Analyse und Optimierung von Testabläufen während der Entwicklung einer Perzeptionsplattform

Bachelorarbeit

vorgelegt am 5. Dezember 2012

am Institut FORWISS der Universität Passau

Name:	Lukas Elsner
Matrikelnummer:	55344
Studiengang:	Informatik
Prüfer:	Prof. Dr. Tomas Sauer
Betreuer :	Dipl.-Inf. Sebastian Pangerl

Zusammenfassung

Im Rahmen von interactIVe wird eine Perzeptionsplattform entwickelt, welche wesentlicher Bestandteil und Voraussetzung der nächsten Generation von Sicherheitssystemen in Kraftfahrzeugen ist. Durch die Installation der Plattform in mehreren Versuchsfahrzeugen sind ausgiebige Tests von Funktionalität, Korrektheit und Existenz der erforderlichen Daten zwingend notwendig. Die technische Basis der Plattform ist das Automotive Data and Time-triggered Framework ADTF der Firma elektrobit. Dieses erlaubt eine dynamische Steuerung nicht nur über die grafische Oberfläche, sondern auch über das mitgelieferte Software Development Kit. Bisher gibt es keine Möglichkeit, das Testen der Plattform zumindest teilweise zu automatisieren. Bisherige Tests werden ausschließlich visuell über das GUI von ADTF durchgeführt. Mit dieser Arbeit ist eine einfach zu bedienende Klassenbibliothek auf dem ADTF SDK entstanden, die den Testansprüchen des interactIVe Projekts genügt und seine Arbeit weitgehend vereinfacht.

Inhaltsverzeichnis

1 Einleitung **5**
1.1 Perzeption . . . 5
1.2 interactIVe . . . 8
1.3 Framework . . . 9
1.4 ADTF . . . 10
1.5 Rechtliche Situation . . . 12

2 Motivation **13**
2.1 Software Testen . . . 13
2.1.1 Verifikation und Validierung . . . 15
2.1.2 Integrationstests . . . 16
2.1.3 Auslieferungstests . . . 16
2.1.4 Leistungstests . . . 16
2.1.5 Schnittstellentests . . . 17
2.2 Testen des Perzeptionssystems . . . 17
2.2.1 Ist-Situation . . . 17
2.2.2 Testen auf Anwendungsebene . . . 20

3 Rahmenbedingungen **23**
3.1 Fehlende Referenzdaten . . . 23
3.2 ADTF SDK . . . 23
3.3 ADTF Konfigurationen . . . 23
3.4 Standard Ausgabe . . . 23
3.5 Testen ohne Quelltext . . . 23
3.6 Werkzeuge . . . 24

4 Implementierung **25**
4.1 Klassenbibliothek . . . 27
4.1.1 Interfaces . . . 27
4.1.2 Klassen . . . 29
4.2 Beispielhafte Anwendung . . . 32
4.2.1 ITestcase und IPintrigger Implementierung . . . 33
4.2.2 IDebugFilter Implementierung . . . 34
4.2.3 Der Test . . . 35
4.3 Mögliche Anwendung in interactIVe . . . 36
4.3.1 Speicherleck in einem Filter . . . 36
4.3.2 Ausfall von Sensoren . . . 36
4.3.3 Qualitätscheck . . . 37

5 Ausblick **39**
5.1 Automatisierung 39
5.2 Fehlerbehandlung 39
5.3 Grafische Testergebnisse 39
5.4 Konfigurationsmanagement 39
5.5 Testsuite GUI 39
5.6 Vordefinierte Testfälle 40

Glossar **41**

Quellenverzeichnis **43**

Abbildungsverzeichnis **44**

Eidesstattliche Erklärung **45**

1 Einleitung

Durch den technischen Fortschritt decken Automobile aufgrund intelligent konzipierter Software ein immer breiter werdendes Spektrum an Funktionalität ab. So genannte elektronische Fahrerassistenzsysteme erhöhen im Straßenverkehr Sicherheit, Effizienz und Komfort. Schon Ende der achtziger Jahre befanden sich die ersten dieser Systeme in Form von automatischen Blockierverhinderern in Serienfahrzeugen. Heutzutage kommt kein Auto mehr ohne eine Vielzahl dieser Helfer aus. Antiblockiersystem (ABS), Antriebsschlupfregelung (ASR), Elektronisches Stabilitätsprogramm (ESP), Elektronische Differentialsperre (EDS), Lichtautomatik und Tempomat sind nur wenige Beispiele aus dem breit gefächerten Angebot, welches die Automobilindustrie zu bieten hat. Ein Beispiel für ein Forschungsprojekt, welches sich mit allen Bereichen beschäftigt, ist FAMOS, das sich auf die Entwicklung der Fahrerassistenzsysteme Unfallstellenschwerpunktwarnung, Green Driving und Einfädelassistent konzentriert [5]. Bei Fahrerassistenzsystemen wird zwischen aktiven und passiven Systemen unterschieden. Während die passiven Systeme akustisch, visuell oder haptisch mit dem Fahrer kommunizieren, greifen die aktiven Systeme zum Beispiel durch Ausweichen, Bremsen oder Spurhalten direkt in das Geschehen ein.

1.1 Perzeption

Zu den wichtigsten Komponenten in einem Fahrerassistenzsystem gehören die Sensoren. Die Software in einem Kraftfahrzeug kann nur Entscheidungen treffen, wenn sie die Möglichkeit hat, dessen Umgebung wahrzunehmen. Um eine möglichst präzise Wahrnehmung zu erreichen, benötigt man mehrere Sensoren unterschiedlicher Art, deren Daten gemeinsam ausgewertet werden. Jeder Sensortyp hat unterschiedliche Vor- und Nachteile. Ein Kamerabild enthält sehr viele Informationen, woraus sich die Dimensionen von Objekten berechnen oder die Fahrspur erkennen lässt. Da allerdings durch die Abbildung in eine Ebene Tiefeninformation verloren geht, ist dieses Verfahren zum Messen von Distanzen nicht optimal. Mit Radartechnik hingegen lassen sich sehr gut Entfernungen und durch den Doppler-Effekt Geschwindigkeiten messen. Informationen über die Abmessungen erkannter Objekten fehlen hier, sodass lediglich Punktantworten vorliegen. Erst die kombinierte Auswertung dieser beiden Informationen führt zu einer exakteren Erfassung der Umgebung, als wenn nur die Informationen einer der beiden Sensoren zur Verfügung stünden. Aufgrund von Vielzahl und Heterogenität der Sensoren müssen ihre Daten zu einer Informationsbasis zusammengeführt werden. Diese Vorgehensweise wird Sensordatenfusion (Abbildung 1) genannt und von so genannten Perzeptionssystemen durchgeführt.

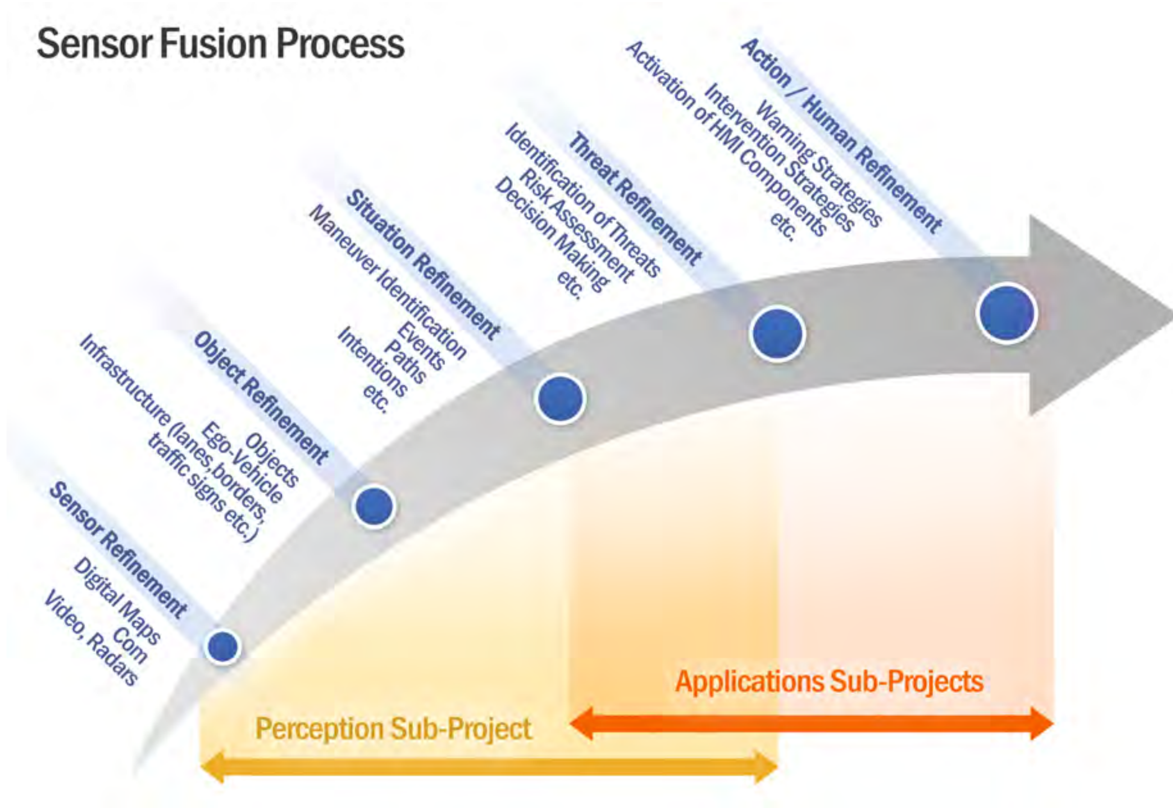

Abbildung 1: Sensordatenfusion [6, p. 15]

Im Gegensatz zu Kraftfahrzeugen, in denen jedes Fahrerassistenzsystem die Sensoren separat auswertet, erschließen sich in einem Perzeptionssystem neben der erhöhten Präzision weitere Vorteile. Aufgrund von Überlagerung der Informationsbereiche verschiedener Sensoren kann das System unter Umständen trotz des Ausfalls eines Sensors weiterarbeiten. Durch die Möglichkeit, Sensoren unterschiedlich zu positionieren und Informationen von anderen Kraftfahrzeugen zu beziehen, erweitern sich die Sichtbereiche.
In den Abbildungen 2 und 3 ist die Arbeitsweise eines Perzeptionssystems veranschaulicht. Als Kernfunktionalität wird die Sensordatenfusion zugrunde gelegt, welche Zugriff auf alle Informationen der im Kraftfahrzeug vorhandenen Sensoren hat. Zusätzlich werden GPS- und Kartendaten sowie weitere Kommunikationseinheiten, zum Beispiel Perzeptionsdaten anderer Verkehrsteilnehmer oder aktuelle Verkehrsinformationen, in den Prozess einbezogen.

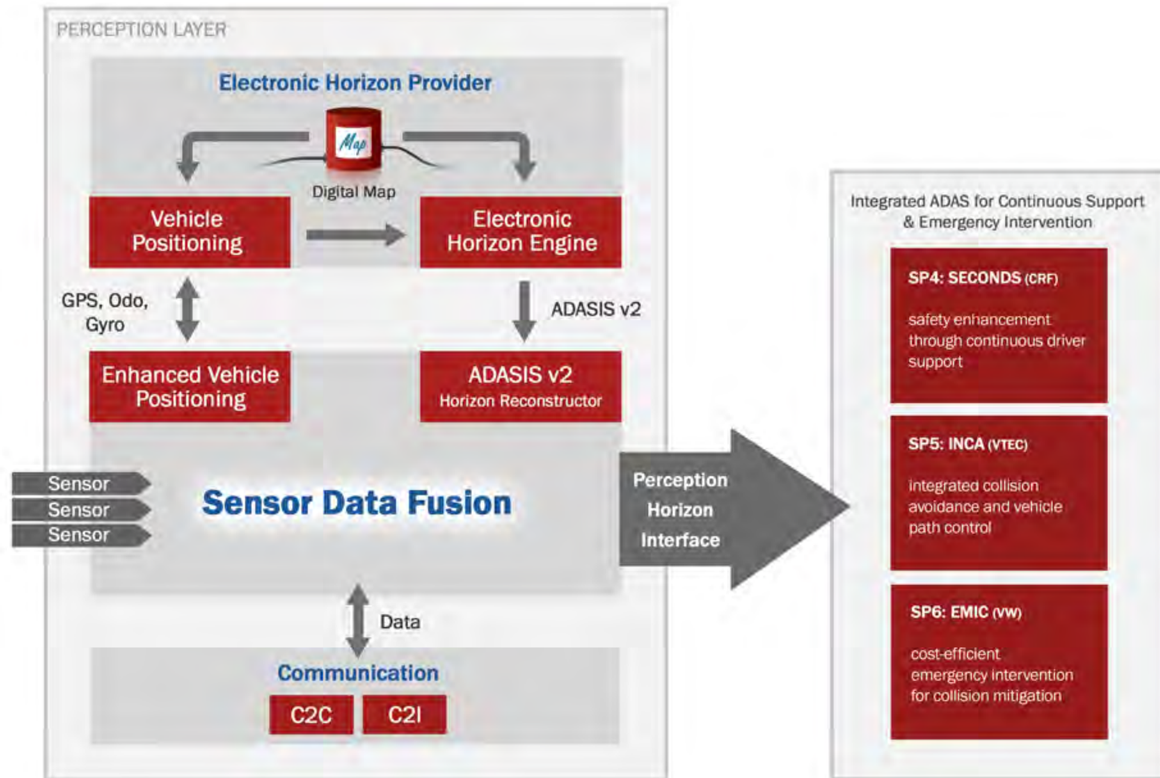

Abbildung 2: Perzeptionsschicht [6, p. 58]

Die gewonnenen Informationen werden von der Perzeptionsplattform, die aus vielen Untermodulen besteht (Abbildung 3), aufbereitet. Über den so genannten Perzeptionshorizont, eine wohldefinierte Schnittstelle, können nachgeschaltete Systeme die aufbereiteten Daten der Perzeptionsplattform abrufen.

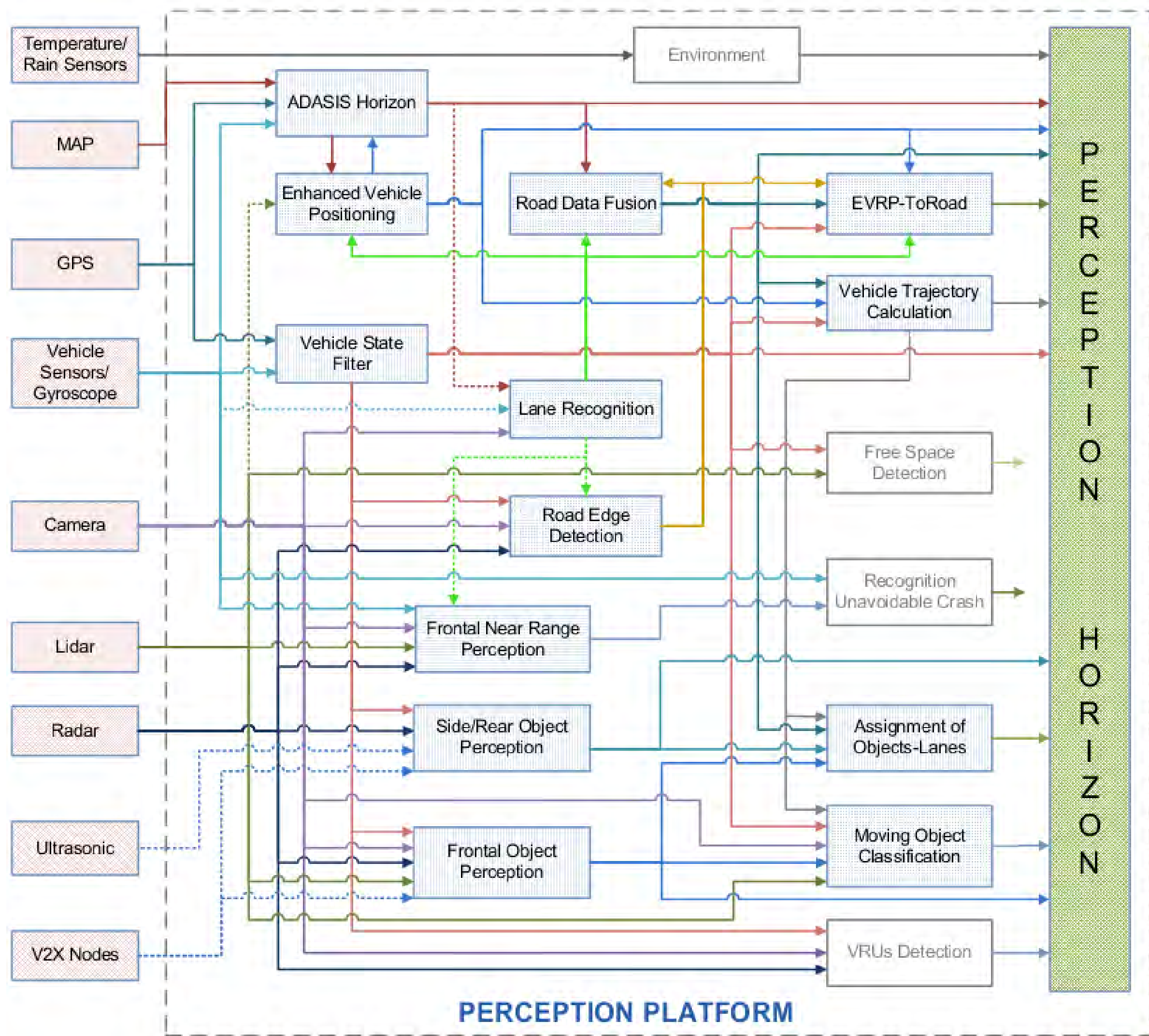

Abbildung 3: Architektur der Perzeptionsplattform [7, p. 14]

1.2 interactIVe

Zitat 1. „InteractIVe vision is accident-free traffic realised by means of affordable integrated safety systems penetrating all vehicle classes, and thus accelerating the safety of road transport." [6, p. 6]

Seit den neunziger Jahren wird viel über die effiziente Nutzung von Sensordaten geforscht. Allerdings sind die Ergebnisse bis heute nicht in ein einheitliches System zusammengefasst, welches auf dem Markt verfügbar ist [6, p. 27]. Das Institut FORWISS an der Universität Passau ist Partner im Forschungsprojekt interactIVe, welches ins Leben gerufen wurde, um ein solches System zu entwerfen. interactIVe ist in sieben Teilprojekte

untergliedert (Abbildung 4). Das Teilprojekt SP2 enthält die Spezifikation, Implementierung und Realisierung einer Perzeptionsplattform, deren Ergebnisse die Grundlage aller nachfolgenden Sicherheitsanwendungen darstellt. Die in diesem Projekt erstellten Module können unabhängig von den Randbedingungen wiederverwendet werden. Das bedeutet, dass dieselbe Perzeptionsplattform in verschiedenen Kraftfahrzeugen mit unterschiedlichen Sensoren benutzt werden kann.

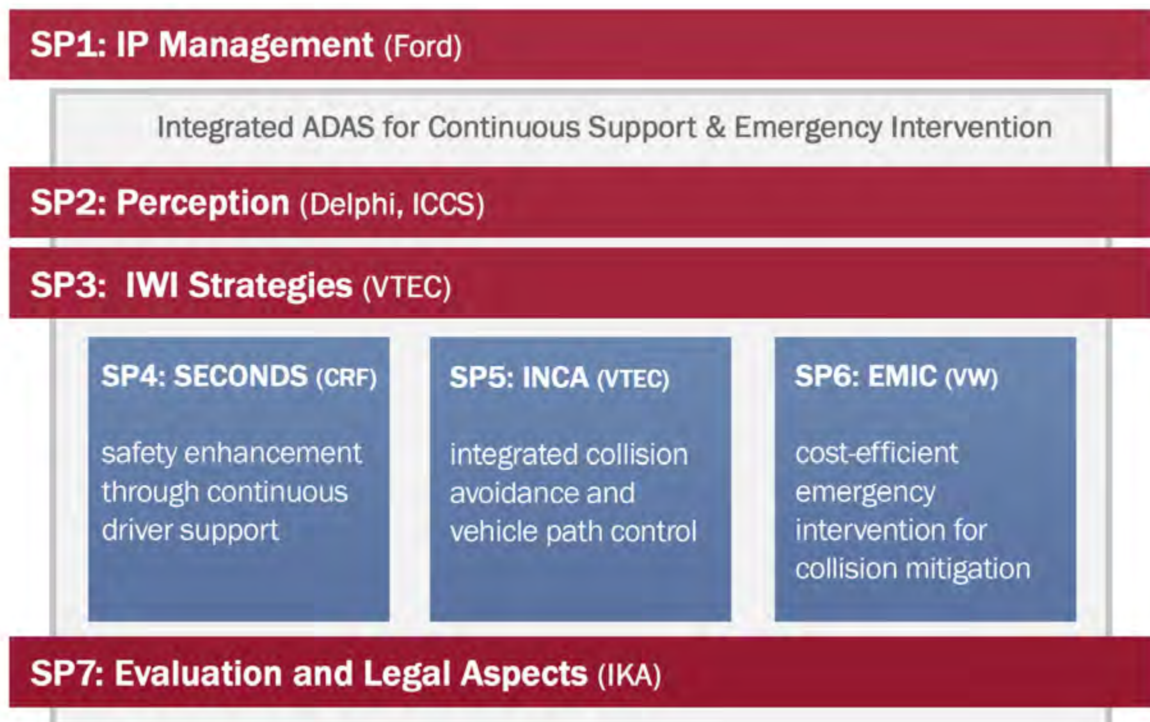

Abbildung 4: Struktur von interactIVe[1] [6, p. 32]

1.3 Framework

Bei einem Projekt dieses Umfangs bietet es sich an, ein Framework zugrunde zu legen, in das die eigentliche Funktionalität integriert wird. Es liegt aus Zeit- und Kostengründen nahe, dass bestehende Lösungen auf dem Markt analysiert werden, bevor man über eine komplette Neuentwicklung nachdenkt. Von Projekten aus der Vergangenheit ist zudem bekannt, dass für verschiedene Kraftfahrzeugtypen jeweils eigene Frameworks genutzt wurden. Das interactIVe Projekt hat beschlossen, auf ein einzelnes, bereits existierendes Framework aufzusetzen, welches kompatibel mit allen beteiligten Kraftfahrzeugen

[1] Die Begriffe Ford, Delphi, ICCS, VTEC, CRF, VW und IKA werden ab Seite 41 erläutert.

ist. Dies ermöglicht den Entwicklern, sich ausschließlich auf die Programmierung des Perzeptionssystems zu konzentrieren. Zur Auswahl standen die Werkzeuge RTMaps von Intempora und ADTF von elektrobit. Nach umfangreicher Analyse von Funktionen und Möglichkeiten der beiden Werkzeuge sowie Vorträgen beider Firmen fiel die Wahl auf ADTF. Da ADTF bereits von vielen Projektpartnern genutzt wird, entschied man sich gegen RTMaps. Für Forschungsinstitute ist ADTF kostenlos. ADTF ist ein flexibles Werkzeug für die Entwicklung neuer Funktionen in Kraftfahrzeugen. Das modulare System bietet zusammen mit Standardkomponenten eine solide Basis mit dokumentierten Schnittstellen. Mit dem betriebssystemunabhängigen Software Development Kit (SDK) können neue Funktionen effizient umgesetzt werden [2].
Die ausschließliche Nutzung von ADTF hat viele Vorteile:

- **Plugin-Basiert**
  Durch das Plugin-Basierte Framework ist ein Komponentenaustausch einfach möglich.

- **GUI und Visualisierung**
  Die Visualisierung der Daten angeschlossener Geräte und Sensoren wird durch die integrierte Qt-Unterstützung gewährleistet.

- **Hardware Schnittstellen**
  Schnittstellen zu angeschlossener Hardware sind bereits vorhanden, zum Beispiel über CAN-Bus.

- **Synchronisation**
  Die Datenströme werden anhand von Zeitstempeln synchronisiert.

- **Datenaustausch**
  Ein einheitliches Datenformat ermöglicht es Benutzern, untereinander Konfigurationen auszutauschen.

Die oben genannten Punkte ermöglichen es den Softwareentwicklern, sich auf die Algorithmen des Perzeptionssystems zu konzentrieren. Natürlich muss eine gemeinsame Spezifikation erstellt werden, um die Austauschbarkeit von Sensoren und Modulen zu gewährleisten. Dieses Plugin-Konzept verkörpert genau die Aufgabenstellung in interactIVe: Eine einheitliche Softwarebasis für mehrere Versuchsfahrzeuge.

1.4 ADTF

ADTF arbeitet datenflussbasiert. Es werden Datenpakete (`IMediaSample`) von einer Komponente zur nächsten übertragen und dort ausgewertet (Abbildung 5). Komponenten werden auch Filter genannt. Jedes Filter kann Ein- und Ausgänge besitzen, über die es mit anderen Filtern verbunden sein kann, um mit ihnen zu kommunizieren. Diese

Schnittstellen werden Pins genannt. Eine Verbindung zwischen Pins verläuft in eine vorgegebene Richtung. Somit ist ein Pin entweder ein Eingabe- oder ein Ausgabepin. Zudem wird Typsicherheit zwischen den einzelnen Pins gewährleistet, indem eine Verbindung zwischen zwei Pins nur hergestellt werden kann, wenn beide mit kompatiblen Datentypen (`IMediaType`) arbeiten. Jedes Filter kann von seinem Entwickler mit eigenen Einstellungsmöglichkeiten, den so genannten Properties, ausgerüstet werden. Damit kann jeder Benutzer das Filter seinen Bedürfnissen entsprechend anpassen. Eine Kombination aus Filtern und deren Verbindungen untereinander wird Filtergraph, oder auch Konfiguration, genannt [4, p. 15]. Durch die implizit festgelegten Übertragungsrichtungen handelt es um einen gerichteten Graphen.

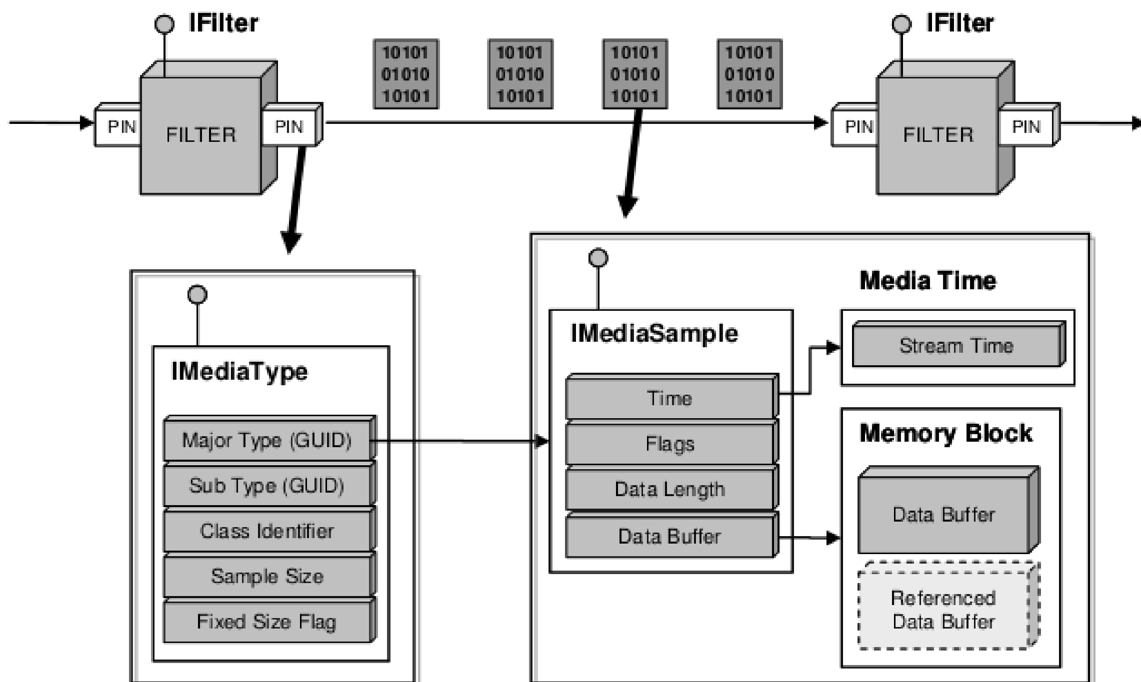

Abbildung 5: ADTF Streaming Architektur [3, p. 13]

ADTF besteht aus mehreren Programmen für Anwender und Entwickler. Wird in dieser Arbeit von dem ADTF GUI geredet, ist die ADTF Entwicklungsumgebung gemeint (Abbildung 6). Sie ist ein essentieller Teil für die Entwicklung eines Filtergraphen und im Rahmen von interactIVe das meist genutzte ADTF Programm.

Abbildung 6: ADTF GUI mit geladener Konfiguration

1.5 Rechtliche Situation

Um neuartige Fahrerassistenzsysteme zur Produktreife zu bringen und in den Markt einführen zu können, muss die komplexe rechtliche Situation beachtet werden. Für diese Aufgaben wurde in interactIVe ein eigenes Teilprojekt geschaffen. Alle in Europa zugelassenen Fahrerassistenzsysteme unterliegen einer gemeinsamen Gesetzgebung. Während die Genehmigung für Bremssysteme relativ einfach zu bekommen ist, gibt es für Lenksysteme sehr strenge Richtlinien. Autonomes Lenken, welches der Kursänderung dient, ist nur bei Geschwindigkeiten von weniger als 12 km/h erlaubt. Dies ist ausreichend für automatische Einparksysteme, nicht aber für Einfädelassistenten oder Ausweichmanöver. Stabilitätsprogramme, die das dynamische Verhalten optimieren, dürfen für kurze Momente in das Lenkverhalten eingreifen, sofern ihr Hauptziel nicht ist, der Richtung abzuweichen, die der Fahrer anstrebt. Eine weitere Auflage ist, dass ein Fahrerassistenzsystem nur Informationen verwenden darf, die an Bord erzeugt wurden. Fahrsicherheitsmanöver auf Basis externer Informationsquellen, sind nach derzeitiger Rechtslage unzulässig. Während interactIVe einerseits versucht ein System zu entwerfen, welches nach den geltenden Richtlinien arbeitet, dringt das Projekt dennoch in Bereiche vor, die zum heutigen Zeitpunkt auf unseren Straßen nicht gestattet sind [8].

2 Motivation

Zitat 2. „Obwohl die Notwendigkeit des Testens selbst geschriebener Software unstreitig ist, gehört es wohl zu den meist gehassten Aufgaben eines Entwicklers. Es ist umständlich, zeitaufwändig, und am Ende weiß er nie, ob er wirklich alle Fehler erwischt hat.“ [10, p. 22]

Einige Sekunden arbeiteten die Softwaresysteme in der Ariane-5-Rakete mit der Seriennummer 501 bei ihrem Jungfernflug V88 am 4. Juni 1996 erwartungsgemäß. Gute 36 Sekunden nach dem Start versuchte der Bordcomputer eine 64-Bit Fließkommazahl in einen vorzeichenbehafteten ganzzahligen 16-Bit Wert umzuwandeln, welches einen arithmetischen Überlauf verursachte. Listing 1 zeigt die fehlerhafte Codezeile. Dieser unscheinbar wirkende Fehler löste eine verheerende Kettenreaktion aus, die 40 Sekunden nach dem Start den Selbstzerstörungsmechanismus aktivierte, der ein Abstürzen der kompletten Rakete im Notfall verhindern soll. Die Untersuchungskommission, welche den Unfall analysierte, führte das Fehlverhalten auf einen systematischen Software-Designfehler zurück, welcher durch eine bessere Qualitätssicherung während der Entwicklung hätte behoben werden können [13]. Selbst erfahrenen Programmierern ist auf den ersten Blick nicht ersichtlich, dass dieses kleine Stück Quelltext einen Materialschaden von 340 Millionen Dollar verursachen kann. Fakt ist, dass ein ausreichendes Konzept zum Testen von Software obligatorisch ist, um nach der Auslieferung nicht mit bösen Überraschungen konfrontiert zu werden.

Listing 1: Fehlerhafte Codezeile der Ariane 5

```
1 P_M_DERIVE(T_ALG.E_BH) :=
2     UC_16S_EN_16NS(TDB.T_ENTIER_16S((1.0/C_M_LSB_BH) *
3     G_M_INFO_DERIVE(T_ALG.E_BH)))
```

2.1 Software Testen

Das Beispiel der Ariane-5-Rakete zeigt deutlich, dass die Qualitätssicherung ein wichtiger Bestandteil im Entwicklungsprozess einer Software ist. Das heute noch viel verwendete Wasserfallmodell (Abbildung 7) sieht Softwaretests lediglich in den letzten Phasen der Entwicklung vor, was verheerende Folgen haben kann. Je später man einen Fehler findet, desto zeit- und kostenaufwändiger wird es, ihn zu beheben.

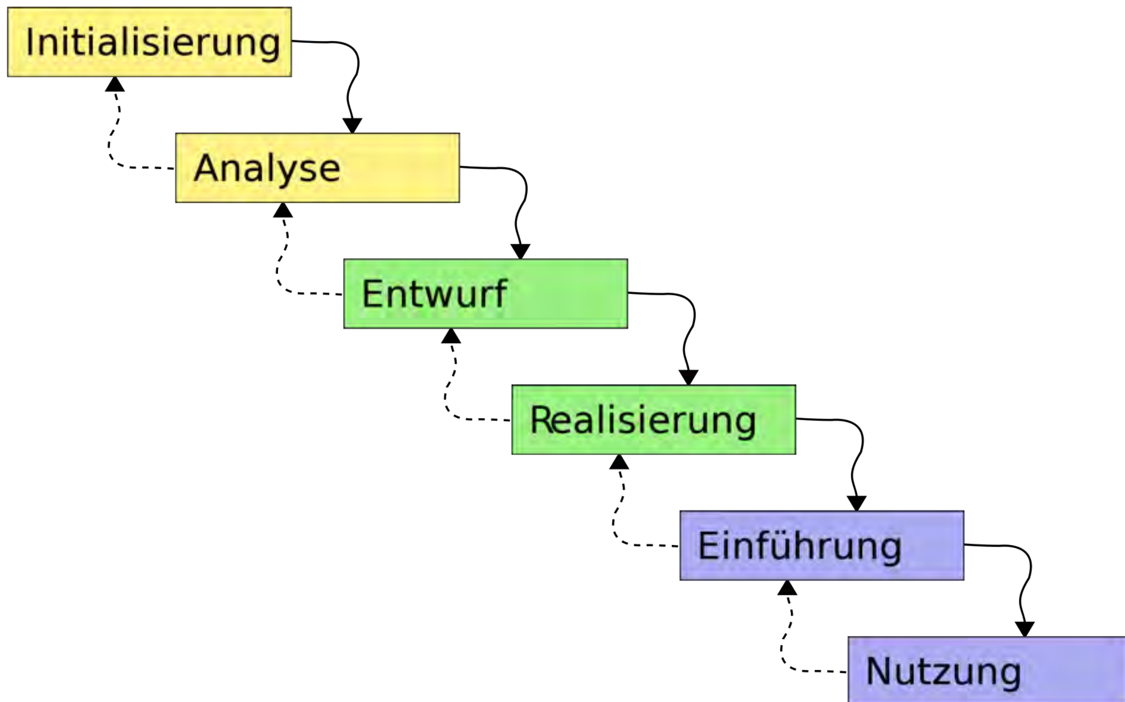

Abbildung 7: Wasserfallmodell [9]

Das Gegenteil des Wasserfallmodells zeigt ein Grey-Box-Testverfahren, welches sich Test-Driven-Development (TDD) nennt. Beim Test-Driven-Development werden immer zuerst die Tests entworfen und danach die entsprechende Komponente entwickelt. Es findet häufig in Verbindung mit agiler Softwareentwicklung Anwendung.

Zitat 3. „Wer zuerst Tests schreibt und dann erst programmiert, erzeugt sauberen Code und geprüfte Programme." [14, p. 32]

Beide Verfahren erscheinen auf den ersten Blick nicht besonders optimal und verfehlen jeweils in einer anderen Richtung ihre Ziel. Wünschenswert wäre ein Mittelweg, der Tests sinnvoll in den gesamten Entwicklungsprozess integriert, wie beispielsweise der ganzheitliche Ansatz in Abbildung 8. Hier wird von Projektbeginn an versucht, die Fehlerzahl gering zu halten, indem über die gesamte Projektlaufzeit hinweg Tests in die Entwicklung integriert werden. Zusätzlich existieren Aufgaben des Testmanagements, welche durchgehend die Vorgehensweise überwachen und steuern.

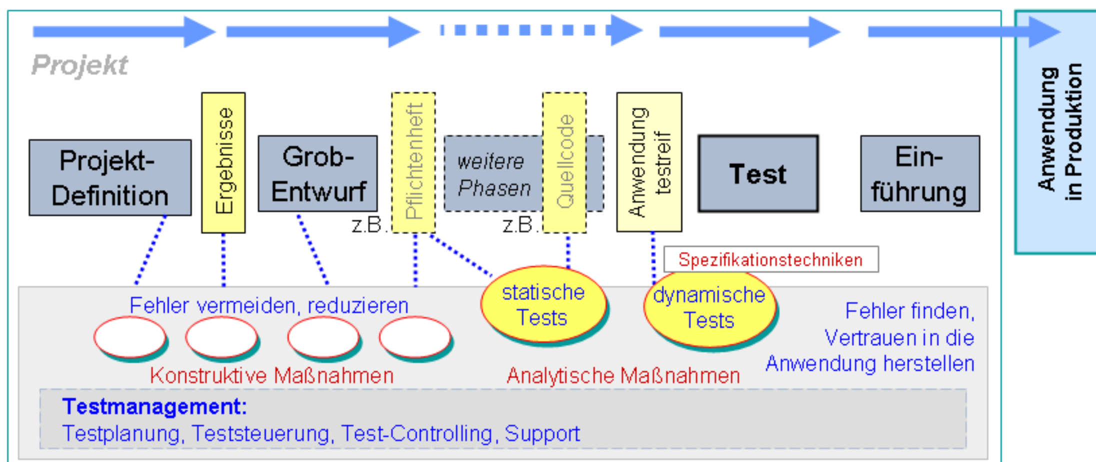

Abbildung 8: Der ganzheitliche Ansatz beim Testen [12]

Es gibt eine Vielzahl an Möglichkeiten Software zu testen, aber nicht immer sind alle Arten von Tests sinnvoll. Dies muss im Einzelfall abgewägt werden. Zuletzt geht es aber immer nur um Eines: Fehler zu finden. Ganz egal für welche Art von Tests man sich entscheidet, ist es wichtig, strukturiert und planvoll vorzugehen.

2.1.1 Verifikation und Validierung

Die Prüf- und Analyseprozesse Verifikation und Validierung, auch „V & V“ genannt, stellen während des Entwicklungsprozesses sicher, dass das Programm seine Spezifikation erfüllt und die von den Auftraggebern erwarteten Funktionen bietet. Verifikation und Validierung werden sehr oft miteinander verwechselt, sind aber nicht dasselbe. Das Ziel der Verifikation besteht darin, festzustellen, ob die Spezifikation eingehalten wird. Es wird verifiziert, dass funktionale und nichtfunktionale Anforderungen erfüllt sind. Während eine funktionale Anforderung festlegt, was das Produkt tun soll, indiziert die nichtfunktionale Anforderung die Eigenschaften des Produktes, bei denen es sich zum Beispiel um Reaktionszeiten der Anwendung handeln kann. Die Validierung hingegen zielt darauf ab, dass alle Kundenerwartungen erfüllt werden [11, p. 556].

Zitat 4. „Boehm (1979) hat den Unterschied zwischen ihnen prägnant ausgedrückt:

- *Validierung:* Erstellen wir das richtige Produkt?
- *Verifikation:* Erstellen wir das Produkt richtig?“

[11, p. 556]

Die wichtigsten System- und Komponententests werden in diesem Abschnitt vorgestellt.

2.1.2 Integrationstests

Integrationstests testen das Zusammenspiel einzelner Komponenten innerhalb eines Softwaresystems. Hierbei handelt es sich um ein White-Box-Testverfahren, bei dem das Testteam Zugriff auf den Quelltext der Software hat. Wird ein Fehler gefunden, sucht das Team dessen Ursache und beseitigt diese sofort. Integrationstests werden hauptsächlich durchgeführt, um Systemfehler aufzuspüren [11, p. 583].

2.1.3 Auslieferungstests

Auslieferungstests hingegen werden als Black-Box-Testverfahren durchgeführt, bei welchen der Quelltext nicht zur Verfügung steht. Das Testteam versucht zu verifizieren, dass das System zuverlässig ist und seine Anforderungen erfüllt. Sollten Probleme auftauchen, werden diese der Entwicklungsabteilung gemeldet, dort reproduziert und behoben. Sind Auftraggeber am Auslieferungstest beteiligt, nennt man diesen auch Abnahmetest [11, p. 583].
Auslieferungstests erfordern strategische Vorgehensweisen, wie zum Beispiel:

- Mit Eingaben die Software zu Fehlermeldungen zwingen
- Erzwingen von ungültigen Ausgaben
- Erzeugen von Eingaben, die die Eingabespeicher zum Überlaufen bringen
- Erzwingen von zu großen oder zu kleinen Berechnungsergebnissen

2.1.4 Leistungstests

Um sicherzustellen, dass das System die vorgesehene Last verarbeiten kann, kann es nach der vollständigen Integration auf Eigenschaften wie Geschwindigkeit und Zuverlässigkeit getestet werden. Eine effektive Methode zur Fehlersuche besteht darin, auf dem System eine Last zu erzeugen die deutlich außerhalb der vorgesehenen Grenzen liegt. Hierbei ist auch oft von Belastungstests die Rede. Diese Art von Tests analysiert einerseits das Ausfallverhalten des Systems, andererseits können hierdurch Fehler gefunden werden, die unter normalen Umständen nicht aufträten [11, p. 589].

2.1.5 Schnittstellentests

Schnittstellentests gehören zur Familie der Komponententests. Es wird zwischen verschiedenen Komponentenarten unterschieden:

- Einzelne Funktionen
- Klassen mit mehreren Attributen und Methoden
- Zusammengesetzte Komponenten, welche aus mehreren Klassen oder Funktionen bestehen.

Schnittstellentests werden in der Regel auf zusammengesetzte Komponenten angewendet, welche eine definierte Schnittstelle bieten, über die der Zugriff erfolgen kann. Es wird in erster Linie überprüft, ob sich die Komponentenschnittstelle gemäß ihrer Spezifikation verhält.

Zitat 5. „Schnittstellenfehler der zusammengesetzten Komponente sind beim Testen der einzelnen Objekte oder Komponenten nicht zu entdecken.“ [11, p. 591]

2.2 Testen des Perzeptionssystems

Da die Perzeptionsplattform ein Kernbestandteil in einem Fahrerassistenzsystem ist, muss eine besondere Zuverlässigkeit gewährleistet werden. Hierzu gehören Absturz- und Fehlerfreiheit, sowie Ausfallsicherheit. Alle Entscheidungen, die die Software trifft, basieren auf den Informationen der Perzeptionsplattform. Falsche Aktionen eines solchen Systems können gravierende Folgen haben. Zudem muss jede Komponente in einem Fahrzeug in vielen Ländern gesondert zugelassen werden, um am öffentlichen Straßenverkehr teilnehmen zu dürfen. In Deutschland liegt diese Zuständigkeit beim Technischen Überwachungsverein (TÜV). Hierzu ist es erforderlich, dass Entwicklungen für die Automobilindustrie einer besonders hohen Qualitätssicherung unterliegen.

2.2.1 Ist-Situation

In der momentanen Situation im Projekt existieren Testpläne für verschiedene Arten von Tests. Für alle Testszenarien gibt es Checklisten, die ausgefüllt werden müssen. Des Weiteren existiert ein Bugtracking-System, um alle gefundenen Fehler festzuhalten.

Tests einzelner Filter
Es werden vier Testfälle durchlaufen.

- **Schnittstellentest**
  Der Schnittstellentest analysiert das Verhalten im Falle fehlender Sensordaten oder nicht vorhandener Verbindungen zu anderen Filtern.

- **Plausibilitätstest**
  Der Plausibilitätstest überprüft ob, Eingangssignale innerhalb der Erwartungswerte liegen. Zudem wird erörtert, inwiefern fehlerhafte Eingaben, wie etwa Nullzeiger oder nicht verfügbare Daten von Sensoren, die Funktionsweise des Filters beeinflussen.

- **Leistungstest**
  Der Leistungstest stellt zum einen sicher, dass jedes Filter innerhalb einer bestimmten Zeitspanne reagiert, zum anderen werden hier Speicherlecks aufgedeckt.

- **Weitere Tests**
  Die finalen Tests analysieren die Modul-Abhängigkeiten und GUI-Elemente.

Testen der ganzen Plattform
Vor jedem Release muss das Zusammenspiel einzelner Komponenten innerhalb der Plattform getestet und dokumentiert werden. Im Falle von Fehlern wird der Checkliste eine Tabelle mit deren Beschreibungen angefügt. Das Testen der vollständigen Plattform wird in unterschiedlicher Form von zwei verschiedenen Teilprojekten durchgeführt.

- **Integrationstest** Das Teilprojekt SP2 führt nach jedem Entwicklungsschritt einen Integrationstest durch. Hier wird sichergestellt, dass alle einzeln getesteten Komponenten auch innerhalb einer kompletten Konfiguration wie erwartet arbeiten. Erst wenn diese Tests erfolgreich abgeschlossen wurden, wird der aktuelle Entwicklungsstand mit den anderen Projektpartnern geteilt.

- **Auslieferungstest** Das SP7 Teilprojekt ist unter anderem für die Auslieferungstests verantwortlich. Es werden die funktionalen Anforderungen der gesamten Plattform getestet. Abbildung 9 zeigt einen beispielhaften Testfall, der überprüft, wie das Kraftfahrzeug auf einen ungewollten Spurwechsel reagiert.

<table>
<tr><td>Test Scenario</td><td>Unintended lane departure-accidents</td></tr>
<tr><td>Test Case</td><td>6.3</td></tr>
<tr><td colspan="2">Unintended lane / road departure (left)</td></tr>
<tr><td colspan="2">y_0 $v_{Host Vehicle\ y}$ $v_{Host Vehicle\ X}$</td></tr>
<tr><td>Description</td><td>Unintended lane departure of the host vehicle to the left side.</td></tr>
<tr><td>Relevant functions</td><td>CS, RORP</td></tr>
<tr><td>Use case</td><td>UC_06_451_v0, UC_06_452_v0, UC_06_503_v2, UC_06_510_v2</td></tr>
<tr><td>Vehicle initial parameters</td><td>
<table>
<tr><td>$v_{Host Vehicle\ x}$ [km/h]</td><td>30</td><td>50</td><td>70</td></tr>
<tr><td>$v_{Host Vehicle\ y}$ [km/h]</td><td>1</td><td>2</td><td>5</td></tr>
<tr><td>y_0 [m]</td><td>0.5</td><td></td><td></td></tr>
<tr><td>Status turn indicator [-]</td><td>on</td><td>off</td><td></td></tr>
</table>
</td></tr>
<tr><td>Number vehicles</td><td>1</td></tr>
<tr><td>Required Equipment</td><td>none</td></tr>
<tr><td>Environmental initial parameters</td><td>
<table>
<tr><td>lane marking</td><td>solid</td><td>Dash[1]</td><td>Solid[1] dash</td></tr>
<tr><td>Road radius</td><td>∞</td><td></td><td></td></tr>
<tr><td>driven lane</td><td>right</td><td></td><td></td></tr>
<tr><td>Number of lanes</td><td>1</td><td>2</td><td></td></tr>
<tr><td>loading of vehicle (only relevant for trucks)</td><td>Basic</td><td>Fully loaded</td><td></td></tr>
</table>
</td></tr>
<tr><td>Assessment</td><td>☒ Technical ☐ User-related</td></tr>
<tr><td>Driver reaction (only relevant for the technical assessment)</td><td>

No reaction of the driver on any warning or intervention before the highest warning level is reached (afterwards the reaction of the driver depends on function).
Driving with constant speed
Driver should steer the vehicle to lane in the way that the defined lateral velocity is reached.
After leaving the right lane the driver should continue (same lateral velocity) up to the moment the vehicle leaves the road on the left side

In order to ensure the correct lateral velocity during the tests a target can be placed target at the side of the road in a defined distance, see figure below. In this case the driver has to start to steer at a certain point and to drive afterwards straight ahead towards the target. The lateral and longitudinal distance of the target should be should be chosen based on the chosen angle of the test case (e.g. $v_{Host Vehicle\ x}$ = 50 km/h & $v_{Host Vehicle\ y}$ = 5 km/h → The angle is 5.74° → distance of the target dy (from the center line) = 3 m, dy = 30 m)
</td></tr>
<tr><td>Comment</td><td>After the 1: If available on the test track</td></tr>
</table>

Abbildung 9: Beispiel Checkliste aus SP7

2.2.2 Testen auf Anwendungsebene

Die bisher vorgestellten Tests sind sehr umfangreich, decken allerdings nicht alle Möglichkeiten ab, Fehler zu finden. Bisher wird ausschließlich über das ADTF GUI getestet, was einige Einschränkungen mit sich bringt. Automatisierte Tests, die eine Konfiguration mehrfach starten, oder Langzeittests sind somit nicht ohne erheblichen Aufwand möglich. Listing 2 zeigt, wie aufwändig die Initialisierung der ADTF Umgebung über das SDK ist. Dieses aus der ADTF Dokumentation stammende Beispiel enthält noch keine Testroutinen. Dies würde sicherlich die Anzahl der Quelltextzeilen noch einmal verdoppeln. Eine solch komplizierte Vorgehensweise ist nicht besonders hilfreich, um sinnvoll und strukturiert zu testen. Das in ADTF integrierte Test-Framework ist sehr schwach, weil sich seine Funktionalität auf das Testen der korrekten Initialisierung eines einzelnen Filters beschränkt, was für die Ansprüche von interactIVe nicht ausreichend ist. Es fehlt die Möglichkeit, sinnvoll einzelne Komponenten zu entfernen oder zu deaktivieren, um das Verhalten der Konfiguration ohne eine bestimmte Komponente zu analysieren. Des Weiteren wäre es sinnvoll, das Verhalten von einzelnen Filtern innerhalb einer kompletten Konfiguration zu analysieren. Hierfür ist es notwendig, die an Pins übertragenen Mediasamples abzufangen und mit eventuellen Referenzdaten zu vergleichen. Neben der Evaluation von aktuellen Testtechniken im Projekt interactIVe und dem Verstehen vom ADTF SDK, ist die Entwicklung der libadtftest ein essenzieller Bestandteil dieser Arbeit. Sie ermöglicht das Testen von einzelnen Komponenten oder ganzen Konfigurationen und beseitigt die oben genannten Schwachstellen. Mit ihrer Hilfe können im Rahmen des Projekts interactIVe strukturiert Testfälle erstellt werden, um eine möglichst hohe Fehlertoleranz innerhalb der Plattform zu erreichen. Die folgenden Kapitel enthalten eine Einführung in die libadtftest.

Listing 2: Initialisierung vom ADTF SDK

```
1  #include "stdafx.h"
2
3  ADTF_DEFINE_RUNTIME()
4
5
6  struct sServices
7  {
8      const tChar* strPlugin;
9      const tChar* strOID;
10 };
11
12 const sServices pServices[] = {{"adtf_clock.srv", "adtf.core.reference_clock" },
13                               {"adtf_kernel.srv", "adtf.core.kernel" },
14                               {"adtf_namespace.srv", "adtf.core.namespace" },
15                               {"adtf_console.srv", "ucom.core.console_device" },
16                               {"adtf_memory.srv", "adtf.core.memory_manager" },
17                               {"adtf_media_description.srv",
18                               "adtf.core.mediadescription_manager" },
19                               {"adtf_sample_pool.srv", "adtf.core.sample_pool" } ,
20                               {"adtf_session.srv", "adtf.core.session_manager" },
21                               {"adtf_filtergraph_manager.srv",
22                               "adtf.core.filtergraph_manager"}};
23
24
```

```
25
26 tResult LoadPlugins(const tChar* strPluginDir, __exception = NULL)
27 {
28     // We just require the plugins to be in the same directory as the executable.
29     cFilename strTmpDir = strPluginDir;
30     strTmpDir.AppendTrailingSlash();
31
32     // We register (load) all plugins with the help of the runtime
33     tInt nNumPlugins = sizeof(pServices) / sizeof(sServices);
34     for (tInt nPlugin = 0; nPlugin < nNumPlugins; ++nPlugin)
35     {
36         RETURN_IF_FAILED(_runtime->RegisterPlugin(strTmpDir + pServices[nPlugin].strPlugin,
37          IRuntime::RL_Kernel, NULL, 0, __exception_ptr));
38     }
39     RETURN_NOERROR;
40 }
41
42 tResult CreateServices(__exception = NULL)
43 {
44     // We create all service instances and register them at the runtime.
45     tInt nNumServices = sizeof(pServices) / sizeof(sServices);
46     for (tInt nService = 0; nService < nNumServices; ++nService)
47     {
48         cObjectPtr<IService> pService;
49         RETURN_IF_FAILED(_runtime->CreateInstance(pServices[nService].strOID, IID_SERVICE,
50         (tVoid**) &pService, NULL, __exception_ptr));
51         RETURN_IF_FAILED(_runtime->RegisterObject(pService, pServices[nService].strOID,
52         IRuntime::RL_Kernel, 0, __exception_ptr));
53     }
54     RETURN_NOERROR;
55 }
56
57 tResult LoadGlobalsAndConfig(const tChar* strGlobals, const tChar* strConfig,
58 __exception = NULL)
59 {
60     // We tell the session manager which globals and which config to load
61     cObjectPtr<ISessionManager> pSessionManager;
62     RETURN_IF_FAILED(_runtime->GetObject(NULL, IID_ADTF_SESSION_MANAGER,
63     (tVoid**) &pSessionManager, __exception_ptr));
64     RETURN_IF_FAILED(pSessionManager->LoadGlobalsFromFile(strGlobals, __exception_ptr));
65     RETURN_IF_FAILED(pSessionManager->LoadConfigFromFile(strConfig, __exception_ptr));
66     RETURN_NOERROR;
67 }
68
69 tResult Init(const tChar* strPluginDir, const tChar* strGlobals, const tChar* strConfig,
70  __exception = NULL)
71 {
72     RETURN_IF_FAILED(LoadPlugins(strPluginDir, __exception_ptr));
73     RETURN_IF_FAILED(CreateServices(__exception_ptr));
74     RETURN_IF_FAILED(_runtime->SetRunLevel(IRuntime::RL_System, __exception_ptr));
75     RETURN_IF_FAILED(LoadGlobalsAndConfig(strGlobals, strConfig, __exception_ptr));
76     return _runtime->SetRunLevel(IRuntime::RL_Running, __exception_ptr);
77 }
78
79 tInt main(tInt nArgc, const tChar** pArgv)
80 {
81     if (nArgc != 5)
82     {
83         LOG_ERROR(cString::Format("usage:␣%s␣<plugin_dir>␣<globals>␣<config>␣<seconds>",
84          pArgv[0]));
85         return 1;
86     }
87
```

```
88      tInt nSeconds = cString(pArgv[4]).AsInt();
89
90      // Create the global runtime object
91      _runtime = new cRuntime(nArgc, pArgv);
92
93      // Register our macro resolver, otherwise the filters will
94      //not be able to resolve relative config paths
95      {
96          cObjectPtr<IMacroResolver> pResolver = new cMacroResolverImpl;
97          _runtime->RegisterObject(pResolver, OID_MACRORESOLVER, IRuntime::RL_Kernel);
98      }
99
100     // Start the initialization procedure
101     cException oException;
102     if (IS_FAILED(Init(pArgv[1], pArgv[2], pArgv[3], &oException)))
103     {
104         __catch_exception(oException)
105         {
106             LOG_EXCEPTION(oException);
107         }
108         return 1;
109     }
110
111     // Let the application run for 5 seconds.
112     cSystem::Sleep(nSeconds * 1000000);
113
114     // Shut it down properly
115     _runtime->SetRunLevel(IRuntime::RL_Shutdown);
116
117     delete _runtime;
118
119     return 0;
120 }
```

3 Rahmenbedingungen

Während der Entwicklung der libadtftest sind Probleme aufgetreten, die den Verlauf des Projektes beeinflusst haben. Zudem haben viele Werkzeuge die Entwicklung vereinfacht. Dieses Kapitel stellt die wichtigsten Rahmenbedingungen vor und erläutert diese.

3.1 Fehlende Referenzdaten

Bis zur Fertigstellung dieser Arbeit gab es innerhalb des Projekts interactIVe keine Referenzdaten. Erst mithilfe dieser ist es möglich, die Plattform auf eine sinnvolle Art und Weise zu testen. Beispielsweise kann man die Ergebnisse eines Filters, der die Anzahl der Fahrspuren ermittelt nur genau überprüfen, wenn der Testumgebung diese Daten für jeden Zeitpunkt des Tests bereitgestellt werden.

3.2 ADTF SDK

Eine große Hürde für Einsteiger ist das SDK, welches nur ein paar einfache Beispielprogramme und eine unvollständige Dokumentation mitbringt. Es ist viel Geduld und Eigeninitiative erforderlich, um sich in dieses SDK einzuarbeiten.

3.3 ADTF Konfigurationen

Für das Laden der Konfigurationen des interactIVe Projekts in ADTF müssen diverse Abhängigkeiten aufgelöst werden, was je nach verwendetem Betriebssystem unterschiedlich kompliziert ist. Für einige Konfigurationen sind die Abhängigkeiten zudem nur unter Microsoft Windows zu erfüllen, da proprietäre Bibliotheken verwendet werden, die nicht für alle Betriebssysteme erhältlich sind.

3.4 Standard Ausgabe

Die Ausgaben, die von ADTF getätigt werden, verhindern eine Sinnvolle Nutzung des Terminals, da dieses mit Meldungen überschüttet wird. ADTF erlaubt es nicht dieses Verhalten zu deaktivieren. Später werden Klassen vorgestellt, die es dennoch unterbinden können, damit das Terminal wieder verwendbar ist.

3.5 Testen ohne Quelltext

Innerhalb des Projekts interactIVe werden keine Quelltexte geteilt, sondern nur fertig übersetzte Binärdateien. Dies verkompliziert das Testen um ein Vielfaches. Stünden die Quelltexte der anderen Projektpartner zur Verfügung, könnten Fehler in Modulen viel schneller identifiziert und beseitigt werden.

3.6 Werkzeuge

Um das bestmögliche Ergebnis zu erzielen wurden folgende Werkzeuge und Techniken auf Ihre Tauglichkeit und Kompatibilität mit diesem Projekt getestet und gegebenenfalls produktiv eingesetzt.

- **CMake**
  CMake erzeugt aus Skriptdateien Makefiles und Projekte für integrierte Entwicklungsumgebungen. Dabei werden Abhängigkeiten automatisch überprüft.

- **Doxygen**
  Doxygen ist ein Software-Dokumentationswerkzeug, um aus Quelltext und Kommentaren eine Dokumentation zu erstellen. Es wurde verwendet um automatisch eine HTML und LaTeX Dokumentation für die libadtftest zu generieren.

- **Microsoft Visual Studio**
  Das Visual Studio von Microsoft ist eine integrierte Entwicklungsumgebung für Microsoft Windows. Es wurde für alle windows-spezifischen Aufgaben dieses Projekts verwendet. Während die Version 2008 einwandfrei mit ADTF funktioniert, muss man sich bei der Version 2010 mit Linker-Fehlern auseinandersetzen. Die aktuelle Version 2012 wurde nicht getestet.

- **Minimalist GNU für Windows**
  Mit MinGW ist es möglich, Windows Programme auf fremden Betriebssystemen zu erzeugen. Es wurde erfolglos versucht, unter Linux gegen ADTF Bibliotheken für Microsoft Windows zu linken, was weitere Nachforschungen in diese Richtung unterband.

- **Netbeans**
  Netbeans ist eine plattformunabhängige Entwicklungsumgebung, die hauptsächlich für die Programmiersprache Java genutzt wird, aber auch eine sehr gute C/C++ Unterstützung liefert.

- **Visual Paradigm for UML**
  Für die Übersichtlichkeit eines Projektes ist es wichtig, den aktuellen Stand als UML-Diagramm (Abbildung 10) vorzuhalten. Das UML-Diagramm sollte mit dem Quelltext möglichst ohne zusätzlichen Aufwand synchronisiert werden. Hierfür bietet sich Visual Paradigm for UML von der gleichnamigen Firma an, welches Code-Rountrips ermöglicht. Mithilfe des Code-Roundtrip-Engineerings wird eine ständige Konsistenz zwischen Quelltext und UML-Diagramm gewährleistet.

- **Wine**
  Wine ist eine Ausführungsumgebung für Windows Programme. Solche lassen sich mithilfe von Wine unter Betriebssystemen wie Linux ausführen. Sowohl ADTF, als auch die Beispielanwendungen dieser Arbeit, die gegen die libadtftest gelinkt wurden, ließen sich unter Wine ausführen.

4 Implementierung

Die libadtftest ist eine Klassenbibliothek mit folgendem Funktionsumfang:

- ADTF Konfigurationen können erstellt, geladen und modifiziert werden.
- Der Datenverkehr an Pins kann mithilfe des Interfaces `IPinTrigger` überwacht werden.
- Bestehende Filter können um einen Debug-Pin erweitert werden, um zur Laufzeit bessere Debugging-Möglichkeiten zu haben (`IDebugFilter`)

Durch die vielfältige Funktionsweise der libadtftest ist eine Klassifizierung der Testart nicht eindeutig möglich. Der Umgang mit ADTF Konfigurationen und das Überwachen von Pins stellt ein Black-Box-Testverfahren dar. Das Debuggen innerhalb eines Filters, welches zur Laufzeit von der libadtftest ausgewertet wird, kann man als White-Box-Testverfahren bezeichnen.

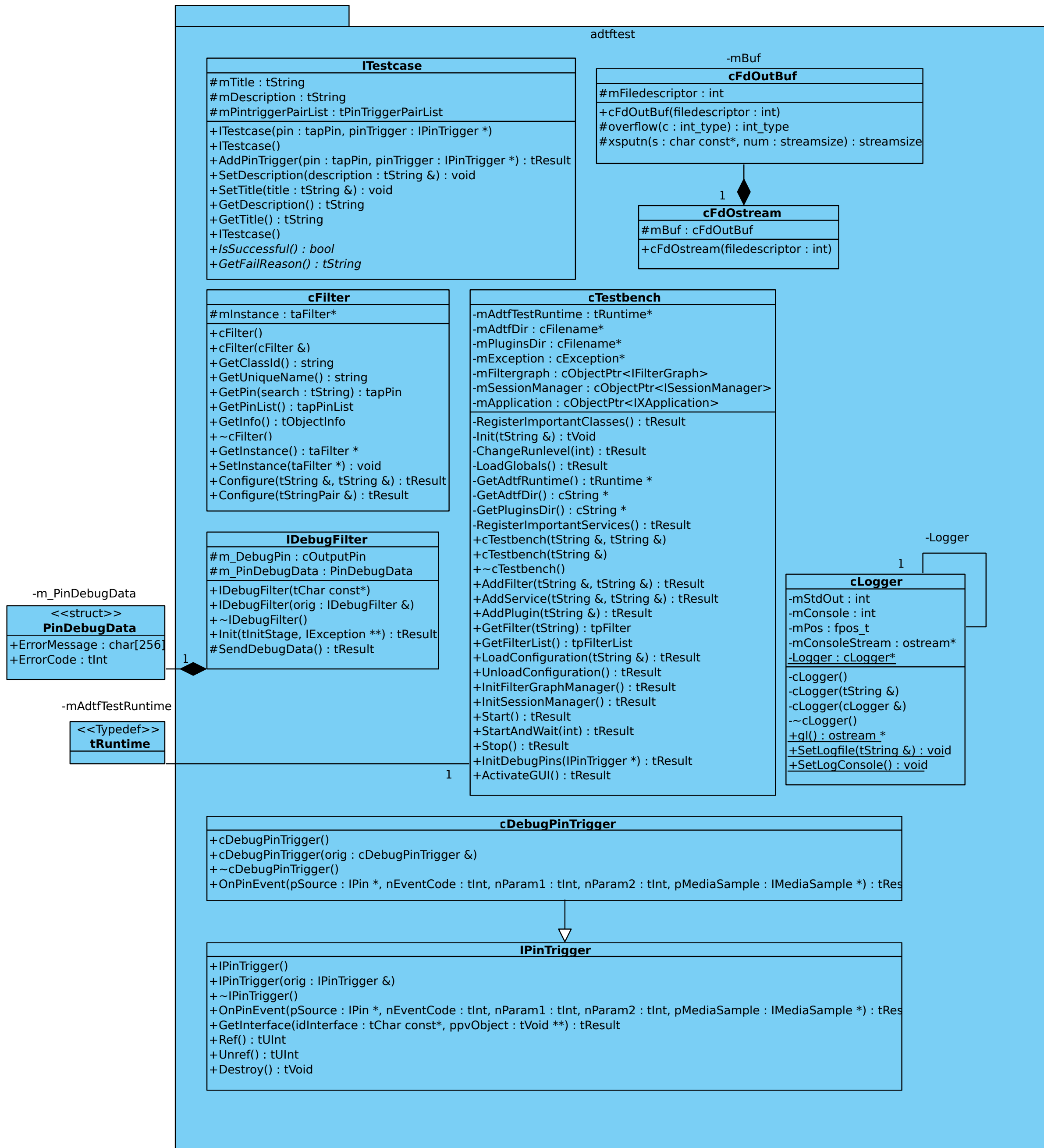

Abbildung 10: UML-Diagramm der libadtftest

4.1 Klassenbibliothek

Die Interfaces und Klassen der libadtftest werden in diesem Abschnitt vorgestellt.

4.1.1 Interfaces

- `IPinTrigger`

IPinTrigger
+IPinTrigger() +IPinTrigger(orig : IPinTrigger &) +~IPinTrigger() +OnPinEvent(pSource : IPin *, nEventCode : tInt, nParam1 : tInt, nParam2 : tInt, pMediaSample : IMediaSample *) : tRes +GetInterface(idInterface : tChar const*, ppvObject : tVoid **) : tResult +Ref() : tUInt +Unref() : tUInt +Destroy() : tVoid

Abbildung 11: Das Interface IPinTrigger

Das Interface `IPinTrigger` erbt von den Interfaces `adtf::IPinEventSink` und `ucom::IObject`. Um auf Pin-Events zu reagieren, muss man das Interface `IPinTrigger` implementieren und die Methode `OnPinEvent` überschreiben. Mit dieser Technik ist es möglich, an beliebigen Pins die Mediasamples abzufangen.

- `ITestcase`

ITestcase
#mTitle : tString #mDescription : tString #mPintriggerPairList : tPinTriggerPairList
+ITestcase(pin : tapPin, pinTrigger : IPinTrigger *) +ITestcase() +AddPinTrigger(pin : tapPin, pinTrigger : IPinTrigger *) : tResult +SetDescription(description : tString &) : void +SetTitle(title : tString &) : void +GetDescription() : tString +GetTitle() : tString +ITestcase() *+IsSuccessful() : bool* *+GetFailReason() : tString*

Abbildung 12: Das Interface ITestcase

Das Interface `ITestcase` ist die Grundlage für jeden Testfall. Es verwaltet die Assoziationen zwischen Pins und Pin-Trigger für die jeweilige Umgebung. Zudem stellt es Methoden zur Identifizierung und Validierung bereit. Es ist möglich, Testfälle mit Namen und Beschreibung zu versehen. Nachdem der Testfall abgeschlossen

ist, kann man das Ergebnis (`IsSuccessful`) und eventuelle weitere Informationen abrufen (`GetFailReason`).

- `IDebugFilter`

IDebugFilter
#m_DebugPin : cOutputPin
#m_PinDebugData : PinDebugData
+IDebugFilter(tChar const*)
+IDebugFilter(orig : IDebugFilter &)
+~IDebugFilter()
+Init(tInitStage, IException **) : tResult
#SendDebugData() : tResult

Abbildung 13: Das Interface IDebugFilter

Während des Testens mit der libadtftest ist eine direkte Kommunikation mit den Filtern nicht möglich, da diese als Plugins geladen werden und in verschiedenen Threads laufen. Alle Ausnahmebehandlungen innerhalb eines Filters können von außen nicht wahrgenommen werden. Steht der Quellcode eines Filters zur Verfügung, kann man dies bedingt umgehen, indem der Filter optional das Interface IDebugFilter implementiert. Dies führt dazu, dass dieses einen zusätzlichen Ausgabepin erhält, über den Informationen gesendet werden können, die der Testbench auswertet. Über die Methode `cTestbench::InitDebugPins` können alle in der Konfiguration vorhandenen Debug-Pins einem Pin-Trigger zugeordnet werden.

4.1.2 Klassen

- `cFilter`

cFilter
#mInstance : taFilter*
+cFilter() +cFilter(cFilter &) +GetClassId() : string +GetUniqueName() : string +GetPin(search : tString) : tapPin +GetPinList() : tapPinList +GetInfo() : tObjectInfo +~cFilter() +GetInstance() : taFilter * +SetInstance(taFilter *) : void +Configure(tString &, tString &) : tResult +Configure(tStringPair &) : tResult

Abbildung 14: Die Klasse cFilter

Die Klasse `cFilter` kapselt das Interface `adtf::IFilter` und stellt Methoden bereit, um sowohl auf die Informationen vom Typ `ucom::IObjectInfo::tObjectInfo` des Filters, als auch auf seine Pins zuzugreifen. Des Weiteren können die Properties des Filters gesetzt werden.

- `cTestbench`

cTestbench

-mAdtfTestRuntime : tRuntime*
-mAdtfDir : cFilename*
-mPluginsDir : cFilename*
-mException : cException*
-mFiltergraph : cObjectPtr<IFilterGraph>
-mSessionManager : cObjectPtr<ISessionManager>
-mApplication : cObjectPtr<IXApplication>

-RegisterImportantClasses() : tResult
-Init(tString &) : tVoid
-ChangeRunlevel(int) : tResult
-LoadGlobals() : tResult
-GetAdtfRuntime() : tRuntime *
-GetAdtfDir() : cString *
-GetPluginsDir() : cString *
-RegisterImportantServices() : tResult
+cTestbench(tString &, tString &)
+cTestbench(tString &)
+~cTestbench()
+AddFilter(tString &, tString &) : tResult
+AddService(tString &, tString &) : tResult
+AddPlugin(tString &) : tResult
+GetFilter(tString) : tpFilter
+GetFilterList() : tpFilterList
+LoadConfiguration(tString &) : tResult
+UnloadConfiguration() : tResult
+InitFilterGraphManager() : tResult
+InitSessionManager() : tResult
+Start() : tResult
+StartAndWait(int) : tResult
+Stop() : tResult
+InitDebugPins(IPinTrigger *) : tResult
+ActivateGUI() : tResult

Abbildung 15: Die Klasse cTestbench

Die Klasse `cTestbench` ist das Kernelement der Bibliothek. Sie verwaltet die ADTF Instanz und bietet eine einfach zu bedienende Schnittstelle zu allen benötigten ADTF Funktionalitäten.

- `cLogger`

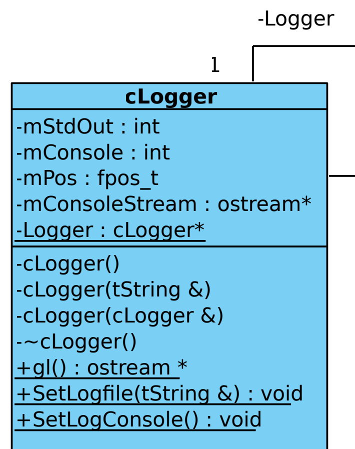

Abbildung 16: Die Klasse cLogger

Das ADTF Framework ist sehr verbos auf der Konsole. Um dies zu unterbinden, kann man die statischen Methoden der Klasse `cLogger` nutzen. Sie leitet mithilfe der beiden Klassen `cFdOstream` und `cFdOutBuf` die Standardausgabe in eine Datei um und stellt eine Schnittstelle zur Konsole bereit. Da die Funktionalität im Testbench gekapselt ist, braucht sich ein Benutzer der libadtftest in der Regel nicht mit dieser auseinanderzusetzen.

- `cDebugPinTrigger`

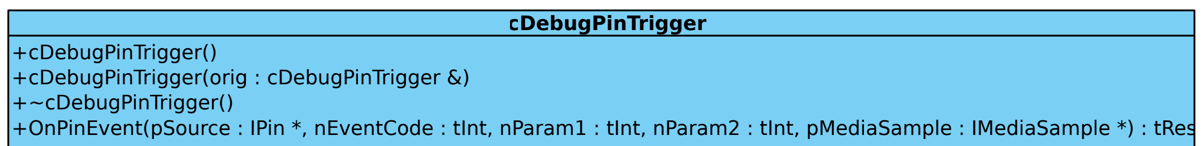

Abbildung 17: Die Klasse cDebugPinTrigger

Der `cDebugPinTrigger` ist eine eigens für den `IDebugFilter` entworfene Implementierung des Interfaces `IPinTrigger`. Wird der `cDebugPinTrigger` in der `cTestbench::InitDebugPins` genutzt, werden alle Meldungen, die über die Debugpins gesendet werden, automatisch auf die Konsole geschrieben.

4.2 Beispielhafte Anwendung

Eine Klassenbibliothek ist nur sinnvoll, wenn sie ausreichend dokumentiert ist und durch Beispiele die Funktionalität und ihre Eigenheiten erklärt werden. Mit der libadtftest wird ein Beispielprogramm ausgeliefert, welches für Einsteiger eine sehr gute Ergänzung zu der beiliegenden Dokumentation darstellt. Das folgende Kapitel erläutert das Testframework und dessen Funktionalität anhand von mehreren Beispielen und zeigt die verschiedenen Einsatzmöglichkeiten. In diesem Beispiel wird eine leere ADTF Konfiguration geladen, die manuell mit Filtern bestückt wird. Alternativ ist es möglich, mithilfe der GUI von ADTF die Konfiguration vorher zu erstellen. Es werden zwei Filter genutzt, die über ihre Pins Daten des Typs `tTeststruct` (Listing 3) austauschen. Abbildung 18 zeigt die fertige Konfiguration.

Listing 3: tTeststruct Definition

```
1 typedef struct Teststruct {
2     char Name[256]; /**< Name value */
3     float Width; /**< Width value */
4     float Length; /**< Length value */
5     float Height; /**< Height value */
6 } tTeststruct;
```

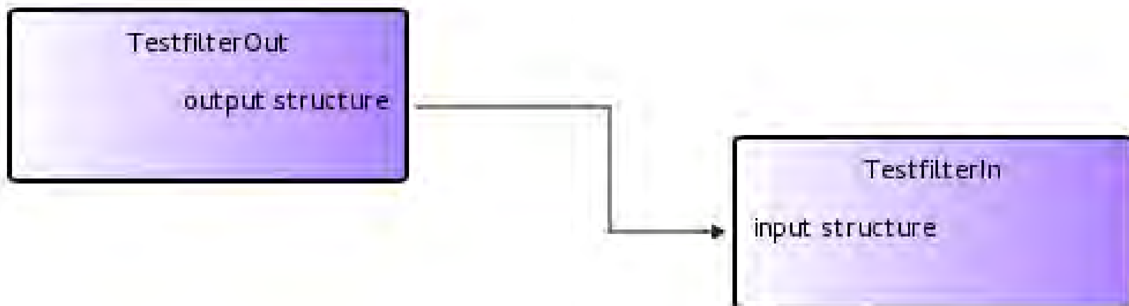

Abbildung 18: ADTF Testfilter Konfiguration

4.2.1 ITestcase und IPintrigger Implementierung

Die einfachste Methode mithilfe eines Pin-Triggers einen Testfall zu erstellen ist, innerhalb derselben Klasse die Interfaces `IPintrigger` und `ITestcase` zu implementieren. Hierzu erstellen wir eine Klasse mit dem Namen `cAdtfExampleTestCase` und folgendem Header:

Listing 4: cAdtfExampleTestcase Header

```
#ifndef CADTFEXAMPLETESTCASE_H
#define CADTFEXAMPLETESTCASE_H

#include "stdafx.h"
#include "../types.h"
#include "ITestcase.h"

class cAdtfExampleTestcase : public adtftest::IPinTrigger, public adtftest::ITestcase {
private:
    tTeststruct mExpectedData;
    tTeststruct mLastReceived;
public:
    cAdtfExampleTestcase(tapPin pin, tTeststruct &expectedData);
    cAdtfExampleTestcase(const cAdtfExampleTestcase& orig);
    virtual ~cAdtfExampleTestcase();
    virtual tBool IsSuccessful();
    virtual tString GetFailReason();
    virtual tResult OnPinEvent(IPin* pSource, tInt nEventCode, tInt nParam1,
                               tInt nParam2, IMediaSample* pMediaSample);
};

#endif /* CADTFEXAMPLETESTCASE_H */
```

Es werden zwei Member-Variablen benötigt:

- `mExpectedData`
  Der Erwartungswert für diesen Test, welcher im Konstruktor mit dem zugehörigen Pin übergeben wird
- `mLastReceived`
  Das zuletzt abgefangene Mediasample

Zusätzlich müssen mindestens die drei folgenden Methoden implementiert werden:

- Listing 5 zeigt die Methode `cAdtfExampleTestcase::OnPinEvent`, welche für jedes Pin-Event an dem entsprechenden Pin aufgerufen wird. Wurde ein Mediasample empfangen, wird der Inhalt in die Member-Variable `mLastReceived` kopiert.

Listing 5: Die Methode OnPinEvent

```
tResult cAdtfExampleTestcase::OnPinEvent(IPin* pSource, tInt nEventCode,
        tInt nParam1, tInt nParam2, IMediaSample* pMediaSample) {
    if(nEventCode == PE_MediaSampleReceived) {
        RETURN_IF_POINTER_NULL(pMediaSample);
        pMediaSample->CopyBufferTo(&mLastReceived, sizeof(tTeststruct), 0, 0);
    }
    RETURN_NOERROR;
}
```

- Listing 6 zeigt die Methode `cAdtfExampleTestcase::IsSuccessful`, welche die erwarteten Daten mit den zuletzt abgefangenen vergleicht. Ihre Rückgabe besteht aus einem Wahrheitswert (`tBool`), der den Erfolg des Tests widerspiegelt.

Listing 6: Die Methode IsSuccessful

```
tBool cAdtfExampleTestcase::IsSuccessful() {
    return mExpectedData.Height == mLastReceived.Height
        && mExpectedData.Length == mLastReceived.Length
        && mExpectedData.Width == mLastReceived.Width
        && !strcmp(mExpectedData.Name, mLastReceived.Name);
}
```

- Listing 7 zeigt die Methode `cAdtfExampleTestcase::GetFailReason`, die eine Zeichenkette (`tString`) generiert, welche beim Fehlschlagen des Tests Rückschlüsse auf dessen Ursache ziehen lässt.

Listing 7: Die Methode GetFailReason

```
tString cAdtfExampleTestcase::GetFailReason() {
    if(IsSuccessful()) return "No error";
    else return "Unexpected data";
}
```

4.2.2 IDebugFilter Implementierung

Sofern der Quelltext des genutzten Filters zur Verfügung steht, bietet das Interface `IDebugFilter` optional die Möglichkeit, aus dem Filter heraus mit der libadtftest zu kommunizieren. Der Filter muss lediglich statt von der Klasse `adtf::cFilter` zu erben das Interface `IDebugFilter` implementieren und während der Initialisierungsphase die Methode `IDebugFilter::Init` aufrufen. Nun lässt sich die Member-Variable `m_PinDebugData` mit Daten füllen und per `this->SendDebugData()` über den Debug-Pin senden. Ein laufender Testbench ist dann in der Lage, auf diese gesendeten Daten zu reagieren. Abbildung 19 zeigt die Konfiguration aus Abbildung 18 mit zusätzlichen Debug-Pins.

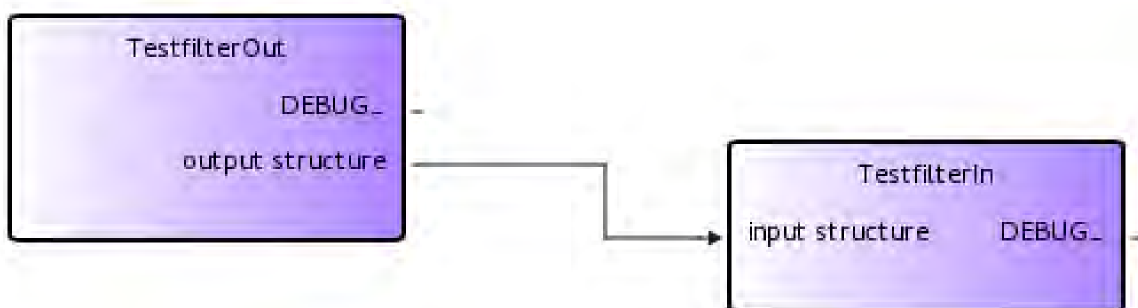

Abbildung 19: ADTF Testfilter Konfiguration mit Debug-Pins

4.2.3 Der Test

Sind alle Vorbereitungen getroffen, kann das eigentliche Testen beginnen. Als Erstes wird eine Instanz der Klasse `cTestbench` erstellt. `ADTFDIR` ist eine Zeichenkette, welche den Pfad zur lokalen ADTF Installation enthält.

```
tString adtfdir = ADTFDIR;
cTestbench *pTestbench = new cTestbench(adtfdir);
```

Danach wird die leere Konfiguration geladen und mit den benötigten Filtern bestückt.

```
testbench->LoadConfiguration(config);
testbench->AddFilter("adtf.forwiss.testfilterin", "TestfilterIn");
testbench->AddFilter("adtf.forwiss.testfilterout", "TestfilterOut");
```

Nun können optional alle Filter, die das `IDebugFilter` Interface implementieren, mit einem Pin-Trigger ausgestattet werden, um auf die Debugausgaben des Filters zu reagieren. Die libadtftest enthält zu diesem Zweck einen Pin-Trigger, der auf die Konsole loggt. Dieser wird hier verwendet.

```
cDebugPinTrigger debugPinTrigger;
testbench->InitDebugPins(&debugPinTrigger);
```

Danach wird eine Instanz vom Typ `tTeststruct` erstellt und mit Erwartungswerten gefüllt.

```
tTeststruct expectedData;
expectedData.Height = 300;
expectedData.Length = 100;
expectedData.Width = 200;
std::sprintf(expectedData.Name, "%s", "Testfilter");
```

Als nächstes werden Zeiger auf die benötigten Filter und Pins erzeugt, um eine Verbindung zwischen diesen herzustellen.

```
cFilter *outfilter = testbench->GetFilter("TestfilterOut");
cFilter *infilter = testbench->GetFilter("TestfilterIn");
ucom::cObjectPtr<adtf::IPin> outpin = outfilter->GetPin("output structure");
ucom::cObjectPtr<adtf::IPin> inpin = infilter->GetPin("input structure");
outpin->Connect(inpin);
```

Jetzt wird die Instanz von dem im Kapitel 4.2.1 erstellten `cAdtfExampleTestcase` erstellt. Diese benötigt im Konstruktor zum einen den Pin, an dem Daten abgefangen werden sollen, und zum anderen den Erwartungswert `expectedData`.

```
cAdtfExampleTestcase *t = new cAdtfExampleTestcase(inpin, expectedData);
t->SetTitle("Expected data");
```

Nun wird der Testbench gestartet.

```
testbench->Start();
```

Als Letztes werden die Testergebnisse auf die Konsole geschrieben und Aufräumarbeiten durchgeführt.

```
1 if (testcase->IsSuccessful()) {
2     LOG << testcase->GetTitle() << ":␣success" << std::endl;
3 } else {
4     LOG << testcase->GetTitle() << ":␣" << testcase->GetFailReason() << std::endl;
5 }
6 testbench->Stop();
7 delete testcase;
8 delete testbench;
```

4.3 Mögliche Anwendung in interactIVe

Zur Veranschaulichung, wie das ganze in der Praxis aussehen kann, folgen jetzt mögliche Anwendungsfälle, welche im Rahmen von interactIVe denkbar sind.

4.3.1 Speicherleck in einem Filter

Die in interactIVe verwendeten Filter stammen von verschiedenen Projektpartnern. Alle Filter liegen nicht als Quelltext, sondern nur in Binärform vor. Ein Debugging ohne solide Assembler-Erfahrung ist somit nicht möglich. Insgesamt handelt es sich um mehr als 30 Filter, die in einer aktiven Konfiguration geladen werden. In einer Version der Plattform wurde ein Speicherleck gefunden, was schließlich zum Absturz der gesamten Plattform führte. Daraufhin wurden sukzessive einzelne Filter manuell aus der Konfiguration entfernt, um die Quelle zu identifizieren. Dies bedeutet aber, dass man eine Vielzahl an Konfigurationen erstellen, ausführen und beobachten muss. Dieser Aufwand ließe sich mit Unterstützung der libadtftest durch eine einfache Schleife erheblich reduzieren, indem aus der Filterliste jeweils ein Filter deaktiviert wird.

4.3.2 Ausfall von Sensoren

Eine wichtige Anforderung an ein Perzeptionssystem besteht darin, dass die Plattform auch bei Ausfall eines Sensors weiterarbeiten kann. Dies lässt sich mit der libadtftest ebenfalls testen, indem manipulativ in den Datenfluss eingegriffen wird. So kann man bewusst fehlerhafte Daten senden, oder Verbindungen zwischen Filtern unterbrechen, um den Ausfall eines Teils der Plattform zu simulieren. Das Road-Edge-Detection Filter des Projekts interactIVe enthält Funktionen zur Selbstkontrolle. Man kann abrufen, an welchen Eingängen Daten empfangen wurden. Die Live-Demonstration dieser Arbeit zeigt einen Testfall mit der libadtftest, welcher überprüft, ob diese Funktion einwandfrei arbeitet. Hierzu wird eine Konfiguration mit dem Filter geladen und eine Verbindung an einem der Eingabepins getrennt. Dies wird für jeden Pin des Filters wiederholt. Listing 8 Zeigt die Ausgabe des Programms.

Listing 8: Die Ausgabe des Live-Demo Programms

```
-----------------------------------------------------------------
Leaving configuration untouched
All Pins are receiving data
-----------------------------------------------------------------
-----------------------------------------------------------------
Disconnecting Pin: Sensor Conf
=====================
camera: no data
video: no data
radar: no data
vsf: no data
=====================
-----------------------------------------------------------------
-----------------------------------------------------------------
Disconnecting Pin: PH Conf
All Pins are receiving data
-----------------------------------------------------------------
-----------------------------------------------------------------
Disconnecting Pin: VSF
=====================
camera: data
video: no data
radar: data
vsf: data
=====================
-----------------------------------------------------------------
-----------------------------------------------------------------
Disconnecting Pin: Video_In
=====================
camera: no data
video: no data
radar: no data
vsf: no data
=====================
-----------------------------------------------------------------
-----------------------------------------------------------------
Disconnecting Pin: RadarFront
=====================
camera: data
video: data
radar: no data
vsf: data
=====================
-----------------------------------------------------------------
```

4.3.3 Qualitätscheck

Da noch keine Referenzdaten zur Verfügung stehen, beruht die Güte und Qualität der Perzeption ausschließlich auf einer visuellen Einschätzung, welche aber einer objektiven Beurteilung des Systems nicht genügt. In der Praxis heißt das, dass Sensordaten und Fusionsergebnis über mehrere Fenster und Zeitspannen hinweg vom Tester im Auge behalten werden müssen, was sich als ungeeignet für eine Beurteilung der Ergebnisse darstellt. Dieser Umstand lässt sich ebenfalls mit der libadtftest beseitigen, indem beispielsweise im Falle einer Objekterkennung die eingehenden Sensordaten mit dem Ergebnis der Plattform assoziiert und verglichen werden.

Dies wird im folgenden Beispiel näher beschrieben:

Beispiel 1. Liefert der Kamera Sensor ein Objekt mit den Koordinaten $obj_1 = (44.0, 3.0)$ und der Radar Sensor ein Objekt mit den Koordinaten $obj_2 = (43.4, 2.9)$, so wird vom System erwartet, dass die Fusion beider Messwerte innerhalb einer Toleranzschwelle t liegt, also $d(obj_1, obj_2) < t$ gefordert wird, wobei $d(x, y)$ den euklidischen Abstand darstellt.

5 Ausblick

In der Idee, eine Testumgebung auf Applikationsebene zu schaffen, steckt eine Menge Potential. Im Laufe der Entwicklung entstanden Erweiterungsideen, die aus Zeitgründen nicht umgesetzt werden konnten. Zudem sind durch das ADTF SDK Einschränkungen gegeben, die in einer der nächsten Versionen aufgehoben sein können.

5.1 Automatisierung

Durch Erweiterung der libadtftest könnte es möglich gemacht werden, bestimmte Abläufe zu automatisieren, indem beispielsweise kompatible Pins automatisch miteinander verbunden werden, sofern dies eindeutig möglich ist.

5.2 Fehlerbehandlung

Die Fehler- und Ausnahmebehandlung von ADTF ist nicht besonders intuitiv und sehr unkomfortabel. Eine Kapselung dieser ist wünschenswert. Dies würde zu einem komfortableren Arbeiten führen und gegebenenfalls auch das Aufspüren von Fehlern in Konfigurationen erleichtern.

5.3 Grafische Testergebnisse

Darüber hinaus könnten Testergebnisse grafisch aufbereitet werden. Hierzu sind Formate wie HTML oder LaTeX wünschenswert. Die Liste der Formate lässt sich verlängern.

5.4 Konfigurationsmanagement

ADTF erlaubt es nicht, Konfigurationen von Grund auf über das SDK zu erstellen. In Kapitel 3 wurde beschrieben, wie dieses Problem umgangen werden kann, indem eine leere Konfiguration geladen wird.

5.5 Testsuite GUI

Anhand der minimalen Qt-Unterstützung der libadtftest lässt sich erkennen, dass es möglich ist, ein eigenes GUI mit ADTF-Funktionalität zu entwerfen. Für viele Anwender ist es wünschenswert, eine grafische Testumgebung zur Verfügung zu haben. Damit wäre es möglich, Debug-Ausgaben und Datenverkehr der Pins grafisch darzustellen und Testfälle übersichtlich zu verwalten.

5.6 Vordefinierte Testfälle

Für bestimmte Filter und Szenarien könnten vordefinierte Testfälle sinnvoll sein. Würde die libadtftest mit vordefinierten Tests ausgeliefert, könnte somit der Arbeitsaufwand der Testteams reduziert werden.

Glossar

ADTF Automotive Data and Time-Triggered Framework
ADTF ist ein flexibles Werkzeug um neue Funktionen für Kraftfahrzeuge zu entwickeln. . 2, 10–12, 20, 23–25, 30–32, 34, 35, 39, 41, 42, 44

Black-Box-Testverfahren Black-Box-Test bezeichnet eine Art und Weise Software zu testen, bei der man keine Kenntnisse über das zu testende System besitzt. Der Quelltext steht bei dieser Art von Test nicht zur Verfügung. . 16, 25, 41

CAN-Bus Das Controller Area Network ist ein asynchrones, serielles Bussystem, welches zur internen Kommunikation in einem Kraftfahrzeug entwickelt wurde. . 10

CRF Centro Ricerche Fiat SCPA ist ein Partner im Projekt interactIVe. . 9

Delphi Delphi Delco Electronics Europe GmbH ist ein Partner im Projekt interactIVe. . 9

elektrobit Entwickler von ADTF (`http://www.elektrobit.com/`) . 2, 10

FAMOS Galileo for Future Automotive Systems
FAMOS ist ein Forschungsprojekt mit Konzentration auf die Entwicklung dreier Fahrerassistenzsysteme. (`http://www.famos-project.eu/`) . 5, 41

Ford Ford ist ein Partner im Projekt interactIVe. . 9

FORWISS Institut für Softwaresysteme in technischen Anwendungen der Informatik an der Universität Passau . 8, 42

Global Positioning System Das Global Positioning System ist ein weltweites, satellitengestütztes System zur Ortsbestimmung. . 41

GPS Global Positioning System. 6

Grey-Box-Testverfahren Grey-Box-Tests verbinden die Vorteile von White-Box-Testverfahren und Black-Box-Testverfahren. Die Tests werden erstellt, bevor der eigentliche Quelltext geschrieben wird. . 14

ICCS Das Institute of Communications and Computer Systems ist ein Partner im Projekt interactIVe. . 9

IKA Die Rheinisch-Westfälische Technische Hochschule Aachen ist ein Partner im Projekt interactIVe. . 9

Intempora Entwickler von RTMaps (`http://www.intempora.com/`) . 10

interactIVe Accident avoidance by active intervention for Intelligent Vehicles Forschungsprojekt an dem das FORWISS Institut beteiligt ist [1] . 2, 8–12, 20, 23, 36, 41, 42

libadtftest Die libadtftest ist eine Klassenbibliothek, die im Rahmen dieser Arbeit entstanden ist. Sie ermöglicht das Erstellen von Tests auf Basis des ADTF SDK. . 20, 23–28, 31, 32, 34–37, 39, 40, 42, 44

Qt Qt ist eine betriebssystemunabhängige Klassenbibliothek für die Programmierung grafischer Benutzeroberflächen. . 10, 39

RTMaps RTMaps ist ein Komponenten-basiertes Framework für die schnelle Entwicklung von multimodalen Anwendungen. . 10, 41, 42

SDK Software Development Kit. 2, 10, 20, 23, 39, 42

Software Development Kit Ein Software Development Kit ist eine Sammlung von Werkzeugen und Anwendungen inklusive Dokumentation um eine Software zu entwickeln. . 10, 42

VTEC Die Volvo Technology AB ist ein Partner im Projekt interactIVe. . 9

VW Die Volkswagen AG ist ein Partner im Projekt interactIVe. . 9

White-Box-Testverfahren White-Box-Test, auch Glass-Box-Test genannt, bezeichnet eine Art und Weise Software zu testen, bei der man Kenntnisse über das zu testende System besitzt. Der Quelltext steht bei dieser Art von Test zur Verfügung. . 16, 25, 41

Quellenverzeichnis

[1] European Commission. Accident avoidance by active intervention for Intelligent Vehicles. http://www.interactive-ip.eu/, 10 2012.

[2] elektrobit. Driver assistance application and safety system development with ADTF. http://ww.automotive.elektrobit.com/home/driver-assistance-software/eb-assist-adtf.html, 10 2012.

[3] Elektrobit Automotive GmbH. EB Assist ADTF 2.7 Developer's Manual. Technical report, 2011.

[4] Elektrobit Automotive GmbH. EB Assist ADTF 2.7 User's Manual. Technical report, 2011.

[5] ITS Niedersachsen GmbH. Galileo for Future Automotive Systems. http://www.famos-project.eu/, 10 2012.

[6] interactIVe. Accident avoidance by active intervention for Intelligent Vehicles. Technical report, 2009.

[7] interactIVe. Deliverable D1.7: System Architecture and Updated Specifications. Technical report, 2011.

[8] interactIVe. Deliverable D7.3: Legal Aspects. Technical report, 01 2012.

[9] Maciej Jaros. Wasserfallmodell. http://de.wikipedia.org/w/index.php?title=Datei:Wasserfallmodell.svg, 2006.

[10] Linux New Media. Software testen. Linux Magazin, pages 22–24, 8 2012.

[11] Ian Sommerville. Software Engineering. Pearson Studium, 8 edition, 2007.

[12] VÖRBY. Der ganzheitliche Ansatz beim Testen. http://de.wikipedia.org/w/index.php?title=Datei:Test_ganzheitlich.png, 2011.

[13] Wikipedia. Ariane V88. http://de.wikipedia.org/wiki/Ariane_V88, 10 2012.

[14] Gunnar Wrobel. Präziser Antrieb. Linux Magazin, pages 32–37, 8 2012.

Abbildungsverzeichnis

1 Sensordatenfusion . . . 6
2 Perzeptionsschicht . . . 7
3 Architektur der Perzeptionsplattform . . . 8
4 Struktur von interactIVe . . . 9
5 ADTF Streaming Architektur . . . 11
6 ADTF GUI mit geladener Konfiguration . . . 12
7 Wasserfallmodell . . . 14
8 Der ganzheitliche Ansatz beim Testen . . . 15
9 Beispiel Checkliste aus SP7 . . . 19
10 UML-Diagramm der libadtftest . . . 26
11 Das Interface IPinTrigger . . . 27
12 Das Interface ITestcase . . . 27
13 Das Interface IDebugFilter . . . 28
14 Die Klasse cFilter . . . 29
15 Die Klasse cTestbench . . . 30
16 Die Klasse cLogger . . . 31
17 Die Klasse cDebugPinTrigger . . . 31
18 ADTF Testfilter Konfiguration . . . 32
19 ADTF Testfilter Konfiguration mit Debug-Pins . . . 34

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind sowie dass ich diese Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Lukas Elsner

5. Dezember 2012