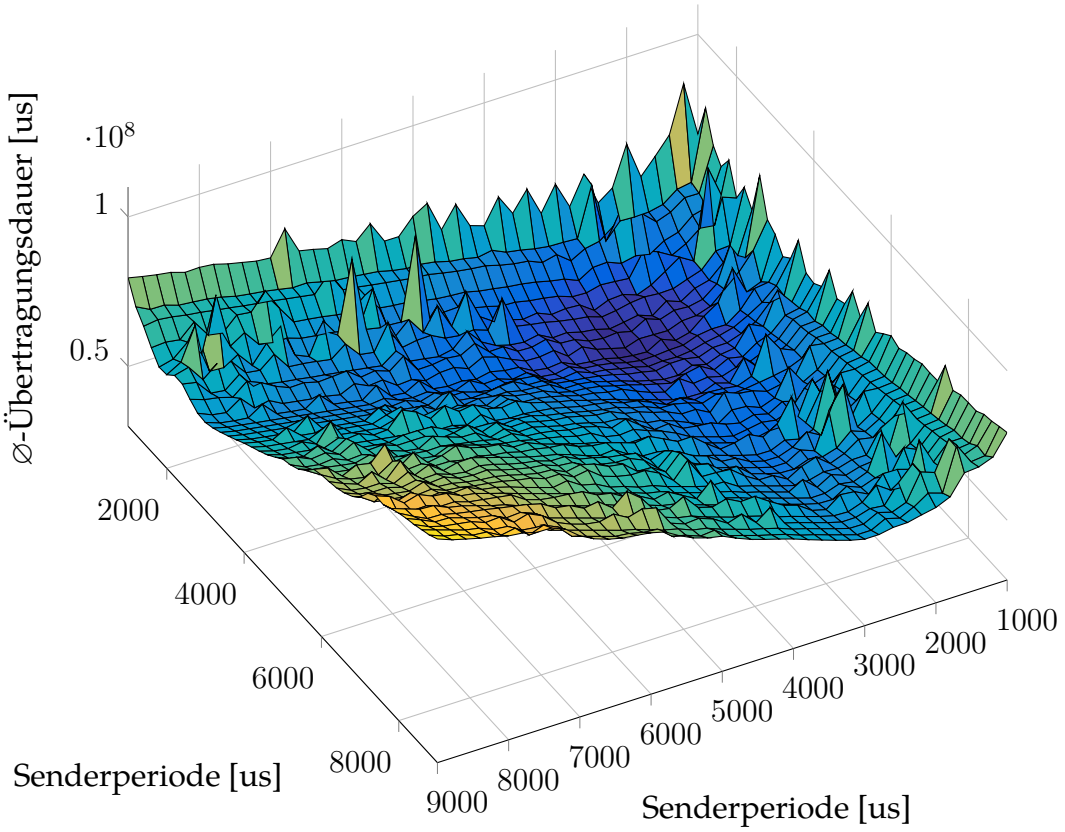


Abbildung 7.8: Übertragungsdauer im untersuchten MHP-Szenario

dungscharakteristiken angepasst. Anstelle von dem MHP wird der TCP/IP-Stack lwIP [33] verwendet. Der originale Quellcode der Bibliothek wird in mehrfach instanziierbarer Art gekapselt. Hierzu wird die Funktionalität des ANSI-C-Quellcodes in C++ eingebunden und eine Kontextklasse erstellt, die alle objektspezifischen Informationen beinhaltet. Beim Aufruf von Stack-Funktionen wird die Kontextklasse mit übergeben. Benötigt die Funktionen objektspezifische Informationen, greift sie auf den übergebenen Ausführungskontext zu. Auf diese Weise ist eine mehrfache Instanziierung der Stack-Funktionalität durch Vorhalten mehrerer Kontexte möglich. Als Übertragungsmedium wird diesmal ein detaillierteres MOST-Modell verwendet. Abbildung 7.9 zeigt das beobachtete Systemverhalten. Hierbei wird verdeutlicht, ob die beiden TCP-Verbindungen sich die Bandbreite gerecht teilen und welche Auswirkungen zusätzliche Verbindungen auf die VoIP-Kanäle haben.

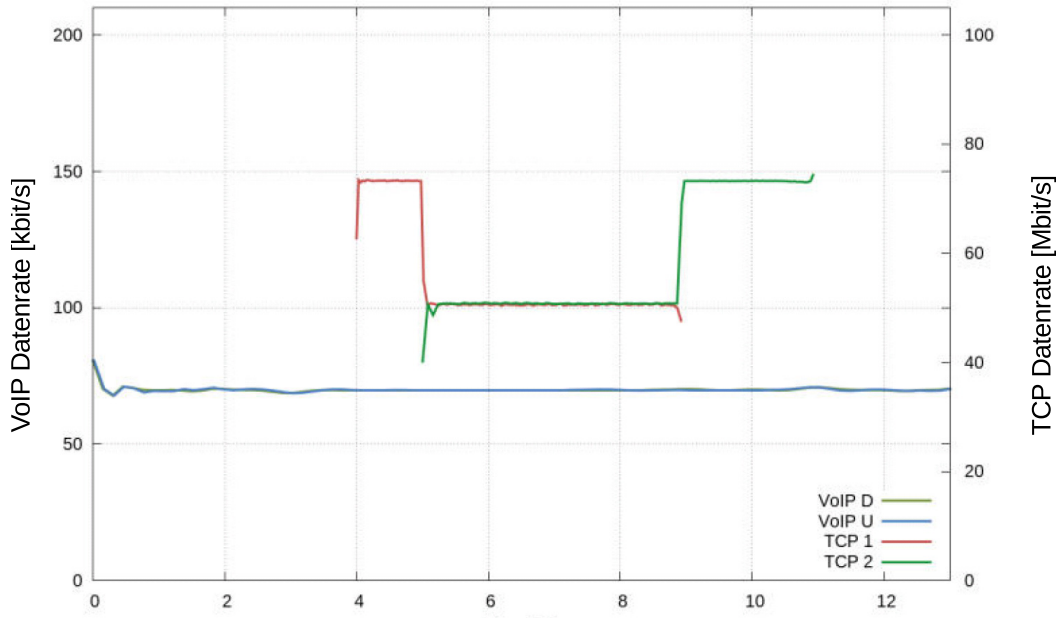


Abbildung 7.9: Simulationsergebnis im TCP-UDP-Szenario

7.2.3 Softwarefehlereffekte am FB2

Das Fallbeispiel der Wassertankregelung wird in dieser Arbeit durchgängig als Anschauungsbeispiel verwendet. In Abbildung 2.2 wird das physikalische Modell der Regelung vorgestellt und in Abbildung 2.3 das Verhalten bei einer diskreten Hysterese-Regelung. Ein Auszug aus der IP-XACT-Beschreibung des physikalischen Modells, mit den bereitgestellten Parametern, ist in Abbildung 5.9 und Abbildung 5.10 dargestellt. Die IP-XACT-Spezifikation der Instanziierung der BSEs ist in Abbildung 5.8 gezeigt. Die grafische Spezifikation der Simulationsmodelle wird in der Abbildung 6.4 sowie Abbildung 6.5 dargelegt. Hierbei handelt es sich um ein vereinfachtes Modell, das die Regelalgorithmen direkt als Host-Code einbindet, ohne die Interpretation durch einen Instruction Set Simulator (ISS). Diese Beispiele geben bereits einen guten Überblick über die Fehlereffektsimulation in dem Fallbeispiel. Im folgenden Abschnitt werden zusätzliche Fehlerfälle und die benötigten Anpassungen vorgestellt.

In den bisherigen Fällen verwenden das Simulationsmodell der physikalischen Abstraktion zur Fehlerinjektion. Der Fehler wird hierbei in das Sensorsignal des Wassertanks injiziert. Diese Erweiterung ist in Abbildung 5.11 dargestellt. Abbildung 5.22 zeigt hingegen die Spezifikation eines zeitgesteuerten, transienten Fehlers. Weitere Fehlereffekte auf die Hysterese-Regelung zeigt Abbildung 5.21. Der bisher betrachtete Fehler befindet sich direkt im Wassertankmodell und verändert den erfassten Sensorwert. Hierbei wird eine Variable vom Datentype `double` verändert. In der grafischen Spezifikation, dargestellt in Abbildung 6.5, wird ein weiterer transienter Fehler spezifiziert, der den Sensorwert reduziert und somit zu einem erhöhten Wasserstand führt.

Im Folgenden wird durch die Portierung der Regelalgorithmen auf das Microprocessor without interlocked pipeline stages (MIPS)-Prozessormodell, die Auswirkungen von Hardwarefehlern auf die Software dargelegt. Diese Ausprägung des Fallbeispiels verdeutlicht den Zusammenhang zwischen Hardware- und Softwarefehler. Soft Errors treten in Hardwarekomponenten, wie Speicher oder Logikschaltungen auf. Die Software verwendet diese Ressourcen zum Speichern von Daten oder zur Durchführung von Berechnungen. Hierdurch wirken sich die auftretenden Fehler auf die Software aus. Wird bei den Simulationsmodellen die Hardware nicht explizit modelliert, ist es möglich die Fehlereffekte der Hardware direkt in die Software zu injizieren, wie z. B. bei der Veränderung des Sensorwerts. Hierbei spielt die Mächtigkeit der Fehlerbeschreibung eine wichtige Rolle. Bei der expliziten Modellierung der Hardware können Fehler, meist spezifiziert durch die traditionellen, Hardwarefehlermodelle, direkt in die Hardware injiziert werden und die Effekte auf die Software beobachtet werden. Bei Softwarefehler entstehen hingegen häufig komplexere Abhängigkeiten zwischen Fehlermodell und Simulation.

Um die Auswirkungen von Hardwarefehlern zu untersuchen, wird ein Prozessormodell eines MIPS-Prozessors verwendet. Die Steueralgorithmen, d. h. der Hysterese-Regler und der PI-Regler werden kompiliert auf dem Modell zur Ausführung gebracht. Die Kommunikation mit dem physikalischen Modell, respektive den Sensoren und Aktoren des Wassertanks, erfolgt über eine Memory-Mapped-Kommunikation. In diesem Fall liest bzw. schreibt der Prozessor spezielle Speicherbereiche, die durch eine IO-Schnittstelle mit dem Wassertankmodell interagieren. Mit diesem Systemaufbau ist es möglich, Fehler in die Hardware einzustreuen, und die Effekte auf die Software zu beobachten. Im Speziellen werden die Auswirkungen einer Verfälschung der Register untersucht. Hierbei wird das Register, das den aktuellen Sensorwert zur Berechnung zwischenspeichert, verfälscht. Abbildung 7.10 zeigt das verwendete BTM. Hierbei wird der Fehler in das 16. Register injiziert. Dies entspricht,

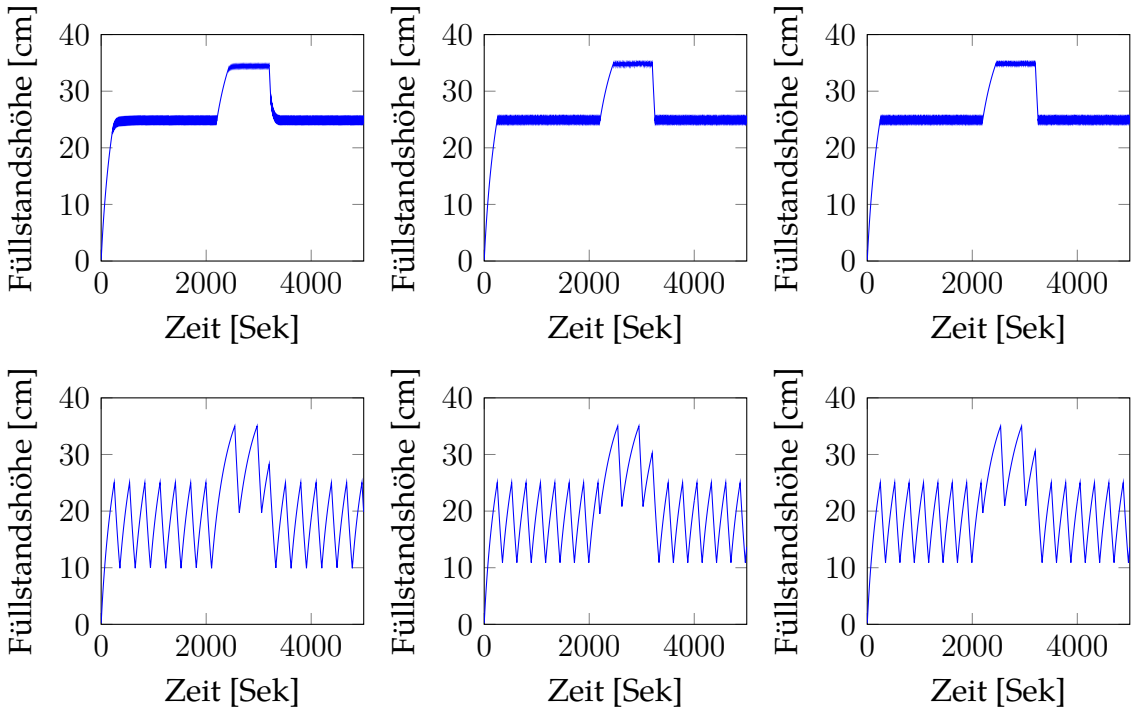
```
1  <ESname> errFree <EStype> initial
2  <ESname> errState
3  <ETsrc> errFree <ETtgt> errState
4    <ETguard> BTMclock('OKTime').read() == sc_time(2200, sc_time_unit.SC_SEC)
5    <ETaction> VPvar('triPortRam')[16].force(VPvar('triPortRam')[16].read()-10)
6    <ETupdate> BTMclock('ERRTime').reset()
7  <ETsrc> errState <ETtgt> errState
8    <ETguard> BTMclock('ERRTime').read() < sc_time(1000, sc_time_unit.SC_SEC)
9    <ETaction> VPvar('triPortRam')[16].force(VPvar('triPortRam')[16].read()-10)
10 <ETsrc> errState <ETtgt> errFree
11   <ETguard> BTMclock('ERRTime').read() >= sc_time(1000, sc_time_unit.SC_SEC)
12   <ETaction> VPvar('triPortRam')[16].release()
```

Abbildung 7.10: BTM der Registerverfälschung im Wassertankbeispiel

aus der Softwaresicht, dem logischen Register $s0$, das zum Speichern von Werten, über Funktionsaufrufe hinweg, verwendet wird.

Dieser Abschnitt stellte zwei Fehlerszenarien mit unterschiedlich abstrakten Modellen vor, die ähnliche Fehlereffekte hervorrufen. Die Injektion in den Sensorwert direkt im Wassertankmodell, der in der bisherigen Arbeit als Beispiel diente und

die Injektion in die Prozessorarchitektur. Als Brückenschlag wird zusätzlich das detailliertere Prozessormodell mit der Fehlerinjektion in den Sensorwert untersucht. Abbildung 7.11 zeigt das beobachtete Fehlverhalten in den drei untersuchten Fällen.



(a) Sensorfehler Host-Code-Regelung

(b) Sensorfehler SoC-Regelung

(c) Registerfehler SoC-Regelung

Abbildung 7.11: Fehlereffekte in unterschiedlichen Systemsimulationen

In der linken Spalte ist die bereits vorgestellte Injektion, aus Abbildung 5.21, dargestellt, wobei die Parametrisierung des Fehlers unterschiedlich ist. Der Fehler wird direkt in den vom Wassertank erfassten Sensorwert injiziert und dem Regelalgorithmus zur Verfügung gestellt. Hierbei werden die Effekte mit Steueralgorithmen die auf einer Hysterese-Regelung als auch auf einer PI-Regelung aufbauen, untersucht. Die verwendete Fehlerinjektion führt dazu, dass die Regelung den Füllstand auf ein höheres Niveau regelt. Der Fehler tritt nach 2200 Sekunden auf, wie in Abbildung 7.10 zu entnehmen ist. Die Reduktion des erfassten Sensorwerts führt dazu, dass der Füllstand des Wassertanks höher geregelt wird. Bei der PI-Regelung ist die durch den höheren Wert zu erkennen und in der Hysterese-Regelung durch ein Verschieben der Hysterese-Regelung auf ein höheres Niveau. Bei dem untersuchten Fehler handelt es sich um einen transienten Fehler, der nur 1000 Sekunden andauert. Danach verrichtet die Regelung ihren Dienst wie gewohnt.

In der mittleren Spalte wird die gleiche Fehlerinjektion verwendet aber anstelle den Regelalgorithmus als Host-Code auszuführen, wird die kompilierte Version auf dem SoC-Modell ausgeführt. Es ist zu erkennen, dass es zu leichten Verschiebungen über die Zeit kommt. Gerade am Beispiel der Hysterese-Regelung ist zu erkennen,

dass der Fehler bei abnehmenden Füllstand injiziert wird, wobei beim abstrakten Modell im Zustand des zunehmenden Füllstands injiziert wurde, vergleiche hierzu die rechte Spalte. Dies liegt daran, dass der Prozessor ein leicht abweichendes Zeitverhalten aufweist und somit zu anderen Zeiten die Abtastung des Sensorwerts vornimmt. Des Weiteren unterscheiden sich die Flanken bei der Fehlerinjektion. Der Zeitpunkt der Fehlerinjektion und Fehlerbehebung als auch die Schwere des Fehlers sind unverändert. Trotz der leichten zeitlichen Unterschiede ist zu erkennen, dass die abstrahierte Simulation, d. h. ohne die konkrete Simulation des Prozessors, bereits eine sehr gute Modellierung des Verhaltens darstellt. Zum Beispiel ist die Schwere des Fehlers, d. h. seine Übersteuerung, in beiden Fällen die Gleiche. Anzumerken gilt, dass die Host-Code-Ausführung der Algorithmen eine erhebliche Reduktion des Simulationsaufwands mit sich bringt.

In der rechten Spalte wird der Fehler in die Registerbank des Prozessors injiziert. Hierbei wird lediglich das Register, das den Sensorwert speichert, verändert. Wie die Abbildung veranschaulicht, sind die Fehlereffekte sehr ähnlich zu dem SoC-Modell. Dies liegt daran, dass das gleiche SoC-Modell verwendet wird und somit das gleiche Zeitverhalten vorliegt. Der einzige Unterschied liegt in der Manifestation des Fehlers im Register. Bei der Injektion in das Sensorsignal wird der Fehler erst mit dem nächsten Taktzyklus übernommen. Bei der Injektion direkt in das Register, wird der Fehlerzustand sofort mit der Injektion übernommen. Da die Auswertung des Registerinhalts mit dem regulären Lesezugriff stattfindet, sind die Auswirkungen für diesen Fehlerfall die Gleichen. Den Zustand zwischen Injizieren des Fehlers und des Lesens des verfälschten Werts wird verborgener Fehlerzustand genannt. Die Ähnlichkeit der Fehlereffekte demonstriert, dass die Evaluation mit frühen Simulationsmodellen bereits eine Aussage über die zu erwartenden Fehlereffekte zulässt. Im untersuchten Fall wird das Register, das den Berechnungswert zwischenspeichert, gesondert betrachtet. Die randomisierte Fehlerinjektion in die komplette Registerbank veränderte zusätzlich sensible Steuerregister, was vorrangig zu einem Aussetzen des Regelalgorithmus führte. In diesem Fall füllt sich der Wassertank bis zu der oberen Sättigungsgrenze, die sich aus dem steigenden Wasserabfluss bei steigendem Pegel ergibt.

In den bisher betrachteten Szenarien ist die Fehlerinjektion in den Sensorwert durch eine C++-Modellierungsprimitive als auch in die Registerbank durch eine aggregierte C++-Modellierungsprimitive realisiert. Die Erweiterungen zur Fehlerinjektion in diesen beiden Fällen sind Bestandteil der generischen Werkzeugumgebung. Im Folgenden wird die Injektion in eine applikationsspezifische Datenstruktur vorgestellt.

Neben der Übertragung der Sensorsignale mit dedizierten Steuerleitungen ist es möglich, die Kommunikation auf ein Bussystem abzubilden. Abbildung 5.4 zeigt die Steuerungssoftware mit einem CAN-Modell. Dieses Modell simuliert die Datenübertragung mit Hilfe einer applikationsspezifischen CAN-Datenstruktur. Um Fehler in diese Struktur zu injizieren, benötigt der Python-Interpreter Informationen über die Datenstruktur. Abbildung 7.12 zeigt die benötigten Deklarationen zur Bekanntgabe der CAN-Datenstruktur. Vorrangig bildet die Deklaration die Einträge der CAN-

```
BOOST_PYTHON_MODULE(CANFrame)
{
  boost::python::class_<CANFrame>("CANFrame")
    .def_readwrite("unID", &CANFrame::unID)
    .def_readwrite("unRTR", &CANFrame::unRTR)
    .def_readwrite("unIDE", &CANFrame::unIDE)
    .def_readwrite("unRsvd", &CANFrame::unRsvd)
    .def_readwrite("unDLC", &CANFrame::unDLC)
    .add_property("unData", make_array(&CANFrame::unData))
    .def_readwrite("unCRC", &CANFrame::unCRC)
    .def_readwrite("unACK", &CANFrame::unACK)
  ;
}
```

Abbildung 7.12: Veröffentlichung der CAN-Datenstruktur in Python

Datenstruktur auf ein Python-Modul ab. Hierbei werden dem Python-Interpreter sowohl Lese- als auch Schreibzugriff gewährt. Zur Modifikation der Nutzdaten wird das Datenarray abgebildet. Die verwendete Bibliothek stellt wohldefinierte Makros bereit, um die Abbildung vorzunehmen.

Zusätzlich muss der Anwender die Deklaration bei dem Python-Interpreter registrieren. Hierzu bietet das Framework wie beschrieben eine Registrierungsfunktion an. Abbildung 7.13 zeigt den benötigten Aufruf. Die Funktion `initCANFrame` wird

```
C_PythonSingleton::instance().registerPythonExtension(
                    &initCANFrame, "from CANFrame import *\n");
```

Abbildung 7.13: Registrierung und Initialisierung der CAN-Datenstruktur

automatisch bei der Definition des Boost-Python-Moduls in Abbildung 7.12 generiert. Das heißt, hier ist kein weiteres Eingreifen des Anwenders notwendig. Je nachdem wie die Einbindung in Python gewünscht wird, kann der Anwender unterschiedliche import-Anweisungen angegeben. Im Beispiel wird das gesamte Modul in den globalen Namensraum geladen. Durch die beiden Erweiterungen steht der Datentyp des CAN-Frames zur Fehlerinjektion zur Verfügung und die Fehlerspezifikation kann auf einzelne Elemente der Struktur zugreifen. Abbildung 7.14 zeigt einen Auszug der Fehlerspezifikation zur Injektion in die übertragenen Nutzdaten. Als Auslöser des

```
...
<ETsrc> eFree <ETtgt> eState
  <ETguard> VPvar('m_SharedCANBusphyCANInject').read().unID == 0xAF
 <ETaction> forceValue = VPvar('m_SharedCANBusphyCANInject').read();
            forceValue.unData[0] = forceValue.unData[0]-10;
            VPvar('m_SharedCANBusphyCANInject').force(forceValue);
...
```

Abbildung 7.14: Fehlerinjektion in eine CAN-Datenstruktur

Fehlers wird die Übertragung eines speziellen CAN-Frames verwendet. Hierbei wird

auf die CAN-ID der Datenstruktur zugegriffen. Bei Vorliegen dieses Frames werden die Nutzdaten modifiziert. Hierbei wird das niederwertigste Byte der Nutzdaten um 10 dekrementiert. Der durch die Injektion hervorgerufene Fehlereffekt unterscheidet sich von der zuvor betrachteten, transienten Übersteuerung, da der CAN-Controller den Fehler mittels eines Cyclic Redundancy Check (CRC) detektiert und das Frame verwirft. Dieses Beispiel zeigt, wie auf die Daten einer applikationsspezifischen Datenstruktur zugegriffen wird. Neben den eigentlichen Nutzdaten kann die Fehlerinjektion alle freigegebenen Informationen der Struktur verfälschen.

```
<ESname> eInit <EStype> initial
<ESname> eFree
<ESname> eState
<ETsrc> eInit <ETtgt> eFree
  <ETguard> BTMclock('OKTime').read() == sc_time(2, sc_time_unit.SC_SEC)
  <ETaction> injVal = VPvar('m_TLMAdapterTSideTLMSendPayloadInj').read()
<ETsrc> eFree <ETtgt> eState
  <ETguard> BTMclock('OKTime').read() == sc_time(33, sc_time_unit.SC_SEC)
  <ETaction> VPvar('m_TLMAdapterTSideTLMSendPayloadInj').force(injVal);
<ETsrc> eState <ETtgt> eInit
  <ETaction> VPvar('m_TLMAdapterTSideTLMSendPayloadInj').release();
  <ETupdate> BTMclock('OKTime').reset()
```

Abbildung 7.15: Verdopplung eines verzögerten Transaktionsobjekts

Im nächsten betrachteten Fall wird kein detailliertes Kommunikationsmodell verwendet, sondern eine transaktionsbasierte Kommunikation. Das heißt, der Regler und das physikalische Modell kapseln die Sensor- und Aktorwerte in Transaktionen und schreiben diese in eine Nachrichtenschlange. Der jeweilige Empfänger liest die

```
class M_TLMWatertankAdapter;
template<typename InjType, int Selector>
class ees_trigger<M_TLMWatertankAdapter, InjType, Selector>
{
public:
  void triggerInj()
  {
    m_trigger.inj(m_ptrObj);
  };


  void triggerRel()
  {
    // do nothing
  };

  M_TLMWatertankAdapter*                      m_ptrObj;
  ees_trigger_spec<M_TLMWatertankAdapter, 0> m_trigger;
};
```

Abbildung 7.16: Die Template-Klasse zur Spezifikation eines Sensitivitätsverhaltens

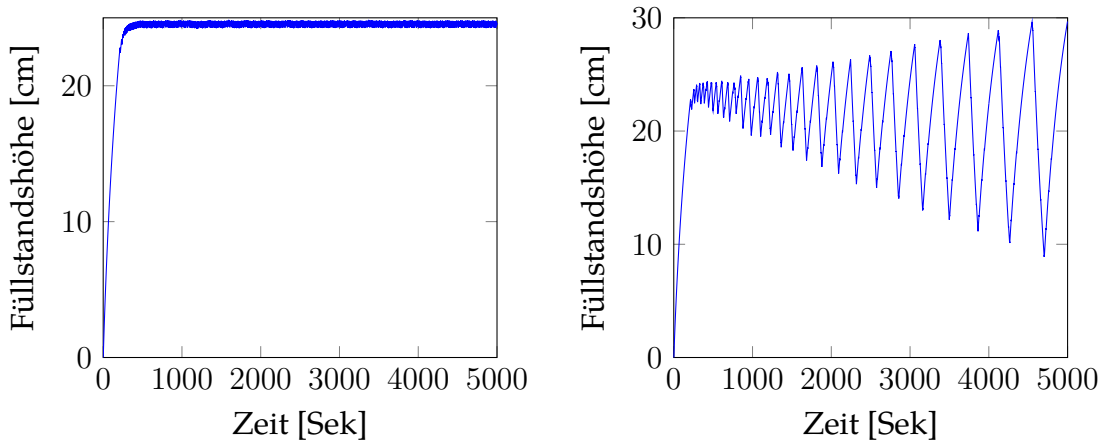
Daten aus der Nachrichtenschlange. Durch die Verwendung von TLM-Schnittstellen, d. h. der Wahrung der Makroarchitektur, ist es möglich, die Kommunikation im CAN-Modell durch die transaktionsbasierte Kommunikation auszutauschen. Mit

dem Modell wird der Effekt einer Verdopplung von Nachrichten untersucht. Hierbei wird ein alter Sensorwert bzw. ein altes Transaktionsobjekt für 31 Sekunden gespeichert und erneut versendet. Abbildung 7.15 zeigt die zugehörige Fehlerspezifikation. Nach 2 Sekunden wird das aktuelle Transaktionsobjekt, lokal im Stressor, zwischen-

```
1  class M_TLMWatertankAdapter;
2  template<>
3  class ees_trigger_spec<M_TLMWatertankAdapter, 0>
4  {
5  public:
6    void inj(M_TLMWatertankAdapter* cbObj)
7    {
8      sc_core::sc_time delay = sc_core::SC_ZERO_TIME;
9
10     cbObj->m_comInterface->b_transport(*(cbObj->m_tlmpSendPayload), delay);
11
12     if(cbObj->m_tlmpSendPayload->get_response_status() == tlm::TLM_OK_RESPONSE)
13     {
14         std::cout << "Send injection successful" << std::endl;
15     }
16   }
17 };
```

Abbildung 7.17: Beispiel: Spezifikation des Sensitivitätsverhaltens

gespeichert. Nach insgesamt 33 Sekunden wird das gespeicherte Objekt injiziert. Da es sich bei dem Fehler um eine Nachrichtenverdopplung handelt, muss die Fehlerinjektion eine sofortige Reaktion im System hervorrufen. Um ein erneutes Senden des Transaktionsobjekts auszulösen, müsste der Sendethread sensitive zum Transaktionsobjekt modelliert sein. Dies ist im Regelfall nicht gegeben, weswegen der Anwender ein eigenes Sensitivitätsverhalten spezifizieren muss. Abbildung 7.16 zeigt das vom Anwender spezifizierte Template zur Spezifikation des Sensitivitätsverhaltens. Das Template sieht sowohl eine Funktion für die Fehlerinjektion als auch für die Freigabe eines Fehlers vor. Hiermit kann der Anwender unterschiedliches Verhalten bei Eintritt der beiden Ereignisse vorsehen. Im dargestellten Fall wird bei der Fehlerinjektion die Aktion `m_trigger.inj()` ausgelöst, wohingegen bei dem Beheben des Fehlers keine Aktion unternommen wird. Die eigentlich auszuführende Aktion wird in einem zweiten Template spezifiziert. Wie in Abschnitt 5.2 motiviert wird durch diese Unterteilung die Wiederverwendbarkeit gesteigert. Abbildung 7.17 zeigt das eigentliche Verhalten bei der Fehlerinjektion. Wie durch die skizzierte Implementierung in Zeile 10 dargelegt, wird lediglich die blockierende Übertragungsfunktion aufgerufen. Dies bewirkt, dass wenn das Transaktionsobjekt durch den Fehlerinjektor abgeändert wird, dieses erneut gesendet wird. Abbildung 7.18 zeigt das resultierende Verhalten in der fehlerfreien Simulation und im Fehlerfall. Hierbei ist zu erkennen, wie das periodische Senden eines veralteten Wertes zu einem Aufschaukeln des Fehlers führt. Die anfängliche Über- bzw. Untersteuerung ist gering. Da der Fehler aber periodisch injiziert wird, führt dies dazu, dass die Korrekturversuche des Regelalgorithmus, d. h. die starken Regelabweichungen, in den Fehlerzustand einfließen und zeitlich verzögert, dupliziert wiedergegeben werden. Somit regelt der Algorithmus verstärkt nach und es kommt zu dem Aufschaukeln der Füllstandshöhe. In dem Beispiel mit



(a) Verhalten im fehlerfreien Fall (b) Verhalten im Fehlerfall

Abbildung 7.18: Fehlereffekt bei der Sensorwert Verdopplung

dem verfälschten Sensorsignal fand eine konstante Übersteuerung statt. In diesem Fall wird eine angefachte Schwingung der Füllstandshöhe beobachtet, wie sie lediglich, wenn auch mit vorgegebenen Amplitudenschwankung, nur bei der Hysterese-Regelung beobachtet wurde. Es ist ersichtlich, dass eine solche Bestimmung der Fehlerauswirkung, lediglich durch Kenntnis der Spezifikation, sehr schwierig ist.

7.2.4 Zuverlässigkeitsbewertung am FB3

Die letzte Fallstudie stellt drei typische Modelle, bei der Entwicklung eines vernetzen eingebetteten Systems, vor. Das rein funktionale Modell ist ein sehr abstraktes Modell, das häufig in frühen Entwicklungsphasen anzutreffen ist. Es dient vorrangig für erste Abschätzungen sowie zur Identifikation von Anforderungen an die zu entwickelnde Hardware. Das Modell besteht meistens aus einer rein funktionalen Implementierung bzw. aus Mock-Objekten, welche die Funktionalität imitieren. Das zweite Modell gruppiert die Funktionalität in logische Einheiten und führt erste Kommunikationsassoziationen ein. Transaktionen zwischen funktionalen Gruppen modellieren die Kommunikationsassoziationen. Das Modell ist in einer Entwicklungsphase angesiedelt, in der Konzepte der späteren Hardware existieren. Das letzte Modell verwendet detaillierte Hardwaremodelle und ist meist in den fortgeschrittenen Phasen anzutreffen. Die Hardwareressourcen sind identifiziert und die Abbildung der Software auf die Hardwareressourcen ist durchgeführt. Hierbei kommen häufig RTL-Modelle oder sogar Gattermodelle zum Einsatz. Abbildung 7.19 zeigt die drei Modelle in Bezug auf den Entwurfsablauf. Des Weiteren sind die für jede Ebene benötigten Schritte, dargestellt. In jeder neuen Abstraktionsebene wird das Modell verfeinert, um das zu analysierende System detaillierter zu modellieren und sich der finalen Implementierung anzunähern. Aus dem Spezifikationsmodell wird dann einmalig das Simulationsmodell, genauer die Simulationseinheitenbibliothek generiert. Mit der SEB ist es möglich, eine Vielzahl von Zuverlässigkeitsanalysen durchzuführen,

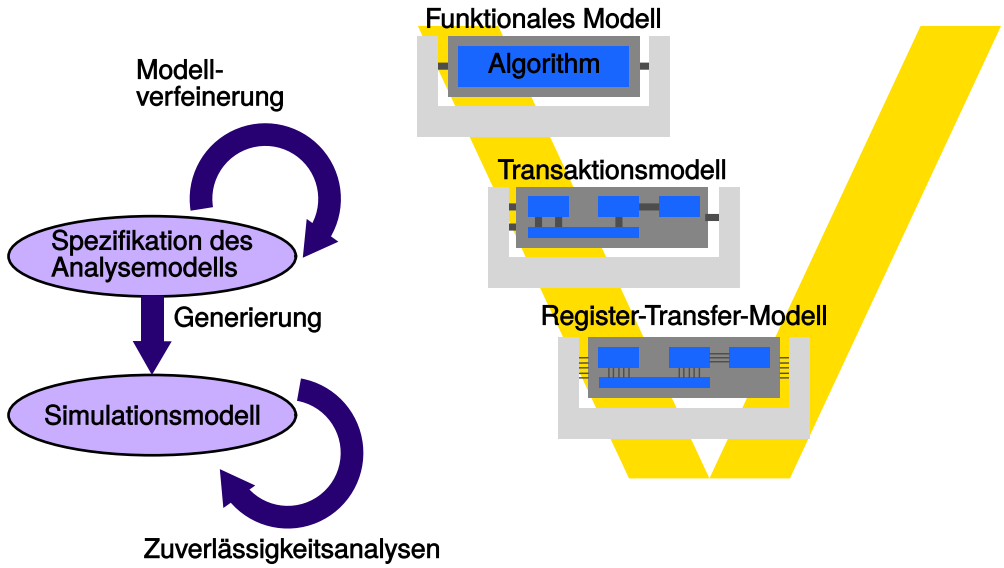


Abbildung 7.19: Die in der ADAS-Fallstudie untersuchten Modelle

ohne die SEB neu zu generieren. Erst bei einem erneuten Verfeinerungsschritt muss die Codegenerierung die SEB neu erzeugen.

Die einzelnen Modelle und einige, auf ihnen basierende, Analysen sind im Folgenden vorgestellt. Als Stimulation, d. h. als Kamera-Stream, wird die Fahrzeugumgebungssimulation CarMaker, der Firma IPG, verwendet. Hierbei wird ein Streckenmodell verwendet das konform zum Neuen Europäischen Fahrzyklus (NEFZs) ist. Im Beispiel ist eine Überlandfahrt (engl. highway driving) modelliert [5]. Die Spezifikation des Fahrzyklus erfolgt mit der Angabe der Fahrgeschwindigkeit und der Dauer des Abschnitts. Die verwendete Strecke bildet diese Spezifikation mit den entsprechenden Geschwindigkeitsbeschränkungen nach. Hierbei vernachlässigt die Teststrecke die Abschnitte, in denen die Geschwindigkeit gehalten wird.

Das Beispiel verdeutlicht, dass die beobachteten Fehlereffekte neben dem Simulationsmodell auch von dem Stimulus der Simulation abhängen. In den nachfolgend vorgestellten Analysen wird lediglich ein Stimulus, d. h. eine abgefahrene Fahrstrecke verwendet.

7.2.4.1 Funktionales Modell

Im folgenden Abschnitt wird die Vorgehensweise zur Erstellung der Fehlereffektsimulation anhand eines rein funktionalen Modells vorgestellt. In den untersuchten Simulationsmodellen wird die Gesamtfunktionalität, also die Berechnung von Fahrempfehlungen, in funktionale Blöcke gruppiert. Jede Funktionsgruppe wird als eine BSE spezifiziert. Im untersuchten Abstraktionsgrad kommunizieren die Simulationseinheiten über Methodenaufrufe, d. h. in Bezug auf die Makroarchitektur findet die Kommunikation innerhalb einer Schicht statt. Abbildung 7.20 zeigt ausschnittsweise die Schnittstellendefinitionen des funktionalen Spezifikationsmodells. Der grafische Editor wird neben der Definition der Schnittstellen zur Spezifikation der eigentli-

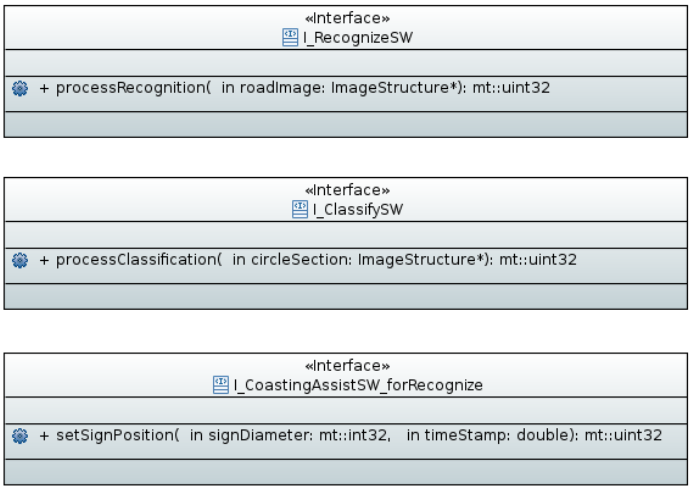


Abbildung 7.20: Ausschnitt über die funktionalen Schnittstellen des Modells

chen Simulationseinheiten verwendet. Die Spezifikation erfolgt mit je einer UML-Klasse für jede BSE. Abbildung 7.21 zeigt exemplarisch die Spezifikation der Simulationseinheit `M_Recognize`. Hierbei ist zu erkennen, dass die Klasse sowohl ein zuvor

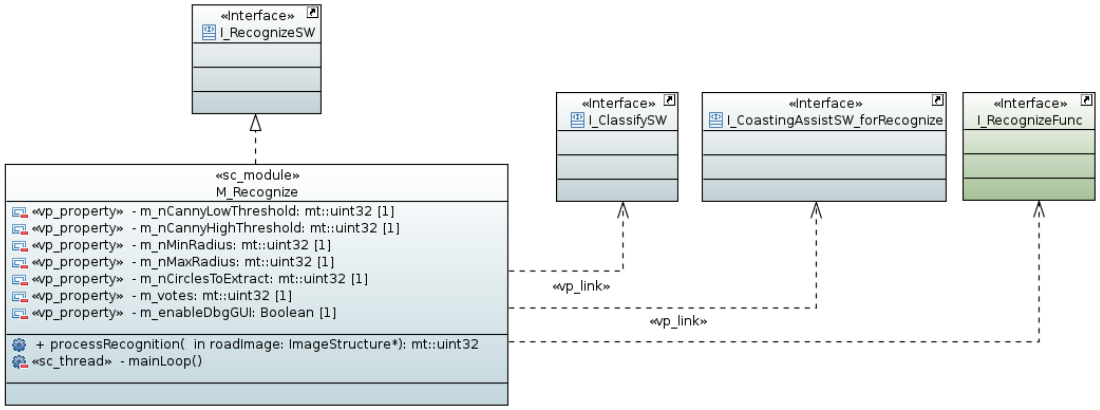


Abbildung 7.21: Funktionales Modell des Anwendungsfalls

spezifiziertes Interface bereitstellt, als auch zwei weitere Interface-Definitionen benötigt. Mittels dem Stereotyp `vp_link` wird die Art der Beziehung zwischen den Simulationseinheiten spezifiziert. Neben den Assoziationen spezifiziert die UML-Klasse alle Membervariablen und Membermethoden. Zur Spezifikation wird neben den klassischen UML-Modellierungsprimitiven wie z. B. `public` oder `private` auch die Stereotyperweiterung in Bezug auf SystemC verwendet. Die Methode `mainLoop` wird z. B. als `sc_thread` spezifiziert. Das Diagramm verwendet noch weitere UML-Erweiterungen, die in Bezug zur Konfigurationsdatei stehen. Die Membervariable `m_nCannyLowThreshold` ist mit dem Stereotypen `vp_property` versehen.

Nach der Spezifikation generiert die Codegenerierung die eigentliche SystemC-Bibliothek. Der Generierungsprozess generiert für jede Klasse eine header- und cpp-

```
::conf::ConfBase* M_Recognize::getInstance(
          ::conf::C_IPXACTConfigurator* clModuleConf)
{
    // create module name
    std::string pchModuleName(clModuleConf->getCurrentInstID());

    // instantiate actual object
    M_Recognize*   clInst = new M_Recognize(pchModuleName.c_str());
    ::conf::ConfBase*   clBase(clInst);

    // assign properties
    while(clModuleConf->hasParameter()) {
        if(clModuleConf->getCurrentParameterID()=="CannyLowThreshold") {
            while(clModuleConf->hasValue()) {
                clInst->m_nCannyLowThreshold =
                  clModuleConf->getCurrentValue_longint();
                clModuleConf->getNextValue();
            }
        }
    ...
} // getInstance()

void M_Recognize::linkInstance(::conf::ConfBase* cpclBase, const std::string&
    csLinkID) {
    if(csLinkID=="RecognizeClassifyInterface") {
        I_ClassifySW* linkTmp = dynamic_cast<I_ClassifySW*>(cpclBase);

        if(linkTmp != NULL) {
            m_pRecognizeClassifyInterface = linkTmp;
        }
    }
    ...
}

void M_Recognize::mainLoop()
{
    // <!-- begin-user-behavior -->
    // <!-- end-user-behavior -->
} // mainLoop()
```

Abbildung 7.22: Generierter Quellcode für die Simulationseinheit `M_Recognize`

Datei. In diesem Fall wird die SystemC-Klassendefinition, die Membermethoden sowie die Membervariablen generiert. Für die beiden Methoden erzeugt die Codegenerierung zwei Implementierungsstubs in der cpp-Datei. Die Spezifikation einer SystemC-Thread-Methode führt zur Generierung der benötigten SystemC-Primitiven, wie `SC_HAS_PROCESS` und `SC_THREAD`. Im UML-Klassendiagramm ist zu erkennen, dass die Konfigurationsdatei einzelne Membervariablen initialisiert. Bei der Generierung wird dies berücksichtigt und automatisch die entsprechenden Parser und Zuweisungsmethoden der Klasse hinzugefügt. Diese Methoden werden bei der dynamischen Erzeugung der Simulation aufgerufen. Die dynamische Generierung der BSEs entspricht dem Entwurfsmuster der Fabrikmethode. Der notwendige Managementcode wird automatisiert aus der UML-Spezifikation generiert. Ähnlicher funktionaler Quellcode wird zur Verknüpfung der Module generiert. Abbildung 7.22 zeigt auszugsweise die generierte Klasse. Die erste Methode zeigt die Fabrikmethode zur Erzeugung der Klasse und zur Zuweisung der Parameter. Die zweite generierte

Methode verknüpft die Simulationseinheiten. Die letzte dargestellte Methode zeigt exemplarisch die generierten Methodenrümpfe, die vom Anwender zu implementieren sind. Zusätzlich zur Generierung der SystemC-Simulationseinheitenbibliothek wird für jede Simulationseinheit eine IP-XACT-Datei generiert, welche die Simulationseinheit beschreibt. Dies ist notwendig, da die Konfigurationsdatei nicht alle benötigten Informationen, wie z. B. den Datentyp einer parametrisierbaren Membervariable, bereitstellt.

Im nächsten Schritt instanziiert, verknüpft und parametrisiert ein Kompositionsstrukturdiagramm die erstellten UML-Klassen. Dargestellt ist die Parametrisierung

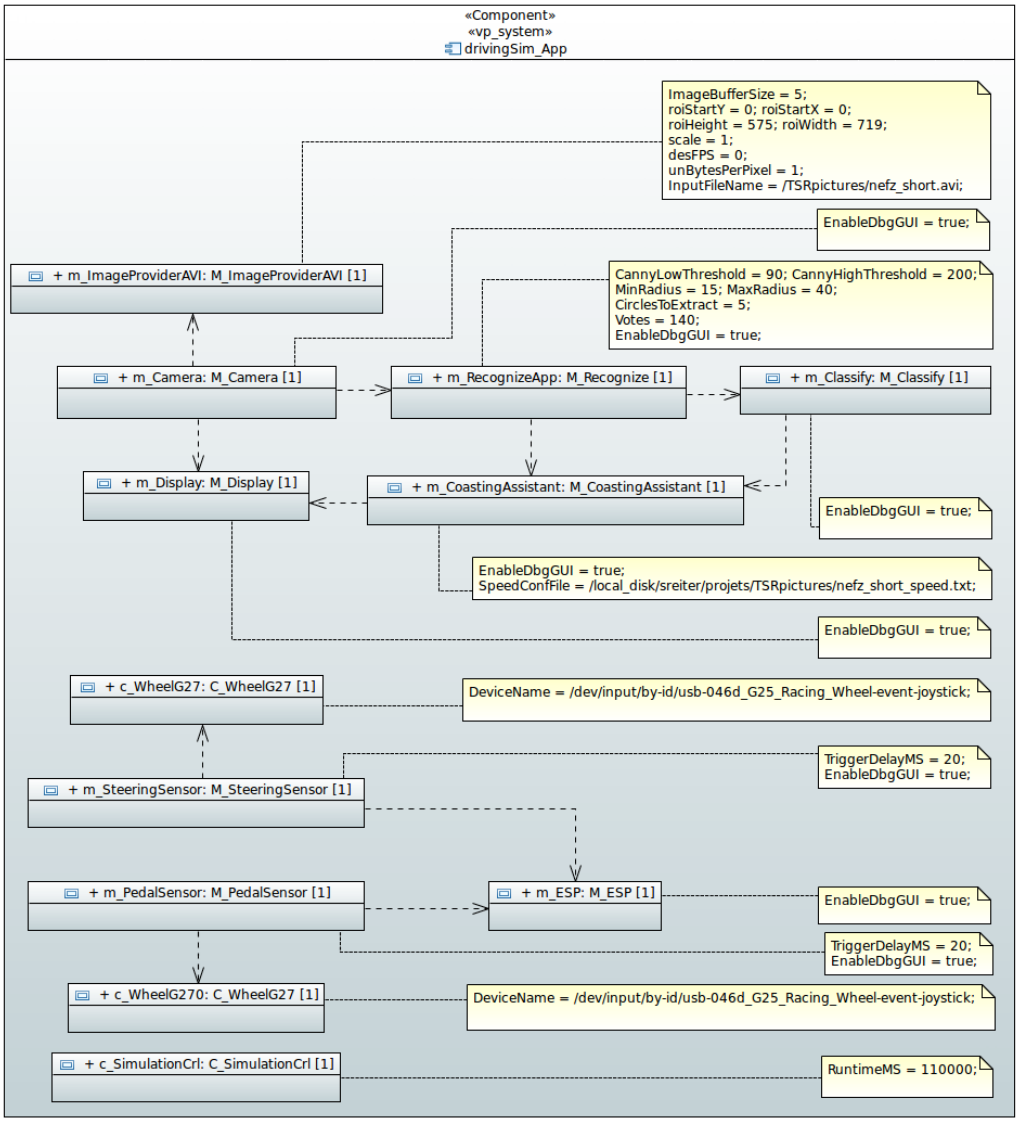


Abbildung 7.23: Struktur des funktionalen Modells

der Simulationsinstanzen, z. B. der Instanz von `M_Recognize`. Außerdem werden die Simulationseinheiten verknüpft, hierbei wird Bezug zu den im Klassendiagramm spezifizierten Assoziationen genommen. Ausgehend von dem Diagramm wird au-

tomatisch die Konfigurationsdatei generiert, die auszugsweise in Abbildung 7.24 dargestellt ist.

Abbildung 7.23 zeigt die finale Spezifikation der Simulationsinstanz. Die durch die Konfiguration erzeugte Simulation dient zur Erstellung einer Nominalsimulation, d. h. eine Systemsimulation ohne jegliche Fehlerinjektion. Um Fehler in die Simulation zu injizieren, muss der Anwender lediglich die zu verändernde Membervariable im UML-Klassendiagramm mit dem Stereotyp `es_injector` assoziieren. Im Folgenden werden unterschiedliche Fehler in den von der Kamera aufgezeichneten Bildstrom injiziert. Hierzu wird die Membervariable in der Simulationseinheit `M_Camera`, welche die Nutzdaten des zuletzt aufgenommenen Bilds speichert, mit einem Fehlerinjektor versehen. Die erweiterte UML-Spezifikation ist in Abbildung 7.25 dargestellt. Bei der Codegenerierung wird der Stereotyp durch die Generierung der Typendeklaration des Fehlerinjektors und des Fehlerinjektor-Konstruktors berücksichtigt. Zur Spezifikation des Fehlverhaltens stehen in der UML-Umgebung zwei Ansätze zur Verfügung, die Spezifikation als UML-Zustandsdiagramm und die Spezifikation des BTMs als textueller Kommentar. Abbildung 7.26 zeigt die Spezifikation eines Fehlers, bei welchem die Kamera ab einem gewissen Zeitpunkt immer das gleiche Bild liefert. Der Fehler tritt nach 23 Sekunden auf und bleibt ab diesem Zeitpunkt bestehen, da es keine ausgehende Kante, mit einer Fehlerbehebung, aus dem Zustand `eState` gibt. Beim Übergang in den Zustand `eState` wird die Fehlerinjektion ausgelöst. Die Fehlerinjektion liest einen Bildpunkt aus den Nutzdaten des aktuellen Bildinhaltes und modifiziert den Bildpunkt. Ab diesem Zeitpunkt wird das komplette Array, d. h. alle dem Fehlerinjektor zugeordneten Daten, auf den injizierten Wert gehalten. Dies führt dazu, dass der Fehlerinjektor ab diesem Zeitpunkt die regulären Änderungen des Bilds mit dem zu injizierenden Wert überschreibt. Eine Änderung durch die Simulation wird durch den Fehlerinjektor verhindert. Abbildung 7.27 zeigt das beobachtete Fehlverhalten in der Simulationseinheit `M_Recognize`. Die obere Zeitschiene zeigt das von `M_Recognize` empfangene Bild in der Nominalsimulation. Die untere Achse zeigt die Bilder im Fehlerfall. Es ist zu erkennen, dass ab dem Zeitpunkt 23 Sekunden sich das empfangene Bild nicht mehr ändert. Um die Fehlerspezifikation in einen transienten Fehler abzuändern, ist es ausreichend eine ausgehende Kante aus dem Zustand `eState` zu spezifizieren, bei dem der Fehler aufgehoben wird. Dies erfolgt mit dem Ausdruck:

```
VPvar('bytesPerPixel')[0].release()
```

Ein weiterer Fehlerfall, der im Folgenden vorgestellt wird, ist die bitweise Verfälschung des Bildstroms. Um eine fortschreitende Anpassung des Fehlerbilds mit dem aktuellen Bild aus dem Bildstrom zu gewährleisten, wird der Fehler über eine rekursive Kante freigegeben und direkt im Anschluss neu injiziert. Dies sorgt dafür, dass sich Änderungen durch die Simulation auf das Fehlerbild auswirken. Abbildung 7.28 zeigt das zugehörige BTM. Wichtig ist hierbei, dass bei der rekursiven Kante im Zustand `eState` zuerst die Fehlerinjektion freigegeben wird, um sie anschließend erneut zu aktivieren. Dies erlaubt der Simulation, den internen Speicher des Fehlerinjektors mit dem aktuellen Videobild zu überschreiben, bevor die Injektion erneut aktiviert wird. Durch diese Spezifikation wird das Fehlerbild immer auf das aktuellste von

```
<?xml version="1.0" encoding="UTF-8"?>
<ipxact:design xsi:schemaLocation="http://www.accellera.org/XMLSchema/IPXACT
    /1685-2014 ../../../src/configuration/xsd/index.xsd">
  <ipxact:vendor>FZI</ipxact:vendor>
  <ipxact:library>Library</ipxact:library>
  <ipxact:name>DrivingSim_App</ipxact:name>
  <ipxact:version>0.1</ipxact:version>
  <ipxact:componentInstances>
...
    <ipxact:componentInstance>
      <ipxact:instanceName>m_RecognizeApp</ipxact:instanceName>
      <ipxact:componentRef library="Library" name="M_Recognize" vendor="Library"
          version="0.1">
        <ipxact:configurableElementValues>
          <ipxact:configurableElementValue referenceId="m_nCannyLowThreshold">
               90</ipxact:configurableElementValue>
          <ipxact:configurableElementValue referenceId="m_nCannyHighThreshold">
               200</ipxact:configurableElementValue>
          <ipxact:configurableElementValue referenceId="m_nMinRadius">
               15</ipxact:configurableElementValue>
          <ipxact:configurableElementValue referenceId="m_nMaxRadius">
               40</ipxact:configurableElementValue>
          <ipxact:configurableElementValue referenceId="m_nCirclesToExtract">
               5</ipxact:configurableElementValue>
          <ipxact:configurableElementValue referenceId="m_votes">
               140</ipxact:configurableElementValue>
          <ipxact:configurableElementValue referenceId="m_enableDbgGUI">
               true</ipxact:configurableElementValue>
        </ipxact:configurableElementValues>
      </ipxact:componentRef>
      <ipxact:vendorExtensions>
        <ipxact:file>
          <ipxact:name>src/TSR_coreEntities/M_Recognize.xml</ipxact:name>
          <ipxact:fileType user="xml">user</ipxact:fileType>
        </ipxact:file>
      </ipxact:vendorExtensions>
    </ipxact:componentInstance>
...
  </ipxact:componentInstances>
  <ipxact:interconnections>
    <ipxact:interconnection>
      <ipxact:name>m_RecognizeApp_m_CoastingAssistant</ipxact:name>
      <ipxact:activeInterface busRef="RecognizeCoastingInterface" componentRef="
          m_RecognizeApp"/>
      <ipxact:activeInterface busRef="unkownBusRef" componentRef="
          m_CoastingAssistant"/>
    </ipxact:interconnection>
    <ipxact:interconnection>
  </ipxact:interconnections>
</ipxact:design>
```

Abbildung 7.24: Auszug der Simulationskonfiguration für das funktionale Modell

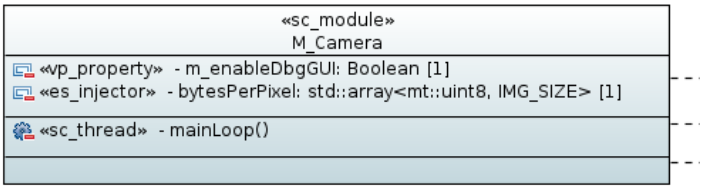


Abbildung 7.25: Spezifikation des Fehlerinjektors in `M_Camera`

der Kamera verschickte Bild angewendet. Abbildung 7.29 zeigt den Effekt von unterschiedlichen Pixelverfälschungen. Um die unterschiedlichen Verfälschungen zu erzeugen, ist es ausreichend den Wert der Variable `ids` im BTM anzupassen. Die vertikale Linie ist in Abbildung 7.28 dargestellt. Die horizontale Linie wird über den Ausdruck:

```
ids = [id for id in range(719*200*1,719*210*1)]
```

berechnet. Das zufällige Bitmuster verwendet hingegen die Random-Bibliothek von Python. Hierbei wird der Ausdruck:

```
ids=[]; idstmp = [randint(0,720*576*1) for id in range(0,400)];
        [ids.extend([id, id+1, id+2]) for id in idstmp];
```

verwendet. Der Ausdruck wählt zufällige Bildpunkte aus und modifiziert immer einen Block von drei sequenziellen Bildpunkten. Die spezifizierten BTMs gelten für Graustufenbilder, bei denen jeder Bildpunkt durch ein Byte repräsentiert wird. Bei der Verwendung von farbigen RGB-Daten müsste das BTM drei Bytes manipulieren. Die drei Beispiele zeigen die Vielfältigkeit des BTMs. Der Anwender kann das gewünschte Fehlverhalten frei spezifizieren und ist nicht auf vordefinierte Fehlerfälle eingeschränkt. Durch die Spezifikation des Fehlverhaltens mittels grafischen Diagramme ist das injizierte Fehlverhalten mit dem Anwendungsfall verknüpft und beides grafisch dokumentiert. Diese grafische Spezifikation vereinfacht die Nachverfolgbarkeit der Analyseergebnisse erheblich.

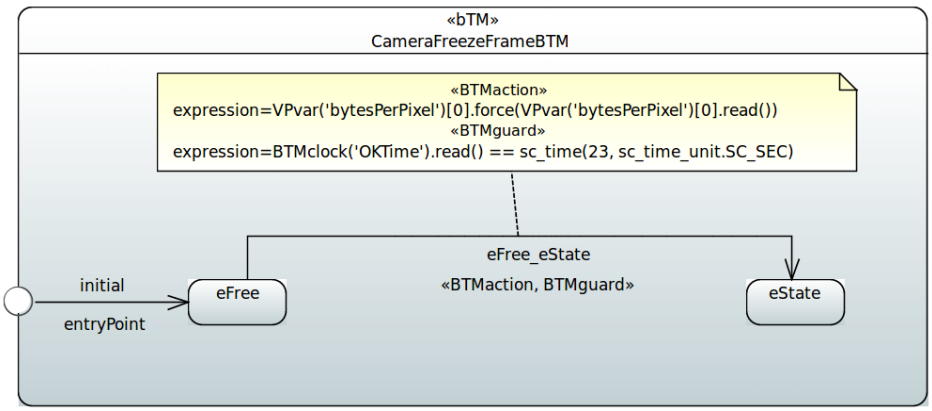


Abbildung 7.26: Permanenter Standbildfehler in der Kamera

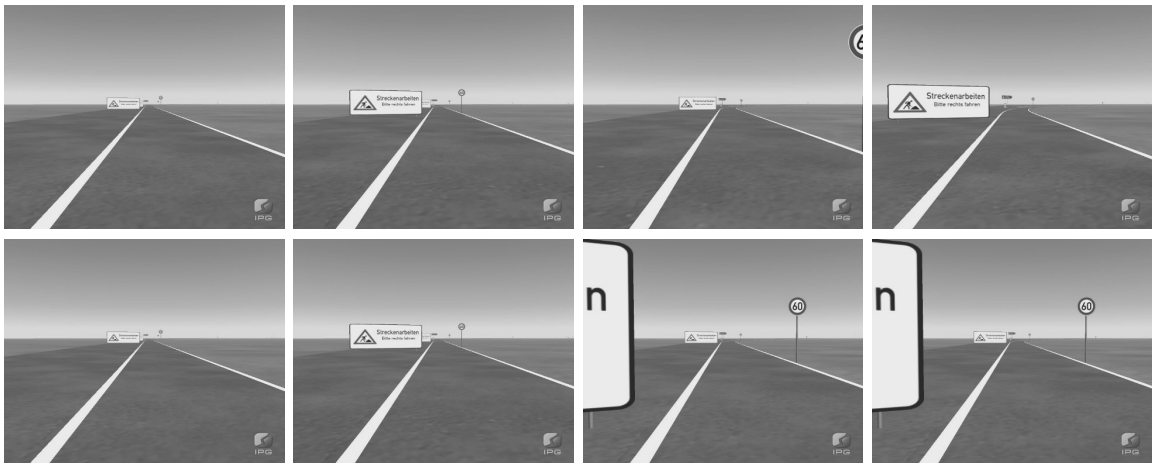


Abbildung 7.27: Fehlereffekt bei `M_Recognize`

7.2.4.2 Transaktionsmodell

In der zweiten Betrachtung des Anwendungsfalls wird ein Transaktionsmodell verwendet. Hierbei wird weitestgehend vom zeitlichen Verhalten der Kommunikation abstrahiert und lediglich die Struktur und die Funktionalität des Nachrichtenaustauschs beschrieben. Der Anwendungsfall fasst z. B. die Steuergeräte zu unterschiedlichen Bus-Clustern zusammen. Um die Austauschbarkeit des Kommunikationsmediums zu gewährleisten, d. h. eine Kommunikation über die Schichtengrenzen der Makroarchitektur, wird jede Anwendung aus dem vorherigen Abschnitt über eine TLM-Schnittstelle gekapselt. Der funktionale Teil der Anwendung bleibt bestehen und wird aus dem vorherigen funktionalen Modell wiederverwendet. Ein Template (`vp_system_template`) gruppiert den funktionalen Kern der Anwendung, den TLM-Adapter sowie eventuell benötigte Unterstützungsklassen. Die eigentliche Systemspezifikation wird mit dem Stereotyp `vp_system` spezifiziert, wie in Abbildung 7.23 dargestellt. Der Stereotyp `vp_system_template` markiert hingegen Teilsystem-Templates. Dies ermöglicht die Verwendung der Komponentengruppe durch einfache Instanziierung des Templates. Auch die hierarchische Verwendung von Templates innerhalb von Templates ist möglich. Abbildung 7.30 zeigt das mit der UML spezifizierte Template. Das Template besteht aus der zentralen Anwendung, dem Simulationsmodul zum Einlesen des Videostreams sowie dem TLM-Adapter, der mit dem ausgehenden Port verbunden ist. Bis auf den TLM-Adapter verwendet das rein funktionale Modell die gleichen Simulationsmodule, wie in Abbildung 7.23 zu sehen ist. Ähnlich zu dem Kompositionsstrukturdiagramm, aus dem vorherigen Abschnitt, wird die Template-Spezifikation zur Parametrisierung, der darin enthaltenen BSEs, verwendet. Die Simulationsspezifikation verwendet die Templates, um den kompletten Anwendungsfall zu spezifizieren. Abbildung 7.31 zeigt das transaktionsbasierte Spezifikationsmodell des Anwendungsfalls. Für die Fehlerinjektion stehen, in diesem Abstraktionsgrad, neben den Daten in den Anwendungen zusätzlich die übertragenen Transaktionsobjekte zur Verfügung. Das heißt, Fehler können nicht nur intern in

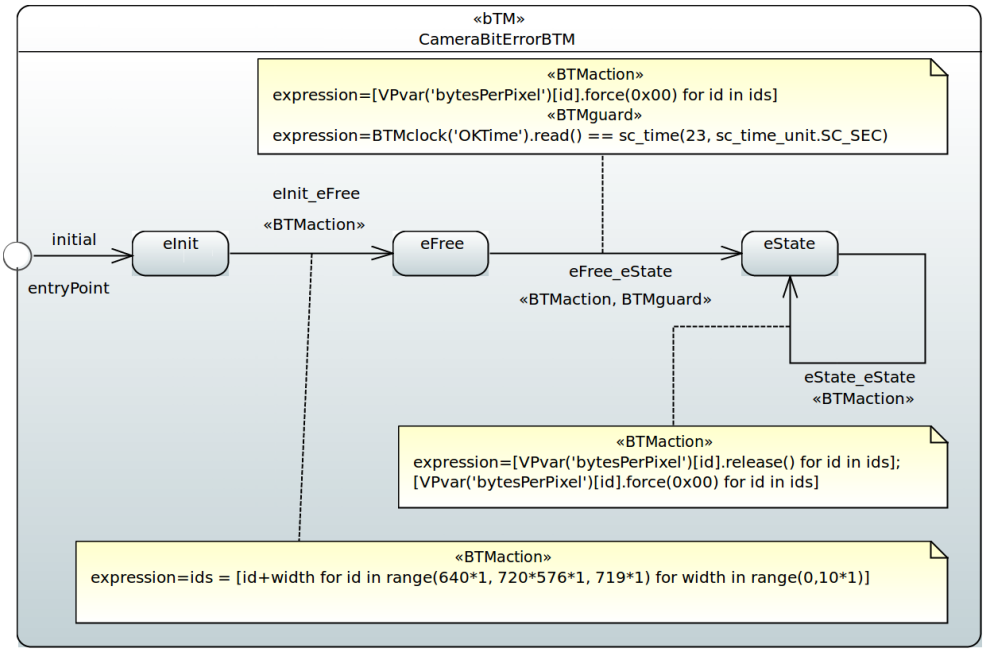


Abbildung 7.28: Permanenter Bitfehler in der Kamera

der Anwendung injiziert werden, sondern auch in die übertragenen Transaktionsobjekte. Um die Bitfehler aus dem vorherigen Abschnitt in die Transaktionsobjekte zu injizieren, wird das BTM aus Abbildung 7.32 verwendet. Hierbei ist hervorzuheben, dass der Ansatz die Modifikation aller Einträge des Transaktionsobjekts ermöglicht. Neben den hier verwendeten Nutzdaten (`get_data()`) ist es möglich, auf folgende Informationen zu zugreifen:

- die Adresse (`get_address()`),
- ob es sich um einen Lese- oder Schreibzugriff handelt (`get_command()`),
- die angegebene Datenlänge (`get_data_length()`) oder
- den Rückgabewert (`get_response_status()`).

Abbildung 7.29: Unterschiedliche Bit-Fehlereffekte in der Kamera

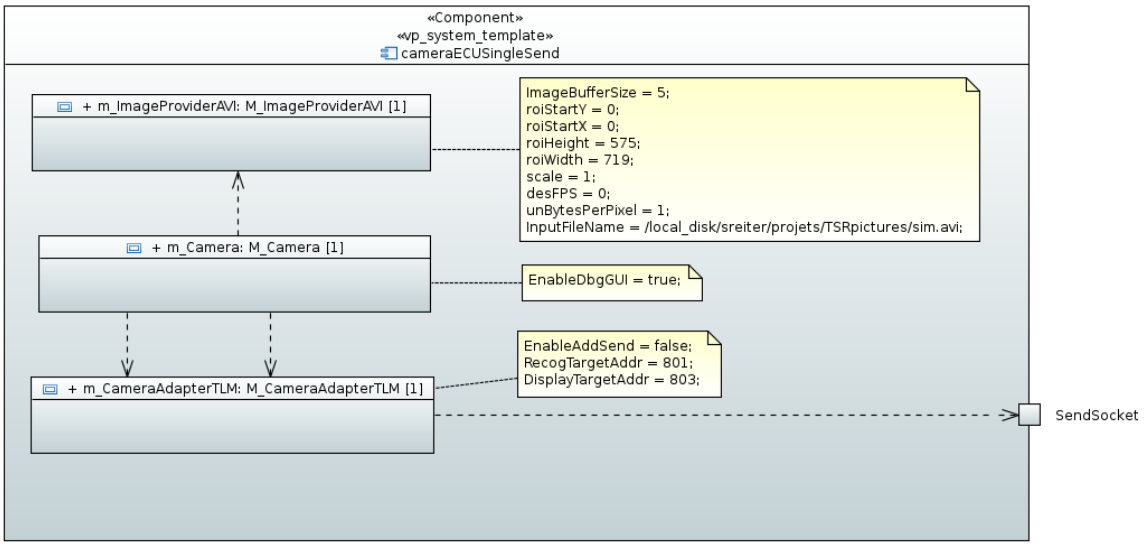


Abbildung 7.30: Template zur Kapselung der Kamerafunktionalität

Die Fehlerbeschreibung ermöglicht die Nachbildung der Verfälschungen aus dem rein funktionalen Modell. In den dargelegten Beispielen entsprechen die im funktionalen Modell modellierten Fehlerursachen, Fehlereffekte aus dem detaillierteren Kommunikationsmodell.

Die Modellierung des Datenaustauschs über Transaktionen berücksichtigt zum ersten Mal ein abstraktes Zeitverhalten für die Weiterleitung der Daten. Die Fehlerinjektoren ermöglichen, dieses Zeitverhalten zu verändern, um z. B. eine längere Übertragungszeit zu simulieren. Dies ist erforderlich um z. B. Bussysteme mit zuverlässigen Übertragungsprotokollen zu berücksichtigen. Häufig kommen CRC-Checks zum Einsatz, um Bitfehler in den empfangenen Daten zu detektieren. Bei Erkennen von solchen Fehlern wird häufig eine Sendewiederholung gestartet, was zu einer Verlängerung der Übertragungsdauer führt. Da im TLM-Modell die spezifischen Busse nicht modelliert sind, bleibt der Fehlerinjektion lediglich die Veränderung der Übertragungsdauer. Das Beispiel verdeutlicht, dass es bei abstrakten Modellen unverzichtbar ist, dass das spezifizierte Fehlerverhalten auch das Systemverhalten mit modelliert, das nicht im Systemmodell beinhaltet ist. Abbildung 7.33 zeigt die Spezifikation eines Verzögerungsfehlers. Hierbei wird nicht das Transaktionsobjekt verändert, sondern die Verzögerung, die für die Transaktion spezifiziert wird. Bei der Fehlerspezifikation wird die Übertragungsdauer jedes 5. Transaktionsobjekt mit einer zusätzlichen Verzögerung von 15 ms modifiziert. Hierbei wird eine im Modell enthaltene Hilfsvariable verwendet, die zu Debug-Zwecken im Modell bereits vorhanden ist. Die Variable zählt die während der Simulation versendeten Transaktionsobjekte. Durch die Zuweisung eines Fehlerinjektors zu der Hilfsvariablen kann der Stressor diese zurücksetzen und als relativen Zähler verwenden. Eine weitere Möglichkeit ist die Verwendung einer lokalen Variablen innerhalb des BTMs.

Anhand des Transaktionsmodells wird im Folgenden die Unterstützung einer FTA dargelegt. Für die vorliegende Abstraktion des Systems wird ein Fehlerbaum

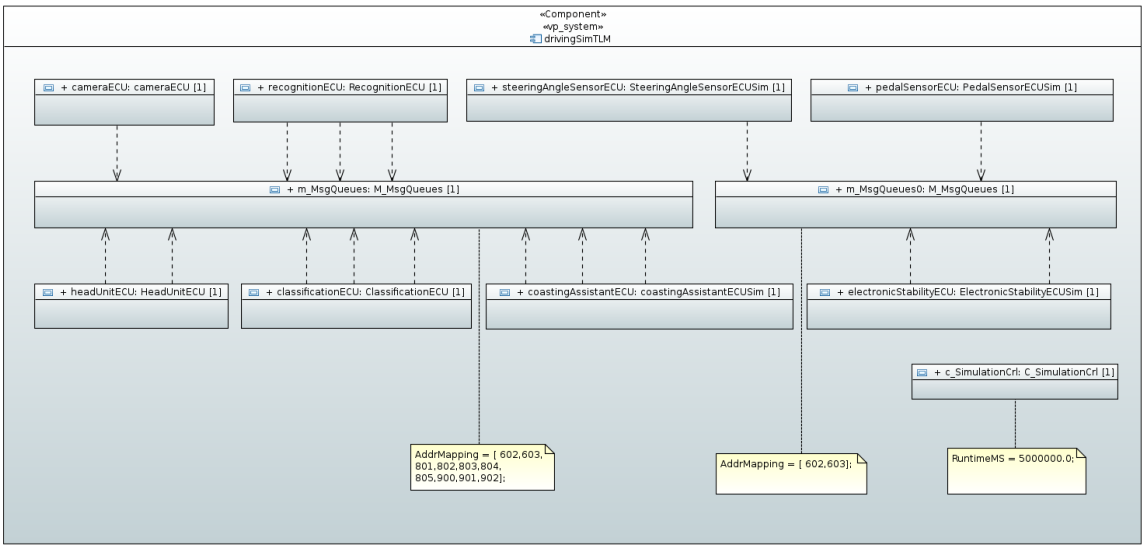


Abbildung 7.31: Spezifikation des Transaktionsmodells des Anwendungsfalls

verwendet, der die Abhängigkeiten von Fehlern und somit die Propagierung von Fehlern im System modelliert. Die hier gewählte Repräsentationsform basiert auf einem CFT. Aufgrund der sehr guten Kombination zwischen komponentenbasierter Fehlereffektsimulation und komponentenbasierter FTA sowie wegen der besseren Lesbarkeit wird das Vorgehen am CFT-Ansatz demonstriert. Abbildung 7.34 zeigt den Komponentenfehlerbaum der Kamera. Hierbei werden folgende Fehlerursachen modelliert:

- content_failure
  Der interne Bildspeicher verhindert ein Überschreiben mit aktuelleren Daten, d. h. es wird immer das letzte Bild gesendet.

- pixel_failure
  Pixelfehler in dem gesendeten Bildstrom. Können sowohl von Fehlern im Zwischenspeicher als auch von Fehlern in der optischen Einheit der Kamera verursacht werden.

- camera_defect
  Ein kompletter Ausfall der Kamera, d. h. es werden keine Bilder an die nachfolgende Verarbeitungsstufe gesendet.

- sampletime_deviation
  Schwankungen in der zeitlichen Dauer, in der die Bilder gesendet bzw. vom optischen Sensor erfasst werden.

Die Fehlerursachen treten in der Kamera auf und beeinflussen die ausgehenden Daten und somit den erbrachten Dienst der Kamera. Der CFT modelliert die Fehlerursachen als *Basic-Events*, die sich zu den Fehlermodi am Ausgangsport propagieren. Die Fehlereffektsimulation modelliert die Basic-Events als BTM. Um die im BTM

```
<ESname> eInit <EStype> initial
<ESname> eFree
<ESname> eState
<ETsrc> eInit <ETtgt> eFree
     <ETaction> ids = [id+width for id in range(640*1, 720*576*1, 719*1) for
         width in range(0,10*1)]
<ETsrc> eFree <ETtgt> eState
    <ETguard> BTMclock('OKTime').read() == sc_time(23, sc_time_unit.SC_SEC)
    <ETaction> injVal = VPvar('tlmpayloadProbeCamera').read();
               injVal.set_data([item if i not in ids else 0 for i,item in
                   enumerate(injVal.get_data())]);
               VPvar('tlmpayloadProbeCamera').force(injVal);
 <ETsrc> eState <ETtgt> eState
    <ETaction> VPvar('tlmpayloadProbeCamera').release();
               injVal = VPvar('tlmpayloadProbeCamera').read();
               injVal.set_data([item if i not in ids else 0 for i,item in
                   enumerate(injVal.get_data())]);
               VPvar('tlmpayloadProbeCamera').force(injVal);
```

Abbildung 7.32: Bitfehlerbeschreibung für Transaktionsobjekte

spezifizierten Fehler zu injizieren, wird das Simulationsmodell mit den benötigten Fehlerinjektoren erweitert. Der in Abbildung 7.28 modellierte Fehler entspricht dem pixel_failure in der durchgeführten FTA. Sein textuelles Pendant im TLM-Kanal zeigt Abbildung 7.32. Der Fehlerinjektor in Abbildung 7.28 bezieht sich auf das in der Kamera zwischengespeicherte Bild. Abbildung 7.35 zeigt ein Auszug aus dem Komponentenfehlerbaum der Komponente Coasting Assistant. Hierbei existieren mehrere Input-Fehlermodi, die durch die Propagierung von Fehlern über Komponentengrenzen hinweg entstehen. Zusätzlich hat die Komponente ein Basic-Event, das einen internen Fehler hervorruft. In beiden Komponentenfehlerbäumen werden die propagierten Fehler als Output-Fehlermodi deklariert. Die Fehlereffektsimulation weist jedem Output-Fehlermodus einen Monitor zu, um die Fehlerauswirkungen zu detektieren und zu klassifizieren. Zumindest die Output-Fehlermodi an den Systemgrenzen sollten mit Monitoren ausgestattet werden, da hier Abweichungen, vom System erbrachten Dienst, beobachtet werden können. Zusätzlich ist es möglich die internen Output-Fehlermodi mit Monitoren auszustatten, um die Fehlerpropagierung zu untersuchen. Bei der hier betrachteten FTA-Unterstützung wird die vom Coasting

```
<ESname> eFree <EStype> initial
<ESname> eState
<ETsrc> eFree <ETtgt> eState
  <ETguard> VPvar('TLMtlmMsgCounter').read() >= 5
  <ETaction> VPvar('m_ClassifyAdapterTLMtlmTimeProbeClassify').force(
                 VPvar('m_ClassifyAdapterTLMtlmTimeProbeClassify').read() +
                 sc_time(15, sc_time_unit.SC_MS));
<ETsrc> eState <ETtgt> eFree
  <ETguard> VPvar('TLMtlmMsgCounter').read() >= 6
  <ETaction> VPvar('TLMtlmTimeProbeClassify').release();
             VPvar('TLMtlmMsgCounter').force(1);
             VPvar('TLMtlmMsgCounter').release(clearStrategy.RELEASE);)
```

Abbildung 7.33: Spezifikation eines transienten Verzögerungsfehlers

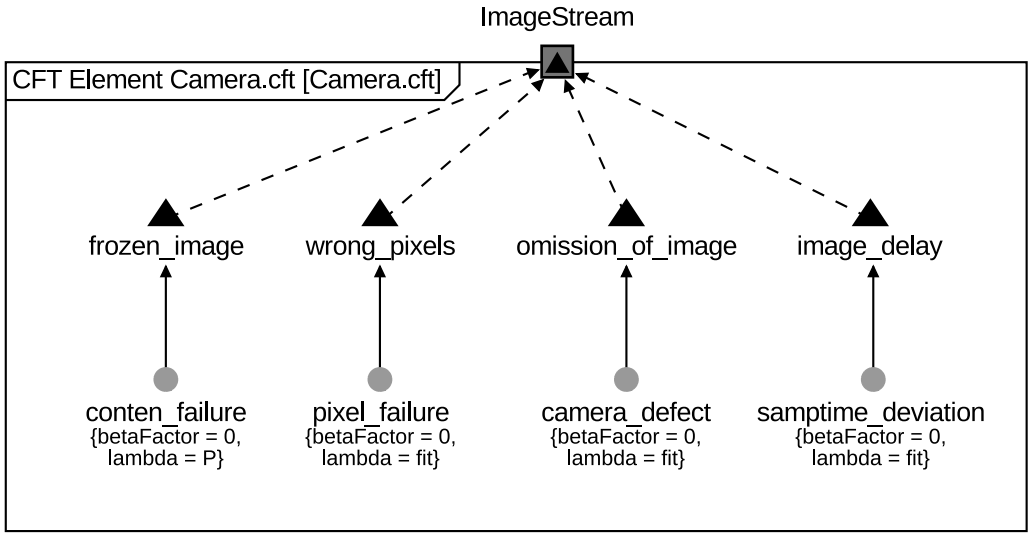


Abbildung 7.34: Komponentenfehlerbaum der Kamera

Assistant bereitgestellte Information zu Geschwindigkeitsbegrenzungen mit einem Monitor ausgestattet und die Abweichungen vom erwarteten Verhalten untersucht. Korrespondieren die zu untersuchende Fehlermodi mit den klassischen Fehlermodi, wie in Abschnitt 5.4 dargelegt, reichen die bereitgestellten Fehlerklassifikatoren aus und es sind keine applikationsspezifischen Anpassungen notwendig. Die in Abbildung 7.35 verwendeten Fehlermodi, entsprechen den unterstützten Fehlermodi, weswegen kein manueller Eingriff des Anwenders notwendig ist. Der Fehler late_hint entspricht einer verzögerten Weiterleitung der detektierten Geschwindigkeitsbegrenzung. Die Fehlerklassifikation in UPPAAL, dargestellt in Abbildung 5.32, muss somit den Zustand `late` durchlaufen, um diesen Fehlermodus zu detektieren. Ähnlich verhält es sich mit den restlichen dargestellten Fehlermodi, die alle auf einen Zustand in der Fehlerklassifikation abgebildet werden. Um diese automatisierte Fehlerklassifikation zu erzielen, ist es ausreichend die gesendete Geschwindigkeitsbegrenzung mit einem Monitor auszustatten. Nach Erzeugen einer Nominalsimulation, detektiert und klassifiziert der Monitor die unterschiedlichen Fehlermodi automatisch.

Mithilfe der Fehlereffektsimulation und der automatischen Fehlerklassifikation wird der Fehlerbaum verifiziert. Hierzu werden die unterschiedlichen Fehlerursachen einzeln injiziert und überprüft, ob der korrekte Fehlereffekt hervorgerufen wird. Nach dem Erzeugen eines Referenz-Trace, wird in der Konfigurationsdatei die UPPAAL-Verifikation aktiviert und anschließend die unterschiedlichen Fehler injiziert. Die durchgeführte Analyse bestätigte alle Assoziation im Fehlerbaum bis auf eine. Der manuell erstellte Fehlerbaum beinhaltet eine Assoziation, die es ermöglicht, dass ein verfälschtes Kamerabild (pixel_fault) sich zu einer verfälschten Geschwindigkeitsbeschränkung propagiert. Die Fehlereffektsimulation konnte diese Fehlerkette nicht bestätigen. Grund liegt vor allem daran, dass das verfälschte Bild, bzw. dessen Bildausschnitt, ein Klassifizierungsschritt durchläuft. Bei vielen Verfälschungsmustern liefert die Klassifizierung die korrekte Geschwindigkeitsbegrenzung. Erhöht

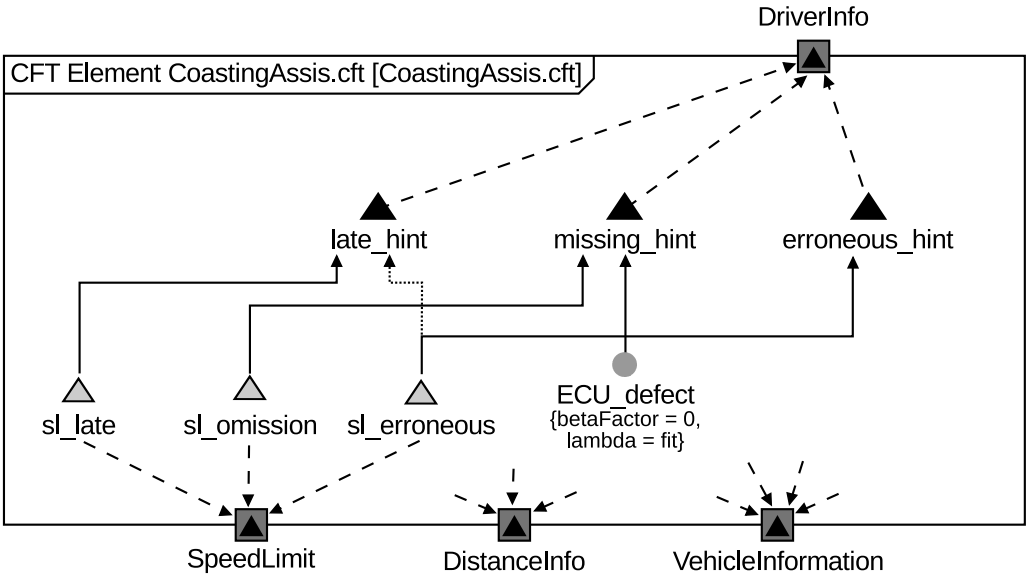


Abbildung 7.35: Komponentenfehlerbaum der Komponente Coasting Assistant

der Anwender die Pixelverfälschung bzw. tritt eine Häufung an der Position des Verkehrszeichens auf, liefert die Klassifizierung kein Ergebnis anstelle einer falschen Klassifikation. Der Coasting Assistant ignoriert die nicht klassifizierbaren Schilder, was zu einem Ausbleiben der Geschwindigkeitsbegrenzungsnachricht führt. In der Simulation konnte keine Pixelverfälschung, mit den oben vorgestellten Verfahren, erzeugt werden, um eine falsche Klassifikation hervorzurufen. Auf der anderen Seite wurde eine verzögerte Erkennung der Geschwindigkeitsbeschränkung, hervorgerufen durch die Verfälschung der Daten, bei der manuellen Erstellung des Fehlerbaums nicht berücksichtigt. Das heißt, nach der Durchführung der Fehlereffektsimulation muss der Fehlerbaum angepasst werden. Hierbei muss eine zusätzliche Relation zwischen der Fehlerursache des verfälschten Kamerabilds (pixel_fault) und der Fehlerauswirkung der verspäteten Erkennung der Geschwindigkeitsbeschränkung (late_hint) hinzugefügt werden. Diese Relation drückt die häufiger auftretende Fehlerpropagierung aus. Die vom Anwender berücksichtige Fehlklassifizierung (erroneous_hint) tritt hingegen eher selten auf.

Hieran lassen sich die Vorteile einer kombinierten Analyse mittels FTA und Fehlereffektsimulation erkennen. Die Fehlereffektsimulation hat gezeigt, dass neben der Verfälschung des Geschwindigkeitslimits, häufig eine verspätete Anzeige erfolgt. Dies liegt daran, dass vor allem Schilder, die weit weg sind, stark von einer Pixelverfälschung betroffen sind. Aufgrund der Tatsache, dass das Fahrzeug meist auf das Schild zu fährt, existiert eine Folge von Bildern mit der zu erkennenden Geschwindigkeitsbeschränkung. Hierdurch wird meist die erste Klassifizierung des Schilds verworfen, was zu einer verspäteten Erkennung der Geschwindigkeitsbeschränkung führt. Der Fall einer verfälschten Erkennung der Geschwindigkeit ist theoretisch möglich, die Menge der durchgeführten Simulationen konnte den Effekt aber nicht bestätigen. Hier zeigt sich der Vorteil der Kombination aus der manuellen, stark vom Anwender

bestimmten, Sicherheitsanalysen mit der simulationsgestützten Analyse.

7.2.4.3 Zyklenapproximierendes Modell

Im letzten betrachteten Modell sind die Bussysteme detaillierter beschrieben. Sowohl das MOST- als auch das FlexRay-Modell setzen abstrahiert die interne Struktur eines realen Controllers sowie das verwendete Kommunikationsprotokoll um. Es existieren Simulationseinheiten für Puffer, Datenverarbeitung und interne Busse. Das CAN-Modell ist abstrakter gehalten und modelliert vorrangig den konkurrierenden Buszugriff mit dem CSMA/CR-Algorithmus und die Frame-Struktur. Insgesamt besteht die komplette Systemsimulationsinstanz aus 77 Simulationseinheiten, 106 Verbindungen und 364 Parameterzuweisungen. Diese Werte verdeutlichen den Vorteil einer grafischen Spezifikation, da die Übersichtlichkeit der Konfigurationsdatei allein durch ihre Größe eingeschränkt ist.

Der CAN-Controller beinhaltet, in dieser Entwurfsstufe des Systems, einen Mechanismus zur zuverlässigen Übertragung der Daten. Detektiert der CAN-Controller einen CRC-Fehler, wird das ACK-Bit nicht gesetzt, was beim Sender zu einer erneuten Übertragung des CAN-Frames führt. Zur Fehlerinjektion wird der Datenstrom D_{sl} verwendet, der die detektierte Geschwindigkeitsbegrenzung an den Coasting Assistenten weiterleitet. Hierbei wird der Fehler in den CRC des CAN-Segments injiziert, um einen CRC-Fehler hervorzurufen. Da die durch den CRC unentdeckte Fehlermenge sehr klein ist, verursacht das Vorgehen einen ähnlichen Effekt wie die direkte Injektion in die Nutzdaten. Abbildung 7.36 zeigt das BTM des injizierten Fehlers. Für die Initialisierung wird ein Nachrichtenzähler mit null initialisiert. Hierzu wird eine, im BTM lokal genutzte, Variable verwendet. Bei der Transition von Zustand `eFree` in den Zustand `eState` wird der eigentliche Fehler injiziert. Die zugewiesene Aktion setzt den CRC des CAN-Frames auf $0x11$. Damit der Fehler nur den zu untersuchenden Datenstrom verfälscht, wird in der Bedingung des Zustandsübergangs die ID des aktuellen CAN-Frames überprüft. Da der CAN-Bus mehrere Daten überträgt, ist es wichtig, den korrekten Datenstrom anhand der CAN-ID zu identifizieren. Der Fehler wird in jede 5. CAN-Nachricht injiziert. Das Verhalten wird durch den Nachrichtenzähler modelliert. Hierzu wird zwischen Zustand `eState` und `eCount` bei jeder Nachricht mit der entsprechenden CAN-ID der Zähler inkrementiert. Hat der Zähler einen Wert größer als 5 erreicht, wird der Fehler injiziert. Sobald der Fehler injiziert wurde, wird dieser mit dem Beenden des Sendevorganges wieder aufgehoben und der Nachrichtenzähler zurück auf null gesetzt. Das Ende des Sendevorgangs wird durch das Auslesen des Busy-Flags des CAN-Controllers detektiert. Ein positiver Wert zeigt an, dass eine Übertragung im Gange ist. Sobald der Wert auf `false` wechselt, ist die Übertragung beendet und der Bus ist wieder frei. Dieses Verhalten wird sowohl bei der Freigabe der Injektion als auch dem Inkrementieren des Nachrichtenzählers ausgenutzt. Abbildung 7.37 zeigt die beobachtete Verzögerung der einzelnen CAN-Nachrichten sowie die durchschnittliche Verzögerung als horizontale Linie. Die Verzögerungen sind für die fehlerfreie Simulation als auch die durch die Fehlerinjektion modifizierte Simulation aufgezeigt. Hierbei werden lediglich die Nachrichten des injizierten Datenstroms, d. h. die Weiterleitung des Geschwindig-

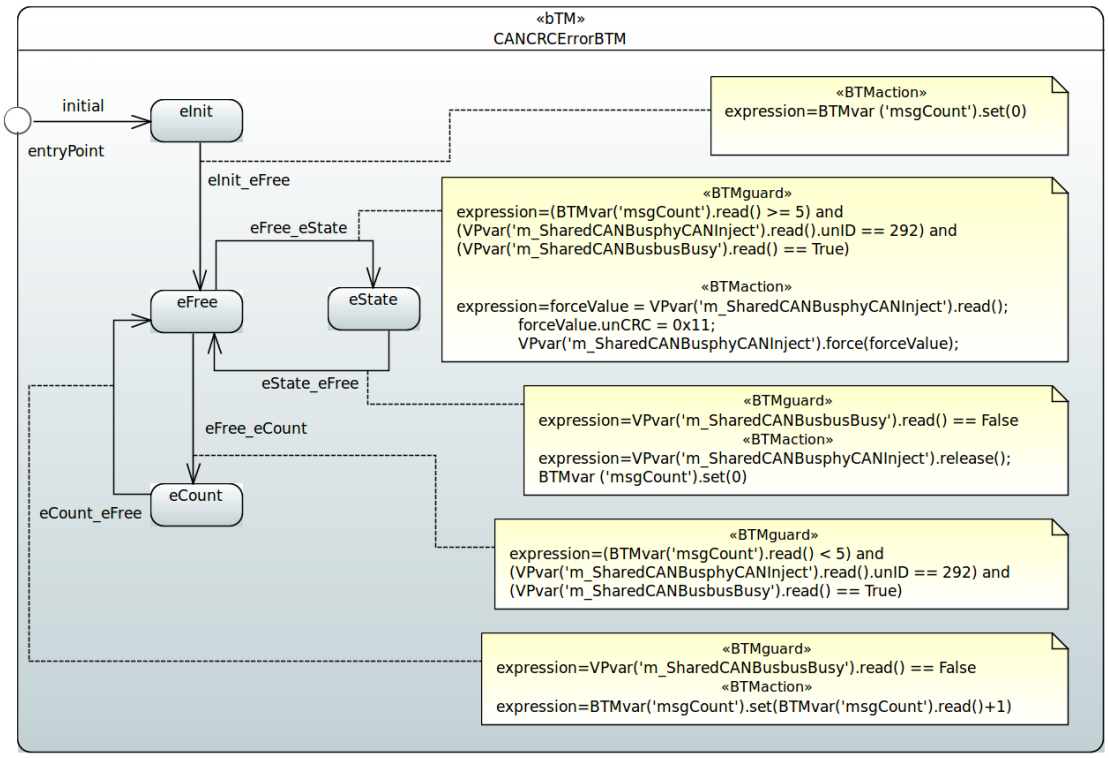


Abbildung 7.36: Fehlermodell eines CRC-Fehlers im physikalischen CAN-Frame

keitslimits, beobachtet. Die im Simulationsmodell beobachtete Verzögerung wird bei der Übergabe der Daten an den CAN-Controller gestartet. Dies bedeutet, dass Verzögerungen durch die Anwendung nicht beachtet werden. Die in Blau dargestellte Verzögerung konzentriert sich um 1 ms, wobei auch hier leichte Ausreißer enthalten sind. Dies liegt vor allem an Kollisionen auf dem physikalischen Bus. Im Falle der Fehlerinjektion sind die Ausreißer hingegen gravierender. Im schlechtesten Fall ist eine Verzögerung von bis zu 19 ms zu beobachten.

Die Analyse zeigt, dass neben der Unterstützung bei der Bestimmung der Fehlerpropagierung, die Fehlereffektsimulation auch eine quantitative Analyse der Fehlereffekte zulässt. Die quantitative Analyse unterstützt den Anwender in der Festlegung der Kritikalität von Fehlerauswirkungen.

Ein weiteres Beispiel wie der vorgestellte Ansatz den Anwender unterstützt, ist in Abbildung 7.38 dargestellt. Hierbei wird der resultierende Fehlereffekt auf die gemeldete Geschwindigkeitsbegrenzung untersucht, in Abhängigkeit des zuvor untersuchten Bitfehlers in den CAN-Daten. Die vorgestellte Evaluierung vergleicht zwei Implementierungsalternativen. Die Datenreihe woCRC_complete zeigt die auftretenden Fehlereffekte bei der Verwendung einer CAN-Implementierung ohne CRC und ohne Sendewiederholungen. Es ist ersichtlich, dass jeder auftretende Fehler sich auf die gemeldete Geschwindigkeitsbegrenzung auswirkt. Effekte auf die zeitliche Verfügbarkeit der Informationen existieren nicht. Da ein Schild meist in einer Abfolge von Bildern erkannt wird, wird in der Evaluierung weiterhin untersucht, wie häufig das erste Auftreten der Klassifizierung betroffen ist. Dies wird in der Datenreihe

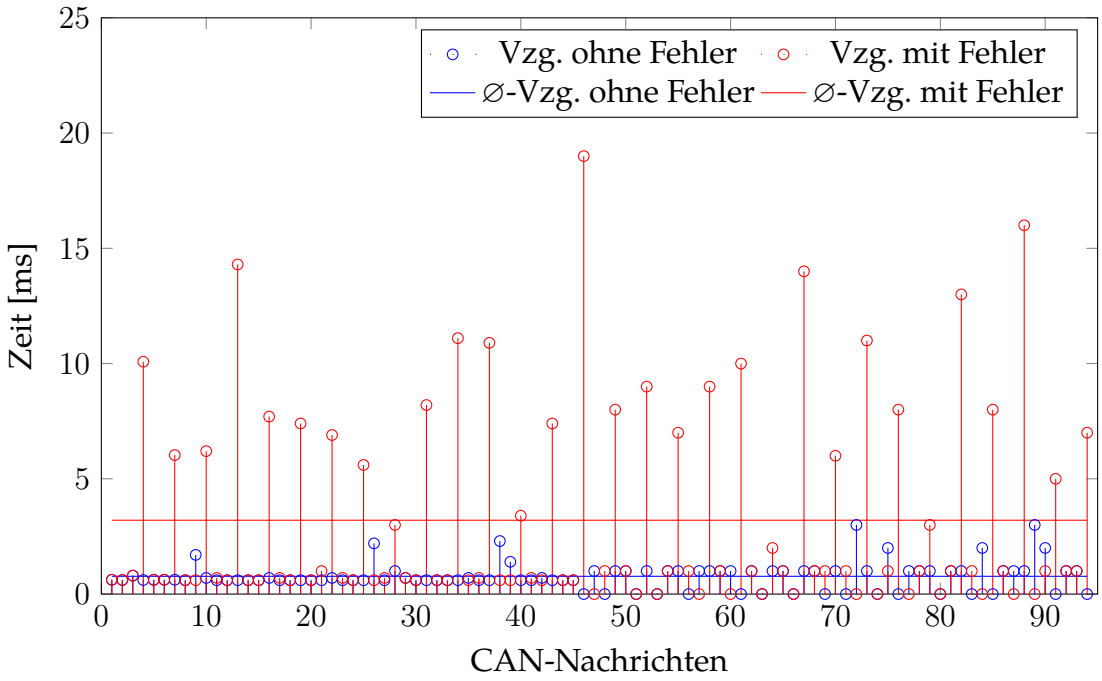


Abbildung 7.37: Verzögerung (Vzg.) des CAN-Datentransfers

woCRC_border angegeben. In den Datenreihen wCRC_complete und wCRC_border wird das gleiche Fehlermuster in ein System injiziert, welches das CAN-Protokoll mit CRC und Sendewiederholung implementiert. Es ist zu erkennen, dass keine verfälschten Datenwerte auftauchen, sondern alle Fehlereffekte die zeitliche Abfolge betreffen. Interessant ist, dass durch die zusätzliche Verzögerung, die in Abbildung 7.37 bereits genauer untersucht wurde, das System Geschwindigkeitsbegrenzungen früher erkennt als in der Nominalsimulation ohne Fehler. Dies liegt daran, dass die Fehlerklassifizierung periodisch ausgeführt wird. Innerhalb einer Periode klassifiziert der Algorithmus all vorliegende Bilder und leitet sie weiter. Sind alle Eingabebilder abgearbeitet, wird der Thread suspendiert und nach einer festen Zeit reaktiviert, um erneut alle in der Zwischenzeit empfangenen Bilder zu klassifizieren. Aufgrund der Tatsache, dass das Versenden der Bilder über einen blockierenden Aufruf erfolgt, wird hierdurch die Abarbeitungszeit verlängert. Hierdurch sind bereits neue Bildsegmente in der Eingangswarteschlange vorhanden, die in der Nominalsimulation erst mit der nächsten Aktivierung des Threads vorliegen. Die Thread-Wartezyklen reduzierten sich im Falle der Fehlerinjektion von 2199 auf 2196. Diese Reduktion bewirkt, dass einige Geschwindigkeitsbeschränkungen früher als in der Nominalsimulation zur Verfügung stehen.

Das Beispiel zeigt, wie der vorgestellte Ansatz unerwartete Fehlereffekte, in diesem Fall die frühere Erkennung von Geschwindigkeitsbegrenzungen, detektiert. Diese Erkenntnisse über die Fehlereffekte unterstützen den Anwender bei der Durchführung der benötigten Sicherheitsanalysen, wie z. B. eine FMEA.

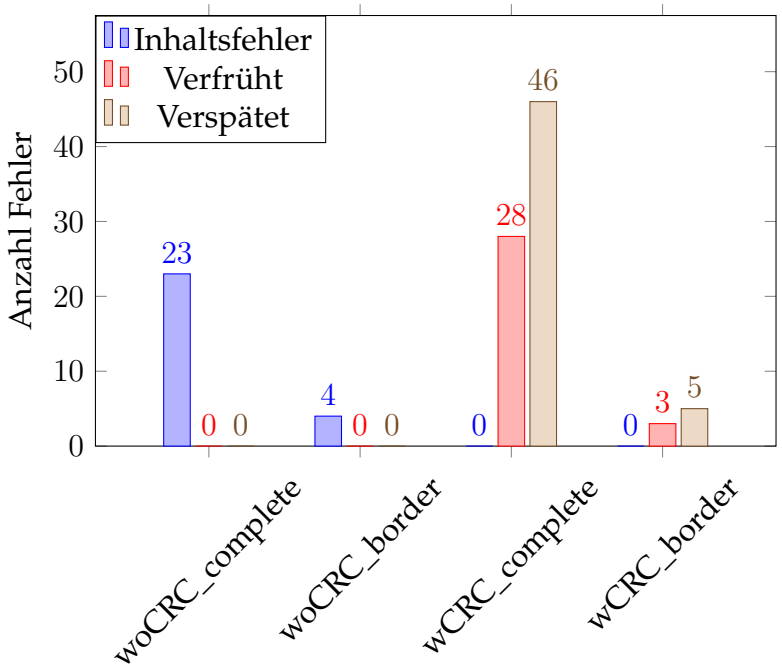


Abbildung 7.38: Systemarchitekturen und die beobachteten Fehlereffekte

7.2.5 Bewertung der Fallstudien

Die betrachteten Fallbeispiele verdeutlichen die Anwendbarkeit des vorgestellten Ansatzes. Im Rahmen einer Performanzbewertung wird die generelle Anwendbarkeit des vorgestellten Simulationsansatzes zur Herleitung quantitativer Bewertungen von Systemen dargelegt. Eine solche Betrachtung ist insbesondere zu Sicherheitsbewertung mittels FMEDA wichtig, welche die Ausfallwahrscheinlichkeiten und Verteilungen von Fehlerzuständen berücksichtigt. Insbesondere die entwickelten Ansätze zur dynamischen Spezifikation unterstützen solche Abschätzungen, durch die einfache Wiederholbarkeit und Modifikation von Simulationsläufen. Weitere Aspekte, die diese Fallstudie verdeutlicht, sind die Konzepte zur Steigerung der Wiederverwendbarkeit existierender Simulationsmodelle. Hierunter fallen die transaktionsbasierte Makroarchitektur, die modulare und parametrisierbare Mikroarchitektur als auch der Ansatz zur Abstraktion von physikalischer Eigenschaften. Der Aspekt der Wiederverwendung ist wichtig um den Aufwand zur Durchführung von simulationsbasierter Sicherheitsanalysen zu reduzieren, indem Modelle aus vorherigen Entwicklungsstufen oder aus weiteren Projekten angewandt werden können. Das zweite betrachtete Fallbeispiel zeigt die Anwendbarkeit des Fehlerinjektionsansatzes. Hierbei wird demonstriert, wie der Ansatz eine breite Anzahl von Abstraktionsebenen unterstützt. Dies ist vor allem für die angestrebte entwurfsbegleitende Sicherheitsbetrachtung wichtig. Das Fallbeispiel zeigt, wie Fehlereffekte über Systemmodelle unterschiedlicher Abstraktionsebene analysiert werden. Es verdeutlicht wie erste abstrakte Fehlereffektbeobachtungen auf detaillierte HW-Modelle heruntergebrochen werden können. Das letzte Fallbeispiel zeigt wie die vorgestellte Methodik über einen

grafischen Spezifikationsansatz erweitert und somit der Aufwand reduziert werden kann. Zusätzlich verdeutlicht das Fallbeispiel die Unterstützung im Rahmen einer FTA. Heutige Sicherheitsanalysen wie die FTA oder FMEA sind vorrangig dokumentengestützt, weswegen die durchführenden Personen häufig eine abstraktere Sicht auf das System bevorzugen. Um eine Unterstützung in diesem Prozess zu leisten, ist es wichtig die Ebene des Programmcodes des Simulationsmodells zu verlassen und eine abstraktere, stark an die benötigte Dokumentation angelehnte Repräsentation der Fehlereffektsimulation zu bieten.

Insbesondere die schwer abzuschätzenden bzw. zu identifizierenden Fehlerauswirkungen der Fallstudien zeigen die Notwendigkeit einer Unterstützung der manuellen Sicherheitsbewertung. Die Fallbeispiele verdeutlichen dies mit der Identifikation eines Fehlers, der einen verstärkenden Effekt besitzt. Die einfache Abschätzung des Fehlereffekts einer verzögerten Nachricht im Wassertankbeispiel und die damit angefachte Schwingung des Wasserstands, durch die alleinige Kenntnis des Systems ist sehr schwer. Ein weiteres Beispiel der Vorteile einer simulationsgestützten Analyse ist die nicht berücksichtigte Fehlerauswirkung in der FTA des untersuchten Fahrerassistenzsystems. Bei der Durchführung der FTA, waren der Ersteller der Simulationsmodelle, d.h. eine Person mit sehr guten Systemkenntnissen, als auch eine externe Person, die den FTA-Prozess systematisch strukturiert sowie im Rahmen der Analyse Systemkenntnisse erworben hat beteiligt. In mehreren vorbereitenden Treffen, die die Erstellung des Fehlerbaums als Ziel hatten, ist die fehlende Fehlerauswirkung nicht aufgefallen. Erst mit der anschließenden simulationsgestützten Überprüfung des Fehlerbaums ist die fehlende Fehlerauswirkung aufgefallen. Bei komplexeren Systemen mit mehreren Zulieferern und somit mehrere Systemexperten erhöht sich diese Problematik. In beiden Fällen leistet die simulationsbasierte Sicherheitsbewertung eine wichtige Unterstützung der traditionellen Sicherheitsbewertung.

Neben der reinen Methodik zur Unterstützung traditioneller Sicherheitsbewertungen durch Analysen basierend auf Simulationen, stand auch die Reduktion des Aufwands zur Durchführung dieser Analysen im Fokus der Arbeit. Hierbei ist vorrangig der Ansatz zur Fehlerinjektion zu nennen. Die Fallstudien demonstriert, wie bestehenden Simulationsmodell durch einfaches Ersetzen von Datentypen um die Fähigkeit zur Injektion von Fehlern zur Simulationslaufzeit erweitert werden können. In Kombination mit dem generischen Fehlerspezifikationsansatz, den sogenannten BTMs, und der Interpretation zur Laufzeit, bietet dies eine leichtgewichtige Erweiterungsmöglichkeit, die mit wenig manuellen Aufwand bestehende Simulationsmodell zur Analyse von Fehlerauswirkungen erweitert. Ohne diese Unterstützung müssten bestehende Simulationsmodelle umgeschrieben werden, um die Modifikation von Simulationsvariablen zu ermöglichen. Häufig werden diese applikationsspezifischen Injektoren, mit dem zu injizierenden Verhalten assoziiert, was eine Änderung für jeden Fehlerfall als auch für Änderungen am Simulationsmodell mit sich bringen können. Aspekte wie die Reaktivierung von Simulationsprozessen bei der Fehlerinjektion als auch die werteabhängige Bestimmung von Fehlereffekten und die somit verbundene Weiterleitung von Simulationsinformationen, über Simulationsmodulgrenzen, müsste für jedes Simulationsmodell implementiert werden. Mit dem vorge-

stellten Ansatz beschränken sich die Modifikationen auf das Platzieren der Injektoren und Wertesonden sowie die Spezifikation des zu injizierenden Fehlerverhaltens mittels des vorgestellten BTM-Ansatz. Neben der Einbringung von Fehlern reduziert auch die die Unterstützung bei der Auswertung der Fehlereffekte den Gesamtaufwand. Durch den vorgestellten Ansatz können sehr einfach unterschiedliche Simulationsläufe verglichen oder per CTL-Ausdrücke Zusicherungen überprüft werden. Der Anwender muss lediglich die gewünschte Information im Simulationsmodell mit Monitoren versehen. Die automatisierte Transformation in ein UPPAAL-Modell ermöglicht die sofortige Detektion von Unterschieden zwischen Simulationsaufzeichnungen als auch die einfache Erweiterung mit applikationsspezifischen Abfragen. Ohne diesen Ansatz müsste der Anwender auf applikationsspezifische Ansätze wie die Spezifikation und Überprüfung des zu erwartenden Systemverhaltens mittels einer Testbench oder auf assertion-basierte Ansätze zurückgreifen. Der vorgestellte Ansatz ermöglicht dieses Vorgehen zusätzlich, bietet aber durch den Vergleich von Simulationsaufzeichnungen ein aufwandreduziertes Vorgehen zur Eingrenzung von Fehlereffekten. Die letzten Erweiterungen zur Reduktion des Aufwands beziehen sich auf das eigentliche Simulationsmodell. Hierunter fallen Mechanismen wie die dynamische Konfiguration, sowie die daraus resultierenden Konzepte zur transaktionsbasierten Makroarchitektur als auch die modulare, parametrisierbare Mikro-Architektur. Dies wird vor allem im Fallbeispiel zur Performanzbewertung verdeutlicht. Zur aufwandsreduzierten Umsetzung dieser Konzepte wird ein modellgetriebener Ansatz verwendet. Dies reduziert den Implementierungsaufwand der Simulationsmodelle. Der Anwender kann z. B. auf die Implementierung eines Top-Moduls verzichten, wird bei der Steigerung der Wiederverwendbarkeit der Simulationsmodule unterstützt und die Implementierung zum dynamischen Einlesen von Konfigurationen wird automatisch generiert.

Zusammenfassend, zeigen die Fallbeispiele den Mehrwert der Nutzung simulationsgestützter Analysen zur Sicherheitsbewertung. Der Ansatz integriert unterschiedlichen Techniken zur Reduktion der Aufwände für den Anwender und somit die Schwelle zur Nutzung des Ansatzes. Ein Mehraufwand durch Nutzung des Ansatzes ist immer gegeben, aber gerade durch die Wiederverwendung von Simulationsmodellen und die vorgestellten Techniken wird dieser stark reduziert.

Kapitel 8

Zusammenfassung und Ausblick

Die stetig steigende Anzahl und Komplexität von sicherheitskritischen, elektronischen Systemen stellt die herkömmlichen Sicherheitsanalysen vor neue Herausforderungen. Durch die gestiegene Vernetzung der Systeme ist es notwendig, nicht nur isolierte Systemteile auf ihre funktionale Zuverlässigkeit zu untersuchen, sondern es ist notwendig, das Gesamtsystem zu analysieren. Die Sicherheitsanalysen müssen die Wechselwirkungen zwischen Systemteilen, sogar mit entfernten Systemen, beachten. Im gleichen Schritt werden die einzelnen Systemteile in ihrer Funktion und in ihrer Architektur immer komplexer. Sicherheitsexperten müssen bei den herkömmlichen Sicherheitsanalysen die internen und externen Systemzusammenhänge kennen und in die Analyse einbringen. Hierdurch steigt nicht nur der Aufwand zur Ausführung der Sicherheitsbewertung kontinuierlich, sondern es ist sehr herausfordernd alle Systemaspekte zu berücksichtigen.

Häufig kommen Komponenten von Drittanbietern zum Einsatz, um die Komplexität während der Entwicklung beherrschbar zu halten. Aufgrund der Tatsache, dass die Sicherheitsanalyse das Gesamtsystem betrachten muss, ist es notwendig, dass die Analyse diese fremd entwickelten Systemteile mit einbezieht. Die Sicherheitsanalyse muss zumindest die Weiterleitung bzw. Behebung von Fehlern für die fremd entwickelten Systemteile abschätzen.

Eine weitere Herausforderung stellt die frühe Verfügbarkeit von Sicherheitsanalysen dar. Um die Entwicklung des Systems, unter dem Gesichtspunkt der funktionalen Sicherheit, zu leiten, ist es erforderlich, bereits zur Entwicklung des Systems unterschiedliche Systemalternativen zu bewerten. Mithilfe solcher Bewertungen ist es möglich, Entwurfsentscheidungen zu treffen und mögliche Neuentwürfe zu unterbinden.

In der vorliegenden Arbeit wird ein Simulationsrahmenwerk entwickelt, das den Anwender in die Lage versetzt, Sicherheitsanalysen mittels Simulation zu unterstützen. Hierbei wird ein Simulationsmodell des zu entwickelnden Systems verwendet, in das unterschiedliche Fehler injiziert werden und die Reaktion des Systems ausgewertet wird. Ein solcher Ansatz berücksichtigt automatisch alle Systemteile sowie deren Interaktion, wenn diese im Simulationsmodell modelliert sind. Mock-Objekte entfernen irrelevante Systemteile aus der Analyse, um den Aufwand zu reduzieren.

Auch bietet ein solcher Ansatz Vorteile in Szenarien, in denen Drittanbieter Systemteile beisteuern. Anstelle das Know-how der Systemexperten direkt in die Sicherheitsanalyse einfließen zu lassen, wird es über die zur Verfügung gestellten Simulationsmodell bereitgestellt. Hierbei liefern die Drittanbieter neben dem eigentlichen Teilsystem, zusätzlich ein Simulationsmodell. Durch die mitgelieferten Simulationsmodelle berücksichtigt die Sicherheitsanalyse des Gesamtsystems alle Eigenschaften und Abhängigkeiten der Teilsysteme. Vorteil ist, dass die Anbieter die Implementierungsdetails des Systems und demnach der Simulationsmodelle durch Verschlüsselung oder durch Abstraktion verbergen können.

Ein wichtiger Aspekt ist, dass die Systemsimulation bereits sehr früh im Entwurf zur Verfügung steht und somit alle Phasen des Entwurfs unterstützen. Die Simulationsmodelle unterstützen in frühen Entwurfsphasen die Entwickler bei Entwurfsentscheidungen, indem sie unterschiedliche Alternativen simulieren und bewerten. Ebenso unterstützt der Ansatz bei der Exploration des Entwurfsraums, da er unterschiedliche Entwürfe, wie z. B. unterschiedliche Parametrisierungen, mit Hilfe eines leicht konfigurierbaren Simulationsmodells bewertet.

Um eine umfangreiche Unterstützung zu gewähren, muss das Simulationsrahmenwerk gewisse Eigenschaften aufweisen. Zur Unterstützung der Exploration, unterschiedlicher Systemalternativen, besitzt die Simulation eine modulare, parametrisierbare Struktur. Durch die Verwendung von standardisierten Interfaces wird ein hoher Grad an Wiederverwendbarkeit von Simulationskomponenten erzielt. Hierzu wird das Konzept der transaktionsbasierten Makroarchitektur eingeführt. Um ein Höchstmaß an Flexibilität zu erreichen, wird ein Ansatz zur Abstraktion von physikalischen Eigenschaften angewandt, der es erlaubt die applikationsspezifischen Eigenschaften der genutzten Ebene zu abstrahieren.

Die flexible Systemsimulation wird mit Mechanismen zur Stimulation von Fehlern erweitert. Hierzu wird ein Fehlerinjektorkonzept bereitgestellt, dass es erlaubt bestehende Simulationsmodell ohne Änderung der Funktionalität bzw. deren Struktur, für die Fehlerinjektion zu erweitern. Wichtiger Punkt hierbei ist die Bereitstellung eines minimal invasiven Ansatzes. Das Framework stellt einen Mechanismus zur Spezifikation eines Auslöseverhaltens der Simulation bei Fehlerinjektion oder Fehlerbehebung bereit. Für die Fehlerbehebung werden unterschiedliche Strategien zur Fehlerbehebung unterstützt. Die Mechanismen sind notwendig, um Fehler auf unterschiedlichen Abstraktionsebenen zu unterstützen. Vor allem bei abstrakten Simulationsmodellen mit unterschiedlichen applikationsspezifischen Fehlern.

Neben der Injektion des Fehlers ist es notwendig ein umfassendes Spezifikationsformat, für das zu injizierende Fehlerverhalten, bereitzustellen. Vor allem bei sehr abstrakten Simulationsmodellen überbrückt die Fehlerspezifikation die Simulationsabstraktion, in die der Fehler injiziert wird, mit bekannten Low-Level-Fehlermodellen. In der Arbeit wird eine zustandsbasierte Fehlerspezifikation, das sogenannte Behavioral Threat Model (BTM), vorgestellt. Diese Spezifikation wird zur Laufzeit ausgewertet und die in der Simulation enthaltenen Fehlerinjektoren angesteuert.

Neben der Fehlerinjektion benötigt die Fehlereffektsimulation Ansätze zur Detektion der Fehlerauswirkungen. Hierzu beinhaltet das Rahmenwerk eine Anbindung

an einen Model Checker, der genutzt wird, um während der Simulation aufgezeichnete Traces auszuwerten. Der aktuelle Ansatz erlaubt die Detektion von allgemeinen Fehlermodi, wie Verspätungs- oder Inhaltsfehler. Daneben ist es möglich, eine zustandsbasierte Detektion von applikationsspezifischen Fehlerreaktionen bereitzustellen. Mit diesem Ansatz ist es möglich, automatisiert die Systemreaktion auf Fehler zu überprüfen.

Abgerundet wird der vorgestellte Ansatz durch die Integration in modellgetriebene Entwurfsabläufe. Hierbei ist es möglich, Teile des Simulationsmodells automatisch zu generieren, wie z. B. den Simulationsquellcode oder die verwendeten Konfigurationsdateien. Dies versetzt den Anwender in die Lage mittels grafischer Beschreibung, die Fehlereffektsimulation zu konfigurieren und auszuführen. Ein weiterer Vorteil des Ansatzes ist die inhärente Dokumentation einer solchen grafischen Spezifikation.

Unterschiedliche Fallstudien belegen die Anwendbarkeit des Ansatzes. Hierbei wird anhand einer Performanzanalyse von zwei Übertragungsprotokollen, die Vielseitigkeit des Simulationsrahmenwerks gezeigt. Mithilfe einer Regelung eines Wassertanks wird der Fehlerinjektions- und Fehlerspezifikationsansatz vorgestellt. Es wird aufgezeigt, wie der Anwender Fehlerinjektoren zu bestehenden Simulationsmodellen hinzufügt und wie Fehlerspezifikationen eine Vielzahl von unterschiedlichen Fehlern realisieren. Abgeschlossen wird die Arbeit durch die Analyse einer Fahrzeugbusarchitektur. Hierbei wird anhand von drei unterschiedlichen Modellen, des zu entwickelnden Systems, die durchgängige Anwendbarkeit des Ansatzes gezeigt. Für ein dediziertes Abstraktionsniveau wird die Möglichkeit zur Unterstützung der FTA aufgezeigt. Die unterschiedlichen Beispiele zeigen die Anwendbarkeit, den resultierenden Aufwand und die Vorteile bei der Unterstützung von Sicherheitsanalysen bei der Verwendung des vorgestellten Ansatzes.

8.1 Ausblick

Der in dieser Arbeit vorgestellte Ansatz bietet die Grundlage zur automatisierten Bewertung von Systemen unter dem Gesichtspunkt der funktionalen Sicherheit. Bei Erreichen eines gewissen Grades an Automatisierung könnte in Kombination von Explorationsalgorithmen ein automatisiertes Einbringen von Sicherheitsmechanismen in Systemen ermöglicht werden. In diesem Kontext gibt es einige Herausforderungen, die ergänzend zu dieser Arbeit betrachtete werden müssen.

Ein wichtiger Aspekt ist die automatisierte Erstellung von Prüfbedingungen bzw. Bewertungsfunktionen. In dieser Arbeit wird ein Ansatz zur Fehlereffektbeobachtung sowie mehrere Ansätze zur Fehlereffektbewertung vorgestellt. Für eine automatisierte Bewertung, müssten die Bewertungsgrundlage, z. B. die UPPAAL-Queries automatisiert aus Systemanforderungen generiert werden.

Ein weiter Aspekt, der zu einer vollständigen Automatisierung adressiert werden müsste, ist die Bereitstellung von Explorationsstrategien. Bei der Einbringung und Bewertung von Sicherheitsmechanismen existiert eine Vielzahl von Freiheitsgraden: Welche Absicherungsmechanismen sollen angewandt werden, an welchen Stellen sollen sie platziert werden, wie werden sie parametrisiert und gegen welche Fehler

sollen sie geprüft werden. Gerade der letzte Punkt wird in dieser Arbeit mit einer durchgängigen Methode zur Spezifikation von Fehlerverhalten berücksichtigt. Aber die Wahl einer geeigneten Parametrisierung der Fehlerspezifikation wird in dieser Arbeit nicht adressiert und muss im Rahmen der Explorationsstrategien gelöst werden.

Literaturverzeichnis

[1] Accellera (2015). Universal Verification Methodology (UVM) 1.2 User's Guide.

[2] Alur, R. und Dill, D. L. (1994). A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235.

[3] Avizienis, A., Laprie, J.-C., Randell, B., und Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33.

[4] Aynsley, J. (2009). *OSCI TLM-2.0 Language Reference Manual*, JA32. Auflage.

[5] Barlow, T. J., Latham, S., McCrae, I. S., und Boulter, P. G. (2009). *A reference book of driving cycles for use in the measurement of road vehicle emissions*. TLR Limited: Published project report PPR354.

[6] Batteux, M., Prosvirnova, T., Rauzy, A., und Kloul, L. (2013). The AltaRica 3.0 project for model-based safety assessment. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, Seiten 741–746.

[7] Behrmann, G., David, A., und Larsen, K. G. (2006). A tutorial on UPPAAL 4.0.

[8] Beltrame, G., Bolchini, C., Fossati, L., Miele, A., und Sciuto, D. (2008). ReSP: A non-intrusive Transaction-Level Reflective MPSoC Simulation Platform for design space exploration. In *Proc. of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Seiten 673–678.

[9] Beltrame, G., Fossati, L., und Sciuto, D. (2009). ReSP: A Nonintrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(12):1857–1869.

[10] Bernardi, S., Merseguer, J., und Petriu, D. C. (2012). Dependability Modeling and Analysis of Software Systems Specified with UML. *ACM Comput. Surv.*, 45(1):2:1–2:48.

[11] Birolini, A. (2013). *Reliability engineering: theory and practice: with 190 figures, 60 tables, 140 examples, and 70 problems for homework*. Heidelberg: Springer.

[12] Black, D. C., Donovan, J., Bunton, B., und Keist, A. (2009). *SystemC: From the Ground Up, Second Edition*. Springer Publishing Company, Incorporated, 2nd. Auflage.

[13] Bless, R. und Doll, M. (2004). Integration of the freebsd TCP/IP-stack into the discrete event simulator OMNET++. In *Simulation Conference. Proceedings of the 2004 Winter*.

[14] Bohn, B., Garcke, J., Iza-Teran, R., Paprotny, A., Peherstorfer, B., Schepsmeier, U., und Thole, C. (2013). Analysis of Car Crash Simulation Data with Nonlinear Machine Learning Methods. In *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013*, Seiten 621–630.

[15] Bolchini, C., Miele, A., und Sciuto, D. (2008). Fault Models and Injection Strategies in SystemC Specifications. In *2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, Seiten 88–95.

[16] Bombieri, N., Fummi, F., und Pravadelli, G. (2008). A Mutation Model for the SystemC TLM 2.0 Communication Interfaces. In *Design, Automation and Test in Europe, 2008. DATE '08*, Seiten 396–401.

[17] Burger, A. (2015). *Zielgerichtete Generierung valider Systemvarianten für eingeschränkte Entwurfsräume*. Dissertation, Eberhard-Karls-Universität Tübingen.

[18] Burger, A., Reiter, S., Viehl, A., Bringmann, O., und Rosenstiel, W. (2015). Systemmodellierung zur Fehlereffektsimulation. In *27. Workshop für Testmethoden und Zuverlässigkeit von Schaltungen und Systemen (TuZ 2015)*.

[19] Burger, A., Reiter, S., Viehl, A., Bringmann, O., und Rosenstiel, W. (2016). Constraint-based Platform Variant Specification for Early System Verification. In *International MOST Conference & Exhibition 2016*.

[20] Burger, A., Viehl, A., Braun, A., Haedicke, F., Große, D., Bringmann, O., und Rosenstiel, W. (2014). Constraint-based platform variants specification for early system verification. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[21] Burton, M., Aldis, J., Günzel, R., und Klingauf, W. (2007). Transaction Level Modelling: A reflection on what TLM is and how TLMs may be classified. In *FDL*, Seiten 92–97. ECSI.

[22] Cacilo, A., Schmidt, S., Wittlinger, P., Hermann, F., Sawade, O., Doderer, H., und Scholz, V. (2015). *Hochautomatisiertes Fahren auf Autobahnen - Industriepolitische Schlussfolgerungen*. Frauenhofer IAO.

[23] Cai, L. und Gajski, D. (2003). Transaction level modeling: an overview. In *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*, Seiten 19–24.

[24] Cassandras, C. G. und Lafortune, S. (2006). Introduction to Discrete Event Systems.

[25] Chang, K.-C., Wang, Y.-C., Hsu, C.-H., Leu, K.-L., und Chen, Y.-Y. (2008). System-Bus Fault Injection Framework in SystemC Design Platform. In *Secure System Integration and Reliability Improvement, 2008. SSIRI '08. Second International Conference on*, Seiten 211–212.

[26] Chang, K.-J. und Chen, Y.-Y. (2007). System-Level Fault Injection in SystemC Design Platform.

[27] Charette, R. N. (2009). This Car Runs on Code. *IEEE Spectrum*.

[28] Chen, Y.-Y., Wang, Y.-C., und Peng, J.-M. (2008). SoC-level fault injection methodology in SystemC design platform. In *System Simulation and Scientific Computing, 2008. ICSC 2008. Asia Simulation Conference - 7th International Conference on*, Seiten 680–687.

[29] Cuenot, P., Frey, P., Johansson, R., Lönn, H., Papadopoulos, Y., Reiser, M.-O., Sandberg, A., Servat, D., Kolagari, R. T., Törngren, M., und et al. (2007). The EAST-ADL Architecture Description Language for Automotive Embedded Software. In *Model-Based Engineering of Embedded Real-Time Systems*, Lecture Notes in Computer Science, Seiten 297–307. Springer.

[30] Department of defense (1980). MIL-STD-1629A, Procedures for Performing a Failure Mode, Effects and Criticality Analysis.

[31] DIN (2016). DIN EN 50126-1:2016-03, Bahnanwendungen - Spezifikation und Nachweis von Zuverlässigkeit, Verfügbarkeit, Instandhaltbarkeit und Sicherheit (RAMS) - Teil 1: Generischer RAMS Prozess. Norm, Deutsches Institut für Normung.

[32] Dingel, J., Rudie, K., und Dragert, C. (2009). Bridging the Gap: Discrete-Event Systems for Software Engineering (Short Position Paper). In *Proceedings of the 2Nd Canadian Conference on Computer Science and Software Engineering*, C3S2E '09, Seiten 67–71.

[33] Dunkels, A. (2001). Design and Implementation of the lwIP TCP/IP Stack.

[34] Fin, A., Fummi, F., und Pravadelli, G. (2001). AMLETO: a multi-language environment for functional test generation. In *Test Conference, 2001. Proceedings. International*, Seiten 821–829.

[35] France, R. und Rumpe, B. (2007). Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering*, FOSE '07, Seiten 37–54.

[36] Freescale Semiconductor, I. (2013). Future Advances in Body Electronics.

[37] Frost & Sullivan (2017). Global Autonomous Driving Market Outlook.

[38] Gajski, D. D. und Kuhn, R. H. (1983). Guest Editors' Introduction: New VLSI Tools. *Computer*, 16(12):11–14.

[39] Gasser, T. M., Arzt, C., Ayoubi, M., Bartels, A., Bürkle, L., Eier, J., Flemisch, F., Häcker, D., Hesse, T., Huber, W., Lotz, C., Maurer, M., Ruth-Schumacher, S., Schwarz, J., und Vogt, W. (2012). Rechtsfolgen zunehmender Fahrzeugautomatisierung. *Berichte der Bundesanstalt für Straßenwesen*.

[40] Ghenassia, F. (2006). *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ,USA.

[41] Grechenig, T. (2010). *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten*. IT Informatik. Pearson Studium.

[42] Gurke, S. (2009). Nationale Roadmap Embedded Systems. *ZVEI - Zentralverband Elektrotechnik- und Elektronikindustrie e.V., Kompetenzzentrum Embedded Software & Systems*.

[43] Gérard, S., Dumoulin, C., Tessier, P., und Selic, B. (2010). 19 Papyrus: A UML2 Tool for Domain-Specific Language Modeling. In Giese, H., Karsai, G., Lee, E., Rumpe, B., und Schätz, B., Herausgeber, *Model-Based Engineering of Embedded Real-Time Systems*, Band 6100 in *Lecture Notes in Computer Science*, Seiten 361–368. Springer Berlin Heidelberg.

[44] Höfig, K., Zeller, M., und Heilmann, R. (2015). ALFRED: A Methodology to Enable Component Fault Trees for Layered Architectures. In *41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Seiten 167–176.

[45] Hsueh, M.-C., Tsai, T. K., und Iyer, R. K. (1997). Fault Injection Techniques and Tools. *Computer*, 30(4):75–82.

[46] IEC (2010). IEC 61508:2010, Functional safety of electrical/electronic/programmable electronic safety-related systems – Parts 1 to 7. Norm , International Electronical Commission.

[47] IEEE (2006). IEEE 1364-2001, Standard for Verilog Hardware Description Language. *Std 1364-2005 (Revision of IEEE Std 1364-2001)*.

[48] IEEE (2009). IEEE 1076-2002, Standard VHDL Language Reference Manual. *Std 1076-2008 (Revision of IEEE Std 1076-2002)*.

[49] IEEE (2012). ANSI/IEEE 1666-2005, IEEE Standard for Standard SystemC Language Reference Manual. *Std 1666-2005 (Revision of IEEE Std 1666-2005)*.

[50] IPG Automotive GmbH (2015). *CarMaker Software, Version 5.0.2*. Karlsruhe, Germany.

[51] ISO (2011a). ISO 26262, Road vehicles – Functional safety. Norm, International Organization for Standardization, Geneva, Switzerland.

[52] ISO (2011b). ISO/IEC 14882:2011, Information technology – Programming languages – C++. Norm, International Organization for Standardization.

[53] Jenn, E., Arlat, J., Rimen, M., Ohlsson, J., und Karlsson, J. (1994). Fault injection into VHDL models: the MEFISTO tool. In *Fault-Tolerant Computing, FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, Seiten 66–75.

[54] Jia, Y. und Harman, M. (2011). An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678.

[55] Joshi, J., Reiter, S., Viehl, A., Bringmann, O., und Rosenstiel, W. (2013). Evaluation Framework for MOST Based Driver Assistance Systems Based on Virtual Prototypes. In *International MOST Conference & Exhibition*.

[56] Josuttis, N. M. (1999). *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[57] Kaiser, B., Liggesmeyer, P., und Mäckel, O. (2003). A New Component Concept for Fault Trees. In *Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software - Volume 33*, SCS '03, Seiten 37–46.

[58] Kammler, D., Guan, J., Ascheid, G., Leupers, R., und Meyr, H. (2009). A Fast and Flexible Platform for Fault Injection and Evaluation in Verilog-Based Simulations. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, Seiten 309–314.

[59] Kitchin, J. F. (1988). Practical Markov modeling for reliability analysis. In *Annual Reliability and Maintainability Symposium, 1988, Proceedings.*, Seiten 290–296.

[60] Koch, A., Reiter, S., Viehl, A., Bringmann, O., und Rosenstiel, W. (2015). A Performance Evaluation Framework for MOST Ethernet supported by Virtual Prototyping Technology. In *International MOST Conference and Exhibition*.

[61] Koehler, S., Viehl, A., Bringmann, O., und Rosenstiel, W. (2012). Optimized recuperation strategy for (Hybrid) Electric Vehicles based on intelligent sensors. In *Control, Automation and Systems (ICCAS), 2012 12th International Conference on*.

[62] Koehler, S., Viehl, A., Bringmann, O., und Rosenstiel, W. (2013). Advanced driver assistance system for optimized recuperation under consideration of parameter uncertainties. In *Intelligent Vehicles Symposium (IV), 2013 IEEE*, Seiten 738–743.

[63] Kropf, T. (2013). *Introduction to Formal Hardware Verification*. Springer Berlin Heidelberg.

[64] Kuroki, Y., Yoo, M., und Yokoyama, T. (2016). A Simulink to UML model transformation tool for embedded control software development. In *2016 IEEE International Conference on Industrial Technology (ICIT)*, Seiten 700–706.

[65] Laprie, J. (2013). *Dependability: Basic Concepts and Terminology: In English, French, German, Italian and Japanese*. Dependable Computing and Fault-Tolerant Systems. Springer Vienna.

[66] Laufenberg, J., Reiter, S., Viehl, A., Bringmann, O., Kropf, T., und Rosenstiel, W. (2016). Combining Graph-based Guidance with Error Effect Simulation for Efficient Safety Analysis. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. Interactive Presentation.

[67] Laufenberg, J., Reiter, S., Viehl, A., Bringmann, O., und Rosenstiel, W. (2015). Graph Guided Error Effect Simulation. In *Proceeding of the 1st International ESWEEK Workshop on Resiliency in Embedded Electronic Systems*.

[68] Lisherness, P. und Cheng, K.-T. (2010). SCEMIT: A SystemC error and mutation injection tool. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, Seiten 228–233.

[69] Lu, W. und Radetzki, M. (2011). Efficient Fault Simulation of SystemC Designs. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*, Seiten 487–494.

[70] Marinescu, P. D. und Candea, G. (2009). LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, Seiten 379–388.

[71] Mei, K. C. Y. (1974). Bridging and Stuck-At Faults. *IEEE Transactions on Computers*, C-23(7):720–727.

[72] Meneu, J. J. (2016). Altera: Fortschritt in der Automobiltechnik.

[73] Misera, S. (2007). Simulation von Fehlern in digitalen Schaltungen mit SystemC.

[74] Misera, S. und Vierhaus, H. (2004). FIT - A Parallel Hierarchical Fault Simulation Environment. In *Parallel Computing in Electrical Engineering, 2004. PARELEC 2004. International Conference on*, Seiten 289–294.

[75] Misera, S., Vierhaus, H., Breitenfeld, L., und Sieber, A. (2006). A Mixed Language Fault Simulation of VHDL and SystemC. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, Seiten 275–279.

[76] Misera, S., Vierhaus, H., und Sieber, A. (2007). Fault Injection Techniques and their Accelerated Simulation in SystemC. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, Seiten 587–595.

[77] Modarres, M., Kaminskiy, M., und Krivtsov, V. (1999). *Reliability Engineering and Risk Analysis: A Practical Guide*. Taylor & Francis.

[78] Moon, T. (2005). *Error Correction Coding: Mathematical Methods and Algorithms*. Wiley.

[79] Oetjens, J. H., Bannow, N., Becker, M., Bringmann, O., Burger, A., Chaari, M., Chakraborty, S., Drechsler, R., Ecker, W., Grüttner, K., Kruse, T., Kuznik, C., Le, H. M., Mauderer, A., Müller, W., Müller-Gritschneder, D., Poppen, F., Post, H., Reiter, S., Rosenstiel, W., Roth, S., Schlichtmann, U., von Schwerin, A., Tabacaru, B. A., und Viehl, A. (2014). Safety Evaluation of Automotive Electronics Using Virtual Prototypes: State of the Art and Research Challenges. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, DAC '14, Seiten 113:1–113:6, New York, NY, USA. ACM.

[80] OMG (2011). A UML Profile for MARTE Modeling and Analysis of Real-Time Embedded Systems. Spezifikation, Object Management Group.

[81] OMG (2015a). OMG Meta Object Facility (MOF) Core Specification. Spezifikation, Object Management Group. Version 2.5.

[82] OMG (2015b). OMG Systems Modeling Language (OMG SysML). Spezifikation, Object Management Group. Version 1.4.

[83] OMG (2015c). OMG Unified Modeling Language. Spezifikation, Object Management Group. Version 2.5.

[84] OMG (2016). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Spezifikation, Object Management Group. Version 1.3.

[85] OpenCores (2017). Plasma - most MIPS I(TM).

[86] Papadopoulos, Y. und McDermid, J. A. (1999). Hierarchically Performed Hazard Origin and Propagation Studies. In Felici, M. und Kanoun, K., Herausgeber, *Computer Safety, Reliability and Security*, Seiten 139–152. Springer Berlin Heidelberg.

[87] Patel, J. H. (1998). Stuck-at fault: a fault model for the next millennium. In *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*.

[88] Perez, J., Azkarate-askasua, M., und Perez, A. (2010). Codesign and Simulated Fault Injection of Safety-Critical Embedded Systems Using SystemC. In *2010 European Dependable Computing Conference*, Seiten 221–229.

[89] Petre, M. (2013). UML in Practice. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, Seiten 722–731.

[90] Potyra, S., Sieh, V., und Cin, M. D. (2007). Evaluating Fault-tolerant System Designs Using FAUmachine. In *Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems*, EFTS '07.

[91] Pour, G. (1998). Component-based software development approach: new opportunities and challenges. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*.

[92] Prosvirnova, T. und Rauzy, A. (2013). AltaRica 3.0 project: compile Guarded Transition Systems into Fault Trees. In *European Safety and Reliability Conference, ESREL 2013*.

[93] Rauzy, A. B. (2008). Guarded transition systems: A new states/events formalism for reliability studies. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, 222(4):495–505.

[94] Reiter, S., Becker, M., Bringmann, O., Burger, A., und et al. (2015a). Fehlereffektsimulation mittels virtueller Prototypen. In *27. Workshop für Testmethoden und Zuverlässigkeit von Schaltungen und Systemen (TuZ 2015)*.

[95] Reiter, S., Braun, A., Hauke, R., Leonhardi, A., Bringmann, O., und Rosenstiel, W. (2011). Automated Performance Evaluation of the MOST High Protocol Using Virtual Prototypes. In *International MOST Conference & Exhibition 2011*.

[96] Reiter, S., Bringmann, O., und Rosenstiel, W. (2012). Reliability analysis of a MOST based advanced driver assistance system using virtual prototypes. In *International MOST Conference & Exhibition 2012*.

[97] Reiter, S., Burger, A., Viehl, A., Bringmann, O., und Rosenstiel, W. (2014). Virtual Prototyping Evaluation Framework for Automotive Embedded Systems Engineering. In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*.

[98] Reiter, S., Pressler, M., Viehl, A., Bringmann, O., und Rosenstiel, W. (2013). Reliability assessment of safety-relevant automotive systems in a model-based design flow. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, Seiten 417–422.

[99] Reiter, S., Viehl, A., Bringmann, O., und Rosenstiel, W. (2015b). White-Box Error Effect Simulation for Assisted Safety Analysis. In *Proceedings of the 18th Euromicro Conference on Digital System Design*. Poster Presentation.

[100] Reiter, S., Viehl, A., Bringmann, O., und Rosenstiel, W. (2016). Fault Injection Ecosystem for Assisted Safety Validation of Automotive Systems. In *18th IEEE International High-Level Design Validation and Test Workshop*.

[101] Rothbart, K., Neffe, U., Steger, C., Weiss, R., Rieger, E., und Muehlberger, A. (2004). High level fault injection for attack simulation in smart cards. In *Test Symposium, 13th Asian*, Seiten 118–121.

[102] Sayed-Mouchaweh, M. (2014). Discrete Event Systems : Diagnosis and Diagnosability.

[103] Schmidt, D. C. (2006). Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31.

[104] Schram, R., Williams, A., van Ratingen, M., Ryrberg, S., und Sferco, R. (2015). Euro NCAP's First Step to Assess Autonomous Emergency Braking (AEB) for Vulnerable Road Users. *ESV*.

[105] Shafik, R., Rosinger, P., und Al-Hashimi, B. (2008). SystemC-based Minimum Intrusive Fault Injection Technique with Improved Fault Representation. In *International On-line Test Symposium (IOLTS)*, Seiten 99–104. IEEE Computer Society.

[106] Sieh, V., Tschache, O., und Balbach, F. (1997). VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, Seiten 32–36.

[107] Society, I. C. (2014). IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows. *IEEE Std 1685-2009*.

[108] Sohofi, H. und Navabi, Z. (2015). System-level assertions: approach for electronic system-level verification. *IET Computers Digital Techniques*, 9(3):142–152.

[109] Sutherland, M. (2004). Developing Complex Systems using DOORS and UML. In *Telelogic 2004 User Group Conference -Americas and Asia/Pacific*.

[110] Synopsys, Inc. (2011). Datasheet: DesignWare TLM Library.

[111] Synopsys, Inc. (2016). Synopsys' New Virtualizer Studio Integrated Development Environment Accelerates Virtual Prototyping Productivity.

[112] Tabacaru, B.-A., Chaari, M., Ecker, W., und Kruse, T. (2014). Runtime Fault-Injection Tool for Executable SystemC Models. In *DVCon India, Bangalore, IN*.

[113] Tisler, P. (2006). *Aspects of weather simulation by numerical process*. Dissertation, University of Helsinki, Faculty of Science, Department of Physical Sciences.

[114] Vedder, B., Arts, T., Vinter, J., und Jonsson, M. (2007). Combining Fault-Injection with Property-Based Testing. In *Proceedings of International Workshop on Engineering Simulations for Cyber-Physical Systems*, ES4CPS '14, Seiten 1:1–1:8.

[115] Waez, M. T. B., Dingel, J., und Rudie, K. (2013). A survey of timed automata for the development of real-time systems. *Computer Science Review*, 9:1–26.

[116] Wang, F. und Agrawal, V. D. (2008). Single Event Upset: An Embedded Tutorial. In *21st International Conference on VLSI Design (VLSID 2008)*, Seiten 429–434.

[117] Weinkopf, J. T., Harbich, K., und Barke, E. (2006). Parsifal: A Generic and Configurable Fault Emulation Environment with Non-Classical Fault Models. In *2006 International Conference on Field Programmable Logic and Applications*, Seiten 1–6.

[118] Willems, J. (2007). The Behavioral Approach to Open and Interconnected Systems. *Control Systems, IEEE*, 27(6):46–99.

[119] Winner, H. und Wachenfeld, W. (2013). Absicherung Automatischen Fahrens, München. In *6. FAS Tagung*.

[120] Ziade, H., Ayoubi, R., und Velazco, R. (2004). A Survey on Fault Injection Techniques. *The International Arab Journal of Information Technology*, 1:171–186.

[121] Ziegler, S. (2010). Eingebettete Systeme – Anwendungsbeispiele, Zahlen und Trends. *BITKOM Bundesverband Informationswirtschaft, Telekommunikation und neu Medien e.V*.

Anhang A

Übersicht Modultests

A.1 Injektion via C++ Modellierungsprimitiven

Die Anwendungsbeispiele im folgenden Abschnitt sind an einem Verhaltensmodell eines parallel geladenen Registers dargestellt. Das Register wird über zwei funktionale Prozesse modelliert. Abbildung A.1 zeigt einen Auszug der strukturellen Beschreibung des Registers. Der erste Prozess modelliert die kombinatorische Schaltung zur Aktivierung des Ladevorgangs und der zweite Prozess die Speicherung des Datenwertes. Die Injektion erfolgt, in die lokale C++-Variable, die der Speicherung des Datenwerts dient. Die Variable ist ein einfacher Integer-Datentyp. Die Schaltung wird

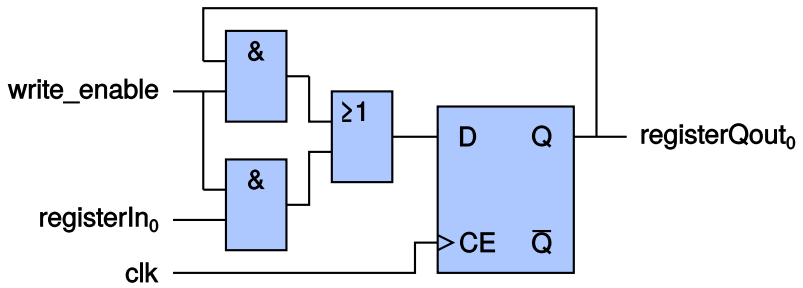


Abbildung A.1: Schaltung eines parallel geladenen Registers

in einer Testbench getestet, die alle 400 ns einen neuen Wert schreibt. Hierzu wird das `write_enable` Signal für 150 ns auf den 1-Pegel gelegt und der zu schreibende Wert angelegt. Die Werte werden zwischen jedem Schreibzugriff um 2 inkrementiert, wobei mit einem initialen Wert von 4 begonnen wird. Mit der nächsten positiven Taktflanke sollte der anliegende Wert ins Register geschrieben werden und bis zum nächsten Schreibzugriff stabil bleiben.

A.1.1 [MT1.1] Zeitbedingter, transienter Fehler

Injektion eines konstanten Wertes in eine `uint16` Variable, die der Ausgabe der kombinatorischen Schaltung und der Eingabe des D-FFs entspricht. Der konstante Wert, wird mithilfe des BTMs spezifiziert. Die Injektion erfolgt zeitbedingt nach 525 ns und repariert sich selbst nach weiteren 200 ns. Zur Fehlerbehebung wird der zuletzt geschriebene Wert wiederhergestellt. Abbildung A.2 zeigt das verwendete BTM.

```
1 <ESname> errFree <EStype> initial
2 <ESname> errState1
3 <ETsrc> errFree <ETtgt> errState1
4   <ETguard> BTMclock('OKTime').read() == sc_time(425, sc_time_unit.SC_NS)
5   <ETaction> VPvar('combiOut').force(0xFF)
6   <ETupdate> BTMclock('ERRTime').reset()
7 <ETsrc> errState1 <ETtgt> errFree
8   <ETguard> BTMclock('ERRTime').read() >= sc_time(200, sc_time_unit.SC_NS)
9   <ETaction> VPvar('combiOut').release()
```

Abbildung A.2: BTM eines zeitbedingten, transienten, konstanten Fehlers

Abbildung A.3 zeigt den beobachteten Fehlereffekt. Zu beachten ist, dass die Fehlerinjektion zum Zeitpunkt 425 ns innerhalb eines Taktpulses liegt. Der Fehler wird somit erst mit der nächsten steigenden Taktflanke in die FF gespeichert. Des Weiteren ist zu erkennen, dass der Schreibzugriff zum Zeitpunkt 450 ns, während der aktiven Injektion keine Auswirkung auf die Variable haben. Der Fehlerinjektor behält seinen fehlerhaften Wert. Beim Freigeben des Fehlerinjektors zum Zeitpunkt 625 ns wird der zuletzt geschriebene Wert sofort wiederhergestellt.

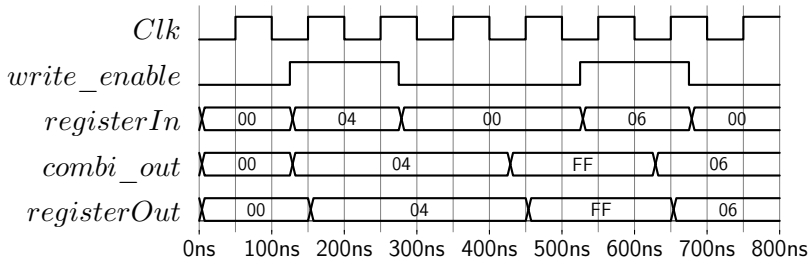


Abbildung A.3: Beobachtetes Fehlerverhalten im Testfall MT1.1

A.1.2 [MT1.2] Zeitbedingter, intermittierender Fehler

Aufbauend auf der transienten Fehlerbeschreibung aus dem Testfall MT1.1 wird in diesem Testfall ein intermittierender Fehler beschreiben. Hierzu wird lediglich auf der Rückkante der Zeitgeber `OKTime` zurückgesetzt. Dies resultiert in einen, alle 625 ns, wiederkehrenden Fehler. Abbildung A.4 zeigt das verwendete BTM. Des Weiteren zeigt das BTM die Verwendung eines lokalen Zählers, der das Auftreten des Fehlers auf zwei Fehlerinjektionen beschränkt. Neben der Verwendung einer lokalen Variablen wäre die Modellierung über einen weiteren Zeitgeber möglich. In Zeile 13 aus Abbildung A.4 ist zu erkennen, dass mehrere Anweisungen in einer Aktion verkettet sind. Es wird sowohl der interne Zähler inkrementiert als auch die eigentliche Fehlerinjektion vorgenommen.

```
1  <ESname> initState <EStype> initial
2  <ESname> errFree
3  <ESname> errState1
4  <ETsrc> initState <ETtgt> errFree
5    <ETaction> BTMvar('counter').set(0)
6  <ETsrc> errFree <ETtgt> errState1
7    <ETguard> (BTMclock('OKTime').read() == sc_time(425, sc_time_unit.SC_NS)) and
8              (BTMvar('counter').read() < 2)
9    <ETaction> VPvar('combiOut').force(0xFF)
10   <ETupdate> BTMclock('ERRTime').reset()
11 <ETsrc> errState1 <ETtgt> errFree
12   <ETguard> BTMclock('ERRTime').read() >= sc_time(200, sc_time_unit.SC_NS)
13   <ETaction> BTMvar('counter').set(BTMvar('counter').read()+1);
14              VPvar('combiOut').release()
15   <ETupdate> BTMclock('OKTime').reset()
```

Abbildung A.4: BTM eines zeitbedingten, intermittierenden Fehlers mit maximaler Injektionshäufigkeit

Das beobachtete Fehlerverhalten ist in Abbildung A.5 dargestellt. Es weist bis zum Zeitpunkt 800ns das gleiche Fehlerverhalten wie der Testfall MT1.1 auf. Durch die Spezifikation eines intermittierenden Fehlers wird die Injektion nach 1050 ns wiederholt. Die Festlegung einer oberen Schranke für die Injektionshäufigkeit führt dazu, dass zum Zeitpunkt 1625 ns keine weitere Injektion stattfindet.

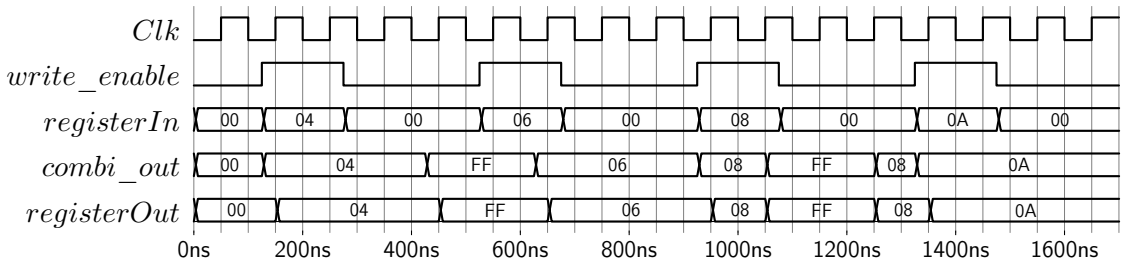


Abbildung A.5: Beobachtetes Fehlerverhalten im Testfall MT1.2

A.1.3 [MT1.3] Synchronisierte Fehler

Der Testfall MT1.3 modelliert den gleichen Fehler wie Testfall MT1.1. Hierbei wird aber das Auslösen der Injektion über ein separates BTM gesteuert. Der Zustandsautomat löst ein internes Ereignis aus, das einen Zustandswechsel im zweiten Automaten bedingt. Bei dem ausgelösten Zustandsübergang wird die Injektion vorgenommen. Abbildung A.6 zeigt den spezifizierten Fehler.

```
<ESname> errFree <EStype> initial
<ESname> errState
<ETsrc> errFree <ETtgt> errState
  <ETsync> syncVar?
  <ETupdate> BTMclock('ERRTime').reset()
  <ETaction> VPvar('combiOut').force(0xFF)
<ETsrc> errState <ETtgt> errFree
  <ETguard> BTMclock('ERRTime').read() >= sc_time(200, sc_time_unit.SC_NS)
```

```
<ESname> timeState <EStype> initial
<ETsrc> timeState <ETtgt> timeState
  <ETguard> BTMclock('OKTime').read() >= sc_time(425, sc_time_unit.SC_NS)
  <ETsync> syncVar!
  <ETupdate> BTMclock('OKTime').reset()
```

Abbildung A.6: Synchronisierte BTMs eines zeitbedingten, transienten, konstanten Fehlers

Das beobachtete Fehlerverhalten ist in Abbildung A.7 dargestellt. Es weist das gleiche Fehlerverhalten wie der Testfall MT1.1 auf. Lediglich die Spezifikation ist über zwei Zustandsautomaten verteilt.

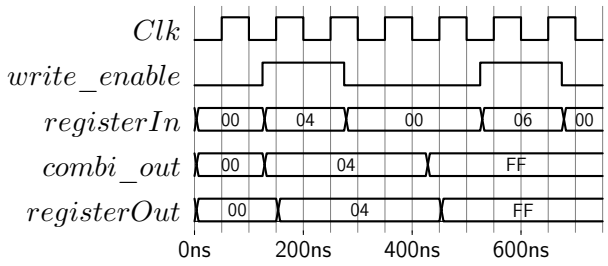


Abbildung A.7: Beobachtetes Fehlverhalten im Testfall MT1.3

A.1.4 [MT1.4] Wertebedingter, transienter Fehler

Im folgenden Testfall wird die Injektion abhängig vom aktuellen Wert des Fehlerinjektors ausgelöst. Abbildung A.8 zeigt die spezifizierte Fehlerbeschreibung. Sobald der Fehlerinjektor den Wert 0x4 aufweist, wird der Wert durch eine Konstante überschrieben.

```
<ESname> errFree <EStype> initial
<ESname> errState
<ETsrc> errFree <ETtgt> errState
  <ETguard> VPvar('combiOut').read() == 0x04
  <ETaction> VPvar('combiOut').force(0xFF)
  <ETupdate> BTMclock('ERRTime').reset()
<ETsrc> errState <ETtgt> errFree
  <ETguard> BTMclock('ERRTime').read() == sc_time(200, sc_time_unit.SC_NS)
  <ETaction> VPvar('combiOut').release()
```

Abbildung A.8: BTM eines wertebedingten, transienten, konstanten Fehlers

Abbildung A.9 zeigt das resultierende Fehlverhalten. Der Fehler ist aktiv, solange am Ausgang der kombinatorischen Schaltung der Wert 0x04 anliegt. Zum Zeitpunkt 125 ns wird von der Testbench der Wert 0x04 geschrieben, was in eine sofortige Injektion des konstanten Werts resultiert. Nach 200 ns würde die Injektion aufgehoben, da aber zu diesem Zeitpunkt immer noch, der Wert 0x04 am Ausgang anliegt, wird die Injektion erneuert. Nach 400 ns wird der Wert 0x06 geschrieben und somit die Injektion nach weiteren 200 ns aufgehoben.

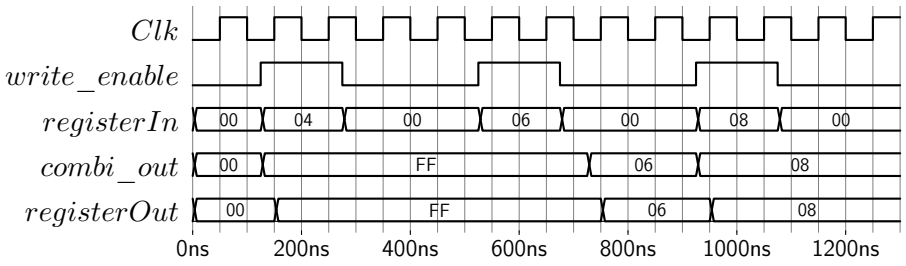


Abbildung A.9: Beobachtetes Fehlverhalten im Testfall MT1.4

A.1.5 [MT1.5] Zeitbedingter, permanenter Fehler

Abbildung A.10 zeigt das verwendete BTM, das einen permanenten Fehler spezifiziert. Der Fehler wird nach 425 ns injiziert und bleibt ab diesem Zeitpunkt bestehen. Der zu injizierende Wert wird abhängig von dem aktuellen Ausgang der kombinatorischen Schaltung bestimmt. Im spezifizierten Fall soll der Wert um eins inkrementiert werden.

```
1 <ESname> errFree <EStype> initial
2 <ESname> errState
3 <ETsrc> errFree <ETtgt> errState
4   <ETguard> BTMclock('OKTime').read() == sc_time(425, sc_time_unit.SC_NS)
5   <ETaction> VPvar('combiOut').force(VPvar('combiOut').read() + 1)
```

Abbildung A.10: BTM einer zeitbedingten, permanenten Injektion abhängig vom Systemzustand

Der resultierende Fehlereffekt ist in Abbildung A.11 dargestellt. Nach 425 ns wird der aktuell anliegende Wert um eins inkrementiert, was in der Injektion des Wertes 0x05 resultiert. Ab diesem Zeitpunkt bleibt der Wert konstant und alle weiteren Schreibzugriffe durch die Simulation werden ignoriert.

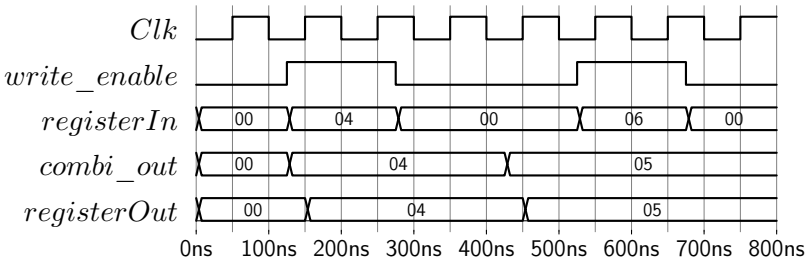


Abbildung A.11: Beobachtetes Fehlerverhalten im Testfall MT1.5

A.1.6 [MT1.6] Permanenter, adaptiver Fehler

In diesem Testfall wird ein permanenter Fehler injiziert, der abhängig vom Sollwert des Fehlerinjektors ist. Abbildung A.12 zeigt das verwendete BTM. Es ist ähnlich des BTMs aus MT1.5, wobei im Fehlerzustand eine Schleife dafür sorgt, dass immer der aktuelle Wert zur Fehlerinjektion verwendet wird. Die Kante weist keinerlei Bedingung auf, was dazu führt, dass jede Änderung sofort die Injektion beeinflusst.

```
1 <ESname> errFree <EStype> initial
2 <ESname> errState
3 <ETsrc> errFree <ETtgt> errState
4   <ETguard> BTMclock('OKTime').read() == sc_time(425, sc_time_unit.SC_NS)
5   <ETaction> VPvar('combiOut').force(VPvar('combiOut').read() + 1)
6 <ETsrc> errState <ETtgt> errState
7   <ETaction> VPvar('combiOut').force(VPvar('combiOut').read() + 1)
```

Abbildung A.12: BTM eines zeitbedingten, permanenten Fehlers, der sich mit dem Systemzustand ändert

Der beobachtete Fehlereffekt in Abbildung A.13 ist stets abhängig von dem geschriebenen Wert. Nach 425 ns wird die Fehlerinjektion aktiviert. Jeder weitere Registerschreibzugriff wird bei der Fehlerinjektion berücksichtigt und sofort um eins inkrementiert.

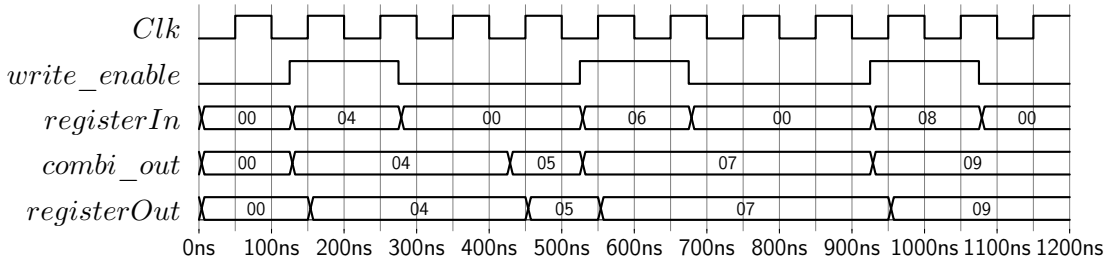


Abbildung A.13: Beobachtetes Fehlerverhalten im Testfall MT1.6

A.1.7 [MT1.7] Rücksetzverhalten: Original Wert

Im folgenden Testfall wird der Testfall A.1.1 mit einer abgeänderten Rücksetzstrategie verwendet. Abbildung A.14 zeigt die verwendete Fehlerbeschreibung.

```
<ESname> errFree <EStype> initial
<ESname> errState1
<ETsrc> errFree <ETtgt> errState1
  <ETguard> BTMclock('OKTime').read() == sc_time(425, sc_time_unit.SC_NS)
  <ETaction> VPvar('combiOut').force(0xFF)
  <ETupdate> BTMclock('ERRTime').reset()
<ETsrc> errState1 <ETtgt> errFree
  <ETguard> BTMclock('ERRTime').read() >= sc_time(200, sc_time_unit.SC_NS)
  <ETaction> VPvar('combiOut').release(clearStrategy.RESTORE_OV)"
```

Abbildung A.14: BTM eines transienten Fehlers, der den originalen Datenwert wiederherstellt

Abbildung A.15 zeigt das resultierende Fehlerverhalten. Hierbei ist zu beachten, dass im Gegensatz zur Abbildung A.14 nicht der zum Zeitpunkt des Rücksetzens normalerweise gültige Wert wiederhergestellt wird, sondern der zum Zeitpunkt der Injektion gültige Wert.

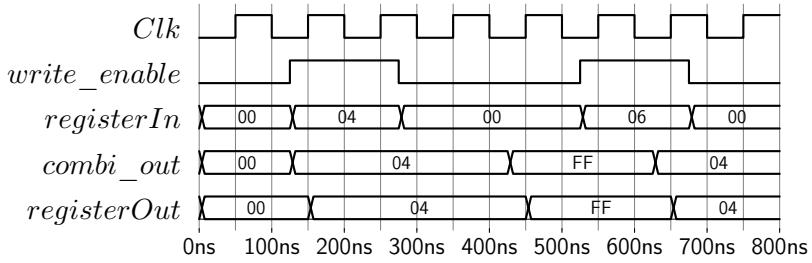


Abbildung A.15: Beobachtetes Fehlerverhalten im Testfall MT1.7

A.1.8 [MT1.8] Rücksetzverhalten: Freigabe der Injektion

Im folgenden Testfall wird der Testfall A.1.1 mit einer abgeänderten Rücksetzstrategie verwendet. Abbildung A.16 zeigt die verwendete Fehlerbeschreibung.

```
<ESname> errFree <EStype> initial
<ESname> errState1
<ETsrc> errFree <ETtgt> errState1
  <ETguard> BTMclock('OKTime').read() == sc_time(425, sc_time_unit.SC_NS)
  <ETaction> VPvar('combiOut').force(0xFF)
  <ETupdate> BTMclock('ERRTime').reset()
<ETsrc> errState1 <ETtgt> errFree
  <ETguard> BTMclock('ERRTime').read() >= sc_time(200, sc_time_unit.SC_NS)
  <ETaction> VPvar('combiOut').release(clearStrategy.RELEASE)"
```

Abbildung A.16: BTM eines transienten Fehlers mit der Freigabe der Injektion

Abbildung A.17 stellt das resultierende Fehlerverhalten dar. Hierbei ist zu beachten, dass im Gegensatz zur Abbildung A.16 nicht der zum Zeitpunkt des Rücksetzens normalerweise gültige Wert wiederhergestellt wird, sondern der zum Zeitpunkt der Injektion gültige Wert.

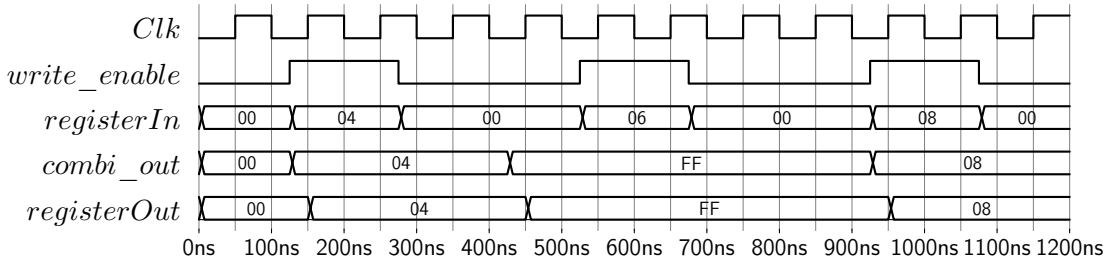


Abbildung A.17: Beobachtetes Fehlverhalten im Testfall MT1.8

A.2 Injektion via aggregierte C++-Modellierungsprimitiven

Das folgende Anwendungsbeispiel basiert auf der Registerbank des Plasma Prozessors. Ähnlich des vorherigen Beispiels wird die Funktionalität über zwei Prozesse modelliert. Der erste Prozess modelliert die Verarbeitung der Steuersignale und der zweite Prozess modelliert das Lesen und Schreiben der Registerinhalte. Der Registerinhalt wird in C++ Variable gespeichert. Der Unterschied zum vorherigen Beispiel ist, dass nicht nur eine Registerzelle beschrieben wird, sondern ein Array von Registerzellen. Abbildung A.18 zeigt das Schaltbild der Registerbank. Der untersuchte

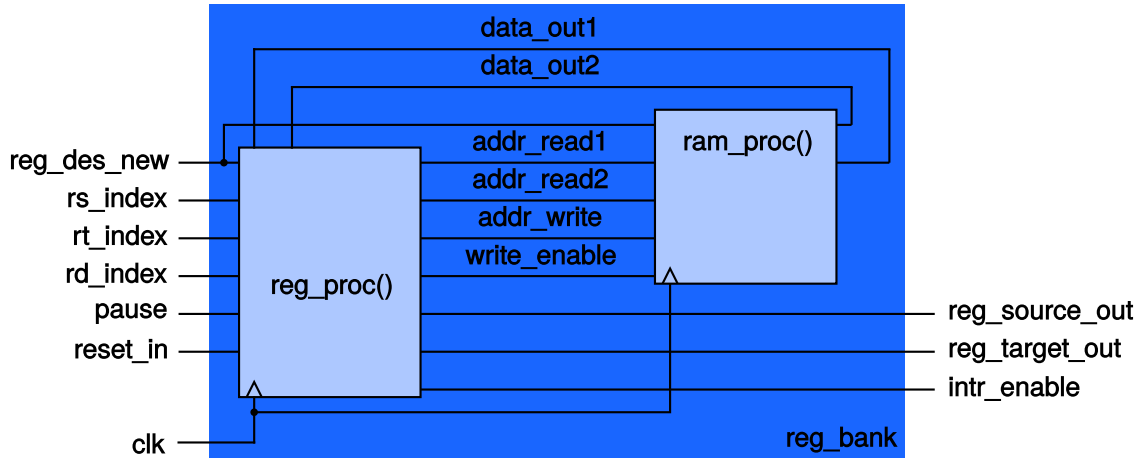


Abbildung A.18: Schaltbild der modellierten Registerbank

Testfall schreibt sequenziell alle 200 ns ein Register. Der geschriebene Wert entspricht der Register-ID im unteren und oberen Byte des Registers.

A.2.1 [MT2.1] Fehlerinjektion in eine Array-Datenstruktur

In diesem Testfall wird ein konstant definierter Fehler in eine Array-Datenstruktur injiziert. Die Fehlerspezifikation adressiert die jeweiligen Elemente des Arrays mittels des Indexoperators. Abbildung A.19 zeigt den spezifizierten Fehler.

```
<ESname> errFree <EStype> initial
<ESname> errState1
<ETsrc> errFree <ETtgt> errState1
  <ETguard> BTMclock('OKTime').read() == sc_time(725, sc_time_unit.SC_NS)
  <ETaction> VPvar('triPortRam')[2].force(0XFFFF)
  <ETupdate> BTMclock('ERRTime').reset()
<ETsrc> errState1 <ETtgt> errFree
  <ETguard> BTMclock('ERRTime').read() >= sc_time(300, sc_time_unit.SC_NS)
  <ETaction> VPvar('triPortRam')[2].release()
```

Abbildung A.19: BTM transienter Fehler auf aggregierter Datenstruktur

Abbildung A.20 zeigt das beobachtete Fehlverhalten. Hierbei sind zu erkennen, dass nach der Injektion zum Zeitpunkt 725 ns alle nachfolgenden Schreibzugriffe auf das Array blockiert sind. So würde z. B. zum Zeitpunkt 800ns der Wert $0x0303$ in das Register mit dem Index 3 geschrieben. Der Schreibzugriff hat während der aktiven Injektion aber keine Auswirkung. Zum Zeitpunkt der Injektion wird der Wert der Fehlerinjektion durch Erweiterung des aktuellen Zustands mit der Fehlerinjektion bestimmt und diese aufrechterhalten, bis die Injektion wieder freigegeben wird. Ab dem Zeitpunkt 1025 ns wird die Injektion freigegeben und alle in der Zwischenzeit geschriebenen Werte, wiederhergestellt.

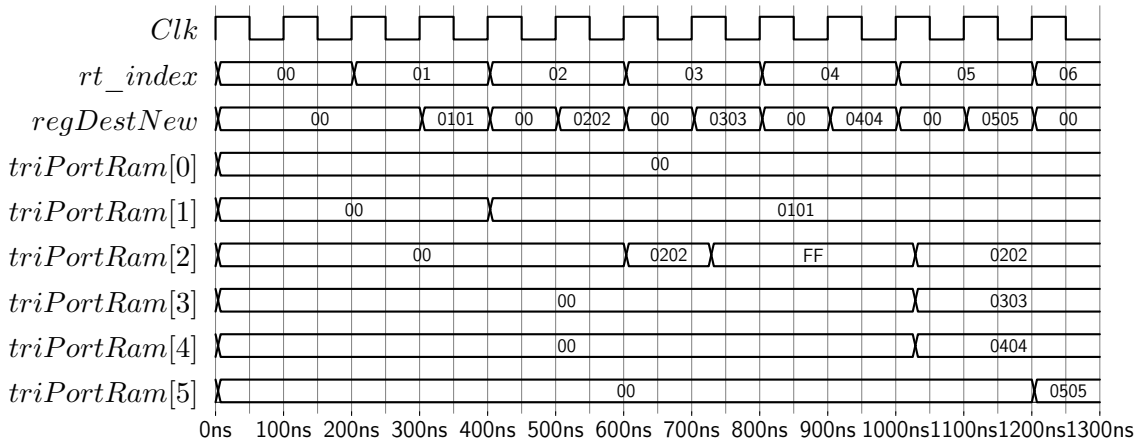


Abbildung A.20: Beobachtetes Fehlverhalten im Testfall MT2.1

A.2.2 [MT2.2] Fehlerinjektion auf einzelne Elemente eines Arrays

In diesem Testfall wird ein konstant definierter Fehler in die Elemente einer Array-Datenstruktur injiziert, wobei im Gegensatz zu A.2.1, der Fehler sich lediglich auf ein Element auswirken soll. Die zugehörige Fehlerspezifikation ist in Abbildung A.21 dargestellt. Im Gegensatz zur Fehlerspezifikation des Testfalls A.2.1 besitzt die Fehlerspezifikation eine rekursive Kante im Fehlerzustand, die den Fehler wiederholt injiziert. Vor der wiederholten Injektion wird die Injektion aufgehoben und der Fehler wieder freigegeben. Durch Verkettung der Freigabe und der Injektion innerhalb eines Python-Ausdrucks ist sichergestellt, dass beide Anweisungen in einer ununterbrochenen Abfolge ausgewertet werden. Vorteil dieses Vorgehens ist, dass anhand der Fehlerspezifikation eindeutig erkennbar ist, ob die Injektion Auswirkungen auf das gesamte Array oder einzelne Elemente hat.

```
1  <ESname> errFree <EStype> initial
2  <ESname> errState1
3  <ETsrc> errFree <ETtgt> errState1
4    <ETguard> BTMclock('OKTime').read() == sc_time(725, sc_time_unit.SC_NS)
5    <ETaction> VPvar('triPortRam')[2].force(0XFFFF)
6    <ETupdate> BTMclock('ERRTime').reset()
7  <ETsrc> errState1 <ETtgt> errFree
8    <ETguard> BTMclock('ERRTime').read() >= sc_time(300, sc_time_unit.SC_NS)
9    <ETaction> VPvar('triPortRam')[2].release()
10 <ETsrc> errState1 <ETtgt> errState1
11   <ETaction> VPvar('triPortRam')[2].release();
12              VPvar('tri_port_ram')[2].force(0XFFFF)
```

Abbildung A.21: BTM transienter Fehler auf aggregierter Datenstruktur

Abbildung A.22 zeigt das beobachtete Fehlverhalten. Es ist zu erkennen, dass sich die Fehlerinjektion lediglich auf das Register mit dem Index 2 auswirkt. Die Schreibzugriffe zum Zeitpunkt 800 ns und 1000 ns haben einen sofortigen Effekt auf das Register-Array.

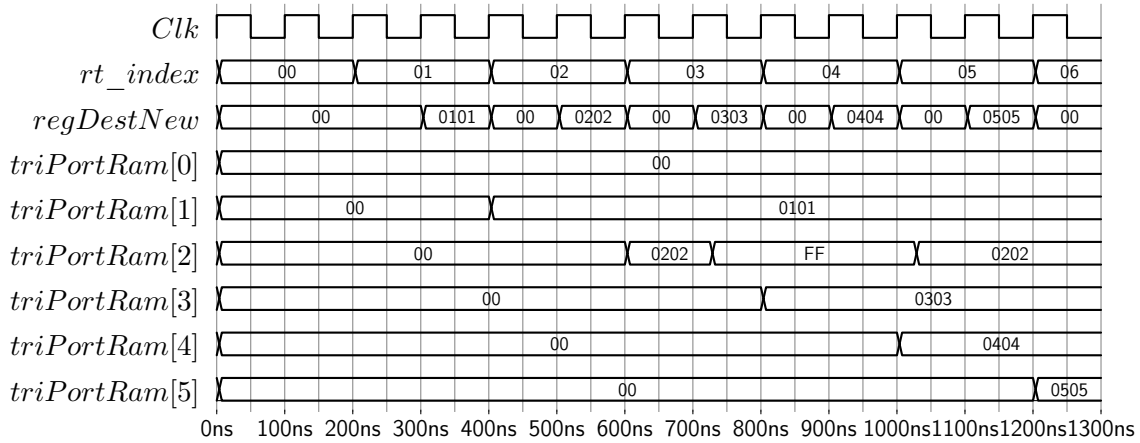


Abbildung A.22: Beobachtetes Fehlerverhalten im Testfall MT2.2

A.2.3 [MT2.3] Randomisierte Fehlerinjektion in Arrays

Der Testfall MT2.3 zeigt die Verwendung der Python-Bibliothek `random` zur Spezifikation eines gleich verteilten Fehlers auf dem Register-Array. Abbildung A.23 zeigt die verwendete Fehlerspezifikation. Hierbei wird ein zusätzlicher Initialisierungsknoten eingefügt. Während der Transition wird der Pseudo-Random Generator initialisiert. Dies bewirkt, dass die Simulationen wiederholbar sind, da die Zufallszahlengeneratoren immer die gleichen Zufallszahlen erzeugen. Über die Änderung des Seeds ist es möglich, unterschiedliche Abfolgen von Zufallszahlen zu generieren. Im nächsten Schritt werden zwei Arrays erzeugt, die der späteren Injektion dienen. In der Liste 'index' sind alle Indizes gespeichert, welche die Fehlerinjektion manipuliert. In der Liste 'array' sind alle Werteänderungen an den jeweiligen Positionen gespeichert. Die BTM-Infrastruktur speichert die beide Datenstrukturen lokal und sie stehen hierdurch bei weiteren Fehlerinjektionen zur Verfügung. Dies ist notwendig, da durch die Randomisierung nicht sichergestellt ist, dass die Zufallszahlengeneratoren bei jedem Folgeaufruf die gleichen Zufallszahlen generieren. Das weitere BTM entspricht größtenteils dem Testfall in Abschnitt A.2.2. Die Injektion wird über eine rekursive Kante wiederholt, damit berücksichtigt die Fehlerinjektion aktuelle Schreibvorgänge durch die Simulation. Hierbei wird der zuvor generierte und abgespeicherte Fehlervektor verwendet, um eine Neugenerierung der Zufallszahlen zu verhindern.

```
1  <ESname> errFree <EStype> initial
2  <ESname> errInitD
3  <ESname> errState
4  <ETsrc> errFree <ETtgt> errInitD
5    <ETaction> seed(123456);
6               array = [randint(0,1) for item in range(0,32)];
7               index = [item for item in range(0,32) if array[item]!=0];
8               array = [item * randint(0,0xFFFF) for item in array];
9               BTMvar('injValue').set(array);
10              BTMvar('injIndex').set(index)
11 <ETsrc> errInitD <ETtgt> errState1
12   <ETguard> BTMclock('OKTime').read() == sc_time(725, sc_time_unit.SC_NS)
13   <ETaction> [VPvar('tri_port_ram')[id].force(BTMvar('injValue').read()[id])
14                 for id in BTMvar('injIndex').read()]
15   <ETupdate> BTMclock('ERRTime').reset()
16 <ETsrc> errState1 <ETtgt> errInitD
17   <ETguard> BTMclock('ERRTime').read() >= sc_time(300, sc_time_unit.SC_NS)
18   <ETaction> VPvar('tri_port_ram')[2].release()
19 <ETsrc> errState1 <ETtgt> errState1
20   <ETaction> VPvar('tri_port_ram')[2].release();
21              [VPvar('tri_port_ram')[id].force(BTMvar('injValue').read()[id])
22                for id in BTMvar('injIndex').read()]
```

Abbildung A.23: BTM randomisierter Fehler in aggregierter Datenstruktur

Abbildung A.24 zeigt das beobachtete Fehlverhalten. Zu erkennen ist, dass die Fehlerinjektion zum Zeitpunkt 725 ns mehrere Register auf einen zufälligen Wert ändert. Durch Verändern des Seeds würden sich die Register sowie die Werte der Injektion ändern. Der Fehlerinjektor behebt zum Zeitpunkt 1025 ns die injizierten Fehler und stellt die originalen Werte wieder her. Nur durch Zwischenspeichern der

Injektion lässt sich der Fall eines konstanten Fehlers in einem Teil der Array-Elemente realisieren.

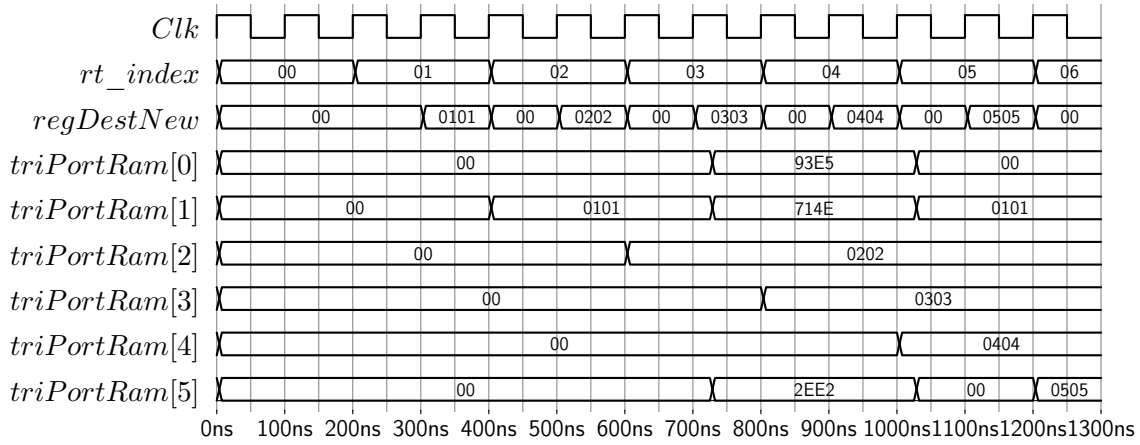
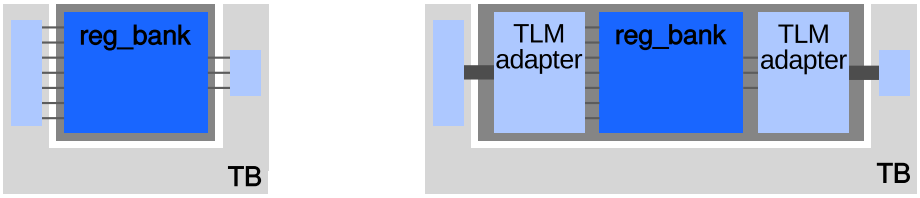


Abbildung A.24: Beobachtetes Fehlerverhalten im Testfall MT2.3

A.3 Injektion via SystemC Modellierungsprimitiven

Im Folgenden wird das Anwendungsbeispiel, der Registerbank, aus Abschnitt A.2 aufgegriffen. Statt der Injektion in die C++-Variablen, die den eigentlichen Speicherinhalt modellieren, injiziert dieser Fall die Fehler in die Steuerleitungen. Abbildung A.25 zeigt die beiden Anwendungsfälle, die in diesem Abschnitt untersucht werden. Die



(a) Direktes Setzen der Signale (b) Setzen über TLM-Adapter

Abbildung A.25: Testaufbau für die Registerbank

folgenden Abschnitte untersuchen zwei ähnliche Testfälle. In Abschnitt A.3.1 wird der Fehler direkt in die Steuersignale zwischen Testbench und DUT injiziert. In Abschnitt A.3.2 wird die Signalschnittstelle durch einen TLM-Adapter gekapselt und die Fehlerinjektion erfolgt in die Transaktionsobjekte.

A.3.1 [MT3.1] Injektion in SystemC Signale

Der folgende Testfall verwendet einen einfachen zeitbedingten transienten Fehler. Nach 475 ns wird das Rücksetzsignal auf High-Potenzial gezogen, was dazu führt, dass sich die Einträge in der Registerbank zurücksetzen. Abbildung A.27 zeigt das

```
1 <ESname> errFree <EStype> initial
2 <ESname> errState1
3 <ETsrc> errFree <ETtgt> errState1
4   <ETguard> BTMclock('OKTime').read() == sc_time(475, sc_time_unit.SC_NS)
5   <ETaction> VPvar('resetSig').force(True) <ETupdate> BTMclock('ERRTime').reset()
6 <ETsrc> errState1 <ETtgt> errFree
7   <ETguard> BTMclock('ERRTime').read() >= sc_time(100, sc_time_unit.SC_NS)
8   <ETaction> VPvar('resetSig').release()
```

Abbildung A.26: BTM zeitbedingter Fehler in ein `sc_signal<bool>`

beobachtete Fehlverhalten. Die Registerbank implementiert einen high-aktiven, asynchronen Rest, sobald der Rest-Eingang einen High-Pegel erreicht, wird der komplette Registerinhalt zurückgesetzt. Die Fehlerinjektion erfolgt in den Datentyp `sc_signal`, der bei Wertänderung automatisch ein Ereignis im SystemC-Kernel auslöst und hierdurch alle sensitiven Prozesse getriggert werden. Aufgrund dieser Tatsache ist es nicht notwendig, eine eigene Auslösebedingung anzugeben. Sobald der Fehler injiziert ist, werden alle sensitiven Prozesse ausgelöst und der Fehler kann sich im Simulationsmodell propagieren. Im Anwendungsfall kann dies anhand des sofortigen Rücksetzens der Register zum Zeitpunkt 475 ns beobachtet werden.

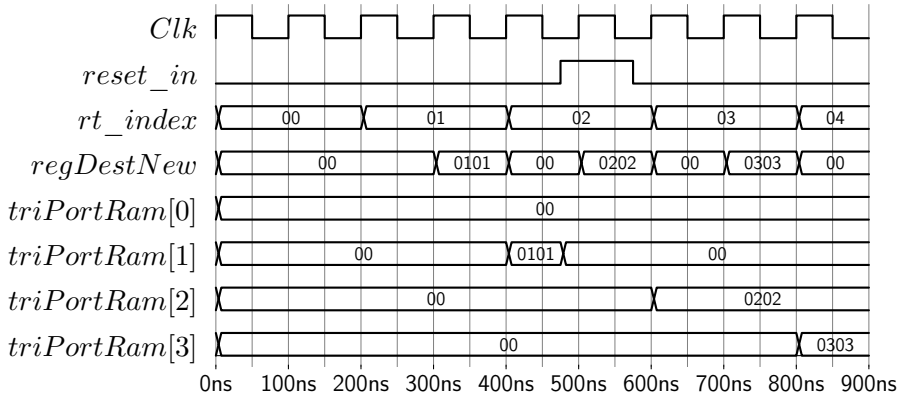


Abbildung A.27: Beobachtetes Fehlverhalten im Testfall MT3.1

A.3.2 [MT3.2] Injektion in Transaktionsobjekte

Im Gegensatz zum vorherigen Testfall, bei dem die SystemC-Primitive `sc_signal` das Ziel der Injektion war, wird in diesem Testfall in ein TLM-Transaktionsobjekt ein Fehler injiziert. Hierzu wird das gleiche DUT verwendet, lediglich werden TLM-Adapter verwendet- um Transaktionen in Signalansteuerungen zu übersetzen. Der

```
1  <ESname> errFree <EStype> initial
2  <ESname> errState1 \
3  <ETsrc> errFree <ETtgt> errState1
4    <ETguard> VPvar('tlmpayloadProbe').read().get_address() == 4
5    <ETaction> injVal = VPvar('tlmpayloadProbe').read();
6               injVal.set_data([0x99, 0x99, 0, 0]);
7               VPvar('tlmpayloadProbe').force(injVal)
8  <ETsrc> errState1 <ETtgt> errFree
9    <ETguard> VPvar('tlmpayloadProbe').read().get_address() != 4
10   <ETaction> VPvar('tlmpayloadProbe').release()
```

Abbildung A.28: BTM datenabhängiger Nutzdatenfehler eines Transaktionsobjekts

in Abbildung A.29 spezifizierte Fehler, injiziert fehlerhafte Nutzdaten, immer wenn das Transaktionsobjekt auf das Register mit der ID 04 zugreifen möchte. Bei der Injektion wird zuerst das Transaktionsobjekt der Simulation gelesen und anschließend verändert. Dies reduziert den Aufwand, da nicht das komplette Transaktionsobjekt durch das BTM spezifiziert werden muss. Das vorgestellte BTM verändert z. B. sowohl Lese- als auch Schreibzugriffe, da das zu injizierende Transaktionsobjekt zuvor gelesen wird. Abbildung A.29 zeigt das beobachtete Fehlverhalten. Es ist zu erkennen, dass bei der Beschreibung des Registers 04 ein falscher Wert geschrieben wird. Im Gegensatz zu den Testfällen in Abschnitt A.2 ist zu erkennen, dass nicht der interne Zustand verändert wird, sondern die Kontrollsignale. So wird zum Zeitpunkt 500 ns das Signal `reg_dest_new` überschrieben.

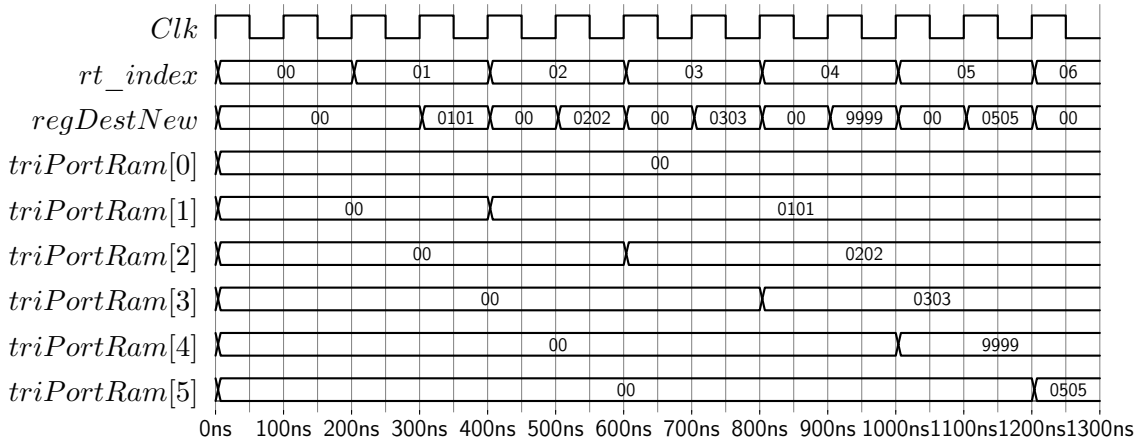


Abbildung A.29: Beobachtetes Fehlerverhalten im Testfall MT3.2

Die Bedeutsamkeit von eingebetteten Systemen steigt kontinuierlich. Neben der reinen Anzahl steigt auch die Komplexität der einzelnen Systeme. Dies resultiert nicht nur in einem steigenden Entwurfsaufwand, sondern betrifft auch den Analyseaufwand. Hierbei ist zu beachten, dass die Systeme vermehrt sicherheitsrelevante Aufgaben übernehmen. Ein anschauliches Beispiel stellen Systeme zur Fahrerassistenz bzw. Fahrzeugautomatisierung dar. Für solche Systeme bedeutet ein Ausfall bzw. falsch erbrachter Dienst schwerwiegende Folgen. Eine Sicherheitsbewertung ist zwingend vorgeschrieben. Die hohe Vernetzung der einzelnen Systeme bedingt, dass eine isolierte Betrachtung nicht mehr ausreichend ist. Deshalb muss die Analyse neben der gestiegenen Komplexität der einzelnen Systeme zusätzlich die Interaktionen der Systeme beachten. Aktuelle Standards empfehlen zur Sicherheitsbewertung häufig Verfahren wie Brainstorming, Fehlermöglichkeits- und Fehlereinflussanalysen oder Fehlerbaumanalysen. Der Erfolg dieser Verfahren ist meist sehr stark von den beteiligten Personen geprägt und fordert ein umfassendes Systemwissen.

Diese Arbeit stellt einen Ansatz zur Unterstützung der Sicherheitsbewertung vor. Ziel ist, das benötigte Systemwissen von den beteiligten Personen, auf ein Simulationsmodell zu übertragen. Der Anwender ermittelt anhand des Simulationsmodells die systemweiten Fehlereffekte. Die Analyse der Fehlerpropagierung bildet die Grundlage der traditionellen Sicherheitsanalysen. Da das Simulationsmodell die Systemkomplexität und die Systemabhängigkeiten beinhaltet, reduzieren sich die Anforderungen an die beteiligten Personen und folglich der Analyseaufwand. Um solch ein Vorgehen zu ermöglichen, wird eine Methode zur Fehlerinjektion in Simulationsmodelle vorgestellt. Hierbei ist vor allem die Unterstützung unterschiedlicher Abstraktionsgrade, insbesondere von sehr abstrakten System-Level-Modellen, wichtig. Des Weiteren wird ein Ansatz zur umfassenden Fehlerspezifikation vorgestellt. Der Ansatz ermöglicht die Spezifikation von Fehlerursachen auf unterschiedlichen Abstraktionsebenen sowie die automatisierte Einbringung der Fehler in die Simulation. Neben der Einbringung der Fehler bildet die Beobachtung und Analyse der Fehlereffekte weitere wichtige Aspekte. Eine modellbasierte Spezifikation rundet den Ansatz ab und vereinfacht die Integration in einen modellgetriebenen Entwurf.