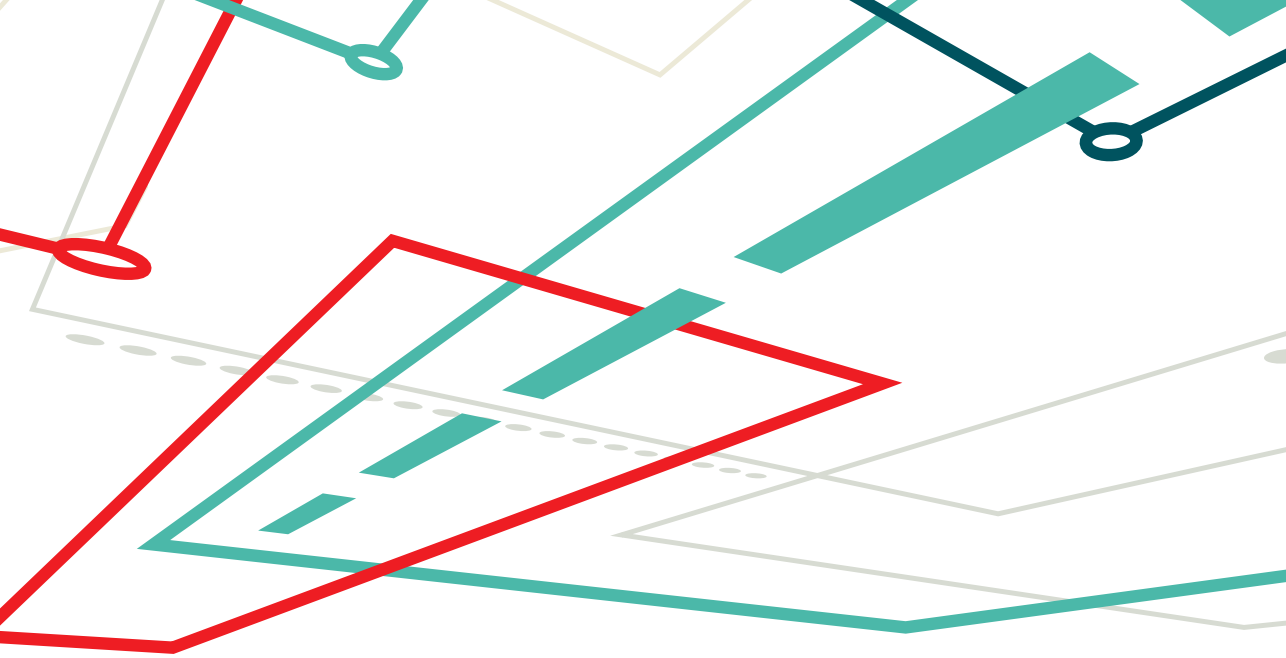
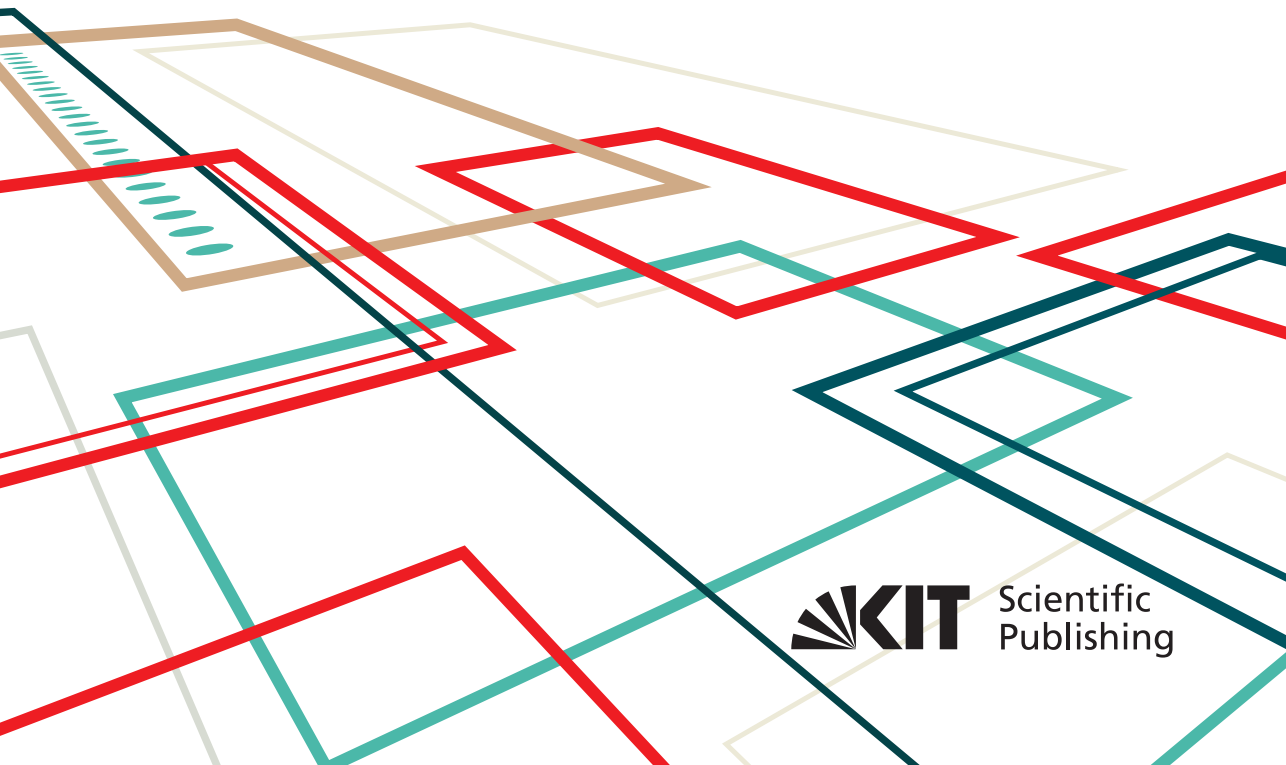


EINE TECHNOLOGIE FÜR DAS DURCHGÄNGIGE UND AUTOMATISIERTE TESTEN EINGEBETTETER SOFTWARE

TILL FISCHER



KIT Scientific Publishing

Till Fischer

Eine Technologie für das durchgängige und automatisierte Testen eingebetteter Software

Eine Technologie für das durchgängige und automatisierte Testen eingebetteter Software

von
Till Fischer

KIT Scientific Publishing

Dissertation, Karlsruher Institut für Technologie
KIT-Fakultät für Elektrotechnik und Informationstechnik

Tag der mündlichen Prüfung: 15. Dezember 2016
Referenten: Prof. Dr.-Ing. Klaus D. Müller-Glaser
Prof. Dr. Ralf H. Reussner

Impressum

Scientific Publishing

Karlsruher Institut für Technologie (KIT)
KIT Scientific Publishing
Straße am Forum 2
D-76131 Karlsruhe
KIT Scientific Publishing is a registered trademark
of Karlsruhe Institute of Technology.
Reprint using the book cover is not allowed.
www.ksp.kit.edu

Print on Demand 2017 – Gedruckt auf FSC-zertifiziertem Papier

ISBN 978-3-7315-0663-8
DOI 10.5445/KSP/1000069280

KURZFASSUNG

Kritische Eingebettete Software unterliegt hohen Sicherheitsanforderungen, die einen ausgiebigen Test der Software erfordern. Über den gesamten Testprozess hinweg werden dabei verschiedene Technologien eingesetzt. In dieser Arbeit wird diesbezüglich zwischen „Quelltextebene“ und „Systemebene“ unterschieden. Auf der Systemebene steht häufig der Test von Regelungs- und Steuerungsfunktionen im Fokus. Das macht eine Simulation der Umgebung des Systems erforderlich. Werden die Tests auf der realen Hardware (Target) ausgeführt, muss die Simulation in Echtzeit erfolgen. Dazu werden Hardware-in-the-Loop (HiL) Systeme eingesetzt. Auf der Quelltextebene steht demgegenüber der gezielte Test einzelner Softwarekomponenten im Fokus, wobei Zugriff auf interne Softwaredetails erforderlich ist. In einigen Fällen werden dazu Unit-Test Frameworks verwendet, was auf dem Target jedoch nicht immer möglich ist. Das kann mit Hilfe sogenannter Debugger kompensiert werden, speziellen Hardwarelösungen, die vornehmlich für das Finden von Fehlerursachen bei der Softwareentwicklung eingesetzt werden. Voraussetzung für die Verwendung von Debuggern zu Testzwecken ist, dass Zugriff auf geeignete Debug- und Trace-Schnittstellen der Prozessorhardware besteht.
Debugger und HiL Testsysteme ergänzen sich gegenseitig. Mit Hilfe von Debuggern können auch im Rahmen von HiL Tests relevante Details ausgelesen werden, ohne die Software zu instrumentieren. Demgegenüber ist die in HiL Testsystemen vorhandene Hardware zur Signalkonditionierung und für den Netzwerkzugriff auch im Rahmen Unit-Tests zur Erzeugung von Stimuli und zur Testauswertung nützlich. Derzeit gibt es jedoch keine wiederverwendbare Lösung zur Testautomatisierung, welche die Vorteile beider Technologien miteinander kombiniert und benutzerfreundlich zur Verfügung stellt.
Diese Arbeit hat eine solche Lösung zum Ziel. Dadurch soll die Durchgängigkeit des Testprozesses verbessert, und eine Effizienzsteigerung bei der Implementierung vollautomatisch ausführbarer Testfälle erzielt werden. Die Lösung umfasst ein Framework, das einen standardisierten Zugriff auf das Testobjekt bereitstellt, sowie eine Programmiersprache zur Implementierung von Testfällen, welche mit dem Framework verknüpft ist. Die neue Sprache kann als syntaktische Erweiterung zu einer bestehenden objektorientierten

Programmiersprache aufgefasst werden, in welche der Quelltext der neuen Sprache übersetzt wird. Die Umsetzbarkeit der Idee wird durch eine Implementierung der Testlösung gezeigt. Diese wird anhand von zwei Fallstudien evaluiert.

DANKSAGUNG

Diese Arbeit entstand in meiner Zeit als Wissenschaftlicher Mitarbeiter am FZI Forschungszentrum Informatik.
An erster Stelle möchte ich mich bei Herrn Prof. Müller-Glaser für die Möglichkeit zur Promotion und die langjährige gute Betreuung auch in der Zeit nach seiner Emeritierung bedanken. Ihre Ratschläge waren mir immer sehr wertvoll, und in unseren Gesprächen habe ich sehr viel gelernt. Dieser Dank geht auch an Herrn Prof. Sax, seinem Nachfolger als Leiter des Instituts für Technik der Informationsverarbeitung am Karlsruher Institut für Technologie.
Großen Dank an Herrn Professor Reussner, Leiter des Lehrstuhls für Software-Design und -Qualität am KIT, der das Korreferat meiner Arbeit übernommen hat.
Des Weiteren bedanke ich mich bei unserem Bereichsleiter Herrn Dr. Hillenbrand für die ständige Unterstützung während meiner Arbeit am FZI und für die Schaffung der notwendigen Rahmenbedingungen zur Anfertigung der Dissertation. Aus den gleichen Gründen geht mein Dank an Herrn Prof. Philipp Graf, der unsere Abteilung zu dem Zeitpunkt geleitet hat, als ich mit meiner Arbeit begann. Sie beide haben mir durch viele spannende Projekte die Möglichkeit gegeben, das Thema meiner Dissertation zu entdecken und zu vertiefen.
Für die vielen fröhlichen Stunden während und neben der Arbeit, sowie für den inhaltlichen Austausch danke ich meinen Kollegen. Besonderer Dank gilt Stefan Otten, der als Abteilungsleiter große Verantwortung übernommen hat.
Meiner Familie danke für das jederzeit offene Ohr und für die moralische Unterstützung, wenn etwas nicht ganz so lief wie erhofft. Ganz besonderes danke ich meiner Freundin, die mich immer wieder an die wirklich wichtigen Dinge im Leben erinnert.

INHALTSVERZEICHNIS

Teil 1

EINFÜHRUNG

Im ersten Teil wird das Umfeld und Ziel dieser Arbeit beschrieben. Der Leser wird in die nötigen Grundlagen zum Problemverständnis eingeführt und erhält einen Überblick über den aktuellen Stand der Technik.

1 EINLEITUNG

Unsere Welt ist von einer großen Zahl weitgehend unsichtbarer elektronischer Geräte durchdrungen, die als „Eingebettete Systeme“ bezeichnet werden. Über Sensoren registrieren sie Geschehnisse in der realen Welt und beeinflussen sie mit Hilfe von Aktoren. Beliebte Beispiele sind die Steuergeräte in Kraftfahrzeugen und Flugzeugen, medizinische Geräte sowie Systeme zur Überwachung und Steuerung von Industrieanlagen.
In den genannten Bereichen bestehen durchaus große Unterschiede bei der Entwicklung solcher Systeme. Eine Gemeinsamkeit ist jedoch, dass die Software der Geräte eine immer größere Rolle spielt. Sie ist zur Bewältigung der zunehmend komplizierten Probleme und für die Koordination in komplexen Systemverbänden unerlässlich. Gleichzeitig ist sie eine Fehlerquelle und kann für schwere Schäden verantwortlich sein.[1]
Diese Arbeit bezieht sich auf die Software eingebetteter Systeme im Allgemeinen, d.h. sie ist nicht spezifisch für einen bestimmten Bereich oder eine Branche. Das ist möglich, weil die technischen Randbedingungen in den relevanten Punkten vergleichbar sind (beispielsweise wird in allen Fällen die Programmiersprache C zur Softwareentwicklung eingesetzt). Trotzdem sind die folgenden Beispiele vorwiegend dem Umfeld Automobilindustrie entnommen, weil sich die Zusammenhänge anhand einer konkreten Branche besser darstellen lassen.

1.1 KRITISCHE EINGEBETTETE SOFTWARE

In dieser Arbeit geht es um das automatisierte Testen kritischer eingebetteter Software. Die hier gültige Definition von „kritisch“ wurde zu Beginn bereits gestreift: wesentlich ist die Tatsache, dass das

1 Am 7. Mai 2016 kommt es zu einem Unfall mit einem Tesla Model S, weil die Software einen kreuzenden LKW nicht korrekt erkennt. Die Person im Fahrzeug kommt dabei ums Leben [Onl/Tesla]. Am 4. Juni 1996 explodiert die Trägerrakete Ariane 5 bei ihrem ersten Start. Die technische Ursache war ein Überlauf infolgedessen die weiterhin korrekt berechneten Lagedaten nicht weitergeleitet wurden [Dow97]. Im Zeitraum von 1985 bis 1987 verursachen Softwarefehler im Strahlentherapiegerät Therac-25 den Tod von drei Patienten, drei weitere werden schwer verletzt [Lev93].

Risiko hoch ist, durch einen Fehler in der Software großen Schaden zu verursachen. Große Schäden sind die Verletzung oder der Tod von Menschen, Zerstörung der Umwelt, Fehlschlag einer Mission und ein großer finanzieller Verlust (wobei Letztgenanntes aufgrund von Haftungsfällen oder Imageschäden fast immer eine Begleiterscheinung ist).

Hohe Sicherheitsanforderungen

Für kritische Software bestehen *hohe Sicherheitsanforderungen*. Durch internationale Standards wie die ISO 26262 („Road vehicles – Functional safety“) [Std/ISO11a] werden diese Anforderungen genauer spezifiziert, und es werden geeignete Maßnahmen empfohlen, um die Anforderungen zu erfüllen.[2]

In der ISO 26262 wird der Softwaretest als wichtige Maßnahme zur Erfüllung der Anforderungen genannt. Es wird dort zwischen „Software unit testing“, „Software integration and testing“ und „Verification of software safety requirements“ unterschieden. Auf diese Unterscheidung wird gleich noch genauer eingegangen. An dieser Stelle sind zunächst jedoch zwei andere Erkenntnisse wichtig: Erstens sollte die Software auf der realen Gerätehardware (Zielplattform) getestet werden, da identisches Verhalten auf einer anderen Plattform, z.B. einem Simulator, kaum nachweisbar ist, und entsprechende Tests deswegen weniger vertrauenswürdig sind. Zweitens muss die Testdurchführung und -auswertung soweit wie möglich automatisch erfolgen, damit die Testergebnisse reproduzierbar sind [Sax08]. Der zweite Punkt wird als „Testautomatisierung“ bezeichnet, und stellt ein wesentliches Thema dieser Arbeit dar. Testautomatisierung wird außerdem durch Kosteneinsparungen im Zuge des reduzierten Personalbedarfs motiviert.

Die zweite wichtige Eigenschaft der in dieser Arbeit betrachteten Software ist durch den Begriff „eingebettet“ beschrieben. Im Rahmen dieser Arbeit ist der Begriff durch eine Wechselwirkung der Software mit der realen, physischen Welt definiert. Daraus ergeben sich technische Rahmenbedingungen, die zur Testautomatisierung *spezielle Testwerkzeuge* erforderlich machen.

Spezielle Testwerkzeuge

2 Die ISO 26262 bezieht sich auf Kraftfahrzeuge. Für Systeme in der Luftfahrt existiert die DO-178C („Software Considerations in Airborne Systems and Equipment Certification“) [Std/RTC12], bezüglich der Software medizinischer Geräte die IEC 62304 („Medical device software – Software lifecycle processes“) [Std/IEC06] und im Eisenbahnwesen die EN 50128 („Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme – Software für Eisenbahnsteuerungs- und Überwachungssysteme“) [Std/VDE12]. Weitere Bereiche werden durch die allgemeiner gefasste Norm IEC 61508 („Functional safety of electrical/electronic/programmable electronic safety-related system“) [Std/IEC10] abgedeckt.

So ist für den reproduzierbaren Tests von Regel- und Steuerfunktionen die Simulation der Umgebung des Systems erforderlich. Beim Test auf der Zielplattform ist das mit harten Echtzeitanforderungen verbunden, und es werden sogenannte Hardware-in-the-Loop Systeme (HiL Systeme) verwendet (mehr dazu in Abschnitt 3.3). Für den Test von Betriebssystemkomponenten, Treibern und Standardsoftware sind andere Werkzeuge erforderlich, welche eine Untersuchung von Softwaredetails auf Quelltextebene ermöglichen (mehr dazu in Abschnitt 3.4). Daraus ergibt sich das zentrale Problem, mit dem sich diese Arbeit beschäftigt, und im nächsten Abschnitt genauer vorgestellt wird.

1.2 PROBLEMSTELLUNG

Die ISO 26262 unterscheidet bezüglich dem Softwaretest zwischen den Phasen „Software unit testing“, „Software integration and testing“ und „Verification of software safety requirements“. Abbildung 1 zeigt das entsprechende Referenzmodell zur Softwareentwicklung nach ISO 26262.

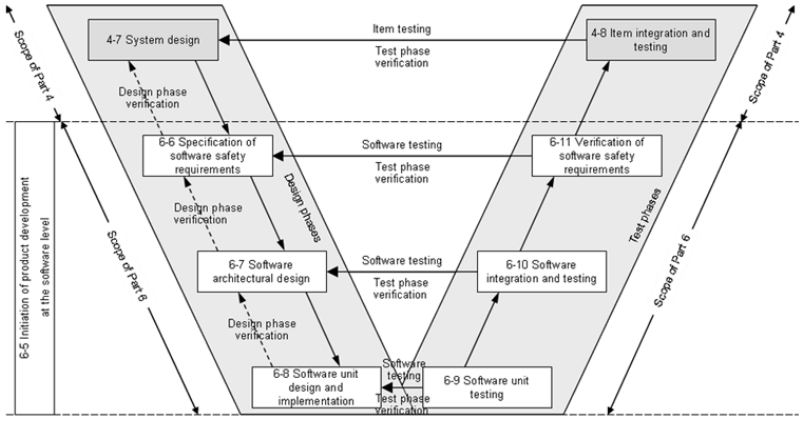


Abbildung 1: Referenzmodell zur Softwareentwicklung nach ISO 26262 [Std/ISO11a]

Die einzelnen Phasen unterscheiden sich hinsichtlich Testfällen, Fokus und Zielsetzung, und gehen, wie am Ende des letzten Abschnitts schon festgestellt wurde, mit der Verwendung spezieller Testwerkzeuge einher. Diese lassen sich sehr scharf in zwei Kategorien einteilen, die in dieser Arbeit mit „Systemebene“ und „Quelltextebene“ bezeichnet werden. Der Systemebene sind die Werkzeuge für den Test der Anwendungsfunktion zugeordnet, welche das

eingebettete System als „Black Box“ betrachten. Hierzu zählen insbesondere die bereits erwähnten HiL Systeme. Der Quelltextebene sind die Werkzeuge für den Test logischer Funktionalität zugeordnet, welche Zugriff auf alle Softwaredetails bieten.
Die Werkzeuge der zweiten Kategorie werden eher für den Test unterer Softwareschichten eingesetzt, weil sich die Anwendungsfunktion eingebetteter Systeme meistens besser über die externen Schnittstellen des Geräts verifizieren lässt, während sich der Zugriff auf Softwaredetails den Werkzeugen der ersten Kategorie entzieht. Der Übergang von der Phase „Software unit testing“ zu „Verification of software safety requirements“ geht also mit einer Umstellung auf andere Testwerkzeuge einher.
Eine solche Umstellung ist nicht vermeidbar, da sich die technischen Anforderungen für den Zugriff auf das Testobjekt grundlegend ändern (Quelltext → Externe Schnittstellen). Theoretisch ist es jedoch möglich, diese Tatsache vor dem Anwender durch geeignete Abstraktion zu verbergen. Das ist vor Allem in Bezug auf die Testautomatisierung interessant. In der Praxis existiert jedoch noch keine geeignete Technologie, welche tatsächlich alle genannten Testphasen abdeckt. Aktuell wird Testautomatisierung in den einzelnen Phasen auf unterschiedliche Weise betrieben. Der Gesamtprozess ist also nicht durchgängig.
Die größte Herausforderung bei der Lösung dieser Problemstellung ist das Finden eines Beschreibungsmittels für vollautomatisch ausführbare Testfälle. Es muss flexibel genug sein, um sowohl auf der Quelltextebene als auch der Systemebene eingesetzt zu werden, und gleichzeitig mindestens genauso gut für jede Ebene im Einzelnen geeignet sein, wie es die aktuell dort bestehenden Beschreibungsmittel zur Testautomatisierung bereits sind.

1.3 ZIEL DER ARBEIT

Das primäre Ziel dieser Arbeit ist die Effizienzsteigerung des Testens eingebetteter Software. Als Ausgangspunkt dient dabei die Annahme, dass die Automatisierung des Testbetriebs bereits weit fortgeschritten ist, und eine weitere Reduzierung der manuellen Aufwände zur Durchführung und Auswertung von Labortests nur noch in wenigen Fällen möglich ist.[3] Vor diesem Hintergrund be-

3 Das soll nicht heißen, dass es keine großen manuellen Aufwände mehr gibt, sondern dass sich diese mit den aktuell denkbaren Möglichkeiten nur schwer oder gar nicht automatisieren lassen.

steht ein größeres Potential in der Reduzierung des Implementierungsaufwands von Testfällen.
Dieses Ziel soll mit Hilfe verbesserter Werkzeugunterstützung erreicht werden. Konkret wird eine universelle Lösung zur Testautomatisierung für eingebettete Software angestrebt. Im Kern handelt es sich dabei um eine neue Programmiersprache zur Implementierung von Testfällen, welche testspezifische und allgemeine Sprachkonstrukte geschickt miteinander kombiniert. Damit soll die Implementierung intuitiver erfolgen können, und es sollen besser wartbare Resultate erzielt werden, als es mit Standardprogrammiersprachen der Fall ist, ohne demgegenüber an Mächtigkeit einzubüßen.
Untergeordnete Ziele, welche mit der Einführung der neuen Technologie ebenfalls verfolgt werden, sind:

- *Frühzeitiger Test der Software*: Die Notwendigkeit frühzeitiger Tests in der Automobilindustrie wird in [SA08] sehr gut beschrieben. Diesbezüglich dürfen die Methoden zur Verifikation und Validierung der Anwendungsfunktion *vor* Verfügbarkeit der Zielhardware nicht außer Acht gelassen werden. Testautomatisierung ist in dieser Erprobungsphase weniger weit verbreitet, aber zunehmend.

- *Wiederverwendbarkeit von Testimplementierungen*: Bereits existierende Testimplementierungen zu Softwarefragmenten, die auf einer anderen Plattform erneut verwendet werden, sollen ohne Änderung übernommen und auf der neuen Plattform ausgeführt werden werden können.

- *Strikte Trennung von Testimplementierung und Softwareimplementierung*: Für Unit-Tests wird der Test-Code derzeit meistens zusammen mit dem Software-Code entwickelt und übersetzt. Ein solches Vorgehen soll vermieden werden, um eine strikte Trennung der Rollen „Softwareentwickler“ und „Testentwickler“ zu unterstützen, denn das gewährleistet höhere Testqualität.

2 GRUNDLAGEN

Für das Verständnis der Arbeit ist Wissen aus verschiedenen Bereichen erforderlich, das nicht von jedem Leser erwartet werden kann. Zur besseren Nachvollziehbarkeit der grundlegenden Problematik ist zunächst Abschnitt 2.1 gedacht. Hier wird das Umfeld von eingebetteter Software näher beleuchtet. Insbesondere werden die heute gelebten Entwicklungsprozesse vorgestellt, welche bezüglich dem Test von Software interessant sind.
Anschließend werden in Abschnitt 2.2 einige Begrifflichkeiten aus dem Umfeld „Softwaretest“ genauer definiert, um diesbezüglich keine Missverständnisse aufkommen zu lassen.
Abschnitt 2.3 und Abschnitt 2.4 sind schließlich zum besseren Verständnis von Teil III gedacht, in welchem die Umsetzung der Idee dieser Arbeit beschrieben ist. Es werden die wichtigsten Grundlagen bezüglich formaler Sprachen und der modellgetriebenen Softwareentwicklung kompakt zusammengefasst.

2.1 ENTWURF EINGEBETTETER SOFTWARE

Der Entwurf eingebetteter Software muss mit dem Entwurf von Hardware abgestimmt werden. Der starke Zusammenhang ergibt sich aus der Anforderung, mit der physischen Umgebung des Systems in Echtzeit zu interagieren. Das ist ohne spezialisierte Hardware nicht möglich.
Vor diesem Hintergrund ist ein besonders planvolles Vorgehen bei der Softwareentwicklung wichtig. Die Automobilindustrie orientiert sich dazu genauso wie die anderen in der Einleitung genannten Branchen am sogenannten „V-Modell“, auf das im folgenden Abschnitt kurz eingegangen wird. Gleichzeitig dringt aus anderen Bereichen der Softwareentwicklung, die sich vorwiegend mit nichtkritischer Software beschäftigen (z.B. „Apps“ und Desktopsoftware), auch ein gänzlich anderes Prozessmodell in die vergleichsweise konservativen Brachen vor: die sogenannte „agile Softwareentwicklung“. Darauf wird in Abschnitt 2.1.2 eingegangen.

2.1.1 *Das V-Modell*

Das V-Modell beschreibt ein Vorgehen zur Entwicklung komplexer Systeme. Es ist wichtig anzumerken, dass es *das* V-Modell eigentlich nicht gibt. Es existieren heute recht unterschiedliche Vorstellungen, die im jeweiligen Kontext auch alle einen Sinn ergeben. Eine äußerst präzise Definition liefert das „V-Modell XT“, ein durch den Beauftragten der Bundesregierung für Informationstechnik bereitgestellter Standard [Onl/VMXT]. Dieses „deutsche V-Modell“ hat sich von der grundlegenden Idee hinter der Darstellung in Form eines „V“ aber schon recht weit entfernt.
In seiner ursprünglichsten Form ist das V-Modell eine Ergänzung des Wasserfallmodells um dedizierte Testphasen zur Verifikation und Validierung. Das Wasserfallmodell wird meistens mit einer berühmten Veröffentlichung von Winston Royce aus dem Jahr 1970 in Verbindung gebracht („Managing the Development of Large Software Systems“ [Roy70]), in welcher er den schrittweisen Entwicklungsprozess zum ersten Mal formal beschreibt. Was dabei manchmal übersehen wird ist die Tatsache, dass das Wasserfallmodell von Royce als Negativbeispiel benutzt wurde [Chro8].
Seine Kritik am Wasserfallmodell betrifft auch die in dieser Arbeit verwendete Darstellung des V-Modells (Abbildung 2), welche sich bezüglich Terminologie und Anordnung der Phasen am Entwicklungsprozess nach ISO 26262 orientiert. Leider suggeriert diese Darstellung einen sequentiellen Prozess, in dem die Verifikation und Validierung erst am Ende erfolgt. In diesem Fall werden Fehler erst spät bemerkt, und ihre Beseitigung ist entsprechend teuer. Genau dieses Problem zeigt Royce bereits 1970 am Wasserfallmodell, und in der Praxis sollte kein Produkt auf diese Weise entwickelt werden.

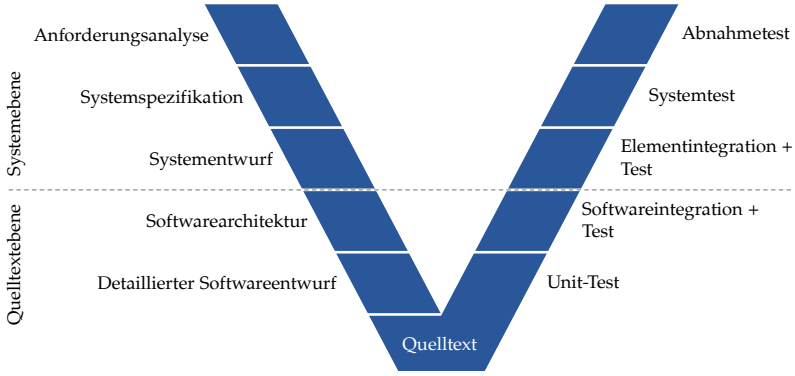


Abbildung 2: In dieser Arbeit verwendete Darstellung des V-Modells

Die Darstellung in Abbildung 2 bietet jedoch einen hervorragenden Überblick über den Entwicklungsprozess und erleichtert das Verorten verschiedener Aktivitäten erheblich. In der Praxis wird das „V“ aber nicht in einem Durchgang durchlaufen, sondern es finden Phasen- und Abschnittsweise Iterationen statt, die teilweise auch parallel möglich sind. Insbesondere werden die Phasen Systemspezifikation und Systementwurf von Methoden zur frühen Verifikation und Validierung begleitet (siehe dazu Kapitel 4). Es gibt unterschiedliche Versuche diesen Sachverhalt in der „V“-Darstellung deutlicher zum Ausdruck zu bringen, mit teilweise zweifelhaftem Nutzen.

2.1.2 *Agile Softwareentwicklung*

Seit einigen Jahren kommt man um das Schlagwort „agil“ bezüglich dem Vorgehen bei der Softwareentwicklung nicht mehr herum. Eine jährlich im Auftrag von VersionOne durchgeführte Umfrage kommt zu dem Ergebnis, dass im Jahr 2014 bei 94% der Umfrageteilnehmer agile Entwicklung im Unternehmen praktiziert wird [VO15]. 2011 waren das bereits 80% [VO13]. Zur Bewertung der tatsächlichen Verbreitung agiler Vorgehensweisen eignen sich diese Zahlen nur bedingt; unbestritten ist jedoch der rasante Zulauf, den sie erfahren haben.
Hinter dem Begriff steckt in erster Linie kein konkreter Prozess, sondern eine Aufstellung von Werten. Aufgeschrieben sind sie im *Agile Manifest*: *Agile Manifest*

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - Individuals and interactions over processes and tools
> - Working software over comprehensive documentation
> - Customer collaboration over contract negotiation
> - Responding to change over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.
>
> — *[BBB+01]*

Agile Softwareentwicklung ist also der Versuch, den Entwicklungsprozess durch neue Priorisierung zu verbessern. Während in der

traditionellen Sichtweise die Einhaltung der Prozessvorgaben sowie exakte Dokumentation als wesentliche Erfolgskriterien gelten, werden genau diese Merkmale nun als „weniger wichtig“ betrachtet.

Vor diesem Hintergrund mag es fast widersprüchlich erscheinen, „agile Prozesse“ zu definieren, aber natürlich gibt es sie trotzdem. Wesentlich ist, dass der Prozess eine hohe Dynamik während der Durchführung ermöglicht. Der wohl bekannteste agile Prozess nennt sich *Scrum*. Bei Scrum steht das Entwickler-Team im Mittelpunkt. Die grundlegende Idee ist dessen stärkere Einbeziehung in das Projektmanagement, da es das notwendige Wissen bündelt.

Scrum

Der Scrum-Prozess ist vereinfacht in Abbildung 3 dargestellt. Es wird in kurzen Iterationszyklen von wenigen Wochen entwickelt; im Jargon von Scrum heißt ein solcher Zyklus Sprint. Dem Sprint geht ein Planungstreffen voraus, in welchem die Teammitglieder aus einer priorisierten Liste von Anforderungen (Product Backlog) die Aufgaben für den kommenden Sprint heraussuchen. Die Schätzung der Aufwände, Kriterien der Erfolgskontrolle und Vorgehensfragen werden dabei vom Team diskutiert. Dabei entsteht ein Plan für den kommenden Sprint (Sprint Backlog). Der Sprint wird begleitet von üblicherweise täglichen Treffen aller Teammitglieder mit einer Dauer von maximal 15 Minuten. Der Sprint wird durch ein Sprint Review abgeschlossen, in dessen Rahmen auch die Aktualisierung des Product Backlogs für die nächste Iteration erfolgt.

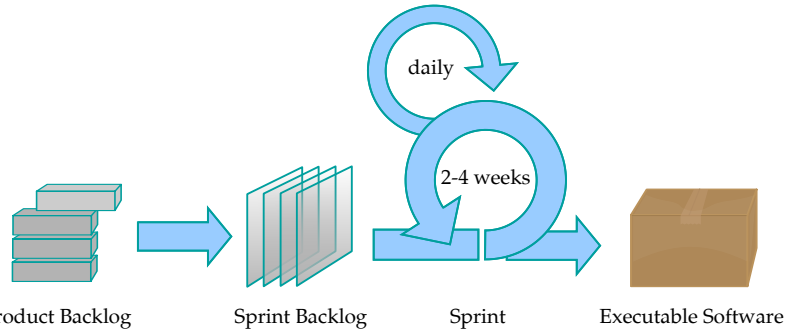


Abbildung 3: Scrum-Prozess und Artefakte [Wiki/Scr]

Agile Methoden und Eingebettete Systeme

Nach der gängigen Auffassung muss am Ende des Sprints eine lauffähige Version der Software zur Verfügung stehen. Im Umfeld eingebetteter Systeme ist das wegen dem starken Zusammenhang mit dem Hardwareentwurf weniger einfach möglich als in anderen Bereichen. Ein weiteres Hemmnis ist, dass rechtliche Fragen zur Haftung oder Vertragserfüllung bei agiler Entwicklung nicht so

klar wie bei klassischen Entwicklungsprozessen beantwortet werden können, da die hierzu relevanten Dokumente erst während der Projektdurchführung und unter Beteiligung beider Seiten konkretisiert werden. Gerade für kritische Software sind Haftungsfragen aber sehr relevant (vgl. Abschnitt 1.1).
Es wird daher versucht, agile Methoden mit herkömmlichen Entwicklungsprozessen zu kombinieren. Häufig werden dazu Teilabschnitte in einem klassischen Entwicklungsprozess agil durchgeführt. Prinzipiell kann auf diese Weise ein Großteil des Gesamtprozesses auf agile Entwicklung umgestellt werden, lediglich die Hardwareentwicklung muss weiterhin traditionell erfolgen [Mar14].
Dennoch bleibt die Anwendbarkeit agiler Methoden für eingebettete Software schwierig, und das hängt auch mit dem Aufwand zur Testautomatisierung zusammen. Dieser ist für eingebettete Software deutlich höher als in Bereichen, aus denen die Idee agiler Entwicklung ursprünglich kommt (die Gründe dafür wurden bereits in der Einleitung ersichtlich). Bei der agilen Entwicklung ist der Test jedoch wegen der kurzen Iterationszyklen besonders stark mit dem übrigen Entwicklungsprozess verknüpft.[1] Übersteigt der Aufwand zur Testimplementierung ein gewisses Maß, wird der Entwicklungsprozess stark ausgebremst, und es könnte passieren, dass sich der agile Ansatz als ineffizient erweist. Das primäre Ziel dieser Arbeit, die Reduzierung des Aufwands zur Testimplementierung, kann also eine entscheidende Rolle dabei spielen, agilen Methoden im Umfeld eingebetteter Systeme den Weg zu ebnen.

2.1.3 *Software in Verteilten Systemen*

System von Systemen

Eingebettete Systeme sind häufig selbst Einzelkomponenten in einem größeren Systemverbund. Man spricht hier auch von einem *System von Systemen*. Ein gutes Beispiel sind Steuergeräte (ECUs) im Automobil. Für manche Funktionen müssen mehrere ECUs wie in Abbildung 4 dargestellt vernetzt werden. Die Einzelkomponenten sind räumlich voneinander getrennt, die tatsächliche Anwendungsfunktion ist jedoch erst im Verbund gegeben. Die Anwendungsfunktion manifestiert sich in der Software, welche auf den einzelnen ECUs ausgeführt wird.
Damit der Entwurf entsprechender Software nicht zu kompliziert wird, kann eine zusätzliche Softwareschicht eingeführt werden, die von den einzelnen ECUs abstrahiert. Dieser zunächst rein technisch

1 Der agile Ansatz geht beim „Test Driven Development“ sogar so weit, die Testimplementierung *vor* der Softwareimplementierung zu fordern [Bec03; Gre11].

motivierte Gedanke wird in einigen Bereichen wie der Automobilindustrie auch durch einen wirtschaftlichen Aspekt getrieben. Das wird im folgenden Abschnitt erklärt.

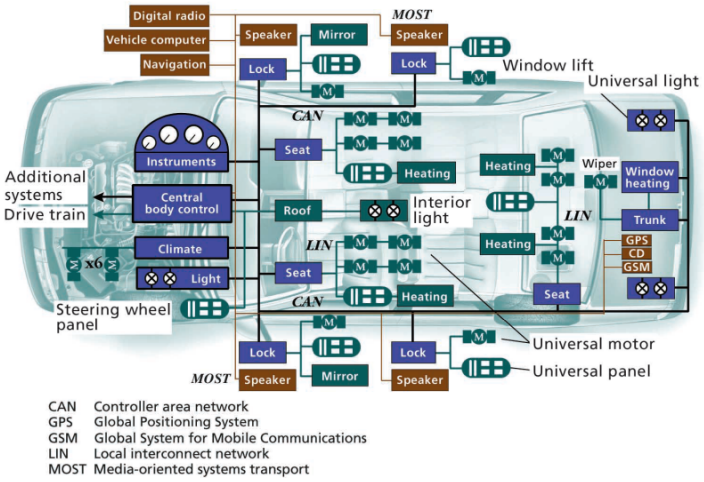


Abbildung 4: Steuergeräte im Automobil [LH02]

2.1.4 *Verteilte Softwareentwicklung*

In einigen Bereichen muss nicht nur verteilte Software entwickelt werden, sondern die Entwicklung selbst ist auf mehrere Akteure verteilt. Ein gutes Beispiel ist die Automobilindustrie: historisch wird die Software einer ECU vom Zulieferer (Tier-1) entwickelt. Der Automobilhersteller (OEM) ist jedoch aus unterschiedlichsten Gründen[2] gezwungen, Steuergeräte verschiedener Anbieter in ein Fahrzeug zu integrieren, die dennoch gemeinsame Funktionen realisieren. An der Außenbeleuchtung eines PKWs können beispielsweise über 10 ECUs beteiligt sein [Har01].
Die Anwendungssoftware muss daher vom OEM angepasst werden können. Zudem kommt es im Fall wettbewerbsdifferenzierender Funktionen vermehrt vor, dass der OEM entscheidend an der Funktionsentwicklung beteiligt ist, oder diese sogar komplett selbst übernimmt. Beispiele finden sich vor Allem im Bereich der Fahrerassistenz und mit Blick auf das autonome Fahren (z.B. erweiterter Abstandsregeltempomant). Große Teile der Software bleiben jedoch generisch. Das umfasst das Betriebssystem sowie Standardkomponenten wie Protokollstacks, Diagnosemodule etc.

2 Kosten, Verfügbarkeit, Redundante Lieferketten, Konzernpolitik, ...

AUTOSAR

In der Automobilindustrie setzt hier die *AUTOSAR* Entwicklungspartnerschaft an. AUTOSAR steht für *AUT*omotive *O*pen *S*ystem *AR*chitecture und hat das Ziel die Softwareentwicklung für Anwendungsfunktionen unabhängig von der Steuergerätehardware sowie Standardsoftware zu ermöglichen. Dazu sind eine Reihe von Schnittstellen spezifiziert, welche in Abbildung 5 dargestellt sind. Die zentrale Komponente der Softwarearchitektur ist die AUTOSAR Runtime Environment (RTE), welche durch Bereitstellung eines virtuellen Kommunikationsbusses (Virtual Functional Bus) von der physischen Kommunikationsinfrastruktur im ECU-Verbund abstrahiert [KF09]. Im Standard sind nun die Schnittstellen zwischen der RTE und den übrigen Softwarekomponenten eindeutig spezifiziert, sodass die Implementierung einzelner Blöcke ausgetauscht werden kann.

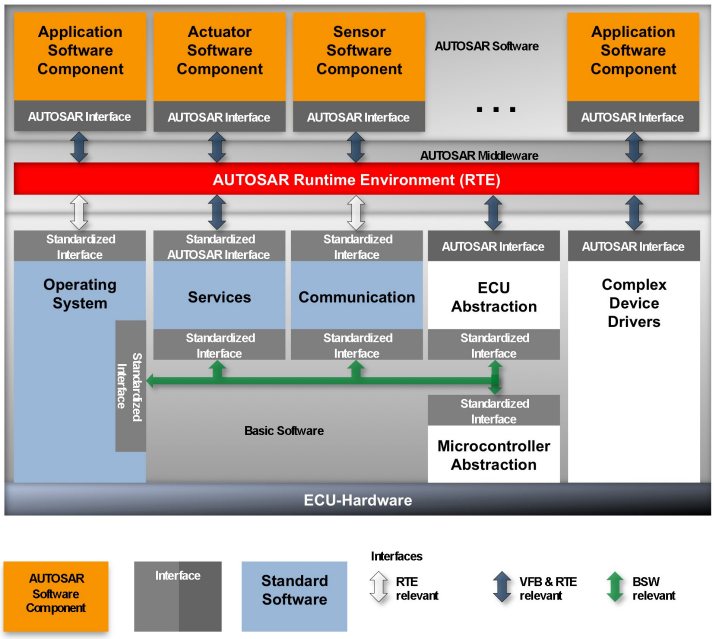


Abbildung 5: Blockdiagramm der AUTOSAR Architektur [Onl/Auto]

Auch in anderen Branchen werden verteilt entwickelte Systeme unterschiedlicher Hersteller zu einem Gesamtprodukt integriert. Beispiele sind die Luft- und Raumfahrt oder die Industrieautomation. Technisch unterscheiden sich entsprechende Standards jedoch durchaus deutlich. In der Industrieautomation findet sich die Norm IEC 62541 („Open Platform Communications Unified Architecture“) [Std/IEC10b]. In der Luftfahrt existieren gleich

zwei konkurrierende Standards: ARINC 653 („Avionics Application Software Standard Interface“) [Std/AR13] mit Ursprung 1996, sowie STANAG 4626 („Modular and Open Avionics Architectures“) [Std/AC10], dessen Entwicklung bereits 1990 begonnen wurde und seit 2010 eine Europäische Norm ist.
Testautomatisierung ist für Regressionstest der Implementierung genannter Standards außerordentlich wichtig. Schließlich wird dieselbe Codebasis auf unterschiedlichen Hardwareplattformen eingesetzt, muss aber auf jeder individuell getestet werden. Die Testfälle sind dabei weitgehend identisch und es ist wichtig, dass die entsprechende Testimplementierung ebenfalls wiederverwendet werden kann.

2.2 BEGRIFFLICHKEITEN ZUM SOFTWARETEST

Es wurde bereits einiges über den Test von Software gesagt, aber noch nichts über den Begriff „Analyse“, der sich ebenfalls im Titel dieser Arbeit findet. Das wird an dieser Stelle nachgeholt, wobei der Begriff „Debugging“ hinzugezogen wird, da alle drei in einem engeren Zusammenhang stehen. Anschließend folgen noch einige weitere Definitionen, um mögliche Missverständnisse auszuschließen.

2.2.1 *Test vs. Analyse vs. Debugging*

Der Zusammenhang von Test, Analyse und Debugging wird am deutlichsten, wenn man die zugehörigen Fragestellungen betrachtet. Die Analyse gibt eine Antwort auf die Frage: „*Wie* verhält sich das System?“. Beim Test hingegen wird gefragt: „Verhält sich das System *korrekt*?“. Und Debugging klärt schließlich: „*Warum* verhält sich das System *nicht* korrekt?“.
Etwas präziser erklärt: Mit Analyse ist die Interpretation von direkt beobachteten Tatsachen gemeint. Das kann im Rahmen eines Tests geschehen, muss aber nicht. Beim Test wird die gewonnene Interpretation anhand von *Erwartungen* bewertet, wobei die Erwartung häufig aus der Spezifikation abgeleitet wird. Es ist durchaus möglich eine Analyse ohne konkrete Erwartung durchzuführen, beispielsweise um ein Gefühl für die Möglichkeiten zu erhalten. Ein Test ist ohne Erwartung hingegen nicht möglich.

Erwartung

Wird der Erwartung nicht entsprochen, hat man einen Fehler gefunden. Dieser kann nun im untersuchten System oder in der Erwartung liegen. An dieser Stelle beginnt man mit dem Debugging.

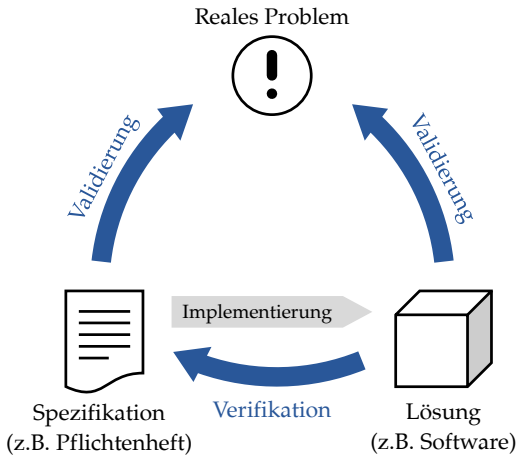


Abbildung 6: Verifikation vs. Validierung

Der wesentliche Teil ist die dabei Identifikation der Fehlerursache. Meistens wird jedoch auch die anschließende Korrektur, also die Anpassung der Erwartung und/oder des Systems als Teil des Debuggings angesehen (so auch in dieser Arbeit).

2.2.2 *Weitere Begrifflichkeiten*

Ziel dieses Abschnitts ist die Klarstellung allgemeiner Begrifflichkeiten bezüglich dem Test von Software. Das ist notwendig, weil in dieser Arbeit Begriffe verwendet, die inhaltlich sehr nah beieinander liegen, aber eine klare Differenzierung erfordern. Außerdem sind manche Begriffe nicht eindeutig, weil sie in der Literatur unterschiedlich verwendet werden.

Verifikation und Validierung

Die Begriffe Verifikation und Validierung werden leider nicht immer so klar unterschieden wie in Abbildung 6. Im Zuge der Verifikation ist nachzuweisen, dass die realisierte Lösung eines Problems der Spezifikation genügt. Diese Spezifikation ist üblicherweise ein Teil der vertraglichen Vereinbarung zwischen Auftraggeber und Auftragnehmer und bei der Klärung von Haftungsfragen die relevante Grundlage. Demgegenüber wird bei der Validierung geprüft, ob die Spezifikation bzw. die Lösung den tatsächlichen Bedürfnissen für das zugrundeliegende Problem entspricht.

Barry Boehm bringt den Unterschied folgendermaßen auf den Punkt: Verifikation gehe der Frage nach „Are we building the product right?“, während Validierung sich mit der Frage „Are we building the right product?“ beschäftige. Die „Lösung“ in der Abbildung entspricht dem „product“ bei Boehm.
Tests sind in diesem Zusammenhang ein Hilfsmittel, das sowohl der Verifikation als auch der Validierung dienen kann. In dieser Arbeit steht die Verifikation im Vordergrund.

Testfall

Ein Testfall beschreibt alles Notwendige zum Test elementarer Eigenschaften des Testobjekts. Das umfasst die Vorbedingungen und Eingabedaten, die Schritte zur Durchführung (Testablauf) sowie die erwarteten Reaktionen des Testobjekts.

Testspezifikation und Testimplementierung

Die Testspezifikation ist die informelle Beschreibung der Testfälle und deren Dokumentation [Saxo8]. Traditionell entsteht sie durch die manuelle Analyse der Produktspezifikation sowie der Benutzeranforderungen. Wird modellbasiertes Testen (siehe Abschnitt 2.4.4) praktiziert, können Teile der Testspezifikation aber auch automatisch erzeugt werden. Demgegenüber ist die Testimplementierung ein maschinenlesbares Format zur automatischen Ausführung und Bewertung der Testfälle. Die Testimplementierung wird traditionell ebenfalls manuell erstellt, aber auch hier können Teile durch Modellierung und Codegenerierung automatisiert werden.
In der Praxis ist nicht immer eine scharfe Trennung zwischen Testspezifikation und Testimplementierung gegeben. In vielen Bereichen der Softwareentwicklung ist Testautomatisierung so selbstverständlich, dass für manuell durchgeführte Softwaretests kaum ein Bewusstsein existiert. Der Begriff „Testspezifikation“ wird dann als Synonym für die Testimplementierung verwendet. Im Rahmen dieser Arbeit ist eine Unterscheidung jedoch wichtig.

Testphase

Der Begriff Testphase wird in dieser Arbeit im Sinne der ISO 26262 verwendet, also für die einzelnen Abschnitte im V-Model. In der Literatur ist dafür häufiger der Begriff „Teststufe“ zu finden, während der Begriff „Testphase“ etwas Anderes meint. So zählt das International Software Testing Qualifications Board (ISTQB) die folgenden

Phasen auf: Testplanung und Steuerung, Testanalyse und Testentwurf, Testrealisierung und Testdurchführung, Bewertung von Endekriterien und Bericht, sowie Abschluss der Testaktivitäten [ISTQB11]. Die Bezeichnung „Testaktivitäten" trifft das hervorragend. Etwas prägnanter sind die Testaktivitäten nach dem Phasenmodell von Martin Pol: Testplanung, Testvorbereitung, Testspezifikation, Testdurchführung, Testauswertung und Testabschluss [PKS02].

Testaktivität

Regressionstest

Ein Regressionstest ist das erneute Ausführen bestehender Testfälle für bereits getestete Teile Software. Damit soll sichergestellt werden, dass eine Modifikation keine unbeabsichtigten Auswirkungen hat. Mögliche Modifikationen sind die Korrektur von Fehlern, allgemeine Verbesserungen (z.B. Performancesteigerung) oder zusätzliche Features. Testautomatisierung ist für Regressionstests aufgrund der häufigen Wiederholung besonders sinnvoll.

Der Regressionstest wird manchmal zu den diversifizierenden Testmethoden gezählt, z.B. in [Lig09]. Das passt nicht zu der vorhergehenden Definition, denn ein direkter Vergleich des Verhaltens von zwei unterschiedlichen Softwareversionen findet dort nicht statt. Stattdessen wird beim Regressionstest lediglich das jeweilige Test*ergebnis* gegenübergestellt, die jeweils zuvor erfolgte Bewertung ist unabhängig vom Ablauf des Referenztests.

Diversifizierender Test

Ein echtes Beispiel für diversifizierende Testmethoden ist der Back-to-Back Test. Hier erfolgt die Bewertung des Testfalls tatsächlich durch den Vergleich des Verhaltens verschiedener Softwareversionen. Falls modellbasierte Entwicklung (siehe Abschnitt 2.4) praktiziert wird, lässt sich ein spezieller Back-to-Back Test relativ einfach realisieren, nämlich durch den Vergleich des Modellverhaltens mit dem generierten Code. Im Allgemeinen ist der Back-to-Back Test jedoch ausgesprochen teuer, da mehrere parallel entwickelte Softwareversionen benötigt werden (n-Versionen-Programmierung).

Back-to-Back Test

Testart

Der Begriff Testart bezieht sich auf die Art und Weise der Testdurchführung. Hier wird zwischen statischem und dynamischem Test unterschieden. Bei statischen Tests wird die Software nicht ausgeführt, sondern ihre Implementierung, beispielsweise der Quelltext, analysiert. Statische Tests spielen in dieser Arbeit keine Rolle.[3]

3 Ein guter Überblick über statische Testmethoden findet sich in [Lig09; Gru13].

Diese Arbeit betrachtet also ausschließlich den dynamischen Test, d.h. die Software wird ausgeführt und ihr Verhalten beobachtet. Dabei muss bezüglich dem Ort der Ausführung unterschieden werden, was in Abschnitt 3.1 genauer betrachtet wird.

Testautomatisierung

Automatisierung lässt sich auf die meisten Testaktivitäten anwenden (mit Ausnahme von Testplanung und Testvorbereitung, die als nicht automatisierbar gelten). In dieser Arbeit bezieht sich der Begriff Testautomatisierung jedoch ausschließlich auf die Aktivitäten Testdurchführung und Testauswertung. Abbildung 7 verdeutlicht die verwendete Terminologie.

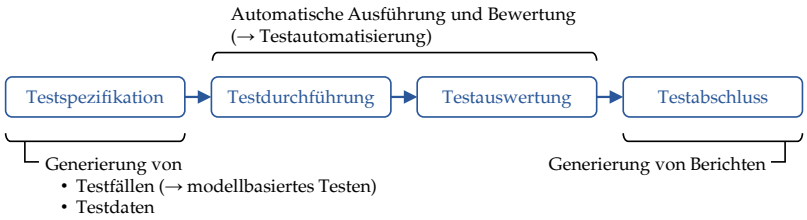


Abbildung 7: Automatisierung von Testaktivitäten

Begriffe zum Debugging

Zu Beginn von Abschnitt 2.2 wurde bereits der Begriff Debugging eingeführt und von Test und Analyse unterschieden. Auch wenn Debugging demnach nicht im Fokus dieser Arbeit steht, spielen die dazu eingesetzten Methoden in den späteren Kapiteln eine wichtige Rolle. Aus diesem Grund müssen die damit in Zusammenhang stehenden Begriffe ebenfalls kurz erläutert werden:

- *Breakpoint*: Bezeichnet einen Haltepunkt im Programmablauf der Software. Markiert eine Instruktion, bei deren Erreichen die Ausführung angehalten werden soll. Die markierte Instruktion wird nicht mehr ausgeführt.
- *Watchpoint*: Bezeichnet eine Bedingung, bei deren Eintreten die Ausführung angehalten werden soll. Ein Beispiel ist das Beschreiben einer Variablen mit einem bestimmten Wert. Ein Spezialfall ist die Kombination mit einem Breakpoint (z.B. Anhalten an Instruktion *foo*, falls Variable *bar* Wert *x* besitzt), was als Conditional Breakpoint bezeichnet wird.

- *Stepping*: Bezeichnet die gezielte Ausführung weniger Instruktionen aus dem angehaltenen Zustand heraus, bevor die Ausführung wieder angehalten wird. Typisch sind folgende Schrittweiten: genau eine Instruktion (single step), bis zum nächsten Rücksprung (step return) oder zur nächsten Instruktion, welche auf der aktuellen oder einer vorhergehenden Stufe in der Aufrufhierarchie (call stack) liegt, d.h. es wird in Unterfunktionen nicht angehalten (step over).

2.3 FORMALE SPRACHEN

Dieser Abschnitt kann keine umfassende Einführung in das Thema der Formalen Sprachen liefern, denn das würde ihren Rahmen sprengen. Das Ziel ist vielmehr die Einführung einer speziellen Darstellung von Typ-2 Grammatiken in der Chomsky-Hierarchie, welche in Kapitel 6 zur Definition einer neuen Programmiersprache für Testfälle verwendet wird. Es werden im Folgenden daher nur die wichtigsten Grundbegriffe erklärt, und bei Kenntnis des Themas kann direkt zu Abschnitt 2.3.5 gesprungen werden. Eine weiter gefasste Einführung findet sich im Buch „Formale Sprachen: Endliche Automaten, Grammatiken, lexikalische und syntaktische Analyse“ von Hans-Joachim Böckenhauer [BH13]. Diesem Buch sind auch die im folgenden Abschnitt hervorgehobenen Definitionen entnommen.

2.3.1 *Grundlegende Begriffe*

Alphabet

Eine endliche, nichtleere Menge Σ heißt *Alphabet*. Die Elemente eines Alphabets werden Buchstaben (Zeichen, Symbole) genannt.

Im Kontext von Programmiersprachen entspricht das Alphabet dem Zeichensatz des Quelltextes, z.B. ASCII[4] oder UTF-8[5].

Wort

Sei Σ ein Alphabet. Ein *Wort* über Σ ist eine endliche (eventuell leere) Folge von Buchstaben aus Σ. Das leere Wort ϵ ist die leere Buchstabenfolge.

4 American Standard Code for Information Interchange
5 8-Bit Universal Character Set Transformation Format

Der Fachbegriff „Wort“ ist etwas irreführend. Die Definition entspricht dem, was man umgangssprachlich als „Text“ bezeichnen würde. In Zusammenhang mit einer Programmiersprache ist der Begriff „Quelltext“ geläufig.

Sprache

Eine Menge von Wörtern über einem Alphabet Σ bezeichnen wir als eine *Sprache* über Σ.

Wortproblem

Eine Sprache L ist also eine Teilmenge aller Wörter über Σ. Ein Wort aus L wird auch ein gültiges Wort der Sprache L genannt. Die Fragestellung, ob ein Wort $w \in \Sigma^*$ ein gültiges Wort einer gegebenen Sprache ist, wird als *Wortproblem* bezeichnet.[6] Eine Sprache wird *entscheidbar* genannt, wenn es einen Algorithmus gibt, der das Wortproblem in endlicher Zeit löst.

Token

Innerhalb einer Sprache lassen sich bestimmte Zeichenfolgen identifizieren, sogenannte *Token*, die nicht weiter zerlegbar sind. Beispiele sind die Schlüsselwörter[7], Bezeichner und Literale der Sprache.

2.3.2 *Formalismus zur Definition von Sprachen*

Grammatik

Zur Definition einer Sprache wird ein Formalismus benötigt, der die Sprache exakt beschreibt. Ein solcher Formalismus ist mit der *Grammatik* einer Sprache gegeben. Eine Grammatik ist ein 4-Tupel $G = (N, \Sigma, P, S)$ bestehend aus:

- N, einer Menge von Zeichen, die *Nichtterminalzeichen* genannt werden.
- Σ, dem Alphabet. Die Zeichen des Alphabets werden jetzt *Terminalzeichen* genannt. Die Schnittmenge von N und Σ ist leer.
- P, einer Menge von *Produktionsregeln*, die beschreiben, wie sich Worte aus $\Sigma \cup N$ ineinander überführen lassen.
- $S \in N$, dem Startsymbol.

Eine Produktionsregel aus P wird in der Form $A \to B$ notiert. Das besagt, dass in einem Wort $w \in N \cup \Sigma$ jedes Vorkommen von A durch B ersetzt werden darf. Damit erhält man ein Wort $w' \in N \cup \Sigma$.

6 Σ^* ist die kleenesche Hülle von Σ. Das ist die Menge aller Wörter, die durch beliebige Konkatenation der Zeichen in Σ gebildet werden können.

7 Hier zeigt sich erneut die Unzulänglichkeit des Wortbegriffs: Schlüsselwörter sind in der Regel keine gültigen Wörter derselben Sprache.

Man sagt das Wort w' wurde durch die Produktionsregel $A \to B$ aus w *abgeleitet*, und verwendet dafür folgende Notation: $w \rightsquigarrow_{A \to B} w'$.
Lässt sich ein Wort w' durch die Anwendung beliebiger Produktionsregeln der Grammatik G in einer beliebigen Anzahl von Schritten aus w ableiten, notiert man das in der Form $w \rightsquigarrow_G w'$. Die von der Grammatik definierte Sprache $L(G)$ erhält man jetzt ausgehend vom Startsymbol S:

$$L(G) := \{w \in \Sigma^* | S \rightsquigarrow_G w\}$$

Durch Einschränkungen bei den Produktionsregeln lassen sich Klassen bilden, die unterschiedlich umfangreiche Sprachen enthalten. Die bekannteste Einteilung dieser Art ist die Chomsky Hierarchie, welche im nächsten Abschnitt erläutert wird.

2.3.3 *Chomsky Hierarchie*

Die Chomsky Hierarchie wurde 1956 von Noam Chomsky beschrieben. Sie unterteilt Grammatiken in vier Klassen (Typ-0 bis Typ-3), wobei mit aufsteigendem Typ die erlaubten Produktionsregeln immer weiter eingeschränkt werden. Die Chomsky Hierarchie ist unter anderem deswegen interessant, weil die Laufzeitkomplexität zur Lösung des Wortproblems mit dem jeweils nächsthöheren Typ abnimmt.

- *Typ-0 Grammatik*: Die einzige Einschränkung bezüglich den Produktionsregeln besteht darin, dass auf der linken Seite mindestens ein Nichtterminalzeichen steht. Die von Typ-0 Grammatiken erzeugten Sprachen sind nicht entscheidbar.
- *Typ-1 Grammatik*: Die Produktionsregeln müssen *längenbeschränkt* sein, d.h. sie haben die Form $\alpha A \beta \to \alpha \gamma \beta$, wobei gilt:
  - $\alpha, \beta \in \Sigma^*$, also beliebige Wörter über dem Alphabet
  - $A \in N$
  - $\gamma \in (N \cup \Sigma)^* \setminus \{\epsilon\}$

  Wenn das Startsymbol S auf keiner rechten Seite vorkommt, ist außerdem die Regel $S \to \epsilon$ erlaubt. Die von Typ-1 Grammatiken erzeugten Sprachen sind entscheidbar und die Laufzeitkomplexität liegt in $2^{\mathcal{O}(n)}$.

- *Typ-2 Grammatik*: Produktionsregeln müssen auf der linken Seite genau ein Nichtterminalzeichen besitzen, auf der rechten Seite ist weiterhin eine beliebige, nichtleere Folge von Zeichen aus $N \cup \Sigma$ erlaubt. Die Sonderregel $S \to \epsilon$ (siehe Typ-1) ist auch hier möglich. Die Laufzeitkomplexität zur Lösung des Wortproblems reduziert sich auf $\mathcal{O}(n^3)$.
- *Typ-3 Grammatik*: Produktionsregeln dürfen auch auf der rechten Seite nur noch ein Terminalzeichen und eventuell ein Nichtterminalzeichen besitzen. Dabei muss die Position des Nichtterminalzeichens auf der rechten Seite (falls vorhanden) einheitlich sein, d.h. in allen Produktionsregeln muss es entweder links oder rechts vom Terminalzeichen stehen. Außerdem ist die Produktionsregel $A \to \epsilon$ mit $A \in N$ erlaubt. Die Laufzeitkomplexität zur Lösung des Wortproblems liegt jetzt nur noch in $\mathcal{O}(n)$.

Die von Grammatiken des jeweiligen Typs beschriebenen Sprachen stehen in einer echten Inklusionsbeziehung zueinander. Es gilt:

$$L_{\text{Typ-3}} \subset L_{\text{Typ-2}} \subset L_{\text{Typ-1}} \subset L_{\text{Typ-0}}$$

2.3.4 *Bedeutung Formaler Sprachen in der Praxis*

Sprachen werden in der Informatik benutzt, um Algorithmen präzise zu beschreiben, sodass sie von einem Rechner eindeutig verstanden und ausgeführt werden können. Zur Abgrenzung gegenüber der *natürlichen* Sprache, bei welcher die Kommunikation im Vordergrund steht, spricht man von einer *formalen* Sprache.
Rechner verstehen üblicherweise nur eine „Maschinensprache“ basierend auf einem binären Alphabet (meistens dargestellt durch die Zeichen „0“ und „1“). Gültige Wörter dieser Sprache sind beliebige Folgen von Instruktionen der jeweiligen Rechnerarchitektur. Weil es außerordentlich mühselig ist, komplexe Algorithmen in Maschinensprache zu formulieren, wurden Sprachen mit einer für den Menschen besser geeigneten Darstellung (Syntax) entworfen. Diese sind für den Rechner jedoch nicht direkt verständlich und müssen erst übersetzt werden. Das lässt sich ebenfalls automatisieren, das entsprechende Programm nennt man Compiler. Der erste Schritt dieser Übersetzung wird *Parsen* genannt.

Parsen

Vor dem Parsen ist der Quelltext nichts anderes als eine ungeordnete Folge von Zeichen aus dem Alphabet. Durch einen Parser wird

er in eine abstrakte Struktur überführt, die zur späteren Weiterverarbeitung besser geeignet ist. Als Erstes werden dazu die einzelnen Token identifiziert. Diesen Schritt nennt man lexikalische Analyse, das zuständige Programm heißt lexikalischer Scanner oder kurz Lexer. Der eigentliche Parser erzeugt daraus in einem zweiten Schritt einen Baum, in dem jeder Knoten einem Sprachkonstrukt (z.B. einer Funktion, Schleife oder eines Ausdrucks) entspricht. Diese Struktur nennt man den *abstrakten Syntaxbaum* (AST). Alle weiteren Schritte zur Verarbeitung basieren auf dem AST.

Abstrakter Syntaxbaum

Bezüglich dem Parsen spielt das Wortproblem eine zentrale Rolle, denn es entspricht der Frage, ob die Sprache von einem Parser „erkannt" werden kann, oder nicht, und falls ja, mit welcher Laufzeitkomplexität dies möglich ist. Weil das Wortproblem für Typ-0 Sprachen nicht entscheidbar ist, und für Typ-1 Sprachen exponentielle Laufzeit besitzt, kommen solche Sprachen für praktische Anwendungen kaum in Frage. Die Syntax der meisten Programmiersprachen wird daher durch eine Typ-2 Grammatik spezifiziert. Typ-2 Sprachen werden auch als *kontextfreie Sprachen* bezeichnet, weil die Anwendbarkeit einer Regel nicht von der umgebenden Syntax abhängt (denn auf der linken Seite steht immer ein einzelnes Nichtterminalzeichen).

Typ-3 Sprachen sind aufgrund der linearen Laufzeitkomplexität zur Lösung des Wortproblems ebenfalls äußerst praxisrelevant. Sie werden auch als *reguläre* Sprachen bezeichnet, weil sie durch reguläre Ausdrücke dargestellt werden können. Reguläre Ausdrücke werden unter anderem als Suchmuster verwendet, was wieder direkt auf das Wortproblem zurückzuführen ist („lässt sich der gegebene Text durch das Suchmuster erzeugen?").

2.3.5 *Notation von Typ-2 Grammatiken*

In der Praxis werden also reguläre Ausdrücke zur Darstellung von Typ-3 Grammatiken verwendet. Auch zur Beschreibung der Syntax von Programmiersprachen, was eine Typ-2 Grammatik erfordert, ist die mathematische Notation aus Abschnitt 2.3.2 ungeeignet. Das wird recht schnell deutlich, wenn man versucht die notwendigen Produktionsregeln für elementare Sprachkonstrukte aufzustellen (z.B. Literale wie „`123`" und „`1.0f`", Bezeichner wie „`foo`" und „`_bar0`" oder Ausdrücke wie „`foo + 123`").

In der Praxis werden daher andere Notationen verwendet, die häufig auf der „Backus-Naur-Form" (BNF) basieren. Die BNF wurde erstmals Ende der 50er Jahre in Zusammenhang mit der Program-

miersprache Algol 60 eingesetzt. Es gibt verschiedene Erweiterungen der BNF, von denen manche als Eingabe eines sogenannten Parsergenerators geeignet sind. Ein Parsergenerator erzeugt anhand der Eingabe automatisch einen Parser für die beschriebene Sprache. Ein Beispiel ist das von Terence Parr entwickelte Werkzeug ANTLR[8].

Parsergenerator

Die Grammatik eines Parsergenerators muss nicht nur die Syntax der Sprache beschreiben, sondern auch, wie daraus der AST zu erstellen ist. Diese Zusatzinformationen erschweren einem Menschen das „Lesen“ der Grammatik. Andererseits sind manche Informationen hilfreich zum Verständnis der Semantik. Das führt zu der in dieser Arbeit verwendeten Notation.

In dieser Arbeit verwendete Notation

Die in dieser Arbeit verwendete Notation entspricht exakt der Eingabe eines Parsergenerators und orientiert sich an der BNF. Zur späteren Implementierung wird ANTLR verwendet, jedoch wird diesem das Werkzeug Xtext vorgeschaltet (siehe Abschnitt 3.7.2). Die ANTLR-Grammatik wird vollständig aus der Xtext-Grammatik generiert. Deren Notation ist ebenfalls noch recht komplex, und wird in der vollständigen Form daher nur in Anhang B benutzt. Eine Beschreibung der Xtext-Notation findet sich in [Onl/XtG]. In Kapitel 6 wird hingegen eine vereinfachte Version verwendet, deren Kenntnis für das Verständnis dieser Arbeit ausreichend ist.

Typ-2 Grammatiken müssen auf der linken Seite jeder Regel genau ein Nichtterminalzeichen besitzen. Nichtterminalzeichen werden jetzt durch einen Bezeichner dargestellt (Nichtterminal). Das hat zwei Vorteile: erstens ist man damit nicht mehr auf eine beschränkte Menge von Zeichen angewiesen, zweitens erhalten Regeln einen ausdrucksstarken Namen. Der Bezeichner wird durch einen Doppelpunkt von der rechten Seite der Regel getrennt.

Damit Nichtterminale von Terminalzeichen unterscheidbar sind, werden letztere in Anführungszeichen eingeschlossen. Auf diese Weise können außerdem bestimmte Folgen von Terminalzeichen (z.B. Schlüsselwörter) direkt zusammengefasst werden. Eine solche von Anführungszeichen eingeschlossene Zeichenfolge heißt Terminal. Vor und nach Terminalen werden beliebig viele Leerzeichen akzeptiert, wozu auch Zeilenumbrüche zählen.

In Regeln werden Alternativen und Wiederholungen durch eine Reihe von Operatoren leicht nachvollziehbar dargestellt. Die ver-

8 ANother Tool for Language Recognition [Par13].

fügbaren Operatoren sind in Tabelle 1 aufgelistet. Runde Klammern erlauben dabei die Bildung von Gruppen.

Tabelle 1: Operatoren der Grammatik-Notation

Operator	Bedeutung
`?`	Keinmal oder einmal
`*`	Keinmal oder mehr
`+`	Einmal oder mehr
`!`	Negation
`|`	Alternative
`..`	Bereichsnotation für mehrere Alternativen (nur zwischen ausgewählten Terminalzeichen möglich)

Mehr ist nicht nötig, um die Syntax einer Typ-2 Sprache kompakt zu beschreiben. Im Folgenden dazu ein Beispiel anhand einer Regel, die beliebige ganze Zahlen akzeptiert:

```
1 GanzeZahl:
2     ('-')? ('0'..'9')+
```

Ein Regel, die darauf aufbauend einfache Ausdrücke der Grundrechenarten wie „`123 * -5`“ akzeptiert, könnte dann beispielsweise so aussehen:

```
1 EinfacherAusdruck:
2     GanzeZahl ( '+' | '-' | '*' | '/' ) GanzeZahl
```

Durch diese Notation ist es jedoch nicht möglich, Querbeziehungen zwischen den Elementen der beschriebenen Sprache auszudrücken. Eine solche Querbeziehung besteht beispielsweise zwischen dem Bezeichner einer Variablen in einem Ausdruck und der zugehörigen Deklaration. Diese Querbeziehung enthält eine wertvolle Information, die beim Verständnis der Sprache hilft: der Bezeichner, sonst nicht mehr als eine Zeichenkette, besitzt jetzt einen Typ – er *ist* eine Variable.
Diese Informationen soll in der Grammatik hinterlegt werden. Xtext bietet dafür einen geeigneten Mechanismus, der in die vereinfachte Notation übernommen wird.

Darstellung von Querbeziehungen

Durch eckige Klammern können Querverweise ausgedrückt werden. Innerhalb dieser Klammern werden zwei Nichtterminale erwartet, die durch „`|`“ voneinander getrennt sind. Das erste Nichtterminal bestimmt den Typ des Querverweises, das zweite Nichtter-

minal die Syntax. Das wird anhand des zuvor genannten Beispiels und den folgenden zwei Regeln veranschaulicht:

```
1 VariablenDeklaration:
2     'var' Bezeichner ';'
3
4 Ausdruck:
5     ( GanzeZahl | [VariablenDeklaration|Bezeichner] )
6     ( '+' | '-' | '*' | '/' )
7     ( GanzeZahl | [VariablenDeklaration|Bezeichner] )
```

Angenommen „`foo`“ ist ein gültiger Bezeichner. Dann ist „`foo * -5`“ ein syntaktisch korrekter Ausdruck. Tatsächlich akzeptiert wird der Ausdruck aber nur, wenn eine Variable mit dem Bezeichner „`foo`“ deklariert ist. Die Syntax der Deklaration ist „`var foo;`“.
Dieser neue, zusätzliche Schritt zur Akzeptanzprüfung gehört nicht mehr zum Wortproblem. Auch wird der Schritt nicht durch den Parser durchgeführt, sondern erst nach dem Parsen durch den sogenannten Linker. Die Tatsache, dass Querverweise bereits in der Grammatik beschrieben sind, darf davon nicht ablenken. Auch ist die beschriebene Notation im Allgemeinen nicht ausreichend, um den Linker automatisch zu generieren (vgl. Parsergenerator), da für das Linken der weitere Kontext eine Rolle spielt.

2.4 MODELLGETRIEBENE SOFTWAREENTWICKLUNG

Eine gute Definition für modellgetriebene Softwareentwicklung (MDSD) findet sich im gleichnamigen Buch von Thomas Stahl, Markus Völter, Sven Efftinge und Arno Haase:

> Modellgetriebene Softwareentwicklung [...] ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.
>
> — *Thomas Stahl, Markus Völter, Sven Efftinge und Arno Haase* [SVE+07]

Die Definition muss genauer erklärt werden. Am Anfang stehen formale Modelle. Formal bedeutet, dass die Modelle nach exakt spezifizierten Regeln erstellt werden, wozu später noch der Begriff „Metamodell“ eingeführt wird. Die Modelle beschreiben bestimmte Aspekte der Software (beispielsweise Verhalten oder Struktur), und zwar auf einer abstrakteren Ebene, als es mit einer Standardprogrammiersprache möglich wäre. Der folgende Schritt, die Erzeugung lauffähiger Software, kann auf zwei Weisen geschehen:

erstens durch die Generierung von Quelltext oder zweitens durch Interpretation des Modells. In dieser Arbeit ist nur die erste Variante relevant. Schließlich ist der Vorgang automatisiert. In diesem Zusammenhang ist damit gemeint, dass der Prozess wiederkehrend ist, und das Modell tatsächlich Teile vom Quelltext ersetzt.

Modellbasierte Entwicklung

Im Umfeld eingebetteter Systeme spielt eine spezielle Form von MDSD eine wichtige Rolle, die zur besseren Differenzierung als *modellbasierte Entwicklung* bezeichnet wird. Beispiele für entsprechende Technologien sind Simulink von The MathWorks, TargetLink von dSPACE, LabVIEW von National Instruments oder ASCET von ETAS, die vorwiegend für den Entwurf von Steuerungs- und Regelsystemen eingesetzt werden.

Der Unterschied modellbasierter Entwicklung gegenüber MDSD im Allgemeinen ist, dass dem Anwender nur eine Modellebene zur Verfügung steht, für die bereits ein domänenspezifisches Metamodell vorgegeben ist. Dadurch ist die Verwendung nur im Rahmen der zugehörigen Domäne sinnvoll (im Fall der zuvor genannten Beispiele sind das Steuer- und Regelanwendungen). Bei MDSD ist demgegenüber die Erstellung von Metamodellen ein zentraler Bestandteil der Softwareentwicklung. Dieser Unterschied ist für das Verständnis wichtig.

Zur Veranschaulichung des Unterschieds seien an dieser Stelle einige Beispiele für allgemeine MDSD Technologien genannt, ohne auf weitere Details einzugehen: die Standards der Object Management Group (OMG), Meta Object Facility (MOF) und Unified Modeling Language (UML), das für diese Arbeit genutzte Eclipse Modeling Framework (EMF), sowie die Extensible Markup Language (XML) im Zusammenhang mit XML Schemas (XSD).

2.4.1 *Metamodellierung*

Modelle enthalten zunächst lediglich Information, die, egal um was für eine Information es sich handelt, nach klaren Regeln erfasst werden muss, damit ihre Bedeutung nicht verloren geht. Diese Regeln stehen sozusagen „über“ dem Modell, was zwei zueinander in Beziehung stehenden Ebenen entspricht: eine für das Modell und eine für die zugehörigen Regeln. Man sagt, die Regeln liegen auf der *Meta-Ebene* zum Modell (der Vorsatz „meta“ kommt aus dem Griechischen und steht wörtlich für etwas „in der Rangfolge dahinter liegendes“).

Meta-Ebene

Prinzipiell reicht es aus, wenn die Regeln informell beschrieben oder sogar nur implizit bekannt sind. Zur automatisch Weiterverar-

beitung der Daten ist es aber nötig, die Regeln explizit zu machen. Bei MDSD geschieht das durch ein weiteres formales Modell, das *Metamodell*.[9]

Metamodell

Das Metamodell legt fest, wie zugehörige Modellinstanzen erstellt werden dürfen. Damit auch das Metamodell selbst formal beschrieben werden kann, bedarf es einer weiteren Ebene, auf welcher die Regeln für das Metamodell festgelegt sind. Dementsprechend gibt es ein Meta-Metamodell, und so weiter und so fort. In der Praxis werden Modelle ab einer bestimmten Ebene durch sich selbst beschrieben, sodass eine Hierarchie wie in Abbildung 8 typisch ist.

Selbstreferenzielles Modell

Das Kernmodell der MOF ist ein Beispiel für ein solches *selbstreferenzielles Modell*, welches die Anzahl der Modellebenen begrenzt. Prinzipiell gibt es jedoch keine obere Grenze für die Zahl der Ebenen. Auch liegt, wie mitunter zu lesen, das Minimum nicht bei vier, sondern bei lediglich zwei Ebenen. Diese sind nötig, um zwischen Instanzen und ihrer Beschreibung navigieren zu können. Die OMG nennt Beispiele mit zwei, drei und vier Ebenen [OMG15].

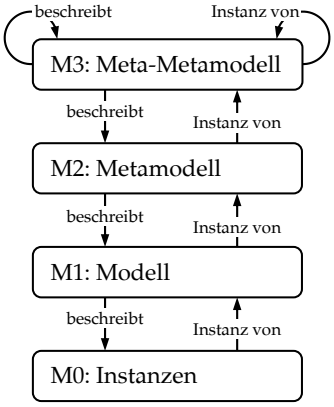


Abbildung 8: Meta-Ebenen der OMG nach [SVE+07]

Abbildung 9 zeigt, wie verschiedene MDSD Technologien den Meta-Ebenen in typischen Anwendungsfällen zugeordnet werden. Die Zuordnung ist – genauso wie bei den in Abbildung 8 gewählten Bezeichnungen – nicht absolut definiert, sondern ergibt sich immer erst in einem konkreten Zusammenhang.

9 Auch ohne explizites Metamodell sind die Regeln explizit, sobald das Modell automatisiert verarbeitet wird. Sie sind nur im Quelltext zur Modellverarbeitung „versteckt“.

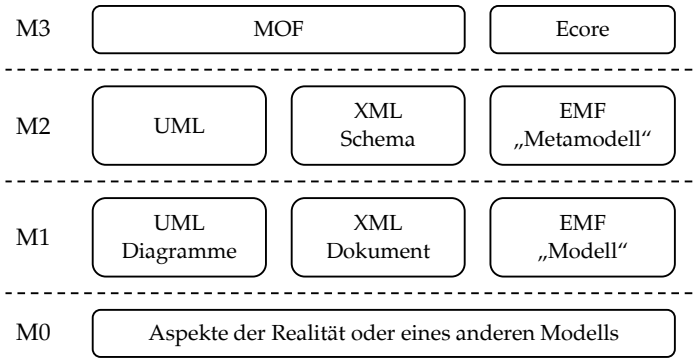


Abbildung 9: MDSD Technologien und ihre typischen Meta-Ebenen

2.4.2 *Domänenspezifische Sprachen*

Wenn im letzten Abschnitt von „Regeln zur Modellerstellung“ die Rede war, dann waren damit die *abstrakten* Elemente des Modells und ihre Zusammensetzung gemeint, und nicht, wie das Modell *konkret* zu notieren ist. Das entspricht der Unterscheidung zwischen abstrakter und konkreter Syntax, die bereits in Abschnitt 2.3 deutlich wurde. Bezieht sich die abstrakte Syntax nun auf eine bestimmte Domäne, wie es bei Modellen normalerweise der Fall ist, so spricht man von einer domänenspezifischen Sprache (DSL).
Im Abschnitt über die formalen Sprachen wurde bei konkreter Syntax von Zeichen und Wörtern („Text“) ausgegangen. Genauer gesagt ist die konkrete Syntax jedoch „die konkrete Ausprägung der textuellen oder *grafischen* Konstrukte, mit denen modelliert wird“ [SVE+07]. Grafische Syntax findet sich z.B. bei der UML und den Werkzeugen zur modellbasierten Entwicklung, welche weiter vorne erwähnt wurden.

Grafische Syntax

Zur formalen Definition grafischer Syntax ist kein mathematischer Formalismus bekannt. Es gibt aber bereits praktische Ansätze (sozusagen ein Gegenstück zu den BNF-Varianten), welche jedoch noch nicht breit eingesetzt werden. Beispiele sind das Graphical Modeling Projekt [Onl/GMP] oder das Microsoft Modeling SDKs [Onl/MSDK].

2.4.3 *Grenzen von MDSD*

Ein großer Vorteil von MDSD ist die Möglichkeit, bestimmte Aspekte auf einer abstrakteren Ebene zu betrachten, als es mit traditio-

nellen Programmiersprachen möglich ist. Real existierende Komplexität kann jedoch nicht einfach „wegabstrahiert“ werden; es ist lediglich möglich sie an eine andere Stelle zu verschieben. Im Fall von MDSD kommt dafür nur der Codegenerator (bzw. ein Interpreter) in Frage. Daraus ergibt sich eine wichtige Erkenntnis über den Entwicklungsaufwand: die auf der Modellebene durch Abstraktion gewonnene Effizienzsteigerung muss durch die Realisierung eines Codegenerators erkauft werden. Bei regelmäßigem Einsatz des Codegenerators ist jedoch zu erwarten, dass sich die initiale Investition im Laufe der Zeit amortisiert. Das lässt sich am Erfolg der Werkzeuge zur modellbasierten Entwicklung auch hervorragend beobachten.

Bei MDSD mit mehreren Modellebenen muss man allerdings berücksichtigen, dass eine Änderung des Metamodells auch eine Anpassung des Codegenerators erfordert! Diese Anpassung ist nur durch einen Menschen möglich, denn ein entsprechender Algorithmus müsste die Semantik des Metamodells tatsächlich *begreifen*. Das ist mit herkömmlicher Algorithmik nicht berechenbar [Hro11]. Die Erforschung künstlicher Intelligenz vermag hier neue Erkenntnisse zu liefern, aktuell ist jedoch kein Durchbruch absehbar.

Es ist deswegen nicht realistisch, bei der Entwicklung ausschließlich auf der Modellebene zu arbeiten. Die Integration verschiedenartiger Modelle anhand eines allgemeinen, aber doch abstrakten „Integrationsmodells“ kann nicht gelingen. Douglas C. Schmidt hat diesbezüglich bereits 2006 in einem vielzitierten Artikel zusammengefasst: „... these [recent] MDE[10] efforts recognize that models alone are insufficient to develop complex systems.“ [Sch06]. Bei MDSD erfolgt daher ein signifikanter Teil der Softwareentwicklung durch traditionelle Programmierung. Dieses Zusammenspiel ist nur effizient, wenn sich jedes Modell auf einen separaten Aspekt der Software konzentriert und aufwändige Anpassungen in Folge von Modelländerungen vermieden werden.

Abgrenzung zu CASE-Tools

Die bewusste Entscheidung, einen Großteil der Softwareentwicklung auf traditionelle Weise durchzuführen, ist ein wichtiger Unterschied gegenüber dem Ansatz früherer CASE[11]-Tools, nach welchem die Software möglichst *vollständig* aus abstrakten Modellen generiert werden sollte. Derartige CASE-Tools ließen sich aber nicht mit dem gewünschten Erfolg umsetzen [Sch06]. Noch einmal zur Klarstellung: innerhalb einer eng umgrenzten Domäne ist ein voll-

10 Schmidt verwendet für MDSD die Bezeichnung „Model Driven Engineering“ (MDE).

11 Computer-Aided Software Engineering

ständiger Automatismus möglich und auch effizient, z.B. Codegenerierung für Regelfunktionen mit den zuvor genannten Werkzeugen oder Erzeugung von Maschinencode durch einen Compiler. Die Herausforderung bezüglich komplexer Softwaresysteme besteht jedoch darin, Aspekte mit ganz unterschiedlichen Abstraktionsanforderungen zu verbinden.
MDSD verspricht dabei eine Verbesserung gegenüber rein traditioneller Programmierung. Aber auch die herkömmlichen Programmiersprachen entwickeln sich ständig weiter. Ein gutes Beispiel ist das Aufkommen objektorientierter Programmierung und der spätere Siegeszug darauf ausgerichteter Sprachen wie C++, C# und Java. Die Mächtigkeit dieser Sprachen, vor Allem durch Vererbung und Polymorphie, aber auch neuen Möglichkeiten zur generischen Programmierung, darf man nicht unterschätzen.
Im Prinzip können domänenspezifische Aspekte in objektorientierten Programmiersprachen bereits gut durch Klassenbibliotheken und Schnittstellen zur Anwendungsprogrammierung (APIs) abgebildet werden (sogenannte „interne“ DSLs basieren auf genau diesem Ansatz). Aber auch syntaktische Erweiterungen kommen in Frage. Dabei sind durchaus auch grafische Spracherweiterungen möglich – ein gutes Beispiel dafür findet sich in [VRK+12]. Der Vorteil gegenüber separaten Modellen liegt darin, dass die Notwendigkeit einer späteren Modellintegration schon konzeptbedingt entfällt.
Damit verschwimmen die Grenzen zwischen Programmierung und Modellierung. In Zukunft könnte es in einigen Fällen schwer fallen zu entscheiden, ob es sich um eine Programmiersprache handelt, die um domänenspezifische Aspekte erweitert wurde, oder es in Wirklichkeit nicht ein modellgetriebener Ansatz ist, bei dem die „Programmierung“ nur noch ein mögliches Vorgehen unter vielen darstellt.

2.4.4 *Modellbasiertes Testen*

Modellbasiertes Testen (MBT) ist viel weiter gefasst, als die Definition des Begriffs „modellbasierte Softwareentwicklung“ weiter vorne, der Vergleich führt also in die Irre. Thomas Roßner unterteilt MBT in modellorientierten, modellgetriebenen und modellzentrischen Test:

> - *Modellorientiertes Testen*: (Grafische) Modelle werden primär zur Kommunikation und zur Dokumentation genutzt. Durch die Nutzung werden Missverständnisse vermieden, frühzeitig Fehler gefunden und Grundlagen für einen systematischen Testfallentwurf geschaffen.
> - *Modellgetriebenes Testen*: (Formale) Modelle dienen zur (teilweisen) werkzeuggestützten Generierung von Testartefakten. Die Generierung ermöglicht die Erhöhung der Testabdeckung und/oder die Senkung des Aufwands für die Testfallerstellung und -wartung.
> - *Modellzentrisches Testen*: Modelle dienen nicht nur als Quelle zur Generierung von Testartefakten, sondern auch als Ziel für die Ablage von abgeleiteten Informationen. Alle wesentlichen Aktivitäten des Testprozesses arbeiten auf der Basis von Modellen. Durch die gemeinsame Arbeitsgrundlage für alle Beteiligten und den Wegfall von Medienbrüchen entstehen zusätzliche Transparenz und Effizienz.
>
> — *Thomas Roßner, Christian Brandes, Helmut Götz und Mario Winter [RBG+10]*

Diese Definition entspricht einer Rangfolge, wie stark Modelle in den Testprozess involviert sind. Auf der letzten Stufe werden dazu zwei Formen der Modellnutzung verknüpft: die Verwendung von Modellen als Grundlage zur Generierung von Testartefakten (modellgetriebenes Testen) mit der Verwendung von Modellen als allgemeine „Informationsablage“, was alles Mögliche meinen kann.[12] Das sind zwei unterschiedliche Angelegenheiten, die zwar eine Beziehung zueinander eingehen können, grundsätzlich jedoch unabhängig voneinander zu betrachten sind. Später in dieser Arbeit werden sowohl Modelle vorgestellt, die der „Informationsablage“ die-

12 z.B. die Rückverfolgbarkeit von Anforderungen (Traceability) oder die Speicherung und der Austausch von Testfällen, Testdaten, Werkzeugkonfigurationen, Eigenschaften des Testobjekts etc.

nen, als auch andere Modelle, aus denen Testartefakte generiert werden. Letzteres muss noch etwas genauer diskutiert werden.

Modellgetriebenes Testen

Modellgetriebenes Testen entspricht der Idee, MDSD auf die Testspezifikation zu übertragen. Es geht also um eine ähnliche Fragestellung: Inwiefern lassen sich Testfälle, Testdaten oder die Testimplementierung automatisch generieren?

Wie bei MDSD kann man mehrere Modellebenen unterscheiden. Dabei offenbaren sich zwei Unterarten des modellgetriebenen Testens: die erste Unterart beschäftigt sich mit der Generierung von Testfällen, z.B. durch die Bildung von Äquivalenzklassen [OB88] oder anhand von System- und Verhaltensmodellen. Es wird die Frage beantwortet: „*Was* soll getestet werden?“. Die zweite Unterart beschäftigt sich mit der Generierung der Testimplementierung anhand von Modellen des Testablaufs und der Testdaten. Die zugehörige Frage lautet „*Wie* wird der Test ausgeführt und bewertet?“.

Ansätze, die für sich beanspruchen beide Fragen *gleichzeitig* zu beantworten, müssen vorsichtig betrachtet werden. Die Gründe sind keine Anderen als die, welche in Abschnitt 2.4.3 bereits zu dem Schluss geführt haben, dass MDSD nicht effizient zur Generierung *vollständiger* Software verwendet werden kann, sondern jedes Modell nur für einen Teil bzw. Aspekt geeignet ist.

Für das modellgetriebene Testen ist also zu erwarten, dass vollautomatische Prozesse zwar möglich sind, diese aber nur Teilergebnisse erzielen. Ein Beispiel: mit Hilfe statischer Codeanalyse ist es nicht schwer, einige Funktionsaufrufe auszuwählen, welche die Codeabdeckung maximieren. Diese „Testfälle“ sind auch ohne manuellen Eingriff direkt ausführbar. Findet dieser Ansatz relevante Fehler?

Ja, das tut er. So können eine Division durch Null, unbehandelte Ausnahmen und das Erreichen anderer eindeutiger Fehlerzustände zuverlässig erkannt werden. Allein die bloße Tatsache, dass eine große Anzahl möglicher Programmpfade „ohne Absturz“ durchlaufen wird, schafft Vertrauen in die Implementierung. Aber man wird auch eine große Zahl irrtümlich als Fehler klassifizierte Befunde („false positive“) erhalten, schließlich wird in der Realität längst nicht jeder Programmpfad tatsächlich durchlaufen. Das Auffinden der tatsächlich relevanten Programmpfade ist mit herkömmlichen Methoden nicht berechenbar, denn auch dazu müsste der Algorithmus die Semantik des Programms *begreifen* (vgl. Abschnitt 2.4.3).

Gleichzeitig bleiben viele Fehler unentdeckt. Das größte Problem im konstruierten Beispiel ist, dass die Eingabe für den Testfallgenerator mit dem Testobjekt übereinstimmt. In der Eingabe sind deswegen genau die logischen Fehler enthalten, die eigentlich gefunden werden sollen. Verwendet man hingegen ein unabhängig

erstelltes Modell, welches das gewünschte Verhalten der Software beschreibt, lassen sich (hoffentlich) bessere Testfälle generieren. Ein solches Modell muss jedoch per Definition abstrakt sein. Die Details, welche zur automatischen Ausführung benötigt werden, lassen sich nur dann durch einen Algorithmus berechnen, wenn dabei Annahmen über das Testobjekt gemacht werden, was im Allgemeinen natürlich nicht zulässig ist.
In der Praxis erscheint daher nur eine getrennte Betrachtung zielführend: entweder werden Testfälle generiert, und auf die sofortige automatische Ausführung verzichtet, oder es werden einzelne Testfälle modelliert, die dafür sofort automatisch ausführbar sind. Die in dieser Arbeit vorgestellte Lösung fokussiert sich auf Letztgenanntes. Natürlich lassen sich die beiden Schritte hintereinander schalten, dazwischen ist aber immer ein irgendwie gearteter Eingriff erforderlich. Im einfachsten Fall handelt es sich dabei um manuelles „Nachimplementieren“ der generierten Spezifikation, aber auch die Entwicklung einer spezifischen Ausführungsumgebung, welche die notwendigen Details „verinnerlicht“ hat, stellt einen solchen Eingriff dar.

3

STAND DER TECHNIK

Dieses Kapitel gibt einen Überblick über den aktuellen Stand der Technik, der für diese Arbeit relevant ist. In Abschnitt 3.1 wird zunächst zwischen den grundsätzlich möglichen Ausführungsplattformen differenziert, welche im Rahmen der Testautomatisierung für eingebettete Software eingesetzt werden können. Anschließend werden in Abschnitt 3.2 die Alternativen für den Zugriff auf Testdaten erläutert. Die Kenntnis dieser Zugriffstechnologien ist erforderlich für das Verständnis der zwei folgenden Abschnitte, welche sich mit konkreten Lösungen zur Testautomatisierung befassen. Hier wird zwischen den bereits zu Beginn erwähnten Ebenen unterschieden: Abschnitt 3.3 befasst sich mit Tests auf der Systemebene, während Abschnitt 3.4 die Lösungen für Tests auf der Quelltextebene vorstellt.
In beiden Fällen spielt die Implementierung von Testfällen eine zentrale Rolle. In Abschnitt 3.5 wird auf die jeweiligen Programmiersprachen genauer eingegangen. Anschließend werden die verfügbaren Technologien in Abschnitt 3.6 zusammenfassend gegenübergestellt. In Abschnitt 3.7 am Ende des Kapitels wird noch kurz auf die verwendeten MDSD Technologien für die Eclipse IDE eingegangen.

3.1 PLATTFORMEN ZUR TESTAUSFÜHRUNG

Testplattform und Zielplattform

Die Plattform, auf welcher die zu testende Software während der Testdurchführung läuft, wird als *Testplattform* bezeichnet. Dazu bieten sich verschiedene Alternativen an. Naheliegend ist es, die Software auf der Plattform zu testen, die im späteren Endprodukt tatsächlich verwendet wird. Diese wird als *Zielplattform* bezeichnet.
Ist die Testplattform mit der Zielplattform identisch, kann ausgeschlossen werden, dass sich im späteren Einsatz plattformspezifische Fehler auswirken, die auf einer abweichenden Testplattform nicht zu entdecken waren. Die ISO 26262, aktuelle Norm zur Absicherung sicherheitsrelevanter Funktionen in Kraftfahrzeugen, ist dazu sehr deutlich:

> The test environment for software unit testing shall correspond as closely as possible to the target environment. If the software unit testing is not carried out in the target environment, the differences in the source and object code, and the differences between the test environment and the target environment, shall be analysed in order to specify additional tests in the target environment during the subsequent test phases.

— ISO 26262-6 Abs. 9.4.6 [Std/ISO11a]

Tests auf der Zielplattform sind also insbesondere für sicherheitskritische Software wichtig, und diese Arbeit konzentriert sich darauf. Dennoch sollen zur besseren Einordnung die Alternativen kurz erläutert werden, welche durch die Verwendung von Emulatoren (Abschnitt 3.1.1) oder durch Simulation (Abschnitt 3.1.2) gegeben sind. Darüber hinaus ist es üblich, dass zur frühzeitigen Erprobung der Anwendungsfunktion Prototypen der Zielplattform in unterschiedlichen Reifegraden genutzt werden (Musterstände, siehe Abschnitt 3.1.3), welche damit eine weitere Kategorie von Testplattformen darstellen.

3.1.1 *Emulator*

Emulatoren können die Zielplattform bei der Entwicklung und Erprobung von eingebetteter Software ersetzen. Der Emulator muss dazu das Verhalten der Zielplattform genau nachbilden, sodass sowohl aus Sicht der Umgebung als auch aus Sicht der ausgeführten Software kein Unterschied wahrgenommen wird.

In-Circuit-Emulator

In Zusammenhang mit eingebetteten Systemen ist mit dem Begriff Emulator in der Regel ein *In-Circuit-Emulator* (ICE) gemeint. In diesem Fall umfasst der Begriff die Prozessorhardware und nicht die übrigen Komponenten eines eingebetteten Systems. In-Circuit-Emulatoren basieren auf speziell gefertigten Hardwarebausteinen oder FPGAs[1]. Die wesentliche Motivation zur Realisierung von ICEs ist die Schaffung besserer Analysemöglichkeit des Softwareablaufs und dessen Kontrollierbarkeit zwecks Debugging. Bei älteren Prozessoren basierten ICEs üblicherweise auf speziellen Varianten der Prozessorhardware, mit zusätzliche nach außen geführten Leitungen, um den internen Zustand zu beobachten (Bond-Out Chip).

1 Field Programmable Gate Array

Da sich ein ICE von der nachgebildeten Prozessorhardware technologisch unterscheidet, wird es in der Praxis fast immer leichte Abweichungen im Verhalten geben. Am ehesten betroffen sind das Zeitverhalten, beispielsweise bezüglich Speicherzugriffen, hervorgerufen durch Ungenauigkeiten bei der Nachbildung der Prozessorarchitektur (z.B. Pipeline oder Cache und entsprechende Latenzen), sowie der Stromverbrauch.
Aufgrund der hohen Vielfältigkeit und steigenden Performanz der Prozessorhardware wird es immer schwieriger, geeignete ICEs anzubieten. Die erforderlichen hohen Taktraten sind nur mit teuren Fertigungsverfahren zu realisieren, welche sich bei den für Emulatoren typischen Stückzahlen nicht rentieren. Als Ausgleich bietet heutige Prozessorhardware entsprechende Debug- und Trace-Schnittstellen, sodass bei der Entwicklung auf ICEs zunehmend verzichtet werden kann. In diesem Fall benötigt man zur Verbindung der Prozessorhardware mit einem Host-PC jedoch weiterhin spezielle Werkzeuge. In der Anfangszeit wurden sie weiterhin als „ICE“ vermarktet, was eine Fehlbezeichnung ist, denn eine Emulation findet nicht mehr statt und der Ansatz darf nicht mit Verwendung eines Bond-Out Chips verwechselt werden. Heute sind verschiedene Bezeichnungen gebräuchlich, z.B. In-Circuit-Debugger, In-System-Debugger und On-Chip-Debugger. In dieser Arbeit werden die Begriffe Debug- und Trace-Hardware bzw. kurz Debugger verwendet (mehr zu Debuggern findet sich in Abschnitt 3.2.3).

3.1.2 *Simulator*

Simulatoren versuchen ebenfalls das Verhalten der Zielplattform möglichst originalgetreu nachzubilden. Der Unterschied ist, dass die Simulation von außen betrachtet gänzlich anders als in der Realität ablaufen kann, beispielsweise deutlich schneller oder verlangsamt. Somit kann ein Simulator im Allgemeinen nicht in die reale Umgebung eingebettet werden, sondern es muss diese ebenfalls simuliert werden.
Die Simulation ermöglicht eine äußerst detaillierte Analyse des Verhaltens, beispielsweise kann der Ablauf jederzeit ohne Nebenwirkungen pausiert werden. Es sind jedoch äußerst präzise Modelle der Prozessorhardware erforderlich, welche in hinreichender Genauigkeit nicht immer verfügbar sind, da viele Details, beispielsweise zum Zeitverhalten, von den Chipherstellern nicht veröffentlicht werden.

Genauso wie bei einem Emulator kann bei einem Simulator die Exaktheit der Verhaltensnachbildung in der Praxis kaum garantiert werden. Wie bereits zuvor gesagt, ist daher für Tests im Rahmen der Absicherung kritischer Systeme die Ausführung auf der Zielplattform entscheidend.

3.1.3 *Musterstände*

In einigen Branchen gibt es den Begriff Musterstände für unterschiedliche Reifegrade der Zielplattform. Musterstände sind insbesondere im Kontext verteilter Entwicklung relevant, da entsprechende Prototypen dort zur frühzeitigen Erprobung des Gesamtsystems durch den Integrator erforderlich sind.
In der Automobilindustrie wird zwischen A, B, C und D-Muster unterschieden [Saxo8]. Das A-Muster ist demnach recht gut mit einem Emulator vergleichbar, wobei hier das gesamte Steuergerät und nicht nur die Prozessorhardware emuliert wird. Ab dem B-Muster wird eine weitgehend der Zielplattform entsprechende Hardwareplattform verwendet, wobei diese noch nicht unter Serienbedingungen gefertigt wird (z.B. Leitungsrouting, Kontaktierung). Auf mechanischer Ebene (z.B. Gehäuseform) entspricht das B-Muster bereits weitgehend der Zielplattform. Das C-Muster unterscheidet sich nur noch bezüglich der verwendeten Fertigungswerkzeuge von der Zielplattform beziehungsweise dem D-Muster.
Darüber hinaus unterscheiden sich die Musterstände bezüglich der in Software implementierten Anwendungsfunktionen, was im Kontext der Betrachtung von Musterständen als Testplattform nur insofern relevant ist, als dass sich natürlich nur die Funktionen testen lassen, die auch implementiert wurden. Bezüglich den Eigenschaften Beobachtbarkeit und Kontrollierbarkeit des Softwareablaufs können B-, C- und D-Muster als gleichwertig betrachtet werden.

3.2 ZUGRIFF AUF DAS TESTOBJEKT

Testgröße

Im weiteren Verlauf dieser Arbeit taucht ein Begriff häufiger auf, der im Rahmen dieser Arbeit eine spezielle Bedeutung hat: *Testgröße*. Es handelt sich dabei um einen Oberbegriff für die atomaren Einheiten, anhand denen Testfälle spezifiziert werden. Beispiele für Testgrößen finden sich in Tabelle 2. Es ist eine wesentliche Aufgabe eines Testautomatisierungssystems, Zugriff auf diese Testgrößen bereitzustellen.

Tabelle 2: Beispiele für Testgrößen mit spezifischen Eigenschaften

	Testgröße	Eigenschaften
intern	Variable	Logischer Wert
	Funktion	Rückgabewert, Ausführungszeit
extern	Signal	Elektrischer Wert, zeitlicher Verlauf
	Nachricht	Nachrichteninhalt, Latenz, Zykluszeit

In der Tabelle wird zwischen internen und externen Testgrößen unterschieden. Intern bedeutet in diesem Zusammenhang, dass die Testgröße Teil der auf der Zielplattform ausgeführten Software ist. Extern bedeutet das entsprechende Gegenteil – meint also Testgrößen, welche kein Teil der ausgeführten Software sind. Die teils sehr unterschiedlichen Eigenschaften der Testgrößen sind ebenfalls in der Tabelle dargestellt.

Zugriffspunkt

Für den Zugriff auf Testgrößen sind geeignete *Zugriffspunkte* erforderlich. Ein Zugriffspunkt ist die Schnittstelle für den Zugriff auf Testgrößen (vgl. auch „Zugangspunkt“ in [Saxo8]). Gegebenenfalls müssen Zugriffspunkte künstlich erzeugt werden, um Testgrößen wie gewünscht verfügbar zu machen. Das entspricht einer Modifikation des Testobjekts zu Testzwecken. In diesem Fall wird von indirektem Zugriff gesprochen, andernfalls von direktem Zugriff. Außerdem kann zwischen internem und externem Zugriff unterschieden werden, was analog zu der Definition oben zu verstehen ist: interner Zugriff erfolgt aus der auf der Zielplattform ausgeführten Software heraus, externer Zugriff geht von einem Hostrechner aus.

3.2.1 *Direkter externer Zugriff*

Auf einen Teil der Testgrößen lässt sich über externe Schnittstellen des eingebetteten Systems direkt zugreifen. Externe Schnittstellen lassen sich in die folgenden Kategorien einteilen:

- *Elektrische Schnittstellen*: Einfache elektrische Pins für digitale oder analoge Signale.
- *Netzwerkschnittstellen*: Bus- und Netzwerkschnittstellen für protokollbasierte Übertragung von Nachrichten.

- *Mechanische Schnittstellen*: Sensoren und Aktoren für Größen wie Beschleunigung, Drehzahl oder Druck.
- *Optische Schnittstellen*: Sensoren und Aktoren für Größen wie Kamera- oder Radarbilder.

Testgrößen, welche an den mechanischen und optischen Schnittstellen verfügbar sind, werden durch die jeweiligen Sensoren oder Aktoren des eingebetteten Systems unmittelbar in Größen einer der übrigen Kategorien überführt oder daraus erzeugt. Weil das unabhängig von der im Rahmen dieser Arbeit betrachteten Software erfolgt, kann dieser Schritt auch unabhängig davon getestet werden. Aus diesem Grund werden in dieser Arbeit bezüglich externer Schnittstellen nur Elektrische und Netzwerkschnittstellen betrachtet.
Der direkte externe Zugriff auf Signale ist nur möglich, wenn entsprechende Pins physisch erreichbar sind. Das kann z.B. durch das Gehäuse des Geräts verhindert werden. Auf Nachrichten kann von extern fast immer direkt zugegriffen werden, da entsprechende Zugriffspunkte (Netzwerkschnittstellen) erreichbar sein müssen, um ihren grundlegenden Zweck, nämlich die Kommunikation mit anderen Geräten, erfüllen zu können.[2] Auf interne Testgrößen kann von extern nicht direkt zugegriffen werden.

Zugriffshardware

Der Zugriff auf externe Schnittstellen des eingebetteten Systems erfordert spezielle *Zugriffshardware*. Deren Ausprägung hängt stark vom jeweiligen Zugriffspunkt ab, der zur Verbindung mit dem Testobjekt genutzt wird. Für den Zugriff auf einen elektrischen Pin kommt beispielsweise ein Oszilloskop oder Signalgenerator in Frage, für Netzwerkschnittstellen sogenannte Datenlogger. Für automatisierte Tests im größeren Stil wird individuell gefertigte Zugriffshardware zu einem komplexen Testsystem integriert (siehe Abschnitt 3.3.1).

3.2.2 *Direkter und indirekter interner Zugriff*

Auf interne Testgrößen kann von außerhalb der Software nicht direkt zugegriffen werden. Ist zu Testzwecken jedoch direkter Zugriff gewünscht, wie es beispielsweise beim Unit-Test der Fall ist (siehe

2 Ausgenommen sind Netzwerkschnittstellen, auf welche der Zugriff bewusst eingeschränkt wird, beispielsweise durch Verschlüsselung oder durch das Verbergen der Schnittstelle im Gehäuse. Das kann vom Gerätehersteller z.B. zum Schutz des geistigen Eigentums gewünscht sein. In diesem Fall ist der externen Zugriff auf entsprechende Nachrichten für andere Parteien nicht möglich.

Abschnitt 3.4), müssen Testfälle auf Quelltextebene implementiert und mit der zu testenden Software gelinkt werden. Anschließend werden Test- und Softwarecode gemeinsam auf der Zielplattform ausgeführt.[3] Dieses Vorgehen wird in Abschnitt 3.4 genauer erklärt. Auch intern kann nicht immer direkt auf interne Testgröße zugegriffen werden. Ein gutes Beispiel sind geschützte Klassenmitglieder, also Funktionen oder Variablen einer Klasse, die mit dem Schlüsselwort `private` oder `protected` versehen sind.

Codeinstrumentierung

In diesem Fall kann *Codeinstrumentierung* eingesetzt werden, um einen Zugriffspunkt zu schaffen (z.B. `get` oder `set`-Methode). Codeinstrumentierung bezeichnet die Modifikation oder Erweiterung der bestehenden Softwarequelltexte, damit die Software einfacher oder überhaupt getestet werden kann. Neben dem Hinzufügen zusätzlicher Schnittstellen für den Zugriff auf geschützte Klassenmitglieder sind das Einfügen von Codefragmenten am Beginn und Ende einer Funktion zur Zeitmessung der Ausführung oder Zählen der Aufrufe weitere Beispiele.
Von intern ist der Zugriff auf externe Testgrößen nicht sinnvoll.

3.2.3 *Indirekter externer Zugriff*

Nicht immer bietet sich interner Zugriff an. Es muss dann ein Zugriffspunkt für interne Testgrößen geschaffen werden, der von außen erreichbar ist. Das entspricht indirektem externen Zugriff. Dazu bieten sich grundsätzlich zwei Möglichkeiten an, die sich darin unterscheiden, ob die Realisierung in Hardware oder in Software erfolgt. Im nächsten Abschnitt wird zunächst kurz auf die Möglichkeit einer Softwarelösung eingegangen, bevor der für diese Arbeit wesentliche Ansatz durch spezielle Hardwareunterstützung beschrieben wird.

Nutzung spezieller Softwaremodule

XCP

Interne Testgrößen können durch zusätzliche Softwaremodule (Diagnosemodule) über Netzwerkschnittstellen des eingebetteten Systems verfügbar gemacht werden. Dabei müssen geeignete Kommunikationsprotokolle eingesetzt werden. Ein weit verbreiteter Standard für Steuergeräte in Kraftfahrzeugen ist das „ASAM Universal Measurement and Calibration Protocol“ (XCP) [Std/ASA15a]. XCP

3 In [Haro1] wird der Begriff „Testmaschine“ für die Ausführungsplattform der Testimplementierung definiert. Hier gilt nun der Sonderfall, dass Testmaschine und Zielplattform identisch sind.

ermöglicht das Auslesen und Schreiben des Speichers der Zielplattform und kann damit für den Zugriff auf Variablen genutzt werden. Dabei ist sowohl zyklische als auch Ereignis-getriebene Datenübertragung möglich, was auch dynamisch konfiguriert werden kann. Außerdem kann durch definierte Kommandos spontaner Zugriff realisiert werden. Der Aufruf von Funktionen und die damit verbundene Übertragung der Funktionsparameter und Rückgabewerte sind durch XCP jedoch nicht spezifiziert. Der Standard bietet aber eine Möglichkeit zur Ergänzung benutzerdefinierter Kommandos, was für den Zugriff auf Funktionen genutzt werden könnte.
Weil Diagnosemodule zusammen mit der übrigen Software auf der Zielplattform ausgeführt werden, beanspruchen sie einen Teil der Rechenleistung für sich. Zudem ergibt sich eine erhöhte Auslastung des Kommunikationsnetzwerks, falls die Daten über die gleichen Schnittstellen übertragen werden müssen, welche auch von der Anwendungsfunktion genutzt werden. Der Ansatz kommt daher nur in Frage, wenn ausreichende Reserven bestehen und der Einfluss auf die Testergebnisse kein Problem darstellt.

Nutzung von Debug- und Trace-Hardware

Eine andere Möglichkeit für den Zugriff auf interne Testgrößen erfordert Unterstützung durch die Prozessorhardware. Nahezu jeder Mikrocontroller besitzt eine dedizierte Debug-Schnittstelle, welche das Debuggen der Software von einem entfernten Entwicklungsrechner aus und ohne die Notwendigkeit spezieller Softwaremodule auf der Zielplattform ermöglicht. Dazu werden die grundlegenden Debug-Features angeboten, d.h. Starten und Stoppen der Ausführung, das Setzen von Breakpoints und Watchpoints, Stepping sowie direkter Speicherzugriff.
Sehr häufig erfolgt die Realisierung auf Basis des „IEEE Standard for Test Access Port and Boundary-Scan Architecture“ (JTAG) [Std/IEE13]. Von diesem Standard ist insbesondere der Test Access Port relevant, welcher eine serielle Schnittstelle für den Zugriff auf Register des Prozessors spezifiziert.[4]
Bezüglich der grundlegenden Debug-Features gibt es je nach Prozessorarchitektur Einschränkungen. So ist beispielsweise mit der ARM CoreSight Debug-Architektur der *Echtzeit-Zugriff* auf Register und Speicher über die JTAG-Schnittstelle möglich [ARM09].

Echtzeit-Zugriff

4 JTAG wird in dieser Arbeit als Synonym für den IEEE Standard verwendet, und meint nicht das ursprünglich für den Standard verantwortliche Industriekonsortium Joint Test Action Group. Der Test Access Port wird dementsprechend mit JTAG-Schnittstelle bezeichnet.

Echtzeitzugriff bedeutet in diesem Zusammenhang, dass die Softwareausführung dazu nicht gestoppt werden muss. Das ist wichtig für echtzeitkritische Software, bei der das Anhalten der Ausführung nicht in Frage kommt, wird aber nicht bei jedem Mikrocontroller unterstützt.
Neben den zuvor genannten grundlegenden Debug-Features wird darüber hinaus häufig die Beobachtung des Programmflusses gewünscht (Program-Trace). Ein typisches Anwendungsszenario ist die Ermittlung der Codeabdeckung. Das kann mit Hilfe schrittweiser Ausführung geschehen, was aus dem zuvor genannten Grund bei echtzeitkritischer Software jedoch nicht in Frage kommt. Eine unterbrechungsfreie Ausführung ist in diesem Fall aber nicht mit der JTAG-Schnittstelle realisierbar, weil sie nicht die erforderliche Bandbreite bietet. Immer mehr Mikrocontroller besitzen für diesen Zweck jedoch eine dedizierte Trace-Schnittstelle.
Ist eine solche Schnittstelle verfügbar, kann darüber meistens auch der jeweils laufende Threads eines Betriebssystems verfolgt werden (Ownership-Trace). Dazu muss das Betriebssystem beim Threadwechsel ein spezielles Register mit der entsprechenden Thread ID beschreiben. Diese Information ist wichtig, um Speicheradressen von Variablen über die Symboltabelle des Betriebssystems auflösen zu können [IEE04]. Zuletzt ist vermehrt sogar die Beobachtung aller Speicherzugriffe durch die ausgeführte Software möglich (Daten-Trace).
Die Aufzeichnung der an einer dedizierten Trace-Schnittstelle verfügbaren Daten wird als *Echtzeit-Tracing* bezeichnet. Durch Echtzeit-Tracing werden Anwendungsfälle unterstützt, welche über die Ermittlung der Codeabdeckung weit hinausgehen. Beispielsweise ist eine detaillierte Analyse des Zeitverhaltens anhand von Zeitstempeln in den aufgezeichneten Daten möglich.

Echtzeit-Tracing

Wegen der stark unterschiedlichen Realisierung und daher auch Verwendung entsprechender Debug- und Trace-Funktionalität wurde der „IEEE-ISTO Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface“ [Std/IEE12] ins Leben gerufen. Dieser teilt Prozessorhardware in vier Klassen ein, welche sich bezüglich der verfügbaren Debug- und Trace-Funktionalität unterscheiden [Kee00]:

- *Class 1*: Unterstützung der Standard Debug-Features. Speicherzugriffe sind nur bei gestoppter Ausführung über die JTAG-Schnittstelle möglich. Es ist keine spezielle Trace-Funktionalität gegeben.

- *Class 2*: Ergänzt eine dedizierte Schnittstelle für Program-Trace und Ownership-Trace.
- *Class 3*: Ergänzt Daten-Trace für Schreibzugriffe der ausgeführten Software über die Trace-Schnittstelle. Echtzeit-Zugriff ist möglich. Optional ist eine spezielle Variante des Daten-Trace verfügbar, bei welcher beliebig große Speicherbereiche über die Trace-Schnittstelle angefordert werden können (Data Acquisition).
- *Class 4*: Ergänzt Möglichkeit Program- Owernship- und Daten-Trace durch Watchpoints zu triggern. Ergänzt die Möglichkeit Lese- und Schreibzugriffe der ausgeführten Software auf die Trace-Schnittstelle umzuleiten und somit effektiv den internen Speicher der Prozessorhardware zu ersetzen (Memory Substitution).

Neben den Klassen definiert der Standard Protokolle für den Zugriff auf Trace- und Debug-Schnittstellen, sodass es Werkzeugherstellern einfacher möglich ist, spezielle Hardwareprodukte für die einfache Verbindung mit einem Entwicklungsrechner, beispielsweise über USB oder Ethernet, bereitzustellen. Im Rahmen dieser Arbeit nehmen derartige Werkzeuge die Rolle spezieller Zugriffshardware für interne Testgrößen ein und werden als Debug- und Trace-Hardware bzw. kurz als Debugger bezeichnet.

3.3 TEST AUF SYSTEMEBENE

Anwendungsfunktionen eingebetteter Systeme stehen in direktem Bezug zur physikalischen Umgebung des Systems (vgl. Kapitel 1). Häufig erfüllen sie eine regelnde Funktion, das heißt die Ausgaben des Systems wirken sich über die Umgebung wieder auf die Eingaben des Systems aus. Ohne die Existenz dieser Schleife ist kein sinnvoller Test der Anwendungsfunktion möglich. Ein Test in der realen Umgebung ist jedoch sehr aufwändig, schlecht reproduzierbar oder praktisch unmöglich.
Einen Ausweg bietet die sogenannte *Umgebungssimulation*, bei welcher das Verhalten der Umgebung durch Modelle beschrieben wird. Die äußere Feedbackschleife (Loop) existiert dann virtuell, was für die getestete Anwendungsfunktion im Idealfall transparent ist. Entsprechende Testmethoden werden unter dem Begriff *X-in-the-Loop* (XiL) zusammengefasst. Es lassen sich vier Methoden unterscheiden: Model-in-the-Loop (MiL), Software-in-the-Loop (SiL), Processor-in-the-Loop (PiL) und Hardware-in-the-Loop (HiL).

X-in-the-Loop

Die bei allen Methoden wesentliche Umgebungssimulation ist grundsätzlich nicht zu verwechseln mit der zuvor in Abschnitt 3.1.2 beschriebenen Verwendung eines Simulators als Testplattform, bei welchem auch die Ausführung selbst simuliert wird. Das ist nur bei MiL und SiL der Fall, wobei der Unterschied zwischen diesen beiden Methoden darin besteht, auf welcher Ebene die Ausführung simuliert wird.

MiL, SiL

Im Fall von MiL ist das die Modellebene, d.h. es werden die Verhaltensmodelle der Anwendungsfunktion, z.B. eines Reglers, direkt mit den Modellen der Umgebungssimulation gekoppelt. Im Fall von SiL liegt die Software bereits als ausführbares Programm vor, d.h. die Simulation erfolgt auf Instruktionsebene. Die Anwendungsfunktion muss dazu an eine Simulatorplattform „andocken“, welche auch die Funktionalität der späteren Basissoftware und des Betriebssystems simuliert (man spricht hier auch von einer „virtuellen ECU“). Durch Standardisierungsbemühungen wie im Rahmen von AUTOSAR (vgl. Abschnitt 2.1.4) wird dieser Schritt erleichtert, da die relevanten Schnittstellen klar spezifiziert sind.

PiL, HiL

Demgegenüber haben PiL und HiL zunächst gemeinsam, dass die Ausführung nicht mehr simulativ, sondern real auf der Zielplattform erfolgt. Daraus ergeben sich verschärfte Anforderungen an die Umgebungssimulation, welche nun harten Echtzeitanforderungen genügen muss [Haro1]. Der Unterschied zwischen PiL und HiL liegt in der Art der Verbindung mit der simulierten Umgebung. Im Fall von PiL ist diese softwaretechnisch realisiert (d.h. indirekt) und im Fall von HiL über die externen Schnittstellen (d.h. direkt). Im Rahmen dieser Arbeit ist vor Allem der HiL Test relevant und wird daher im Folgenden näher beleuchtet.

3.3.1 *Hardware-in-the-Loop Test*

Das Prinzip der Umgebungssimulation im Rahmen von HiL Tests ist in Abbildung 10 am Beispiel einer ECU als Testobjekt dargestellt. Die für XiL-Methoden typische Rückkopplung durch ein Modell der Umgebung ist gut ersichtlich.

Beim HiL Verfahren ist zwischen dem Umgebungsmodell und den externen Schnittstellen des Testobjekts spezielle Zugriffshardware erforderlich (vgl. Abschnitt 3.2.1). Diese muss mit der Ausführungsplattform des Umgebungsmodells zu einem echtzeitfähigen Gesamtsystem integriert werden, um den speziellen Anforderungen der HiL Simulation gerecht zu werden. Dieses Gesamtsystem wird als HiL Testsystem bezeichnet.

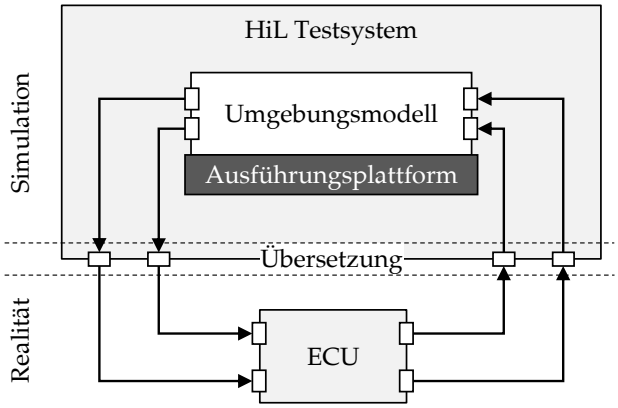


Abbildung 10: Prinzip der HiL Simulation am Beispiel einer ECU

Die Zugriffshardware kümmert sich um die Übersetzung von Größen der realen Welt (z.B. Spannung oder elektrische Last) in ihre virtuelle Repräsentation im HiL Testsystem und umgekehrt. In solchen Fällen wäre es häufig präziser, von „Umgebungsemulation“ anstelle von „Umgebungssimulation“ zu sprechen [Koe14]. In dieser Arbeit wird auf die genaue Unterscheidung dieser Begriffe jedoch verzichtet.

Restbussimulation

Ist das Testobjekt eine ECU, welche über ein Kommunikationsnetzwerk mit anderen ECUs kommuniziert, und sind diese anderen ECUs nicht real vorhanden, sondern ihr Verhalten wird auch nur durch das HiL Testsystem simuliert, so wird das häufig mit dem Begriff *Restbussimulation* bezeichnet. In dieser Arbeit ist die Simulation weiterer Teilnehmer in einem Kommunikationsnetzwerk im allgemein gehaltenen Oberbegriff „Umgebungssimulation“ inbegriffen.

3.3.2 *Werkzeuge für den HiL Test*

Es gibt eine große Zahl Anbieter von HiL Testsystemen und darüber hinaus unabhängige Dienstleister, welche ebenfalls bei der Integration kundenspezifischer Lösungen unterstützen. Meistens bieten diese spezielle Softwarewerkzeuge zur einfacheren Bedienung durch den Kunden oder Außendienstmitarbeiter an. Im Rahmen dieser Arbeit sind entsprechende Werkzeuge zur Testautomatisierung von Bedeutung, weswegen zur späteren Referenz einige wichtige Vertreter dieser Kategorie vorgestellt werden:

- *MBtech PROVEtech:TA*: PROVEtech:TA ist ein Werkzeug zur Testautomatisierung der MBtech GmbH & Co. KGaA, welches in Zusammenhang mit HiL Testsystemen verschiedener Hersteller eingesetzt werden kann und sich auf die Automobilindustrie fokussiert. Es bietet eine grafische Oberfläche zur Unterstützung bei interaktiven Tests durch Visualisierung von Testgrößen in Echtzeit. Automatisierte Abläufe können mit Hilfe einer integrierten Skriptsprache implementiert werden, welche auf WinWrap Basic [Onl/WWB] basiert, einer mit Visual Basic for Applications (VBA) [Onl/VBA] kompatiblen Skriptsprache der Firma Polar Engineering.
- *dSPACE AutomationDesk*: Die dSPACE (Digital Signal Processing And Control Engineering) GmbH ist ein branchenübergreifender Anbieter von HiL Testsystemen und zugehörigen Softwarewerkzeugen. In dieser Werkzeugpalette dient AutomationDesk der Testautomatisierung und Implementierung von Testfällen. Diese können grafisch in Form von Flussdiagrammen oder textuell mit der Skriptsprache Python oder in C# implementiert werden. In beiden Fällen wird der Zugriff über die ASAM XIL API (siehe Abschnitt 3.3.4) realisiert.
- *ETAS LABCAR-AUTOMATION*: Unter der LABCAR Produktfamilie bietet die ETAS Group GmbH Komponenten für HiL Testsysteme und zugehörige Softwarewerkzeuge an. LABCAR-AUTOMATION dient dabei der Testautomatisierung und Implementierung von Testfällen unter Verwendung von LABCAR Hardware. Die Eckdaten sind vergleichbar mit AutomationDesk. Die Implementierung erfolgt durch eine .NET API ebenfalls in C# oder in einer grafischen Form, welche an Flussdiagramme angelehnten ist.
- *National Instruments TestStand*: Beim Werkzeug TestStand von National Instruments liegt der Fokus auf der Verwaltung

von Testfällen, welche mit Hilfe LabVIEW kompatibler Zugriffshardware ausgeführt werden. Zur Beschreibung eines automatisiert ausführbaren Ablaufs müssen einzelne Schritte über Dialogfelder zu Sequenzen kombiniert werden. Die Implementierung einzelner Schritte erfolgt in der Regel mit LabVIEW. Alternativ kann für einen Schritt auf separat kompilierte Bausteine zurückgegriffen werden, beispielsweise .NET Assemblies, die dann eine bestimmte Schnittstelle bieten müssen.

- *PikeTec TPT*: TPT (Time Partition Testing) der PikeTec GmbH ist ein Werkzeug zur grafischen Implementierung von Testfällen, welche anschließend auf verschiedenen Plattformen (MiL, SiL, PiL, HiL) gleichermaßen ausgeführt werden können. Die Beschreibung von Abläufen geschieht mit Flussdiagrammen oder einfachen Listen. Auf Basis der Skriptsprache Python können mit Hilfe spezieller Bibliotheken komplexe Berechnungen, etwa zur Auswertung kontinuierlicher Signalverläufe, durchgeführt werden.

- *Vector Informatik CANoe*: Das Werkzeug CANoe der Vector Informatik GmbH fokussiert sich auf Netzwerke von Steuergeräten in Kraftfahrzeugen (CAN, FlexRay etc.). Es dient vor Allem als Bedienoberfläche für die von Vector Informatik vertriebene Zugriffshardware für Netzwerke im Automobilbereich und unterstützt so bei der Steuergeräteentwicklung. Die Automatisierung von Tests ist also nicht der primäre Verwendungszweck des Werkzeugs, wird von CANoe jedoch unterstützt. Testfälle können mit der domänenspezifischen Sprache CAPL (CAN Access Programming Language) implementiert werden (siehe Abschnitt 3.5). Darüber hinaus ist eine .NET API verfügbar, welche ebenfalls zur Testautomatisierung verwendet werden kann.

3.3.3 *Implementierung von Testfällen*

Eine Voraussetzung für Testautomatisierung ist die Existenz einer Testimplementierung. Im vorhergehenden Abschnitt wurden die für das jeweilige Testautomatisierungswerkzeug relevanten Beschreibungsmethoden bereits erwähnt.

grafische DSLs

Häufig können Testfälle dort in grafischer Form, z.B. durch Flussdiagramme, beschrieben werden. Dies erleichtert Anwendern ohne

Programmierkenntnisse den Zugang, führt bei komplexen Abläufen und Beziehungen jedoch zu sehr großen Darstellungen, welche nicht mehr wesentlich einfacher erfasst werden können, als textuell programmierte Abläufe. Beispielsweise lassen sich Kontrollstrukturen wie Schleifen und Bedingungen mit zunehmender Verschachtelungstiefe nicht mehr in einer gemeinsamen Ansicht darstellen. Grafische Programmierung wird außerdem für den hohen Zeitaufwand zur grafischen Anordnung von Elementen kritisiert. Bei Änderungen ist die grafische Anordnung häufig komplett zu überarbeiten, um die Lesbarkeit zu wahren. Weil damit keine echte Information hinterlegt wird, ist das äußerst unproduktiv. Durch das Einfügen von Kommentaren in der Implementierungsquelle wird das Problem noch verschärft. Schließlich ist man zur Bearbeitung der grafischen Darstellung fast immer an ein spezielles Werkzeug gebunden, zu dem es keine Alternative gibt (Vendor-Lock-in). Ein Vorteil ist, dass der Dokumentationsaufwand durch das Vorhandensein grafischer Modelle sinkt.

GPLs

Alle HiL Testsysteme bieten die Möglichkeit der Programmierung flexibler Abläufe mit Hilfe von Standardprogrammiersprachen (GPLs), indem entsprechende APIs bereitgestellt werden. Als Basistechnologie vorherrschend sind hier die Programmiersprachen C# und Python.

textuelle DSLs

Bei einigen Werkzeugen steht darüber hinaus eine textuelle DSL zur Verfügung, welche ähnlich grafischer Beschreibungsmethoden in das jeweilige Testautomatisierungswerkzeug integriert sein können. Ansätze auf Basis textueller DSLs versuchen sich zwischen GPLs und grafischer Testfallbeschreibung zu positionieren. Einerseits sollen komplexe Abläufe weiterhin effizient und einfach wartbar aufgeschrieben werden können, andererseits der Abstraktionsgrad gegenüber einer GPL erhöht werden, sodass weniger Spezialwissen bezüglich einer Programmiersprache notwendig ist.

In der Regel werden Beschreibungen in einer textuellen DSL genauso wie grafische Darstellungsformen vor der Testausführung in äquivalenten Quelltext einer GPL übersetzt. Bei der Ausführung wird dann auf die für das jeweilige HiL Testsystem bereitgestellte API zurückgegriffen. Die in dieser Arbeit vorgestellte Testlösung stellt diesbezüglich keine Ausnahme dar.

3.3.4 *ASAM XIL API*

Die für HiL Testsysteme bereitgestellten APIs sind demnach ein wesentliches Hilfsmittel bei der Implementierung automatisierter Tests. Die Association for Standardization of Automation and Measuring Systems (ASAM e.V.) hat daher 2009 die erste Version eines Standards für eine Hersteller- und Technologie-unabhängige API bezüglich HiL-Testsystemen veröffentlicht (ASAM HIL API) [Std/ASA09]. Ende 2013 wurde dieser durch die Version 2.0.0 abgelöst (ASAM XIL API), welche einerseits eine Erweiterung bezüglich der übrigen „In-the-Loop“ Methoden darstellt, in der Hauptsache jedoch die Herstellerunabhängigkeit durch eine zusätzliche Abstraktionsschicht (Framework API, s.u.) verbessert. Inzwischen ist der Standards in Version 2.0.2 verfügbar [Std/ASA15b]. Abbildung 11 zeigt die dort beschriebene Architektur eines Testsystems.

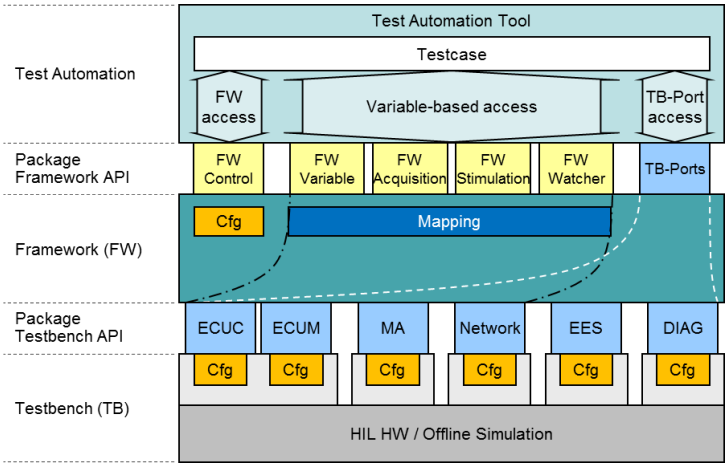


Abbildung 11: Überblick über ASAM XIL API [Std/ASA15b]

Die unterste Schicht (HIL HW) entspricht dem HiL Testsystem gemäß der Terminologie in dieser Arbeit, d.h. umfasst neben der Hardware zur Signalkonditionierung auch die Ausführungsplattform des Umgebungsmodells, bzw. der entsprechenden Plattform zur Offline-Simulation im Fall von MiL oder SiL. Die darüber dargestellten hellgrauen Boxen repräsentieren die möglichen Schnittstellen zum HiL Testsystem, welche jeweils herstellerspezifisch angesprochen und konfiguriert werden müssen. Die darüber liegende Schicht *Testbench API* hat ihren Ursprung in der ersten Version des Standards. Durch sogenannte „TB Ports“ erfolgt eine Abstraktion von der herstellerspezifischen Schnittstelle womit bereits

Testbench API

weitgehende Herstellerunabhängigkeit erreicht wird. Der Standard unterscheidet hier zwischen den folgenden TB Ports:

- *MA*: Model Access; Schnittstelle zum Zugriff auf Modellgrößen. Das umfasst nicht nur Umgebungsmodell-interne Variablen, sondern auch externe Testgrößen, falls diese durch Modellvariablen repräsentiert sind.
- *Network*: Schnittstelle für den direkten Netzwerkzugriff, d.h. ohne „Umweg“ über MA, z.B. für CAN. Erst ab Version 2.0.0 Teil des Standards.
- *ECUM/ECUC*: Schnittstelle zu Server für Messung und Kalibrierung (basierend auf älterem Standard ASAM MCD-3 MC [Std/ASA11a]). Das entspricht in der Terminologie dieser Arbeit indirektem externem Zugriff auf interne Testgrößen durch Nutzung spezieller Protokolle, z.B. XCP (vgl. Abschnitt 3.2.3).
- *DIAG*: Schnittstelle zu Server für Diagnose (basierend auf älterem Standard ASAM MCD-3 D [Std/ASA11b]). Der Unterschied zu ECUM/ECUC besteht in der Fokussierung auf ECU-interne Funktionen für den Selbsttest und zur Fehlerspeicherung. DIAG ist im Kontext dieser Arbeit ebenfalls indirektem externen Zugriff zuzuordnen.
- *EES*: Electric Error Simulation; Schnittstellen zum Steuern der Fehlersimulation (z.B. Kurzschlüsse), welche von vielen HiL Testsystemen für einfache elektrische Pins unterstützt wird.

Auf Basis der Testbench API ist bei jedem Zugriff auf Testgrößen explizit festzulegen, über welchen TB Port der Zugriff erfolgen soll. Darüber hinaus ist initial der genutzten TB Ports herstellerspezifisch zu konfigurieren. Dadurch ist die Wiederverwendbarkeit von Testfällen beeinträchtigt. Dieses Problem wird mit der zweiten Version des Standards behoben, welche dazu als weitere Abstraktionsschicht ein Framework (FW) mit zugehöriger *Framework API* einführt. Das Framework übernimmt die Konfiguration und Auswahl des jeweils passenden TB Ports anhand einem Mapping, welches einmalig konfiguriert werden muss. Diese Konfiguration kann global und unabhängig von einzelnen Testfällen erfolgen, was deren direkte Wiederverwendbarkeit ermöglicht.

Framework API

Aus einem Testfall heraus erfolgt der Zugriff auf Testgrößen dann anhand von Objektinstanzen, welche die jeweilige Testgröße repräsentieren, und vom Framework auf Basis eines eindeutigen Namens zur Verfügung gestellt werden. Damit können die weiteren

Schnittstellen der Framework API genutzt werden, welche für die identifizierte Testgröße die Rückgabe bzw. das Setzen eines Werts (FW Variable), die Messung über einen Zeitraum (FW Acquisition), die Generierung eines Signals (FW Stimulation), sowie das Setzen von Triggern bezüglich Ereignissen (FW Watcher) unterstützt.
Sowohl für Framework API als auch Testbench API existiert eine Referenzimplementierung in den Programmiersprachen C# und Python. Die jeweils relevanten Teile des Standards werden von einigen Werkzeugen aus dem HiL Umfeld unterstützt. Von den in Abschnitt 3.3.2 genannten Herstellern umfasst das die dSPACE GmbH, ETAS GmbH, MBtech Group GmbH & Co. KGaA, National Instruments Corporation und Vector Informatik GmbH.

3.4 TEST AUF QUELLTEXTEBENE

In den Softwareschichten unterhalb der Applikationssoftware finden sich Softwaremodule für den Hardwarezugriff, zur Hardwareabstraktion oder für allgemeine Dienste (vgl. Abbildung 5), welche ebenfalls automatisiert getestet werden müssen. Gerade bei Treibermodulen und Modulen zur Hardwareabstraktion ist ein Test auf der Zielplattform wichtig, da diese in direktem Bezug zur ausführenden Prozessorhardware bzw. Peripherie stehen.
Bei Testobjekten dieser Ebene ist es nicht möglich, alle relevanten Zustände über die externen Schnittstellen des Systems zu provozieren. Stattdessen ist für einen gezielten Test der Zugriff auf interne Testgrößen erforderlich (vgl. Abschnitt 3.2). Außerdem steht jetzt das durch Funktionsaufrufe direkt hervorgerufene Verhalten im Fokus, während die physische Umgebung des Systems (meistens) keine Rolle mehr spielt.

klassischer Unit-Test

Unter diesen Voraussetzungen bietet sich zunächst der klassische Unit-Tests Ansatz basierend auf Codeinstrumentierung an. Das bedeutet, dass Testfälle zusammen mit der zu testenden Software auf der Zielplattform ausgeführt werden, und die Implementierung mit Hilfe derselben Technologie erfolgt, mit der auch die Software erstellt wird. Dieses grundsätzliche Vorgehen ist in Abbildung 12 dargestellt.

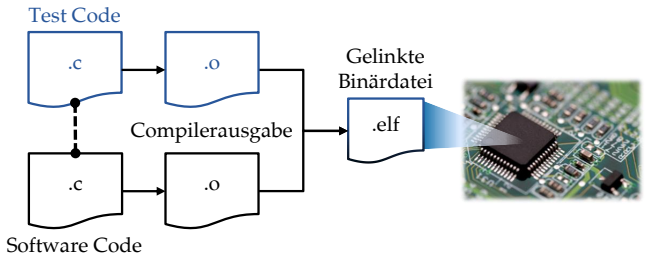


Abbildung 12: Grundsätzliches Vorgehen zum Unit-Test eingebetteter Software auf der Zielplattform

3.4.1 *Unit-Test Frameworks*

Der zuvor beschriebe Ansatz kann durch Frameworks unterstützt werden, die unter dem Begriff *xUnit* zusammengefasst werden. Für C/C++ Code bietet sich dazu CppUnit[5] an [SFL+13]. Ein weiteres Beispiel ist CUTE [SG07]. In beiden Fällen wird die Verwendung von C++ voraussetzt, was für eingebettete Software nicht immer in Frage kommt. Das trifft insbesondere auf sicherheitskritische Systeme zu, weil sich dort durch dynamische Speicherallokierung und virtuelle Funktionen zusätzliche Probleme bei der Zertifizierung ergeben. Es gibt jedoch noch weitere Gründe für die Ablehnung von C++, von fehlender Compilerunterstützung bis hin zu unzureichendem Wissen der Entwickler [Her15]. Dementsprechend gibt es Unit-Test Frameworks speziell für C-Code, Beispiele sind CUnit[6], CuTest[7] und AceUnit[8]. Das Prinzip zur Verwendung von Unit-Test Frameworks ist in Abbildung 13 dargestellt.

xUnit

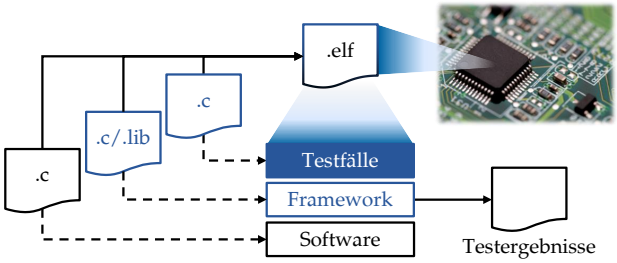


Abbildung 13: Verwendung von Unit-Test Frameworks für eingebettete Software auf der Zielplattform

5 http://freedesktop.org/wiki/Software/cppunit/
6 http://cunit.sourceforge.net/
7 http://cutest.sourceforge.net/
8 http://aceunit.sourceforge.net/

Gegenüber dem naiven Ansatz bestehen besteht zunächst der Vorteil, dass die Implementierung der Testfällen einheitlich und klarer von den übrigen Quelltexten getrennt ist, da die Schnittstellen des Frameworks genutzt werden müssen. Darüber hinaus übernimmt das Framework Aufgaben zur Steuerung und Protokollierung der Testfälle. So ist es möglich, mehrere Tests zu einer einer sogenannten *Test Suite* zusammenzufassen. Das ermöglicht die logische Strukturierung, sodass zusammengehörige Tests automatisiert an einem Stück ausgeführt werden können und ist insbesondere für Regressionstest interessant.

Test Suite

Hinsichtlich der Testauswertung stellen Unit Test Frameworks einheitliche Methoden zur Verfügung, meistens in Form von Zusicherungen (Assertions). Schlägt eine Assertion fehl, wird das durch das Framework automatisch protokolliert und dem entsprechenden Testfall zugewiesen, sodass nach Abarbeitung eines Tests bzw. einer Test Suite ein übersichtlicher Report erstellt werden kann.

Bei Ausführung auf der Zielplattform kommt Unit Test Frameworks dabei eine besondere Bedeutung zu, da die Bereitstellung eines solchen Reports nicht trivial ist. Üblicherweise erfolgt die Ausgabe direkt auf ein Terminal oder durch das Speichern im Dateisystem. Beides ist bei eingebetteten Systemen, welche im Fokus dieser Arbeit stehen, kaum verfügbar. Die Testergebnisse müssen daher auf anderem Weg an den Entwicklungsrechner übertragen werden. Da dieser Aspekt für jedes eingebettet System spezifisch ist, bieten Unit Test Frameworks dazu keine vollständige Implementierung, sondern spezifizieren lediglich Schnittstellen, welche dem Anwender eine systemspezifische Realisierung erlauben.

3.4.2 *Alternativen zu Unit-Test Frameworks*

Die Verwendung von Unit Test Frameworks ist bei eingebetteter Software also zunächst mit erhöhtem Aufwand verbunden. Außerdem ist spezielles Entwicklerwissen erforderlich, um entsprechende Frameworks in Betrieb zu nehmen. Der grundsätzlich schlechten Praxis, dass Softwareentwickler die Tests für ihren Code selbst schreiben, wird dadurch Vorschub geleistet.

Es gibt daher eine Reihe von Softwarewerkzeugen für den Unit-Test eingebetteter Systeme, welche entsprechende Frameworks ersetzen und den genannten Problemen begegnen. An dieser Stelle sollen zunächst drei Beispiele genannt werden, welche das gleiche Prinzip verfolgen: Tessy der Razorcat Development GmbH, VectorCAST der Vector Software Corporation und C/C++ Test der ParaSoft Corporation.

Die Gemeinsamkeit der Werkzeuge besteht in der Verwendung eines sogenannten *Test Harness*, oder auch Testrahmen, welcher im Wesentlichen die Rolle des Unit Test Frameworks übernimmt. Der Unterschied ist, dass der Test Harness durch das betreffende Testwerkzeug generiert wird. Er umfasst alle Software, die zur Ausführung von Tests nötig sind, d.h. die Testdaten, den Code der Testtreiber sowie möglicherweise erforderlichen Glue Code. Der *Testtreiber* entspricht dabei dem Testfall bei Unit-Tests im klassischen Sinne: er ruft die für den Test relevanten Funktionen mit der gewünschten Parametrierung auf und überprüft Rückgabewerte sowie den globalen Zustand.

Test Harness

Testtreiber

Die Implementierung von Testfällen erfolgt bei den genannten Werkzeugen jedoch nicht mehr direkt in Form von Quelltext, sondern anhand einer abstrakteren Testfallspezifikation. Üblicherweise sind das Listen der aufzurufenden Funktionen mit entsprechender Parametrierung und den erwarteten Rückgabewerten.

Die genannten Werkzeuge bieten eine intuitive Bedienoberfläche, sodass auch Personen ohne Programmiererfahrung automatisiert ausführbare Tests erstellen können. Das grundsätzliche Problem bezüglich dem Anstoßen von Tests auf der Zielplattform sowie dem Bereitstellen der Testergebnisse auf dem Host-PC bleibt davon zwar zunächst unberührt, wird jedoch durch verschiedene Ansätze vor dem Anwender verborgen. Dazu kommen einerseits reine Softwarelösungen zum Einsatz, beispielsweise auf Basis von speziellen Diagnosefunktionen des Betriebssystems auf der Zielplattform. Im Rahmen dieser Arbeit ist jedoch vor Allem die Integration von Debug Hardware interessant. In diesem Fall werden Skripte generiert, welche die generierten Testtreiber über die Debug-Schnittstelle des Mikrocontrollers triggern, wie es in Abbildung 14 veranschaulicht wird.

3.4.3 *Vermeidung von Codeinstrumentierung*

Die im letzten Abschnitt vorgestellten Werkzeuge für den Unit-Test basieren auf Codeinstrumentierung, denn es werden zu Testzwecken zusätzlicher Programmcode und Testdaten auf die Zielplattform geladen (vgl. Seite 43). Codeinstrumentierung kann aus verschiedenen Gründen unerwünscht oder nicht möglich sein. Ein wichtiger Grund zur Vermeidung ist, dass sich das Zeitverhalten der Software dadurch verändert, denn es werden zusätzliche Instruktionen ausgeführt und die Prozessorpipeline sowie der Cache sind anders gefüllt, als es ohne den zusätzlichen Testcode der Fall

wäre. Testergebnisse sind dadurch weniger belastbar. In der ISO 26262 wird bezüglich Codeinstrumentierung empfohlen:

> If instrumented code is used to determine the degree of coverage, it can be necessary to show that the instrumentation has no effect on the test results. This can be done by repeating the tests with non-instrumented code.
>
> — *ISO 26262-6 Abs. 9.4.5 Note 3 [Std/ISO11a]*

Diese Empfehlung ist auf einen Spezialfall eingeschränkt, nämlich der Nutzung von Codeinstrumentierung zur Messung der Codeabdeckung. Zu anderen Fällen von Codeinstrumentierung, beispielsweise für den Zugriff auf Testgrößen und zur Testauswertung auf der Zielplattform, macht die Norm keine Vorgaben. Dennoch wird an diesem Beispiel deutlich, dass Codeinstrumentierung bei sicherheitskritischen Systemen ein Problem darstellen kann. Darüber hinaus kann das Hinzufügen von Testcode auf der Zielplattform wegen begrenztem Speicher ausgeschlossen sein.
Der in Abbildung 14 dargestellte Ansatz deutet bereits einen Ausweg an. Bietet die Prozessorhardware geeignete Debug- und Trace-Schnittstellen, ist ein generierter Test Harness auf der Zielplattform nicht mehr erforderlich. Vielmehr kann das dargestellte Testskript auch direkt die Rolle des Testtreibers übernehmen.

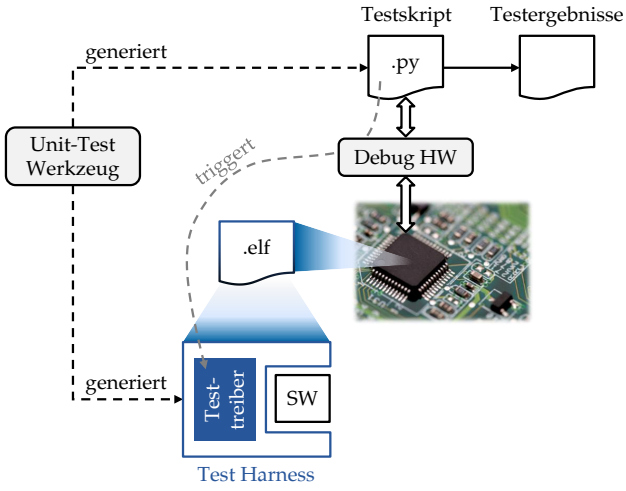


Abbildung 14: Typische Verwendung von Debug Hardware durch Unit-Tests Werkzeuge für eingebettete Software

Listing 1: Beispiel eines Testskripts auf Basis der isystem.connect API

```
1  import isystem.connect as ic
2
3  # init connection to debugger
4  cmgr = ic.ConnectionMgr()
5  cmgr.connectMRU()
6  debug = ic.CDebugFacade(cmgr)
7
8  # boot target software
9  debug.runUntilFunction("main")
10 debug.waitUntilStopped()
11
12 # call function 'mul' and check result
13 result = int(debug.call("mul", "2", "4"))
14
15 if (result != 8)
16   print("Failure!")
```

Beispiel für ein Debugger Testskript

Um zu veranschaulichen, wie eine solches Testskript in der Praxis aussehen kann, ist in Listing 1 ein einfaches Beispiel gegeben, das die isystem.connect API [Onl/iSys] der Firma iSYSTEM zur Implementierung eines einfachen Testfalls in der Programmiersprache Python nutzt.
Die Zeilen 4 bis 10 dienen dabei der Vorbereitung. Das umfasst den Verbindungsaufbau mit der Debug Hardware (Zeile 4-6) und die Initialisierung der Software auf der Zielplattform (Zeile 9 und 10). Die übrigen Zeilen implementieren den eigentlichen Testfall für eine einfache Funktion zur Multiplikation zweier Ganzzahlen (`mul`). Zeile 13 triggert die Ausführung von `mul` auf der Zielplattform, wobei die Parameter „2“ und „4“ übergeben werden. Anschließend wird der Rückgabewert überprüft und bei Abweichung von der Erwartung der Text „Failure!“ ausgegeben.
Die hinter der API verborgene Komplexität ist schon bei diesem einfachen Beispiel größer, als es zunächst den Anschein hat: zuerst müssen die Adressen für die Funktion „`mul`“ sowie des Stacks bestimmt werden, und anschließend die Parameter auf den Stack kopiert, der Stack- und Instruktionszeiger angepasst, Breakpoints an den Rücksprungpunkten der Funktion gesetzt und der Prozessor aus dem Haltemodus genommen werden. Je nach Art und Weise Anbindung des Debuggers an den Host-PC ist dabei USB- oder Ethernet-Kommunikation involviert und außerdem das für den Mikrocontroller spezifische Protokoll (z.B. via JTAG-Schnittstelle).

Dieses Beispiel zeigt, dass es bereits sehr einfach sein kann, automatisiert ausführbare Tests auf Quelltextebene zu implementieren, die vollständig ohne Codeinstrumentierung auskommen und auch ansonsten frei von Abhängigkeiten bezüglich der Software auf der Zielplattform sind (z.B. Betriebssystem).

Auf Debugger spezialisierte Werkzeuge

Die Verwendung von APIs in Testskripten bietet sehr große Flexibilität bezüglich möglicher Abläufe. Ist diese Flexibilität nicht erforderlich, kann auch auf Testwerkzeuge zurückgegriffen werden, welche sich auf die Nutzung von Debug Hardware spezialisiert haben. Beispiele sind CTestIt! der COSMIC Software GmbH und TestIDEA der iSYSTEM AG. Gegenüber dem Anwender stellen sich diese sehr ähnlich zu den bereits im vorhergehenden Abschnitt vorgestellten Werkzeugen dar, d.h. sie bieten eine intuitive Dialogfeld-basierte Oberfläche um Testfälle zu erstellen. Das entsprechende Vorgehen ist in Abbildung 15 dargestellt.

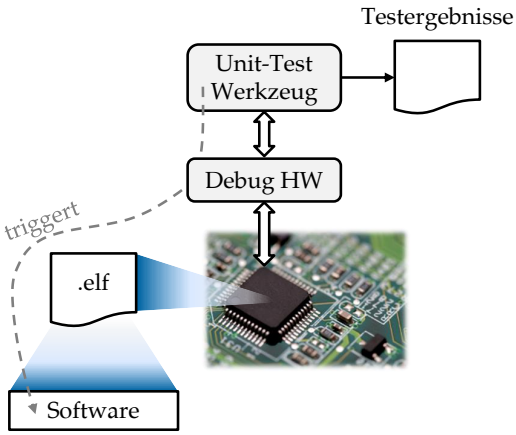


Abbildung 15: Spezielle Verwendung von Debug Hardware durch Unit-Tests Werkzeuge für eingebettete Software bei Vermeidung von Codeinstrumentierung

3.5 SPRACHEN ZUR TESTAUTOMATISIERUNG

Zur Automatisierung von Tests werden also vorwiegend GPLs, manchmal aber auch textuelle DSLs eingesetzt. Im folgenden Abschnitt wird zunächst auf die wesentlichen Eigenschaften von GPLs eingegangen, bevor in Abschnitt 3.5.2 die Testing and Test Control Notation Version 3 (TTCN-3) vorgestellt wird, welche im Kontext dieser Arbeit besonders relevant ist. Weitere DSLs zur Automatisie-

rung von Tests für eingebettete Software werden anschließend in Abschnitt 3.5.3 zusammengefasst.

3.5.1 *Standardsprachen*

Die GPLs, welche zur Testautomatisierung für eingebettete Software ihm Rahmen der genannten Lösungen am häufigsten eingesetzt werden, sind C, C++, C# und Python. C/C++ wird vorwiegend im Rahmen von Unit-Tests auf der Quelltextebene verwendet, d.h. wenn der Testcode ebenfalls auf der Zielplattform ausgeführt wird. C# und Python finden Anwendung, wenn der Testcode auf einem Host-PC ausgeführt wird, d.h. wenn eine externe Sicht auf das Testobjekt besteht. Das ist im Rahmen von HiL Tests auf der Systemebene immer der Fall, und auch auf der Quelltextebene, falls dort Debug- und Trace-Hardware direkt eingesetzt wird.
C und C++ haben ihren Ursprung in der hardwarenahen Softwareentwicklung. C wurde Anfang der 70er Jahre von Dennis Ritchie entwickelt [Rit93], C++ ist eine Erweiterung von C durch Bjarne Stroustrup zu Beginn der 80er Jahre. Python wurde 1991 veröffentlicht, C# (englisch gesprochen „see sharp“) 2001. Beide wurden als hardwareunabhängige Programmiersprachen entworfen und kennen keine *Pointer* wie C/C++. Alle Sprachen werden zunächst kompiliert. Die Standardcompiler für C# und Python erzeugen jedoch einen Zwischencode, der durch eine Virtuelle Maschine ausgeführt bzw. interpretiert wird.[9] C/C++ Quelltext wird direkt in nativen Maschinencode übersetzt.
Im Rahmen dieser Arbeit ist noch der Unterschied bezüglich *Typisierung* erwähnenswert. C++ und C# sind großteils[10] statisch typisiert, Python dynamisch. Statische Typisierung bedeutet, dass der Typ eines Datums bereits zur Kompilierungszeit fest steht, während er bei dynamischer Typisierung erst zur Laufzeit bekannt ist. Statische Typisierung verhindert sehr viele Programmierfehler und entsprechende Sprachen gelten daher als robuster. Statische Typisierung spielt in dieser Arbeit zur Sicherstellung von Konsistenz zwischen Testfällen und dem Quelltext der getesteten Software eine besondere Rolle (siehe Kapitel 5).

Typisierung

9 Im April 2014 hat Microsoft mit „.NET Native“ einen Compiler vorgestellt, der nativen Code für einige Architekturen erzeugt.

10 Nur wenige Sprachen sind tatsächlich vollständig statisch typisiert. Durch verschiedene Mechanismen, in C++ z.B. durch Pointer, kann das Typsystem meistens umgangen werden.

Davon abgesehen existieren viele Gemeinsamkeiten zwischen den genannten Sprachen. So sind nicht nur syntaktische Ähnlichkeiten offensichtlich, sondern auch die vorherrschende Programmiertechnik: alle Sprachen implizieren an erster Stelle die Verwendung imperativer Programmierparadigmen, d.h. strukturierte, prozedurale und modulare Programmierung.
Von keiner dieser Sprachen wird der Test besonders berücksichtigt. Eine genauere Betrachtung der Sprachen ist für diese Arbeit daher nicht nötig. Für weitere Informationen zu den einzelnen Sprachen sei auf entsprechende Standards und Dokumentation verwiesen [Std/C11; Std/CPP14; RD12; MS12].

3.5.2 *TTCN-3*

Die Testing and Test Control Notation in Version 3 (TTCN-3) [Std/ETS15] ist eine „international standardisierte Sprache zur Testspezifikation für eine große Auswahl an Computer- und Telekommunikationssystemen“ [Wil05]. TTCN-3 wird vor Allem in der Telekommunikationsbranche eingesetzt und ein wesentliches Augenmerk liegt auf dem Test von Kommunikationsprotokollen und deren Schnittstellen in Software. Das ist mitbegründet durch die Vorgänger der Sprache (TTCN-2 und TTCN), da diese gezielt für diesen Anwendungsfall entworfen wurden. Hier steht die Abkürzung für „Tree and Tabular Combined Notation“. TTCN-3 ist gegenüber den Vorgängern jedoch nicht länger auf kommunikationsorientierten Systeme beschränkt, sondern kann auch in anderen Domänen wie der Automobilindustrie und dem Eisenbahnsektor eingesetzt werden [BDI06; Hen04; BIV05].

Beschreibung von Echtzeitverhalten

Es gibt Anzeichen, dass für die hier vorherrschenden Steuerungs- und Regelsysteme eine Erweiterung von TTCN-3 zur besseren Beschreibung von Echtzeitverhalten und zur Erfassung kontinuierlicher Verläufe wie im Fall analoger Signale sinnvoll wäre [GSS09; SBG05].
Bisher sind derartige Vorschläge jedoch nicht in den Standard eingegangen. Ein wichtiger Grund hierfür dürfte die Tatsache sein, dass in TTCN-3 mit dem `timer` Objekt bereits ein Mechanismus zur Beschreibung von Zeit existiert. Die Autoren von [GSS09] argumentieren, dass TTCN-3 `timer` auf dem Konzept von Momentaufnahmen (engl. snapshot) basieren, und daher nicht echtzeitfähig seien. Tatsächlich können aktuell existierende TTCN-3 Testsysteme keine echtzeitfähige Ausführung im Rahmen der Genauigkeit von Mikrosekunden garantieren. Das ist aber vielmehr ein Problem der

Laufzeitumgebung und prinzipiell ist die Beschreibung von entsprechendem Zeitverhalten durchaus möglich.
Jedoch ist in der Praxis eine echtzeitfähige Ausführung von Szenarien, welche anhand beliebiger TTCN-3 `timer` Objekte beschrieben sind, nicht realistisch. Insofern ist der Vorschlag nachvollziehbar, die Sprache um intuitive und weniger flexible Konstrukte speziell zur Beschreibung von Echtzeitverhalten zu erweitern. Leider impliziert der konkrete Vorschlag zwei konzeptuell und syntaktisch stark unterschiedliche Methoden mit dem grundsätzlich gleichen Zweck (nämlich der Behandlung von Zeit). Im primären Einsatzgebiet der Sprache reicht die aktuell verfügbare Präzision jedoch bereits aus, weswegen eine Spracherweiterung dort vorrangig den Nachteil gesteigerter Komplexität mit sich bringt.

Eignung für Steuerungs- und Regelsysteme

Diese Überlegungen führen zu der Frage, wie gut TTCN-3 tatsächlich für den Tests von Software echtzeitkritischer Steuerungs- und Regelsysteme geeignet ist, die nicht anhand von Protokolldaten durchgeführt werden können, und ob in Zukunft eine bessere Unterstützung solcher Anwendungsfälle zu erwarten ist, wenn sich daraus Nachteile in der ursprünglichen Domäne ergeben. Es ist derzeit nicht möglich diese Frage mit Bestimmtheit zu beantworten. Tatsache ist, dass die Verbreitung von TTCN-3 in den Bereichen der Automobilindustrie, Luft- und Raumfahrt sowie Eisenbahnwesen im Vergleich zum Telekommunikationssektor geringer ist. Wird hier TTCN-3 eingesetzt, dann ebenfalls vorwiegend im Rahmen des ursprünglichen Anwendungsszenarios, d.h. bezüglich protokollbasierter Netzwerkkommunikation. Für den Hardware-in-the-Loop Test auf Systemebene oder Tests auf Quelltextebene, welche im Fokus dieser Arbeit stehen, wird TTCN-3 derzeit nicht eingesetzt.
Dennoch ist TTCN-3 von den etablierten Sprachen diejenige, welche der im Rahmen dieser Arbeit entwickelten Programmiersprache zur Testimplementierung am nächsten kommt. Das grundlegende Prinzip von TTCN-3 soll daher mit Fokus auf die im Kontext dieser Arbeit besonders relevanten Sprachkonstrukte vorgestellt werden. Eine Schilderung aller Aspekte der ausgewählten Konstrukte würde den Rahmen dieser Arbeit sprengen, und für eine genauere Einführung ist das Buch „An Introduction to TTCN-3“ [Wil05] zu empfehlen, das auch als Grundlage für die folgenden Beispiele diente.

Alternative Darstellungsformen

Bezüglich TTCN-3 muss als erstes die Besonderheit erwähnt werden, dass neben einer textuellen Notation (Core Language, siehe [Std/ETS15]) auch eine grafische Darstellung von Abläufen (Graphical Presentation Format, [Std/ETS07a]) basierend auf Message Sequence Charts [Std/ITU11], sowie eine tabellarische Darstellung (Tabular Presentation Format, [Std/ETS07b]) exakt spezifiziert sind. Diese alternativen Darstellungsformen bilden jeweils eine echte Untermenge der Core Language. Keine der beiden alternativen Darstellungsformen haben jedoch größere Akzeptanz in der Gemeinschaft der TTCN-3 Nutzer erlangt [Wil05].

TTCN-3 Module

Auf höchster Ebene besitzt TTCN-3 genau ein Sprachkonstrukt, das `module`. Alle übrigen Definitionen für Datentypen, Testfälle und Funktionen sind darin enthalten, ebenso wie maximal ein `control` Block. Er ist der Einstiegspunkt zur späteren Ausführung, ähnlich der `main`-Funktion in einem C-Programm. Von dort wird die Ausführung separat implementierter Testfälle angestoßen.

```
1  module
2  {
3    // Definitionen zu Beginn
4
5    // Optionaler control Block am Ende
6    control
7    {
8      execute(MeinTestfall());
9    }
10 }
```

TTCN-3 Port und Component

TTCN-3 wurde mit Augenmerk auf kommunikationsbasierte Systeme entworfen. Testfälle basieren daher auf zwei Arten von Kommunikation: erstens dem Austausch von Nachrichten zwischen den einzelnen Komponenten eines verteilten Systems, und zweitens dem Remote Procedure Call (RPC) zum Aufruf einer fernen Prozedur.

Modellierung von Komponenten

In TTCN-3 ist zunächst die Modellierung beteiligter Komponenten erforderlich, bevor die Implementierung eines Testfalls erfolgen kann. Hierzu stehen zwei spezielle Typen zur Verfügung: `port` und `component`. Der Typ `component` enthält im Wesentlichen eine Liste

der verfügbaren Schnittstellen einer Komponente, welche wiederum durch den Typ `port` beschrieben werden.
Für einen `port` ist als erstes anzugeben, ob die Kommunikation auf dem Austausch von Nachrichten oder RPC beruht. Anschließend ist für jeden möglichen Nachrichtentyp (bzw. jede Funktionssignatur im Fall von RPC) die Richtung anzugeben. Möglich sind `in`, `out` und `inout`. Das entspricht dem Empfangen von Nachrichten, Senden von Nachrichten oder beidem, bzw. im Fall von RPC, welche Seite den Aufruf initiiert.
Das soll mit einem Beispiel veranschaulicht werden. Als Testobjekt dient dazu eine Komponente mit genau einer Schnittstelle zum Senden und Empfang von Nachrichten und einem primitiven Protokoll: Nachrichten enthalten eine ganze Zahl als Nutzdaten und sonst keine weiteren Informationen. Sobald die Komponente eine Nachricht empfängt, sendet sendet sie eine identische Nachricht zurück.
Zum Test dieser Funktionalität muss in TTCN-3 ein passender Kommunikationspartner beschrieben werden, der die Daten an das Testobjekt sendet bzw. empfängt. Dieser Aufbau ist in Abbildung 16 dargestellt.

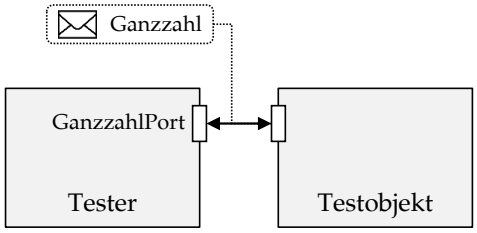


Abbildung 16: Testaufbau für ein einfaches TTCN-3 Beispiel

Der Kommunikationspartner (`Tester`) wird durch folgenden TTCN-3 Code realisiert:

```
1  // Modell der Schnittstelle passend zum Protokoll
2  type port GanzzahlPort message
3  {
4    inout integer
5  }
6
7  // Komponente des Testsystems
8  type component Tester
9  {
10   port GanzzahlPort gz_port
11 }
```

TTCN-3 Testcase

Testfälle werden in TTCN-3 im Konstrukt `testcase` beschrieben. Dies erfolgt immer aus Sicht einer Komponente, d.h. der entsprechende Typ muss durch das Schlüsselwort `runs on` explizit angegeben werden. Im Implementierungsteil des Testfalls stehen dann die Schnittstellen der Komponente zum Senden und Empfangen von Nachrichten (bzw. für RPC) zur Verfügung. Die übrige Programmierung erfolgt dann weitgehend analog zu imperativen Programmiersprachen, wobei einige wenige spezielle Statements zur Verfügung stehen.

Setzen des Testurteils

Ein solches Statement ist `setverdict`, wodurch der Testfall bewertet wird. Initial ist der Wert `none` und es können außerdem die folgenden Werte angenommen werden: `pass`, `inconc`, `fail`, `error`. Dabei ist die Rückkehr zu einem zuerst aufgeführten Zustand nicht möglich – es wird also immer der „schlechteste“ Zustand beibehalten. Der folgende Code implementiert einen einfachen Testfall:

```
1  testcase MeinTestfall() runs on Tester
2  {
3    gz_port.send(42);
4    gz_port.receive(42); // Erwartung. Blockiert falls
5                         // nicht "42" empfangen wird.
6    setverdict(pass);
7  }
```

Behandlung von Fehlerfällen

Dieses Beispiel berücksichtigt jedoch noch keine Fehlerfälle. Bei Ausbleiben der Antwort blockiert sogar die Testausführung. Um solche Situationen elegant zu behandeln, bietet TTCN-3 ein spezielles Konstrukt zur Beschreibung von Alternativen (`alt`). Der folgende Code zeigt eine verbesserte Version der obigen Implementierung:

```
1  testcase MeinTestfall() runs on Tester
2  {
3    timer antwortTimer;
4    gz_port.send(42);
5    antwortTimer.start(10.0); // Stoppuhr, 10 Sekunden
6
7    alt
8    {
9      // Behandlung bei korrekter Antwort
10     [] gz_port.receive(42)
11     {
12       setverdict(pass);
13       antwortTimer.stop;
14     }
```

```
15      // Behandlung bei unerwarteter Antwort
16      [] gz_port.receive
17      {
18        setverdict(fail, "Unerwartete Antwort");
19        antwortTimer.stop;
20      }
21
22      // Behandlung bei ausbleibender Antwort
23      [] antwortTimer.timeout
24      {
25        setverdict(fail, "Keine Antwort");
26      }
27    }
28  }
```

TTCN-3 Testsystem

TTCN-3 Testfälle sind abstrakt, denn es finden sich im gezeigten Beispiel keine Informationen darüber, wie die Kommunikation zwischen dem `Tester` und dem Testobjekt in Abbildung 16 tatsächlich implementiert ist. Möglich wäre z.B. die Übertragung via CAN oder Ethernet, ggfs. mit Hilfe zusätzlicher Protokolle darunterliegender Schichten (UDP, TCP).
Diese Details können im TTCN-3 Quelltext nicht beschrieben werden. Stattdessen ist es die Aufgabe des Testsystems einen passenden Adapter bereitzustellen und zu verwenden. Dieser wird im Rahmen von TTCN-3 als *SUT Adpater* bezeichnet und ist in Abbildung 17 zusammen mit den übrigen Diensten dargestellt.

SUT Adapter

Der in der Abbildung zentral aufgeführte Block *TTCN-3 Executable* behandelt die Ausführung der Testfälle und interagiert über zwei Schnittstellen mit den anderen Diensten im Testsystem. Das sind erstens das TTCN-3 Control Interface (TCI) [Std/ETS15c] für allgemeine Dienste aus dem Bereich der Testverwaltung und -kontrolle, und zweitens das TTCN-3 Runtime Interface (TRI) [Std/ETS15b], welches bezüglich dem SUT Adapter und damit zur Kommunikation mit dem Testobjekt relevant ist.
Darüber hinaus bietet das TRI Zugriff auf einen sogenannten *Platform Adapter*, der für Dienste benötigt wird, die spezifisch für die Plattform sind, auf der das Testsystem ausgeführt wird (z.B. Zugriff auf das Dateisystem).

Platform Adapter

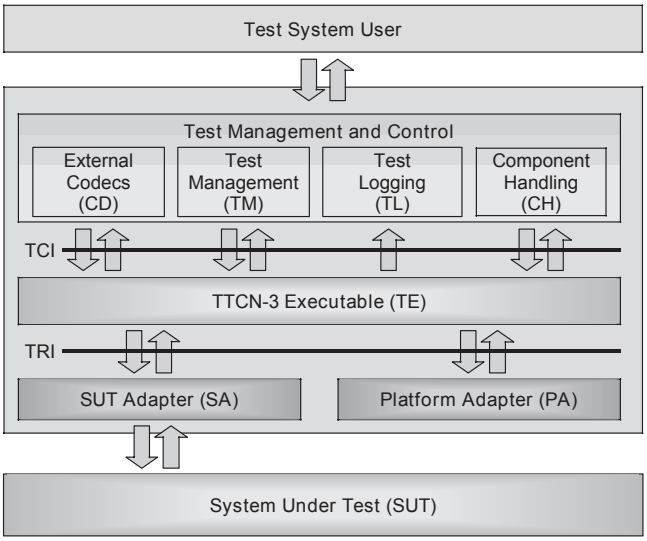


Abbildung 17: TTCN-3 Testsystem [Wilo5]

3.5.3 *Weitere DSLs*

Neben TTCN-3 finden sich im Umfeld eingebetteter Systeme weitere textuelle DSLs zur Beschreibung automatisiert ausführbarer Testfälle. Im Gegensatz zu TTCN-3 werden die hier vorgestellten Sprachen jeweils in Zusammenhang mit einem konkreten Testwerkzeug eingesetzt und lassen sich damit sehr klar entweder der System- oder der Quelltextebene zuordnen.

PROVEtech:TA Testsprache

In PROVEtech:TA wird eine Skriptsprache zur Testautomatisierung verwendet, welche in der offiziellen Dokumentation lediglich als „Testsprache“ bezeichnet wird [MBT14]. Dabei handelt es sich um die Sprache WinWrap Basic [Onl/WWB] der Firma Polar Engineering, die selbst keine testspezifischen Sprachkonstrukte besitzt.
Mit PROVEtech:TA wird eine testspezifische Klassenbibliothek bereitgestellt, welche die Funktionalität für den Zugriff auf die Testumgebung kapselt und bei der Auswertung unterstützen. Syntaktische Änderungen oder Erweiterungen der Sprache WinWrap Basic existieren jedoch nicht. Die PROVEtech:TA Testsprache ist somit als GPL anzusehen, zu der eine „interne“ DSL in Form einer Klassenbibliothek existiert.

CAPL

CAPL ist eine domänenspezifische Sprache von Vector Informatik. Ursprünglich stand das Akronym für „CAN Access Programming Language“ [Vec04]. Die heutige Bedeutung „Communication Access Programming Language“ geht auf die Unterstützung weiterer Bussysteme zurück.
Eine kurze Vorstellung von CAPL findet sich in [LM14]. Der Fokus liegt wie bei TTCN-3 auf kommunikationsorientierten Systemen, jedoch vor dem speziellen Hintergrund der Restbussimulation, und die Sprachkonzepte sind sehr verschieden. Der größte Unterschied besteht darin, dass CAPL auf *ereignisorientierter Programmierung* und damit auf einem speziellen Programmierparadigma basiert. Ein CAPL-Programm besteht aus Programmblöcken, die Reaktionen auf Ereignisse beschreiben. Tritt das zugeordnete Ereignis ein, beispielsweise der Empfang einer Botschaft, wird der zugehörige Programmblock ausgeführt. Dieser Block ist wiederum konventionell imperativ implementiert.

ereignisorientierte Programmierung

CAPL zeichnet sich darüber hinaus durch sehr einfachen Zugriff auf Testgrößen der Systemebene aus. Diese können durch den in einer Datenbank[11] verwendeten Namen direkt referenziert werden, es wird also kein Laufzeitobjekt wie z.B. eine Zeichenkette verwendet. Die Referenz kann deswegen bereits zum Zeitpunkt der Kompilierung aufgelöst werden, was die Robustheit der Implementierung erhöht.

CCDL

CCDL steht für Check Case Definition Language und ist eine Skriptsprache, welche in Verbindung mit Werkzeugen der Razorcat Development GmbH verwendet wird. CCDL ist „geeignet für Systemtests, bei welchen das Testobjekt als Black-Box mit definierten Eingängen und Ausgängen angesehen wird“ [Wit10]. Die Sprache stellt ein Gegenstück zum ebenfalls von Razorcat entwickelten Unit-Test Werkzeug Tessy dar (vgl. Abschnitt 3.4.2).
CCDL zeichnet sich wie CAPL dadurch aus, dass Referenzen auf Testgrößen bereits zur Kompilierungszeit aufgelöst werden. CCDL vereinfacht aber das Setzen von Triggern und das Warten auf Bedingungen durch spezielle Syntax. Darüber hinaus unterstützt CCDL physikalische Einheiten auf Sprachebene.

11 Als Datenbank kommt u. a. DBC (data base CAN, ein Dateiformat zum Speichern von Informationen über die auf einem CAN Bus ausgetauschten Daten) in Frage

TESLA

Die Sprache TESLA (Test Specification Language) ist im Rahmen eines Forschungsprojekts der Firma ABB entstanden. Die Sprache wurde in [WFS+12] vorgestellt und wird seitdem im Rahmen ausgewählter Projekte bei ABB eingesetzt. Diese Sprache hat ebenfalls die Automation von Tests auf der Systemebene im Fokus.

PRACTICE

Die Skriptsrapche PRACTICE [Lau15a] wird in Zusammenhang mit Debug- und Trace-Hardware der Firma Lauterbach GmbH verwendet. Sie dient in erster Linie der Automatisierung wiederholt anfallender Tätigkeiten bei der Entwicklung und dem Debuggen von Software auf der Zielplattform mit den entsprechenden Hardwarewerkzeugen (Übertragung des binären Programmcodes etc.), kann aber auch zur Testautomatisierung verwendet werden [Lau15b]. Die Sprache bietet nur einen geringen Funktionsumfang und ist sehr stark auf die Werkzeuge der Firma Lauterbach ausgerichtet.

Austauschformate

Neben den diversen Programmiersprachen gibt es weitere Beschreibungsformate für Testfälle, die dem Austausch von Testdaten und -abläufen zwischen Testwerkzeugen dienen, zur direkten Verwendung durch einen Programmierer aus syntaktischen Gründen jedoch nicht geeignet sind. Derartige Austauschformate werden an dieser Stelle erwähnt, um sich klar davon Abzugrenzen. Beispiele sind der ASAM ATX Standard [Std/ASA12] sowie das in [GFK+08] beschriebene Austauschformat TestML. Beide Formate wurden für den Austausch zwischen Testwerkzeugen der Systemebene mit dem Ziel entworfen, Testfälle auf anderen Testsystemen wiederverwenden zu können. Dieses Ziel kann durch eine standardisierte Abstraktionsschicht ebenfalls erreicht werden (z.B. ASAM XIL API, siehe Abschnitt 3.3.4).

3.6 ÜBERBLICK ÜBER DIE TESTLÖSUNGEN

In den Abschnitten 3.3, 3.4 und 3.5 wurden aktuelle Lösungen zur Testautomatisierung auf der Zielplattform vorgestellt. Alle Lösungen fokussieren sich auf eine der zwei identifizierten Ebenen (Systemebene oder Quelltextebene). In Tabelle 3 sind sie zum einfachen

Vergleich entsprechend dieser Fokussierung in zwei Gruppen eingeteilt. Die Eignung für die jeweilige Ebene ist zudem noch einmal in den ersten beiden Spalten dargestellt.

Eignung für System- und Quelltextebene

Die Lösungen mit Fokus auf der Quelltextebene sind für Tests auf Systemebene gänzlich ungeeignet, da sie die entsprechenden Testgrößen vollständig ignorieren.
In der anderen Gruppe ergibt sich ein nur leicht abweichendes Bild. Keine Lösung bietet den für Unit-Tests notwendigen Zugriff auf Softwarefunktionen. Nur TTCN-3 geht mit der Unterstützung von RPC einen Schritt in diese Richtung, erfordert dazu aber spezielle Software im Testobjekt. Die meisten Lösungen ermöglichen zwar den Zugriff auf den internen Speicher der Prozessorhardware, was im Prinzip dem Zugriff auf globale Variablen der Software entspricht, jedoch wird das in allen Fällen über spezielle Protokolle zur Messung und Kalibrierung realisiert (vgl. ECUM/ECUC der ASAM XIL API in Abschnitt 3.3.4). Debug- und Trace-Hardware wird von keiner dieser Testlösung direkt unterstützt. Die Testauswertung anhand einer Aufzeichnung des Program- oder Data-Trace ist daher ebenfalls nicht möglich.

Weitere Differenzierung

Die übrigen Spalten gehen bereits auf einige der zusätzlichen Anforderungen an eine Testlösung ein, welche in Abschnitt 4.2 genauer erklärt werden. Spalte 3 gibt eine Einschätzung, ob die Software beim Test auf der Zielplattform ausgeführt werden kann. Spalte 4 betrachtet, ob dabei unerwünschte Seiteneffekte, die das Testergebnis beeinflussen könnten, vermieden werden. Hier zeigen insbesondere die Werkzeuge der Quelltextebene Defizite, da sie großteils auf Codeinstrumentierung basieren. In Spalte 5 geht es um die Mächtigkeit der verfügbaren Beschreibungsmechanismen für Testabläufe und Ausdrücke. Diese ist eingeschränkt, wenn Abläufe nur über vordefinierte Dialogfelder erstellt werden können. In Spalte 6 wird bewertet, inwiefern die jeweilige Lösung zu einem benutzerfreundlichen Gesamtprodukt integriert ist. In der letzten Spalte ist gezeigt, ob grafisch Modellierung eine Rolle spielt.

3.7 VERWENDETE MDSD WERKZEUGE

Die Umsetzung in Teil III erfolgt mit Hilfe von zwei MDSD Werkzeugen, welche eine Integration in die Eclipse Entwicklungsumgebung (IDE) erleichtern, und im Folgenden kurz vorgestellt werden. Dieser Abschnitt hat nicht das Ziel einen allgemeinen Überblick über die Werkzeuglandschaft zu bieten. Allein im Umfeld der Eclipse IDE gibt es eine große Zahl weiterer MDSD Werkzeuge. Deren

Tabelle 3: Vergleich von Lösungen zur Testautomatisierung bezüglich Eingebetteter Software auf der Zielplattform

		Quelltextebene	Systemebene	Auf Zielplattform	Minimierung der Seiteneffekte	Mächtigkeit	Werkzeugintegration	Grafische Modellierung
Systemebene	Spezifische HiL APIs	-	++	++	++	++	-	-
	ASAM XIL API (Standard)	ø[1]	++	++	++	++	-	-
	TTCN-3 (Standard)	-	ø[2]	+	++	++	-	+
	Provetech:TA (MBtech)	ø[1]	+	++	++	++	ø	-
	AutomationDesk (dSPACE)	ø[1]	+	++	++	++	+	+
	LABCAR-AUTOMATION (ETAS)	ø[1]	+	++	++	++	+	+
	TestStand (National Instruments)	ø[1]	+	++	++	+	++	++
	CANoe / CAPL (Vector Informatik)	ø[1]	ø[2]	++	++	++	+	ø
	TPT (PikeTec)	ø[1]	+	+	++	+	++	++
	TESLA (ABB)	-	ø[2]	+	ø[5]	ø	++	-
	CCDL (Razorcat)	-	ø[3]	++	++	+	++	-
Quelltextebene	Unit Test Frameworks	++	-	ø	-[5]	++	-	-
	Spezifische Debug APIs	++	-	++	++	++	-	-
	Tessy (Razorcat)	+	-	+	ø[4]	ø	++	ø
	VectorCAST (Vector Software)	+	-	+	ø[4]	ø	++	ø
	C/C++test (ParaSoft)	+	-	+	ø[4]	ø	++	ø
	CTestIt! (Cosmic Software)	+	-	++	++	ø	++	ø
	TestIDEA (iSYSTEM)	+	-	++	++	ø	++	ø
	PRACTICE (Lauterbach)	+	-	++	++	ø	-	-

++ Vollst. abgedeckt + Großt. abgedeckt ø Tw. abgedeckt - Nicht abgedeckt
[1] Keine entfernter Funktionsaufruf (RPC) [2] Keine elektrischen Schnittstellen
[3] Keine Analyse von Analogsignalen [4] Tw. Instrumentierung [5] Instrumentierung

Betrachtung ist nicht notwendig, da neben den konkret eingesetzten Werkzeugen nur die grundlegenden Konzepte von MDSD relevant sind, welche bereits in Abschnitt 2.4 vorgestellt wurden.

3.7.1 *Eclipse Modeling Framework*

Das Eclipse Modeling Framework dient der Metamodellierung in der Eclipse IDE. Das EMF unterstützt dabei durch Modellierungswerkzeuge und einen Codegenerator. Auf alle Aspekte des EMFs einzugehen würde den Rahmen sprengen, daher wird an dieser Stelle nur auf das grundlegende Prinzip zur Codegenerierung eingegangen. Für weitere Informationen sei auf [Ste09] verwiesen.

Ecore-Modell

Zentrales Element ist ein durch den Anwender erstelltes *Ecore-Modell* (Dateiendung „.ecore“), welches in Abbildung 18 auf der Meta-Ebene M2 zu finden ist. Es ist das Metamodell zu den Modellen (.mymdl), um die es dem Endanwender des entwickelten Werkzeugs letztendlich geht.

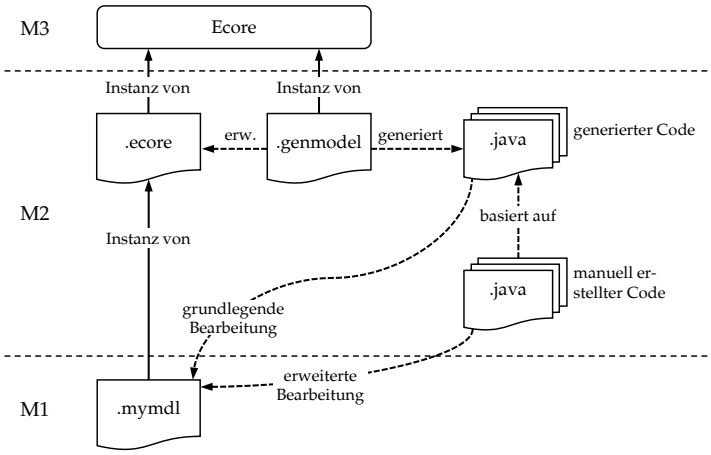


Abbildung 18: Codegenerierung mit dem EMF

Generator-Modell

Zur Codegenerierung benötigt das EMF ein weiteres Modell, das *Generator-Modell* (.genmodel). Es erweitert das Ecore-Modell um spezifische Aspekte zur Codegenerierung wie Ausgabeverzeichnisse für generierte Quelltexte, Java Paketnamen, Parameter zur (De-)Serialisierung der Endanwendermodelle und Weiteres.

Der EMF Codegenerator erzeugt Java Quelltext, der eine Datenstruktur und einen rudimentären baumbasierten Editor für die Endanwendermodelle implementiert. Der für die Datenstruktur generierte Code bietet außerdem einfache Schnittstellen zur program-

matischen Arbeit mit Modellinstanzen. Wurde das Ecore-Modell außerdem mit Randbedingungen in OCL (Object Constraint Language) annotiert, werden diese ebenfalls bereits durch den generierten Code geprüft.
Für eine sinnvolle Nutzung der Endanwendermodelle – beispielsweise wenn aus ihnen erneut Quelltext generiert werden soll – muss der durch das EMF generierte Code modifiziert und erweitert werden. Dieser Teil ist nicht automatisierbar (vgl. Abschnitt 2.4.3).

3.7.2 *Xtext*

Xtext ist ein Framework zur Erstellung domänenspezifischer Sprachen in Eclipse, das erstmals in [EV06] vorgestellt wurde. Auch hier werden aus Modellen einer Metaebene Java Quelltexte generiert, welche die Realisierung eigener Werkzeuge auf Basis von Eclipse erleichtern. Xtext basiert diesbezüglich auf dem EMF. Der in Abbildung 19 gezeigte Ablauf zur Codegenerierung ähnelt daher dem zuvor geschilderten Prinzip für das EMF.

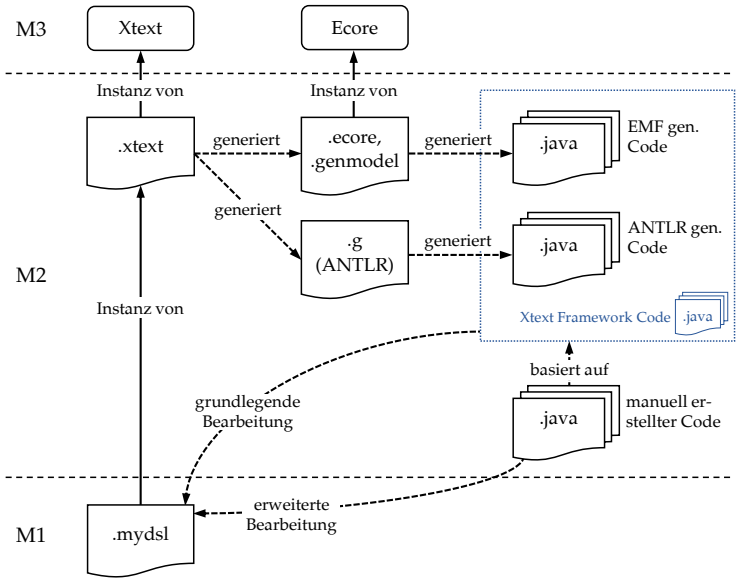


Abbildung 19: Codegenerierung mit Xtext

Zentrales Element ist die durch den Anwender erstellte Grammatik der Sprache (Dateiendung .xtext). Hierdurch wird die Syntax einer DSL (.mydsl) beschrieben. Anhand zusätzlicher Annotationen in der Grammatik wird eine Eingabe für den Parsergenerator ANTLR

(.g) sowie ein Ecore-Modell des ASTs (.ecore und .genmodel) generiert (vgl. Abschnitt 2.3.5).
Das EMF generiert, wie schon im letzten Abschnitt erklärt, den Java Quelltext einer Datenstruktur des ASTs, ANTLR generiert den Quelltext eines Parsers. Xtext kapselt, wie ebenfalls in Abbildung 19 dargestellt, alle generierten Quellen durch Framework Code. Dadurch stehen erneut eine Reihe von Eclipse Plugins zur Verfügung, welche grundlegende Features zur Verwendung der beschriebenen DSL bereitstellen. Die zentrale Komponente ist erneut ein Editor, diesmal textbasiert, der unter Anderem Syntaxhervorhebung, Code-Navigation und Autovervollständigung unterstützt. Auch hier ist zur sinnvollen Nutzung der entworfenen DSL manuelle Entwicklungsarbeit auf Quelltextebene zu leisten.

Teil II

LÖSUNGSANSATZ

Im zweiten Teil wird als Erstes zusammengefasst, inwiefern aktuelle Möglichkeiten für Test und Analyse von eingebetteter Software nicht optimal sind. Dazu wird anhand des V-Modells aufgezeigt, an welchen Stellen der Testprozess noch nicht durchgängig ist. Es werden die Anforderungen zum Schließen dieser Lücke abgeleitet, und anschließend ein Konzept für die Lösung erstellt.

4

DURCHGÄNGIGER TEST AUF DER ZIELPLATTFORM

Diese Arbeit hat das Ziel die Durchgängigkeit des Testprozesses für eingebettete Software zu verbessern (vgl. Kapitel 1). Damit besteht eine Beziehung zu anderen Arbeiten aus dem Bereich des Testens eingebetteter Systeme, in denen Durchgängigkeit ebenfalls eine wesentliche Motivation ist. Dabei muss jedoch zwischen zwei unterschiedlichen Aspekten unterschieden werden: häufig steht die Integration der Testmethoden MiL, SiL, PiL und HiL im Mittelpunkt. In dieser Arbeit geht es hingegen um die Integration quelltextnaher und systembezogener Tests. Die beiden Aspekte stehen jedoch in einem Zusammenhang, was im folgenden Abschnitt zunächst verdeutlicht wird.

4.1 EINORDNUNG IM TESTPROZESS

Bei der Funktionsentwicklung eingebetteter Systeme ist eine frühzeitige und entwurfsbegleitende Absicherung wesentlich [SA08]. Das hat zwei wichtige Gründe: Erstens steigen die Kosten zur Fehlerbeseitigung mit dem Zeitpunkt der Entdeckung stark an. Dieser Zusammenhang wird oft als „Zehnerregel“ (10er-Regel, Rule of Ten) bezeichnet. Zweitens leidet die Produktqualität, wenn mit Testaktivitäten erst spät im Entwicklungsprozess begonnen wird. Denn Verzögerungen in der Produktentwicklung haben dann zur Folge, dass für den Test weniger Zeit zur Verfügung steht – zumindest wenn die Deadline des Gesamtprojekts fix ist [Chr08].

4.1.1 *Durchgängigkeit im V-Modell*

In Abschnitt 3.3 wurden die XiL Testmethoden als Maßnahme zur frühzeitigen Absicherung bereits vorgestellt. Bezüglich der Anwendungsfunktion decken sie den Testprozess bereits sehr gut ab. Das lässt sich am besten anhand der allgemeinen Darstellung in Form eines „V“ zeigen, welche in Abschnitt 2.1.1 eingeführt wurde. In Abbildung 20 sind dort nun die XiL Testmethoden verortet.
Wie zu erwarten liegen sie auf der Systemebene. Das passt zu der Feststellung, dass man für den Test der Anwendungsfunkti-

Durchgängigkeit auf Systemebene

on kaum auf die Quelltextebene vordringen muss, sondern das eingebettete System besser als Black-Box betrachtet. Hier ist der Testprozess durchgängig: so früh wie möglich erfolgen Verifikation und Validierung durch MiL Tests. Mit fortschreitendem Systementwurf (Verteilung auf Steuergeräte, Scheduling etc.) wird zu SiL Tests übergegangen. In diesen Phasen werden ein ECU Muster oder die Zielplattform noch nicht benötigt. Wenn die entsprechende Hardware verfügbar ist, wird mit PiL und HiL Tests fortgesetzt. Zu jedem Zeitpunkt kann zu einer „früheren" Testmethode zurückgekehrt werden. Standards wie die ASAM XIL API helfen zudem bei der Wiederverwendung von Testimplementierungen.

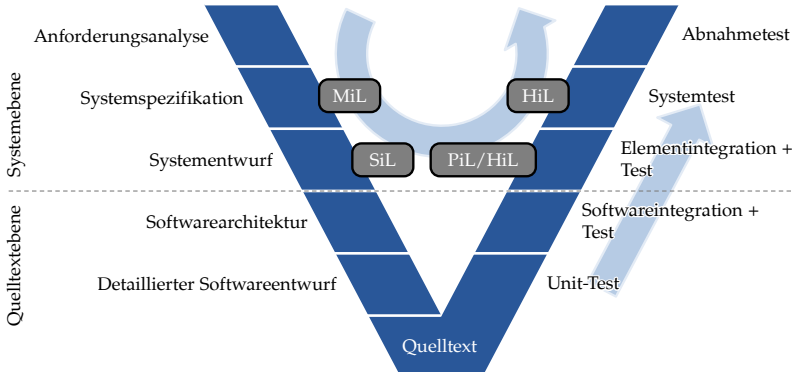


Abbildung 20: Durchgängigkeit im Testprozess

Bedeutung der Quelltextebene

Bei dieser Fokussierung auf die Anwendungsfunktion darf man aber nicht vergessen, dass die Entwicklung der Zielplattform mit in der Schleife hängt, und zwar auch dann, wenn sie an andere Parteien ausgelagert ist. Das umfasst neben dem Hardwareentwurf, der in dieser Arbeit nicht betrachtet wird, die Softwarearchitektur sowie Implementierung und Integration von Basissoftwarekomponenten. Die oben beschriebene Durchgängigkeit besteht nur solange, wie die Ebenen scharf voneinander getrennt sind. Standards wie AUTOSAR tragen einen erheblichen Teil dazu bei (vgl. Abschnitt 2.1.4).

Innovative Anwendungsfunktionen erfordern jedoch zunehmend Anpassungen und mehr Flexibilität auf unteren Softwareschichten. Beispiele aus dem Automobilbereich sind die Car-To-X Kommunikation, das Online-Update von ECUs, Big-Data Anwendungen oder das autonome Fahren. Weil diese Funktionen teilweise wettbewerbsrelevant sind, ist die Bereitschaft der OEMs zur Entwicklung

eines gemeinsamen Standards geringer. Ohne einen solchen ist die scharfe Trennung der Anwendungsfunktion von der Softwarearchitektur jedoch nicht mehr gegeben.
Vor diesem Hintergrund wird auch für OEMs die Durchgängigkeit zwischen den Ebenen wichtig, dargestellt durch den Pfeil rechts. Für Hersteller von Steuergeräten spielt dieser Übergang schon länger eine zentrale Rolle. Dort werden die Testaktivitäten jedoch stark isoliert voneinander betrachtet. Mit zunehmend kürzeren Zyklen zur Produktentwicklung wird es in Zukunft jedoch für alle Beteiligten immer wichtiger, dass sich der Übergang fließend gestaltet.

Durchgängigkeit zwischen den Ebenen

4.1.2 *Lücke im Stand der Technik*

Derzeit ist die Implementierung eines durchgängigen Testprozesses, welcher sowohl die System- als auch die Quelltextebene abdeckt, kaum möglich: es ist schlicht keine Testlösung vorhanden, welche die Anforderungen beider Ebenen erfüllt (vgl. Tabelle 3). Ein Bruch in der Werkzeugkette lässt sich derzeit nicht vermeiden. Durchgängigkeit auf der Systemebene unterstützt von den untersuchten Lösungen die ASAM XIL API am besten. In dieser Arbeit wird für eine insgesamt durchgängige Lösung daher darauf aufgebaut.

4.2 ANFORDERUNGEN AN EINE TESTLÖSUNG

Aus der Sicht des Testentwicklers stellt sich die beschriebene Durchgängigkeit zwischen den Ebenen wie in Abbildung 21 dar. Eine entsprechende Testlösung muss alle Aspekte abdecken, die im Laufe der Entwicklung in den Fokus eines Testentwicklers geraten. Das umfasst den Quelltext der ausgeführten Software (1), die Netzwerkkommunikation (2), sowie das physikalische Verhalten an elektrischen Schnittstellen (3) und der simulierten Umgebung (4).
Neben dieser Grundanforderung A1 (Durchgängigkeit) existiert eine Reihe weiterer Anforderungen, welche im Folgenden strukturiert aufgelistet werden.

A1 *Durchgängigkeit*: Diese Anforderung umfasst das zentrale Alleinstellungsmerkmal der neuen Testlösung, welches zu Beginn dieses Kapitels bereits genau beschrieben wurde, also die Abdeckung von Tests auf System- und Quelltextebene, sowie eine Anknüpfung an die ASAM XIL API.

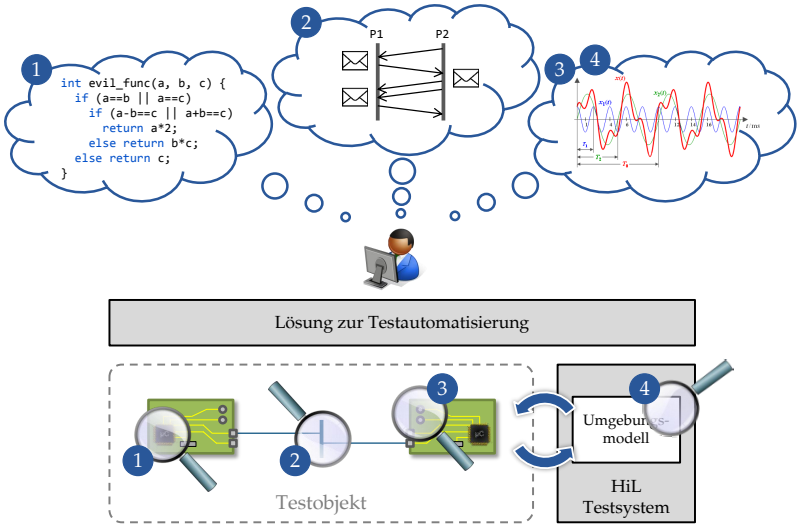


Abbildung 21: Durchgängigkeit aus Sicht des Testentwicklers

A2 *Funktionale Anforderungen*: Die funktionalen Anforderungen ergeben sich aus den Anwendungsszenarien und Benutzeranforderungen an die Testlösung. Mögliche Anwendungsszenarien wurden bereits ausreichend beschrieben. Benutzer sind an erster Stelle Testentwickler, aber auch das Management soll von der Lösung profitieren.

A2.1 *Vollautomatisierter Test*: Zentrale Aufgabe der Testlösung ist der vollautomatisierte Test eingebetteter Software auf der Zielplattform. Hierzu ist die Implementierung von Testfällen notwendig. Diesbezüglich ergeben sich weitere Anforderungen, welche unter A3 separat aufgeführt werden.

A2.2 *Interaktiver Test*: Zur entwicklungsbegleitenden Funktionserprobung muss auch der interaktive Test unterstützt werden. Testdaten müssen dazu im visualisiert bzw. durch den Testentwickler manipuliert werden können. Dazu ist eine abgeschwächte Form der unter A5 beschriebenen Echtzeitfähigkeit notwendig (beschränkt auf die menschliche Wahrnehmung).

A2.3 *Werkzeugintegration*: Die Testlösung soll eine hohe Werkzeugintegration bieten und die genannten Aufgaben durch eine grafische Oberfläche unterstützen. In diesem Zusammenhang sind weitere häufig durchzuführende

Tätigkeiten zu automatisieren. Das umfasst das Anlegen von Projekten und Dokumenten anhand von Vorlagen, die Konfiguration der Testumgebung, sowie die Konsistenz- bzw. Gültigkeitsprüfung von Testimplementierungen.

A2.4 *Automatische Reports*: Zu Testläufen müssen Reports automatisch generiert werden können.

A2.5 *Test Management Schnittstelle*: Die Testlösung muss sich mit gängiger Software für das Test Management verbinden lassen. Hier gibt es bereits eine große Palette an Werkzeugen, welche häufig in Verbindung mit Lösungen zur Versionsverwaltung und dem Bugtracking stehen.[1]

A3 *Testimplementierung*: Die Testlösung muss die Implementierung vollautomatisch ausführbarer Testfälle unterstützen. Diesbezüglich ergeben sich die folgenden Anforderungen:

A3.1 *Kein Paradigmenwechsel*: Eine neue Lösung darf von Testentwicklern keinen Paradigmenmwechsel fordern. Die jeweils gewohnte Sichtweise auf das Testobjekt soll beibehalten werden. Auf Systemebene entspricht das einem externen Blick auf das System, auf Quelltextebene einem internem Blick auf den Quelltext. Der Wechsel zwischen diesen beiden Blickwinkeln muss nahtlos sein, insbesondere muss es möglich sein, beide Positionen zugleich einzunehmen.

A3.2 *Transparente Testgrößen*: Der Zugriff auf und die Verwendung von Testgrößen beider Ebenen (vgl. Tabelle 2 auf Seite 41) muss transparent sein. In einer Testimplementierung müssen entsprechende Größen direkt miteinander verglichen, zugewiesen und verrechnet werden können, soweit sie kompatibel sind.

A3.3 *Typsicherheit*: Fehler in der Implementierung durch Verwendung falscher Datentypen soll soweit möglich verhindert werden. Das ist vor dem Hintergrund transparenter Testgrößen (s.o.) besonders wichtig.

A3.4 *Mächtigkeit*: Komplexe Abläufe müssen erfasst werden können. Dazu muss das Mittel zur Implementierung ein hohes Maß an Flexibilität bieten.

1 z.B. Subversion, Git, Team Foundation Server etc.

A3.5 *Abstraktion*: Implementierungen sollen einen möglichst hohen Abstraktionsgrad aufweisen. Details, welche nicht direkt der Beschreibung von testspezifischem Verhalten dienen, dürfen in der Implementierung nicht auftauchen, z.B. die Ansteuerung der Zugriffshardware.

A4 *Tool-Qualifizierung*: Die Testlösung muss zur Absicherung sicherheitsrelevanter Funktionen verwendet werden können. Das setzt eine Qualifizierung der Lösung entsprechend den jeweils geltenden Standards für sicherheitsrelevante Anwendungen voraus (z.B. ISO 26262, DO-178C). Das Durchführen einer Tool-Qualifizierung ist im Rahmen dieser Arbeit nicht möglich, es können jedoch einige Anforderungen genannt werden, die damit in Zusammenhang stehen:

A4.1 *Test auf der Zielplattform*: Tests werden auf der realen Zielplattform ausgeführt. Das erleichtert Anwendern den Nachweis, dass die Testergebnisse der Realität entsprechen.

A4.2 *Keine Codeinstrumentierung*: Auf Codeinstrumentierung wird verzichtet. Das erleichtert den Nachweis, dass durch die Testlösung das Verhalten nicht verfälscht wird.

A5 *Echtzeitfähigkeit*: Der Test von Echtzeitanforderungen muss unterstützt werden. Die dafür nötige zeitliche Auflösung lässt sich nicht ohne weiteres benennen. Auf der Systemebene ist eine Millisekunde ein häufig genannter Wert. Je nach Anwendung kann dieser jedoch auch um einige Größenordnungen kleiner sein, beispielsweise unter einer Mikrosekunde bei der Darstellung von Transienten [Koe14]. Auf Quelltextebene sind z.B. die Zeit für einen Kontextwechsel im Betriebssystem (unter einer bis wenige Mikrosekunden, je nach Betriebssystem) und die Ausführungszeit einer Softwarefunktion (einige Nanosekunden) relevant. Es muss daher zwischen drei Formen der Echtzeitfähigkeit unterschieden werden:

A5.1 *Offline-Prüfung von Zeitverhalten*: Die Prüfung von Zeitverhalten auf Basis von Echtzeit-Traces muss unterstützt werden. Die erreichbare Genauigkeit entspricht der Auflösung der im Trace vorhandenen Zeitstempel und wird durch die verwendete Zugriffshardware vorgegeben. Das Zusammenführen von Traces aus mehreren Quellen auf eine gemeinsame Zeitbasis muss möglich sein. Ohne spe-

zielle Hardwareunterstützung ist bezüglich dem Zusammenführen eine Genauigkeit von 1 bis 10 Millisekunden realistisch.

A5.2 *Echtzeitfähige Stimuli*: Das Abspielen von Stimuli in Echtzeit muss unterstützt werden. Die erreichbare Genauigkeit wird durch die verwendete Zugriffshardware vorgegeben. Eine Synchronisation unabhängiger Zugriffshardware muss auf einer gemeinsamen Zeitbasis möglich sein. Ohne spezielle Hardwareunterstützung ist bezüglich der Synchronisation eine Genauigkeit von 1 bis 10 Millisekunden realistisch.

A5.3 *Closed-Loop Echtzeitfähigkeit*: Die Echtzeitfähige Reaktion auf Ausgaben des Testobjekts ist zur Umgebungssimulation im Rahmen von HiL-Test erforderlich. Das entsprechende Verhalten soll im Umgebungsmodell beschrieben werden. Im Rahmen des Testablaufs sind vergleichbar kurze Reaktionszeiten nicht nötig, von einer Ausnahme abgesehen: als Sicherheitsmaßnahme müssen Grenzwerte im Voraus konfiguriert und in Echtzeit überwacht werden können, z.B. zur Notabschaltung. Bezüglich dieser Reaktionszeit ist wieder 1 bis 10 Millisekunde realistisch.

4.3 KONZEPT EINER NEUEN TESTLÖSUNG

Die im Rahmen dieser Arbeit realisierte Testlösung heißt ETSpec Framework. ETSpec steht für „*E*mbedded Software *T*est *Spec*ification“. Das Framework besteht grob aus drei Bausteinen. Erstens einer formalen Sprache zur Implementierung von Testfällen (ETSpec Sprache), zweitens der Plattform zur Ausführung der entsprechend spezifizierten Testfälle (ETSpec Laufzeitumgebung), und drittens einer Sammlung von Softwarewerkzeugen. Im Folgenden wird zuerst die Gesamtarchitektur des Frameworks beschrieben, bevor in Abschnitt 4.3.2 auf das Konzept zur ETSpec Sprache eingegangen wird.

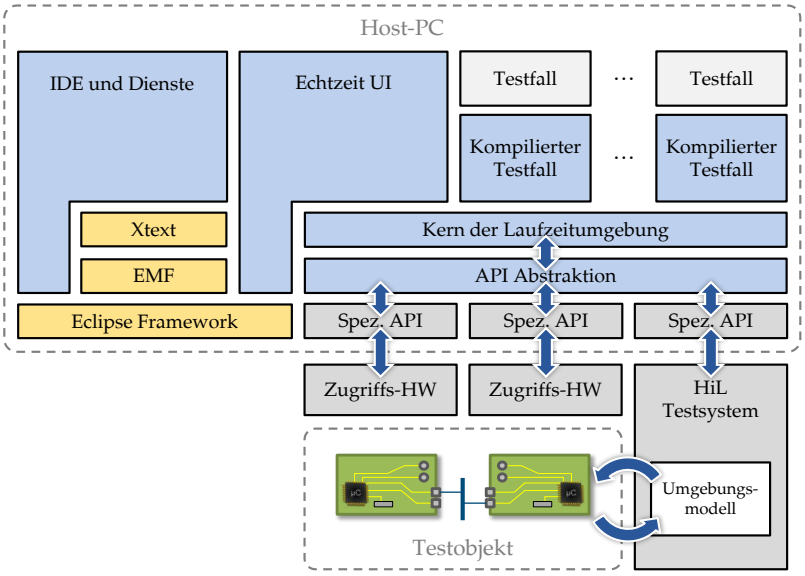


Abbildung 22: Blockdiagramm der Architektur des ETSpec Frameworks

4.3.1 *Softwarearchitektur*

Ein Blockdiagramm der Gesamtarchitektur des ETSpec Frameworks ist in Abbildung 22 dargestellt. Zur Erklärung geht man am besten von Abbildung 21 aus, in welcher die Anforderung A1 (Durchgängigkeit) aus der Sicht des Testentwicklers darstellt wurde.

Integration Zugriffshardware

Bezüglich Anforderung A1 sind als erstes die notwendigen Zugriffspunkte im Framework verfügbar zu machen, d.h. es muss die jeweilige Zugriffshardware in das Framework integriert werden.

API Abstraktion

Das lässt sich auf Basis der jeweils spezifischen APIs erreichen, indem darüber eine Abstraktionsschicht eingefügt wird. Dieses Prinzip ist in Abbildung 22 gut zu erkennen (Blöcke *Zugriffs-HW*, *Spez. API* und *API Abstraktion*). Zur Anbindung eines HiL Testsystems ist anstelle einer herstellerspezifischen API natürlich auch die Verwendung der ASAM XIL API vorgesehen.
An dieser Stelle fällt auf, dass die Zugriffshardware getrennt von einem HiL Testsystem dargestellt ist, obwohl sie auch als Teil eines HiL Testsystems angesehen werden kann. Die separate Betrachtung ist erforderlich, weil ein HiL Testsystem nicht für jeden Test notwen-

dig ist und stattdessen die Verwendung „einfacher“ Zugriffshardware, z.B. eines analogen Messgeräts, gewünscht sein kann. Auch ist Debug- und Trace Hardware bisher nicht in HiL Testsysteme integriert.

Tool-Qualifizierung

Vor diesem Hintergrund sind die wesentlichen Voraussetzungen für die Anforderungen unter A4 (Tool-Qualifizierung) bereits erfüllt. In allen Fällen wird die Zielplattform zur Ausführung der getesteten Software verwendet, und durch die Verwendung von Debug- und Trace-Hardware wird Codeinstrumentierung vermieden. Erforderlich ist dazu natürlich, dass die Prozessorhardware entsprechende Schnittstellen bietet.

Echtzeitfähigkeit

Auch die notwendigen Voraussetzungen bezüglich A5 (Echtzeitfähigkeit) sind gegeben, wobei eine Erfüllung stark von der verfügbaren Zugriffshardware und der konkreten Implementierung abhängt. Zur Offline-Prüfung des Zeitverhaltens müssen hinreichend genaue Zeitstempel generiert werden, und bei der Synchronisation spielen Latenzen zwischen dem Host-PC und der Zugriffshardware eine entscheidende Rolle. Die prototypische Implementierung der Laufzeitumgebung, welche in Kapitel 7 vorgestellt wird, basiert auf Java, und wird auch aufgrund weiterer Seiteneffekte wie der automatischen Speicherbereinigung nicht alle Echtzeitanforderungen erfüllen können.

Automatisierter und Interaktiver Test

Die weitere Architektur in Abbildung 22 erklärt sich anhand der funktionalen Anforderungen (A2). Auf die Schicht zur API Abstraktion setzt der Kern der Laufzeitumgebung auf. Sie stellt die Grundlage zur automatischen Ausführung von Testfällen dar und ergibt sich dementsprechend aus A2.1 (Vollautomatisierter Test). Dieser Kernkomponente obliegt die Kontrolle der Testausführung. Gleichzeitig ist sie für die Ansteuerung der jeweils richtigen Zugriffshardware verantwortlich, das heißt sie übernimmt das notwendige „Mapping“ der im Testfall referenzierten Testgrößen. Darüber hinaus bietet sie weitere Schnittstellen für Testfälle, z.B. zum Entgegennehmen von Logmeldungen.
Konsequenterweise setzen die Testfälle auf die Laufzeitumgebung auf. In der Abbildung ist hier bereits dargestellt, dass Testfälle vor

der Ausführung kompiliert werden. Dieser Mechanismus wird in Kapitel 5 genauer erklärt. Eine Alternative wäre die Interpretation des Quelltextes, welcher die Erfüllbarkeit der Echtzeitanforderungen jedoch zunichtemachen würde.
Zur Erfüllung der Anforderung A2.2 (Interaktiver Test) wird zudem eine Benutzerschnittstelle (UI) zur Echtzeit-Visualisierung (*Echtzeit UI*) benötigt. Diese muss ebenfalls mit der Zugriffshardware kommunizieren und setzt daher auch auf die Laufzeitumgebung auf.

Echtzeit UI

IDE und Dienste

Die verbleibende Komponente „IDE und Dienste“ aus Abbildung 22 ist weniger spezifisch. Die IDE umfasst das UI zur Projektverwaltung, Modellierung und Testimplementierung, dem Anstoßen automatisierter Testläufe etc. Sie geht aus der Anforderung A2.3 (Werkzeugintegration) hervor. Die IDE basiert auf dem Eclipse Framework, welches eine ideale Grundlage bereitstellt, da es einen Großteil der benötigten Basisfunktionalität (Fenstermanagement, Verwaltung von Projektdateien etc.) direkt zur Verfügung steht.
Hinter den „Diensten“ stecken die funktionalen Anforderungen A2.4 (Automatische Reports) und A2.5 (Test Management Schnittstelle), das heißt sie umfassen einen Report-Generator sowie Schnittstellen zur Anbindung Werkzeuge Dritter, z.B. für den Zugriff auf Testergebnisse. Zu den Diensten wird darüber hinaus alle von den übrigen Komponenten der Architektur gemeinsam genutzte Funktionalität gezählt. Hierzu gehört insbesondere ein Verzeichnis aller verfügbaren Testgrößen mit ihren Eigenschaften (siehe Abschnitt 5.3.1). Ebenfalls zu den Diensten gezählt werden der Compiler für Testfälle, sowie statische Codeanalyse zur Konsistenz- und Gültigkeitsprüfung.
IDE und Dienste basieren zu großen Teilen auf EMF und Xtext, welche in Abschnitt 3.7 bereits vorgestellt wurden.

4.3.2 *Testimplementierung*

Auf die meisten Anforderungen wurde bereits eingegangen, lediglich die Punkte unter A3 (Testimplementierung) wurden bisher nicht erwähnt. Dort sind die Anforderungen A3.4 (Mächtigkeit) und A3.5 (Abstraktion) am wichtigsten, weswegen der in dieser Arbeit verfolgte Ansatz anhand dieser Punkte ausgewählt wird. Es lassen sich die folgenden Möglichkeiten zur Testimplementierung identifizieren:

- *Formularbasierte Oberflächen*: Der Anwender kann einzelne Schritte des Testablaufs in Form einer Liste oder Tabelle zusammenstellen. Er ist dabei strikt an die Möglichkeiten der GUI gebunden. Komplexe Ausdrücke und verschachtelte Abläufe lassen sich auf diese Weise nur mit Einschränkungen beschreiben, dafür ist eine sehr abstrakte Implementierung möglich.

- *Grafische DSLs*: Der Anwender beschreibt den Testablauf grafisch mit Hilfe von Blöcken und Verbindungen. Diese können meist frei in zwei Dimensionen auf einer Zeichenfläche platziert werden. Solche Lösungen vermögen es dem Anwender größere Freiheiten zu bieten, als eine rein formularbasierte Oberfläche. Der Abstraktionsgrad ist ebenfalls hoch.

- *Textuelle DSLs*: Der Anwender nutzt eine Programmiersprache, welche für die Beschreibung von Testabläufen optimiert ist. In der Praxis sind textuelle DSLs mächtiger als ihr grafisches Pendant. Zum Beispiel stellt die grafische Darstellung von TTCN-3 eine echte Untermenge der textuellen Notation (Core Language) dar. Bezüglich Abstraktion ist es schwer einen Unterschied auszumachen. Eine grafische Repräsentation erschließt sich jedoch häufig intuitiv, während Quelltext genaues Lesen erfordert, weswegen letzterer oft als weniger abstrakt wahrgenommen wird.

- *Standardprogrammiersprachen*: Der Anwender programmiert den Testablauf mit Hilfe einer GPL. Dieser Ansatz ist am weitesten verbreitet und bietet die größte Mächtigkeit. Ein Beispiel ist die direkte Verwendung der ASAM XIL API für Tests der Systemebene. APIs und Bibliotheken können Testspezifische und wiederkehrende Aufgaben kapseln, womit der Ansatz auch Abstraktion unterstützt. Verglichen mit dem Entwurf einer DSL, bei dem bereits die Syntax eine Abstraktion darstellen kann, ist die Abstraktionsfähigkeit jedoch reduziert.

Hinsichtlich Mächtigkeit und Abstraktion muss also eine Abwägung erfolgen. Diesbezüglich wird der Ansatz formularbasierter Oberflächen verworfen, denn diese sind für die angestrebte Mächtigkeit der neuen Testlösung nicht flexibel genug.

Nachteile von GPLs

GPLs sind hingegen ausgesprochen mächtig, und auch der erreichbare Abstraktionsgrad ist nicht zu niedrig. Vor dem Hintergrund der Durchgängigkeit offenbaren sich jedoch Probleme: die Anfor-

derungen A3.1 (Kein Paradigmenwechsel), A3.2 (Transparente Testgrößen) und A3.3 (Typsicherheit) lassen sich nicht wie gewünscht erfüllen, wie im Folgenden erklärt wird.

Die Sichtweise des klassischen Unit-Tests erfordert, dass die gewählte GPL direkt kompatibel mit der Sprache der getesteten Software ist, wobei von C/C++ ausgegangen wird. Beispielsweise muss eine Funktion der Target-Software direkt aufgerufen werden können. Tatsächlich handelt es sich dabei jedoch immer um einen entfernten Funktionsaufruf (RPC), denn der Test-Code wird schließlich auf dem Host-PC ausgeführt (vgl. Abschnitt 4.3.1). Das lässt sich mit den üblichen GPLs nicht vollständig transparent darstellen.

Hinzu kommt, dass auf Systemebene physikalische Größen eine wichtige Rolle spielen. Dazu ist neben Datentypen, die kompatibel mit den Datentypen der getesteten Software sind, auch noch ein Datentyp mit physikalischer Einheit erforderlich. Das kann mit Standardprogrammiersprachen zwar prinzipiell durch entsprechende Klassen realisiert werden, eine Unterstützung auf Sprachebene ist jedoch benutzerfreundlicher und bietet bessere Voraussetzungen für die Implementierung von Typsicherheit.[2] Abschließend, und dieser Punkt ist am einfachsten nachzuvollziehen, bieten GPLs keine testspezifischen Sprachkonstrukte, welche die Testentwicklung vereinfachen. Ein gutes Beispiel hierfür ist das Konstrukt `testcase` aus TTCN-3 (siehe Abschnitt 3.5.2).

Textuelle vs. grafische DSL

Aus diesem Grund verengt sich die Auswahl auf den Entwurf einer grafischen oder einer textuellen DSL. Diesbezüglich zeigt die Praxis, dass grafische Ansätze auf geringe Akzeptanz stoßen, sobald die Beschreibung von Kontrollfluss im Vordergrund steht. Gut beobachten lässt sich das an TTCN-3: die textuelle Notation wurde in der Telekommunikationsbranche schnell angenommen, während die grafische Notation kaum Aufmerksamkeit erfährt [Wilo5]. Demgegenüber lassen sich strukturelle Eigenschaften grafisch meist eleganter ausdrücken als textuell. Das zeigt der Erfolg grafischer Ansätze, wenn der Datenfluss im Vordergrund steht, oder bei der Beschreibung integrierter Schaltkreise, wo eine grafische Beschreibung zumindest des groben Blockdesigns einer Programmierung in VHDL oder Verilog vorgezogen wird.

Kombination textueller und grafischer DSL

Im Rahmen dieser Arbeit ist sowohl die Beschreibung von Kontrollfluss (für den Testablauf) also auch die Beschreibung von Struktur

2 Typsicherheit bedeutet in diesem Zusammenhang, dass Werte mit verschiedenen, aber kompatiblen Einheiten, beispielsweise Temperaturen in °C und °F, implizit konvertiert und miteinander verrechnet werden können, während inkompatible Typen bereits zur Implementierungszeit erkannt werden.

(für den Testaufbau) notwendig. Aus diesem Grund wird eine Kombination beider Ansätze verfolgt. Zur Beschreibung des Testablaufs wird dem Anwender eine textuelle DSL zur Verfügung gestellt. Diese wird als *ETSpec Sprache* bezeichnet. Der Grundgedanke ist vergleichbar mit TTCN-3: es werden zentrale Elemente einer Standardprogrammiersprache übernommen, gleichzeitig erfolgt jedoch eine Spezialisierung auf den Test kritischer eingebetteter Software. Zur Beschreibung des Testaufbaus steht dem Anwender hingegen ein Werkzeug zur grafischen Modellierung zur Verfügung. Beides wird in Kapitel 5 genauer erklärt.

4.3.3 *Workflow*

Abbildung 23 zeigt den resultierenden Workflow für die konzipierte Lösung. Die erste Aufgabe des Testentwicklers ist die grafische Modellierung des Testaufbaus. Das umfasst die relevanten Komponenten des Testobjekts sowie die angeschlossene Zugriffshardware und deren Konfiguration. Das entsprechende Modell wird *Test Rig Model* (TRM) genannt.

Modellierung des Testaufbaus

Weitere Modelle, genannt *Test Quantity Container* (TQC), dienen der Beschreibung aller Testgrößen, die bei der späteren Entwicklung von Testfällen verwendet werden können. Sie können teilweise automatisch erzeugt werden. So ergibt sich die vollständige Beschreibung der Funktionen und Variablen der Software (Symbole) aus dem Quelltext und den Debug-Informationen des Compilers. Für Nachrichten können die entsprechenden Informationen häufig aus bereits existierenden Beschreibungen importiert werden. Im Automobilbereich sind beispielsweise das Field Bus Data Exchange Format (FIBEX), welches von der ASAM definiert wird [Std/ASA14], und data base CAN (DBC) verbreitet. In jedem Fall ist die Testgröße anschließend durch einen Identifier eindeutig bestimmbar.

Definition der Testgrößen

Anschließend werden Testfälle manuell implementiert. Dazu existiert ein spezieller Editor, welcher die Informationen aus allen Modellen nutzt, um den Entwickler zu unterstützen. Beispiele hierfür sind Codevorschläge, das Prüfen und Vorschlagen von Namen der im jeweiligen Kontext verfügbaren Testgrößen oder die Prüfung von Datentypen.

Implementierung der Testfälle

Der Quelltext wird anschließend kompiliert und die Testfälle werden und mit Hilfe der Laufzeitumgebung ausgeführt, welche die Zugriffshardware bzw. ein HiL Testsystem entsprechend ansteuert. Wird eine im Testfall spezifizierte Erwartung verletzt, wird das Testfall-bezogen protokolliert, woraus anschließend Berichte generiert werden, die den aktuellen Projektstatus abbilden.

Automatisierte Ausführung

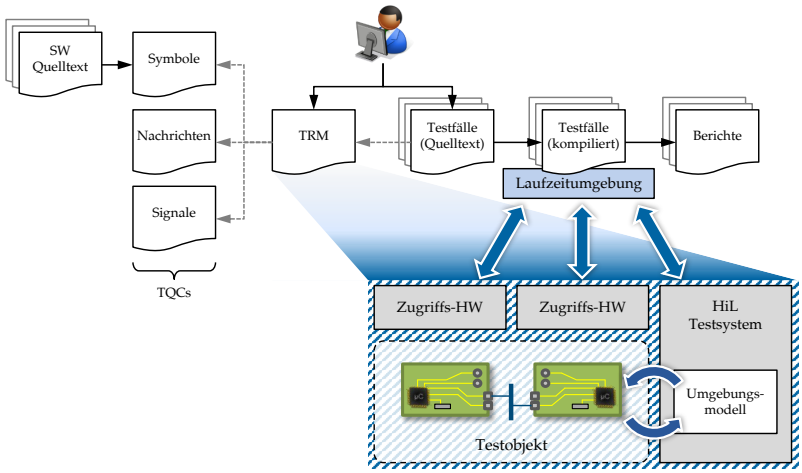


Abbildung 23: Workflow zur Testentwicklung und automatisierter Ausführung

4.4 BEZIEHUNG ZUR ASAM XIL API

Die ASAM XIL API, welche in Abschnitt 3.3.4 bereits vorgestellt wurde, umfasst einige Komponenten, die im beschriebenen Konzept in ähnlicher Form auftauchen, aber parallel zu dieser Arbeit entwickelt wurden. Es bietet sich an, beide Lösungen miteinander zu verschmelzen, anstelle die XIL API als Schnittstelle zu Systemen für MiL, SiL und HiL Tests zu nutzen, wie es das zuvor beschriebene Konzept vorsieht. Zum Zeitpunkt der Veröffentlichung der XIL API war diese Arbeit für eine entsprechende Umsetzung bereits zu weit fortgeschritten. Deswegen sollen an dieser Stelle lediglich die Gemeinsamkeiten und Unterschiede zum Konzept in dieser Arbeit aufgezeigt werden.
In Abbildung 24 sind diese veranschaulicht. In der Darstellung ist das Blockdiagramm zur XIL API enthalten, welches in Abschnitt 3.3.4 bereits vorgestellt wurde. Daneben ist Debug- und Trace Hardware als weitere Schnittstelle zum Testobjekt aufgeführt, welche von der XIL API nicht berücksichtigt wird. Analog zu den anderen Schnittstellen der XIL API lässt sich dort von den spezifischen APIs abstrahieren (DBG).

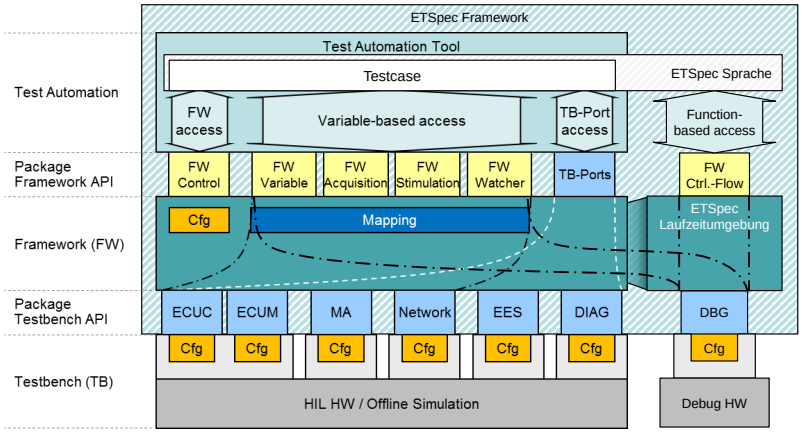


Abbildung 24: Beziehung zwischen ETSpec und ASAM XIL API

Die Debug Hardware stellt bezüglich dem Zugriff auf den Speicher des Testobjekts eine Alternative zu der Schnittstelle ECUM/ECUC dar. Deswegen kann entsprechender „Variable-based access“ auch darauf gemappt werden. Mit Debug Hardware ist zusätzlich jedoch auch direkte Ausführungskontrolle möglich (RPC, setzen von Breakpoints etc.). Das erfordert einen „Function-based access“, der von der XIL API nicht vorgesehen ist.
Die „ETSpec Laufzeitumgebung“ entspricht weitestgehend dem „XIL API Framework“. Beide sind für das Mapping auf die Zugriffshardware verantwortlich.
Die „Testbench API“ des Standards entspricht genau der „API Abstraktionsschicht“ des ETSpec Frameworks. Beide abstrahieren von der Zugriffshardware verschiedener Hersteller.[3]
Der Begriff „ETSpec Framework“ umfasst hingegen die gesamte „XIL API“ zuzüglich eines „Test Automation Tools“. Letzteres ist im Blockdiagramm zur XIL API zwar dargestellt, jedoch nicht standardisiert. Ebenfalls gar nicht berücksichtigt wird von der ASAM XIL API eine spezielle Sprache zur Testimplementierung.

3 Die Schnittstellen EES und DIAG werden vom ETSpec Framwork nicht berücksichtigt. Das deckt sich mit dem Vorgehen der ASAM, welche diese Schnittstellen bei der Erweiterung der HIL API ebenfalls außen vor ließ.

5 KONZEPT ZUR TESTIMPLEMENTIERUNG

Das Fundament eines Konzepts zur Testimplementierung wurde bereits in Abschnitt 4.3.2 gelegt. Demnach dient die ETSpec Sprache der Testimplementierung, während grafische Modelle den Testaufbau beschreiben. Das Zusammenspiel von Sprache und Modellen wird in Abschnitt 5.3 erläutert.
Die ETSpec Sprache ist eine DSL für den Test eingebetteter Software, welche im Kern einer GPL sehr nahe kommt. Es wurde bereits ausgeführt, warum eine GPL für die angedachte Aufgabe nicht gut geeignet ist. Im Bereich eingebetteter Systeme gibt es jedoch bereits einige andere DSLs, die ebenfalls auf den Test von Software spezialisiert sind (siehe Abschnitt 3.5.2 und 3.5.3), und als Grundlage deswegen in Betracht gezogen werden müssen. Damit beschäftigt sich Abschnitt 5.2. Zuvor müssen jedoch die Anforderungen an eine solche Testprogrammiersprache noch weiter konkretisiert werden.

5.1 ANFORDERUNGEN AN EINE TESTPROGRAMMIERSPRACHE

In Kapitel 4 wurde durch Anforderung A3 (Testimplementierung) schon indirekt auf eine zu diesem Zweck entworfene Sprache eingegangen. Mit der Entscheidung für eine DSL muss zunächst der unter A3.4 (Mächtigkeit) bereits aufgeführte Punkt weiter konkretisiert werden (im Folgenden AD1). Zusätzliche Anforderungen ergeben sich für spezielle Sprachkonstrukte, mit denen eine Anpassung an die konkrete Domäne erfolgt (im Folgenden AD2).

AD1 *Mächtigkeit*: In Bezug auf die Mächtigkeit einer DSL bietet sich die Verwendung des Begriffs der Turing-Vollständigkeit aus der Berechenbarkeitstheorie an. Er besagt, dass alle Funktionen berechnet werden können, die eine universelle Turingmaschine berechnen kann.[1] Alle bekannten GPLs sind Turingvollständig, und in der Literatur wird Turing-Vollständigkeit mitunter sogar als notwendiges Kriterium für den Begriff „Programmiersprache“ verwendet [Wiki/TV; MLe99]. Turing-Vollständigkeit ist allein also keine ausreichende Anforderung, auch weil das den Implementierungsaufwand unberücksichtigt lässt. Es ist jedoch kein besseres, mathematisch

1 Zur Berechenbarkeitstheorie seien die Bücher [Sip12; Mar10] empfohlen

erfasstes Kriterium bekannt. Die Anforderung wird deswegen intuitiv erweitert: die DSL muss „vergleichbar mächtig" sein, wie bestehende GPLs. Das setzt Turing-Vollständigkeit voraus und wird durch die folgenden Punkte erweitert:

AD1.1 *Typsystem*: Die DSL muss ein Typsystem besitzen, dass die strukturierte Speicherung und Verarbeitung von Daten ermöglicht.

AD1.2 *Ausdrücke*: Die DSL muss die Verarbeitung von Daten durch beliebig komplexe mathematische Ausdrücke unterstützen. Hierzu sind geeignete Operatoren erforderlich.

AD1.3 *Kontrollfluss*: Die DSL muss die flexible Beschreibung von Kontrollfluss ermöglichen.

AD1.4 *Modularisierung*: Die DSL muss die Modularisierung der Quelltexte und die Wiederverwendung von Teilimplementierungen unterstützen.

AD1.5 *Standardbibliothek*: Die DSL muss Zugriff auf eine Standardbibliothek bieten, deren Funktionsumfang mit der Standardbibliothek üblicher GPLs vergleichbar ist. Beispiele für die in einer Standardbibliothek hinterlegten Funktionalitäten sind der Zugriff auf das Dateisystem, mathematische Funktionen (Sinus, Cosinus, Logarithmus, Minimum, Maximum etc.) und Datenstrukturen (Verkettete Listen, Hashtabellen etc.).

AD2 *Angepasstheit*: Die DSL muss an die spezielle Domäne der angestrebten Testlösung angepasst sein. Das stellt den entscheidenden Unterschied zu einer GPL dar. Es sollen die folgenden Anforderungen erfüllt werden:

AD2.1 *Zentrale Testfeatures*: Testfälle, Test Suiten und Erwartungen müssen durch spezielle Sprachkonstrukte eindeutig gekennzeichnet werden können.

AD2.2 *Externe Deklaration*: Die Anforderungen A3.1 (Kein Paradigmenwechsel) und A3.2 (Transparente Testgrößen) setzen voraus, dass alle Testgrößen im Quelltext der DSL referenziert werden können. Zur Vermeidung von Inkonsistenzen sollte dazu die Deklaration zusätzlicher Hilfsvariablen in der DSL nicht nötig sein. Zur Aufrechterhaltung der Typsicherheit muss eine Typprüfung gegen die ursprüngliche Deklaration möglich sein.

AD2.3 *Physikalische Datentypen*: Die DSL muss die Verwendung physikalischer Größen durch geeignete Konstrukte unterstützen.

AD2.4 *Komplexe Testobjekte*: Das Testobjekt kann sich aus unabhängigen Einzelkomponenten zusammensetzen. Die DSL muss die Beherrschung der Komplexität durch geeignete Sprachkonstrukte unterstützten.

AD2.5 *Kontrollierte Nebenläufigkeit*: Testabläufe enthalten häufig Nebenläufigkeit. Die DSL muss daher die Beschreibung parallel ausgeführter Kontrollfäden und deren Synchronisation durch geeignete Sprachkonstrukte unterstützen.

AD2.6 *Vorgabe von Echtzeitverhalten*: Die DSL muss die Beschreibung von Abläufen unterstützen, für die eine Ausführung in Echtzeit garantiert werden kann. Bei beliebig komplexen Beschreibungen lässt sich die Echtzeitfähigkeit nicht im Voraus prüfen. Die Sprache muss daher einschränkende Sprachkonstrukte und Mechanismen bieten, um Echtzeitverhalten zu beschreiben.

AD2.7 *Analyse von Echtzeitdaten*: Die DSL muss die Prüfung von Echtzeitverhalten und die Analyse analoger Signalverläufe anhand aufgezeichneter Daten durch geeignete Konstrukte unterstützen.

5.2 NACHTEILE DER VERWENDUNG BESTEHENDER DSLS

In Abschnitt 3.5 wurden bestehende DSLs aus dem Bereich des Tests eingebetteter Software vorgestellt. Der Entwurf einer neuen Sprache wurde der Verwendung oder Erweiterung einer bestehenden DSL aus mehreren Gründen vorgezogen.

Von GPLs bekannte Nachteile

Zunächst kann keine der aufgeführten DSLs die Anforderung AD2.2 (Externe Deklaration) erfüllen. AD2.3 (Physikalische Datentypen) wird ausschließlich von CCDL unterstützt. Diese Punkte haben bereits zur Ablehnung bestehender GPLs als Sprache zur Testimplementierung beigetragen (vgl. Abschnitt 4.3.2).
Die Testsprache des Werkzeugs PROVEtech:TA der Firma MBtech ist als GPL zu betrachten, da keine der unter AD2 (Angepasstheit) aufgeführten Anforderungen durch ein Sprachkonstrukt unterstützt wird, sondern stattdessen mit speziellen Klassenbibliotheken gearbeitet wird.

Niedrige Mächtigkeit

Die Sprachen CCDL, TESLA und PRACTICE bieten nicht die gewünschte und mit GPLs vergleichbare Mächtigkeit (Anforderung AD1). Insbesondere die Möglichkeiten zur Modularisierung und die Verfügbarkeit von Standardbibliotheken sind stark eingeschränkt.

Keine freie Verfügbarkeit

Für keine der bestehenden DSLs ist eine quelloffene Werkzeugkette verfügbar, die als Grundlage für die Implementierung eines Prototyps und zur Evaluation verwendet werden könnte. Daher müsste der relevante Sprachumfang vollständig nachgebildet werden, sodass sich gegenüber einem vollständig neuen Ansatz keine Aufwandsersparnis bei der Implementierung erzielen lässt. Um zu einem ursprünglichen Sprachstandard kompatibel zu bleiben, wäre vielmehr sogar ein Mehraufwand zu erwarten.

Nachteile von TTCN-3

Am besten abgedeckt werden die aufgestellten Anforderungen durch TTCN-3. Die Mächtigkeit von TTCN-3 ist hoch, und in Bezug auf AD2 (Angepasstheit) lassen sich zumindest zu den Unterpunkten AD2.1 und AD2.4 spezielle Sprachkonstrukte identifizieren. Die übrigen Unterpunkte werden nicht erfüllt. Auf das Fehlen einer expliziten Trennung von der Beschreibung echtzeitkritischen Verhaltens wurde bereits in Abschnitt 3.5.2 und in [GSS09; SBG05] eingegangen. Es existieren auch keine Sprachkonstrukte, welche bei der Analyse analoger Signalverläufe unterstützen könnten.

5.3 ETSPEC SPRACHE UND MODELLE

In Abschnitt 4.3.2 und Abschnitt 4.3.3 wurden ein grafisches Modell zur Beschreibung des Testaufbaus (TRM) sowie weitere Modelle zur Beschreibung der Testgrößen (TQCs) genannt und begründet. Ein weiterer Grund für diese Modelle ist, dass ihre geschickte Verknüpfung mit der ETSpec Sprache die Grundlage zur Erfüllung wesentlicher Anforderungen darstellt. Zum Verständnis dieses Zusammenhangs muss im Folgenden zuerst das TRM genauer erklärt werden. Wissen über den Aufbau der TQCs ist noch nicht erforderlich, sie werden in Abschnitt 7.1 beschrieben.

5.3.1 *Das Test Rig Model*

Das TRM lässt sich am einfachsten anhand des entsprechenden Metamodells erklären. Es ist in Abbildung 25 durch ein Klassendiagramm in der Ecore Notation [Ste09] dargestellt.

AccessAPI

Oben in der Darstellung finden sich die Interface-Klassen *AccessAPI*, *Inspectable* und *ScopeProvider*. *AccessAPI* ist eine Generalisierung

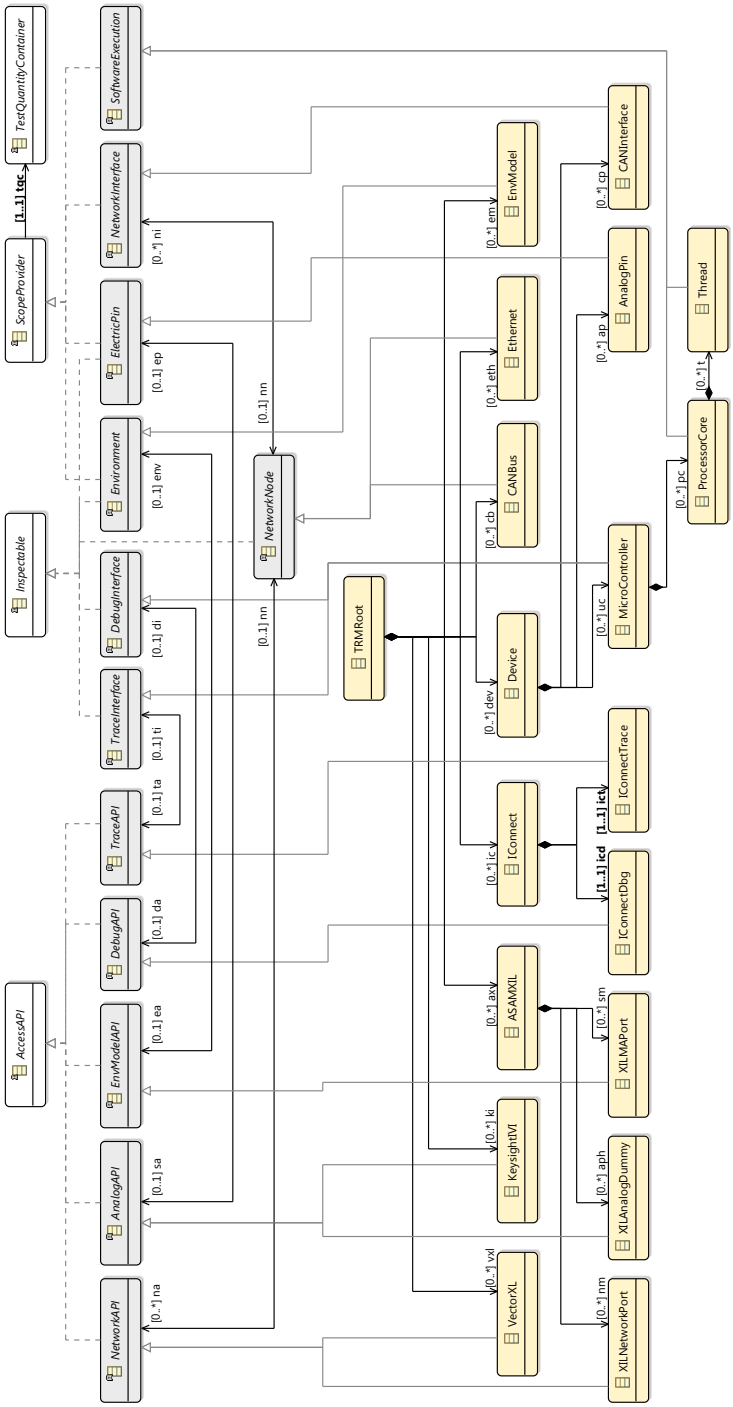


Abbildung 25: Metamodell des TRMs (vereinfacht)

der spezifischen APIs, die für den Zugriff auf das Testobjekt und das Umgebungsmodell zur Verfügung stehen. Durch eine Reihe davon abgeleiteter und abstrakter Klassen erfolgt eine Unterteilung in verschiedene Kategorien, je nach Art des Zugriffs.
NetworkAPI bezieht sich auf Zugriffshardware für ein Kommunikationsnetzwerk, z.B. ein CAN Bus Interface. *AnalogAPI* bezieht sich auf Zugriffshardware, welche die Messung, Aufzeichnung oder Generierung einfacher digitaler und analoger Signalverläufe unterstützen. Ein Oszilloskop oder Funktionsgenerator sind dazu einfache Beispiele. *EnvModelAPI* ist eine Schnittstelle zum Umgebungsmodell. *DebugAPI* und *TraceAPI* beziehen sich auf Debug- und Trace-Hardware.

Inspectable und ScopeProvider

Inspectable ist eine Generalisierung der Schnittstellen des Testobjekts, auf die mit Hilfe einer der durch *AccesAPI* repräsentierten Werkzeuge zugegriffen werden kann. *ScopeProvider* ist eine Generalisierung der Komponenten des Testobjekts, die aus Sicht des Testentwicklers relevant sind, und auf die er sich in der Testimplementierung beziehen kann (die Namensgebung erklärt sich später).
Zwischen *Inspectable* und *ScopeProvider* gibt es Überschneidungen, die auch im Metamodell sichtbar sind. Zum Beispiel repräsentiert *ElectricPin* eine elektrische Schnittstelle des Testobjekts, auf welche über einfache analoge Messgeräte zugegriffen werden kann. *ElectricPin* ist daher von *Inspectable* abgeleitet. *ElectricPin* erbt jedoch auch von *ScopeProvider*, weil elektrische Schnittstellen direkt mit einem Signal assoziiert sind, und deswegen für den Testentwickler relevant sind.
NetworkNode repräsentiert hingegen die Netzwerkinfrastruktur zwischen mehreren ECUs. Weil die entsprechende Zugriffshardware damit verbunden wird, ist *NetworkNode* von *Inspectable* abgeleitet. Ein Testentwickler möchte sich über die Netzwerkinfrastruktur jedoch keine Gedanken machen, sondern hat eine bestimmte Netzwerkschnittstelle, die durch *NetworkInterface* repräsentiert wird, im Blick.
Die Klasse *SoftwareExecution* repräsentiert einen Kontrollfaden der Software. Werden sie durch ein Betriebssystem verwaltet, spricht man von einem *Thread*, und sie werden durch die gleichnamige Klasse im Metamodell repräsentiert. Einfache eingebettete Systeme besitzen jedoch nicht immer ein Betriebssystem. Wenn die Prozessorhardware mehrere Rechenkerne besitzt, muss trotzdem zwischen mehreren Kontrollfäden unterschieden werden, was im Metamodell durch die Klasse *ProcessorCore* erfasst wird.

Repräsentation von Verbindungen

Die Verbindung zwischen *AccessAPI* und *Inspectable* wird im Metamodell durch eine bidirektionale Referenz zwischen den abgeleite-

ten Klassen dargestellt. Tatsächlich wird jede dieser Verbindungen durch eine eigene Klasse repräsentiert, die zusätzliche Eigenschaften aufnehmen kann. Sie sind der Übersichtlichkeit halber jedoch nicht dargestellt. Das Metamodell schließt einige unmögliche Verbindungen bereits aus, aber nicht alle. Beispielsweise kann nach den Regeln des Metamodells jede beliebigen Konkretisierung von *NetworkAPI* mit der Konkretisierung *CANBus* verbunden werden, auch wenn die repräsentierten Komponenten in der Realität nicht kompatibel sind.[2]

Grafische Repräsentation

Das Metamodell in Abbildung 25 zeigt im unteren Bereich zu jeder abstrakten Klasse mindestens eine instanziierbare Konkretisierung. Diese werden später durch einen Editor grafisch repräsentiert. Für jede vom ETSpec Framework unterstützte API, Netzwerktechnologie bzw. Komponente des Testobjekts muss eine solche Repräsentation existieren. Wird sie mit Hilfe des Editors auf der Zeichenfläche platziert, wird das zugehörige Modellelement erstellt und dem Modell hinzugefügt. Anschließend lassen sich die jeweils spezifischen Konfigurationsparameter anpassen, welche ebenfalls als Teil des Modells gespeichert werden.

Hierarchie

Alle Komponenten werden dem TRM im Rahmen einer festgelegten Hierarchie hinzufügt. Diese lässt sich im Metamodell an den Kompositionsbeziehungen ablesen (dargestellt durch die ausgefüllte Raute) und wurde der Realität nachempfunden. Auf der höchsten Ebene, also direkt auf der Zeichenfläche des Editors, können nur diejenigen Elemente erstellt werden, welche direkt in das Wurzelelement des Modells, *TRMRoot*, eingehängt sind. Einige Komponenten wie *Device* sind selbst ein Container, der weitere Komponenten aufnehmen kann.

Abbildung 26 zeigt ein Beispiel zur grafischen Repräsentation des TRMs. Es geht hier nicht um ein realistisches Anwendungsszenario, sondern darum, möglichst kompakt zu jeder instanziierbaren Klassen des Metamodells eine Repräsentation zu zeigen. Praktische Beispiele finden sich in Kapitel 8.

5.3.2 *Zweck der Modelle*

Was gewinnt man durch die separaten Modelle (TRM und TQCs) nun konkret? Zunächst bietet das TRM durch seine grafische Repräsentation eine übersichtliche Darstellung, welche die Konfiguration der gesamten Testumgebung erleichtert. Wichtiger ist jedoch

2 Es ist normal, dass nicht alle Einschränkungen der Realität im Metamodell erfasst werden.

Auflösung des Zugiffs

die Tatsache, dass jedem *ScopeProvider* genau eine *AccessAPI* eindeutig zugeordnet ist. Mit dieser Information kann durch die ETSpec Runtime für jede gegebene Testgröße bestimmt werden, wie der Zugriff tatsächlich zu bewerkstelligen ist. Für die in Anforderung A3.5 geforderte Unabhängigkeit der Testimplementierung von der Zugriffshardware ist das eine entscheidende Voraussetzung.
Auch zur Erfüllung der Anforderung AD2.2 sind die Modelle hilfreich. Durch die referenzierten TQCs sind alle notwendigen Informationen über die Testgrößen vorhanden, sodass eine Deklaration entsprechender Hilfsvariablen in der ETSpec Sprache nicht erforderlich ist. Weil die Information in den TQCs außerdem zentralisiert ist, und diese großteils automatisch erzeugt werden, sind Inkonsistenzen nicht möglich oder zumindest leicht zu bemerken.

Vermeidung von Inkonsistenz

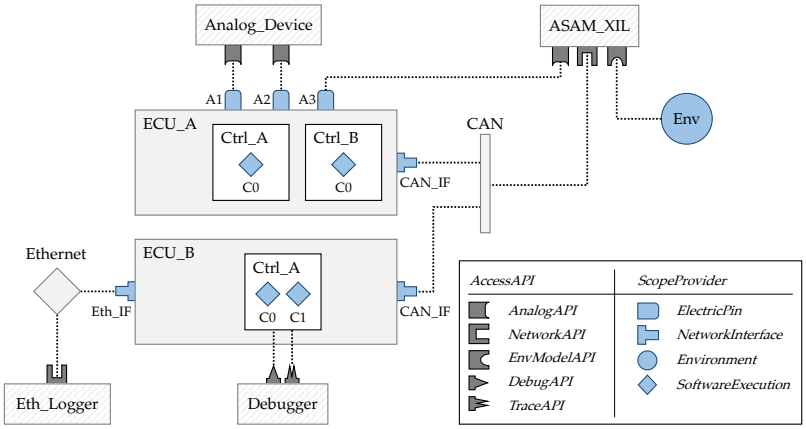


Abbildung 26: Beispiel zur grafischen Repräsentation des TRMs

5.3.3 *Nutzung der Modellinformation*

Die in den Modellen gespeicherte Information muss spätestens bei der Kompilierung des ETSpec Quelltexts geladen werden. Die Modelle müssen dazu im Dateisystem gespeichert werden, sodass sie durch den Anwender leicht nachvollziehbar aus dem ETSpec Quelltext referenziert werden können, um die enthaltenen Informationen zu laden. Zur Speicherung ist das XML Metadata Interchange Format (XMI) [OMG15b] eine Möglichkeit.
Nach dem Laden der Modelle müssen Testgrößen im ETSpec Quelltext über einen einfachen Namen referenziert werden können. Dabei müssen Namenskonflikte vermieden werden. Außerdem dürfen einige Testgrößen nur in Zusammenhang mit einem *ScopeProvider*

verwendet werden, z.B. ist es bei einem entfernten Funktionsaufruf notwendig zu wissen, durch welche Komponente des Testobjekts (Thread oder Prozessorkern) die Funktion auszuführen ist.
Beide Probleme lassen sich durch eine Einschränkung der *Sichtbarkeit* lösen, indem alle Testgrößen ausgeblendet werden, die nicht dem TQC genau eines *ScopeProviders* angehören.[3] Dieser *ScopeProvider* soll dazu vom Testentwickler im Quelltext explizit angegeben werden. Diese Idee bestimmt maßgeblich das Sprachdesign in Kapitel 6.

Sichtbarkeit

5.4 RELEVANTE PROGRAMMIERPARADIGMEN

Das älteste Programmierparadigma ist die imperative Programmierung. Es folgt der grundsätzlichen Arbeitsweise heutiger Computer, nämlich der schrittweisen Abarbeitung einer Abfolge von Befehlen. Die imperative Programmierung ist weiterhin das zentrale Paradigma moderner Programmiersprachen.
Die meisten Testfälle entsprechen ebenfalls einer Abfolge einzelner Schritte, zumeist Stimuli oder Erwartungen. Es ist daher vollkommen natürlich, Testfälle vorwiegend imperativ zu programmieren. Demgegenüber ist deklarative Programmierung, welche als Gegenentwurf zur imperativen Programmierung gilt, und anstelle einzelner Schritte („wie“) das gewünschte Ergebnis („was“) beschreibt, zur Beschreibung eines Testfalls eher ungeeignet. Schließlich ist beim Test der Weg zum Ziel entscheidend, da er direkten Einfluss auf das im Testobjekt hervorgerufene Verhalten hat.

Imperative Programmierung

Daten spielen beim Testen eine wichtige Rolle. objektorientierte Programmierung ist wegen der hervorragenden Unterstützung abstrakter Datentypen deswegen ebenfalls ein wichtiges Programmierparadigma. Die ETSpec Sprache unterstützt objektorientierte Programmierung durch Zugriff auf die sogenannte *Host-Sprache*. Das ist die Sprache, in welche der ETSpec Quelltext übersetzt wird, damit er ausgeführt werden kann. Im Rahmen der prototypischen Implementierung wurde Java verwendet, C# und C++ sind jedoch gleichermaßen als Host-Sprache geeignet. Andere objektorientierte Sprachen kommen ebenfalls in Frage.

Objektorientierte Programmierung

Host-Sprache

Innerhalb eines Testablaufs muss häufig auf Ereignisse gewartet werden, z.B. das Betreten eines bestimmten Zustands oder den Empfang einer Nachricht. In solchen Fällen ist ereignisorientierte Programmierung relevant. Die ETSpec Sprache unterstützt ereig-

Ereignisorientierte Programmierung

3 Das erklärt den Namen „ScopeProvider“

nisorientierte Programmierung durch vordefinierte Ereignisse (Abschnitt 6.3.9) und die *wait-until* Anweisung (Abschnitt 6.5.3).

Synchrone Programmierung

Für den Test eingebetteter Software ist außerdem noch synchrone Programmierung interessant. Der deterministische Aufbau erlaubt die exakte Abschätzung der Laufzeit eines Programms und eignet sich somit sehr gut zur Vorgabe von Echtzeitverhalten entsprechend der Anforderung AD2.6. Die ETSpec Sprache ist keine synchrone Programmiersprache wie z.B. Esterel, adaptiert das Prinzip der synchronen Programmierung jedoch im Rahmen der *realtime* Anweisung (Abschnitt 6.5.6). Weitere Programmierparadigmen wurden beim Design der ETSpec Sprache nicht explizit berücksichtigt.

5.5 AKZEPTANZ DER ETSPEC SPRACHE

Bei der Einführung einer neuen Sprache sieht man sich mit einem Akzeptanzproblem konfrontiert. Als erstes fürchten Anwender häufig, sich in die Abhängigkeit von einem Hersteller zu begeben. Diese Gefahr ist nicht von der Hand zu weisen, aber die Wahrscheinlichkeit ist umso größer, wenn Testfälle in einem rein maschinell lesbaren Formate gespeichert werden, wie es bei vielen der in Abschnitt 3.6 genannten Testlösungen der Fall ist. Ein offener Sprachstandard hingegen fördert den Wettbewerb. Darüber hinaus steht der Quelltext des ETSpec Frameworks unter der Eclipse Public License frei zur Verfügung.[4]

Weitere Bedenken existieren bezüglich der Kompatibilität mit bestehenden Systemen. Langfristig ist es sicherlich nicht sinnvoll, zwei verschiedene Lösungen zur Testautomatisierung parallel zu betreiben. Bestehende Testfälle auf Knopfdruck in die neue Sprache zu übersetzen wird jedoch nicht möglich sein. Eine gewisse Übergangsphase ist daher unvermeidbar. Hierzu ist es wichtig, dass die neue Lösung nahtlos in die bestehende Umgebung integriert werden kann. Dazu definiert das ETSpec Framework geeignete Schnittstellen. Außerdem ist eine Integration selbst dann noch möglich, wenn die Schnittstellen des Frameworks spezielle Anforderungen nicht erfüllen, denn unter Verwendung der Host-Sprache lässt sich das Framework auch umgehen.

Ein zusätzliches Hindernis bei der Akzeptanz einer neuen Sprache ist die Befürchtung, dass hierzu ein langwieriger Lernprozess erforderlich ist. Beim Design der Sprache wird deswegen darauf geachtet, dass zur Beherrschung der wichtigsten Funktionen von

4 Abrufbar unter www.es-tdk.org.

einem durchschnittlichen Programmierer weder eine neue Denkweise noch ungewohnte Syntax erlernt werden müssen. Darüber hinaus ist nicht geplant, dass Testentwickler die ETSpec Sprache „blind“ verwenden, sondern es wurde bereits im Konzept die Unterstützung durch umfangreiche Entwicklungswerkzeuge vorgesehen.

Teil III

UMSETZUNG

Im dritten Teil wird die Umsetzung des Konzepts beschrieben. Zuerst wird das Design der ETSpec Sprache anhand einer Variante der BNF im Detail erläutert, und anschließend auf die Implementierung des ETSpec Frameworks und die Integration in die Eclipse Entwicklungsumgebung näher eingegangen.

6 SPRACHDESIGN

Dieses Kapitel ist dem Design der ETSpec Sprache gewidmet. Bei den Erläuterungen wird „top-down“ entlang der Sprachdefinition vorgegangen, damit der Leser ein Gefühl für den Aufbau und die Zusammenhänge entwickeln kann. Die Überprüfung der Anforderungen erfolgt erst anschließend in Abschnitt 6.7. Die Syntax wird mit Hilfe einer Variante der BNF beschrieben, die in Abschnitt 2.3.5 eingeführt wurde. Dabei wird Regeln für Bezeichner und Literale zurückgegriffen, die in Anhang A separat aufgeführt sind.

6.1 VORBEMERKUNG ZUR SYNTAX

Ob sich die Semantik von einem Stück Quelltext leicht erschließt, hängt stark mit der konkreten Syntax der Programmiersprache zusammen. Die Syntax ist die „Benutzerschnittstelle“ für den Programmierer. Es wäre fatal, sich über diesen Punkt nicht sehr genaue Gedanken zu machen. Die Grundkonzepte einer Sprache können noch so perfekt sein – ist die Syntax durch den Anwender nur schwer zu durchschauen, ist die Sprache zum Scheitern verurteilt.

Berücksichtigung von Konventionen

Nun existieren Programmiersprachen bereits seit einigen Dekaden. Über die Zeit haben sich verschiedene Konventionen herausgebildet und gefestigt. Ob sich der Quelltext einer neuen Sprache weitgehend intuitiv erschließt, ob sich „neue“ Sprachkonstrukte klar von den Bekannten unterscheiden, oder ob intuitives Verständnis in direktem Widerspruch zur tatsächlichen Bedeutung steht, hängt maßgeblich davon ab, inwiefern die neue Sprache kompatibel mit den Konventionen ist, die dem Programmierer geläufig sind.

Anlehnung an C

Die ETSpec Sprache findet in einem Umfeld Anwendung, in dem die Programmiersprache C vorherrschend ist. Die mit C syntaktisch verwandten Sprachen sind heute auch am weitesten verbreitet.[1] Entsprechende Merkmale sollen daher auch für die ETSpec Sprache übernommen werden. Das betrifft beispielsweise alle Arten von Klammerung, d.h. geschwungene Klammern zur Eingrenzung von Codeabschnitten, eckige Klammern zur Indizierung in Datenfeldern (Arrays) und runde Klammern zur Anpassung der

1 Tatsächlich gehen einige der hier angesprochenen Syntaxmerkmale auf die Programmiersprache B zurück, welche durch C abgelöst wurde.

Präzedenz in Ausdrücken. Weitere Beispiele sind ein Strichpunkt am Ende vieler Anweisungen und Kommentare durch `//` bzw. `/* ... */`.

6.2 GRUNDLEGENDER AUFBAU

Das Startsymbol der Grammatik ist die Regel `GrammarRoot`:

```
1 GrammarRoot:
2     Include* ClassImport* ContainerDeclaration*
```

Die Regeln `Include`, `ClassImport` und `ContainerDeclaration` definieren insgesamt sechs Top-Level Konstrukte und damit den grundlegenden Aufbau der ETSpec Sprache. Sie werden im Folgenden der Reihe nach erklärt.

6.2.1 *Organisation auf Dateiebene*

ETSpec Quelltext wird auf Dateisystemebene in Form von Textdokumenten organisiert, wie es bei fast allen Programmiersprachen üblich ist. Wie bereits erwähnt, liegen auch das TRM und die TQCs in Form von Dateien vor. Zusammengehörige Quelltexte können sowohl in einer gemeinsamen Datei als auch durch eine entsprechende Ordnerstruktur zusammengefasst werden. Durch die Sprache gibt es dabei keinerlei Einschränkungen.

Globaler Scope

Deklarationen aus einer anderen Datei (das umfasst insbesondere auch das TRM) können durch das Schlüsselwort ‚`include`' dem *globalen Scope* des aktuellen Dokuments hinzugefügt werden. Der *globale Scope* enthält alle Elemente, die auf oberster Ebene durch einen Namen referenziert werden können. Neben Elementen der ETSpec Sprache, die durch den Testentwickler deklarierten werden können, z.B. Funktionen, Konstanten, und globale Variablen, gehören dazu die *ScopeProvider* des TRMs sowie die Testgrößen in den TQCs, was später noch genauer erklärt wird.

Aus der Regel `GrammarRoot` geht bereits hervor, dass *Includes* ganz zu Beginn des Dokuments stehen. Die Syntax ist sehr einfach:

```
1 Include:
2     'include' STRING_LITERAL ';'
```

6.2.2 *Import von Klassen*

Die Regel `ClassImport` ist vergleichbar mit `Include`, bezieht sich jedoch auf Klassen der Host-Sprache.

Die Host-Sprache wurde im Konzept als Zwischenschritt bei der Übersetzung der ETSpec Sprache vorgesehen. Es ist es naheliegend, dem Anwender die Funktionalität der Host-Sprache ebenfalls zur Verfügung zu stellen. Der *Import* einer Klasse der Host-Sprache ermöglicht deren Verwendung im ETSpec Quelltext. Dazu muss der vollqualifizierte Name (FQN) der Klasse angegeben werden. Optional kann durch das Schlüsselwort ‚as' ein Alias für den Klassennamen vergeben werden, der später zur Referenzierung dient:

```
1 ClassImport:
2     'import' FQN ('as' ID)? ';'
```

Weitere Details zur Integration der Host-Sprache sowie der Verwendung von Klassen und Objekten finden sich in Abschnitt 6.6.

6.2.3 *Basiscontainer*

Die Regel `ContainerDeclaration` definiert die Grundkonstrukte der ETSpec Sprache. Die ETSpec Sprache unterscheidet zwischen vier Grundkonstrukten, den sogenannten *Basiscontainern*. Im Einzelnen heißen sie *Target*, *Test*, *Package* und *Test Suite*:

```
1 ContainerDeclaration:
2     TargetDeclaration | TestDeclaration |
3     PackageDeclaration | TestSuiteDeclaration
```

TargetDeclaration

Ein *Target* der ETSpec Sprache ist eine Menge von *ScopeProvidern*. Es dient der Verknüpfung der Testimplementierung mit dem TRM. In Abschnitt 5.3.3 wurde bereits gesagt, dass alle Testgrößen ausgeblendet werden, die nicht im TQC von genau einem *ScopeProvider* liegen. In der Sprache muss daher auf *ScopeProvider* verwiesen werden können. Die einfachste Möglichkeit dazu ist ein FQN: er setzt sich zusammen aus den Namen aller Komponenten in der Hierarchie des TRMs, getrennt durch einen Punkt. Zur Veranschaulichung wird auf das Beispiel in Abbildung 26 zurückgegriffen: dort finden sich zwei *ScopeProvider* mit dem Namen „`C0`". Der FQN „`Example.ECU_A.Ctrl_A.C0`" ist jedoch eindeutig („Example" ist der Name des TRMs, welches zuvor natürlich mittels ‚`include`' geladen werden muss).

FQN des ScopeProviders

Der FQN ist jedoch unhandlich. Durch das *Target* kann nun ähnlich wie beim Import von Klassen ein Alias definiert werden. Die zugehörige Syntax ist ebenfalls ähnlich:

```
1 TargetDeclaration:
2     'target' ID
3     '{'
4         (TargetDefinition (',' TargetDefinition)*)?
5     '}'
6
7 TargetDefinition:
8     [ScopeProvider|FQN] ('as' ID)?
```

Ist das Alias durch den optionalen Zusatz via ‚`as`' nicht explizit angegeben, wird für den Bezeichner der `TargetDefinition` das letzte Glied des FQN angenommen.
Bei der Deklaration von einem *Test* bzw. einem *Package*, welche als Nächstes erklärt werden, kann eine `TargetDeclaration` referenziert werden. Damit ist die Testimplementierung mit dem TRM verknüpft.

TestDeclaration

Der *Test* ist das zentrale Konstrukt der ETSpec Sprache und enthält die Testimplementierung. Das *Target* wird durch einen Doppelpunkt vom Bezeichner des Tests getrennt:

```
1 TestDeclaration:
2     'test' ID ( ':' [TargetDeclaration|ID] )?
3         ( 'default' [TargetDefinition|ID] )?
4         ( 'observe' Expression )?
5     '{'
6         Statement*
7     '}'
```

Im Implementierungsblock wird, wie bei imperativen Programmiersprachen üblich, eine Abfolge von Anweisungen angegeben. Es wird dabei zwischen „allgemeinen" Anweisungen, die bereits aus GPLs bekannt sind, und „speziellen" Anweisungen, die es nur in der ETSpec Sprache gibt, unterschieden. Sie werden in Abschnitt 6.4 und Abschnitt 6.5 getrennt voneinander vorgestellt.
Durch das Schlüsselwort ‚`default`' (optional) wird ein *ScopeProviders* festgelegt, sodass es später einfacher ist, die entsprechenden Testgrößen zu verwenden. Das wird in Abschnitt 6.2.4 genauer erklärt.
Durch das Schlüsselwort ‚`observe`' (ebenfalls optional) wird ein Ausdruck zur Grenzwertüberwachung festgelegt. Durch die Laufzeitumgebung wir der Ausdruck zyklisch geprüft.

Listing 2: Beispiel mit den wichtigsten Top-Level Konstrukten der ETSpec Sprache

```
include "../model/Example.trm";

target MyTarget
{
    Example.Env             as Environment,
    Example.ECU_A.CAN_IF    as CAN_A,
    Example.ECU_B.CAN_IF    as CAN_B,
    Example.ECU_A.Ctrl_A.C0 as Controller1,
    Example.ECU_A.Ctrl_B.C0 as Controller2
}

test MyTestcase : MyTarget default Environment
{
    // Implementierung des Testablaufs
}

package MyPackage
{
    // Deklaration globaler Variablen und Funktionen
}
```

PackageDeclaration

Ein *Package* dient der Modularisierung und Wiederverwendung von Funktionalität. Beispielsweise sind bei vielen Testfällen immer die gleichen Schritte zur Initialisierung des Testobjekts nötig. Die entsprechende Implementierung kann über Funktionen in ein *Package* ausgelagert werden. Dort werden auch Konstanten, globale Variablen und benutzerdefinierte Datentypen deklariert:

```
PackageDeclaration:
    'package' ID ( ':' [TargetDeclaration|ID] )?
    '{'
        (FunctionDeclaration | DataTypeDeclaration |
         VariableDeclaration | ConstantDeclaration)*
    '}'
```

Genauso wie beim *Test* ist die Angabe eines *Targets* optional und ist mit einem Doppelpunkt vom Bezeichner des *Package* getrennt.
In Listing 2 findet sich ein Quelltextbeispiel mit den wichtigsten bisher erklärten Konstrukten. Es wird angenommen, dass unter dem relativen Pfad „`../model/Example.trm`“ das TRM aus Abbildung 26 abgespeichert ist.

TestSuiteDeclaration

Eine *Test Suite* dient der Vereinfachung der automatisierten Ausführung zusammengehöriger Testfälle. Mehrere Tests können damit zu einer Gruppe zusammengefasst werden:

```
1 TestSuiteDeclaration:
2     'suite' ID
3     '{'
4         ([TestDeclaration|ID] (',' [TestDeclaration|ID])*)?
5     '}'
```

6.2.4 *Festlegung des ScopeProviders*

Testgrößen sind nur im Kontext des zugehörigen *ScopeProviders* sichtbar. Eine Möglichkeit den nötigen Kontext zu erstellen bietet das Schlüsselwort ‚`default`'. Es wurde schon gezeigt, dass es bei der Deklaration eines Tests angegeben werden kann, wodurch die Testgrößen im gesamten Implementierungsblock des Tests sichtbar werden.
Sind mehrere *ScopeProvider* an einem Testfall beteiligt, muss dazwischen ein Wechsel möglich sein. Die ETSpec Sprache bietet dazu ein spezielles Konstrukt, die *scope* Anweisung:

```
1 ScopeStatement:
2     'scope' [TargetDefinition|ID]
3     '{'
4         Statement*
5     '}'
```

Der über das Alias der *TargetDefinition* angegebene *ScopeProvider* verdeckt einen möglicherweise zuvor gültigen, wodurch im entsprechenden Implementierungsblock nur noch die neuen Testgrößen sichtbar sind.
Teilt sich der Testablauf in einzelne Abschnitte, in denen jeweils nur ein *ScopeProvider* relevant ist, z.B. Stimulation und Erwartung für unterschiedliche Komponenten des Testobjekts, sorgen die Blöcke für eine bessere Lesbarkeit des Quelltextes. Sind jedoch Testgrößen verschiedener Herkunft *gleichzeitig* relevant, würde der Quelltext aufgrund der nötigen Verschachtelung schnell unleserlich.
Aus diesem Grund kann die einzelne Testgröße in einem Ausdruck auch in der Form

```
[TargetDefinition|ID] '.' [TestQuantity|ID]
```

referenziert werden.

6.3 TYPSYSTEM

interne und externe Datentypen

Der Umgang mit Datentypen in der ETSpec Sprache ist speziell, denn es muss zwischen *internen* und *externen* Typen unterschieden werden. Der Begriff „intern“ umfasst alle Typen, die durch das im Folgenden vorgestellte Typsystem der ETSpec Sprache definiert sind. Bei der Deklaration einer Variablen, Funktion oder Konstanten im ETSpec Quelltext können nur interne Typen verwendet werden. Es wird zwischen primitiven und komplexen Datentypen unterschieden.
Der Begriff „extern“ umfasst Datentypen, die im Rahmen einer fremden Programmiersprache definiert sind, zu denen jedoch Kompatibilität bestehen muss. Im Folgenden erfolgt dazu eine Fokussierung auf die Programmiersprachen C und Java. C ist relevant, weil es sich dabei um die Sprache der getesteten Software handelt. Sie bestimmt mögliche Datentypen der Testgrößen „Variable“ und „Funktion“. Java wird genauer betrachtet, weil es bei der Implementierung als Host-Sprache ausgewählt wurde. Kompatibilität ist erreicht, wenn der externe Typ auf einen internen Typ abgebildet und im weiteren Testablauf als solcher behandelt werden kann.
Die folgende Regel `DataTypeID` beschreibt die Syntax zur Festlegung eines Datentyps in der ETSpec Sprache:

```
1 DataTypeID:
2     'bool' | 'int8' | 'int16' | 'int32' | 'int64' |
3     'uint8' | 'uint16' | 'uint32' | 'uint64' |
4     ('cint' | 'cuint' ) '[' Expression ']' |
5     'pointer' '[' DataTypeID ']' |
6     'physical' '[' [UnitType|UNITID] ']' |
7     [ComplexType|ID] ('<' DataTypeID (',' DataTypeID)* '>')?
```

Die einzelnen Alternativen werden in den folgenden Abschnitten erklärt.

6.3.1 *Primitive Datentypen*

Die primitiven Datentypen der ETSpec Sprache sind in Tabelle 4 aufgelistet. Die Typen im oberen Teil der Tabelle sind selbsterklärend. Die Ganzzahl- und Gleitkommatypen wurden so ausgewählt, dass sie auf PCs mit Standardprozessoren effizient verarbeitet werden können. Die primitiven Typen der Programmiersprache Java sind in der Tabelle ebenfalls aufgeführt und lassen sich 1:1 darauf abbilden.

Tabelle 4: Primitive Datentypen der ETSpec Sprache

ETSpec	Java	Wertebereich / Bemerkung
`bool`	`boolean`	`true`, `false`
`int8`	`byte`	$[-128, 127]$
`int16`	`short`	$[-32.768, 32.767]$
`int32`	`int`	$[-2^{31}, 2^{31}-1]$
`int64`	`long`	$[-2^{63}, 2^{63}-1]$
`uint8`	-	$[0, 255]$
`uint16`	-	$[0, 65.535]$
`uint32`	-	$[0, 2^{32}-1]$
`uint64`	-	$[0, 2^{64}-1]$
`float`	`float`	IEEE 754, einfache Genauigkeit
`double`	`double`	IEEE 754, doppelte Genauigkeit
`char`	`char`	UTF-16 Zeichen
`string`	*String*	UTF-16 Zeichenkette
`cint[n]`	-	$[-2^{n-1}, 2^{n-1}-1]$
`cuint[n]`	-	$[0, 2^{n}-1]$
`event`	-	Benutzerdef. Ereignisse
`pointer[type]`	-	Hilfstyp für Zeiger
`physical[unit]`	-	Gleitkomma-Format für physikalische Größen

Ganzzahlen benutzerdefinierter Größe

Die Abbildung der primitiven Datentypen aus C auf die ETSpec Sprache ist weniger einfach als es für Java der Fall ist. Durch den Sprachstandard von C ist der Wertebereich der einzelnen Typen nicht festgelegt. Stattdessen ist er bestimmt durch die Anzahl Bits, die auf der Zielarchitektur zur Verfügung stehen. In der Regel werden hier ebenfalls 8, 16, 32 oder 64 Bit verwendet, sodass im oberen Bereich der Tabelle bereits ein passender Typ zu finden ist.
Spezielle Architekturen können jedoch davon abweichen. In der ETSpec Sprache muss ein äquivalenter Datentyp zur Verfügung stehen, um Typsicherheit zu gewährleisten. Es werden dafür zwei spezielle Datentypen `cint` und `cuint` angeboten, bei denen die Anzahl der Bits explizit anzugeben ist (das ‚`c`‘ steht für „custom“). Prinzipiell könnten die übrigen Datentypen für ganze Zahlen dadurch

ersetzt werden. Da eine frei gewählte Wortbreite von der Prozessorhardware des Host-PCs jedoch nicht direkt unterstützt wird, sind entsprechende Berechnungen weniger effizient (schließlich müssen Überläufe abgefangen und gesondert behandelt werden). Deswegen ist eine klare Trennung sinnvoll.

Kompatibilität zu alternativen Kodierungen

Einerkomplement vs. Zweierkomplement

Der C Standard ist jedoch nicht nur bezüglich der Größe, sondern auch bezüglich der Kodierung unspezifisch. Zur Darstellung vorzeichenbehafteter ganzer Zahlen kann neben dem Zweierkomplement auch das Einerkomplement verwendet werden. Das ist in der Praxis zwar extrem selten, dennoch unterscheidet sich dadurch der Wertebereich wegen der doppelten Darstellung der Null. Erfolgt die Repräsentation auf der Zielplattform im Einerkomplement, muss der Wert beim Lesen bzw. Schreiben des Speichers der Zielplattform konvertiert werden.
$\mathbb{Z}_{1,n}$ und $\mathbb{Z}_{2,n}$ seien die Mengen der ganzen Zahlen, die sich im Einer- bzw. Zweierkomplement durch n Bit darstellen lassen. Die Konvertierung in das Zweierkomplement entspricht der Abbildung $\mathbb{Z}_{1,n} \to \mathbb{Z}_{2,n}$. Der Wert bleibt dabei erhalten, denn es gilt:

$$\mathbb{Z}_{1,n} \subset \mathbb{Z}_{2,n}$$

Bei der Abbildung $\mathbb{Z}_{2,n} \to \mathbb{Z}_{1,n}$ tritt jedoch ein Wertverlust auf, denn es gilt:

$$\min(\mathbb{Z}_{2,n}) \notin \mathbb{Z}_{1,n}$$

Trotzdem erfolgt die Konvertierung in das Einerkomplement *implizit*, weil für den seltenen Sonderfall der Verwendung des Einerkomplements auf der Zielplattform kein zusätzlicher Typbezeichner eingeführt werden soll. $\min(\mathbb{Z}_{2,n})$ wird dazu auf die 0 abgebildet:

$$\text{conv}_{2\to1}(x) = \begin{cases} 0, & x \notin \mathbb{Z}_{1,n} \\ x, & \text{sonst} \end{cases}$$

Spezielle Gleitkomma-Formate

Die Kodierung von Gleitkomma-Zahlen wird im C Standard ebenfalls nicht genau festgelegt. In den allermeisten Fällen wird jedoch entsprechend IEEE 754 [Std/IEE754] einfacher bzw. doppelter Genauigkeit kodiert. Ist das nicht der Fall, wird der Wert *implizit* von bzw. nach `double` konvertiert, und zwar durch kaufmännisches Runden auf die nächste darstellbare Zahl. Dabei kann es zu einem Genauigkeitsverlust kommen. Ist Runden nicht möglich, z.B. bei

`NaN` (Not a Number) oder $\pm$ unendlich, wird die entsprechende Darstellung im Zielformat verwendet, falls vorhanden, ansonsten ist das Verhalten unspezifiziert.

Benutzerdefinierte Ereignisse

Die ETSpec Sprache bietet den speziellen Datentyp ‚`event`' um benutzerdefinierte Ereignisse zu deklarieren. Benutzerdefinierte Ereignisse dienen der Synchronisation paralleler Abläufe, was in Abschnitt 6.5.4 erklärt wird.

Zeiger

Die Programmiersprache C unterstützt *Zeiger*, durch die auf mitunter teure Kopieroperationen verzichtet werden kann. Zeiger sind jedoch auch eine häufige Fehlerquelle. Grundsätzlich werden Zeiger in der ETSpec Sprache deswegen vermieden.
Trotzdem muss ein entsprechender Hilfstyp `pointer` existieren. Er ist jedoch nur im Rahmen der Interaktion mit der getesteten Software relevant und kann nicht als Zeiger auf ein lokales Objekt verwendet werden. Ein `pointer` speichert eine Speicheradresse sowie den zugehörigen *ScopeProvider*. Letzteres ist erforderlich, damit bei einer späteren Verwendung nicht versehentlich auf den falschen Speicher zugegriffen wird. Nur die Adresse kann zur Laufzeit verändert werden.
Typischerweise wird ein `pointer` initialisiert, indem eine kompatible Testgröße zugewiesen wird. Kompatibel bedeutet, dass es sich um einen Zeiger des gleichen Typs handelt. Ist der Zeiger in C beispielsweise folgendermaßen deklariert,

```
int* i_ptr;
```

dann sieht die Deklaration und Initialisierung eines `pointers` in der ETSpec Sprache folgendermaßen aus:

```
pointer[int32] copyOfPointer = i_ptr;
```

Die Variable `copyOfPointer` übernimmt den *ScopeProvider* und die Adresse von `i_ptr`. Mit C vergleichbare Zeiger-Arithmetik ist über `copyOfPointer` nicht möglich. Die Manipulation der Zieladresse eines Zeigers und Zugriff auf den referenzierten Speicher wird in der ETSpec Sprache stattdessen über spezielle *Attribute* realisiert. Sie existieren auch für andere Elemente der Sprache und werden in Abschnitt 6.3.8 erklärt.

Physikalische Größen

Zur Behandlung physikalischer Größen steht ein spezieller Datentyp `physical` zur Verfügung. Der Wert wird durch ein Gleitkomma-Format gespeichert, das bezüglich Präzision und Wertebereich mindestens `double` entspricht. Bei der Deklaration einer Variablen entsprechenden Typs kann die physikalische Einheit angegeben werden, mit der Werte dieser Variablen assoziiert sind. Physikalische Einheiten werden in Paketen definiert:

```
UnitType:
    'unit' UNITID
        ('[' [UnitType|UNITID] '=' UnitExpression ']')?
    ';'
```

Optional kann bei der Deklaration eine Umrechnungsformel angegeben werden, welche die Abhängigkeit von einer *Basiseinheit* beschreibt. Die Syntax der Umrechnungsformel ist durch die Regel `UnitExpression` festgelegt. Es handelt sich dabei um eine vereinfachte Form der mächtigeren Regel `Expression`, die in Abschnitt 6.4.1 genauer erklärt wird.

```
UnitExpression:
    (
        '(' UnitExpression ')' |
        [UnitType|UNITID] |
        REAL_LITERAL
    )
    (UNIT_OPERATOR UnitExpression)?
```

Als Operatoren sind nur die vier Grundrechenarten erlaubt:

```
UNIT_OPERATOR:
    '+' | '-' | '*' | '/'
```

Dazu ein Beispiel für die physikalische Größe „Temperatur“ anhand der Einheiten „Kelvin“, „Celsius“ und „Fahrenheit“:

```
package Units
{
    unit K;
    unit C [K = C + 273.15];
    unit F [C = (F - 32) / 1.8];
}
```

Die Regel `UNITID` unterscheidet sich von `ID`, welche bisher für Bezeichner verwendet wurde. Durch spezielle Unicode-Zeichen für hochgestellte Zahlen und ein dediziertes Trennzeichen (·) können auch Beziehungen zwischen inkompatiblen Einheiten kodiert werden.

Zum Beispiel kann „$m \cdot s^{-1}$“ als Einheit für die *Geschwindigkeit* definiert werden. Das Ergebnis einer Multiplikation mit dem Typ *Zeit* (s) ist vom Typ *Meter* (m). Diese Information ist durch die Bezeichner implizit gegeben und kann später zur statischen Prüfung des Datentyps verwendet werden (siehe Abschnitt 7.2.3).
Zur Eingabe der Sonderzeichen bedarf es Unterstützung durch den Texteditor, weil das nicht ohne weiteres über die Tastatur möglich ist. Der Bezeichner `UNITID` ist folgendermaßen aufgebaut:

```
1 UNITID:
2     ID ('⁻' UNITPOWER)? (UNITPOWER)*
3     ('·' UNITID)*
```

wobei `UNITPOWER` folgendermaßen definiert ist:

```
1 UNITPOWER:
2     '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'|'0'
```

6.3.2 *Komplexe Datentypen*

Die ETSpec Sprache bietet drei Arten komplexer Datentypen: Aufzählungen, Strukturen und Arrays. C kennt darüber hinaus die Vereinigung (`union`), um verschiedene Sichten auf dieselben Daten zu ermöglichen. In ETSpec gibt es keinen dazu äquivalenten Typ. Das bedeutet, dass auf eine Testgröße vom Typ `union` nicht „als Ganzes“ zugegriffen werden kann, sondern der Zugriff immer auf ein konkretes Element einer Vereinigung erfolgen muss.

Aufzählungen

Ein spezieller Aufzählungstyp ist wichtig, um Werte mit besonderer Semantik zu erfassen. Aufzählungen werden in einem *Package* nach der Regel `EnumType` deklariert:

```
1 EnumType:
2     'enum' ID
3     '{'
4         ( EnumEntry (',' EnumEntry)* )?
5     '}'
```

Die Kodierung kann dabei für jeden Eintrag (`EnumEntry`) explizit angegeben werden. Ebenfalls optional ist die Angabe einer String-Repräsentation, welche zur Ein-/Ausgabe gegenüber einem Anwender nützlich ist:

```
1 EnumEntry:
2     ID ('=' INTEGER_LITERAL)? ('[' STRING_LITERAL ']')?
```

Strukturen

Eine Struktur definiert einen Datentyp, der sich aus anderen Datentypen *beliebig* zusammensetzt. Eine Struktur kann als ein Spezialfall einer Klasse aus der objektorientierten Programmierung aufgefasst werden.
Klassen sind kein Bestandteil der Sprachdefinition von ETSpec, sondern werden in der Host-Sprache deklariert. Die Verwendung von Klassen und entsprechender Objekte ist in der ETSpec Sprache jedoch auf eine transparente Weise möglich, und wird in Abschnitt 6.6 genauer erklärt.
Es steht also bereits ein mächtigeres Beschreibungsmittel für Datenstrukturen zur Verfügung. Trotzdem wird ein zusätzlicher Datentyp für Strukturen benötigt, um ein direktes Äquivalent für den Datentyp aus C zu bieten. Strukturen werden wie Aufzählungen in einem *Package* deklariert:

```
1 StructType:
2     'struct' ID
3     '{'
4         (ElementDeclarator ';')*
5     '}'
```

Die Regel `ElementDeclarator` definiert die Syntax zur Deklaration eines getypten Elements. Sie wird nicht nur für Strukturen, sondern auch für Funktionsparameter und lokale Variablen verwendet:

```
1 ElementDeclarator:
2     DataTypeID ID ('[' (INTEGER_LITERAL)? ']')*
```

Strukturen und Aufzählungen werden bei einer Zuweisung genauso wie primitive Datentypen *kopiert*.

Arrays

Arrays werden häufig zur Speicherung von Daten in Form von Vektoren oder Matrizen verwendet. Grundsätzlich sind dazu jedoch abstrakte Datentypen, also Klassen der Host-Sprache, besser geeignet. Arrays werden in der ETSpec Sprache daher ebenfalls vorwiegend aus Kompatibilitätsgründen zu C angeboten. Sie werden für den Zugriff auf ganze Speicherbereiche auf der Zielplattform benötigt.
Arrays werden wie in C durch eckige Klammern bei der Deklaration gekennzeichnet (siehe Regel `ElementDeclarator`), und ihre Größe ist statisch. Anders als in C erfolgt bei einer Zuweisung immer eine Kopie.

6.3.3 *Literale*

Zur direkten Darstellung von Werten im Quelltext können die *Literale* aus Tabelle 5 verwendet werden. Sie entsprechen den Literalen gängiger Programmiersprachen und werden daher nur kurz vorgestellt. Die exakte syntaktische Definition findet sich in Anhang A.

Tabelle 5: Literale der ETSpec Sprache

Literal	Datentyp	Beispiele
boolean	`bool`	`true` `false`

integer	`int*`
	`uint*`
		`-1234`
		`1234`
		`0xABCDEF`

real	`float`
	`double`
	`physical`
		`1.123f`
		`1.123`
		`-4.0e-2`

character	`char`	`'a'`
		`'é'`
		`'\n'`
		`'\u21AF'`

string	`string`	`"Text"`
		`"Text mit Zeilenumbruch\n"`
		`"'ü' in Unicode: '\u00FC'"`

null	*Object*	`null`

Der Datentyp eines *integer* Literals entspricht dem „kleinsten“ Datentyp, der den numerischen Wert noch aufnehmen kann. Es wird ein vorzeichenbehafteter Datentyp (`int*`) angenommen, genau dann wenn das Vorzeichen angegeben ist. Das Präfix ‚`0x`‘ bzw. ‚`0X`‘ kennzeichnet, dass der numerische Wert hexadezimal dargestellt ist, wobei kein Vorzeichen möglich ist.
Folgt auf ein *real* Literal eine physikalische Einheit, ist der Datentyp `physical`. Ansonsten entspricht der Typ `float`, falls das Suffix ‚`f`‘ angegeben ist, andernfalls `double`. In Zusammenhang mit einer physikalischen Einheit ist das Suffix ‚`f`‘ nicht erlaubt. Üblicherweise wird der Wert dezimal dargestellt, und die binäre Repräsentation ergibt sich durch kaufmännisches Runden. Um die binäre Reprä-

sentation exakt anzugeben (was praktisch nur für Spezialfälle relevant ist, z.B. `NaN` oder größter und betragsmäßig kleinster Wert), steht ein hexadezimales Format zur Verfügung, das der entsprechenden Definition in C99 [Std/C99] entspricht.
Die Literale *character* und *string* entsprechen ihrem jeweiligen Äquivalent in der Programmiersprache Java. Ebenso das *null* Literal, welches für den Umgang mit Objekten benötigt wird (siehe Abschnitt 6.6).

6.3.4 *Konstanten*

Konstanten werden verwendet, um unveränderliche Größen mit einem aussagekräftigen Namen zu versehen. Konstanten werden in *Packages* deklariert, und es muss dabei ein Ausdruck für den Wert angegeben werden.

```
1 ConstantDeclaration:
2     'const' ElementDeclarator '=' Expression ';'
```

Der Ausdruck wird beim Laden des Pakets ausgewertet und die Konstante initialisiert. Danach ist eine Änderung des Werts nicht mehr möglich.

6.3.5 *Variablen*

Variablen dienen der Zwischenspeicherung von Werten. Variablen können in *Packages* deklariert werden, dann handelt es sich um eine *globale* Variable, oder in den Implementierungsblöcken eines Tests bzw. einer Funktionen, dann handelt es sich um eine *lokale* Variable.

```
1 VariableDeclaration:
2     'capture'? ElementDeclarator ('=' Expression)? ';'
```

Globale Variablen müssen bei der Deklaration explizit initialisiert werden, lokale Variablen vor dem ersten Lesezugriff. Sowohl lokale als auch globale Variablen verlieren ihren Wert, sobald der Test beendet bzw. die Funktion verlassen wird.
Globale Variablen sind jederzeit sichtbar. Lokale Variablen sind nur innerhalb des Implementierungsblocks sichtbar, in dem sie deklariert wurden. Eine Ausnahme stellen lokale Variablen dar, die mit dem Schlüsselwort ‚`capture`‘ deklariert wurden. Es steht nur im `AnalyzeBlock` zur Verfügung (siehe Offline Analyse in Abschnitt 6.5.7).

6.3.6 *Funktionen*

Der Begriff Funktion wird in dieser Arbeit synonym zum Begriff *Unterprogramm* verwendet. Funktionen dienen der Strukturierung und Wiederverwendung von Quelltext und werden entsprechend folgender Syntax deklariert:

```
FunctionDeclaration:
    ('realtime' | 'analyze')? ('void' | DataTypeID) ID
      '(' (FunctionParameter (',' FunctionParameter)*)? ')'
      ('default' [TargetDefinition|ID])?
    '{'
        Statement*
    '}'
```

Den Bezeichner, Datentyp einer möglichen Rückgabe sowie Anzahl, Datentyp, und Reihenfolge der Parameter wird als *Signatur* der Funktion bezeichnet. Besitzt eine Funktion keinen Rückgabewert, wird dies durch das Schlüsselwort ‚`void`' gekennzeichnet. Durch das Schlüsselwort ‚`default`' kann ein *ScopeProvider* angegeben werden (vgl. Abschnitt 6.2.4).
Bezüglich der Funktionsparameter existieren einige Besonderheiten, wie die Regel `FunctionParameter` zeigt:

```
FunctionParameter
    ('ref' ('variable' | 'signal' | 'message' |
            'function' SignatureHint?)?)?
    ElementDeclarator
```

Das Schlüsselwort ‚`ref`' gibt an, dass die Übergabe von Funktionsparametern als Referenz erfolgt (*call-by-reference*). Standardmäßig geschieht das nur für Objekte der Host-Sprache, und in allen anderen Fällen erfolgt eine Kopie des Werts (*call-by-value*). Die Schlüsselworte ‚`variable`', ‚`function`', ‚`signal`' und ‚`message`' sind erforderlich, damit in diesem Fall die Information über die Art der Testgröße nicht verloren geht. Für Testgrößen der Art „Funktion" kann zusätzlich die erwartete Signatur angegeben werden:

```
SignatureHint
    ('void' | DataTypeID)
    '(' (DataTypeID (',' DataTypeID)*)? ')'
```

Die Signatur darf entfallen, falls innerhalb der ETSpec Funktion die referenzierte Testgröße Funktion nicht aufgerufen wird, sondern nur auf entsprechende Attribute Ereignisse (siehe Abschnitt 6.3.8 und Abschnitt 6.3.9) zugegriffen wird.

6.3.7 *Typumwandlung*

Die Datentypen von Operanden einer arithmetischen Operation müssen identisch sein. Die Datentypen der Argumente bei einem Funktionsaufruf und der Rückgabewert einer Funktion müssen mit der Signatur der Funktion übereinstimmen. Das kann eine Typumwandlung erforderlich machen. Diese hat in einigen Fällen keinen Einfluss auf den Wert (z.B. `int16` → `int32`), oder ist mit vertretbarem Genauigkeitsverlust verbunden (z.B. `int32` → `double`).
In solchen Fällen wird in der ETSpec Sprache implizit konvertiert. Genauer gesagt wird immer dann implizit konvertiert, wenn der angegebene Typ im angeforderten Typ „enthalten“ ist, und zwar entsprechend der folgenden Beziehungen:

`physical` ⊃ `double` ⊃ `float` ⊃ `int64` ⊃ ⋯
⋯ ⊃ `int32` ⊃ `int16` ⊃ `int8`

und

`physical` ⊃ `double` ⊃ `float` ⊃ `uint64` ⊃ `uint32` ⊃ ⋯
⋯ ⊃ `uint16` ⊃ `char` ⊃ `uint8`

Die benutzerdefinierten Ganzzahltypen `cint[n]` und `cuint[n]` werden ebenfalls implizit konvertiert, und ihre Position in der vorhergehenden Ordnung ergibt sich aus dem Parameter `n`. Implizit konvertiert werden außerdem Objekte, wenn die angeforderte Klasse eine Generalisierung der Klasse des angegebenen Objekts ist. Aufzählungen, Strukturen und Arrays werden niemals implizit konvertiert.
Eine explizite Umwandlung ist durch den `cast` Operator möglich. Er ist in ETSpec durch spitze Klammern gegeben:

```
CastOperator: '<' DataTypeID '>'
```

Bei der expliziten Konvertierung kann ein Wertverlust auftreten. Ein `pointer` kann nur in einen anderen Zeigertyp oder `string` umgewandelt werden. Eine Konvertierung von `physical` in einen Datentyp ohne physikalische Einheit liefert immer den Wert in der zugehörigen Basiseinheit.
Aufzählungstypen können explizit in einen Werttyp konvertiert werden. Objekte können explizit in eine andere Klasse konvertiert werden, wenn zwischen dieser und der angegebenen Klasse eine Vererbungsbeziehung besteht. Weil sich der Typ eines Objekts möglicherweise erst zur Laufzeit ergibt, kann es dabei zu einem Laufzeitfehler kommen. Für Strukturen und Arrays ist der `cast` Operator nicht verwendbar.

6.3.8 *Attribute*

Einige Elemente der ETSpec Sprachen bieten spezielle *Attribute*. Über sie ist der Zugriff auf spezifische Eigenschaften eines Elements oder die Konfiguration der Laufzeitumgebung möglich. Ein Beispiel wurde in Zusammenhang mit dem primitiven Datentyp `pointer` auf Seite 118 bereits erwähnt: Variablen vom Typ `pointer` bieten Attribute für den Zugriff auf die Zieladresse bzw. auf die von einem Zeiger referenzierten Daten.

Sprachelement „Runtime“

Darüber hinaus sind die in Tabelle 6 genannten Attribute verfügbar. Das Element `Runtime` in der ersten Zeile der Tabelle wurde noch nicht erklärt. Es ist in jedem Implementierungsblock implizit verfügbar und dient dem Zugriff auf Attribute der ETSpec Laufzeitumgebung. `GlobalTime` liefert die seit Start der Testausführung vergangene Zeit. `Timestamp` und `EOT` stehen in Zusammenhang mit Echtzeit Stimulation und Offline Analyse, und werden daher in Abschnitt 6.5.7 erklärt.

Tabelle 6: Standard Attribute

ETSpec Element	Attribute
`Runtime`	`GlobalTime`, `Timestamp`, `EOT`
Array	`Count`
Zeiger	`DstAddress`, `DstData`
ScopeProvider *ProcessorCore*	`InstructionPointer`
ScopeProvider *CANInterface*	`BaudRate`, `BTR0`, `BTR1`
Testgröße *Variable*	`Address`
Testgröße *Funktion*	`Address`
Testgröße *Signal*	`TriggerAbove`, `TriggerBelow`, `TriggerRising`, `TriggerFalling`, `SampleRate`

`Count` liefert die Anzahl der Elemente eines Arrays. `DstAddress` und `DstData` bieten Zugriff auf die Zieladresse des Zeigers und die referenzierten Daten. `InstructionPointer` bietet Zugriff auf den entsprechenden Register des Prozessorkerns. `Address` liefert die Adresse der Variablen bzw. Funktion im Speicher der Zielplattform.

Mit den Attributen der Testgröße *Signal* können Details zur Aufzeichnung des Signalverlaufs konfiguriert werden. Mit den Attribu-

ten `TriggerAbove` bzw. `TriggerBelow` wird ein Schwellwert gesetzt, bei dessen über- bzw. unterschreiten die Aufzeichnung gestartet wird. Über die Attribute `TriggerRising` und `TriggerFalling` wird ein Trigger für die steigende bzw. fallende Flanke gesetzt, es ist der Datentyp `bool` zu verwenden (‚`true`' aktiviert den Trigger). Mit `SampleRate` wird die Abtastrate konfiguriert.
Weitere Attribute können durch Treiber für Zugriffshardware registriert werden. Damit wird dem Anwender die Verwendung spezieller Features ohne den Umweg über die Host-Sprache ermöglicht. Ein Beispiel für den Zweck einer solchen Möglichkeit ist die Konfiguration eines Triggers zur Aufzeichnung des Signalverlaufs. Weil sich die Möglichkeiten dazu je nach Zugriffshardware stark unterscheiden, kann mit den Standardattributen aus Tabelle 6 das Potenzial nicht vollständig genutzt werden.
Zugegriffen wird auf Attribute im ETSpec Quelltext syntaktisch durch einen Doppelpunkt folgend auf den Bezeichner des betroffenen Elements. Zum Beispiel wird durch den folgenden Code das Ziel eines Zeigers gesetzt:

```
1 pointer[int32] myPtr;
2 myPtr:DstAddress = 0x80000000;
```

6.3.9 *Ereignisse*

Auf Seite 118 wurden bereits *benutzerdefinierte* Ereignisse erwähnt (primitiver Datentyp `event`). Darüber hinaus bietet die ETSpec Sprache Zugriff auf bestimmte *vordefinierte* Ereignisse, die von der Laufzeitumgebung ausgelöst werden. Sie sind in Tabelle 7 aufgeführt.

Tabelle 7: Vordefinierte Ereignisse der ETSpec Sprache

ETSpec Element	Mögliche Ereignisse
Testgröße *Signal*	`updated`
Testgröße *Nachricht*	`received`, `sent`
Testgröße *Variable*	`read`, `written`
Testgröße *Funktion*	`called`, `returned`

Syntaktisch wird auf vordefinierte Ereignisse ähnliche zugegriffen, wie auf Attribute. Statt des Doppelpunkts wird der Apostroph (`'`) verwendet. Vordefinierte Ereignisse können nicht beliebig in Ausdrücken verwendet werden, sondern sind nur im Rahmen der *wait-*

until Anweisung (siehe Abschnitt 6.5.3) erlaubt. Benutzerdefinierte Ereignisse sind außerdem im Rahmen der *trigger* Anweisung verwendbar (siehe Abschnitt 6.5.4).

6.4 ALLGEMEINE ANWEISUNGEN

Die Implementierung eines Testablaufs in der ETSpec Sprache besteht aus Anweisungen (Statements), wie bei imperativen Programmiersprachen üblich. Jede Anweisung lässt sich in eine Folge von Instruktionen übersetzen. In diesem Abschnitt sind alle Anweisungen der ETSpec Sprache zusammengefasst, die sich auch in GPLs wiederfinden, und für die gewünschte Mächtigkeit und Flexibilität der Sprache notwendig sind.
Die Standardkonstrukte orientieren sich stark an Java, kleinere syntaktische Details wurden von C# übernommen. Die speziellen Anweisungen der ETSpec Sprache hinsichtlich domänenspezifischer Anforderungen sind in Abschnitt 6.5 zu finden.

6.4.1 *Ausdrücke und Operatoren*

Ein wiederkehrender Bestandteil von Anweisungen sind Ausdrücke (Expressions). Ein Ausdruck ist eine Kombination von Literalen, Konstanten, Variablen, Funktionen und Operatoren, die unter Berücksichtigung der Operatorrangfolge ausgewertet werden, und einen Wert liefern [Wiki/Exp]. Ausdrücke müssen folgender Grammatik entsprechen:

```
1 Expression:
2     UnaryOperator?
3     (
4         ANY_LITERAL |
5         '(' Expression ')' |
6         'new' DataTypeID '(' CallArguments? ')' |
7         [DeclaredElement|FQN]) ('(' CallArguments? ')')?
8     )
9     (BINARY_OPERATOR Expression)?
```

`CallArguments` setzt sich erneut aus Ausdrücken zusammen:

```
1 CallArguments:
2     Expression (',' Expression)*
```

Die möglichen Operatoren sind in Tabelle 8 aufgelistet. Verhalten und Rangfolge sind identisch mit der Programmiersprache Java. Zeile 5 der Regel `Expression` ermöglicht beliebige Klammerung und somit ein Überschreiben der Rangfolge.

Tabelle 8: Alle Operatoren der ETSpec Sprache

Gruppe	Operatoren nach Rang
Unäre Operatoren	`-`, `~`, `!`, `<Type>` (Typumwandlung)
Punktrechnung	`*`, `/`, `%`
Strichrechnung	`+`, `-`, `+` für `string`-Konkatenation
Bitshift	`<<`, `>>`, `>>>`
Vergleich	`<`, `<=`, `>`, `>=`, `is`[1], `==`, `!=`
Bitweise Op.	`&`, `^`, `|`
Logische Op.	`&&`, `||`
Bedingter Op.	`? :`
Zuweisungs-Op.	`=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=`, `|=`

[1] Entspricht `instanceof` in Java (prüft ob Instanz von angegebener Klasse erbt)

6.4.2 *If-Else Anweisung*

Die *if-else* Anweisung dient der Verzweigung des Kontrollflusses anhand einer Bedingung. Die Anweisung wird durch einen `if` Block eingeleitet, auf den beliebig viele `else if` Blöcke folgen können. Durch einen Ausdruck in Klammern wird jeweils eine Bedingung angegeben, die der Reihe nach ausgewertet werden. Es wird der erste Block ausgeführt, dessen Bedingung `true` ergibt. Trifft keine Bedingung zu, wird der `else` Block ausgeführt, sofern er angegeben ist. Die Syntax ist folgendermaßen definiert:

```
IfElseStatement:
    IfBlock (=> ElseIfBlock)* (=> ElseBlock)?
```

Wobei für die einzelnen Blöcke folgende Regeln gelten:

```
IfBlock:
    'if' '(' Expression ')'
    ('{' Statement* '}' | Statement)

ElseIfBlock:
    'else' 'if' '(' Expression ')'
    ('{' Statement* '}' | Statement)

ElseBlock:
    'else'
    ('{' Statement* '}' | Statement)
```

6.4.3 *Switch-Case Anweisung*

Die *switch-case* Anweisung realisiert eine Sprungtabelle, die in einigen Fällen schneller ausgewertet werden kann als eine äquivalente *if-else* Anweisung, und häufig auch einfacher zu lesen ist.

```
SwitchCaseStatement:
    'switch' '(' Expression ')'
    '{' (CaseBlock)* (DefaultBlock)? '}'
```

Für die einzelnen Blöcke gelten die folgenden Regeln:

```
CaseBlock:
    'case' (INTEGER_LITERAL | STRING_LITERAL |
            [EnumEntry|FQN])
        ':' Statement*

DefaultBlock:
    'default' ':' Statement*
```

Es wird nach Auswertung des bei `switch` in Klammern angegebenen Ausdrucks einer von mehreren `case` Fällen direkt angesprungen, und alle folgenden Anweisungen werden ausgeführt. Durch die *break* Anweisung (‚`break;`') wird an das Ende des `switch` Blocks gesprungen. Optional kann mit `default` ein Fall angegeben werden, der angesprungen wird, falls der gesuchte Wert nicht aufgeführt ist. Der bei `switch` angegebene Ausdruck muss dem Datentyp `uint32`, `string` oder einem Aufzählungstyp entsprechen.

6.4.4 *For Schleife*

Die *for* Schleife dient der Realisierung „zählender" Schleifen.

```
ForLoopStatement:
    'for' '(' DeclList? ';' Expression ';' ExpList? ')' '{'
        Statement*
    '}'
```

Die Regeln für den Schleifenkopf sind folgendermaßen definiert:

```
DeclList:
    InitializedDeclarator (',' InitializedDeclarator)*

ExpList:
    Expression (',' Expression)*

InitializedDeclarator:
    ElementDeclarator '=' Expression
```

Die Schleife wird betreten und erneut durchlaufen, genau dann wenn der Ausdruck an zweiter Stelle im Schleifenkopf `true` ergibt. Der erste Abschnitt des Schleifenkopfs dient der Deklarationen und Initialisierung lokaler Variablen, die nur im Schleifenrumpf sichtbar sind. Er wird genau einmal beim Schleifeneintritt ausgeführt. Eine Besonderheit der ETSpec Sprache besteht darin, dass es möglich ist, mehrere Variablen *unterschiedlichen Typs* zu deklarieren, was z.B. in Java nicht der Fall ist.
Die Ausdrücke im dritten und letzten Teil des Schleifenkopfs werden vor jeder weiteren Iteration ausgeführt. Durch die *break* Anweisung wird die Schleife außerordentlich abgebrochen, die *continue* Anweisung setzt mit der nächsten Iteration fort.

6.4.5 *Foreach Schleife*

Die *foreach* Schleife dient der Iteration über alle Elemente eines damit kompatiblen Typs. Das sind eindimensionale Arrays sowie Objekte der Host-Sprache, die ein entsprechendes Interface implementieren (für Java z.B. `java.lang.Iterable` [Onl/JDoc]).

```
1 ForeachLoopStatement:
2     'foreach' '('ElementDeclarator 'in' Expression')' '{'
3         Statement*
4     '}'
```

Ist der Ausdruck nach dem Schlüsselwort ‚`in`' ein Array, wird es anhand des Index durchlaufen (beginnend mit `0` bis `array:Count-1`), andernfalls wird der entsprechende Iterator genutzt (für Java z.B. `java.util.Iterator`). Die Anweisungen *break* und *continue* funktionieren auch hier.

6.4.6 *While Schleifen*

Die *while* Schleife existiert in zwei Varianten. Die kopfgesteuerte Form besitzt folgende Syntax:

```
1 WhileLoopStatement:
2     'while' '(' Expression ')' '{' Statement* '}'
```

Die fußgesteuerte Variante wird durch ‚`do`' eingeleitet:

```
1 DoWhileLoopStatement:
2     'do' '{' Statement* '}' 'while' '(' Expression ')' ';'
```

Die Schleifen werden solange durchlaufen, bis der angegebene Ausdruck `false` ergibt. Auch in *while* Schleifen stehen die Anweisungen *break* und *continue* zur Verfügung.

6.4.7 *Ausnahmebehandlung*

Mit Hilfe der Host-Sprache wird die Behandlung von Ausnahmen unterstützt. ETSpec bietet dazu die bekannte *try-catch* Klausel:

```
TryCatchFinallyStatement:
    TryBlock (CatchBlock)* (FinallyBlock)?

TryBlock:
    'try' '{' Statement* '}'

CatchBlock:
    'catch' '('ElementDeclarator (',' ElementDeclarator)*')'
    '{' Statement* '}'

FinallyBlock:
    'finally' '{' Statement* '}'
```

Beim Auftreten einer Ausnahme im *try*-Block wird die Ausführung abgebrochen und in den zugehörigen *catch*-Block gesprungen. Der *finally*-Block wird anschließend ausgeführt, und zwar auch dann, wenn keine Ausnahme aufgetreten ist, also nach erfolgreicher Abarbeitung des *try*-Blocks.

6.5 SPEZIELLE ANWEISUNGEN

Die ETSpec Sprache bietet eine ganze Reihe von Anweisungen, die bezüglich der domänenspezifischen Anforderungen aus Abschnitt 5.1 notwendig sind, und sich in aktuellen GPLs nicht finden. In Abschnitt 6.2.4 wurde mit der *scope* Anweisung bereits ein Vertreter vorgestellt. Die übrigen Anweisungen sind im Folgenden zusammengefasst.

6.5.1 *Meldungen und Testbewertung*

Die Bewertung eines Tests erfolgt auf Basis der abgesetzten Meldungen während der Ausführung. Dazu stehen zwei Anweisungen zur Verfügung. Die *report* Anweisung besitzt folgende Syntax:

```
ReportStatement:
    'report' Severity (':' Expression)? ';'
```

Sie erzeugt eine Meldung der angegebenen Schwere (Severity, s.u.). Eine Alternative ist die *assert* Anweisung:

```
AssertStatement:
    'assert' '(' Expression ')' Severity(':' Expression)?';'
```

Sie erzeugt nur dann eine Meldung, wenn der in Klammern angegebene Ausdruck `false` ergibt. *Assert* ermöglicht es der Laufzeitumgebung, die Bedingung und die Belegung der Variablen zu sichern, die zu der Meldung geführt haben.
`Severity` ist eines der folgenden Schlüsselworte:

```
Severity:
    'info' | 'warning' | 'error' | 'fatal'
```

Ein Test gilt als *bestanden*, wenn während der Ausführung keine Meldungen oder nur Meldungen der Schwere ‚`info`' erzeugt wurden. Wurde mindestens eine Meldung der Schwere ‚`error`' oder ‚`fatal`' erzeugt, gilt der Test als *fehlgeschlagen*. Ansonsten gilt der Test als *unentschieden*. Nach einer Meldung der Schwere ‚`fatal`' wird die Testausführung abgebrochen, ansonsten fortgesetzt.
Beim Ausdruck nach dem Doppelpunkt wird eine Beschreibung der Meldung für spätere Berichte erwartet. Der Ausdruck muss vom Typ `string` sein.

6.5.2 *Warten einer Zeitspanne*

Die *wait* Anweisung blockiert den Testablauf für die angegebene Zeitspanne.

```
WaitStatement:
    'wait' Expression ';'
```

Der Ausdruck muss kompatibel zu `physical[s]` sein, d.h. eine von „Sekunde“ abgeleitete Einheit besitzen.

6.5.3 *Warten auf Ereignisse und Bedingungen*

Die *wait-until* Anweisung blockiert bis zum Eintreten einer Bedingung bzw. eines Ereignisses (vgl. Abschnitt 6.3.9).

```
WaitUntilStatement:
    'wait' 'until' Expression (',' Expression)*
        ('timeout' Expression)? ';'
```

Es wird jeweils ein boolescher Ausdruck erwartet. Der &&-Operator entspricht dabei Gleichzeitigkeit, was jedoch nicht exakt ausgewertet werden kann. Stattdessen werden die Bedingungen zyklisch geprüft, sowie bei jedem Eintritt eines Ereignisses (da dies asynchron geschieht). Im Rahmen der Offline Analyse (siehe Abschnitt 6.5.7) ist das Vorgehen ein Anders.
Es können mehrere Ausdrücke getrennt durch ein Komma angegeben werden, um eine Folge von Ereignissen bzw. Bedingungen ab-

zuwarten, von denen die Eintrittsreihenfolge egal bzw. unbekannt ist. Durch das Schlüsselwort ‚`timeout`' wird eine Zeitspanne definiert, nach der eine Ausnahme ausgelöst wird.

6.5.4 *Nebenläufigkeit und Synchronisation*

Die Beschreibung paralleler Abläufe wird durch die Anweisungen *concurrent*, *cancel* und *trigger* vereinfacht. Die Syntax der *concurrent* Anweisung ist wie folgt definiert:

```
ConcurrentStatement:
    ConcurrentBlock (FinallyBlock)?

ConcurrentBlock:
    'concurrent' Thread? ([TargetDefinition|ID])?
    '{' Statement* '}'
```

wobei `Thread` direkt durch einen Bezeichner (`ID`) gegeben ist. Der durch *concurrent* eingefasste Block wird nebenläufig ausgeführt, d.h. es wird ein neuer Ausführungsstrang (Thread) abgezweigt. Ist der Ablauf beendet, wird der *finally*-Block ebenfalls noch im Kontext des neu erstellten Threads ausgeführt. Der angegebene *ScopeProvider* hat den gleichen Zweck wie bei der *scope* Anweisung. Der neu erstellte Thread kann explizit mit einem Namen (`ID`) versehen werden, sodass er durch die *cancel* Anweisungen gezielt abgebrochen werden kann. Hierzu wird im entsprechenden Thread eine Ausnahme ausgelöst, und anschließend gewartet, bis der *finally*-Block abgearbeitet wurde.

```
CancelStatement:
    'cancel' [Thread|ID] (',' [Thread|ID])* ';'
```

Zur Synchronisation mehrerer Threads können benutzerdefinierte Ereignisse genutzt werden. Durch die *trigger* Anweisung wird ein benutzerdefiniertes Ereignis „ausgelöst“. Die Anweisung erwartet einen Ausdruck vom Typ `event`, sowie optional einen zweiten Ausdruck vom Typ `uint32`:

```
TriggerStatement:
    'trigger' Expression (',' Expression)? ';'
```

Threads, welche mit *wait-until* auf das Ereignis gewartet haben, werden dadurch fortgesetzt. Wird das zweite Argument (echt größer Null) angegeben, wird nur eine entsprechende Anzahl Threads fortgesetzt, und zwar in der Reihenfolge, in der sie sich auf das Ereignis registriert haben (first-come-first-served). Dabei merkt sich die Laufzeitumgebung einen möglichen Rest. Threads, die sich erst

nach dem Auslösen auf das Ereignis registrieren, werden dann gegebenenfalls direkt fortgesetzt.

6.5.5 *Ausführungskontrolle*

Ist der aktuelle *ScopeProvider* eine *SoftwareExecution* (Abschnitt 5.3.1), stehen die Anweisungen *run*, *run-until*, *stop* und *stop-at* zur Verfügung. Sie ermöglichen die Ausführungskontrolle einer durch *SoftwareExecution* repräsentierten Komponente des Testobjekts, d.h. einen Prozessorkern oder Thread.
Die *run* Anweisung startet die *SoftwareExecution* und blockiert solange, bis die *SoftwareExecution* stoppt (z.B. an einem Breakpoint). War die Ausführung bereits gestartet, hat die Anweisung keine Auswirkung auf das Testobjekt und blockiert ebenfalls.

```
1 RunStatement:
2     'run' ('timeout' Expression)? ';'
```

Die *run-until* Anweisung setzt vor dem Start einen Breakpoint an der durch einen *ExecutionPoint* definierten Position. Ein *ExecutionPoint* ist die Abstraktion einer Positionen im Kontrollfluss der getesteten Software. Funktionen der Software können direkt als *ExecutionPoint* verwendet werden, die entsprechende Position ist der Einsprungpunkt (entry point). Aber auch andere *ExecutionPoints* sind denkbar, beispielsweise eine bestimmte Zeile im Quelltext. Sie müssen dazu im gleichen Modell beschrieben werden, wie die bestehenden Testgrößen. Die *run-until* Anweisung blockiert, bis die *SoftwareExecution* den Breakpoint erreicht hat.

ExecutionPoint

```
1 RunUntilStatement:
2     'run' 'until' Breakpoint
3         ('timeout' Expression)? ';'
4
5 Breakpoint:
6     [ExecutionPoint|ID] ('?' Expression)?
```

Durch ‚?' wird der Breakpoint mit einer Bedingung versehen. Ist die Bedingung beim Erreichen des *ExecutionPoints* `false`, wird die Ausführung fortgesetzt, und *run-until* bleibt im blockierten Zustand.
Durch die *stop* Anweisung wird die *SoftwareExecution* gestoppt, falls sie aktuell im Zustand laufend ist, andernfalls hat die Anweisung keine Auswirkung.

```
1 StopStatement:
2     'stop' ';'
```

Durch *stop-at* wird ein Breakpoint analog zu *run-until* gesetzt und anschließend blockiert.

```
1 StopAtStatement:
2     'stop' 'at' Breakpoint
3         ('timeout' Expression)? ';'
```

Durch das Schlüsselwort ‚`timeout`' wird in allen Fällen eine zeitliche Abbruchbedingung angegeben.
Eine weitere Form der Ausführungskontrolle ist der entfernte Funktionsaufruf (RPC). Syntaktisch ist er nicht anders gekennzeichnet als ein „normaler" Funktionsaufruf, d.h. durch ein Klammerpaar für die Argumente der Funktion (vgl. Regel `Expression` auf Seite 128, Zeile 7). Die *SoftwareExecution*, welche die aufgerufene Funktion ausführt, ist durch den im Kontext eindeutig festgelegt.

6.5.6 *Echtzeit Stimulation*

Die bisher vorgestellten Anweisungen erlauben höchste Flexibilität. Eine Ausführung unter Einhaltung harter Echtzeitanforderungen ist jedoch nicht realistisch, weil durch das Betriebssystem des Host-PCs, die verwendeten APIs und die Zugriffshardware Latenzen entstehen, die kaum vorhersagbar sind.
Die ETSpec Sprache bietet deswegen die *realtime* Anweisung, um Stimulation unter Echtzeitanforderungen zu ermöglichen. Sie ist folgendermaßen aufgebaut:

```
1 RealtimeStatement:
2     RealtimeBlock (AnalyzeBlock | PlotBlock)*
```

`AnalyzeBlock` und `PlotBlock` gehören zur Offline Analyse und werden erst im folgenden Abschnitt beschrieben. Der `RealtimeBlock` enthält die in Echtzeit auszuführende Stimulation:

```
1 RealtimeBlock:
2     'realtime' ('until' Expression)? ('timeout' Expression)?
3     '{' Statement* '}'
```

Durch das Schlüsselwort ‚`until`' wird eine Abbruchbedingung definiert. Es gelten dabei die gleichen Randbedingungen wie bei der *wait-until* Anweisung weiter oben.

Ausführungsprinzip

Die angegebene Implementierung wird noch *vor* der tatsächlichen Ausführung unter Einbeziehung des Testobjekts in eine einfache Folge von Stimuli mit Zeitstempeln übersetzt. Grundlage hierzu bilden die speziellen Anweisungen bezüglich Zeit und Nebenläufigkeit. Der Haupttestablauf sowie jeder durch die *concurrent* Anweisung gestartete Thread besitzen einen privaten, von den anderen Threads unabhängigen Zeitstempel. Dieser wird durch die *wait*

Anweisung um die angegebene Zeitspanne erhöht. Die *trigger* Anweisung setzt den Zeitstempel anderer Threads auf den Wert des eigenen Zeitstempels, falls dort zu einem früheren Zeitpunkt mittels *wait-until* auf das entsprechende benutzerdefinierte Ereignis gewartet wurde.
Während dieser Vorberechnung wird für jeden Schreibzugriff auf eine Testgröße sowie Ausführungskontrolle ein *Stimulus* erzeugt und auf einem Zeitstrahl vermerkt. Danach wird die Testumgebung zuerst anhand des erhaltenen Zeitstrahls konfiguriert, bevor die Testausführung zeitlich synchronisiert fortgesetzt wird.

Einschränkungen

Aufgrund dieses Vorgehens ist dabei keine Beschreibung von closed-loop Verhalten möglich. Anweisungen, die auf das Testobjekt „reagieren“, sind nicht erlaubt, d.h. *run-until*, *stop-at* und das Lesen von Testgrößen ist ebenso verboten, wie die Verwendung der vordefinierten Ereignisse aus Tabelle 7. Die Anweisungen *run* und *stop* sowie RPC blockieren nicht mehr. Auf den Rückgabewert eines RPCs kann nicht zugegriffen werden.
Bei der Implementierung kann auf Funktionen aus einem *Package* zurückgegriffen werden. Sollen dort zeitrelevante Anweisungen oder Testgrößen verwendet werden, muss die Funktion durch das Schlüsselwort ‚`realtime`‘ gekennzeichnet sein. Derartige Funktionen dürfen außerhalb eines `RealtimeBlocks` nicht verwendet werden.
An dieser Stelle ist noch einmal anzumerken, dass die echtzeitfähige Ausführung des beschriebenen Szenarios stark abhängig von der verfügbaren Zugriffshardware ist. Während HiL Systeme und die ASAM XIL API im Speziellen eine auf Zeitstempeln basierte Stimulation gut unterstützt, ist das für Ausführungskontrolle mit aktueller Debug-Hardware kaum möglich, auch weil auf dieser Ebene sehr viel schärfere Echtzeitanforderungen bestehen (vgl. Anforderung A5). Ein möglicher Ansatz zur Umgehung dieser Einschränkung ist die Generierung entsprechenden Target-Codes, wie es in Abschnitt 9.1.1 skizziert ist.

6.5.7 *Offline Analyse*

Die programmatische Analyse aufgezeichneter Daten wird durch `AnalyzeBlocks` der *realtime* Anweisung unterstützt:

```
1 AnalyzeBlock:
2     'analyze'
3     '{' Statement* '}'
```

In diesem Block wird auf Daten zugegriffen, die während der Stimulation aufgezeichnet wurden. Durch das Lesen einer Testgröße wird also nicht auf das Testobjekt zugegriffen, sondern der Wert zu einem bestimmten Zeitpunkt in der Aufzeichnung (Analysezeitpunkt) geliefert. Der Analysezeitpunkt wird analog zur Echtzeit Stimulation durch die Anweisungen *wait*, *trigger* und *wait-until* beeinflusst. Die Anweisungen zur Ausführungskontrolle dürfen nicht verwendet werden, das Schreiben von Testgrößen und RPC sind ebenfalls nicht möglich.

Konfiguration der Aufzeichnung

Es ist die Aufgabe des ETSpec Frameworks alle zur Offline Analyse notwendigen Daten aufzuzeichnen. Hierzu werden vor dem Start der Stimulation alle *AccessAPIs*, zu denen im *Target* des Tests ein *ScopeProvider* vorhanden ist, darüber informiert, welche Testgrößen in mindestens einem der `AnalyzeBlocks` und `PlotBlocks` referenziert werden.

Es ist möglich, dass zur Aufzeichnung einer Testgröße keine geeignete *AccessAPI* zur Verfügung steht, z.B. weil die Zugriffshardware die gewünschte Datenaufzeichnung nicht unterstützt. Ein solcher Testablauf ist ungültig und es wird eine Ausnahme ausgelöst.

Attribute „Timestamp“ und „EOT“

Im `AnalyzeBlock` stehen über das Sprachelement `Runtime` die Attribute `Timestamp` und `EOT` zur Verfügung (vgl. Abschnitt 6.3.8). Sie liefern den aktuellen Analysezeitpunkt bzw. einen booleschen Wert, der angibt, ob das Ende der Aufzeichung erreicht ist („End Of Trace“). Das Attribut `Timestamp` steht auch im `RealtimeBlock` zur Verfügung.

Auch Analysefunktionalität kann ich *Packages* ausgelagert werden. Entsprechende Funktionen sind durch das Schlüsselwort ‚`analyze`‘ zu kennzeichnen, falls dort auf Testgrößen zugegriffen wird, oder zeitrelevante Anweisungen verwendet werden. Derartige Funktionen können außerhalb eines `AnalyzeBlocks` nicht verwendet werden.

Aufzeichnung lokaler Variablen

Lokale Variablen können der Aufzeichnung hinzugefügt werden. Die Variable muss dazu im `AnalyzeBlock` mit dem Schlüsselwort ‚`capture`‘ deklariert werden. Entsprechende Variable sind in den folgenden Blöcken derselben *realtime* Anweisung sichtbar, dürfen jedoch nur im Block der Deklaration beschrieben werden.

Grafische Darstellung

In einigen Fällen ist eine grafische Darstellung der Wertverläufe in den aufgezeichneten Daten gewünscht, z.B. für Testberichte oder bei der Fehlersuche. Die ETSpec Sprache erleichtert die Aufbereitung entsprechender Grafiken durch den `PlotBlock` der *realtime* Anweisung:

```
1 PlotBlock:
2     'plot' ID RangeExpression?
3     '{' PlotEntry* '}'
4
5 PlotEntry:
6     ID ':' Expression RangeExpression? ';'
```

Der Block erzeugt für jeden in `PlotEntry` gegebenen Ausdruck eine Kurve über die Zeit. Der Ausdruck muss einem primitiven Zahlentyp (z.B. `double` oder `physical`) entsprechen. Durch `RangeExpression` wird der darzustellende Wertebereich für die jeweilige Achse festgelegt:

```
1 RangeExpression:
2     'range' ('['Expression ',' Expression']')?
3     ('ln' | 'lg')? ('hold' | 'dots')?
```

Der kleinste und größte darzustellende Wert wird durch die Ausdrücke in den eckigen Klammern angegeben. Wird keine `RangeExpression` nicht angegeben, erfolgt die Skalierung automatisch, sodass der gesamte Wertebereich dargestellt wird. Durch die Schlüsselworte ‚`ln`' und ‚`lg`' kann für die entsprechende Achse eine logarithmische Darstellung aktiviert werden (natürlicher bzw. dekadischer Logarithmus).
Standardmäßig wird die Kurve linear dargestellt, d.h. die einzelnen Samples werden durch eine Gerade verbunden. Mit dem Schlüsselwort ‚`hold`' wird zu einer „Sample-and-Hold" Darstellung gewechselt, ‚`dots`' zeichnet einzelne Punkte.

6.6 INTEGRATION DER HOST-SPRACHE

Die Bekanntmachung von Klassen der Host-Sprache mit der *import* Anweisung wurde in Abschnitt 6.2.2 bereits erklärt. Die Klasse `java.io.File` der Host-Sprache Java für den Dateizugriff wird demnach folgendermaßen importiert:

```
import java.io.File as File;
```

Anhand das gewählten Alias kann die Klasse anschließend referenziert werden. Auf statische Klassenmember kann ohne konkrete Instanz zugegriffen werden. Eine neue Instanz wird durch den `new` Operator erzeugt, also ganz genauso wie aus anderen Programmiersprachen gewohnt:

```
File myFile = new File("/path/to/file");
```

Es wird die Verwendung generischer Klassen unterstützt. Die Typparameter werden dabei wie im folgenden Beispiel innerhalb spit-

zer Klammern angegeben (vgl. Regel `DataTypeID`), d.h. es besteht auch hier kein syntaktischer Unterschied zu den gängigen Programmiersprachen:

```
List<int32> listOfIntegers = new List<int32>();
```

Diesbezüglich ist jedoch anzumerken, dass generische Funktionen von der ETSpec Sprache syntaktisch nicht unterstützt werden. Lassen sich die Typparameter der aufzurufenden Funktion also nicht aus dem Kontext ableiten, was eher selten der Fall ist, muss eine entsprechende Hilfsfunktion in der Host-Sprache erstellt werden.

6.7 VERGLEICH MIT DEN ANFORDERUNGEN

Die ETSpec Sprache erfüllt alle in Abschnitt 5.1 aufgestellten Anforderungen:

AD1.1 *Typsystem*: Die ETSpec Sprache erweitert das aus Java bekannte Typsystem sinnvoll entsprechend den speziellen Anforderungen an die Testlösung.

AD1.2 *Ausdrücke*: Die ETSpec Sprache unterstützt beliebig komplexe Ausdrücke.

AD1.3 *Kontrollfluss*: Die ETSpec Sprache unterstützt Schleifen, bedingte Ausführung und den Aufruf von Methoden.

AD1.4 *Modularisierung*: Die ETSpec Sprache ermöglicht Modularisierung durch Pakete und den Import von Klassen der Host-Sprache.

AD1.5 *Standardbibliothek*: Die Standardbibliotheken der Host-Sprache können transparent in der ETSpec Sprache genutzt werden.

AD2.1 *Zentrale Testfeatures*: Die ETSpec Sprache besitzt Sprachkonstrukte für Testfälle, Test Suiten und zur Testauswertung (Bewertung).

AD2.2 *Externe Deklaration*: Externe Deklarationen sind durch das TRM auf Modellebene mit dem Quelltext der ETSpec Sprache verknüpft.

AD2.3 *Physikalische Datentypen*: Die ETSpec Sprache bietet den Datentyp `physical` für physikalische Größen.

AD2.4 *Komplexe Testobjekte*: Das Sprachkonstrukt *Target*, die *scope* Anweisung und das Schlüsselwort ‚`default`‘ unterstützen die Arbeit mit komplexen Testobjekten.

AD2.5 *Kontrollierte Nebenläufigkeit*: Die *concurrent* Anweisung, benutzerdefinierte Ereignisse und die *wait-until* Anweisung erleichtern die Beschreibung paralleler Abläufe.

AD2.6 *Vorgabe von Echtzeitverhalten*: Die Vorgabe echtzeitkritischer Abläufe wird durch die *realtime* Anweisung unterstützt.

AD2.7 *Analyse von Echtzeitdaten*: Die präzise Analyse analoger Signalverläufe und die Prüfung von Echtzeitanforderungen wird durch die weiteren Blöcke der *realtime* Anweisung unterstützt.

7 IMPLEMENTIERUNG

Das ETSpec Framework wurde als Feature der Eclipse IDE umgesetzt. Die wesentlichen Bestandteile der Implementierung sind:

- *EMF Modelle*: Umfasst die Meta-Modelle des TRMs und der TQCs. Siehe Abschnitt 7.1.
- *ETSpec Sprache*: Umfasst den Lexer, Parser und Codegenerator für die ETSpec Sprache. Siehe Abschnitt 7.2.
- *ETSpec Laufzeitumgebung*: Umfasst die die API Abstraktionsschicht, Kontrolle der Testausführung, Adapter für herstellerspezifische APIs sowie einen Report-Generator. Siehe Abschnitt 7.3.
- *Benutzerschnittstelle*: Umfasst die Editoren für ETSpec Quelltext, TRM und TQCs, die Oberfläche zur Visualisierung von Testdaten („Echtzeit UI“) und weitere Oberflächen zur Projektverwaltung und Logging. Siehe Abschnitt 7.4.

Die Implementierung eines Eclipse Features setzt sich aus einer Reihe sogenannter Eclipse Plugins zusammen. Die im Rahmen dieser Arbeit erstellten Plugins sind in Abbildung 27 dargestellt.
Die Java Implementierung umfasst über 170.000 Zeilen Code (Bibliotheken, Kommentare und Leerzeilen nicht mitgezählt). Auch wenn davon etwa 82 % (140.000 Zeilen) generiert sind, kann nicht auf jedes Detail eingegangen werden. In der folgenden Beschreibung wird daher an den entsprechenden Stellen auf die jeweiligen Plugins verwiesen.

7.1 EMF MODELLE

Die Metamodelle des TRMs und der TQCs wurden mittels Ecore entworfen. Das EMF generiert daraus bestimmte Teile der Implementierung (vgl. Abschnitt 3.7.1). Alle Ecore Modelle sind Teil des Eclipse Plugins `etspec.model`.
Das Metamodell des TRMs wurde in Abschnitt 5.3.1 bereits beschrieben. Im Folgenden wird wird daher gleich auf die TQCs eingegangen. Es wird zwischen der Beschreibung von Softwaregrößen

(Abschnitt 7.1.1), Netzwerkdaten (Abschnitt 7.1.2) und Umgebungsdaten (Abschnitt 7.1.3) unterschieden.

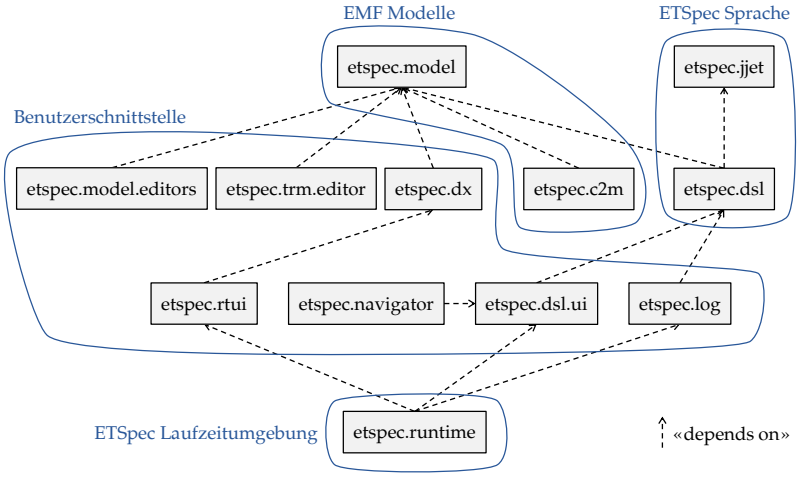


Abbildung 27: Implementierte Eclipse Plugins

7.1.1 *Beschreibung von Softwaregrößen*

Die für den Test relevanten Softwaregrößen sind Variablen und Funktionen (Symbole). Alle relevanten Informationen werden durch das *CodeModel* erfasst. Abbildung 28 zeigt das für die Programmiersprache C spezifische Metamodell.
Die wichtigste gespeicherte Information ist die spätere Adresse des Symbols im Speicher des Targets (Klasse *BinaryLocation*). Die Adresse ist erst nach der Kompilierung bekannt, und muss durch den Compiler bereitgestellt werden. Eine weitere wichtige Information ist der Ort der Deklaration im Quelltext (Klasse *SourceCodeLocation*). Diese Information ermöglicht später die einfache Navigation aus der Implementierung eines Testfalls in den Quelltext der Software („Code Navigation“).
Ebenfalls im Modell enthalten sind die Beziehungen zwischen den einzelnen Deklarationen (z.B. „Variable x gehört zu Funktion *foo*“ und „Variable x besitzt den Datentyp Y“). Informationen über die Programmlogik sind hingegen nicht enthalten. Das Modell kann somit weitergereicht werden, ohne Details über die Implementierung preiszugeben. Das dient zum einen dem Schutz geistigen Eigentums, und verhindert außerdem das „Herumtesten“ um Fehler, da der Testentwickler nicht in die Versuchung gerät sich am Quelltext zu orientieren.

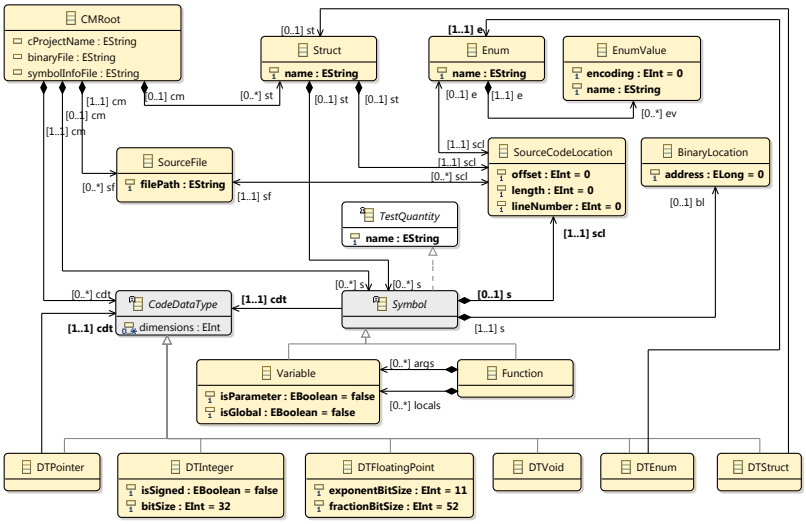


Abbildung 28: Metamodell des CodeModels

Automatische Modellerzeugung

Das CodeModel wird automatisch aus dem Quelltext der Software erzeugt. Informationen aus der vom Compiler bereitgestellten Debug-Information können später ergänzt werden. Das ist wichtig, weil das CodeModel sofort zur Verfügung stehen muss, und nicht erst wenn die Software fehlerfrei kompiliert werden kann. Denn ohne die Informationen aus dem Modell sind die Möglichkeiten zur Unterstützung bei der Testentwicklung eingeschränkt, und ein wesentlicher Vorteil des ETSpec Frameworks ginge verloren.
Die notwendige Transformation des Quelltextes geschieht auf Basis des ASTs vom C-Code, der von den C/C++ Development Tools (CDT) für Eclipse erzeugt wird [Scho9]. Abbildung 29 zeigt einen solchen AST für eine Beispielfunktion. Die zugehörige Instanz des CodeModels ist in Abbildung 30 skizziert.
Die Transformation ist durch das Plugin `etspec.model.c2m` implementiert. Das Plugin überwacht außerdem alle Änderungen an Dateien in CDT Projekten im Eclipse Workspace. Bei einer Änderung des Softwarequelltextes wird das zugehörige Modell im Hintergrund automatisch aktualisiert.

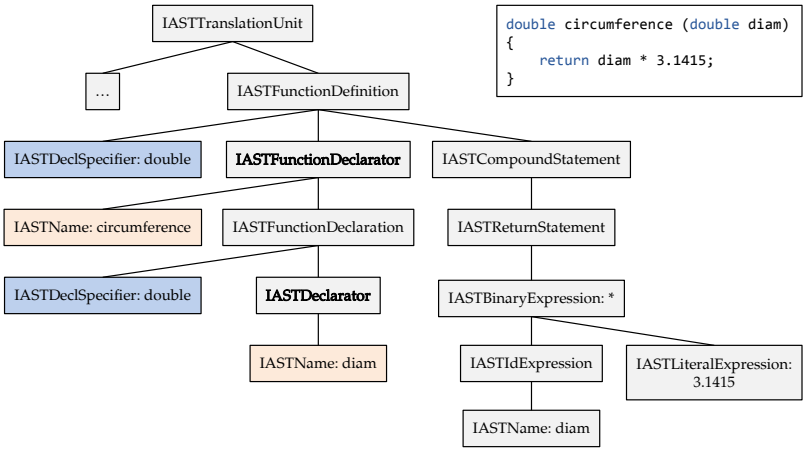


Abbildung 29: Von CDT erzeugter AST [Sch09]

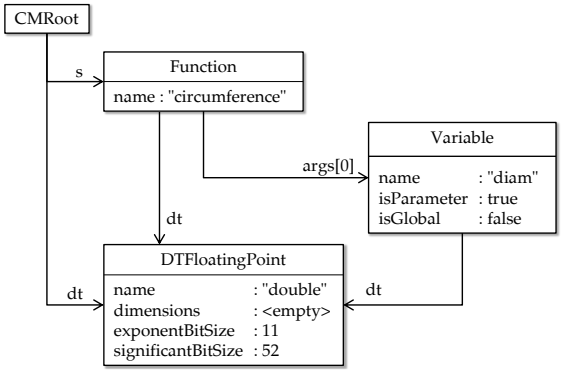


Abbildung 30: Instanz des CodeModels zum Beispiel in Abbildung 29

7.1.2 *Beschreibung von Netzwerkdaten*

Die Beschreibung von Netzwerkdaten erfolgt durch das *NetworkDataModel*. Da Netzwerkdaten immer protokollspezifisch zu interpretieren sind, müssen entsprechende Informationen im Modell enthalten sein. Es ist unvermeidbar, dass sich dazu protokollspezifische Elemente im Metamodell wiederfinden.
Die genauere Betrachtung einiger Kommunikationsprotokolle erfolgte im Rahmen einer studentischen Abschlussarbeit [Hen14]. Eine Berücksichtigung aller relevanten Standards ist für den gewünschten Proof of Concept nicht erforderlich, und es wurde daher ein einfaches Übertragungsprotokoll ausgewählt. Die Ent-

scheidung fiel auf CAN, da es im Automobilbereich weit verbreitet ist.
In Abbildung 31 ist das resultierende Metamodell dargestellt. Es ist ähnlich aufgebaut wie das proprietäre Dateiformat DBC [Onl/DBC], d.h. es wird der Signalbegriff für die in einer Nachricht enthaltenen Nutzdaten verwendet. Durch die Attribute *bitOffset* und *bitLenght* der Klasse *NetworkSignal* wird dem Datenfeld einer Nachricht ein Signal zugeordnet. Die Interpretation dieser Daten wird durch eine Spezialisierung der Klasse *NetworkDataType* spezifiziert. Für physikalische Signale (Klasse *DTPhysical*) sind beispielsweise Faktor und Offset zur Umrechnung der Rohdaten sowie die physikalischen Einheit notwendig.

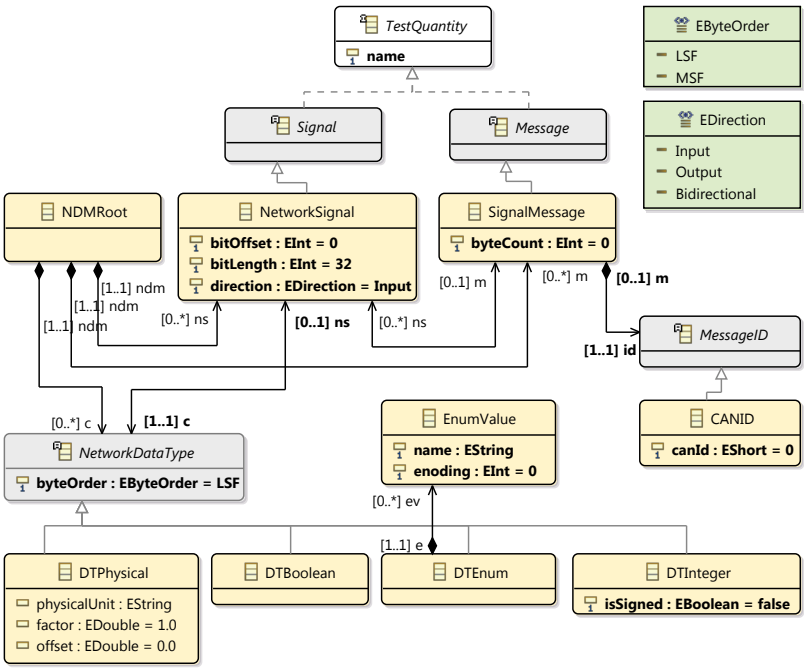


Abbildung 31: Metamodell des NetworkDataModels

Im Metamodell ist nur eine protokollspezifische Klasse zu finden (*CANID*). Soll die Unterstützung für ein weiteres Protokoll hinzugefügt werden, können die übrigen Beschreibungsmittel wiederverwendet werden, falls die kommunizierten Daten ebenfalls als Signale aufgefasst werden können. Häufig reicht es dann schon aus, eine Spezialisierung der Klasse *MessageID* für den neuen Identifier zu ergänzen.

Können die kommunizierten Daten nicht als Signale aufgefasst werden, ist eine protokollunabhängige Erweiterung des Metamodells zur Erfassung komplexer Datenstrukturen sinnvoll. Mit der „Abstract Syntax Notation One“ (ASN.1) existiert bereits ein entsprechender Standard der International Telecommunication Union [Std/ITU11b], der speziell für die Beschreibung von Netzwerkdaten entworfen wurde, und hierzu auch in TTCN-3 Anwendung findet. Im Rahmen dieser Arbeit wurde auf eine entsprechende Implementierung verzichtet.

7.1.3 *Beschreibung von Umgebungsdaten*

Bezüglich Umgebungsdaten wurden in dieser Arbeit ausschließlich Signale betrachtet. Sie werden durch das *EnvironmentDataModel* beschrieben. Das entsprechende Metamodell in Abbildung 32 ist dem NetworkDataModel ähnlich. Protokollspezifische Details und Informationen zur Kodierung von Rohdaten können jedoch entfallen, da grundsätzlich mit dem abstrakten physikalischen Wert gearbeitet wird[1].

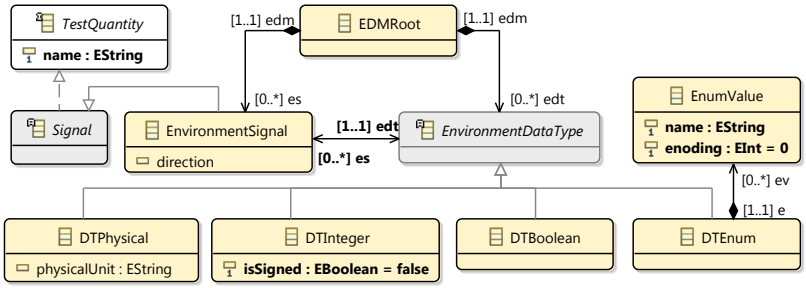


Abbildung 32: Metamodell des EnvironmentDataModels

7.2 ETSPEC SPRACHE

Zur Implementierung der ETSpec Sprache ist die Übersetzung des ETSpec Codes in ein ausführbares Programm am wichtigsten. Dazu wurden drei Werkzeuge implementiert, die in Abbildung 33 dargestellt sind. Als Erstes wird durch einen Parser ein AST erstellt und anschließend das Linking durchgeführt (siehe Abschnitt 7.2.2). Auf dieser Basis erfolgt anschließend die statische Analyse des ETSpec

1 Für Enumerationen ist aus Effizienzgründen zur internen Repräsentation weiterhin eine Kodierungsinformation erforderlich

Codes (siehe Abschnitt 7.2.3) und zum Schluss die Generierung von Java Code (siehe Abschnitt 7.2.4). Der generierte Code wird durch den Standard Java Compiler in Bytecode übersetzt, und anschließend von der Java Virtual Machine (JVM) ausgeführt.

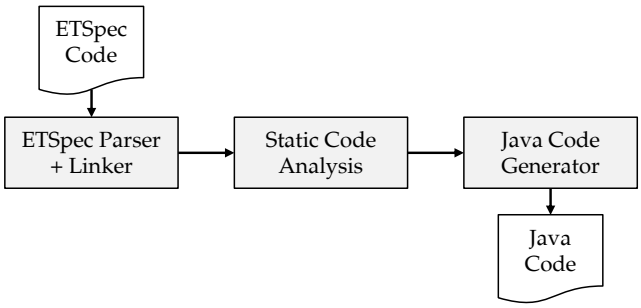


Abbildung 33: Implementierte Werkzeuge zur Übersetzung der ETSpec Sprache

7.2.1 *Vereinfachte Implementierung*

Bei der Implementierung wurde auf die Unterstützung einiger Sprachfeatures verzichtet, die zur Demonstration des Gesamtkonzepts unerheblich sind.

- *Typsystem*: Nicht unterstützt werden die komplexen Datentypen `enum`, `struct` und mehrdimensionale Arrays (eindimensionale Arrays werden unterstützt), sowie die primitiven Datentypen `char`, `pointer`, `cint` und `cuint`.
- *Host-Sprache*: Die Syntax zur Verwendung generischer Klassen wird nicht unterstützt. Der Verweis auf Klassen der Host-Sprache erfolgt über einen Hilfselement (Schlüsselwort ‚`Java`'), was die Implementierung des Linkers erleichtert.
- *Operatoren*: Die Operatoren `<<=`, `>>=`, `>>>=` sowie der bedingte Operator werden nicht unterstützt.
- *Anweisungen*: Die *foreach* Anweisung wird nicht unterstützt. Die Anweisungen *wait-until*, *trigger*, *run-until*, *stop-at* und *concurrent* wurden jeweils leicht vereinfacht, ohne ihre Grundfunktion zu beeinträchtigen (z.B. wird der *finally* Block der *concurrent* Anweisung nicht unterstützt).

Darüber hinaus wird das Schlüsselwort ‚`observe`' (Zur Grenzwertüberwachung während einem Test) und die Verwendung des

Schlüsselworts ‚as' bei *Imports* und *Targets* nicht unterstützt. Die Übergabe von Testgrößen als Referenz (Schlüsselwort ‚ref') wurde ebenfalls vereinfacht. Davon abgesehen wurde streng der Definition in Kapitel 6 gefolgt.

7.2.2 *Anpassung des Linkers*

Die Werkzeuge zur Verarbeitung der ETSpec Sprache wurden mit Hilfe von Xtext implementiert. Ausgangspunkt ist die Grammatik der Sprache in Xtext Syntax, welche in Anhang B vollständig enthalten ist. Daraus wird ein Parser für die ETSpec Sprache generiert (vgl. Abschnitt 3.7.2). Das zugehörige Eclipse Plugin ist `etspec.dsl`. Der für den Parser generierte Code erfordert keine direkten Anpassungen, jedoch muss das Verhalten des Linkers implementiert werden. Das betrifft die Auflösung von Namen bei Referenzen auf Testgrößen, lokale Variablen und Klassen der Host-Sprache. Eine Anpassung ist erforderlich, weil die Semantik der relevanten Sprachkonstrukte nicht automatisch aus der Grammatik abgeleitet werden kann (konkret sind das die *import*, *scope* und *realtime* Anweisung, alle Deklarationen sowie das Schlüsselwort ‚`default`').
Das Xtext Framework ermöglicht die notwendigen Anpassungen über das Entwurfsmuster Dependency Injection [Fow09], genauer gesagt durch Google Guice [Onl/GG]. Die entsprechend gebundene Klasse berechnet in Abhängigkeit vom Kontext die sichtbaren lokalen Variablen, Testgrößen und Java-Klassen.
Die Suche nach den sichtbaren Testgrößen wird durch den Aufbau der ETSpec Sprache begünstigt. Es muss lediglich der bis zu diesem Zeitpunkt bereits aufgespannte Teil des AST so lange nach oben durchwandert werden, bis entweder eine *scope* oder *concurrent* Anweisung, ein Test oder eine Funktionsdeklaration erreicht ist. Über die dort referenzierte *TargetDefinition* findet sich genau ein *ScopeProvider* mit Verweis auf den TQC, der die fraglichen Testgrößen enthält.
Für die Suche nach den sichtbaren lokalen Variablen wird der AST ebenfalls aufwärts durchlaufen, bis ein Test oder eine Funktionsdeklaration erreicht ist. Unterblöcke (z.B. von vorhergehenden Schleifen etc.) werden dabei nicht betreten. Jede erreichte `VariableDeclaration` ist sichtbar. Ein Sonderfall existiert, wenn die Startposition innerhalb eines `AnalyzeBlocks` oder `PlotBlocks` liegt. Es werden dann vorhergehende `AnalyzeBlocks` derselben *realtime* Anweisung betreten und dort alle Unterknoten des AST durch-

laufen. Jede dabei erreichte und mit dem Schlüsselwort ‚`capture`' versehene `VariableDeclaration` ist ebenfalls sichtbar.
Die sichtbaren Java-Klassen ergeben sich aus den *import* Anweisungen des aktuellen Dokuments. Zur Auflösung des dort angegebenen FQN wird auf Xbase zurückgegriffen, einer von Xtext bereitgestellten Infrastruktur zur Integration von Java in die eigene Sprache [EEK+12]. Hier ist anzumerken, dass Xbase ausschließlich für die Namensauflösung verwendet wird. Insbesondere ist eine Nutzung der Kernfunktionalität von Xbase, nämlich die Bereitstellung von Ausdrücken und entsprechender Codegenerierung, im Rahmen dieser Arbeit nicht möglich gewesen, da die gleichzeitige Verwendung eines benutzerdefinierten Typsystems von Xbase nicht unterstützt wird.

7.2.3 *Statische Codeanalyse*

Es wurden einige Algorithmen zur statischen Analyse des Testcodes implementiert. Sie sind Teil des Eclipse Plugins `etspec.dsl`. Besonders erwähnenswert ist die automatische Prüfung der Typkompatibilität, da sie als eine der Gründe für den Entwurf einer neuen Sprache genannt wurde.
Für diese Prüfung muss ein bestimmter Ausschnitt des ASTs untersucht werden, nämlich zur Regel `Expression`. Der interessanteste Teil betrifft hier die Typinferenz bei der Verwendung physikalischer Einheiten. Diese geht so weit, dass bei der Multiplikation und Division physikalischer Größen der resultierende Typ berechnet wird. Das ist nur wegen der speziellen Darstellung physikalischer Einheiten in der ETSpec Sprache möglich. Die `UNITID` beider Operanden wird zuerst in die jeweilige Basiseinheit überführt, und anschließend anhand des dedizierten Trennzeichens in die jeweiligen Bestandteile zerlegt. Danach werden die Bestandteile beider Einheiten unter Berücksichtigung der Exponenten und des Operators zu einer neuen Einheit kombiniert. Die resultierende Einheit wird im Falle einer weiteren Multiplikation oder Division erneut anhand ihrer Bestandteile mit der anderen Einheit verrechnet, oder im Fall einer Zuweisung, Addition, Subtraktion oder eines Vergleichs mit der anderen Einheit verglichen.
Andere Prüfungen betreffen weniger komplexe Randbedingungen. Hierzu zählt beispielsweise die Zusicherung, dass innerhalb eines Sichtbarkeitsbereichs keine Namen doppelt verwendet werden, z.B. bei lokalen Variablen einer Funktion.

7.2.4 *Code Generator*

Der Code Generator ist Teil des Eclipse Plugins `etspec.dsl`. Technologisch basiert er auf einer unabhängigen Template Engine, die durch `etspec.jjet` bereitgestellt wird. Template Engines verarbeiten Vorlagen für Textdateien (Templates) und werden in vielen Bereichen zur Codegenerierung eingesetzt. In Zusammenhang mit Xtext wird einem die Verwendung von Xtend als Template Engine nahegelegt [Bet13]. Im Rahmen dieser Arbeit wurde jedoch die ältere Technologie Java Emitter Templates (JET) [Onl/JET] als Grundlage ausgewählt, da die entsprechende Template-Syntax etwas leichter zu erlernen ist. Die Funktionsweise ist in [SS05] beschrieben, genauere Kenntnis ist für das Verständnis dieses Abschnitts jedoch nicht erforderlich.
Der Code Generator erzeugt aus dem AST der ETSpec Quelltextdatei Java Code. Der generierte Code implementiert bestimmte, durch das Framework vorgegebene Interface-Klassen, sodass er von der ETSpec Laufzeitumgebung aufgerufen werden kann. Gleichzeitig nutzt er die von der Laufzeitumgebung bereitgestellten Schnittstellen, z.B. für den Zugriff auf das Testobjekt, oder zum Absetzen von Meldungen. Die genutzten Schnittstellen werden in Abschnitt 7.3 genauer beschrieben. Die vom generierten Code implementierten Interface-Klassen sind in Abbildung 34 dargestellt.
ITest, *IPackage* und *ITestSuite* entsprechen den Basiscontainern `TestDeclaration`, `PackageDeclaration` und `TestSuiteDeclaration` der ETSpec Sprache (vgl. Abschnitt 6.2.3). Für jeden Basiscontainer wird eine Java Klasse generiert, die das zugehörige Interface implementiert.
Die zur Testausführung wesentliche Funktion ist `run()` der Klasse *ITest*. Sie enthält den Code für den Testablauf, d.h. eine Java-Version der Anweisungen und Ausdrücke der ETSpec Sprache, und wird von der ETSpec Laufzeitumgebung aufgerufen. Die Funktionen `init()` und `cleanup()` enthalten den Testfall-spezifischen Code zur Initialisierung der Laufzeitumgebung bzw. anschließender Bereinigung. Sie werden ebenfalls von der Laufzeitumgebung aufgerufen (vgl. Abschnitt 7.3.3).
Die Funktionen `getSourceFileUri()`, `getSourceOffset()` und `getSourceLength()` dienen dem späteren Auffinden des ETSpec Quelltextes, aus dem die Klasse generiert wurde. Die übrigen Funktionen der Klasse *ITest*, die verwendeten Interfaces *ITestStatistics*, *IReportMessage* und *IFailedAssertion* sowie Aufzählungen sind selbsterklärend.

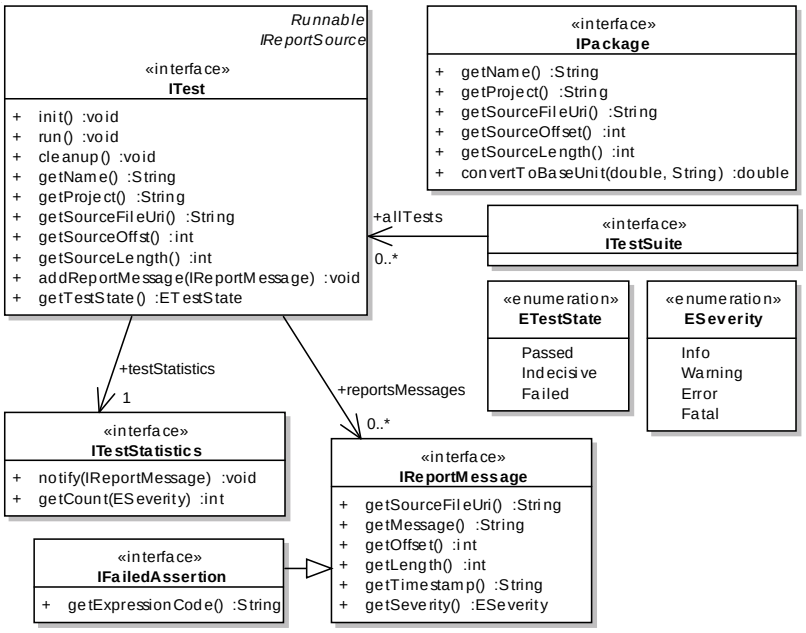


Abbildung 34: Die vom generierten Code implementierten Interface-Klassen *ITest*, *IPackage* und *ITestSuite*

Die zu einem ETSpec Paket generierte und von *IPackage* abgeleitete Klasse enthält für jede globale Variable bzw. Funktion des Pakets ein öffentliches Klassenmember. Außerdem wird die Logik zur Umrechnung aller im Paket deklarierten Einheiten in die Funktion `convertToBaseUnit()` generiert.
Es ist jetzt relativ einfach, Querverweise zwischen Tests und Paketen durch den generierten Java Code abzubilden. Das lässt sich am besten anhand eines Beispiels zeigen. Listing 3 zeigt dazu exemplarischen ETSpec Quelltext. Der daraus generierte Java Code ist in Listing 4 und Listing 5 zu finden.
Ein Querverweis findet sich in Zeile 20 des ETSpec Quelltexts, in welcher die weiter oben definierte Funktion `pow()` des Pakets `Helpers` aufgerufen wird. Dies wird durch die Zeilen 15, 20 und 36 im Java Code der Testklasse abgebildet.
Bereits diesem Beispiel lassen sich weitere Details zur Codegenerierung entnehmen. Für weitere Beispiele zur Codegenerierungen sei auf die unter der Eclipse Public License (EPL 1.0) bereitgestellte Implementierung des ETSpec Frameworks verwiesen, welche auf der Webseite www.es-tdk.org zur Verfügung steht.

Listing 3: Beispiel zur Codegenerierung – ETSpec Code

```
// Beispiel für ein Package
package Helpers
{
    // Ein einfacher Algorithmus zur Potenzberechnung
    func double pow(double base, int32 exponent)
    {
        if (exponent == 0)
            return 1;

        if (exponent == 1)
            return base;

        double result = base;
        for (int32 step = 1; step < exponent; step+=1)
        {
            result = result * base;
        }
        return result;
    }
}

// Beispiel für den Zugriff auf ein Paket aus einem Test
test Example
{
    double powerOf2 = Helpers.pow(2, 8);
    assert (powerOf2 == 256) error;
}
```

Listing 4: Beispiel zur Codegenerierung – Java Code der Testklasse

```
1  package generated.tests;
2  // ... (imports)
3  public class ExampleTest extends AbstractTest
4  {
5      private final String   NAME            = "Example";
6      private final String   SOURCE_PROJECT  = "project";
7      private final String   SOURCE_FILE_URI =
8                             "/project/pts/Example.ets";
9      private final int      SOURCE_OFFEST   = 529;
10     private final int      SOURCE_LENGTH   = 93;
11
12     private final IRuntime _pts_runtime;
13     private final IUIDResolver _pts_uid_resolver;
14     private final ITest _pts_test = this;
15     private final HelpersPackage _pkg_Helpers;
16
17     public ExampleTest(IRuntime runtime) {
18         _pts_runtime = runtime;
19         _pts_uid_resolver = runtime.getUIDResolver();
20         _pkg_Helpers = new HelpersPackage(runtime, this);
21     }
22
23     @Override
24     public void init() {
25         _pts_runtime.init();
26     }
27
28     @Override
29     public void cleanup() {
30         _pts_runtime.cleanup();
31     }
32
33     @Override
34     public void run() {
35         try {
36             double powerOf2=_pkg_Helpers.pow(2,8);
37             if (!(powerOf2==256)) {
38                 _pts_runtime.assertionFailed(_pts_test
39                 , "/project/pts/Example.ets"
40                 , 589, 31, "powerOf2 == 256", 2);
41             }
42         }
43         catch (ExecutionInterruptedException e) {
44             _pts_runtime.report(this, SOURCE_FILE_URI,
45                 SOURCE_OFFEST, SOURCE_LENGTH, 1,
46                 "Cancelation of test execution requested");
47             return;
48         }
49     }
50     // ... (getName(), getProject(), getSourceUri() etc.)
51 }
```

Listing 5: Beispiel zur Codegenerierung – Java Code der Paketklasse

```
1  package generated.packages;
2  // ... (imports)
3  public class HelpersPackage implements IPackage
4  {
5      private final String NAME           = "Helpers";
6      private final String SOURCE_PROJECT  = "project";
7      private final String SOURCE_FILE_URI =
8                           "/project/pts/Example.ets";
9      private final int    SOURCE_OFFEST   = 0;
10     private final int    SOURCE_LENGTH   = 422;
11
12     private final IRuntime _pts_runtime;
13     private final IUIDResolver _pts_uid_resolver;
14     private final ITest _pts_test;
15
16     public HelpersPackage(IRuntime runtime, ITest test){
17         _pts_runtime = runtime;
18         _pts_uid_resolver = runtime.getUIDResolver();
19         _pts_test = test;
20     }
21
22     public double pow(double base, int exponent){
23         if (exponent==0){
24             return 1;
25         }
26         if (exponent==1){
27             return base;
28         }
29         double result=base;
30         int step=1;
31         while(step<exponent){
32             result=result*base;
33             step+=1;
34         }
35         return result;
36     }
37
38     @Override
39     public double convertToSIUnit(double val, String u){
40         throw new RuntimeException("Invalid unit: " + u);
41     }
42     // ... (getName(), getProject(), getSourceUri() etc.)
43 }
```

7.3 ETSPEC LAUFZEITUMGEBUNG

Das Plugin `etspec.runtime` implementiert die ETSpec Laufzeitumgebung. In Abschnitt 7.3.1 wird die durch den generierten Code genutzte Schnittstelle erklärt. Anschließend wird in Abschnitt 7.3.2 auf die Implementierung der darunterliegenden API Abstraktionsschicht eingegangen, was auch die Adapter zur Anbindung der herstellerspezifischen APIs umfasst.
Eine zentrale Aufgabe der Laufzeitumgebung ist die Kontrolle der Testausführung. Abschnitt 7.3.3 enthält die dazu wichtigsten Implementierungsdetails. Zuletzt wird in Abschnitt 7.3.4 auf die Implementierung des Report-Generators eingegangen.

7.3.1 *Schnittstelle für generierten Code*

Die durch den generierten Code genutzte Schnittstelle für den Zugriff auf die Laufzeitumgebung ist zweigeteilt. Der erste Teil ist durch die Interface-Klasse *IRuntime* in Abbildung 35 definiert und umfasst die während der normalen Testausführung genutzte Funktionalität. Der zweite Teil betrifft die Echtzeit Stimulation und Datenaufzeichnung und wird durch die Interface-Klasse *ITrace* definiert (siehe folgenden Abschnitt).
Die in Abbildung 35 dargestellten Klassen *IAssertionHandler* und *IReportHandler* sind selbsterklärend, ebenso der Zweck der Funktionen `init()` und `cleanup()`.

Bezeichnung von Modellelementen im generierten Code

Die Funktion `getUIDResolver()` steht in Zusammenhang mit der Auflösung des im generierten Code verwendeten eindeutigen Bezeichners (GUID) für Elemente der ETSpec Sprache, des TRMs und der TQCs. Es handelt sich dabei um eine Zeichenkette. Den Funktionen von *IRuntime* und *ITrace* wird zur Bezeichnung eines Elements die jeweilige GUID übergeben. Via `getUIDResolver()` besteht Zugriff auf die Instanz einer Klasse, welche eine Hashtabelle mit den tatsächlichen Objekten verwaltet.

Ereignisse und Nebenläufigkeit

Beide Varianten der Funktion `suspendUntil()` blockieren bis zum Eintreten des angegebenen Ereignisses. Entsprechende Aufrufe werden aus der *wait-until* Anweisung generiert. Für die *trigger* Anweisung werden Aufrufe der Funktion `raiseEvent()` generiert. Analog ist die *cancel* Anweisung mit Hilfe der Funktion `cancelConcurrentThread()` umgesetzt. Die Implementierung all dieser Funktionen basiert auf der Java Thread API.

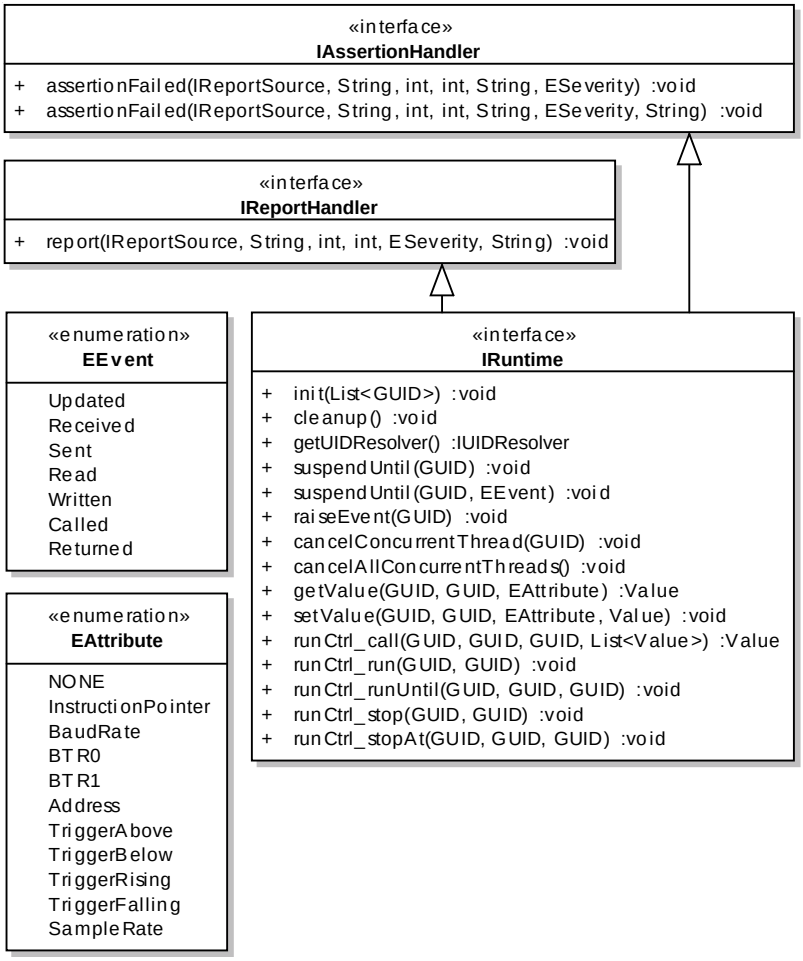


Abbildung 35: Die Schnittstelle *IRuntime* der Laufzeitumgebung

Hinter den verbleibenden Funktionen `getValue()`, `setValue()` und `runCtrl_...()` verbirgt sich die API Abstraktionsschicht (siehe Abschnitt 7.3.2).

Echtzeit Stimulation und Datenaufzeichnung

Der zweite Teil der durch den generierten Code genutzten Schnittstelle betrifft die Echtzeit Stimulation und Datenaufzeichnung (Tracing) und wird von der Interface-Klasse *ITrace* in Abbildung 36 definiert.
Für den Lebenszyklus einer *realtime* Anweisung wird die Instanz einer von *ITrace* abgeleiteten Klasse erzeugt. Als erstes werden alle

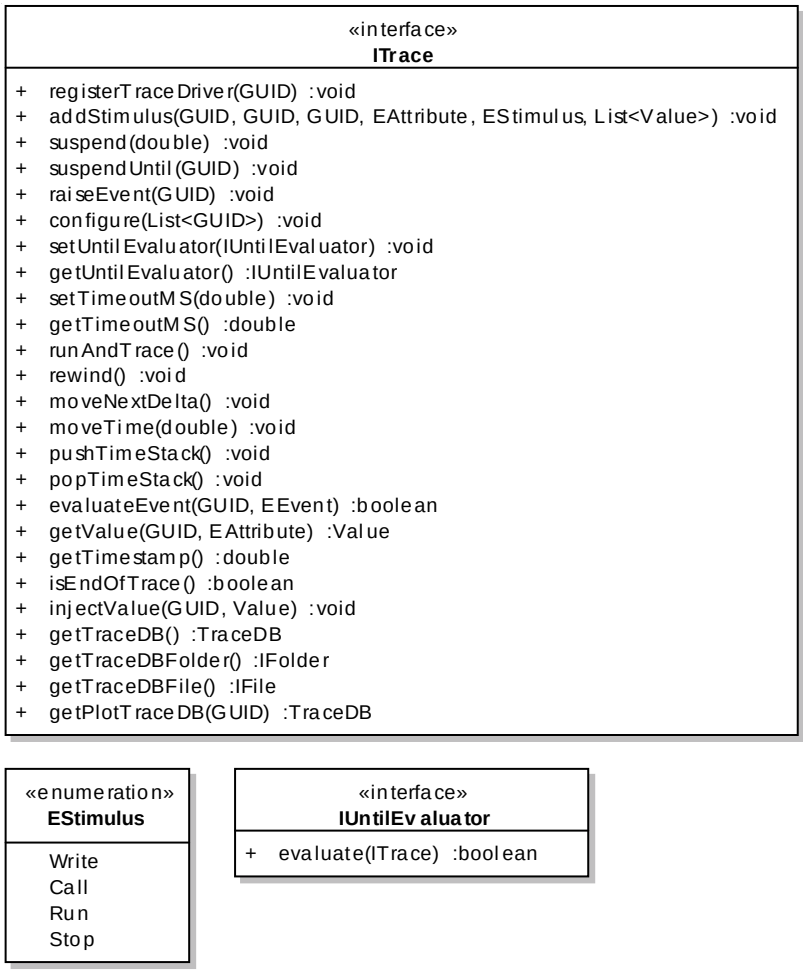


Abbildung 36: Die Schnittstelle *ITrace* der Laufzeitumgebung

an der Stimulation und Datenaufzeichnung beteiligten Treiber der API Abstraktionsschicht über die Funktion `registerTraceDiver()` angemeldet. Während der anschließenden Vorberechnung des Zeitstrahls (vgl. Abschnitt 6.5.6) wird die Funktion `addStimulus()` zur Registrierung der Stimuli verwendet. *Wait*, *wait-until* und *trigger* Anweisungen innerhalb eines `RealtimeBlocks` werden in Aufrufe der Funktionen `suspend()`, `suspendUntil()` und `raiseEvent()` von *ITrace* übersetzt, d.h. die Interface-Klasse *IRuntime* wird nicht mehr verwendet.

Vorkonfiguration

Nach Abschluss der Vorberechnung des Zeitstrahls werden über die Funktion `configure()` diejenigen Testgrößen registriert, de-

ren Signalverlauf und Ereignisse aufgezeichnet werden sollen. Außerdem besteht die Möglichkeit mittels `setUntilEvaluator()` die Instanz einer Klasse für die Prüfung der Abbruchbedingung zur übergeben. Deren einzige Funktion `evaluate()` wird später von der Laufzeitumgebung zyklisch aufgerufen, und muss `true` zurückgeben, wenn die Abbruchbedingung erreicht ist. Die zusätzlich konfigurierbare zeitliche Abbruchbedingung wird durch `setTimeoutMS()` gesetzt. Damit ist die Vorkonfiguration abgeschlossen und die Stimulation und Datenaufzeichnung wird einem Aufruf von `runAndTrace()` gestartet.

Zugriff auf aufgezeichnete Daten

Die übrigen Funktionen von *ITrace* dienen dem Zugriff auf die aufgezeichneten Daten. Entsprechende Aufrufe werden für Lesezugriffe auf Testgrößen und ihre Attribute innerhalb ‚`analyze`' Blöcken sowie in mit dem Schlüsselwort ‚`analyze`' versehenen ETSpec Funktionen generiert.

Die Funktion `rewind()` setzt den aktuellen Zeitpunkt auf den Beginn der Aufzeichnung. Sie wird zu Beginn jedes `AnalyzeBlocks` und `PlotBlocks` sowie vor jedem Aufruf einer Unterfunktion generiert. Die Funktion `moveNextDelta()` setzt den Zeitstempel auf den Zeitpunkt des nächsten Ereignisses in der Aufzeichnung. Sie wird im Rahmen einer *wait-until* Anweisung generiert. Die Funktion `moveTime()` inkrementiert den aktuellen Zeitstempel und wird für die *wait* Anweisung generiert. Die Funktionen `pushTimeStack()` und `popTimeStack()` dienen der Sicherung des aktuellen Zeitstempels beim Aufruf von Unterfunktionen.

Die Funktionen `getValue()` und `getAttributeValue()` liefern den aktuellen (Attributs-)Wert in der Aufzeichnung. Mit Hilfe von `evaluateEvent()` wird im Rahmen der *wait-until* das Auftreten des angegebenen Ereignisses geprüft. Die Funktion `getTimestamp()` liefert den aktuellen Zeitstempel, `isEndOfTrace()` ob das Ende der Aufzeichnung erreicht wurde.

Die Funktion `injectValue()` wird zur Aufzeichnung lokaler Variablen benötigt. Für jeden Schreibzugriff auf eine mit dem Schlüsselwort ‚`capture`' deklarierte Variable wird ein entsprechender Aufruf generiert. Damit wird die Aufzeichnung um die neuen Daten ergänzt.

Die Datenbank der Aufzeichnung wird durch eine Instanz der Klasse *TraceDB* repräsentiert. Teile dieser Implementierung zur Speicherung der aufgezeichneten Daten entstanden im Rahmen einer Abschlussarbeit [Deu14]. Die Funktionen `getTraceDB()`, `getTraceDBFolder()` und `getTraceDBFile()` stehen damit in Zusammenhang, werden jedoch nur intern verwendet.

Für jeden ‚plot' Block der *realtime* Anweisung wird eine zusätzliche Instanz der Klasse *TraceDB* erstellt, welche neben den entsprechenden Daten auch die Metainformationen zur grafischen Darstellung (Formatierung etc.) enthält. Im generierten Code wird diese Datenbank zunächst durch einen Aufruf von `getPlotTraceDB()` erzeugt, anschließend werden die Metainformationen gesetzt und die erforderlichen Daten kopiert.

7.3.2 *API Abstraktionsschicht*

Die API Abstraktionsschicht stellt sicher, dass bei einem Zugriff auf Testgrößen die korrekte im TRM konfigurierte Zugriffshardware angesprochen wird. Dazu ist die Implementierung eines Adapters für die jeweilige herstellerspezifische API erforderlich, welche von der Laufzeitumgebung angesprochen wird. Eine solche Implementierung wird im Folgenden als Treiber bezeichnet.
Für Treiber sind die Interface-Klassen in Abbildung 37 vorgegeben. Soll mit einer Zugriffshardware nur „normaler" Zugriff unterstützt werden, also keine Echtzeit-Stimulation oder Datenaufzeichnung, dann ist für den Treiber die Implementierung der Interface-Klasse *IAccessDriver* ausreichend. Ein solcher Treiber kann jedoch nicht in Zusammenhang mit der *realtime* Anweisung verwendet werden.[2]
Die Funktionen `getValue()`, `setValue()` und `runCtrl_...()` der Interface-Klasse *IRuntime* werden auf das jeweilige Äquivalent von *IAccessDriver* abgebildet. Eine weitere Erklärung erübrigt sich damit.
Soll durch den Treiber Echtzeit Stimulation und Datenaufzeichnung unterstützt werden, muss die Interface-Klasse *ITraceDriver* implementiert werden. Auch die hier definierten Funktionen werden von der Laufzeitumgebung aus der entsprechenden Funktion der Interface-Klasse *ITrace* aufgerufen. Einzig die Funktion `fillTraceDB()` muss kurz erklärt werden, da sie in *ITrace* nicht definiert ist. Sie wird beim Erreichen der Abbruchbedingung der Datenaufzeichnung aufgerufen. Der Treiber muss dann alle möglicherweise noch in einem privaten Cache liegenden Daten in die gemeinsame Datenbank des ETSpec Frameworks kopieren. Gleichzeitig wird dem Treiber damit mitgeteilt, dass die Aufzeichnung beendet werden darf.

2 Es ist jedoch durchaus möglich, dass ein solcher Treiber an einem Testablauf beteiligt ist, der *realtime* Anweisungen enthält. Eine Ausnahme wird nur ausgelöst, wenn ein solcher Treiber tatsächlich zur Stimulation eingesetzt wird.

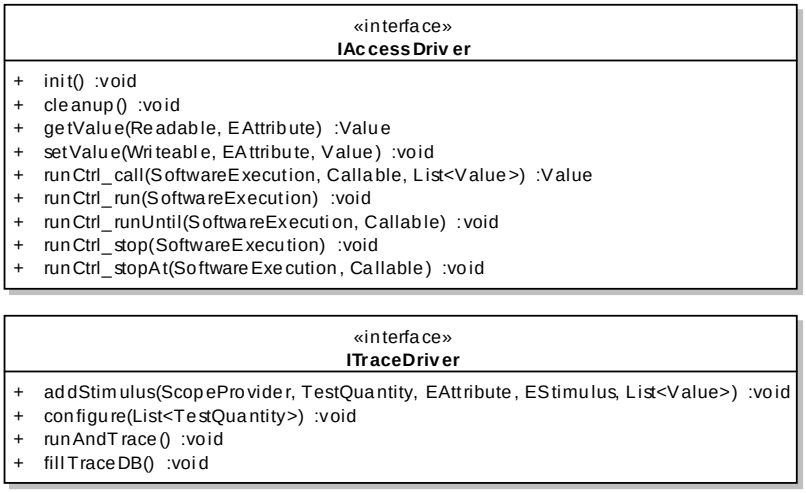


Abbildung 37: Von Treibern zu implementierende Schnittstellen

Durch die Laufzeitumgebung werden bei der Auswahl des Treibers die in den Funktionsaufruf involvierten GUIDs auf das jeweilige Modellelement aufgelöst. Deswegen finden sich in den Interface-Klassen *IAccessDriver* und *ITraceDriver* die konkreten Basistypen anstelle der GUIDs. Neben den aus dem TRM bekannten Klassen *ScopeProvider* und *SoftwareExecution* sind das die Klassen *Readable*, *Writeable* und *Callable*, wodurch die Basisklasse *TestQuantity* genauer differenziert wird.
Im Rahmen dieser Arbeit wurden Treiber für drei Geräte mit stark unterschiedlichen Fähigkeiten erstellt, welche in Abbildung 38 gezeigt sind. Es handelt sich dabei um einen Debugger mit Echtzeit-Tracing Unterstützung von der iSYSTEM AG (IC5000), ein CAN Bus Interface der Vector Informatik GmbH (VN7600), sowie ein Oszilloskop der Agilent Corporation (DSOX2012A). In allen Fällen wird die vom Hersteller angebotene native Treiberbibliothek mit Hilfe des Java Native Interface (JNI) [Li99] angesprochen.

Abbildung 38: Die zur Evaluation verwendete Hardware (v. l. n. r.): IC5000, VN7600, DSOX2012A

7.3.3 *Kontrolle der Testausführung*

Der Lebenszyklus eines Tests wird von der Laufzeitumgebung bestimmt. Sobald der Anwender über die Benutzeroberfläche die Testausführung initiiert, lädt die Laufzeitumgebung die von *ITest* abgeleiteten Klassen. Der anschließende Ablauf für einen Testfall ist exemplarisch in Abbildung 39 dargestellt.
Nach der Konstruktion einer Instanz der Testklasse wird zuerst für die Initialisierung (`init()`) und anschließend für den eigentlichen Ablauf (`run()`) die Kontrolle an die Testklasse abgeben. Von der Testklasse wird im Folgenden die die Schnittstelle *IRuntime* bedient. Kommt es dabei zur Echtzeit Stimulation, wird eine Instanz von *ITrace* erstellt und konfiguriert. Anschließend wird mit dem Aufruf von `runAndTrace()` die Kontrolle erneut abgegeben. Die involvierten *TraceDriver* kommunizieren ab jetzt direkt mit dem Testobjekt, d.h. ohne Umweg über *IRuntime*. Lediglich zur Prüfung der Abbruchbedingung ist die zentrale Interface-Klasse noch involviert.
Nach Ablauf der Stimulation kehrt die Kontrolle zur Auswertung der aufgezeichneten Daten an die Testklasse zurück. Dabei wird über *ITrace* auf die Datenbank zugegriffen.
Am Ende wird die Kontrolle an die Laufzeitumgebung zurückgegeben, welche abschließend die Ressourcenfreigabe initiiert (`cleanup()`). Wurde der Test im Rahmen einer Suite gestartet, verfährt die Laufzeitumgebung mit dem nächsten Test genauso. In jedem Fall wird nach der Abarbeitung eines Tests ein Bericht generiert (siehe folgenden Abschnitt).

7.3.4 *Report Generator*

Der Report Generator erzeugt zur jeder Test Suite sowie zu jedem Test eine HTML-Datei mit einer Statistik über die aufgetretenen Meldungen. Er ist Teil des Plugins `etspec.runtime` und basiert ge-

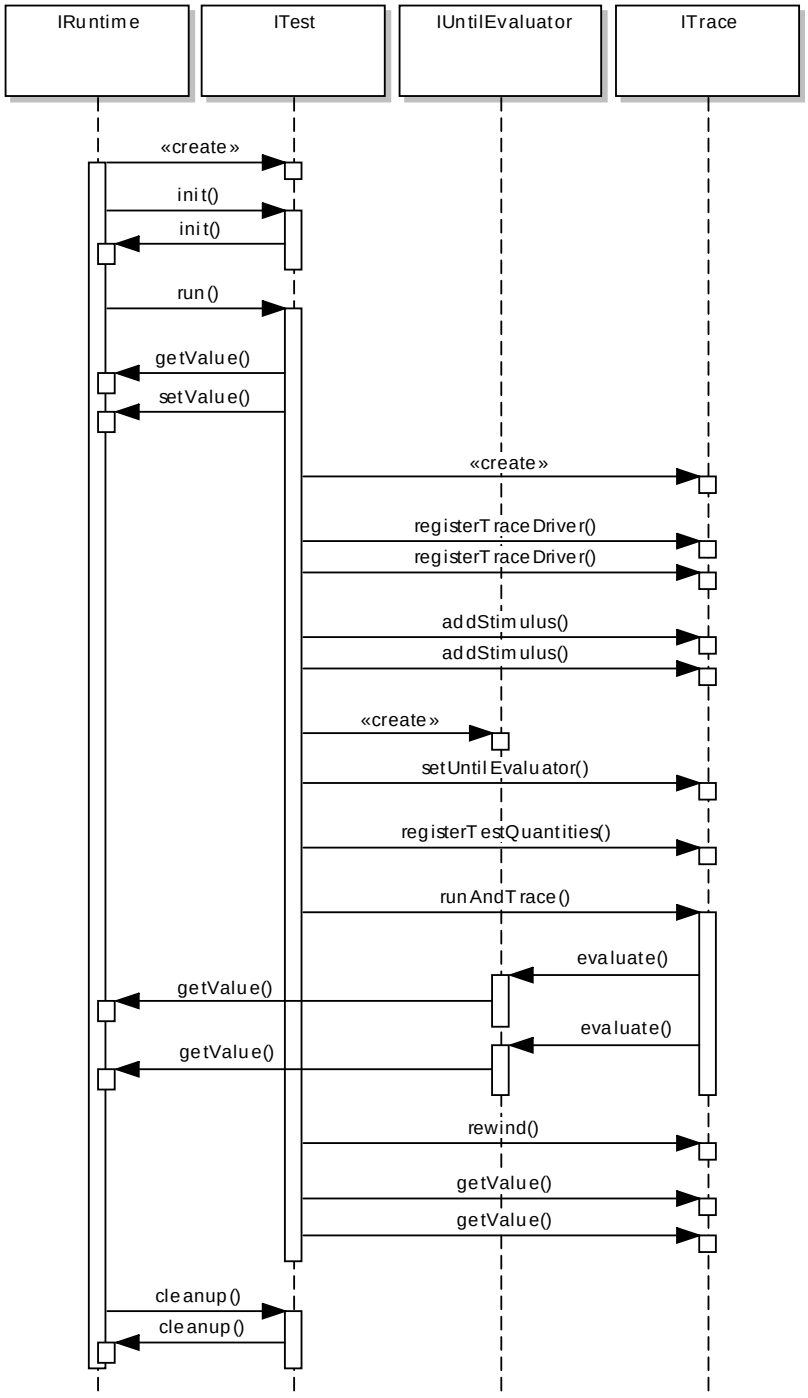


Abbildung 39: Beispiel für den Lebenszyklus eines Tests

nauso wie der Code Generator auf dem Plugin `etspec.jjet` (vgl. Abschnitt 7.2.4). Abbildung 40 zeigt mögliche Ausgaben für eine Test Suite und einen Testfall.

7.4 BENUTZERSCHNITTSTELLE

Das ETSpec Framework ist vollständig in die Eclipse IDE integriert, d.h. die gesamte Funktionalität ist über die Benutzeroberfläche von Eclipse verfügbar. Das ist für die Akzeptanz wichtig. Insbesondere bezüglich der ETSpec Sprache ist entsprechende Werkzeugunterstützung unerlässlich, da von keinem Entwickler erwartet werden darf, die Syntax einer umfänglichen Sprache auswendig zu beherrschen (vgl. Abschnitt 5.5).

7.4.1 *Text Editor*

Ein spezialisierter Texteditor für ETSpec Quelltext wird bereits zu großen Teilen von Xtext generiert (Plugin `etspec.dsl.ui`). Für einige Features sind Anpassungen nötig, worauf in dieser Arbeit jedoch nicht näher eingegangen werden soll. Abbildung 41 zeigt einen Screenshot des Texteditors.
Der Texteditor bietet die Hervorhebung von Syntax (z.B. werden Testgrößen gegenüber lokalen Variablen kursiv dargestellt), die Navigation im Quelltext (z.B. Springen zur Definition, auch bei Funktionen und Variablen der getesteten Software, wenn der Quelltext im gleichen Eclipse Workspace geladen ist), Code-Vorschläge (z.B. Syntax und Namen) sowie automatisiertes Refactoring (z.B. Umbenennen einer Variablen). Durch den Texteditor wird die statische Codeanalyse bereits während der Eingabe im Hintergrund ausgeführt. Dabei gefundene Probleme werden in Echtzeit an der entsprechenden Stelle im Quelltext markiert.

7.4.2 *Grafischer Editor*

Der grafische Editor für das TRM wurde im Rahmen einer Abschlussarbeit [Pri14] entwickelt (Plugin `etspec.trm.editor`). Nach der Evaluation eines generativen Ansatzes auf Basis des Eclipse Plugins „Spray“ [KT11; Onl/Spray] erfolgte die Realisierung durch direkte Verwendung von „Graphiti“ [BGK+11] (Spray basiert auf Graphiti). Der Editor bietet zur Platzierung der Komponenten eine zweidimensionale Zeichenfläche. Neue Komponenten und Verbindungen werden mit Hilfe einer Palette am rechten Rand des Editors erstellt. Abbildung 42 zeigt einen Screenshot des grafischen Editors.

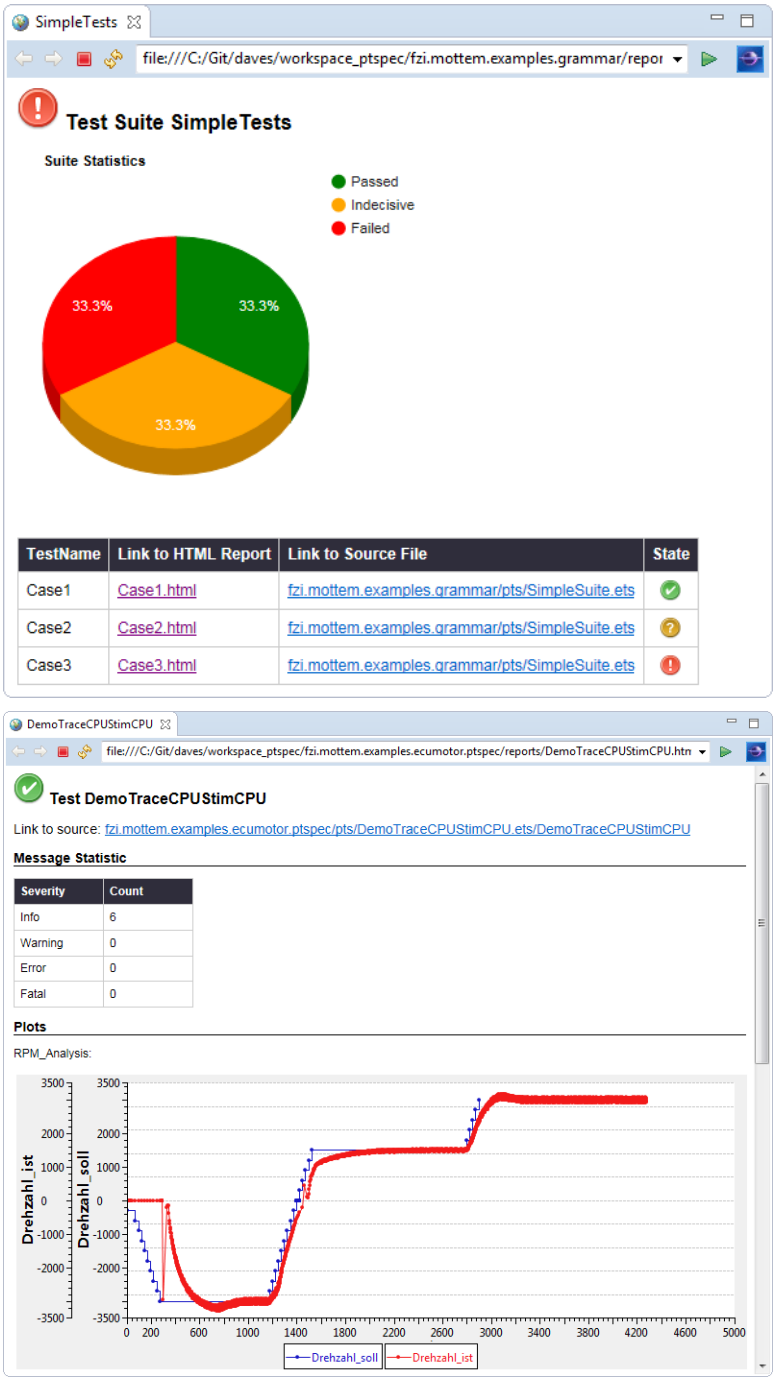


Abbildung 40: Ausschnitte der Reports einer Test Suite (oben) und eines Testfalls (unten)

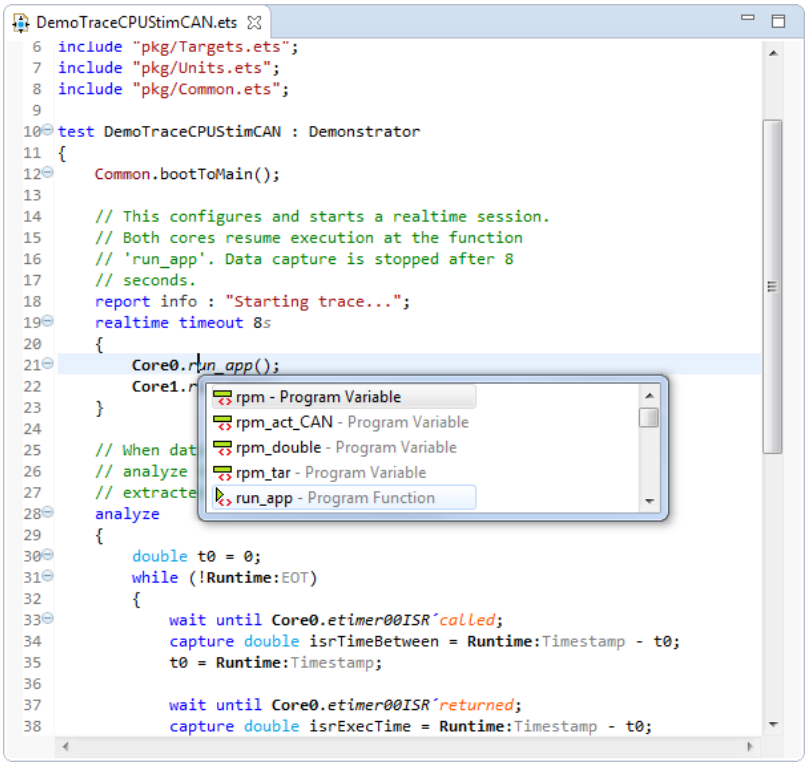


Abbildung 41: Screenshot des Texteditors für ETSpec Quelltext mit Code-Vorschlag

7.4.3 *Echtzeit Visualisierung*

Zur Visualisierung der Daten während der Testausführung und zur Unterstützung interaktiver Tests wurde eine separate grafische Oberfläche realisiert (Echtzeit UI). Die für den Anwender wichtigsten Komponenten sind das *Dashboard* und der *TraceView*. Auf dem Dashboard wird mit Hilfe von Anzeigen der aktuelle Wert einer Testgröße angezeigt, bzw. über Stellknöpfe manipuliert. Im *TraceView* wird der Wertverlauf von Testgrößen dargestellt. Die entsprechende Implementierung in `etspec.rtui` basiert auf „swt-xy-graph“, einer Java Bibliothek zur Visualisierung wissenschaftlicher Daten [Onl/xygraph]. Abbildung 43 zeigt den Screenshot eines Dashboards mit zwei Anzeigen sowie einen TraceView.

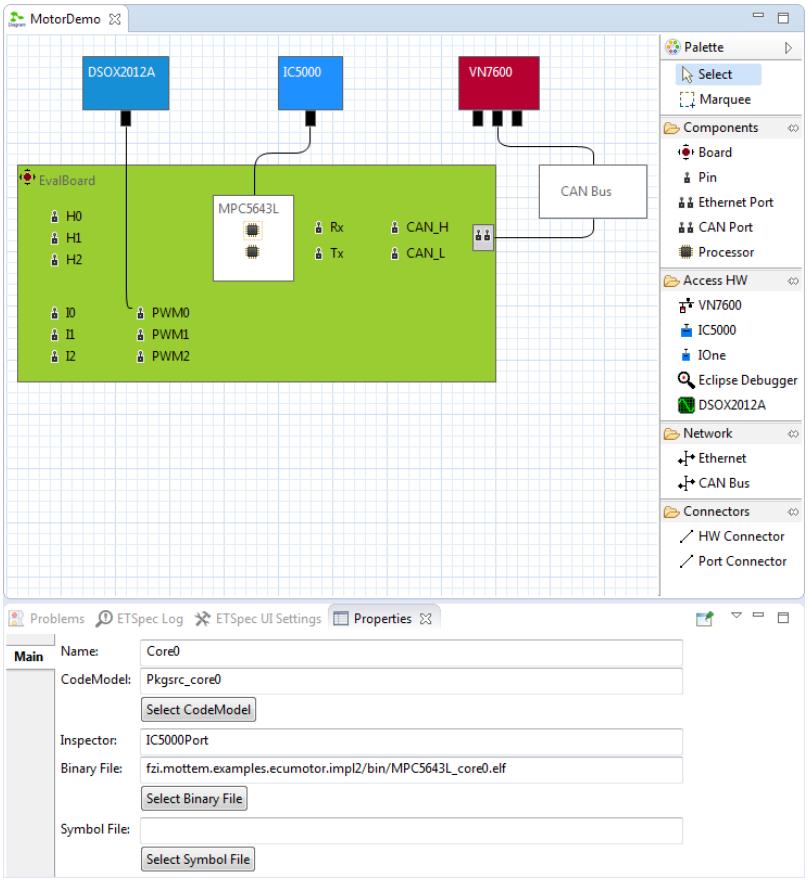


Abbildung 42: Screenshot des grafischen Editors für das TRM

Der Datenaustausch zwischen Echtzeit UI und der Laufzeitumgebung erfolgt über eine separate Komponente, genannt *DataExchanger*, die durch das Plugin `etspec.dx` bereitgestellt wird. Die Anzeigen des Dashboards sowie TraceViews müssen sich beim DataExchanger für die anzuzeigende Testgröße registrieren. Der DataExchanger wird von der Laufzeitumgebung über entsprechende Änderungen informiert und leitet die Werte weiter. In TraceViews können außerdem die Daten aus den Aufzeichnungen vergangener Testläufe zur manuellen Analyse angezeigt werden.

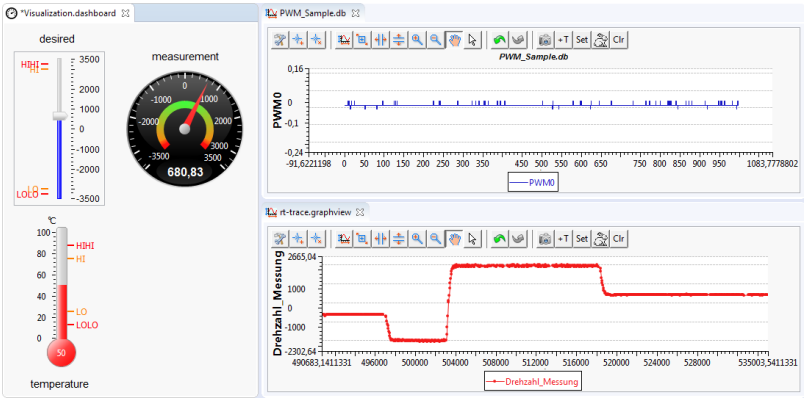


Abbildung 43: Screenshot von Dashboard und TraceView zur Visualisierung

7.4.4 *Projektverwaltung und Logging*

Das Plugin `etspec.navigator` erweitert den „Project Explorer“ der Eclipse IDE, einer Oberfläche zur Darstellung der Verzeichnisse und Dateien im Eclipse Workspace. Mit der Erweiterung können Tests bzw. Test Suiten auf Knopfdruck gestartet werden. ETSpec Quelltextdateien lassen sich dazu „aufklappen“, um die enthaltenen Basiscontainer anzuzeigen. Mit einem Rechtsklick auf einen Test bzw. eine Test Suite wird ein Kontextmenü geöffnet, das den Eintrag „Run“ zum Start der Ausführung bietet. Abbildung 44 zeigt einen entsprechenden Screenshot.
Eine Listenansicht der während einer Testausführung abgesetzten Logmeldungen wird durch das Plugin `etspec.log` implementiert. Durch einen Doppelklick wird automatisch zum Ursprung der Meldung im ETSpec Quelltext navigiert. Abbildung 45 zeigt einen Screenshot mit exemplarischen Logmeldungen.
Es wurden eine Reihe sogenannter „Wizards“ implementiert, die bei der Erstellung von Projekten bzw. Dateien helfen. Wizards werden in Eclipse z.B. durch Rechtsklick im ETSpec Navigator und Auswahl des Unterpunkts „New“) aufgerufen. Die Wizards für ETSpec finden sich im geöffneten Dialogfeld unter dem Menüpunkt „ETSpec“ (siehe Screenshot in Abbildung 46).

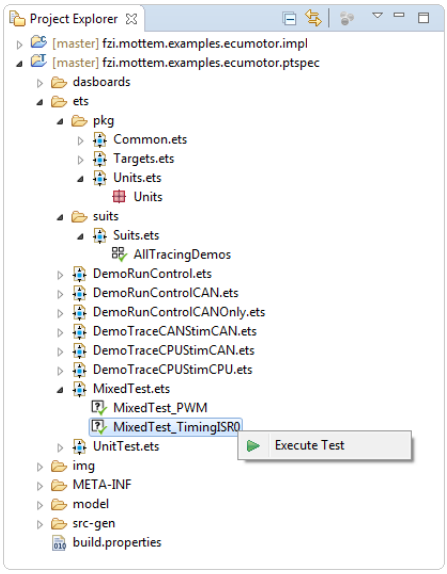


Abbildung 44: Screenshot des erweiterten Project Explorer

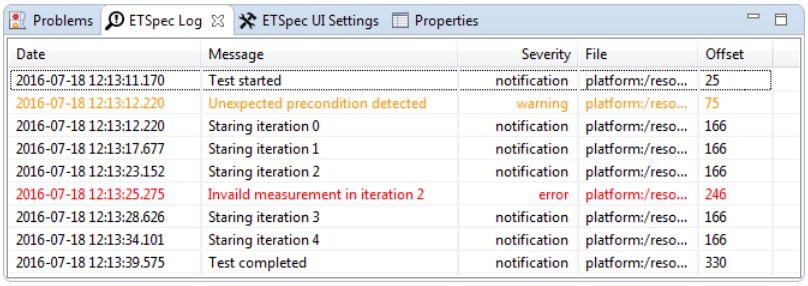


Abbildung 45: Screenshot abgesetzter Meldungen

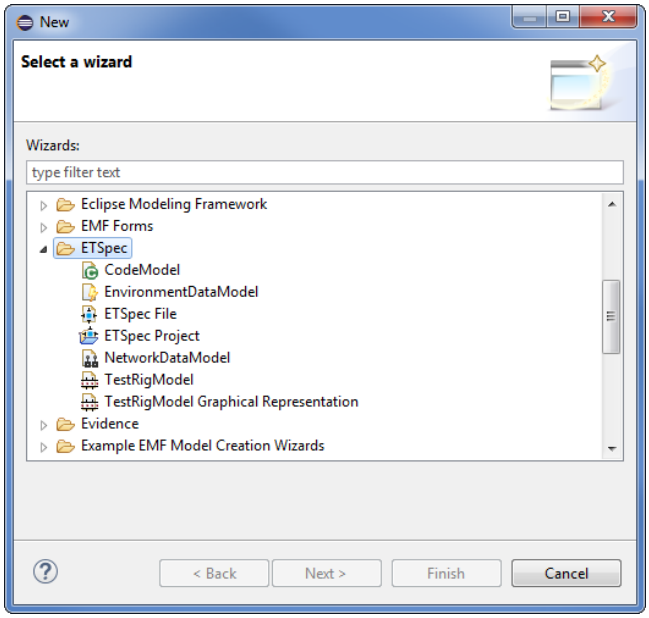


Abbildung 46: Screenshot des Auswahldialogs mit ETSpec Wizards

Teil IV

AUSWERTUNG

Im letzten Teil wird die Implementierung der vorgestellten Testlösung selbst auf den Prüfstand gestellt. Die Einsetzbarkeit wird anhand von Fallstudien aufgezeigt, und es werden mögliche Erweiterungen besprochen.

8 FALLSTUDIEN

Die Idee dieser Arbeit und die Implementierung des ETSpec Frameworks wurde anhand von zwei Anwendungsfällen evaluiert. In beiden Fällen handelt es sich um typische eingebettete Systeme. Im ersten Fall stehen Test und Analyse des Echtzeitverhaltens eines Reglers im Fokus (siehe Abschnitt 8.1). Der zweite Anwendungsfall betrachtet ein verteiltes System bestehend aus zwei ECUs (siehe Abschnitt 8.2). Die Softwareimplementierung beider Anwendungsfälle ist in den Beispielprojekten zum ETSpec Framework enthalten, das unter www.es-tdk.org heruntergeladen werden kann.

8.1 ECHTZEITKRITISCHER REGLER

Als erster Anwendungsfall für das ETSpec Framework dient ein echtzeitkritischer Regler für einen bürstenlosen Gleichstrommotor. Die Motorregelung wurde im Rahmen einer studentischen Abschlussarbeit auf Basis eines MPC5643L Mikrocontrollers mit zwei Rechenkernen implementiert [Kle14].
Auf dem Mikrocontroller kommt kein Betriebssystem zum Einsatz, die Software ist also nicht allzu komplex. Nach der Initialisierung betreten beide Rechenkerne eine Endlosschleife und der weitere Softwareablauf wird durch Interrupts getrieben. Dem ersten Kern ist dabei die Regelungsaufgabe zugeteilt. Drei der Timer-Module des Mikrocontrollers sind so konfiguriert, dass bei einer Änderung der Rotorlage durch die Hall Sensoren des Motors im ersten Kern ein Interrupt ausgelöst wird. Im Kontext dieses Interrupts werden dann die aktuelle Drehzahl und die Ansteuerung zur Kommutierung durch einen PI-Regler[1] berechnet.
Der zweite Kern ist für die Kommunikation über einen CAN Bus zuständig. Darüber werden die Drehzahlvorgabe und Regelparameter empfangen sowie die aktuell gemessene Drehzahl zyklisch versendet. Auch das wird durch Hardware-Interrupts getriggert, die bei der Initialisierung konfiguriert wurden.
Der verwendete Mikrocontroller bietet neben einer einfachen Debug-Schnittstelle auch eine Schnittstelle für das Echtzeit-Tracing (Nexus Class 3, vgl. Abschnitt 3.2.3). Der Mikrocontroller befindet

1 Proportional–Integral Controller.

sich auf einem Evaluationsboard; die elektronischen Komponenten zur Ansteuerung des Motors sind auf einer Lochrasterplatine verlötet. Da es sich um einen einfachen Demonstrationsaufbau handelt, treibt der Motor lediglich einen Propeller an, der zum Schutz in einer Plexiglasröhre montiert ist.
Der reale Testaufbau ist in Abbildung 47 dargestellt, Abbildung 48 zeigt das entsprechende TRM. Neben dem IC5000 Debugger für den Zugriff auf die Debug- und Tracing-Schnittstellen des Mikrocontrollers gehört hierzu auch ein Bus Interface für den Zugriff auf CAN Botschaften sowie ein Oszilloskop zur Aufzeichnung analoger Signale.

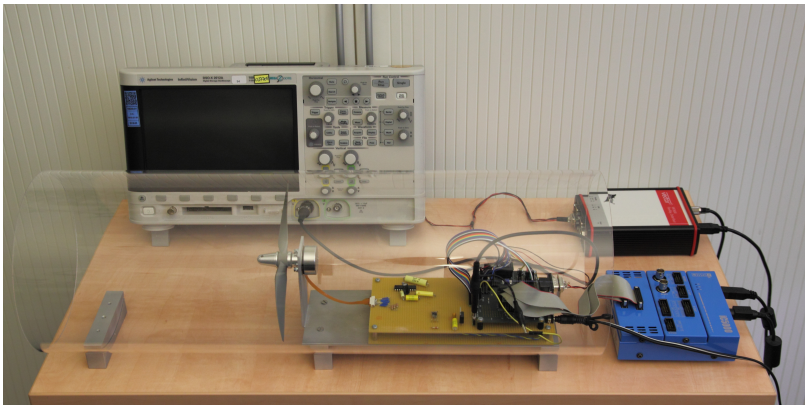


Abbildung 47: Realer Testaufbau zur ersten Fallstudie

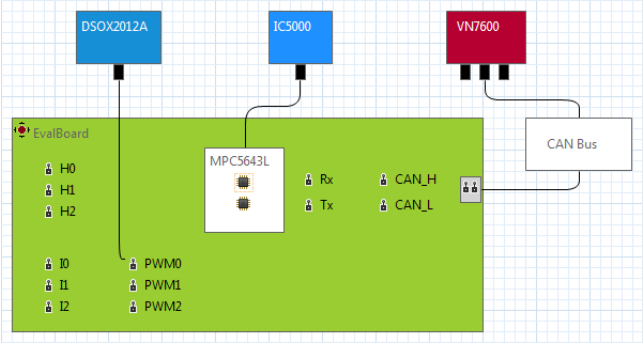


Abbildung 48: TRM zur ersten Fallstudie

8.1.1 *Konkrete Beispiele*

Mit Hilfe der ETSpec Sprache wurden für verschiedene Aspekte der Software Testfälle implementiert, die durch das ETSpec Framework in der beschriebenen Testumgebung automatisch ausgeführt werden können. Bei einer Umstellung auf andere Zugriffshardware bliebe die Testimplementierung erhalten, es müsste lediglich das TRM angepasst werden. Auch von der Ausführungsplattform ist die Testimplementierung unabhängig.
Im Folgenden werden drei Beispiele für Testfälle vorgestellt. Die ersten zwei Beispiele sind Testfälle auf Quelltextebene und Systemebene, das dritte Beispiel ist ein Testfall zur Hardware/Software Integration. Der erste Testfall ließe sich mit einem Unit-Test Framework nahezu identisch darstellen. Wollte man Codeinstrumentierung vermeiden, ginge die Implementierung ohne das ETSpec Framework mit den in Kapitel 5 genannten Nachteilen einher. Die anderen beiden Beispiele lassen sich mit herkömmlichen Ansätzen kaum realisieren.
Im gezeigten Quelltext werden Testgrößen kursiv dargestellt, um sie von Variablen und Funktionen der ETSpec Sprache leichter unterscheiden zu können. Das wird durch den in Eclipse integrierten Editor für die ETSpec Sprache genauso gehandhabt. Die verwendeten Unterfunktionen finden sich in Anhang C.

Einfacher Unit-Test

Eine Software-Funktion, für die sich ein einfacher Unit-Test schreiben lässt, ist die Funktion `update_sector`. Diese „übersetzt“ die Rotorlage, welche durch die drei digitalen Hallsignale gegeben ist, in einen von sechs Sektoren entsprechend Abbildung 49. Der Sektor wird für den nächsten Berechnungsschritt in der Variablen `sector` zwischengespeichert. Der folgende Test prüft den Übersetzungsschritt für alle möglichen Belegungen der Hallsignale.

```
1  test UnitTest_UpdateSector : MPC5643L default Core0
2  {
3      Common.bootToMain();
4
5      // Prüfung erwarteter Zustände.
6      update_sector(1, 0, 1);
7      assert(sector == 1) error;
8      update_sector(1, 0, 0);
9      assert(sector == 2) error;
10     update_sector(1, 1, 0);
11     assert(sector == 3) error;
```

```
12      update_sector(0, 1, 0);
13      assert(sector == 4) error;
14      update_sector(0, 1, 1);
15      assert(sector == 5) error;
16      update_sector(0, 0, 1);
17      assert(sector == 6) error;
18
19      // Prüfung illegaler Zustände.
20      update_sector(0, 0, 0);
21      assert (sector == 0) error;
22      update_sector(1, 1, 1);
23      assert (sector == 0) error;
24  }
```

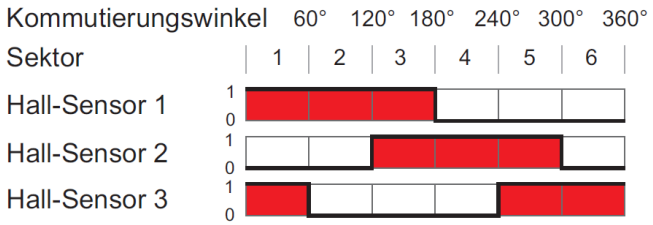


Abbildung 49: Erkennung der Rotorlage mittels Hallsignalen [Kle14]

Dieser Unit-Test ist inhaltlich trivial, zeigt jedoch sehr deutlich, wie nahtlos sich der Zugriff auf Softwaredetails in den Testablauf einfügt. Diese Darstellung entspricht exakt der natürlichen Sichtweise eines Softwareentwicklers.

Einfacher System-Test

Ein einfacher Test der Reglerfunktion besteht in der Vorgabe eines Sollwerts für die Drehzahl und der Prüfung am realen Motor. Da der Testaufbau jedoch keine Zugriffshardware zur direkten Messung der realen Drehzahl umfasst, kann der Test so nicht automatisiert werden. Die Erweiterung des Testaufbaus um eine Umgebungssimulation wäre die beste Möglichkeit, um dieses Problem zu lösen (siehe Abschnitt 8.1.2).
Wenn man jedoch voraussetzt, dass die interne Messung der Drehzahl durch den Mikrocontroller korrekt ist, kann das übrige Reglerverhalten auch mit Hilfe der internen Messung getestet werden. Über die Tracing-Schnittstelle des Prozessors erhält man die gemessene Drehzahl mit einer sehr viel höheren Abtastrate, als es beispielsweise über den CAN Bus möglich wäre. Dieser Ansatz ist im folgenden Testfall implementiert. Die Drehzahlvorgabe wird

durch Setzen eines Werts für das CAN Signal `Drehzahl` über den CAN Bus gesendet (das Signal muss dazu im *NetworkDataModel* entsprechend konfiguriert sein). Die anschließende Analyse basiert auf dem Daten-Trace der Variablen `rpm`.

```
1  test ControllerTest : MPC5643L_and_CANPort default Core0
2  {
3      Common.bootToMain();
4
5      realtime timeout 8s
6      {
7          Core0.run_app();
8          Core1.run_app();
9          wait 1s;
10         CANPort.Drehzahl = 2000 min^-1;
11     }
12     analyze
13     {
14         // Finde Extremwerte der Drehzahl im Bereich
15         // 2 Sekunden nach Start bis Ende der Aufzeichnung
16         float maxRpm = SignalAnalyzer.getMaximumAfter(
17             ref rpm, 2s);
18         float minRpm = SignalAnalyzer.getMinimumAfter(
19             ref rpm, 2s);
20
21         // Erwartung: Abweichung kleiner als 50 U/min
22         assert((maxRpm - 2000) < 50) error;
23         assert((2000 - minRpm) < 50) error;
24     }
25 }
```

Testfall zur Hardware/Software Integration

Die Software umfasst einen Treiber für das PWM Modul des Mikrocontrollers. Beim Start des Mikrocontrollers konfiguriert dieser Treiber alle drei PWM Ausgänge mit einer Periodendauer von 25 µs. Der Tastgrad aller Ausgänge wird durch einen Aufruf der Funktionen `updatePWM()` gesetzt. Dabei wird ein in der Variablen `pwm_dc` gespeicherter Wert verwendet.
Auch dieser Hardware/Software Integrationstest kann mit Hilfe des ETSpec Frameworks sehr einfach über herkömmliche Grenzen hinweg spezifiziert werden. Im folgenden Beispiel erfolgt die Stimulation direkt über die entsprechenden Softwaregrößen, d.h. durch Überschreiben der Variablen `pwm_dc` und anschließendem

Aufruf der Funktion `updatePWM()`. Der geschriebene Wert `0.5` entspricht einem Tastgrad von 50%. Anschließend wird zur Aufzeichnung des Signalverlaufs am Pin `PWM0` der Schwellwert `0.1` Volt als Trigger konfiguriert.
Nach Abschluss der Aufzeichnung werden im `AnalyzeBlock` die Taktflanken identifiziert, und die tatsächliche Periodendauer sowie der Tastgrad berechnet. Zur Vereinfachung verzichtet der Testfall auf eine Fehlerbehandlung (z.B. wenn der konfigurierte Trigger nicht auslöst) und betrachtet außerdem nur die erste Periode im aufgezeichneten Verlauf.

```
1  test PWMModuleIntegration : MPC5643L_and_PWM0
2  {
3      Common.bootToMain();
4
5      Core0.pwm_dc = 0.5;
6      Core0.updatePWM();
7
8      realtime timeout 5s
9      {
10         PWM0.Value:TriggerAbove = 0.1 V;
11     }
12     analyze
13     {
14         physical[s] t0 = SignalAnalyzer.findRisingEdge(
15             ref PWM0.Value, 0 s, 0.1 V);
16         physical[s] t1 = SignalAnalyzer.findRisingEdge(
17             ref PWM0.Value, t0, 0.1 V);
18         physical[s] t2 = SignalAnalyzer.findRisingEdge(
19             ref PWM0.Value, t1, 0.1 V);
20
21         physical[s] period = t2-t0;
22         double duty = <double>((period-(t2-t1))/period);
23         assert ((duty > 0.49) && (duty < 0.51)) error;
24         assert ((period > 24us) && (period < 26us)) error;
25     }
26 }
```

Dieses und das vorhergehende Beispiel veranschaulichen sehr deutlich, wie sich einzelne Teile des Testobjekts sehr einfach „herausschneiden“ und gezielt testen lassen, ohne Teile des Testobjekts zu instrumentieren. Die mit herkömmlichen Werkzeugen schwer zu überwindende Grenze zwischen Hardware- und Softwareschnittstellen ist nicht mehr existent.

8.1.2 *Erweiterung um Umgebungssimulation*

In der beschriebenen Fallstudie kommt keine Umgebungssimulation vor, sondern das „Steuergerät" ist mit dem realen Motor verbunden. Das ist in erster Linie der Tatsache geschuldet, dass für diese Arbeit kein geeignetes HiL Testsystem zur Verfügung stand. Außerdem ist es anschaulicher, wenn sich die Testausführung in der Realität beobachten lässt.
Die Erweiterung der Fallstudie um eine Simulation des Gleichstrommotors ist jedoch leicht vorstellbar. Der schwierigste Aspekt ist die Erstellung des Umgebungsmodells, insbesondere die präzise Modellierung des Motorverhaltens.[2] Das liegt nicht im Fokus dieser Arbeit, und zur Demonstration des ETSpec Frameworks ist die Verwendung des realen Motors daher auch sinnvoller.
Soll der reale Motor dennoch durch eine Simulation ersetzt werden, wäre dies weitgehend transparent. Hinsichtlich der zuvor beschriebenen Testfälle ergeben sich keinerlei Änderungen. Erst wenn zur Testimplementierung die neuen Umgebungssignale verwendet werden sollen (z.B. die in der Simulation berechnete Motordrehzahl anstelle der internen Messung, oder ein Lastmoment an der Motorwelle, was einen neuen Testfall darstellt), muss das TRM erweitert und die Testimplementierung angepasst werden.

8.2 VERTEILTES SYSTEM

Der zweite Anwendungsfall ist ein System bestehend aus zwei ECUs, die auf ein kleines Raupenfahrzeug montiert sind und über einen CAN Bus miteinander kommunizieren. Für die ECUs wurden zwei Evaluationsboards mit einem STM32F4 Mikrocontroller verwendet. Die Anwendungsfunktion ist einfach: über einen Infrarotsensor wird der Abstand zu einem Hindernis gemessen, und es wird versucht diesen Abstand konstant zu halten (einfache ACC[3] Funktion). Die Abstandsregelung und Motoransteuerung wird durch die erste ECU durchgeführt, die Andere stellt die Messwerte des Infrarotsensors zur Verfügung.

2 „Schwierig" ist dabei weniger die (algorithmische) Implementierung, sondern das Finden der korrekten Parametrierung bezüglich der physikalischen Randbedingungen wie Reibung, Trägheit, Temperatur etc.

3 Adaptive Cruise Control

Die Software dieser Fallstudie ist jedoch deutlich komplexer. Auf beiden Controllern wird ERIKA Enterprise ausgeführt, ein quelloffenes OSEK/VDX[4] konformes Echtzeitbetriebssystem. Durch eine Middleware wird von der Steuergerätehardware abstrahiert (das realisierte Konzept ist vergleichbar mit der Runtime Environment aus AUTOSAR, die Implementierung ist jedoch vereinfacht). Auch diese Plattform wurde im Rahmen einer Abschlussarbeit realisiert [Kro16].
Diese Fallstudie zeigt in erster Linie die Skalierbarkeit der implementierten Lösung bezüglich der Größe der Codebasis der getesteten Software. Darüber hinaus wurde in diesem Fall ein spezieller Debugger eingesetzt, zu dem via Bluetooth eine Verbindung aufgebaut werden kann. Durch die drahtlose Kommunikation kann das Verhalten auf der realen Hardware beobachtet werden, während das Testobjekt in Bewegung ist. Die Stimulation und Reaktion auf das Testobjekt ist wegen der hohen Latenz über Bluetooth natürlich nur unter sehr schwachen Echtzeitanforderungen möglich. Der reale Testaufbau ist in Abbildung 50 dargestellt, Abbildung 51 zeigt wieder das zugehörige TRM.

4 Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug/Vehicle Distributed Executive [Std/ISO05], Grundlage von AUTOSAR

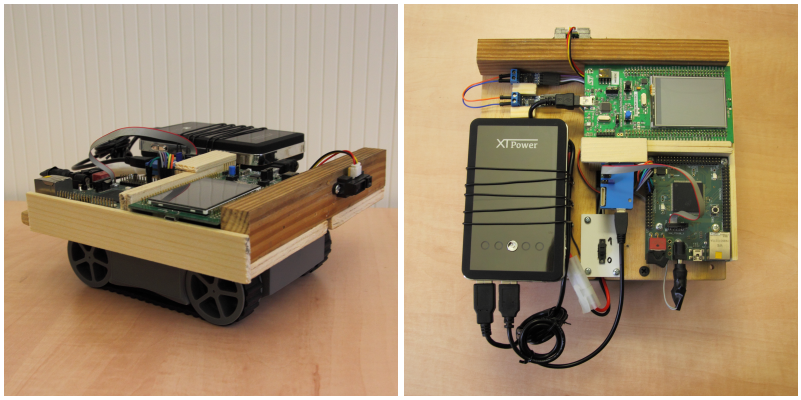


Abbildung 50: Realer Testaufbau zur zweiten Fallstudie

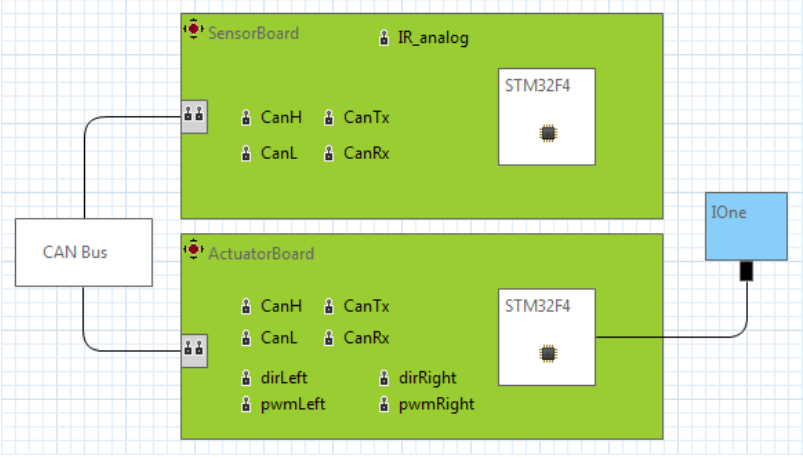


Abbildung 51: TRM zur zweiten Fallstudie

9 AUSBLICK UND FAZIT

In den ersten Teil dieses Kapitels werden mögliche Erweiterungen des Gesamtkonzepts sowie Verbesserungen der Implementierung besprochen. Anschließend wird in Abschnitt 9.3 ein Fazit gegeben.

9.1 KONZEPTUELLE ERWEITERUNG

Zur Erschließung zusätzlicher Einsatzbereiche kann das ETSpec Framework konzeptuell erweitert werden. Diesbezüglich sind die folgenden zwei Ideen besonders interessant.

9.1.1 *Verwendung von Codeinstrumentierung*

Eine grundlegende Entscheidung in dieser Arbeit war der Verzicht auf Codeinstrumentierung. Der wichtigste Grund hierfür war die Tatsache, dass Codeinstrumentierung das Softwareverhalten ändert, und somit zu weniger belastbaren Testergebnissen führt (vgl. Abschnitt 3.4.3). In vielen Fällen kommt Codeinstrumentierung jedoch durchaus in Frage, und es ist sinnvoll, diese Fälle durch das ETSpec Framework ebenfalls zu unterstützen.

Daten-Trace Rekonstruktion

Ein wichtiger Anwendungsfall von Codeinstrumentierung ist die Kompensation des Fehlens einer breitbandigen Trace-Schnittstelle. Im Rahmen des ETSpec Frameworks wäre es beispielsweise denkbar, alle im `AnalyzeBlock` referenzierten Funktionen und Variablen der getesteten Software um sogenannte „Wächter“ zu ergänzen, welche sämtliche Zugriffe auf die Variable bzw. Funktion protokollieren.[1] Damit lässt sich der Daten-Trace im Nachhinein rekonstruieren.

Stimulation auf Quelltextebene

Darüber hinaus bietet es sich an, Codeinstrumentierung zur Stimulation auf Quelltextebene zu verwenden. Mit Unterstützung eines Echtzeitbetriebssystems auf dem Target wäre es möglich, einen zusätzlichen Thread auszuführen, der die gewünschten Stimuli in das Testobjekt injiziert. Der entsprechende Code ließe sich aus der Beschreibung im `RealtimeBlock` generieren. Dieser Ansatz ist gegenüber der Verwendung der Debug-Schnittstelle zur Echtzeit-

1 Das lässt sich als Anwendungsfall aspektorientierter Programmierung [KLM+97] auffassen.

Stimulation besser geeignet, da die Latenz zur Kommunikation mit der Zugriffshardware entfällt.
Alle Vorschläge zur Verwendung von Codeinstrumentierung erfordern selbstverständlich einen Eingriff in den Buildprozess der getesteten Software, der sich mit dem bisherigen Ansatz vollständig vermeiden ließ.

9.1.2 *Übertragung auf Software anderer Bereiche*

In dieser Arbeit wurde ausschließlich eingebettete Software betrachtet. Der Fokus lag dabei auf der Automobilindustrie sowie den in Kapitel 1 genannten Bereichen. Deswegen wurde das ETSpec Framework sehr gezielt für den Test von C-Code und die Verwendung der XiL Testmethoden entworfen, und eine Ausdehnung des Konzepts auf ganz andere Bereiche erscheint zunächst nicht naheliegend. Nichtsdestotrotz können Teile des ETSpec Frameworks für ein allgemeineres Werkzeug zur Testentwicklung verwendet werden, das auch für den Test gewöhnlicher Desktop-Anwendungen, Smartphone-Apps geeignet ist. Auch hier kann der Testprozess durch ein universelles Werkzeug für Quelltextebene und Systemebene profitieren, jedoch in geringerem Maße, da die Werkzeuglandschaft bereits vergleichsweise homogen ist.
Bezüglich der Quelltextebene ist die Übertragbarkeit des Konzepts für das ETSpec Framework leicht einzusehen, denn hier bestehen kaum Unterschiede in den Anforderungen. In allen Fällen sind Unit-Test Frameworks das bevorzugte Werkzeug zur Testautomatisierung, Unterschiede ergeben sich im Wesentlichen nur bezüglich der verwendeten Programmiersprache. Basierend auf der Idee von Testgrößen wird durch das *CodeModel* bereits von der Programmiersprache C abstrahiert. Auf dieselbe Weise ist auch der Unit-Test beliebiger anderer Programmiersprachen möglich. Das bisherige Vorgehen für den Zugriff auf die entsprechenden Testgrößen, also die Verwendung eines Debuggers zur Vermeidung von Codeinstrumentierung, muss nicht geändert werden. Debug- und Trace-Hardware lässt sich dazu durch eine Softwarelösung ersetzen, beispielsweise den Java Debugger (jdb) oder GNU Debugger (gdb), die sich mit einem geeigneten Treiber einfach in die ETSpec Laufzeitumgebung einbinden lässt.

Unit-Test anderer Programmiersprachen

Bezüglich der Systemebene ergeben sich hingegen gänzlich andere Anforderungen. Bei Desktop-Anwendungen und Smartphone-Software spielt Reglerverhalten und somit eine echtzeitkritische Umgebungssimulation keine Rolle, auch der Signalbegriff existiert

Oberflächentest

hier nicht. Stattdessen stellen die vielfältigen Möglichkeiten zur Benutzerinteraktion und grafische Oberfläche neue Herausforderungen dar. Dazu können die Steuerelemente der Oberfläche als weitere Testgröße aufgefasst und in einem zusätzlichen Modell beschrieben werden. Benutzerinteraktionen lassen sich prinzipiell durch zusätzliche Ereignisse in der ETSpec Sprache abbilden. Die größte Gefahr dieses Vorhabens ist, dass die Nutzerfreundlichkeit zu stark darunter leidet, weil sich dem Anwender zu viele Optionen bieten.

9.2 TECHNISCHE ERWEITERUNGEN

Die prototypische Implementierung der vorgestellten Testlösung bietet an verschiedenen Stellen Potenzial für Verbesserungen. In diesem Abschnitt werden einige vielversprechende Punkte angesprochen.

9.2.1 *Erweiterte Echtzeitfähigkeit und Synchronisation*

Die Implementierung der ETSpec Laufzeitumgebung ist nur bedingt echtzeitfähig. Das liegt an der Ausführung durch eine Java Virtual Machine, bei welcher jederzeit Garbage Collection möglich ist, sowie den schwer abschätzbaren Latenzen bei der Ansteuerung von Zugriffshardware. Durch die Verwendung einer anderen Basistechnologie ließe sich sowohl die Performance als auch die Echtzeitfähigkeit der Laufzeitumgebung erheblich verbessern. Idealerweise wird die Laufzeitumgebung in ein Echtzeitbetriebssystem eingebettet, welches ein deterministisches Scheduling der zur Testausführung gestarteten Threads (*concurrent* Anweisung) ermöglicht. Zudem wäre es vorteilhaft, wenn der ETSpec Quelltext in nativen Maschinencode übersetzt wird. Das lässt sich unter anderem durch die Verwendung von C++ als Host-Sprache und entsprechend angepasste Code-Generierung erreichen.

Native Laufzeitumgebung

Darüber hinaus könnte das ETSpec Framework einen Beitrag zur Verbesserung der Qualität von Zeitstempeln leisten, indem es eine Methode zur Uhrzeitsynchronisation bietet. Hierzu bietet sich das Precision Time Protocol (PTP) [Std/IEEo8] an, welches zur Uhrzeitsynchronisation in IP-Netzwerken bereits weit verbreitet ist. Reine Softwareimplementierungen des Protokolls erreichen eine Präzision im Bereich 5 bis 50 µs, die durch Hardwareunterstützung zur Zeitstempelerfassung noch deutlich erhöht werden kann (±60 ns) [Onl/Hir].

Uhrzeitsynchronisation

9.2.2 *Modellbasierte Softwareentwicklung*

Anwendungsfunktionen eingebetteter Systeme werden häufig modellbasiert entwickelt. In diesem Fall finden sich die relevanten Testgrößen im Regler- bzw. Steuerungsmodell. Gängige Codegeneratoren wie Embedded Coder von The MathWorks oder TargetLink von dSPACE liefern glücklicherweise ein Mapping zwischen den Modellvariablen und -signalen und den Variablen im generierten Code. Prinzipiell sind also Informationen verfügbar, um für den Zugriff auf eine der neuen Testgrößen die entsprechende Speicheradresse im Target herauszufinden.
Erschwerend kommt jedoch hinzu, dass im Zuge der Optimierung dieselbe Softwarevariable bzw. Speicheradresse für mehrere Modellvariablen genutzt werden kann. Das Problem lässt sich am einfachsten umgehen, indem die entsprechende Optimierung zu Testzwecken ausgeschaltet wird. Das hat jedoch zur Folge, dass eine andere Version der Software getestet wird, als das finale Produkt, zumindest falls die Optimierungen später wieder aktiviert wird.

9.2.3 *Unterstützung von C++ auf dem Target*

Bei der Entwicklung kritischer eingebetteter Software wird bisher auf die Verwendung von C++ im größeren Stil verzichtet (vgl. Abschnitt 3.4.1). Mit der steigenden Komplexität der Software wird die Verwendung von C++ jedoch zunehmend attraktiv, sodass die Unterstützung von C++ Code eine interessante Erweiterung des ETSpec Frameworks darstellt.
Das erfordert eine Erweiterung des *CodeModels*, was jedoch die kleinste Herausforderung darstellt. Ebenfalls nur eine kleine Hürde ist der RPC einer überladenen Funktion, weil durch das Festhalten an der Typsicherheit über die Sprachgrenze hinweg der gewünschte Kandidat eindeutig bleibt. Schwieriger ist die entfernte dynamische Instanziierung von Objekten, da hierzu erst Speicher auf dem Heap des Targets allokiert werden muss. Schließlich sind virtuelle Funktionen eine Herausforderung, weil hier die Adresse der aufzurufenden Funktion mit Hilfe von *vpointer* und *vtable* berechnet werden muss. Derzeit bieten APIs für Debug- und Trace-Hardware keine geeignete Schnittstelle, welche diese Details verbirgt.

9.2.4 *Grafische Modellierung der Implementierung*

Für die Testimplementierung wurde ein textuelles Beschreibungsmittel einer grafischen Alternative vorgezogen (vgl. Abschnitt 4.3.2). Für manche Anwendungsfälle sind grafische Werkzeuge jedoch äußerst praktisch und von Anwendern gewünscht. Gute Beispiele sind die Modellierung analoger Signalverläufe, die als Testdaten für Stimuli genutzt werden, oder die Beschreibung eines Testablaufs in Form eines Zustandsautomaten.
Das ließe sich durch separate Werkzeuge realisieren, die entsprechenden ETSpec Quelltext generieren. Wie jedoch bereits in Abschnitt 2.4.3 erläutert wurde, kommt es bei einem solchen Vorgehen im späteren Verlauf häufiger zu Integrationsproblemen. Besser wäre es daher, wenn zusätzliche grafische Modelle ähnlich wie das TRM direkt mit der Sprachdefinition verknüpft werden. Das heißt, es wird bereits auf der Metaebene festgelegt, wie die Modelle in der Sprache zu verwenden sind. Das schränkt die Flexibilität zwar etwas ein, vermeidet aber Probleme bei der Integration.
In diesem Fall bieten sich zwei Alternativen: entweder wird aus dem grafischen Modell trotzdem zuerst ETSpec Quelltext erzeugt, oder aber es wird der bereits bestehende ETSpec Code Generator, der den Quelltext der Host-Sprache erzeugt, erweitert, sodass er das grafische Modell direkt verarbeiten kann. Aus softwaretechnischer Sicht ist der zweite Ansatz die bessere Lösung, da im anderen Fall die Wahrscheinlichkeit von Inkonsistenzen im ETSpec Frameworks deutlich größer ist.

9.3 FAZIT

Im Rahmen dieser Arbeit wurden heterogene Technologien für den Test eingebetteter Software identifiziert, und ein neuer Ansatz zur Integration dieser Technologien zu einer durchgängigen Testlösung vorgestellt. Mit einer umfangreichen Implementierung wurde gezeigt, dass sich das Konzept basierend auf der neuen Programmiersprache ETSpec in der Praxis umsetzen lässt. Der Ansatz homogenisiert die Verwendung von Testtechnologien auf Quelltextebene und Systemebene, und damit auch den Testprozess. Teile des Testobjekts, die an der Grenze zwischen Hardware und Software liegen, lassen sich durch neuartige Testfälle gezielter testen.
Testimplementierungen, die unter Verwendung der ETSpec Sprache realisiert werden, sind mit großer Wahrscheinlichkeit robuster, als eine äquivalenter Implementierung mit einer anderen Program-

miersprache. Gleichzeitig ist eine Effizienzsteigerung bei der Testentwicklung zu erwarten. Diese Vermutungen konnten im Rahmen dieser Arbeit leider nicht empirisch belegt werden. Eine Analyse der jeweiligen Werkzeuge legt diesen Schluss jedoch nahe: entscheidend ist in beiden Fällen, dass bei Verwendung des ETSpec Frameworks durch statische Codeanalyse bereits zur Implementierungszeit Konsistenz mit dem Testobjekt sichergestellt wird, was bei anderen Lösungen nicht in diesem Umfang der Fall ist. Hinzu kommen die speziellen Anweisungen der ETSpec Sprache, durch die sich viele Sachverhalte klarer darstellen lassen.
Auch indirekt ist ein positiver Einfluss auf den Testprozess zu erwarten: durch das Vorhandensein einer universellen Lösung zur Testautomatisierung können entsprechend geschulte Entwickler flexibler eingesetzt werden, als wenn sie ein Werkzeug beherrschen, das nur in einer bestimmten Testphase eingesetzt werden kann.
Schließlich ist mit dieser Arbeit ein weiteres interessantes Beispiel für das Ineinandergreifen von Modellierung und Programmierung gegeben, und es wird eine Möglichkeit aufgezeigt, wie sich stark unterschiedliche Abstraktionen praxistauglich zusammenführen lassen.

Teil v

APPENDIX

A

BEZEICHNER UND LITERALE DER ETSPEC SPRACHE

```
ID:
    ('a'..'z'|'A'..'Z'|'_')
    ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

FQN:
    ID ('.' ID)*;

BOOL_LITERAL:
    'true' | 'false';

INTEGER_LITERAL:
    ('0'..'9')+ |
    ( '0x' (
      ('0'..'9') | ('a'..'f') | ('A'..'F')
    )+ );

REAL_LITERAL:
    ('0'..'9')+ '.' ('0'..'9')+
    (('e' | 'E') ('+' | '-')? ('0'..'9')+ )?
    ('F' | 'f')?

CHARACTER_LITERAL:
    "'"
    !('\'|"'") |
    ( "\'" | '\b'  | '\f'  | '\n'  | '\r'  | '\t'  | '\\' |
      '\u' (('0'..'9') | ('a'..'f') | ('A'..'F'))+ )
    "'";

STRING_LITERAL:
    '"' (
    !('\'|'"') |
    ( '\"' | '\b'  | '\f'  | '\n'  | '\r'  | '\t'  | '\\' |
      '\u' (('0'..'9') | ('a'..'f') | ('A'..'F'))+ )
    )* '"';

NULL_LITERAL:
    'null';
```

B

XTEXT GRAMMATIK DER ETSPEC SPRACHE

```
grammar fzi.mottem.ptspec.dsl.ETSpec
        with org.eclipse.xtext.common.Terminals

generate pTSpec "http://www.mottem.fzi/ptspec/dsl/ETSpec"

import "http://www.eclipse.org/emf/2002/Ecore" as Ecore
import "http://www.fzi.de/mottem/model/BaseElements"
       as BaseElements
import "http://www.eclipse.org/xtext/common/JavaVMTypes"
       as JavaTypes

ETSRoot:
  includes+=ETSInclude* imports+=ETSJavaImport*
  containerDeclarations+=ETSContainerDeclaration*
;

ETSInclude:
  'include' importURI=STRING ';'
;

ETSJavaImport:
  'import' importedType=
   [JavaTypes::JvmDeclaredType|FULLQUALIFIEDNAME] ';'
;

ETSContainerDeclaration
  returns BaseElements::IReferenceableContainer:
    ETSTargetDeclaration | ETSTestDeclaration |
    ETSTestSuiteDeclaration | ETSPackageDeclaration
;

/*
 * -------------------------------------------------------
 * Target
 * -------------------------------------------------------
 */

ETSTargetDeclaration returns ETSContainerDeclaration:
  {ETSTargetDeclaration}
  'target' name=ID
```

```
41    '{'
42      (list=ETSTargetDefinitionList)?
43    '}'
44  ;
45
46  ETSTargetDefinitionList:
47    actualTargets+=
48      [BaseElements::IExecutor|FULLQUALIFIEDNAME] (','
49    actualTargets+=
50      [BaseElements::IExecutor|FULLQUALIFIEDNAME])*
51  ;
52
53  /*
54   * -------------------------------------------------------
55   * Package
56   * -------------------------------------------------------
57   */
58
59  ETSPackageDeclaration returns ETSContainerDeclaration:
60    {ETSPackageDeclaration}
61    'package' name=ID
62      ( ':' target=[ETSTargetDeclaration|ID] )?
63    '{'
64      (packageElements+=
65      (ETSPackageFunction | ETSPackageVariable |
66       ETSPackageUnit)
67      )*
68    '}'
69  ;
70
71  /*
72   * -------------------------------------------------------
73   * Package Content
74   * -------------------------------------------------------
75   */
76
77  ETSPackageVariable returns ETSPackageElement:
78    {ETSPackageVariable}
79    (const?='const' | global?='global') dataType=ETSDataType
80    declaration=ETSPackageVariableDeclaration
81    (withInit?='=' initialValue=ETSConstant)? ';'
82  ;
83
84  ETSPackageFunction returns ETSPackageElement:
85    {ETSPackageFunction}
86    (
87      (realtimeFunc?='realtime' | analyzeFunc?='analyze')?
```

```
88      'func' returnDataType=ETSDataType
89        declaration=ETSPackageFunctionDeclaration
90      '(' (parameterList=ETSPackageFuncParameterList)? ')'
91      (withDefault?='default' executor=ETSExecutor)?
92       )
93    '{'
94      implementation=ETSImplementation
95    '}'
96  ;
97
98  ETSPackageFuncParameterList:
99    parameters+=ETSPackageFuncParameter
100   (',' parameters+=ETSPackageFuncParameter)*
101 ;
102
103 ETSPackageFuncParameter:
104   {ETSPackageFuncParameter}
105   (referenceAccess?='ref')? dataType=ETSDataType
106   declaration=ETSPackageFuncParameterDeclaration
107 ;
108
109 /*
110  * -------------------------------------------------------
111  * Package Units
112  * -------------------------------------------------------
113  */
114
115 ETSPackageUnit returns ETSPackageElement:
116   {ETSPackageUnit}
117   'unit' declaration=ETSUnitDeclaration
118   (derived?='[' baseUnit=[ETSUnitDeclaration|UNITNAME] '='
119   expression=ETSUnitExpression ']')? ';'
120 ;
121
122 ETSUnitDeclaration:
123   {ETSUnitDeclaration}
124   name=UNITNAME
125 ;
126
127 ETSUnitExpression:
128   {ETSUnitExpression}
129   (
130     ( bracketed?='('
131       innerExpression=ETSUnitExpression ')' ) |
132     ( unit=[ETSUnitDeclaration|UNITNAME] ) |
133     ( => constant=NUMBER )
134   )
```

```
135   (operationExpression=ETSUnitOperationExpression)?
136 ;
137
138 ETSUnitOperationExpression:
139   {ETSUnitOperationExpression}
140   op=ETS_EUNITOPERATOR rhs=ETSUnitExpression
141 ;
142
143 /*
144  * -------------------------------------------------------
145  * TestSuite
146  * -------------------------------------------------------
147  */
148
149 ETSTestSuiteDeclaration returns ETSContainerDeclaration:
150   {ETSTestSuiteDeclaration}
151   'suite' name=ID '{'
152     (list=ETSTestDeclarationList)?
153   '}'
154 ;
155
156 ETSTestDeclarationList:
157   tests+=[ETSTestDeclaration|ID]
158   (',' tests+=[ETSTestDeclaration|ID])*
159 ;
160
161 /*
162  * -------------------------------------------------------
163  * Test & Implementation
164  * -------------------------------------------------------
165  */
166
167 ETSTestDeclaration returns ETSContainerDeclaration:
168   {ETSTestDeclaration}
169   'test' name=ID (':' target=[ETSTargetDeclaration|ID])?
170   ('default' defaultExecutor=ETSExecutor)?
171   '{'
172     implementation=ETSImplementation
173   '}'
174 ;
175
176 ETSImplementation:
177   {ETSImplementation}
178   (statements+=ETSStatement)*
179 ;
180
181 ETSSingleStatementImplementation returns ETSImplementation:
```

```
182   statements+=ETSStatement
183 ;
184
185 ETSExecutor:
186   {ETSExecutor}
187   actualExecutor=[BaseElements::IExecutor|FULLQUALIFIEDNAME]
188 ;
189
190 /*
191  * -------------------------------------------------------
192  * Statements
193  * -------------------------------------------------------
194  */
195
196 ETSStatement:
197   ETSDeclarationStatement |
198   ETSExpressionStatement |
199   ETSReturnStatement |
200   ETSLoopStatement |
201   ETSLoopControlStatement |
202   ETSTryCatchStatement |
203   ETSIfThenElseStatement |
204   ETSRunStatement |
205   ETSRunUntilStatement |
206   ETSStopStatement |
207   ETSStopOnStatement |
208   ETSConcurrentStatement |
209   ETSCancelStatement |
210   ETSWaitTimeStatement |
211   ETSTriggerStatement |
212   ETSWaitUntilStatement |
213   ETSWaitDeltaStatement |
214   ETSAssertStatement |
215   ETSReportStatement |
216   ETSSwitchCaseStatement |
217   ETSScopeStatement |
218   ETSRealtimeStatement
219 ;
220
221 /*
222  * -------------------------------------------------------
223  * Common statements (like other programming languages)
224  * -------------------------------------------------------
225  */
226
227 ETSDeclarationStatement returns ETSStatement:
228   {ETSDeclarationStatement}
```

```
229   (final?='final' | capture?='capture')?
230   declarator=ETSDeclarator
231   (withAssignment?='=' assignment=ETSExpression)? ';'
232 ;
233
234 ETSExpressionStatement returns ETSStatement:
235   {ETSExpressionStatement}
236   expression=ETSExpression ';'
237 ;
238
239 ETSReturnStatement returns ETSStatement:
240   {ETSReturnStatement}
241   'return' expression=ETSExpression? ';'
242 ;
243
244 ETSLoopStatement returns ETSStatement:
245   ETSForLoopStatement | ETSWhileLoopStatement |
246   ETSDoWhileLoopStatement
247 ;
248
249 ETSForLoopStatement returns ETSLoopStatement:
250   {ETSForLoopStatement}
251   'for' '('
252   (forDecls+=ETSDeclarationStatementForLoop(','
253   forDecls+=ETSDeclarationStatementForLoop)*)? ';'
254   cancelExpression=ETSExpression ';'
255   (iterationExpression=ETSExpression)?
256   ')' '{'
257     implementation=ETSImplementation
258   '}'
259 ;
260 ETSDeclarationStatementForLoop:
261   (capture?='capture')? declarator=ETSDeclarator
262   (withAssignment?='=' assignment=ETSExpression)?
263 ;
264
265 ETSWhileLoopStatement returns ETSLoopStatement:
266   {ETSWhileLoopStatement}
267   'while' '(' cancelExpression=ETSExpression ')' '{'
268     implementation=ETSImplementation
269   '}'
270 ;
271
272 ETSDoWhileLoopStatement returns ETSLoopStatement:
273   {ETSDoWhileLoopStatement}
274   'do' '{'
275     implementation=ETSImplementation
```

```
276   '}' 'while' '(' cancelExpression=ETSExpression ')' ';'
277 ;
278
279 ETSLoopControlStatement returns ETSStatement:
280   {ETSLoopControlStatement}
281   controlType=ETS_ELOOPCTRLSTATEMENT ';'
282 ;
283
284 ETSTryCatchStatement returns ETSStatement:
285   {ETSTryCatchStatement}
286   tryBlock=ETSTryBlock
287   (finallyBlock=ETSFinallyBlock |
288   (catchBlocks+=ETSCatchBlock)+
289   (finallyBlock=ETSFinallyBlock)? )
290 ;
291
292 ETSTryBlock:
293   'try' '{'
294     implementation=ETSImplementation
295   '}'
296 ;
297
298 ETSCatchBlock:
299   'catch' '(' declarators+=ETSDeclarator
300   (',' declarators+=ETSDeclarator)* ')' '{'
301     implementation=ETSImplementation
302   '}'
303 ;
304
305 ETSFinallyBlock:
306   'finally' '{'
307     implementation=ETSImplementation
308   '}'
309 ;
310
311 ETSIfThenElseStatement returns ETSStatement:
312   {ETSIfThenElseStatement}
313   ifBlock=ETSIfBlock
314   (=> elseIfBlocks+=ETSElseIfBlock)*
315   (=> elseBlock=ETSElseBlock)?
316 ;
317
318 ETSIfBlock:
319   {ETSIfBlock}
320   'if' '(' ifExpression=ETSExpression ')'
321   ('{' implementation=ETSImplementation '}' |
322   implementation=ETSSingleStatementImplementation)
```

```
323 ;
324
325 ETSElseIfBlock:
326   {ETSElseIfBlock}
327   'else' 'if' '(' elseIfExpression=ETSExpression ')'
328   ('{' implementation=ETSImplementation '}' |
329   implementation=ETSSingleStatementImplementation)
330 ;
331
332 ETSElseBlock:
333   {ETSElseBlock}
334   'else'
335   ('{' implementation=ETSImplementation '}' |
336   implementation=ETSSingleStatementImplementation)
337 ;
338
339 ETSSwitchCaseStatement returns ETSStatement:
340   {ETSSwitchCaseStatement}
341   'switch' '(' switchExpr=ETSExpression ')' '{'
342   (caseBlocks+=ETSSwitchCaseBlock)*
343   (withDefaultBlock?='default'
344   defaultBlock=ETSSwitchDefaultBlock)? '}'
345 ;
346
347 ETSSwitchCaseBlock:
348   {ETSSwitchCaseBlock}
349   'case' caseExprs+=ETSExpression
350   (',' caseExprs+=ETSExpression)* ':'
351   implementation=ETSImplementation
352 ;
353
354 ETSSwitchDefaultBlock:
355   {ETSSwitchDefaultBlock}
356   ':'
357   implementation=ETSImplementation
358 ;
359
360 /*
361  * -------------------------------------------------------
362  * ETSPec specific statements
363  * -------------------------------------------------------
364  */
365
366 ETSRunStatement returns ETSStatement:
367   {ETSRunStatement}
368   'run' ';'
369 ;
```

```
370
371 ETSRunUntilStatement returns ETSStatement:
372   {ETSRunUntilStatement}
373   'run' 'until' symbolReference=ETSSymbolReference ';'
374 ;
375
376 ETSStopStatement returns ETSStatement:
377   {ETSStopStatement}
378   'stop' ';'
379 ;
380
381 ETSStopOnStatement returns ETSStatement:
382   {ETSStopOnStatement}
383   'stop' 'at' symbolReference=ETSSymbolReference ';'
384 ;
385
386 ETSConcurrentStatement returns ETSStatement:
387   {ETSConcurrentStatement}
388   'concurrent' (conncurrentThread=ETSConcurrentThread)?
389   '{' implementation=ETSImplementation '}'
390 ;
391
392 ETSCancelStatement:
393   {ETSCancelStatement}
394   'cancel' concurrentThreads+=[ETSConcurrentThread|ID]
395   (',' concurrentThreads+=[ETSConcurrentThread|ID])*  ';'
396 ;
397
398 ETSWaitTimeStatement returns ETSStatement:
399   {ETSWaitTimeStatement}
400   'wait' (expression=ETSExpression)? ';'
401 ;
402
403 ETSTriggerStatement:
404   {ETSTriggerStatement}
405   'trigger' expression=ETSExpression ';'
406 ;
407
408 ETSWaitUntilStatement returns ETSStatement:
409   {ETSWaitUntilStatement}
410   'wait' 'until' expression=ETSExpression ';'
411 ;
412 ETSWaitDeltaStatement returns ETSStatement:
413   {ETSWaitDeltaStatement}
414   'wait' 'delta' ';'
415 ;
416
```

```
417 ETSReportStatement returns ETSStatement:
418   {ETSReportStatement}
419   'report' severity=ETS_ESEVERITY
420     (':' messageExpr=ETSExpression
421     (',' contextExpr=ETSExpression)?)? ';'
422 ;
423
424 ETSAssertStatement returns ETSStatement:
425   {ETSAssertStatement}
426   'assert' '(' expression=ETSExpression ')'
427     severity=ETS_ESEVERITY
428     (':' messageExpr=ETSExpression
429     (',' contextExpr=ETSExpression)?)? ';'
430 ;
431
432 ETSScopeStatement returns ETSStatement:
433   {ETSScopeStatement}
434   (
435     'scope' executor=ETSExecutor
436     '{'
437       implementation=ETSImplementation
438     '}'
439   )
440 ;
441
442 ETSRealtimeStatement returns ETSStatement:
443   {ETSRealtimeStatement}
444   (
445     realtimeBlock=ETSRealtimeBlock
446     (postRealtimeBlocks+=ETSPostRealtimeBlock)*
447   )
448 ;
449
450 ETSRealtimeBlock:
451   {ETSRealtimeBlock}
452   'realtime' (breaked?='until'
453     breakExpression=ETSExpression)?
454    (withTimeout?='timeout' timeoutExpression=ETSExpression)?
455    '{' (implementation=ETSImplementation) '}'
456 ;
457
458 ETSPostRealtimeBlock:
459   ETSAnalyzeBlock | ETSPlotBlock
460 ;
461
462 ETSAnalyzeBlock returns ETSPostRealtimeBlock:
463   {ETSAnalyzeBlock}
```

```
464   'analyze' '{' implementation=ETSImplementation '}'
465 ;
466
467 ETSPlotBlock returns ETSPostRealtimeBlock:
468   ETSTimePlotBlock
469 ;
470
471 ETSTimePlotBlock returns ETSPlotBlock:
472   {ETSTimePlotBlock}
473   'plot' name=ID (rangeExpr=ETSPlotRangeExpression)?
474   '{' (timePlotStatements+=ETSTimePlotStatement)* '}'
475 ;
476
477 ETSTimePlotStatement returns ETSStatement:
478   {ETSTimePlotStatement}
479   name=ID ':' plotExpr=ETSExpression
480   (rangeExpr=ETSPlotRangeExpression)? ';'
481 ;
482
483 ETSPlotRangeExpression:
484   {ETSPlotRangeExpression}
485   'range'
486     ('[' startOffset=ETSExpression ','
487        endOffset=ETSExpression ']')?
488     (naturalLog?='ln' | decimalLog?='lg')?
489     (sampleHold?='hold' | dots?='dots')?
490 ;
491
492 /*
493  * -------------------------------------------------------
494  * Expressions
495  * -------------------------------------------------------
496  */
497
498 ETSExpression:
499   {ETSExpression}
500      (
501        instantiation?='new' valueSymbol=ETSJavaReference
502        call?='(' callParameterList=ETSCallParameterList ')'
503      ) |
504   (
505     (
506       (prefixOperator=ETSEWrapPrefixOperator) |
507       (withCast?='<' castDataType=ETSDataType '>')
508     )?
509     (
510       ( bracketed?='(' innerExpression=ETSExpression ')' ) |
```

```
511       ( valueSymbol=ETSValueSymbol )
512     )
513     (
514       call?='(' callParameterList=ETSCallParameterList ')'
515     )?
516     (operationExpression=ETSOperationExpression)?
517   )
518 ;
519
520 ETSOperationExpression:
521   {ETSOperationExpression}
522   op=ETS_EOPERATOR rhs=ETSExpression
523 ;
524
525 ETSCallParameterList:
526   {ETSCallParameterList}
527   (expressions+=ETSExpression (','
528     expressions+=ETSExpression)*)?
529 ;
530
531 /*
532  * -------------------------------------------------------
533  * Value Symbol
534  * -------------------------------------------------------
535  */
536
537 ETSValueSymbol:
538   ETSConstant | ETSRuntimeInstance |
539   ETSSymbolReference | ETSJavaReference
540 ;
541
542 ETSConstant returns ETSValueSymbol:
543   ETSNumberConstant | ETSStringConstant | ETSSpecialConstant
544 ;
545
546 ETSNumberConstant returns ETSConstant:
547   {ETSNumberConstant}
548   value=NUMBER (=> unit=[ETSUnitDeclaration|UNITNAME])?
549 ;
550
551 ETSStringConstant returns ETSConstant:
552   {ETSStringConstant}
553   value=STRING
554 ;
555
556 ETSSpecialConstant returns ETSConstant:
557   {ETSSpecialConstant}
```

```
558   value=ETS_ESPECIALCONSTANT
559 ;
560
561 ETSRuntimeInstance returns ETSValueSymbol:
562   {ETSRuntimeInstance}
563   'Runtime' ':' property=ETS_ERUNTIMEPROPERTY
564 ;
565
566 ETSSymbolReference returns ETSValueSymbol:
567   {ETSSymbolReference}
568   (
569     (
570       (referenceAccess?='ref')?
571     )
572     baseSymbol=[BaseElements::ITestReferenceable|ID]
573     (subSymbols+=ETSSubSymbolReference)*
574     (
575       (propertyAccess?=':' property=ETSEWrapProperty) |
576       (eventAccess?='´' event=ETSEWrapEvent) |
577       (arrayAccess?=
578         '[' arrayAccessExpression=ETSExpression ']')
579     )?
580   )
581 ;
582
583 ETSSubSymbolReference:
584   {ETSSubSymbolReference}
585   '.' actualSymbol=ETSActualSymbol
586 ;
587
588 ETSActualSymbol:
589   {ETSActualSymbol}
590   testReferenceable=[Ecore::EObject|ID]
591 ;
592
593 /*
594  * -------------------------------------------------------
595  * Internal Declarator
596  * -------------------------------------------------------
597  */
598
599 ETSDeclarator:
600   {ETSDeclarator}
601   (dataType=ETSDataType | javaType=ETSJavaReference)
602   declaration=ETSTestVariableDeclaration
603 ;
604
```

```
/*
 * ------------------------------------------------------------
 * Internal Data Types
 * ------------------------------------------------------------
 */

ETSDataType:
  (
    integralType=ETS_EINTEGRALDATATYPE |
    (physicalType ?= 'physical'
    '[' (unit=[ETSUnitDeclaration|UNITNAME])? ']')
  ) (array?='[' arraySizeExpression=ETSExpression ']')?
;

ETSJavaReference returns ETSValueSymbol:
  {ETSJavaReference}
  'Java' '.'
    baseJElement=[JavaTypes::JvmIdentifiableElement|ID]
  (subJElements+=ETSSubJavaReference)*
;

ETSSubJavaReference:
  {ETSSubJavaReference}
  '.' actualJElement=ETSActualJElement
;

ETSActualJElement:
  {ETSActualJElement}
  jElement=[JavaTypes::JvmIdentifiableElement|ID]
;

/*
 * ------------------------------------------------------------
 * Internal Referenceables
 * ------------------------------------------------------------
 */

ETSTestVariableDeclaration
  returns BaseElements::ITestReferenceable:
  {ETSTestVariableDeclaration}
  name=ID
;

ETSPackageVariableDeclaration
  returns BaseElements::ITestReferenceable:
  {ETSPackageVariableDeclaration}
  name=ID
```

```
;

ETSPackageFunctionDeclaration
  returns BaseElements::ITestCallable:
  {ETSPackageFunctionDeclaration}
  name=ID
;

ETSPackageFuncParameterDeclaration
  returns BaseElements::ITestReferenceable:
  {ETSPackageFuncParameterDeclaration}
  name=ID
;

ETSConcurrentThread:
  {ETSConcurrentThread}
  name=ID
;

/*
 * -------------------------------------------------------
 * Terminals
 * -------------------------------------------------------
 */

FULLQUALIFIEDNAME returns Ecore::EString:
  ID ('.' ID)*
;

NUMBER returns Ecore::EString:
  ('-'? INT ('.' INT)?) |
  (HEXINT)
;

terminal HEXINT:
  '0x' (('0'..'9') | ('a'..'f') | ('A'..'F'))*
;

terminal UNITPOWER:
  '¹'|'²'|'³'|'⁴'|'⁵'|'⁶'|'⁷'|'⁸'|'⁹'|'⁰'
;

UNITNAME returns Ecore::EString:
     ID
     ('⁻' UNITPOWER)? (UNITPOWER)*
     (=> '·' UNITNAME)*
;
```

```
699
700 /*
701  * ------------------------------------------------------------
702  * Enum Wrappers
703  * ------------------------------------------------------------
704  */
705
706 ETSEWrapPrefixOperator:
707   {ETSEWrapPrefixOperator}
708   evalue=ETS_EPREFIXOPERATOR
709 ;
710
711 ETSEWrapEvent:
712   {ETSEWrapEvent}
713   evalue=ETS_EEVENT
714 ;
715
716 ETSEWrapProperty:
717   {ETSEWrapProperty}
718   evalue=ETS_EPROPERTY
719 ;
720
721 /*
722  * ------------------------------------------------------------
723  * Enums
724  * ------------------------------------------------------------
725  */
726
727 enum ETS_EPREFIXOPERATOR:
728   Invert='!'
729 ;
730
731 enum ETS_EOPERATOR:
732   Assign='=' |
733   AssignAdd='+=' | AssignSubstract='-=' |
734   AssignMultiply='*=' | AssignDivide='/=' | Lower='<' |
735   LowerEqual='<=' | Equal='==' | GreaterEqual='>=' |
736   Greater='>' | UnEqual='!=' | Add='+' | Substract='-' |
737   Multiply='*' | Divide='/' | Modulo='%' | Or='|' |
738   And='&' | Xor='^' | ShiftLeft='<<' | ShiftRight='>>' |
739   ShiftRightZero='>>>' | InstanceOf='is' |
740   BoolOr='||' | BoolAnd='&&'
741 ;
742
743 enum ETS_EUNITOPERATOR:
744   Add='+' | Substract='-' | Multiply='*' | Divide='/'
745 ;
```

```
746
747 enum ETS_ELOOPCTRLSTATEMENT:
748   Break='break' | Continue='continue'
749 ;
750
751 enum ETS_ESPECIALCONSTANT:
752   True='true' | False='false' | Null='null'
753 ;
754
755 enum ETS_EINTEGRALDATATYPE:
756   Void='void' | Bool='bool' | Int8='int8' | Int16='int16' |
757   Int32='int32' | Int64='int64' | Float='float' |
758   Double='double' | String='string' | Event='event'
759 ;
760
761 enum ETS_EEVENT:
762   Read='read' | Written='written' | Called='called' |
763   Returned='returned' | Received='received' | Sent ='sent'
764 ;
765
766 enum ETS_ESEVERITY:
767   Info='info' | Warning='warning' |
768   Error='error' | Fatal='fatal'
769 ;
770
771 enum ETS_EPROPERTY:
772   InstructionPointer='InstructionPointer' |
773   Address='Address' |
774   TriggerRising='TriggerRising' |
775   TriggerFalling='TriggerFalling' |
776   TriggerAbove='TriggerAbove' |
777   TriggerBelow='TriggerBelow' |
778   SampleRate='SampleRate' |
779   Count='Count'
780 ;
781
782 enum ETS_ERUNTIMEPROPERTY:
783   GlobalTime='GlobalTime' | Timestamp='Timestamp' |
784   EndOfTrace='EOT'
785 ;
```

C

QUELLTEXT VERWENDETER ETSPEC PAKETE

Der im Folgenden aufgeführte ETSpec Quelltext wird im Rahmen der Fallstudie in Abschnitt 8.1 verwendet.

```
1  package Units
2  {
3      unit ms;
4      unit s   [ms  = s * 1000];
5      unit us  [ms  = us / 1000];
6      unit min [s   = 60 * min];
7      unit h   [min = 60 * h];
8
9      unit min^-1;
10
11     unit V;
12 }
13
14 package Common : MPC5643L
15 {
16     // Hilfsfunktion zum Starten des Mikrocontrollers.
17     // Kerne halten bei jeweiliger Hauptroutine 'run_app'.
18     func void bootToMain()
19     {
20         break at Core1.run_app;
21         run until Core0.run_app;
22     }
23 }
24
25 package SignalAnalyzer
26 {
27     // Finden der nächsten steigenden Flanke des Signals.
28     analyze func physical[s] findRisingEdge(
29         ref physical[V] signal,
30         physical[s] timeOffset,
31         physical[V] valueDelta)
32     {
33         wait timeOffset;
34         physical[V] baseValue = signal;
35
36         while(!Runtime:EOT)
37         {
38             wait until signal´written;
```

```
39              if (signal > baseValue + valueDelta)
40              {
41                  return Runtime:Timestamp;
42              }
43          }
44
45          return 0s;
46      }
47
48      // Finden der nächsten fallenden Flanke des Signals.
49      analyze func physical[s] findFallingEdge(
50          ref physical[V] signal,
51          physical[s] timeOffset,
52          physical[V] valueDelta)
53      {
54          wait timeOffset;
55          physical[V] baseValue = signal;
56
57          while(!Runtime:EOT)
58          {
59              wait until signal´written;
60              if (signal < baseValue - valueDelta)
61              {
62                  return Runtime:Timestamp;
63              }
64          }
65
66          return 0s;
67      }
68
69      // Extrahiert maximalen Signalswert.
70      analyze func float getMaximumAfter(
71          ref float signal,
72          physical[s] timeOffset)
73      {
74          wait timeOffset;
75          float maxValue = signal;
76
77          while (!Runtime:EOT)
78          {
79              wait until signal´written;
80              if (signal > maxValue) maxValue = signal;
81          }
82
83          return maxValue;
84      }
85
```

```
86      // Extrahiert minimalen Signalswert.
87      analyze func float getMinimumAfter(
88          ref float signal,
89          physical[s] timeOffset)
90      {
91          wait timeOffset;
92          float minValue = signal;
93
94          while (!Runtime:EOT)
95          {
96              wait until signal´written;
97              if (signal < minValue) minValue = signal;
98          }
99
100         return minValue;
101     }
102 }
```

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

ABKÜRZUNGSVERZEICHNIS

API Application Programming Interface

AST Abstract Syntax Tree

BNF Backus-Naur-Form

DSL Domain Specific Language

ECU Embedded Control Unit

EMF Eclipe Modeling Framework

GPL General Purpose Language

GUID Globally Unique Identifier

HiL Hardware-in-the-Loop

IDE Integrated Development Environment

ISTQB International Software Testing Qualifications Board

MBT Model-Based Testing

MDE Model Driven Engineering

MDSD Model Driven Software Development

MiL Model-in-the-Loop

MOF Meta Object Facility

OEM Original Equipment Manufacturer; in der Automobilindustrie der Fahrzeughersteller

OMG Object Management Group

PiL Processor-in-the-Loop

PTP Precision Time Protocol

RPC Remote Procedure Call

SiL Software-in-the-Loop

TQC Test Quantity Container; Testgrößen beschreibendes Modell

TRM Test Rig Model

UI User Interface

UML Unified Modeling Language

XCP Universal Measurement and Calibration Protocol

XMI XML Metadata Interchange

XML eXtensible Markup Language

XSD XML Schema Definition

FQN Fully Qualified Name

LITERATURVERZEICHNIS

[ARM09] ARM Holdings plc. *CoreSight Components Technical Reference Manual*. Juli 2009.

[BBB+01] Kent Beck u. a. *Manifesto for Agile Software Development*. Online. 2001. URL: http://www.agilemanifesto.org/.

[BDI06] Stefan Blom u. a. „TTCN-3 for Distributed Testing Embedded Software“. In: *6th International Andrei Ershov Memorial Conference, PSI 2006*. Springer, Juni 2006.

[Bec03] Kent Beck. *Test Driven Development By Example*. Addison-Wesley Professional, 2003.

[Bet13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

[BGK+11] Christian Brand u. a. „Development of High-Quality Graphical Model Editors“. In: *Eclipse Magazine* (2011).

[BH13] Hans-Joachim Böckenhauer. *Formale Sprachen: Endliche Automaten, Grammatiken, lexikalische und syntaktische Analyse*. Hrsg. von Juraj Hromkovič. Springer Vieweg, 2013.

[BIV05] Stefan Blom u. a. „Simulated time for testing railway interlockings with TTCN-3“. In: *5th International Workshop, FATES 2005*. Springer, Juli 2005.

[Chr08] James Christie. „The Seductive and Dangerous V-Model“. In: *Testing Experience* (2008), S. 73–77.

[Deu14] Fabian Deuchler. *Aufbereitung und Analyse aufgezeichneter Traces eingebetteter Software*. Karlsruher Institut für Technologie, Masterarbeit, 2014.

[Dow97] Mark Dowson. „The Ariane 5 Software Failure“. In: *SIGSOFT Softw. Eng. Notes* 22.2 (März 1997).

[EEK+12] Sven Efftinge u. a. „Xbase: Implementing Domain-specific Languages for Java“. In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, März 2012, S. 112–121.

[EV06] Sven Efftinge und Markus Völter. „oAW xText: A framework for textual DSLs“. In: *Workshop on Modeling Symposium at Eclipse Summit*. Bd. 32. 2006, S. 118.

[Fow09] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. Online. 2004. URL: http://martinfowler.com/articles/injection.html.

[GFK+08] Juergen Grossmann u. a. „TestML - A Test Exchange Language for Model-Based Testing of Embedded Software“. In: *Model-Driven Development of Reliable Automotive Services*. Springer, 2008, S. 98–117.

[Gre11] James W. Grenning. *Test-Driven Development for Embedded C*. Pragmatic Bookshelf, 2011.

[Gru13] Stephan Grünfelder. *Software-Test für Embedded Systems: Ein Praxishandbuch für Entwickler, Tester und technische Projektleiter*. dpunkt.verlag GmbH, 2013.

[GSS09] Ina Schieferdecker und Jürgen Großmann. „Testing Embedded Control Systems with TTCN-3“. In: *Software Technologies for Embedded and Ubiquitous Systems*. Springer, 2007, S. 125–136.

[Har01] Nico Hartmann. *Automation des Tests eingebetteter Systeme am Beispiel der Kraftfahrzeugelektronik*. Universität Karlsruhe (TH), Dissertation, 2001.

[Hen04] Stefan Hendrata. „Standardisiertes Testen mit TTCN-3: Erhöhung der Zuverlässigkeit von Software-Systemen im Fahrzeug“. In: *Hanser Automotive: Electronics+Systems* 9-10 (2004), S. 64–65.

[Hen14] Melda Henden. *Konzeptionierung eines protokollübergreifenden Datenformats zur Beschreibung von Signaldaten*. Karlsruher Institut für Technologie, Masterarbeit, 2014.

[Her15] Dominic Herity. *Modern C++ in embedded systems – Part 1: Myth and Reality*. Jan. 2015. URL: http://www.embedded.com/design/programming-languages-and-tools/4438660/Modern-C--in-embedded-systems---Part-1--Myth-and-Reality.

[Hro11] Juraj Hromkovič. *Berechenbarkeit: Logik, Argumentation, Rechner und Assembler, Unendlichkeit, Grenzen der Automatisierbarkeit*. Vieweg+Teubner Verlag, 2011.

[IEE04] IEEE Industry Standards and Technology Organization. *Nexus Standard Brings Order to Microprocessor Debugging*. 2004.

[ISTQB11] International Software Testing Qualifications Board. *Certified Tester Foundation Level Syllabus*. 2011.

[Kee00] Hugh O'Keeffe. *The Nexus 5001 Forum™ Standard providing the Gateway to the Embedded Systems of the Future, Version 1.1*. Jan. 2000.

[KF09] Olaf Kindel und Mario Friedrich. *Softwareentwicklung mit AUTOSAR : Grundlagen, Engineering, Management in der Praxis*. dpunkt-Verlag, 2009.

[Kle14] Sebastian Klenk. *Entwurf und Realisierung eines Demonstrators sowie einer Elektromotor-Regelung auf Basis eines Mehrkern-Controllers*. Karlsruher Institut für Technologie, Masterarbeit, 2014.

[KLM+97] Gregor Kiczales u. a. „Aspect-oriented programming". In: *European conference on object-oriented programming*. Springer. 1997, S. 220–242.

[Koe14] Christian Köllner. *Transformation von Multiphysics-Modellen in einen FPGA-Entwurf für den echtzeitfähigen HiL-Test eingebetteter Systeme*. Karlsruher Institut für Technologie (KIT), KIT Scientific Publishing, Dissertation, 2014.

[Kro16] Eduard Krolacsek. *Realisierung einer Standard Softwarearchitektur für eingebettete Steuergeräte unter Berücksichtigung von AUTOSAR*. Karlsruher Institut für Technologie, Masterarbeit, 2016.

[KT11] Karsten Thoms. *Spray – a quick way to create Graphiti*. Online. 2011. URL: https://kthoms.wordpress.com/2011/05/30/spray-a-quick-way-to-create-graphiti/.

[Lau15a] Lauterbach GmbH. *PRACTICE Script Language Reference Guide*. Nov. 2015.

[Lau15b] Lauterbach GmbH. *PRACTICE Script Language User's Guide*. Nov. 2015.

[Lev93] Nancy Leveson und Clark Turner. „An investigation of the Therac-25 accidents". In: *IEEE Computer* 26.7 (Juli 1993).

[LH02] Gabriel Leen und Donal Heffernan. „Expanding automotive electronic systems". In: *IEEE Computer* 35.1 (2002), S. 88–93.

[Li99] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[Lig09] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. 2. Aufl. Spektrum Akademischer Verlag, 2009.

[LM14] Marc Lobmeyer und Roman Marktl. *Steuergerätetests effizienter programmieren - Basics, Tipps und Tricks beim Einsatz von CAPL*. Apr. 2014.

[Mar10] John C. Martin. *Introduction to Languages and the Theory of Computation*. 4. Aufl. McGraw Hill Higher Education, 2010.

[Mar14] Remo Markgraf. „Agile Entwicklung von Embedded-Systemen". In: *Elektronik Praxis* (2014).

[MBT14] MBtech Group GmbH & Co. KGaA. *PROVEtech:TA – Testsprache – Version: 2015*. Juli 2014.

[MLe99] Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation*. 3. Aufl. Oxford University Press, 1999.

[MS12] Microsoft Corporation. *C# Language Specification*. 2012.

[OB88] Thomas J. Ostrand und Marc J. Balcer. „The category-partition method for specifying and generating fuctional tests". In: *Communications of the ACM* 31.6 (1988).

[OMG15] Object Management Group. *Meta Object Facility (MOF) Core Specification*. Juni 2015.

[OMG15b] Object Management Group. *XML Metadata Interchange (XMI) Specification*. Juni 2015.

[Onl/Auto] AUTOSAR development cooperation. *AUTOSAR technical overview*. Online. Feb. 2016. URL: http://www.autosar.org/about/technical-overview/.

[Onl/DBC] Vector Informatik. *DBC-Kommunikations-Datenbasis für CAN*. Online. 2016. URL: http://vector.com/vi_candb_de.html.

[Onl/GG] Google. *Google Guice on Github*. Online. 2016. URL: https://github.com/google/guice.

[Onl/GMP] Eclipse Foundation. *Graphical Modeling Project (GMP)*. Online. 2016. URL: http://www.eclipse.org/modeling/gmp/.

[Onl/Hir] Hirschmann. *White Paper – Precision Clock Synchronization – The Standard IEEE 1588*. Online. URL: http://www.pdv.reutlingen-university.de/rte/White_paper_ieee1588_de_v1-2.pdf.

[Onl/iSys] iSYSTEM AG. *isystem.connect API*. Online. Feb. 2016. URL: http://www.isystem.com/downloads/winIDEA/SDK/iSYSTEM.Python.SDK/documentation/isystem-connect-api.

[Onl/JDoc] Oracle Corporation. *Java Platform, Standard Edition 8, API Specification*. Online. 2016. URL: http://docs.oracle.com/javase/8/docs/api/overview-summary.html.

[Onl/JET] Eclipse Consortium u. a. *Java Emitter Templates (JET)*. Online. 2016. URL: https://eclipse.org/modeling/m2t/?project=jet.

[Onl/MSDK] Microsoft Developer Network (MSDN). *Modeling SDK for Visual Studio - Domain-Specific Languages*. Online. 2016. URL: https://msdn.microsoft.com/en-us/library/bb126259.aspx.

[Onl/Spray] *Spray auf Google Code*. Online. 2016. URL: https://code.google.com/archive/a/eclipselabs.org/p/spray.

[Onl/Tesla] The Guardian. *Tesla driver dies in first fatal crash while using autopilot mode*. Online. 2016. URL: https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-death-self-driving-car-elon-musk.

[Onl/VBA] Microsoft Developer Network (MSDN). *Office VBA language reference*. Online. Feb. 2016. URL: https://msdn.microsoft.com/en-us/library/office/gg264383.aspx.

[Onl/VMXT] Der Beauftragte der Bundesregierung für Informationstechnik. *V-Modell XT*. Online. 2016. URL: http://www.v-modell-xt.de.

[Onl/WWB] Polar Engineering, Inc. *WinWrap Basic Language Reference*. Online. Feb. 2016. URL: https://www.winwrap.com/web2/basic/.

[Onl/XtG] *Xtext Documentation – The Grammar Language*. Online. 2016. URL: https://eclipse.org/Xtext/documentation/301_grammarlanguage.html.

[Onl/xygraph] *swt-xy-graph auf Google Code*. Online. 2016. URL: https://code.google.com/archive/p/swt-xy-graph.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. O'Reilly, 2013.

[PKS02] Martin Pol, Tim Koomen und Andreas Spillner. *Management und Optimierung des Testprozesses: Praktischer Leitfaden für erfolgreiches Software-Testen mit TPI und TMap*. 2. Aufl. dpunkt Verlag, 2002.

[Pri14] Daniel Prill. *Modellbasierter Entwurf eines grafischen Konfigurationswerkzeugs*. Karlsruher Institut für Technologie, Masterarbeit, 2014.

[RBG+10] Thomas Roßner u. a. *Basiswissen Modellbasierter Test*. dpunkt Verlag, 2010.

[RD12] Guido van Rossum und Fred L. Drake. *The Python Language Reference, Release 3.2.3*. Juni 2012.

[Rit93] Dennis M. Ritchie. *The Development of the C Language*. Online. Apr. 1993. URL: http://csapp.cs.cmu.edu/3e/docs/chistory.html.

[Roy70] Winston Royce. „Managing the Development of Large Software Systems". In: *proceedings of IEEE WESCON*. 1970.

[SA08] Eric Sax und Stefan Abendroth. „Testing Automotive System Prototypes Far before Driving on the Proving Ground“. In: *The 19th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP)*. Juni 2008, S. xix–xix.

[Sax08] Eric Sax. *Automatisiertes Testen Eingebetteter Systeme in der Automobilindustrie*. Carl Hanser Verlag GmbH, 2008.

[SBG05] Ina Schieferdecker, Eckard Bringmann und Jürgen Großmann. „Continuous TTCN-3: Testing of Embedded Control Systems“. In: *Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems*. ACM, 2006, S. 29–36.

[Sch06] Douglas C Schmidt. „Model-Driven Engineering“. In: *IEEE Computer* 39.2 (2006).

[Sch09] Markus Schorn. „Using CDT APIs to programmatically introspect C/C++ code“. In: *EclipseCon 2009*. März 2009.

[SFL+13] Eric Sommerlade u. a. *CppUnit Documentation, Version 1.13.2*. Nov. 2015. URL: http://people.freedesktop.org/~mmohrhard/cppunit/index.html.

[SG07] Peter Sommerlad und Emanuel Graf. „Cute: C++ unit testing easier“. In: *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 2007.

[Sip12] Michael Sipser. *Introduction to the Theory of Computation*. 3. Aufl. Cengage Learning, Inc, 2012.

[SS05] Philipp Schill und Ralf Schmauer. „Codegenerierung mit dem Eclipse Modeling Framework“. In: *OBJEKTspektrum* (Jan. 2005).

[Std/AC10] EN 4660. *Aerospace series - Modular and Open Avionics Architectures*. Allied Standard Avionics Architecture Council. März 2010.

[Std/AR13] ARINC 653. *Avionics Application Standard Software Interface - Part 0 - Introduction to ARINC 653*. Aeronautical Radio Incorporated. Juni 2013.

[Std/ASA09] ASAM AE HIL V1.0.0. *Application Programming Interface for ECU Testing via Hardware-in-the-Loop Simulation*. Association for Standardisation of Automation and Measuring Systems. Juni 2009.

[Std/ASA11a] ASAM AE MCD-3 MC V3.0.0. *Application Programming Interface for Measurement and Calibration Server*. Association for Standardisation of Automation and Measuring Systems. Sep. 2011.

[Std/ASA11b] ASAM AE MCD-3 D V3.0.0. *Application Programming Interface for MVCI Diagnostic Server*. Association for Standardisation of Automation and Measuring Systems. Okt. 2011.

[Std/ASA12] ASAM AE ATX V1.0.0. *Automotive Test Exchange Format*. Association for Standardisation of Automation and Measuring Systems. März 2012.

[Std/ASA14] ASAM AE MCD-2 NET V4.1.1 (FIBEX). *Field Bus Data Exchange Format*. Association for Standardisation of Automation and Measuring Systems. Sep. 2014.

[Std/ASA15a] ASAM MCD-1 XCP V1.3.0. *The Universal Measurement and Calibration Protocol Family*. Association for Standardisation of Automation and Measuring Systems. Apr. 2015.

[Std/ASA15b] ASAM AE XIL V2.0.2. *Generic Simulator Interface*. Association for Standardisation of Automation and Measuring Systems. Sep. 2015.

[Std/C11] ISO/IEC 9899:2011. *Information technology – Programming languages – C*. International Organization for Standardization und International Electrotechnical Commission.

[Std/C99] ISO/IEC9899:1999. *Information technology – Programming languages – C*. International Organization for Standardization und International Electrotechnical Commission.

[Std/CPP14] ISO/IEC 14882:2014. *Information technology – Programming languages – C++*. International Organization for Standardization und International Electrotechnical Commission.

[Std/ETS07a] ETSI ES 201 873-2 V3.2.1. *The Testing and Test Control Notation Version 3; Part 2: Tabular Presentation Format*. Methods for Testing and Specification (MTS). Feb. 2007.

[Std/ETS07b] ETSI ES 201 873-3 V3.2.1. *The Testing and Test Control Notation Version 3; Part 3: Graphical Presentation Format*. Methods for Testing and Specification (MTS). Feb. 2007.

[Std/ETS15] ETSI ES 201 873-1 V4.7.1. *The Testing and Test Control Notation Version 3; Part 1: Core Language*. Methods for Testing and Specification (MTS). Juni 2015.

[Std/ETS15b] ETSI ES 201 873-5 V4.7.1. *The Testing and Test Control Notation Version 3; Part 5: TTCN-3 Runtime Interface (TRI)*. Methods for Testing and Specification (MTS). Juni 2015.

[Std/ETS15c] ETSI ES 201 873-6 V4.7.1. *The Testing and Test Control Notation Version 3; Part 6: TTCN-3 Control Interface (TCI)*. Methods for Testing and Specification (MTS). Juni 2015.

[Std/IEC06] IEC 62304:2006. *Medical device software – Software lifecycle processes*. International Electrotechnical Commission.

[Std/IEC10] IEC 61508:2010. *Functional safety of electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission.

[Std/IEC10b] IEC 62541:2010. *OPC Unified Architecture - Part 1: Overview and Concepts*. International Electrotechnical Commission.

[Std/IEE08] IEEE 1588:2008. *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. Institute of Electrical and Electronics Engineers.

[Std/IEE12] IEEE-ISTO 5001:2012. *The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface*. IEEE Industry Standards and Technology Organization.

[Std/IEE13] IEEE 1149.1:2013. *IEEE Standard for Test Access Port and Boundary-Scan Architecture*. Institute of Electrical and Electronics Engineers.

[Std/IEE754] ANSI/IEEE Std 754-2008. *IEEE Standard for Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers und American National Standards Institute. Aug. 2008.

[Std/ISO05] ISO 17356-3:2005. *Road vehicles – Open interface for embedded automotive applications – Part 3: OSEK/VDX Operating System (OS)*. International Organization for Standardization.

[Std/ISO11a] ISO 26262-6:2011. *Road vehicles – Functional safety – Part 6: Product development at the software level*. International Organization for Standardization.

[Std/ITU11] ITU-T Z.120. *Message Sequence Chart (MSC)*. International Telecommunication Union, Telecommunication Standardization Sector (ITU-T). Feb. 2011.

[Std/ITU11b] ITU-T X.680ff. *Abstract Syntax Notation One (ASN.1)*. International Telecommunication Union, Telecommunication Standardization Sector (ITU-T). Aug. 2011.

[Std/RTC12] DO-178C. *Software Considerations in Airborne Systems and Equipment Certification*. Radio Technical Commission for Aeronautics und European Organisation for Civil Aviation Equipment. Jan. 2012.

[Std/VDE12] EN 50128:2012. *Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme – Software für Eisenbahnsteuerungs- und Überwachungssysteme*. Verband der Elektrotechnik, Elektronik und Informationstechnik.

[Ste09] Dave Steinberg. *EMF - Eclipse modeling framework*. 2. Aufl. Addison-Wesley, 2009.

[SVE+07] Thomas Stahl u. a. *Modellgetriebene Softwareentwicklung*. 2. Aufl. dpunkt.verlag GmbH, 2007.

[Vec04] Vector CANtech, Inc. *Programming with CAPL*. Dez. 2004.

[VO13] VersionOne Inc. *7th Annual State of Agile Survey*. 2013.

[VO15] VersionOne Inc. *9th Annual State of Agile Survey*. 2015.

[VRK+12] Markus Voelter u. a. „mbeddr: an extensible C-based programming language and IDE for embedded systems“. In: *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM, 2012.

[WFS+12] Michael Wahler u. a. „CAST: Automating Software Tests for Embedded Systems“. In: *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Apr. 2012, S. 457–466.

[Wiki/Exp] Wikipedia. *Expression (computer science)*. Online. Apr. 2016. URL: https://en.wikipedia.org/w/index.php?title=Expression_%28computer_science%29&oldid=712921965.

[Wiki/Scr] Wikipedia. *Scrum*. Online. Aug. 2015. URL: https://de.wikipedia.org/w/index.php?title=Scrum&oldid=145273161.

[Wiki/TV] Wikipedia. *Turing-Vollständigkeit*. Online. März 2016. URL: https://de.wikipedia.org/w/index.php?title=Turing-Vollst%C3%A4ndigkeit&oldid=149388428.

[Wil05] C. Willcock u. a. *An Introduction to TTCN-3*. Wiley, 2005.

[Wit10] Michael Wittner. *CCDL Whitepaper*. Dez. 2010.

Eingebettete Systeme unterliegen Anforderungen, die einen intensiven Test der Software erforderlich machen. Zur Automatisierung werden dabei verschiedene Technologien eingesetzt: Für den quelltextnahen Test werden spezielle Unit-Test Werkzeuge verwendet, der Test von Steuer- und Regelfunktionen erfordert Model-, Software- und Hardware-in-the-Loop Werkzeuge. In dieser Arbeit wird deren Zusammenspiel untersucht. Der Fokus liegt dabei auf einer neuen Programmiersprache zur Implementierung von Testfällen.

ISBN 978-3-7315-0663-8

9 783731 506638 >

Gedruckt auf FSC-zertifiziertem Papier