Moderne Technologien netzwerkbasierter Laborautomatisierung

Dissertation

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

vorgelegt von
Dipl. Chem. Dominik Furin
aus Nürtingen

Tübingen
2016

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

Tag der mündlichen Qualifikation:	02.08.2016
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Günter Gauglitz
2. Berichterstatter:	Prof. Dr. Herbert Klaeren

Inhaltsverzeichnis

Alexander Graham Bell

Geh nicht immer auf dem vorgezeichneten Weg, der nur dahin führt, wo andere bereits gegangen sind.

1 Einführung

1.1. Motivation

Im Rahmen meiner Dissertation habe ich mich mit der Integration von analytischen Messsystemen beschäftigt. Bei Betrachtung der verschiedenen Systeme konnte ich dabei Gemeinsamkeiten bei der Entwicklung erkennen, vor allem hinsichtlich der Ablaufsteuerung. Konkret habe ich mich mit drei Anwendungsszenarien intensiv beschäftigt. Dabei handelt es sich um ein Fluoreszenzspektrometer, die reflektometrische Interferenzspektroskopie und ein Messsystem, das im EU geförderten Projekt INSTANT entwickelt worden ist. Die Aufgabenstellungen erscheinen auf den ersten Blick unterschiedlich, da es sich in einem Fall um ein einzelnes Gerät, im zweiten Fall um ein Messsystem mit mehreren Gerätekomponenten und im letzten Fall um ein komplexes Messsystem mit zwei unterschiedlichen Detektionsprinzipien handelt. Bei der Implementierung einer Ablaufsteuerung für diese Szenarien ist die Frage aufgekommen, warum der Softwareentwicklungsprozess für die Ablaufsteuerung, trotz vieler Gemeinsamkeiten zwischen den Szenarien, individuell für jedes Messsystem erfolgt. Auf der Suche nach kommerziellen Lösungen konnte ich keine allgemein nutzbaren Komponenten finden, um diese Entwicklungen zu vereinfachen ohne dabei Flexibilität einzuschränken. Bei diesen Lösungen wird immer ein fundiertes Fachwissen in Programmierung und Informatik benötigt oder die Lösung ist auf einen individuellen Fall zugeschnitten und nicht allgemein einsetzbar.
Die von mir entwickelte Lösung für die Integration von Laborsystemen greift diese Problematik auf. Sie bietet dem Geräteentwickler eine Plattform mit einfach zusammensetzbaren Softwarekomponenten, um die Entwicklung von Ablaufsteuerungen für ihn auf ein Minimum zu reduzieren. Dabei wird ein modulares Konzept sowohl für die Gerätekommunikation, die logische Abfolge von Befehlen und das Datenmanagement eingesetzt. Im besten Fall soll dabei kein Systemprogrammierer benötigt werden. Ein weiterer wichtiger Punkt ist der Einsatz von Steuerungssoftware über Computergrenzen hinweg. Moderne Softwarearchitekturen agieren nicht ausschließlich auf dem lokalen Computersystem, sondern bieten Möglichkeiten, die Administration über netzwerkbasierte Schnittstellen vorzunehmen. Aufgrund dieser Entwicklung ist damit zu rechnen, dass nicht nur die Administration über Netzwerke eine Rolle spielen, sondern es wird zukünftig die gesamte Kommunikation *zwischen* den Geräten darüber abgewickelt werden. Diese Punkte stellen den Grundstock für die vorliegende Arbeit dar.

1.2. Strukturierung der Arbeit

Die Thesis ist folgendermaßen strukturiert. In Abschnitt 1.3 wird kurz die Zielsetzung dieser Arbeit dargelegt. In Abschnitt 1.4 wird auf Probleme bei der Entwicklung von Ablaufsteuerungen, insbesondere auf deren Generalisierung eingegangen. Im darauf folgenden Abschnitt 1.5 werden dafür mögliche Lösungsansätze vorgeschlagen.
Um die Lösungsansätze zu überprüfen, werden sie im folgenden Abschnitt 2 aus Sicht des physikalisch-chemischen Labors beleuchtet. Dafür wird zu Beginn auf die theoretischen Grundlagen eingegangen und anschließend die Anforderungen an eine derartige Plattform erläutert.
Im darauf folgenden Abschnitt 3 wird der Lösungsansatz aus Sicht der Informatik beleuchtet. Auch hier werden zunächst die theoretischen Grundlagen besprochen. Anschließend werden Anforderungen in Form einer Bedarfsanalyse spezifiziert, daraus ein Design entwickelt, welches letztendlich implementiert werden soll.

1.3. Zielsetzung

Ziel dieser Arbeit ist es, die Abhängigkeiten zwischen der Steuerungssoftware und den Geräten aufzulösen.
Dabei soll eine Plattform entwickelt werden, welche unabhängig vom analytischen Aufbau eingesetzt und bei welcher jeglicher Aspekt eines analytischen Aufbaus geändert werden kann, ohne auch programmatische Änderungen im zugrunde liegenden Quelltext vornehmen zu müssen. Darüber hinaus soll die Plattform von Grund auf netzwerkbasiert sein, damit Gerätesteuerung, Geräteanbindung und die Benutzerinteraktion unabhängig vom Computersystem miteinander interagieren können.

1.4. Problemanalyse

1.4.1. Einführung in die Problematik

Betrachtet man die Integration von analytischen Laborsystemen und die Entwicklung von Gerätesteuerungen, stößt man häufig auf einige generelle Probleme. Darunter fallen fehlende Flexibilitäten bei Konfigurationsänderungen am Messsystem, vor allem wenn sich, wie es in Forschungslaboratorien häufig der Fall ist, Änderungen im Laboraufbau ergeben. Weitere sind die beschränkte Übertragbarkeit von Modulen zwischen verschiedenen Softwaresteuerungen und wiederholte Programmierung ähnlicher Funktionalitäten trotz ähnlichem Anwendungsfall. Dadurch resultiert ein hoher Zeit- bzw. Kostenbedarf und es wird versucht, diese durch Einsparungen in Qualität und Funktionsumfang, außerdem die Beschränkung auf lokale Administration und Steuerung, auszugleichen. Auf die genannten Probleme wird in diesem Kapitel detailliert eingegangen und auch der Frage nachgegangen, warum die Probleme nach wie vor in modernen Entwicklungen auftreten.

Exemplarisch werden dazu 3 Anwendungsszenarien betrachtet:

1. Ein Laborsystem, bei welchem die Steuerungssoftware nicht auf modernen Computersystemen lauffähig ist (am Beispiel der Fluoreszenzspektroskopie)
2. Ein Laborsystem, bei welchem eine Gerätekomponente ausgetauscht werden muss (am Beispiel der reflektometrischen Interferenzspektroskopie)
3. Ein Laborsystem, welches aus einem komplexen Zusammenspiel vieler Einzelkomponenten besteht und nur ein begrenzter zeitlicher Rahmen zur Verfügung steht (am Beispiel des EU-Projekts INSTANT)

Die Anwendungsszenarien ähneln sich dahingehend, dass für einen vorhandenen Laboraufbau bestehend aus mehreren Geräte-Einzelkomponenten, ein Gesamtsystem aus funktionierender Hardware und Steuerungssoftware entwickelt werden muss. Auf die Unterschiede soll nun genauer eingegangen werden.

Der erste Fall beschreibt konkret ein Spektrometer von der Firma PerkinElmer, welches zwar eine Steuerungssoftware mitbringt, die aber auf modernen Betriebssystemen nicht mehr lauffähig ist. Da sich moderne Betriebssystemarchitekturen schnell ändern, können ältere Softwarepakete nicht ohne Umstände oder ab einem gewissen Architektursprung (z.B. von 32-bit auf 64-bit Systeme) überhaupt nicht mehr portiert werden. Eine neue

Version der Software vom Gerätehersteller zu beziehen, ist dabei oft mit sehr hohen Kosten von mehreren Tausend Euro verbunden, oder sie ist aufgrund des Alters des zugrunde liegenden Geräts nicht mehr erhältlich.

Der zweite Fall behandelt ein RIfS-Analysensystem (s. Abschnitt 2.1.4), bei welchem ein Diodenzeilenspektrometer älterer Generation gegen ein moderneres Spektrometer ausgetauscht werden soll. Der Grund liegt darin, dass ersteres nicht mehr auf dem Markt erhältlich ist und auch keine Ersatzteile mehr zur Verfügung stehen. Weil die Steuerungssoftware vor vielen Jahren selbst entwickelt worden ist und das eingesetzte Spektrometer über komplexe binäre Gerätebefehle angesprochen werden muss, ist dieser Wechsel mit der aktuell eingesetzten Steuerungssoftware nicht möglich. Sie muss durch eine neue Software ersetzt werden, welche sowohl ein neuwertiges Spektrometer unterstützt, als auch das Alte aus Kompatibilitätsgründen.

Im letzten Fall soll für das EU-Projekt INSTANT eine Gerätesteuerung entwickelt werden. In diesem Projekt sollen zwei verschiedene Analysemethoden kombiniert und in einem übergreifenden Gesamtkonzept vereinigt werden. Beide Methoden sollen Ergebnisse liefern, die nur in Kombination den zugrunde liegenden Fall der quantitativen Analyse von Nanopartikeln in komplexen Matrices realisieren können. Eine weitere Schwierigkeit ist die begrenzte Projektlaufzeit gewesen, so dass die Entwicklung einer vollständig funktionsfähigen und verlässlichen Software aufgrund von internen Umstrukturierungen innerhalb von 4 Wochen realisiert werden sollte.

Steuerung aus Sicht des Analytikers

Betrachtet man ein Laborsystem bestehend aus Hardware und Steuerungssoftware, so sind beide Teile nicht unbedingt eine zusammengehörige Einheit, was häufig instinktiv angenommen wird. Während die Hardware für die Messung zuständig ist, wird von der Software die Datenaufbereitung und -speicherung übernommen. Handelt es sich um Laborsystem mit mehreren Einzelkomponenten, ist die Software auch für die Koordination der Ablaufsequenz verantwortlich, in welcher die einzelnen Komponenten Aktionen durchführen sollen. Für derartige Fragestellungen wäre es hilfreich, allgemein erhältliche Software-Komponenten oder Plattformen einsetzen und flexibel anpassbare Automationsszenarien modellieren zu können. Tatsächlich erfolgt nach wie vor der Softwareentwicklungsprozess für jedes Messsystem individuell und nicht universell, da Steuerungssoftware und Hardware nicht unabhängig voneinander betrachtet werden. Zum Einsatz für ein anderes Messsystem können zwar bereits entwickelte Module von

(selbst geschriebenen) Softwaresteuerungen anderer Systeme wiederverwendet werden, doch benötigt man hierfür ein tiefgehendes Wissen von Programmierung, Automatisierungstechnik und Informatik. Derartiges Wissen ist im benötigten Umfang oft nicht in der Ausbildung eines analytischer Chemikers zu finden, obwohl er die Funktionsweise und die zu erwartenden Resultate des vorliegenden Laborsystems exakt kennt und versteht, sogar dann, wenn er die Laborsysteme selbst integriert und entwickelt. Der Grund hierfür ist in der Komplexität selbst moderner Programmiersprachen zu suchen, welche nur ausgebildete Programmierer und Informatiker vollständig verstehen können.

Unterschiedliche Ebenen in der Laborautomation

Um die Problematik aktueller Softwaresteuerungen besser zu verstehen, soll ausgehend von obigen Szenarien der Automatisierungsprozess von Labors genauer betrachtet werden (s. Abbildung 1.1). Dieser kann in drei Ebenen unterteilt werden.

Peripherie-Ebene

Auf der untersten Ebene, der *Peripherie-Ebene*, befinden sich Geräte bzw. Gerätekomponenten, die an gängigen Computerschnittstellen angeschlossen sind. Sie werden über unterschiedlichste vom jeweiligen Gerät abhängige Befehlssätze gesteuert und führen Aktionen aus. Diese Geräte verfügen oft bereits über eine Steuerungslösung auf dem Gerät (in Form eines Touch-Panels oder einfachen Druck-Knöpfen). Zusätzlich können sie an ein normales Computersystem über gängige Schnittstellen wie RS232, USB, o.ä. angeschlossen werden und können anschließend über eine (meist mitgelieferte) vom Hersteller entwickelte, dedizierte Steuerungssoftware administriert werden. Diese Steuerungssoftware ist ausschließlich für das betrachtete Gerät (bzw. für die Produktpalette) des Herstellers anwendbar.

Anlagenebene

Auf der nächsthöheren Ebene, der *Anlagenebene*, werden mehrere Geräte (Komponenten) zu Messsystemen und Laboranlagen zusammengeschlossen. Steuerungssoftware für diese Messsysteme muss speziell entwickelt werden, da die integrierten Geräte oftmals von verschiedenen Herstellern stammen und die einzelnen Gerätesteuerungen der Peripherie-Ebene nicht universell für alle Geräte einsetzbar ist. Bislang werden in diesem Bereich gängige Speicherprogrammierbare Steuerungen (SPS) in Verbindung mit proprietärer Anwendungssoftware eingesetzt. Dabei werden die Geräte an die SPS-Systeme angeschlossen, eine Logik entwickelt und eine grafische Software zur Überwachung, Steuerung und das Datenmanagement entwickelt.

Laborebene

Zuletzt existiert die *Laborebene*, auf welcher mehrere Laboranlagen zu Automationsstra-

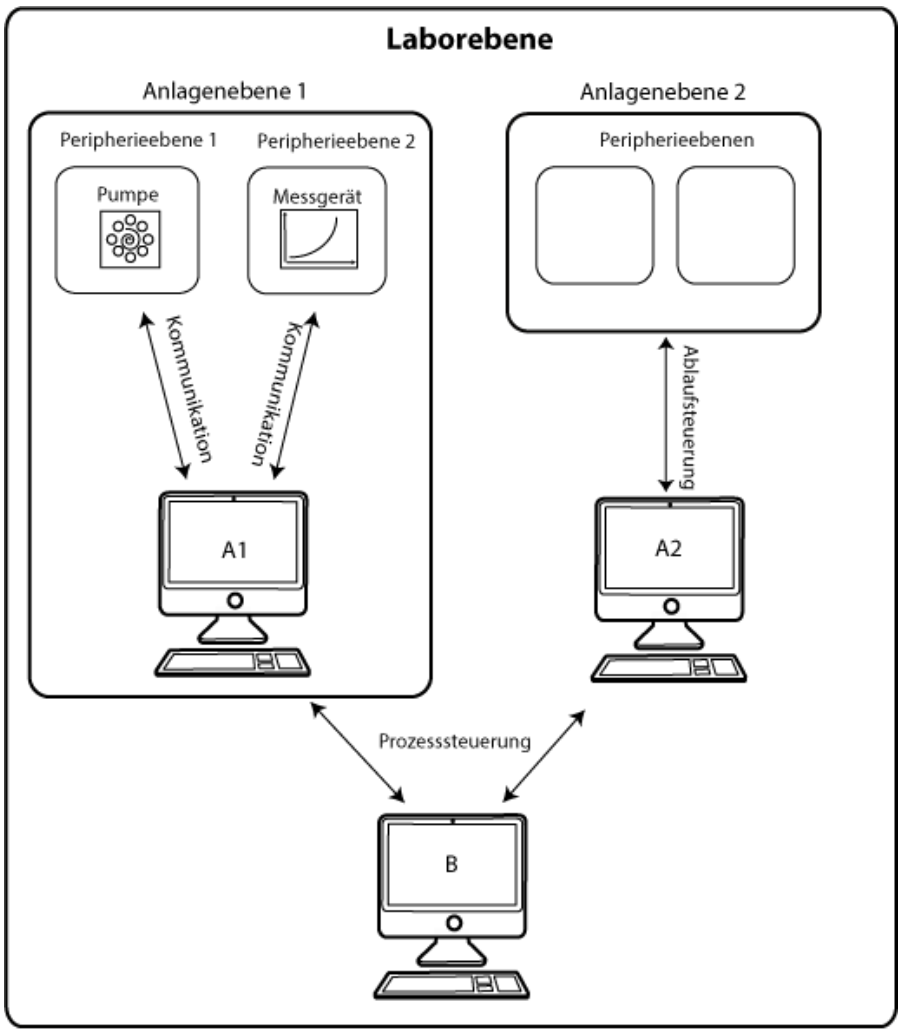


Abbildung (1.1)

Abstraktionslevels der Laborautomation bestehend aus der Peripherie-Ebene, in welcher verschiedene Gerätekomponenten (wie Pumpen, Messgeräte) von einem Computersystem über Schnittstellen (RS232, USB, usw.) oder üben einen integrierten Mikrocontroller gesteuert werden können. Befinden sich mehrere Geräte in einem Verbund können mit Hilfe einer Ablaufsequenz in einer bestimmten Reihenfolge Aktionen ausgeführt werden. In dieser Anlagenebene koordiniert ein Computersystem (oft dasselbe, welches auch die Geräteebene steuert) diesen Ablauf und sendet in der vorgegebenen Sequenz Steuerbefehle an die Geräteebene. Sollen mehrere Anlagen über Prozesse koordiniert oder administriert werden, spricht man von der Laborebene, bei welcher Prozesskontroll- oder direkt die LIMS-Systeme der Labors mit Anlagen über deren individuelle Computersysteme kommunizieren.

ßen oder vollautomatisierten Laboratorien verbunden werden. Auf dieser Ebene spielt die Interaktion der Anlagen untereinander eine große Rolle, darüber hinaus die Kommunikation der einzelnen Anlagen mit den in dieser Einrichtung verwendeten Labor-Informations-Management Systemen (LIMS). Es wird hier Software benötigt, die nicht die einzelnen Geräte, Komponenten oder Anlagen steuert, sondern die darüber liegenden Prozesse und Arbeitsabläufe administriert und die anfallenden Daten für die LIMS-Systeme aufbereitet. Für die Laborebene existieren universell einsetzbare Softwareplattformen[1], denn sie betrachten die Anlagensteuerung aus Sicht von Prozessen und Abläufen, die nur geringfügig von herstellerspezifischen Eigenheiten beeinflusst werden. Diese Ebene soll im Rahmen dieser Arbeit aber nicht betrachtet werden.

Betrachtung der Anwendungsbeispiele

Die zu Beginn des Kapitels gezeigten Anwendungsbeispiele befinden sich auf der Ebene der Peripherie (Fall 1) und der Anlagenebene (Fälle 2 und 3). Betrachtet man dazu Abbildung 1.1 handelt es sich im Fall des Fluoreszenz-Spektrometers um die Kombination aus Messgerät und Computersystem A1. Die Anlagenebene 1 dieser Abbildung kann stellvertretend für die reflektrometrische Interferenzspektroskopie und das INSTANT System angesehen werden.

Für die Peripherie-Ebene existieren im Normalfall steuerungstechnische Lösungen, weil die Gerätehersteller diese direkt mit dem verkauften analytischen Gerät ausliefern. Dennoch können hier Probleme auftreten, falls sich die Architekturen der Computersysteme über die Zeit ändern und die Steuerungssoftware nicht mehr funktionsfähig ist. In den meisten Fällen kann zwar über Aktualisierungen der Steuerungssoftware die Portierung der Geräte auf moderne Systeme gewährleistet werden, doch lassen sich die Unternehmen die Aktualisierungen häufig sehr gut bezahlen.

Auf der Anlagenebene kommt es zu mehreren Problemen. Dadurch dass noch immer Geräte bzw. Gerätekomponenten im Fokus stehen, spielen die Eigenheiten der einzelnen Hersteller und ihrer Produkte eine sehr große Rolle. Aus diesen und anderen Gründen wird die Steuerungssoftware nach wie vor auf den vorliegenden Anwendungsfall, d.h. den zu realisierenden (analytischen) Aufbau, zugeschnitten. Jedes zu entwickeln-

[1] Siehe unter anderem den Bosch Workflow Manager®

de Messsystem ist aus diesem Grund einzigartig, weshalb auch die eingesetzte Software einzigartig ist und nur auf dem System eingesetzt werden kann, für welches sie entwickelt worden ist. Natürlich gibt es verallgemeinerte Prozesse, um derartige Messsysteme zu entwickeln, die aber das Knowhow von Softwareproduzenten oder auf Laboranlagen spezialisierte Unternehmen darstellen. Sie haben hierfür Fachpersonal eingestellt, welche die innerbetrieblichen Prozesse und Verfahren genau kennen und nutzen können, solange die Anforderungen an das zu entwickelnde Messsystem mit jenen realisierbar sind. Eine standardisierte, über Herstellergrenzen hinweg einsetzbare Steuerungssoftware oder -plattform existiert nicht. Diese Art von Plattform wird aber benötigt, wenn die zu Beginn des Kapitels gezeigten Anwendungsszenarien gelöst werden sollen und das von Personal, welches dafür eigentlich nicht ausgebildet ist (Systemintegratoren, Naturwissenschaftlicher, usw). Der Grund für ein Fehler derartiger Technologie ist durch mehrere Probleme bedingt, auf die nun eingegangen werden sollen.

1.4.2. Problemidentifikation

Abhängigkeiten zwischen Aspekten einer Ablaufsteuerung

Unter den *Aspekten* sollen hier folgende unabhängige Teile eines Laboraufbaus verstanden werden:

1. Ein physikalisch(-chemischer) Effekt, welcher von einem Sensor erfasst wird
2. Die Peripherie bestehend aus Geräten zur Nutzung eines physikalisch(-chemischen) Effekts
3. Einer Ablaufsteuerung mit grafischer Benutzeroberfläche zur Überwachung des Laboraufbaus, einer logischen Ablaufsequenz und einer Datenauswertung

Bei der Entwicklung eines Laboraufbaus verfolgt man das Ziel, eine physikalische oder chemische Eigenschaft in ein nutzbares und auswertbares Signal zu übersetzen. Dies kann auf qualitative oder quantitative Weise geschehen. Die Detektion erfolgt über Sensoren (s. Abschnitt 2.1.3), welche die Signale erfassen und als elektrische Größe zur Verfügung stellen; letztere steht in einem mathematischen Zusammenhang mit der zu messenden Eigenschaft. Ein Sensor ist alleinstehend allerdings nur bedingt innerhalb eines bestimmten Laboraufbaus einsetzbar, da die elektrischen Ausgangsgrößen keinen direkten, sichtbaren Nutzen erfüllen. In anderen Worten, sie sollen auf anderen Geräten, wie beispielsweise

einer Zahlenanzeige oder einem Messgerät einen Effekt auslösen (Darstellung, Messung, usw.).
Ausgehend von diesen Grundannahmen kommt es zwischen den einzelnen Aspekten zu Abhängigkeiten, von welchen manche nach bisherigem Stand der Technik zwingend und nicht auflösbar sind. Andere kommen durch den Laboraufbau selbst und den Eigenschaften des Sensors zustande und können als „optional“ betrachtet werden. Letzteres kann durch die Tatsache erklärt werden, dass ein Sensor auch ohne Peripherie und Ablaufsteuerung funktionsfähig, wenn auch nur bedingt einsetzbar ist; auch der physikalische Effekt verliert seine Bedeutung nicht dadurch, dass keine Peripherie vorhanden ist. Dies hat zur Folge, dass die Abhängigkeiten optionaler Natur ausschließlich im Laboraufbau und dessen Komplexität begründet liegen.
Die obligatorischen Abhängigkeiten sind hingegen ungleich komplexer und auf die Natur der Ablaufsteuerung zurückzuführen. Aufgrund der Vielfältigkeit von sowohl Sensoren (physikalischen Effekten) als auch Peripheriegeräten ist diese Abhängigkeit verständlich, wodurch Ablaufsteuerungen einerseits auf einen Laboraufbau zugeschnitten werden müssen oder andererseits für einen gewissen Pool an gleichartigen Geräten (z.B. von demselben Hersteller oder Geräten derselben Klasse) entwickelt werden. Die Steuerungssoftware (meist installiert auf einem prozessorgesteuerten Gerät) wird deshalb nach wie vor statisch auf das vorliegende Umfeld zugeschnitten (s. Einführung oben) und muss bei jeglichen Änderungen der Peripherie (Neukauf von Komponenten, verschiedene Hersteller, usw.) und/oder der Sensoren angepasst und neu übersetzt werden. Das hat den Grund, weil die Befehlssätze der Geräte, die Datenauswertungsstrategien, aber auch grafische Repräsentationen der Ablaufsteuerung, abhängig vom vorliegenden Laboraufbau sind und sehr stark variieren.

Alle Aspekte eines Laboraufbaus (Peripherie, Sensor, Ablaufsteuerung) sind nicht einzeln veränderbar, ohne mindestens einen weiteren Aspekt ändern zu müssen. Dies kann man sich anhand von Abbildung 1.2 verdeutlichen. Vor allem das Anwendungsbeispiel 2 (Austausch eines Spektrometers) spiegelt diesen Fall wider, bei welchem keine Änderungen im eigentlichen physikalischen Prinzip auftritt, sondern ausschließlich eine Komponente des analytischen Aufbaus durch eine „gleichwertige“ neue Komponente ersetzt werden soll.

Wird das physikalische Prinzip geändert (der Sensor), muss auch die Peripherie angepasst werden. Wird beispielsweise statt einer Photodiode ein CCD-Chip eingesetzt, muss der

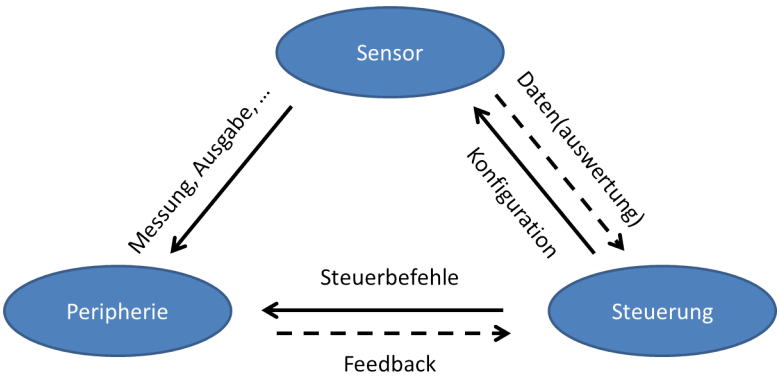


Abbildung (1.2)

Aspekte eines Laboraufbaus und deren Abhängigkeiten zueinander. Die Richtung der Abhängigkeit gibt an, welcher Aspekt (durch die Änderung eines anderen Aspekts) geändert werden muss. Durchgezogene Linien bedeuten dabei obligatorische Abhängigkeiten, die gestrichelten optionale Abhängigkeiten.

elektrische Aufbau an die neuen Rahmenbedingungen (Art des Auslesens, geänderte Datensätze, usw.) angepasst werden. Ebenso muss die Ablaufsteuerung angepasst werden, weil sich die Konfigurationen für die einzelnen Komponenten ändern (Auslesespannung, Integrationszeiten, Datentyp, usw.) und auch die Datenauswertung, die andere Informationen generieren muss. Ist beispielsweise der physikalische Effekt ein Spannungswert, muss eine andere mathematische Prozedur angewendet werden, als wenn Matrizen aus Spannungswerten empfangen werden.

Dasselbe gilt auch, wenn sich „nur“ die Peripherie ändert. Zwar müssen im Sensor dafür keine Änderungen erfolgen (der physikalische Effekt ist derselbe), aber die Ablaufsteuerung muss die neuen Komponenten abbilden können. Das ist verständlich, wenn man bedenkt, dass nun andere Komponenten mit anderen Befehlssätzen und Rückgabewerten abgebildet werden müssen, wodurch auch mögliche Regelketten (Feedback) beeinflusst und an die neuen Rahmenbedingungen angepasst werden müssen.

Ändert sich die Ablaufsteuerung, kann der aktuelle Laboraufbau aus den genannten Gründen in der vorliegenden Konfiguration nicht weiterverwendet werden. Das hängt damit zusammen, dass sie auf einen vorhanden Laboraufbau zugeschnitten und explizit dafür implementiert wird. Sie kann also nur bei einer wohl definierten Konfiguration (Geräte, Datenverarbeitung, Datenauswertung) eingesetzt werden. Ändert sich diese Konfiguration, müssen nicht die Geräte angepasst werden, sondern die Ablaufsteuerung

selbst, wodurch sie sehr stark abhängig sowohl von den einzelnen Peripherie-Elementen als auch von deren Konfiguration ist.
Dies bringt gravierende Probleme seitens der Entwicklung einer Ablaufsteuerung mit sich:

- Sie kann erst nach Fertigstellung der Gerätekonfiguration implementiert werden, da jede Änderung bei den Geräten auch eine Änderung der Ablaufsteuerung nach sich zieht (s. Problem 'Fehlende Flexibilität')
- Sie ist nicht wiederverwendbar, d.h. sie muss bei Änderung der Konfiguration programmatisch angepasst und anschließend neu kompiliert werden (s. Problem 'Beschränkte Übertragbarkeit')
- Unterschiedliche Laboraufbauten mit unterschiedlicher Peripherie aber gleichem physikalischen Effekt bedeuten oft den Einsatz von zwei unterschiedlichen Ablaufsteuerungen (s. Problem 'Wiederholte Programmierung ähnlicher Funktionalität')

Auf diese Probleme wird im Folgenden explizit eingegangen.

Fehlende Flexibilität bei Konfigurationsänderungen

Ändert sich die Konfiguration eines Laboraufbaus, was in der Forschung oder auch im Rahmen der Systemintegration (Anpassung von Spezifikationen, Neukauf von Geräten, Anpassung von Ablaufsequenzen, usw.) oft vorkommt, muss auch die Ablaufsteuerung auf die geänderten Bedingungen angepasst werden. Dies ist auch der Fall, wenn sich Geräte ändern, die eigentlich dieselbe Funktionalität erfüllen, aber von einem anderen Hersteller produziert oder aus einer anderen Produktpalette desselben Herstellers verwendet werden. Dadurch müssen oft tiefgreifende Änderungen vorgenommen werden. Abbildung 1.3 zeigt die einzelnen Unteraspekte der Ablaufsteuerung und deren Abhängigkeiten. In dieser Abbildung ist deutlich zu erkennen, dass bei Änderungen einer Komponente, alle anderen Unteraspekte (Datenauswertung, Grafische Oberfläche und Ablaufsequenz) angepasst werden müssen. Dies ist verständlich, wenn man bedenkt, dass sich mit der Änderungen einer Komponente Datenformate, Steuerbefehle und Funktionsumfang ändern und dadurch natürlich die Datenauswertung (im ersten Fall), die Ablaufsequenz (in zweitem Fall) und die grafische Oberfläche (in letztem Fall) angepasst werden müssen.

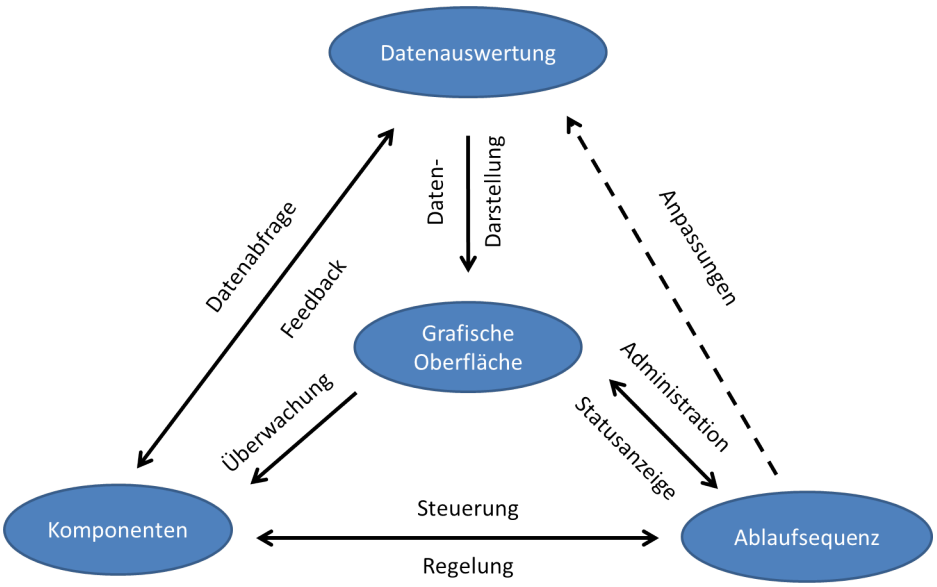


Abbildung (1.3)
Unteraspekte einer Ablaufsteuerung und deren Abhängigkeiten zueinander. Die Richtung der Abhängigkeit gibt an, welcher Aspekt (durch die Änderung eines anderen Aspekts) geändert werden muss. Durchgezogene Linien bedeuten dabei obligatorische Abhängigkeiten, die gestrichelten optionale Abhängigkeiten.

Beschränkte Übertragbarkeit

Da eine Ablaufsteuerung stark abhängig ist von Peripherie und physikalischem Effekt, können sie demnach schwer von einem Laboraufbau auf einen anderen übertragen werden. Es fehlen geeignete Strategien zur Verallgemeinerung von Geräten, deren Befehlssätzen und Schnittstellen. Aus diesem Grund werden Ablaufsteuerungen nach wie vor in gängigen Programmiersprachen implementiert, was aber bedeutet, dass bei Änderungen im Aufbau die Software programmatisch angepasst werden muss. Deshalb muss nach jeder Änderung der vollständige Prozess des Software-Testens durchlaufen werden, d.h. Modultests, Integrationstests, Systemtests, Abnahmetests. Dies kostet sehr viel Zeit und Ressourcen. Zusätzlich muss die Entwicklung in diesen Programmiersprachen von Fachpersonal mit tiefgreifendem Wissen in Programmierung und der Informatik eingesetzt werden. Derartig ausgebildetes Fachpersonal muss oft extra eingestellt oder die Aufgabe muss an ein externes Software-Unternehmen ausgelagert werden. Beide Möglichkeiten

verursachen hohe Kosten, letztere zusätzlich einen hohen Zeitaufwand. Oftmals werden jedoch die Anforderungen unterschätzt und Fachpersonal eingesetzt, welches explizit nicht für derartige Aufgabenstellungen ausgebildet ist. Das Resultat ist (oftmals fehlerbehaftete) Steuerungssoftware, die schlecht dokumentiert, sporadisch getestet, individuell gestaltet und dadurch schwer zu warten ist. Nachträgliche Änderungen sind nur vom Programmierer selbst durchführbar oder die Quelltexte nur unter hohem Aufwand von den Nachfolgern nachzuvollziehen.
Sind zusätzlich unterschiedliche Systeme beteiligt (z.B. verschiedene Windows Versionen oder Prozessorarchitekturen), muss der Quelltext der Software oft vollständig neu übersetzt werden, damit er auf dem zugrunde liegenden System lauffähig ist. Programmiersprachen, welche implizit Schnittstellen für die Gerätekommunikation bieten, sind aber oft auch sehr stark abhängig vom Betriebssystem (C/C++, .NET, usw.), während Programmiersprachen, die plattformunabhängig sind (Java, Python, usw.), nur über Umwege mit den Geräten kommunizieren können. Es ist deshalb einfacher, Software auf einen vorliegenden analytischen Aufbau eines bestimmten Systems zu fixieren als Übertragbarkeit zu gewährleisten.
Durch die fehlende Wiederverwendbarkeit ergeben sich zusätzlich Nachteile bei der Anbindung externer Ressourcen. Diese sollen als Ressourcen definiert werden, die nicht direkt zum Laboraufbau gehören (z.B. Speichersysteme, externe IT-Services und -Protokolle, usw.). Für Software, die dediziert für einen bestimmten Laboraufbau entwickelt wird, ist es meist zu aufwendig eine Vielzahl an externen Ressourcen anzubinden. Durch den beschränkten Einsatzbereich würde sich ein derartiger Aufwand nicht rechnen (s. auch Abschnitt 1.4.2).

Wiederholte Programmierung ähnlicher Funktionalität

Beschäftigen sich mehrere Arbeitsgruppen mit einem ähnlichen Thema, so werden auch die entstehenden Produkte gemeinsame Eigenschaften besitzen. Man betrachte dafür zwei Hersteller, die Messsysteme für ein ähnliches Verfahren herstellen, beispielsweise ein Chromatographiesystem mit automatischer Probenzufuhr. Die Messsysteme haben natürlich unterschiedliche Eigenschaften wie Performance, Durchsatz, Datenauswertung, grafische Darstellung usw. Dennoch haben sie auch gemeinsame Eigenschaften wie den Messablauf (Probennahme, Probentransport, Messung) oder die Datenausgabe (Chromatogramm). Beide Gruppen benötigen sehr viel Zeit und Ressourcen, diese ähnliche Funktionalitäten zu implementieren, worunter auch die Infrastruktur fällt, um diese Ei-

genschaften zu administrieren (grafische Editoren, logische Zusammenhänge, Datenformatierung, usw.). Der Zeitaufwand für wiederholte Programmierung ähnlicher Funktionalität ist hoch und erscheint nicht sinnvoll.

Lokale Steuerung und Datenspeicherung

Für einen bestimmten Laboraufbau werden Komponenten zu einem gemeinsamen System integriert und an einem Computersystem angeschlossen. Es ist naheliegend dasselbe Computersystem auch für die Ablaufsteuerung zu verwenden. Dieses Vorgehen hat Vorteile, weil die Gerätekommunikation über lokale Schnittstellen erfolgt und auf diese einfach zugegriffen werden kann. Dies bedingt aber auch, dass sich der Benutzer der Steuerungssoftware lokal am Computer befinden muss. Das heißt, die Überprüfung des Ablaufprozesses, z.B. der Endpunkt einer Messung, kann nur erfolgen, indem der Benutzer am lokalen System wartet oder in regelmäßigen Abständen den Steuerrechner manuell überprüft. Dieser Vorgang ist vor allem dann nicht möglich, wenn sich das vorliegende System in einer kritischen Umgebung befindet. Darunter fallen beispielsweise Messsysteme, die mit ionisierenden Strahlen arbeiten (Röntgen, Radioaktivität) oder Geräte für die Detektion von Sprengstoffen. Lokale Steuerungen sind in diesen Fällen nicht möglich und es müssen netzwerkbasierte Steuerungen eingesetzt werden. In diesem Fall müssen geeignete Sicherheitskriterien seitens der Steuerungssoftware erfüllt sein (s. Abschnitt 3.2.4).
Eine weitere Eigenschaft einer Steuerungssoftware ist die Datenspeicherung, welche oft auf dem lokalen System erfolgt. Die Einbindung von Schnittstellen zur Datenspeicherung über gängige netzwerkbasierte Dateitransferprotokolle (FTP, WebDAV, usw.) wird oft nicht vorgenommen. Das hängt damit zusammen, dass derartige Schnittstellen geeignete Konfigurationsmasken benötigen, um die Steuerungssoftware in die Infrastruktur des Einsatzgebietes einzubinden. Dadurch dass die Software dediziert für einen Anwendungszweck entwickelt wird, lohnt sich oft die Integration derartiger externer Ressourcen nicht (s. oben).

1.5. Lösungsansatz

In den folgenden Abschnitten sollen für die einzelnen Problematiken Lösungsvorschläge gezeigt werden, welche anschließend im Abschnitt 3.3 analysiert werden sollen. Dafür werden zunächst konkrete Lösungsansätze für die einzelnen Probleme erläutert. Diese werden in ein Konzept mit Anforderungen übersetzt und modellhaft visualisiert. Anschließend werden die Einzelmodelle inklusive Interaktionen in eine Struktur eingebettet, welche letztendlich mit Hilfe einer geeigneten Programmiersprache implementiert werden können.

1.5.1. Entwicklung eines Lösungskonzepts

Lösen der Abhängigkeiten durch Modularisierung

Abhängigkeiten zwischen den Aspekten sind bedingt durch die Vielfalt an verfügbaren Laborgeräten unterschiedlicher Hersteller und deren Syntaxen für die Gerätekommunikation. Deshalb muss bisher Steuerungssoftware für analytische Aufbauten konkret auf die vorliegenden Geräte zugeschnitten werden. Dazu gehören die direkte Gerätekommunikation, die Ablaufsteuerung und die Datenausgabe, welche direkt mit den verwendeten Geräten zusammenhängen. Ändert sich ein Gerät, ändern sich auch die Kommunikationsprotokolle und damit die Syntax der Ansteuerung. Dadurch wird direkt die Ablaufsteuerung beeinflusst, welche auf der Syntax aufgebaut ist. Zusätzlich ändern sich möglicherweise auch die Datentypen für die Datenausgabe, wodurch auch diese programmatisch angepasst werden muss.
Es ist naheliegend, die genannten Aspekte voneinander zu trennen. Dafür muss jeder Aspekt vollständig und unabhängig voneinander mit einer eigenen Grammatik beschrieben werden können. Die Interaktion zwischen den einzelnen Aspekten kann über ein Mapping realisiert werden, was heißt, dass einerseits die Schnittstellen der einzelnen Aspektbeschreibungen klar definiert sein müssen und sie andererseits über eine Zuordnungsvorschrift aufeinander abgebildet werden können. Die einzelnen Aspektbeschreibungen werden im Folgenden als *Deskriptoren* bezeichnet werden.

Beschränkte Übertragbarkeit

Die beschränkte Übertragbarkeit unterschiedlicher Geräte auf unterschiedliche Systeme ist auf die Vielfalt der vorhandenen Architekturen zurückzuführen. Da Geräte über Computerschnittstellen kommunizieren und diese abhängig sind von Betriebssystem, Rechnerarchitektur und deren lokale Konfigurationen, muss die Steuerungssoftware zum einen auf jeder dieser Architekturen lauffähig und die Geräte andererseits von den lokalen Konfigurationen unabhängig sein. Die Steuerungssoftware muss demnach entweder in einer Programmiersprache implementiert sein, die betriebssystemunabhängig ist oder es muss für jedes Betriebssystem und jede Rechnerarchitektur eigene Implementierungen geben. Darüber hinaus müssen die lokalen Konfigurationen jederzeit anpassbar sein, aber dürfen keinen Einfluss auf obige Deskriptoren haben. Diese lokale Konfiguration wird im Folgenden als *Schnittstellenkonfiguration* oder *Schnittstellendeskriptor* bezeichnet werden.

Flexibilität durch Kategorisierung

Ändert sich der analytische Aufbau für dieselbe Methode dahingehend, dass zwar die Geräte ausgetauscht werden, aber die Information des Resultats erhalten bleibt, sollen dafür nicht alle Deskriptoren geändert werden müssen. Es sollen ausschließlich die für die tatsächlichen Geräte verantwortlichen Deskriptoren angepasst werden müssen. Darunter fallen Gerätetreiber und die Konfigurationen für die Schnittstellen, an welchen die Geräte angeschlossen sind.
Um dies zu gewährleisten, werden Mappings zwischen Treiber und Schnittstellenkonfiguration, außerdem zwischen Treiber und Ablaufsequenzen benötigt. Erstere sollen im Folgenden als *Geräte-Aliase*, letztere als *Geräte-Kategorien* bezeichnet werden.

Zeit-/Kosten-Bedarf durch Komplexität der Steuerung

Da moderne Steuerungssoftware immer auf den analytischen Aufbau zugeschnitten ist, müssen jegliche Änderungen, auch sehr kleine, aufwendig programmatisch durchgeführt werden. Das bedeutet, der Quelltext der Steuerungssoftware muss verändert und anschließend neu kompiliert werden. Werden die zu steuernden Elemente eines analytischen Aufbaus vollständig über Deskriptoren und Mappings beschrieben, entfällt der Programmierprozess und alle damit verknüpften Arbeiten (Testen, Kompilieren, Neuinstallation auf das Zielsystem, usw.). Der Entwicklungsprozess vereinfacht sich sehr stark und er

kann vom Systemintegrator selbst anstatt von einem Systemprogrammierer vorgenommen werden.

Standardisierung wiederkehrender Funktionalität

Durch das Zuschneiden einer Steuerungssoftware auf einen vorliegenden analytischen Aufbau, müssen vielfach Module neu implementiert werden, auch wenn sich die Funktionalität nur geringfügig ändert. Auch dies erfolgt auf Ebene des Quelltextes und ist entsprechend komplex. Darüber hinaus werden Module für externe Ressourcen (Speicherung auf verschiedenen Datenbanken, Web- und mobile Anbindung, verschiedene Interaktionsmechanismen) nicht eingesetzt, weil sich für den vorliegenden „Spezialfall“ die Einbindung aus Zeit- und Kostengründen nicht rechnet. Bei einer auf Konfiguration basierenden Lösung können externe Ressourcen unabhängig vom jeweiligen analytischen Aufbau bereitgestellt werden, wodurch sich die Entwicklung trägt. Die Steuerungssoftware spart somit nicht nur Zeit und Kosten für Entwicklung, Wartung und Zertifizierungen, sondern ermöglicht es auch, durch Wiederverwendung von Komponenten, auf vielfältige Art und Weise mit der „Außenwelt“ zu kommunizieren (zentrale Ablage von Messergebnissen, Statusanzeige auf mobilen Geräten, Alarmierung bei Fehlverhalten von lang laufenden Messvorgängen) und Daten zu verarbeiten. Messsysteme erhalten somit einen deutlich umfangreicheren Funktionssatz.

Dezentrale Steuerung und Datenspeicherung

Aktuelle Steuerungslösungen werden als zusammenhängende Einheit entwickelt, d.h. die Gerätesteuerung und die grafische Benutzeroberfläche sind in einem Softwarepaket vereinigt. Dadurch ist der Benutzer aber auf ein Zielsystem beschränkt und muss die Steuerung lokal vornehmen. Kritische analytische Aufbauten in gefährlichen Bereichen (Radioaktive oder Sprengstoffanalytik) sind damit nicht möglich. Soll Steuerungssoftware für derartige Anwendungsfälle entwickelt werden, konstruiert man speziell auf diesen Fall abgestimmte, verteilte Systeme. Um eine derartig spezifische Entwicklung zu vermeiden, soll von Grund auf die Geräte- und die Benutzerinteraktion voneinander getrennt werden. Die Kommunikation zwischen den beiden kann über Standard Netzwerkschnittstellen erfolgen, wofür in den meisten Fällen in den Einsatzbereichen die Infrastruktur für Netzwerke bereits vorhanden ist (Router, Switches, Verkabelung, Funkstandards). Diese Vorgehensweise bietet auch für moderne Anwendungen enorme Vorteile, weil der

Zukunftstrend in die Richtung vernetzter Systeme geht (Stichworte „Internet der Dinge“ und „Industrie 4.0“).

1.5.2. Anwendung der Lösungsansätze auf die Anwendungsszenarien

Generelle Anwendung der Lösungsansätze

Generell ist es ausreichend, Textdateien zu verwenden, welche Anweisungen in Form von Parametern enthalten, um Gerätesteuerungen für ein vorliegendes Anwendungsszenario zu entwickeln und damit bestimmte Prozesse zu initiieren. Dies soll in diesem Kapitel gezeigt werden. Auf die einzelnen Prozesse wird später ab Abschnitt 3.3 näher eingegangen, wie auch auf die einzelnen Begriffe und Deskriptoren genauer im nächsten Abschnitt eingegangen wird. Dadurch, dass es sich um Textdateien handelt, können die Deskriptoren ohne Probleme auf andere Systeme portiert werden. Solange die Prozesse auf diesem System ausgeführt werden können, sind die Deskriptoren übertragbar. Darüber hinaus ist es zeitsparender, Deskriptoren im Gegensatz zu Quelltext in einer Programmiersprache zu entwickeln. Zwar müssen die einzelnen Parameter der Deskriptoren ebenso wie die Sprachmerkmale einer Programmiersprache gelernt werden. Sie sind sie aber speziell auf den zu entwickelnden Anwendungsfall (d.h. *Entwicklung einer Ablaufsteuerung*) zugeschnitten und somit für den Anwendungsfall selbsterklärend im Gegensatz zu den offenen Merkmalen einer Programmiersprache, mit welcher viel mehr realisiert werden kann als ausschließlich Ablaufsteuerungen.
Module können einfach in eine zu entwickelnde Ablaufsteuerung geladen werden, um Funktionalitäten zu erweitern. Auch hierfür müssen nur die Parameter gesetzt werden, ohne die genauen Details und Feinheiten der zugrunde liegenden Programmiersprache verstehen zu müssen. Sie verhalten sich wie Plugins bekannter Softwareentwicklungen wie beispielsweise von Web-Browsern und können ebenso einfach eingesetzt werden.
Zuletzt soll hervorgehoben werden, dass bisher nur von Deskriptoren, aber niemals von Benutzeroberflächen gesprochen worden ist. Die zu entwickelnde Plattform soll im eigentlichen Sinn über keine Benutzeroberfläche verfügen, sondern ausschließlich über Befehlssätze mit einem Benutzer interagieren. Die Befehlssätze werden auf standardisierten Schnittstellen empfangen, welche die Plattform zur Kommunikation anbietet wie beispielsweise Webservices oder Queues. Die Benutzeroberfläche ist dadurch per se von der eigentlichen Ablaufsteuerung getrennt und kann unabhängig entwickelt werden.

Anwendung auf die Fluoreszenzspektroskopie

Bei der Fluoreszenzspektroskopie werden Gerätebefehle direkt an das Gerät gesendet, welches daraufhin Aktionen wie das Messen eines Fluoreszenzspektrums anstößt (s. Abschnitt 2.1.2). Durch die Modularisierung würden die Gerätebefehle, welche komplexe Zeichenfolgen darstellen, die von Menschen nur schwer gelesen werden können, in einen Deskriptor, dem Treiber, geschrieben und in ein menschenlesbares Kommando übersetzt werden. Mit dem Aufruf eines einfachen Befehls wie „measure“ ist es nun für eine Person möglich, einen Messvorgang zu starten. Ein weiterer Deskriptor, die Schnittstellenkonfiguration, ist verantwortlich dafür, dem Betriebssystem die Parameter zur Verfügung zu stellen, damit über eine Hardware-Schnittstelle (hier: RS232) mit dem Gerät kommuniziert werden kann. Zuletzt muss über einen Ausgabe-Deskriptor festgelegt werden, wohin und auf welche Weise empfangene Daten gespeichert werden sollen.

Anwendung auf die reflektometrische Interferenzspektroskopie (RIfS)

Dieselben Deskriptoren wie für das Fluoreszenzspektrometer, d.h. Treiber, Schnittstellenkonfiguration, müssen auch hier geschrieben werden, aber für jedes am Messsystem beteiligte Gerät. Darunter fallen die Laborpumpen, das Spektrometer und das Probenzuführungssystem (Autosampler). Auch der Ausgabedeskriptor muss geschrieben werden. Er ist ungleich komplexer, da dort auch die mathematische Routine festgelegt werden muss, welche aus RIfS-Signalen Informationen extrahiert. Darüber hinaus ist mit einer RIfS-Messung ein Ablauf verbunden, da eine zu vermessende Probe (s. Abschnitt 2.1.4) von einem Probenreservoir mit Reagenzien vermischt, zum Sensor transportiert und erst dort spektroskopisch vermessen werden muss. Dieser Ablauf bedingt einen weiteren Deskriptor, die Ablaufsequenz, welche die einzelnen Schritte der Gerätesteuerung vereinigt.

Anwendung auf INSTANT

Das Vorgehen entspricht weitestgehend dem Entwurf von RIfS, da es sich ebenso um ein Messsystem bestehend aus mehreren Einzelkomponenten handelt, welche in einer klar definierten Reihenfolge Aktionen durchführen müssen. Zwar müssen verschiedene Datenformate (Bilder, Impedanzdaten) berechnet und gespeichert werden, so entspricht dies aber nur einer unterschiedlichen Deskriptor-Konfiguration. Die Tatsache, dass mehr Geräte am Ablauf beteiligt sind, bedeutet zwar mehr Arbeit bei der Konfiguration.

Die Komplexität verändert sich aber nicht durch die Art und Anzahl der Geräte, sondern durch die Mathematik der Datenanalyse (s. Abschnitt 2.1.5). Da diese ebenfalls in Deskriptoren festgelegt ist, gelten obige Anmerkungen auch für derart umfangreiche Messsysteme.

Alexander von Humboldt

Die gefährlichste aller Weltanschauungen ist die Weltanschauung der Leute, welche die Welt nicht angeschaut haben.

2

Automatisierung in physikalisch-chemischen Laboratorien

2.1. Physikalische und Chemische Grundlagen

Um die Anwendungsbeispiele zu verstehen, welche in den folgenden Kapiteln zum Verständnis des entwickelten Konzepts herangezogen werden, sind die nachfolgenden physikalisch-chemischen Grundlagen beschrieben. Darunter fallen Grundlagen über Strahlung und Materie (Abschnitt 2.1.1), Grundlagen der Fluoreszenz-Spektroskopie (Abschnitt 2.1.2), der reflektometrischen Interferenzspektroskopie (Abschnitt 2.1.4), der 1-λ-Reflektometrie (Abschnitt 2.1.5), aber auch grundlegende Aspekte von Sensoren (Abschnitt 2.1.3) und Biosensoren.

2.1.1. Elektromagnetische Strahlung

Als elektromagnetische (EM) Strahlung werden Wellen bezeichnet, welche aus einer Kopplung aus einem elektrischen (E-Feld) und einem magnetischen Feldvektor (B-Feld) bestehen. Beide Vektoren stehen senkrecht zueinander und schwingen in Phase. Die Strahlung hat in verschiedenen Medien unterschiedliche Ausbreitungsgeschwindigkeiten, die im Vakuum der Lichtgeschwindigkeit c entspricht und in dichteren Medien einer entsprechend geringeren. Die Charakterisierung von EM-Strahlung erfolgt über die Angabe einer Frequenz ν und einer Wellenlänge λ, deren Produkt die Ausbreitungsgeschwindigkeit im betrachteten Raum entspricht:

$$c_m = \lambda \cdot \nu \tag{2.1}$$

In einem Medium m wird die Welle gebremst, wodurch die Wellenlänge und die Amplitude verringert werden, die Frequenz aber gleich bleibt und als Resultat auch die Geschwindigkeit im Medium c_m verringert wird. Abhängig von ihrer Frequenz im Vakuum, wird EM-Strahlung in verschiedene Bereiche aufgeteilt, welcher von $10^7 - 10^{21}$ Hz reicht und je nach Frequenz unterschiedliche Arten der Wechselwirkungen mit Materie aufweist. Der sichtbare Bereich befindet sich im Bereich von 10^{15} Hz, in welcher sämtliche in dieser Arbeit betrachteten Wechselwirkungsprozesse ablaufen.
Rein prinzipiell sind folgende Interaktionsmöglichkeiten beim Auftreffen von EM-Strahlung auf Materie möglich:

1. Strahlung wird aufgenommen (Absorption)
2. Strahlung wird zurückgeworfen (Reflexion)

3. Strahlung wird durchgelassen (Transmission)

In den folgenden Kapiteln werden ausschließlich Wechselwirkungen von Strahlung im optischen Bereich und deren Interaktionsmöglichkeiten in diesem Frequenzbereich betrachtet.

Absorption und Emission

Die Absorption von Strahlung im optischen Bereich erfolgt in erster Näherung ausschließlich über schwingende Dipole (Dipol-Näherung). Durch die Aufnahme von EM-Strahlung einer geeigneten Frequenz, verändert sich der Energieinhalt der Materie und es findet ein Übergang von einem tieferen Energieniveau E_1 in ein höheres E_2 statt, wenn folgende Resonanzbedingung erfüllt ist:

$$\Delta E = h \cdot \nu \tag{2.2}$$

Dabei ist ΔE die aufgenommene Energie und h die Plancksche Konstante. Voraussetzung dafür ist die Annahme diskreter Energieniveaus in Atomen und Molekülen. Im betrachteten Frequenzbereich von ca. 10^{14} Hz kommt es zu einem Elektronenübergang zwischen zwei Molekülorbitalen (s. Fachliteratur der physikalischen Chemie[APB+06][WF12]). Die Absorption findet in diesem Frequenzbereich ausschließlich über die Elektronenhülle von Molekülen statt, was bedeutet, dass die Elektronenhülle gegenüber den Atomkernen verschoben wird. Somit wird ein Dipol induziert, weil positive und negative Ladungsschwerpunkte nicht mehr zusammenfallen. Bei Annahme der Dipolnäherung wird dadurch eine Absorption von Strahlung ermöglicht. Der Fall einer Verschiebung der Ladungsschwerpunkte wird als *Verschiebungspolarisation* bezeichnet und kann über die Clausius-Mosotti-Beziehung beschrieben werden:

$$P_V = \frac{\epsilon_r - 1}{\epsilon_r + 2} V_m = \frac{1}{3\epsilon_0} N_L \alpha \tag{2.3}$$

mit ϵ_r als Permittivitätszahl, ϵ_0 als elektrische Feldkonstante, V_m als molares Volumen, N_A als Avogadro Konstante und α als Polarisierbarkeit. Die Polarisierung[1] $P = P_V$ setzt sich aus der Summe aller induzierten Dipolmomente zusammen, die über die Permittivitätszahl ϵ_r einen Zusammenhang zum induzierten E-Feld nach

$$P = (\epsilon_r - 1)\epsilon_0 E \tag{2.4}$$

[1] Orientierungspolarisation P_O wird im optischen Frequenzbereich vernachlässigt

herstellt. Dieses kann von einem externen Wechselfeld $h \cdot \nu$ beeinflusst werden. Dabei spielt die Permittivitätszahl bzw. die Dielektrizitätskonstante (DK) eine wichtige Rolle. Sie spiegelt für das zugrunde liegende Molekül bzw. Medium das Verhalten bei der betrachten Frequenz wider. Ausgehend von einem klassischen Oszillatormodell, kann gezeigt werden, dass die DK eine Funktion der Frequenz darstellt und aus einem Real- und einem Imaginärteil besteht (Herleitung s. [GZ94]). Ersterer beschreibt dabei die Frequenzabhängigkeit der Phasengeschwindigkeit der Strahlung, also die *Dispersion*, letzterer die Dämpfung, also die *Absorption*.

Nach Abschalten des externen Feldes, hat ein Molekül das Bestreben, die absorbierte Energie wieder abzugeben. Die Verschiebung der Elektronenhülle erfolgt jetzt in die Gegenrichtung, wodurch negativer und positiver Ladungsschwerpunkt wieder zusammenfallen. Da zusätzlich bei einer Anregung auch andere Wechselwirkungsprozesse vonstatten gehen, wie beispielsweise Schwingungen oder Rotationen (s. Jablonski Termschema in [GZ94]), wird über interne Prozesse, z.B. Reibung oder Stöße, ein Teil der Energie sofort in Form von Wärme abgegeben. Solange bis ein stabilerer Zwischenzustand erreicht wird, nämlich der Grundzustand des elektronisch angeregten Zustands, ist dies auch die einzige Möglichkeit, den Energiebetrag zu reduzieren. Von dort ausgehend können verschiedene Konkurrenzprozess ausgehen. Eine Möglichkeit ist es, die restliche aufgenommene Energie über weitere Stoßprozesse und Reibung zu reduzieren. Andererseits kann, vor allem im Falle von sehr verdünnten Gasen, die überschüssige Energie in Form von Licht wieder abgegeben werden. Dieser Vorgang wird als *Emission*, in diesem konkreten Fall als *Fluoreszenz* bezeichnet. Als dritte Möglichkeit kann das Molekül durch Spinumkehr in einen Triplett-Zustand wechseln und von dort ebenfalls durch Stoßprozesse Wärme abgeben oder wieder Licht emittieren; letzteres wird als *Phosphoreszenz* bezeichnet. Auf die Prozesse ausgehend vom Triplett-Zustand wird im Rahmen dieser Arbeit nicht weiter eingegangen. Die Emission in Form von Licht ausgehend vom Singulett-Zustand wird im Abschnitt 2.1.2 näher beleuchtet, wenn die Fluoreszenzspektroskopie besprochen wird.

Transmission und Reflexion

Transmission und Reflexion von Strahlung an Materie hängen stark vom Brechungsindex der betroffenen Medien ab. Der Brechungsindex ist ein Maß für die Geschwindigkeit von Strahlung in verschiedenen Medien:

$$n = c/c_{medium} \tag{2.5}$$

mit n als Brechungsindex, c als Lichtgeschwindigkeit des Vakuums und c_{medium} als Lichtgeschwindigkeit des Mediums. Er ist demnach eine einheitenlose Zahl größer als 1. Über die Maxwell-Relation

$$n^2 = \epsilon_r \mu_r \tag{2.6}$$

hängt der Brechungsindex direkt mit der Permittivitätszahl und der magnetischen Permeabilität zusammen. Im Falle von nicht-magnetisierbaren Medien ist $\mu_r \approx 1$ und dadurch $n \approx \sqrt{\epsilon_r}$.
Beim Übergang von Strahlung aus einem Medium in ein anderes gilt das Brechungsgesetz von Snellius:

$$n_1 \sin \phi_1 = n_2 \sin \phi_2 \tag{2.7}$$

Das Gesetz besagt, dass beim Eintritt eines Strahls im Winkel ϕ_1 von einem optisch dünneren (1) in ein optisch dichteres Medium (2), der Strahl zum Lot hin, im Winkel ϕ_2, gebrochen wird. Dies gilt für die Transmission, während bei der Reflexion der Austrittswinkel gleich dem Eintrittswinkel ist. Für die Intensitäten der Strahlen können die Fresnel'schen Formeln herangezogen werden:

$$r_p = \frac{n_2 \cdot \cos \phi_1 - n_1 \cdot \cos \phi_2}{n_2 \cdot \cos \phi_1 + n_1 \cdot \cos \phi_2} \tag{2.8}$$

$$r_s = \frac{n_1 \cdot \cos \phi_1 - n_2 \cdot \cos \phi_2}{n_2 \cdot \cos \phi_2 + n_1 \cdot \cos \phi_1} \tag{2.9}$$

$$t_p = \frac{2 \cdot n_1 \cdot \cos \phi_1}{n_2 \cdot \cos \phi_1 + n_1 \cdot \cos \phi_2} \tag{2.10}$$

$$t_s = \frac{2 \cdot n_1 \cdot \cos \phi_1}{n_2 \cdot \cos \phi_2 + n_1 \cdot \cos \phi_1} \tag{2.11}$$

Hier wird zwischen senkrecht (s) und parallel (p) polarisiertem Licht sowohl für Intensität der Transmission (t) als auch der Reflextion (r) unterschieden. s-Polarisierte Strahlung bedeutet, dass der elektrische Feldvektor senkrecht zur Einfallsebene steht, bei p-polarisierter Strahlung entsprechend parallel dazu.

2.1.2. Fluoreszenzspektroskopie

Fluoreszenz ist eine Möglichkeit der Emission von Licht. Sie geht von elektronisch angeregten Zuständen aus und kann prinzipiell bei jeder Substanz auftreten. Im Gegensatz zur Phosphoreszenz handelt es sich beim elektronisch angeregten Zustand um ein Singulett-

Zustand, in welchem das Elektron im angeregten Orbital gepaart mit unterschiedlichem Spin vorliegt. Die typische Fluoreszenzlebenszeit beträgt 10 ns, die Emissionsraten entsprechend ca. $10^8 s^{-1}$. Typischerweise tritt Fluoreszenz bei aromatischen Molekülen auf, wie z.B. Fluorescein, den Rhodaminen oder auch Quinin (welches Bestandteil von Tonic Water ist).
Fluoreszenz tritt immer bei höheren Wellenlängen auf als die verursachende Absorption. Dieser Effekt wird als *Stokes-Shift* bezeichnet und er hängt damit zusammen, dass nach Anregung des Moleküls in einen angeregten Zustand, die Energie innerhalb weniger pico-Sekunden durch die „innere Umwandlung“ in Wärme überführt wird, bis der Schwingungs-Grundzustand des elektronisch angeregten Zustands erreicht ist, bevor die Emission erfolgt. Dadurch hat der aufgenommene Energiebetrag abgenommen, wodurch sich nach $\Delta E = h\nu = hc/\lambda$ die Wellenlänge erhöht.
Wichtige Eigenschaften der Fluoreszenz sind darüber hinaus die Quantenausbeute und die Lebensdauer. Bei ersterem handelt es sich um das Verhältnis

$$Q = \frac{\Lambda}{\Lambda + k} \tag{2.12}$$

berechnet mit Q als Quantenausbeute, Λ als Ratenkonstante der Emission und k als Ratenkonstante des strahlungslosen Übergangs. Unter der Fluoreszenzlebenszeit wird die durchschnittliche Zeit verstanden, in welcher ein Molekül im angeregten Zustand verbleibt. Typische Fluoreszenzlebenszeiten sind im Bereich von 10 ns. Auch sie werden über die Ratenkonstanten Λ und k über

$$\tau = \frac{1}{\Lambda + k} \tag{2.13}$$

berechnet.

Ein schematischer Aufbau eines typischen Fluoreszenzspektrometers ist in Abbildung 2.1 gezeigt. Eine fluoreszierende Probe in einer Küvette wird mit Licht einer bestimmten Wellenlänge bestrahlt, welche über einen Monochromator eingestellt werden kann. Die emittierte Fluoreszenz wird 90° zum Anregungsstrahl über einen zweiten Monochromator spektral zerlegt und vom Detektor wellenlängenabhängig detektiert. Mit diesem Aufbau sind zwei Detektionstypen möglich: Der Emissionstyp, bei welchem bei einer festen Wellenlänge angeregt und die Emission gescannt wird, und der Extinktionstyp, bei welchem die Anregungswellenlänge variiert und bei einer festen Emissionswellenlänge detektiert wird.

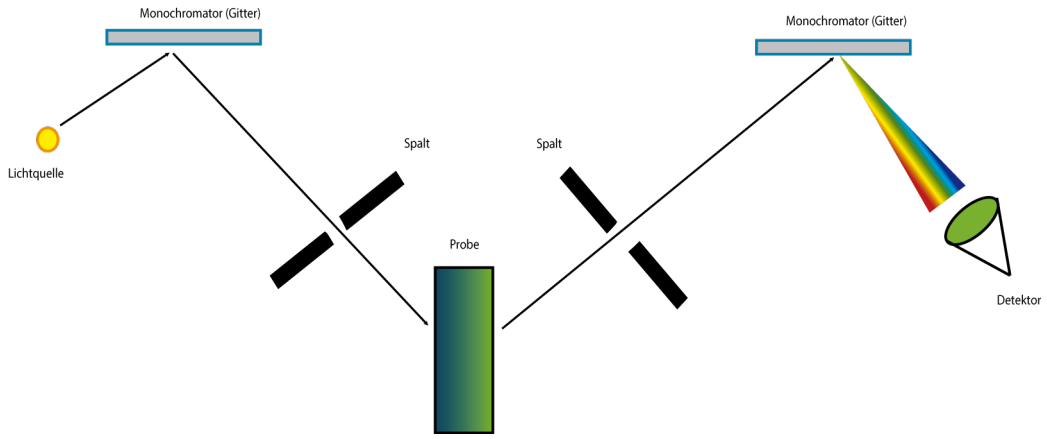


Abbildung (2.1)
Schematischer Aufbau eines Fluoreszenz Spektrometers mit Anregungsquelle, Monochromatoren, Probe und Detektor.

Im Normalfall werden mit einem Fluoreszenzspektrometer Emissions-Spektren aufgenommen, d.h. es wird bei einer festen Wellenlänge angeregt. Dafür muss das Absorptionsverhalten der zu vermessenden Probe bekannt sein, da nur eine Anregungen innerhalb des Absorptionsbereichs des Farbstoffs zu Fluoreszenz führen kann. Viele Fluoreszenzspektrometer bieten die Möglichkeit, gleichzeitig die Anregungs-und die Emissionswellenlänge zu variieren. Auf diese Weise ist es möglich den gesamten sichtbaren Spektralbereich zu durchsuchen.

2.1.3. Sensoren

Ein Sensor ist ein elektrisches Bauteil, um physikalische oder chemische Eigenschaften in ein elektrisches Signal umzuwandeln. Als physikalische oder chemische Eigenschaften werden beispielsweise Temperatur, Druck, Magnetismus, Kraft, aber auch Beschleunigung, Strömungen oder Luftfeuchtigkeit usw. verstanden. Wird ein Sensor zur Erfassung von Strahlungen wie beispielsweise elektromagnetische Strahlung oder Teilchen verwendet, wird von einem *Detektor* gesprochen. Innerhalb dieser Arbeit werden beide Begriffe äquivalent betrachtet, da beide äquivalent die zu beobachtende Eigenschaft als elektrisches Signal zur Verfügung stellen.

Ein wichtiger Detektor, der vor allem in Spektrometern im UV/Vis Bereich Verwendung

findet (z.B. dem Fluoreszenzspektrometer), ist der Photomultiplier. Er besteht aus einer Photozelle in Kombination mit einem Elektronenvervielfacher. Er funktioniert nach dem Prinzip des äußeren Photoeffekts, bei welchem Licht geeigneter Frequenz in der Lage ist, Elektronen aus einem Metall herauszuschlagen. Das Metall befindet sich in einer Vakuumröhre und bildet die Kathode. Die herausgeschlagenen Elektronen werden über eine Beschleunigungsspannung über mehrere sog. Dynoden auf eine Anode hin beschleunigt. Beim Auftreffen der beschleunigten Elektronen werden Sekundärelektronen aus dem Dynodenmaterial herausgeschlagen, welche auf die nächste Dynode hin beschleunigt werden. Es entsteht eine Elektronenkaskade, wodurch auch sehr schwache Signale durch Vervielfachung an der Anode über einen Photostrom detektiert werden können.

Ein weiterer häufig verwendeter optischer Detektor, der vor allem in älteren Geräten Verwendung findet, ist das Photodiodenarray. Es besteht aus einem Bauelement mit vielen in einem Raster angeordneten Photodioden. Eine Photodiode funktioniert nach dem Prinzip des inneren Photoeffekts, bei welchem durch Auftreffen von Photonen geeigneter Frequenz Übergänge zwischen Valenz- und Leitungsband eines Halbleiters auftreten. Durch den Übergang entsteht im Leitungsband ein negativer Ladungsüberschuss (ein Elektron mehr), während im Valenzband das fehlende Elektron ein sog. „Loch“ hinterlässt (ein Elektron zu wenig) und damit einem positiven Ladungsüberschuss. Dieser Zustand ist nicht stabil und über den Abtransport von Elektron bzw. Loch wird versucht, das elektrische Gleichgewicht wieder herzustellen. Im Fall einer Diode besteht das Material aus einem zusammengesetzten p-halbleitenden und einem n-halbleitenden Anteil. Das bedeutet, die Anteile werden mit Materialien dotiert, welche im Vergleich zum Bulk entweder ein Elektron weniger aufweisen (p-dotiert) bzw. eines mehr (n-dotiert). Werden jetzt die beiden Anteile leitend verbunden, Elektroden an die Oberflächen angeschlossen und eine Spannung angelegt, so wandert das angeregte Elektron des Leitungsbands bevorzugt über den n-dotierten Anteil zur angeschlossenen Elektrode, wenn wie oben angesprochen durch Licht ein Elektronenübergang induziert wird. Das Loch verhält sich genau gegensätzlich und „wandert“ über den p-dotierten Anteil, indem es schrittweise durch ein benachbartes Elektron aufgefüllt wird. Dadurch entsteht ein Photostrom, der gemessen werden kann und mit der eingefallenen Lichtmenge korreliert.

Ein Nachfolger des Photodiodenarray ist das Charged-Coupled-Device (CCD), welches heute in jeder Digitalkamera zu finden ist. Sie bestehen aus einer Matrix aus Photodioden und beruhen demnach ebenso auf dem inneren Photoeffekt. Der Unterschied liegt in der Ableitung der durch die Photonen angeregten Elektronen, die nicht direkt über

die angeschlossene Elektrode abfließen, sondern erst in einem Potenzialtopf gesammelt werden. In gewissen Zeitabständen werden die gesammelten Ladungen durch Anlegen einer Auslesespannung Schritt für Schritt, d.h. von Potenzialtopf zu Potenzialtopf nach außen transportiert, wo sie als Ladungspakete über eine Verstärkerschaltung detektiert werden.

Biosensor

Ein Biosensor stellt ein Messsystem dar, welches eine biologische Komponente enthält. Er bestehend aus zwei Teilen, einem biologischen Rezeptor und einem Transducer. Der biologische Rezeptor ist für die Erkennung des zu detektierenden Analyten verantwortlich und verändert dabei eine physikalisch/chemische Eigenschaft des Transducers, der somit das biologische Signal in ein optisches, elektrisches, mechanisches usw. übersetzt. Es gibt verschiedene Arten von biologischen Interaktionen, welche anhand des eingesetzten Rezeptors unterschieden werden. Andere Arten von Biosensoren basieren auf Antikörper-Antigen-, Nukleinsäure-, Zell- und enzymatischen Interaktionen. Während elektrische Signale direkt detektiert werden können, sind für anderweitige Signale weitere Sensoren bzw. Detektoren nötig. Auf Basis der Art der Detektion werden verschiedene Biosensor-Arten unterschieden. So verwenden beispielsweise optische Biosensoren Licht als Sonde und entsprechend einen optischen Detektor. Andere Arten basieren auf elektrochemischer, piezoelektrischer, pyroelektrischer oder gravimetrischer Detektion.

2.1.4. Reflektometrische Interferenzspektroskopie

Unter der Reflektometrischen Interferenzspektroskopie (RIfS) wird die Beobachtung von Weißlichtinterferenz an dünnen Schichten verstanden. Detektiert werden die reflektierten Teilstrahlen der einzelnen Phasengrenzen eines transparenten Mehrschichtsystems, welche durch Interferenz ein charakteristisches Interferenzspektrum ergeben. Die reflektierte Intensität kann über die Fresnel'schen Gleichungen (s. Abschnitt 2.1.1) hergeleitet werden und ist abhängig von den einzelnen Schichtdicken d und deren Brechungsindex n. Eine weitere Variable ist der Einstrahlwinkel ϕ, der allerdings im Fall der RIfS 90° beträgt. Mit Hilfe des Snellius'schen Brechungsgesetzes (s. Abschnitt 2.1.1) kann daraus mit Hilfe des geometrischen, optischen Wegs Θ eine Phasenverschiebung δ berechnet

werden (Herleitung s. [Hec99]):

$$\delta = k_0\Theta = 2k_0 nd\cos\phi_t \tag{2.14}$$

mit k_0 als Wellenzahl $2\pi/\lambda_0$, n als Brechungsindex, d als Schichtdicke und ϕ_t als Einstrahlwinkel. Sei die Phasenverschiebung $\delta = 2m\pi$, so gilt

$$d\cos\phi_t = (2m+1)\lambda_0/(4n)(Maxima) \tag{2.15}$$

$$d\cos\phi_t = 2m\lambda_0/(4n)(Minima) \tag{2.16}$$

und für den senkrechten Einfall

$$d = (2m+1)\lambda_0/(4n)(Maxima) \tag{2.17}$$

$$d = 2m\lambda_0/(4n)(Minima) \tag{2.18}$$

Das Produkt der beiden Größen Schichtdicke d und Brechungsindex n wird als *optische Schichtdicke* bezeichnet und es gilt exemplarisch für Minima des Interferenzspektrums in der RIfS:

$$(nd) = m\lambda/2 \tag{2.19}$$

Bei Änderung der Dicke oder des Brechungsindex des Schichtsystems, z.B. bei Anlagerung von Analyten an der Oberfläche, ändert sich der Term (nd) und damit die Wellenlänge λ des Minimums (oder äquivalent des Maximums), welches über die Zeit aufgetragen, die Veränderung der Oberfläche des Mehrschichtsystems durch Menge, Dichte und Größe des Analyten widerspiegelt (s. Abbildung 2.5).

Die reflektometrische Interferenzspektroskopie stellt einen Biosensor dar, mit dem es möglich ist, biologische Interaktionen auf transparenten Oberflächen zu detektieren. Dazu wird entweder ein biologischer Rezeptor oder dessen korrespondierender Ligand auf der Transduceroberfläche immobilisiert, über ein mikrofluidisches System der Analyt (Ligand bzw. Rezeptor, je nach Oberflächenbelegung) darüber geleitet und über ein glasfasergekoppeltes Diodenzeilenspektrometer detektiert (s. Abbildung 2.2).

Das mikrofluidische System besteht im einfachsten Fall aus einer einzigen Pumpe, welche über Polymer-Schläuche mit einer Flusszelle verbunden ist, die auf die beschichtete Transduceroberfläche gepresst wird (nicht in Abbildung 2.2 dargestellt). Dies wird als „direktes Testformat“ bezeichnet (s. Abschnitt 2.2.2). Als Pumpen kommen entweder

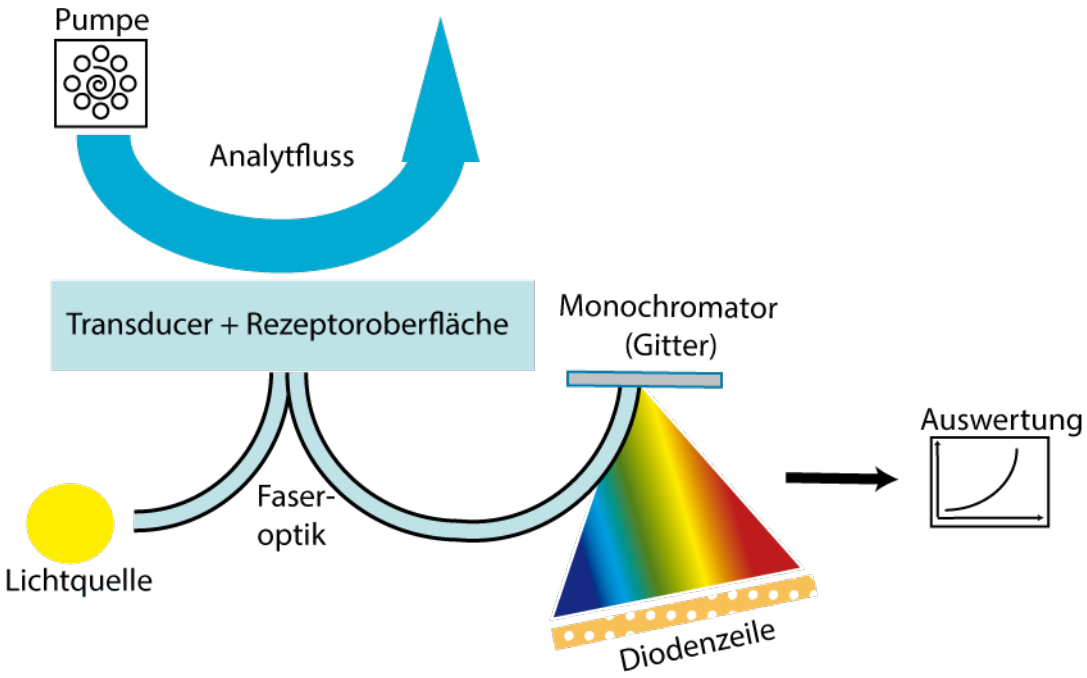


Abbildung (2.2)
Schematischer Aufbau der Reflektrometrischen Interferenzspektroskopie

peristaltische oder Spritzenpumpen zum Einsatz. Bei ersterer wird ein Fluss des zu fördernden Mediums durch das Verdrängungsprinzip realisiert, bei dem durch mechanische Verformung des Schlauches die Flüssigkeit weiterbewegt wird. Das zweite Prinzip stellt die Spritzenpumpe dar, bei welcher, wie es der Name bereits vermuten lässt, eine Spritze gefüllt und wieder entleert wird. Letztere gelten als sehr präzise hinsichtlich der Flussgeschwindigkeit und in der Abgabe geringer Volumina. Komplexere Aufbauten enthalten zusätzlich Ventile und eine Probenschleife, in welcher eine Probe zwischengelagert werden kann. Dadurch können verschiedene Flüssigkeiten nacheinander über den Transducer geleitet werden, z.B. ein Laufpuffer, die Probe und anschließend eine Regenerationslösung. Ein optionales Probenzuführungssystem (Autosampler) erlaubt es, nacheinander mehrere Zyklen mit verschiedenen Proben zu vermessen. Ein typisches RIfS Bindungssignal mit Aufnahme einer Basislinie, Assoziation und Regeneration ist in Abbildung 2.4 gezeigt.

Die Detektion erfolgt über ein Photodiodenarray, welches zu jedem Zeitpunkt pro Diode

Abbildung (2.3)
Links: Ismatec ASIA System mit zwei Peristaltikpumpen, zwei Ventilen und einer manuellen Bedienkonsole. Rechts: Hamilton Microlab Spritzenpumpe.

einen Messwert aufnehmen kann. Dadurch muss nicht durch Bewegung des Monochromators die jeweilige zu beobachtende Wellenlänge eingestellt werden, sondern es wird der gesamte durch den Monochromator abgedeckte Spektralbereich gleichzeitig aufgezeichnet, wobei mit jeder Photodiode eine bestimmte Wellenlänge korreliert wird. Das resultierende Spektrum wird aufgezeichnet, gespeichert und über mehrere Sekunden mit vorher gespeicherten Spektren gemittelt, um das Signal-Rausch-Verhältnis zu verbessern. Anschließend wird das gemittelte Spektrum durch ein Referenzspektrum dividiert, welches mit einem speziellen Glastransducer ohne Interferenzschicht aufgenommen worden ist. Das resultierende referenzierte Spektrum wird nun ausgewertet, indem die Veränderung eines Punktes (z.B. eines Minimums) gegen die aktuelle Zeit aufgetragen wird. Ein RIfS-Bindungssignal ergibt sich aus der Auftragung vieler derartiger Spektren bzw. der Lage von deren Minima gegen die Aufnahmezeit.

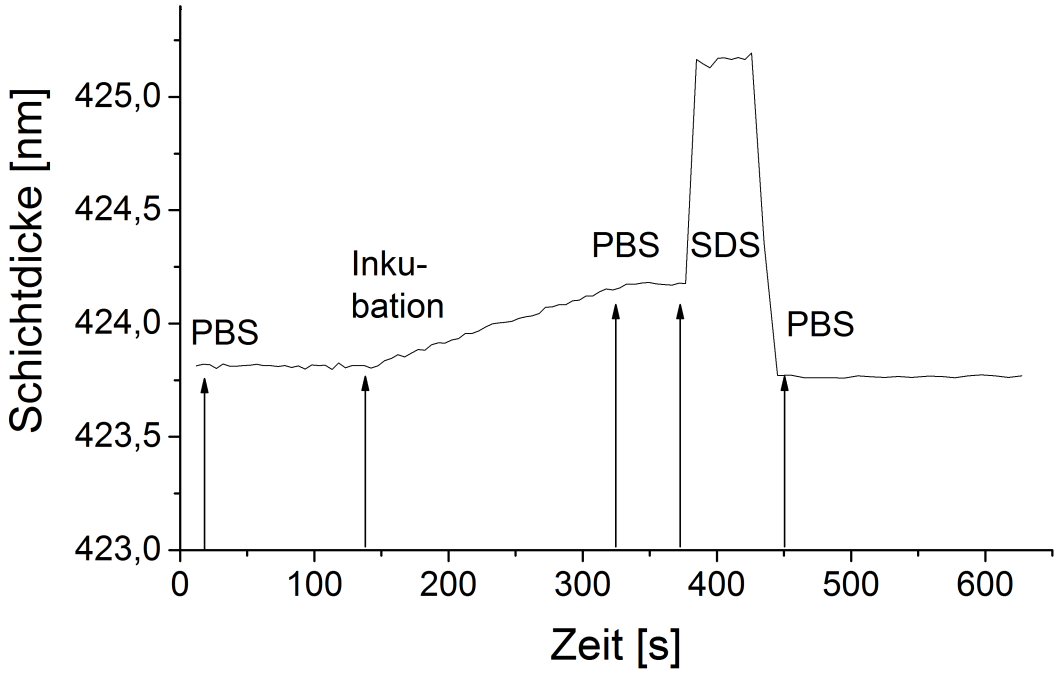


Abbildung (2.4)
Typisches RIfS Bindungssignal mit Aufnahme einer Basisline (PBS), Assoziationphase (Inkubation) und Regeneration (SDS).

2.1.5. 1λ Reflektometrie

Das Prinzip der 1-λ Reflektometrie entspricht dem der normalen spektralen RIfS (s. Abschnitt 2.1.4). Als Lichtquelle wird hier allerdings monochromatisches Licht verwendet, dessen Wellenlänge nicht zwangsläufig mit λ_0 aus Gleichung 2.19 übereinstimmt. Da aber die Intensität nach Gleichung 2.17 und 2.18 zwischen den Extrema moduliert wird, bedeutet dies bei Änderung der optischen Schichtdicke, dass mit der Änderung der Extrema-Positionen auf der Wellenlängen-Skala, ebenso eine Änderung der Intensität eines beliebigen Punktes zwischen den Extrema verändert wird (s. Abbildung 2.5). Dadurch können bei Beobachtung der Intensitätsänderung eines bestimmten Punktes auf dem Interferenzspektrum, ähnliche Informationen extrahiert werden wie für die Beobachtung der Verschiebung des Interferenzspektrums im spektralen Ansatz.

Im Gegensatz zum Photodiodenarray wird bei der 1λ-Reflektometrie für die Detektion ein CMOS oder CCD-basiertes Kamerasystem verwendet (s. Abschnitt 2.1.5). Dadurch können verschiedene Proben bzw. Rezeptoren gleichzeitig gemessen werden, wenn sie in

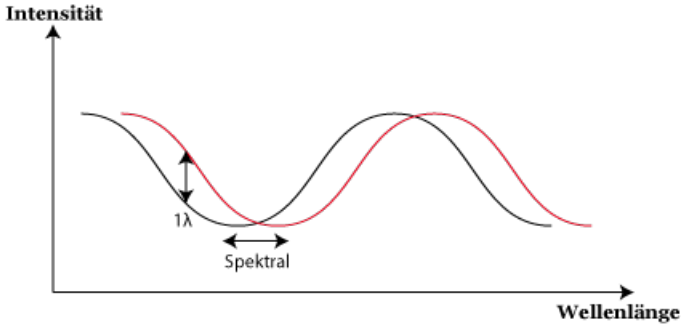


Abbildung (2.5)
Schematische Erklärung der Verschiebung zweier Interferenzspektren mit unterschiedlichen optischen Schichtdicken.

kleinen Bereichen, den sog. *Spots*, auf dem Transducer aufgebracht werden. Der fluidische Aufbau kann wie auch bei der spektralen RIfS über dieselben Pumpen- und Ventilsysteme realisiert werden. Die Auswertung erfolgt aber hier nicht über die Beobachtung des Verschiebens eines Minimums, sondern über die Änderung der Intensität eines gemittelten Spots in Bezug auf eine Referenzumgebung außerhalb des Spots (s. Abbildung 2.6 und 2.7).

INSTANT Das EU-finanzierte Projekt INSTANT soll ein Messsystem entwickeln, um Nanopartikel in Alltagsprodukten wie Sonnencremes, Orangensaft oder Wein nachzuweisen. Es stellt ein kombiniertes Messsystem aus zwei Detektionsmethoden dar. Zum einen eine optische Detektion basierend auf der 1λ-RIfS, welche die Nanopartikelanlagerung auf dem Transducer über die Intensitätsänderung der Reflektion bestimmen soll. Zum anderen wird eine elektrochemische Detektion basierend auf der Impedanzspektroskopie[2] verwendet, um die Nanopartikelanlagerung über das unterschiedliche Strom und Spannungsverhalten bei verschiedenen Wechselspannungfrequenzen bestimmen zu können (s. Abbildung 2.8).

[2]Die Impedanzspektroskopie ist das Spezialgebiet eines Projektpartners gewesen und wird in dieser Arbeit nicht theoretisch beleuchtet. Die zur Vermessung benötigten Softwaremodule sind für die Steuerungssoftware zur Verfügung gestellt worden.

Abbildung (2.6)
Kamerasystem, wie es für die iRIfS-Methode verwendet wird.

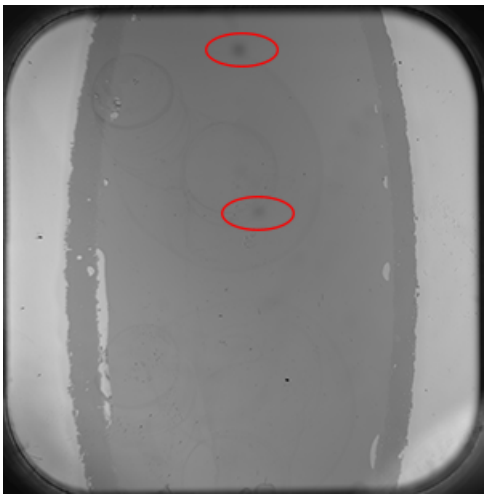


Abbildung (2.7)
iRIfS Aufnahme mit zwei Spots (rot markiert), welche bei Anbindung eines Analyten den Graustufenwert ändern. Diese Änderung kann anschließend (durch die Aufnahme vieler Bilder) wieder, wie inAbbildung 2.4 gezeigt, aufgetragen werden (rechts).

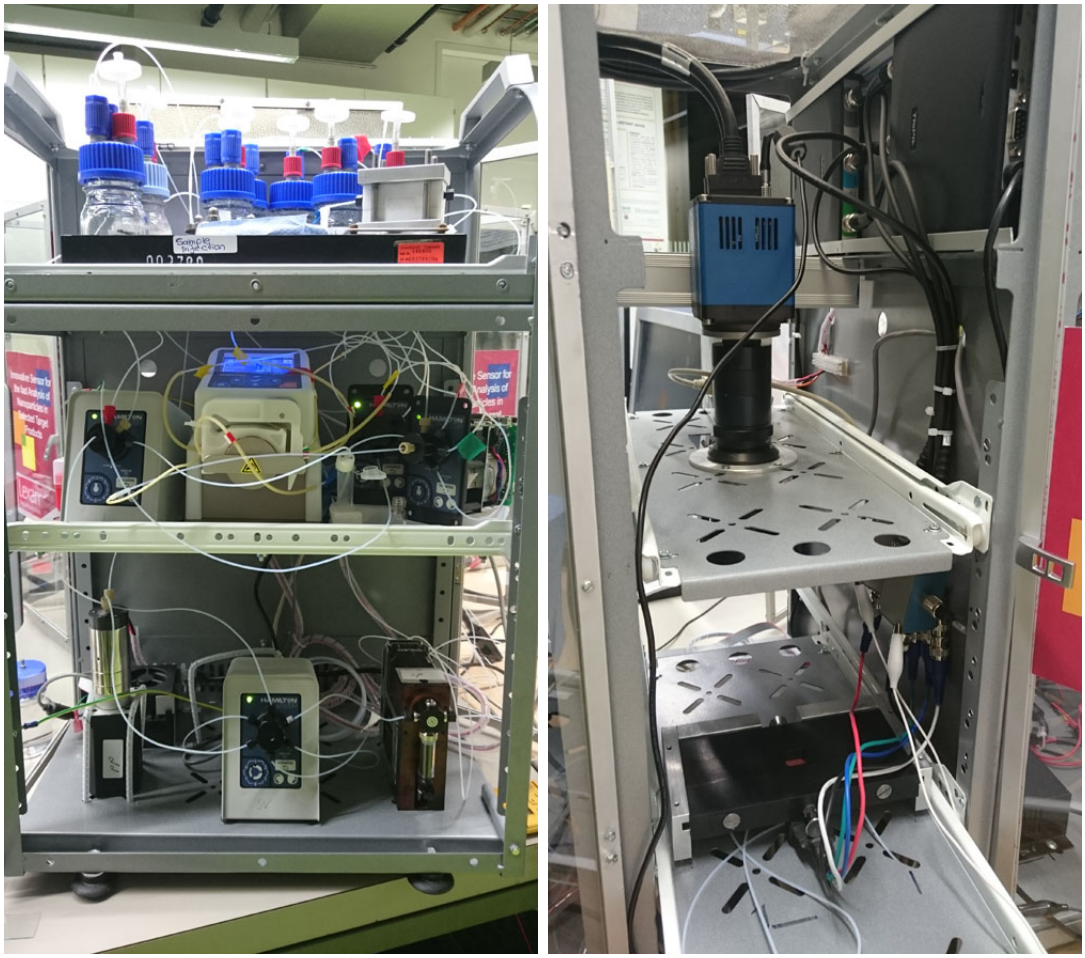


Abbildung (2.8)
INSTANT Aufbau bestehend aus Proben- und Reagenzienreservoir (links-oben), fluidischem System(links) und optischer/elektrochemischer Detektion (rechts).

2.2. Anforderung eines Messsystems aus Sicht von Forschungseinrichtungen

In modernen Forschungseinrichtungen ist die Arbeit ohne softwaregestützte Gerätesteuerungen nicht mehr vorstellbar, unabhängig davon ob es sich um Analysen- oder Syntheselabors handelt. Messsysteme und Gerätekomponenten werden im Allgemeinen durch prozessorbasierte Steuerungen angesprochen und auch die Datenspeicherung und -Auswertung verläuft über ebendiesen Weg. Hinsichtlich Effizienz und Möglichkeiten, welche derartige Steuerungssystem bieten, ist dieser Trend auch durchaus erwünscht, da er die Arbeit mit den Messsystemen erheblich vereinfacht bzw. überhaupt erst ermöglicht (wenn man nur die Menge an produzierten Daten betrachtet). Auch die Mitarbeiter der verschiedenen Forschungseinrichtungen, unabhängig von deren Qualifikation und Ausbildung, haben sich bereits an die Arbeit mit den softwarebasierten Steuerungslösungen gewöhnt.
Gerade weil es sich bei den meisten der modernen Messinstrumente um prozessorgestützte Messsysteme handelt und weil es sehr viele Gerätehersteller für die unterschiedlichsten Anwendungen gibt, ist die Fülle an Softwarepaketen, welche mit den Geräten ausgeliefert werden, vielseitig. Genau hierdurch werden leider auch die in der Einleitung dargelegten Probleme (s. Abschnitt 1.4) verursacht. Um eine Lösung für diese zu entwickeln, muss zuerst einmal verstanden werden, für wen die möglichen Lösungen einen relevanten Nutzen bringen. Dafür werden im folgenden Kapitel Zielgruppen eingeführt, welche direkt oder indirekt mit derartigen Problematiken konfrontiert werden. Weiterhin muss verstanden werden, aus welcher Sichtweise die einzelnen Zielgruppen, softwarebasierte Steuerungslösungen betrachten: Ein Informatiker beispielsweise betrachtet Steuerungslösungen auf eine andere Weise als ein Chemiker oder ein Biologe; ein Systemintegrator betrachtet sie auf eine andere Weise als ein technischer Angestellter.
Ausgehend von der Definition der einzelnen Zielgruppen werden exemplarisch typische Anwendungsbeispiele von Geräten oder analytischen Aufbauten in modernen Forschungseinrichtungen gezeigt. Daraus wird im Anschluss der methodenbasierte Ansatz abgeleitet und deren Umsetzung in einer Softwarelösung diskutiert. Der Ansatz wird letztendlich auf die in der Einleitung gezeigten Anwendungsszenarien angewendet, um die Einsatzfähigkeit einer derartigen Lösung zu demonstrieren.

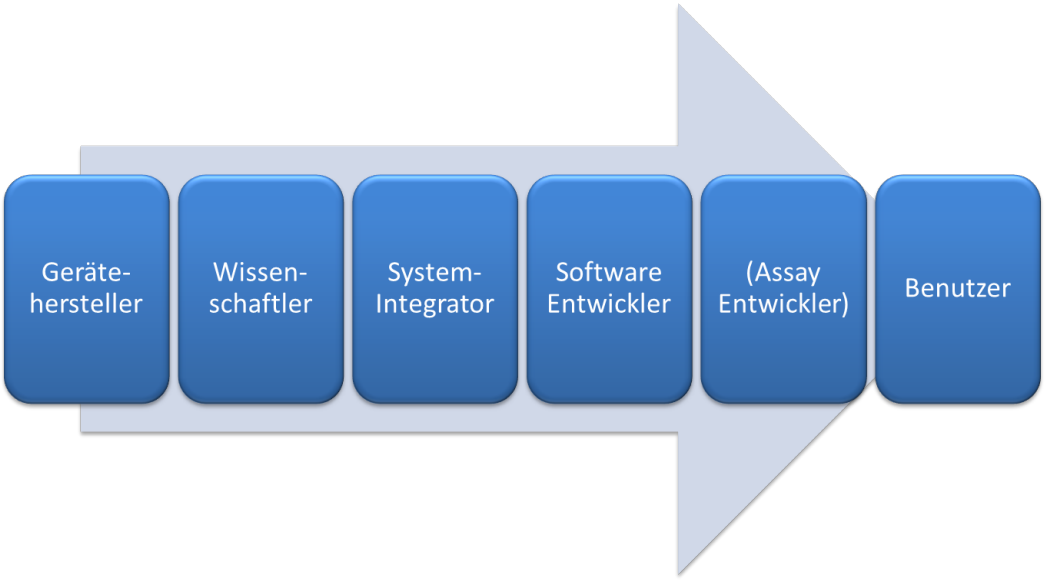


Abbildung (2.9)
Personengruppen, welche am Entstehungsprozess eines analytischen Aufbaus beteiligt sind.

2.2.1. Zielgruppen

An der Entwicklung und dem Einsatz von Steuerungssoftware für Sensoren und deren Peripherie, welche zusammen ein *automatisiertes Messsystem* bilden, sind mehrere Personengruppen beteiligt: Zum einen die Gerätehersteller, welche Teile der Peripherie konstruieren und verkaufen. Sie stellen meist auch eine zum Gerät bzw. deren Produktpalette gehörige Steuerungssoftware zur Verfügung, welche entweder beim Gerätekauf enthalten ist oder separat erworben werden kann; zum anderen die Endbenutzer, welche die fertige Steuerungslösung bedienen sollen. Beide stellen aber nur die beiden Enden der Prozesskette dar. Dazwischen befinden sich noch weitere Personengruppen, welche indirekt oder direkt am Prozess eines analytischen Aufbaus, inklusive Hard- und Softwareentwicklung, beteiligt sind (s. Abbildung 2.9).

Gerätehersteller

Der *Gerätehersteller* liefert die Peripherie-Elemente, ohne die ein Einsatz des zu entwickelnden Sensors, wie bereits in der Problemanalyse besprochen, nur schwer möglich ist. Sie stellen neben ihrem eigentlichen Produkt im Normalfall auch eine Steuerungslösung zur Verfügung, welche allerdings nur für ihre eigene Produktpalette oder gar nur für ein einziges Produkt anwendbar ist.

Wissenschaftler oder Ingenieur

Der *Wissenschaftler* oder *Ingenieur* beschäftigt sich mit der Konstruktion des Sensors bzw. dessen zugrunde liegenden physikalisch-chemischen Effekts. Er entwickelt die theoretischen und praktischen Modelle, nach welchen ein mechanischer und elektrischer Aufbau realisiert werden kann, z.B. Design von Transducern, Elektroden bzw. Opt(r)oden, Detektionsprinzip, usw.

Systemintegrator

Ein *Systemintegrator* wird für den Aufbau, die Verschaltung und die technischen Aspekte der Sensorumgebung benötigt. Er gibt den physikalischen Modellen ein „Gesicht“, verwirklicht die Hardware-Integration, entwirft Platinen und die integrierten Schaltungen, usw.

Software-Entwickler

Weiterhin muss ein *Software-Entwickler* eine Steuerungslösung für den generierten Aufbau entwerfen und implementieren, außerdem eine grafische Oberfläche für den Benutzer modellieren.

Assay Entwickler

Der *Assay Entwickler* plant und generiert Abläufe, welche sich mit dem zugrunde liegenden Messprinzip realisieren lassen.

Benutzer

Letztendlich ist der *Benutzer* derjenige, welcher mit dem analytischen Aufbau inklusive der Steuerungslösung arbeitet.

Auf den ersten Blick scheinen alle beteiligten Personengruppen unabhängig voneinander zu agieren. In der Realität kommt es aber zu einem komplexen Zusammenspiel der einzelnen Gruppen untereinander. Der Wissenschaftler ist auf den ersten Blick relativ unabhängig von den restlichen Gruppen, da sein Sensorprinzip auch ohne Peripherie oder Software funktionieren sollte. In der Realität macht seine Entwicklung natürlich nur Sinn, wenn sie in einen analytischen Aufbau eingebunden wird, der die Signale seines Sensors in Informationen umwandelt, damit sie verwertet werden können.

Der Systemintegrator ist stark von beiden eben genannten Gruppen abhängig, da er ohne Peripherie und ohne Sensor keinen Aufbau realisieren kann. Der Benutzer (und auch der Assay-Entwickler) spielt eine Sonderrolle, da er Sensor, Peripherie und Software verwendet, auch wenn die hauptsächliche Interaktion mit dem analytischen Aufbau über die Software stattfindet. Allerdings ist er am Entwicklungsprozess gar nicht oder nur in beratender Funktion beteiligt, wodurch seine Abhängigkeit ausschließlich darin besteht, die für ihn bestimmten Komponenten zu verstehen und anwenden zu können.

Software-Entwickler

Eine der sehr oft unterschätzten Aufgaben fallen dem Software-Entwickler zu. Nicht nur, dass er eine für den Benutzer grafisch intuitive und funktionale Oberfläche generieren muss, welche alle Aufgaben des analytischen Aufbaus kontrollieren, verändern

[2] wobei oftmals gerade in Forschungseinrichtungen der Assay-Entwickler und der Benutzer zusammenfallen

und darstellen kann. Er muss zusätzlich sämtliche Gerätefunktionalitäten verstehen und in einen sinnvollen Zusammenhang bringen, die Gerätekommunikation realisieren, mathematische Auswertungen für die zugrunde liegende analytische Methode spezifizieren, die Datenspeicherung implementieren und dies alles in sinnvoller Weise dem Benutzer zur Verfügung stellen. Aus diesem Grund hat er zu jeder anderen Personengruppe sehr starke Abhängigkeiten, da er ohne regelmäßige Rücksprachen nur schwer die benötigten Merkmale in ein informatisches Modell übersetzen kann. Wie bereits in der Einleitung angesprochen ist aber die Software-Entwicklung genau die Aufgabe, welche oft ausgelagert oder gar einer anderen Personengruppe wie dem Wissenschaftler oder dem Systemintegrator als Nebenaufgabe aufgetragen wird, die oft nicht über das benötigte Knowhow verfügen. Dieses Vorgehen kann zu gravierenden Konsequenzen führen, welches zu fehlerhafter oder nicht funktionierender Steuerungssoftware führen kann.

Die oben eingeführten Personengruppen werden nun in Zielgruppen eingeordnet und es wird erklärt, worum es sich bei den einzelnen handelt und was ihre Aufgaben sind.

Systemprogrammierer

Der Systemprogrammierer ist für das Design und die Implementierung von Software zuständig. Er verfügt über sehr detailliertes Wissen über Betriebssysteme, Programmierung und Informatik allgemein, zusätzlich von der Gestaltung grafischer Oberflächen.
Als Haupt-Akteur ist hier der Software-Entwickler zu finden, welcher die Softwarelösung für ein zugrunde liegendes analytisches Problem bereit stellen soll. Zusätzlich kann hier auch der Gerätehersteller angesiedelt werden, welcher im Normalfall für seine Produkte eigene Softwarelösungen entwickelt. Letzteres ist natürlich nicht ganz präzise, da im eigentlichen Sinn die Entwicklung eines Peripherie-Gerätes der Entwicklung eines analytisches Aufbaus ähnelt, in welchem Gerätekomponenten zu einem zusammengesetzten System entwickelt werden. Die oben dargelegte Prozesskette ist deshalb auch für jedes seiner Produkte anwendbar. Für hiesige Betrachtungsweise soll der Gerätehersteller allerdings eine „Person“ darstellen, welche eine „Black-Box“ in Form einer Gerätekomponente für den analytischen Aufbau zur Verfügung stellt.

Systemintegrator

Unter dem Systemintegrator soll hier der Entwickler einer analytischen Methode verstanden werden, also jener Personengruppe, welche ein analytisches Problem zu einen fertigen Aufbau bringt. Eine genaue Definition einer analytischen Methode wird in Abschnitt 2.2.2 erbracht.
Unter dieser Zielgruppe sind der Wissenschaftler, natürlich der Systemintegrator selbst und der Assay-Entwickler anzusiedeln, welche die benötigten Sensoren entwickeln, mit Hilfe der Peripherie zu einem System zusammensetzen und ein Protokoll für die Bedienung definieren.

Benutzer

Der Benutzer ist bezeichnend für den bereits oben eingeführten „Benutzer“, welcher den analytischen Aufbau bedienen und Messungen darauf durchführen soll. Er ist im Extremfall kein analytisch ausgebildeter Fachmann und erwartet als Steuerungslösung eine „Push-the-Button“-Steuerungssoftware, bei welcher komplexe Einstellungen hinsichtlich analytischer Methoden- und Geräte-Konfigurationen vermieden werden sollen.

2.2.2. Analytische Methoden im Fokus

Nach dem Duden ist eine Methode ein „*auf einem Regelsystem aufbauendes Verfahren zur Erlangung von [wissenschaftlichen] Erkenntnissen oder praktischen Ergebnissen*“. Ausgehend davon lässt sich der in dieser Arbeit definierte analytische Aufbau exakt auf diese Methodendefinition abbilden. Der analytische Aufbau, bestehend aus einem oder mehreren Sensoren in einem Umfeld aus Peripherie Elementen bildet das *Regelsystem*, welches mit Hilfe eines Ablaufs und einem mathematischen Datenauswertungs-Algorithmus, dem *aufbauenden Verfahren*, wohl definierte physikalisch-chemische Eigenschaften detektiert und in Informationen, also *wissenschaftliche oder praktische Erkenntnisse* umwandelt.

Generelle Überlegungen

Der Systemintegrator betrachtet den von ihm zu entwickelnden Aufbau nicht als Kombination aus Einzelaspekten wie u.a. Geräte-Zusammenschluss oder Hardware-Integration,

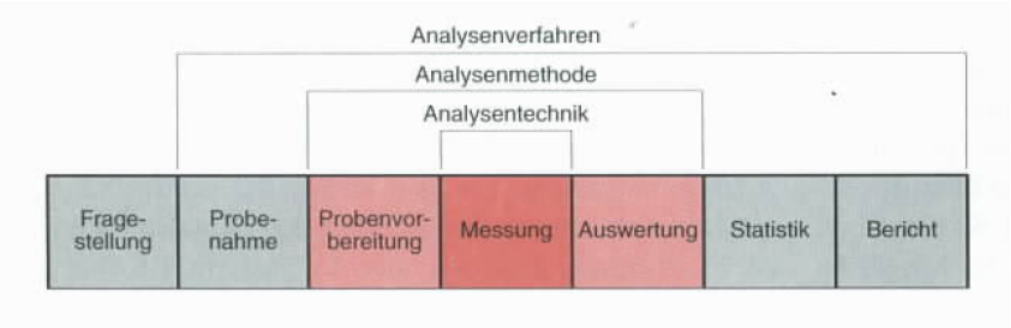


Abbildung (2.10)
Teilaspekte eines analytischen Verfahrens nach [Cam01].

usw. wie es sich ein Software-Entwickler vorstellt, sondern als Lösungsansatz zur Beantwortung einer analytischen Fragestellungen, wie z.B.:

- Fluoreszenzbasierte Messung von Mikroorganismen in Flusswasser
- Gehalt von Kohlenstoffmonoxid in der Luft von Wohnungsanlagen
- Bestimmung der Glucosemenge in Blut
- Materialzusammensetzung von gehärteten Legierungen
- uvm.

Ausgehend von einer konkreten Fragestellung versucht der Systemintegrator einen analytischen Aufbau zu entwickeln, welcher eine Antwort auf jene Frage bringen kann. Dazu überlegt er sich mögliche (physikalische) Größen, welche sich während des betrachteten Prozesses indirekt oder direkt ändern, d.h. er muss feststellen, welche (physikalische) Sonde, wie Licht, Elektrizität, Masse usw. mit der Änderung dieser Eigenschaft am empfindlichsten beeinflusst wird. Der analytische Aufbau, welcher in der Lage ist, die Änderung dieser Eigenschaft zu detektieren, wird als *Analysentechnik* bezeichnet, und sie bildet den Kern eines *Analysenverfahrens*, welches exemplarisch in Abbildung 2.10 dargestellt ist. Werden organisatorische Aspekte bei der Entwicklung eines analytischen Verfahrens in erster Linie vernachlässigt, reduziert sich der Prozess auf die Erstellung einer *Analysenmethode*. Dabei werden zusätzlich zur eigentlichen Analysentechnik die Ein- und Ausgaben definiert, d.h. die Probenvorbereitung und Auswertung. Das heißt im Umkehrschluss, dass eine Analysenmethode aus technischen Gesichtspunkten ausrei-

chend ist, um eine bestimmte Fragestellung wissenschaftlich zu beantworten.

Eigenschaften und Operationen von Analysenmethoden

Wird eine Analysenmethode in der Praxis betrachtet, muss sie die folgenden Eigenschaften aufweisen:

- Sensoren und Peripherie (Laborgeräte) zur Messung der in der Fragestellung geforderten Stoffe
- Steuerungseinheit zur Interaktion und Kontrolle für den Benutzer
- Programme zum sequenziellen Ausführen von Kommandos
- Auswertungsalgorithmen für die entstehenden Daten
- Speicherung generierter Informationen für spätere Statistiken und Berichte

Auf die einzelnen Eigenschaften wird in den folgenden Kapiteln der Reihe nach eingegangen und zusätzlich werden typische praxisbezogene Beispiele gezeigt.

Laborgeräte und deren Schnittstellen

Typische Laborgeräte Eine typische Auswahl an Laborgeräten ist nur schwer treffen. Als erstes Einschränkungskriterium ist hierbei die Art des Labors zu definieren, d.h. ob chemische, biologische oder etwa medizinische Laboratorien betrachtet werden. Aber auch wenn diese Vielfalt eingeschränkt wird, ist es noch immer sehr schwer möglich, eine umfassende Liste wiederzugeben. Trotzdem soll hier eine Auswahl getroffen werden, um exemplarisch darzulegen, mit welcher Art von Geräten gerechnet werden kann. Das Augenmerk soll hauptsächlich darauf gelegt werden, welche Art von Geräten sinnvoll kombiniert und über welche Schnittstellen mit ihnen kommuniziert werden kann. Es ist auch wichtig zu verstehen, dass nicht jedes Laborgerät, welche der hier benötigten Definition genügt, d.h. *elektrisch ist und über eine Schnittstelle mit einem Computersystem gesteuert und geregelt werden kann*, sinnvoll in den zu entwickelnden analytischen Aufbau eingebunden werden sollte. Als Beispiel dafür ist ein Trockenschrank genannt, welcher in fast jeder chemischen Laborumgebung gefunden werden kann. Zwar verfügen alle modernen Modelle über Computerschnittstellen, mit welchen Statusinformationen ausgelesen und ebenso Steuerroutinen ausgeführt werden können. Ob dieser aber sinnvoll in ein

Automatisierungskonzept eingearbeitet werden sollte, ist auf der anderen Seite fraglich, weil eine manuelle Verwendung des Geräts unvermeidlich ist. Als weiteres Beispiel sind Zentrifugen ohne automatische Probenzuführung genannt. Auch hier ist es sinnvoller, die Einstellung direkt am Gerät vorzunehmen.
Anhand dieser Beispiele ist gezeigt, dass es nicht unbedingt sinnvoll ist, jede Art von Gerät in ein Automatisierungskonzept aufzunehmen, nur weil es prinzipiell möglich ist.

Im Folgenden ist eine (subjektive) Auswahl an Geräten gezeigt, bei welchen es abhängig von der zu entwickelnden Analysenmethode sinnvoll erscheinen mag, sie in einen automatisierten Aufbau aufzunehmen:

1. Sensoren: Beschleunigung, Masse, Chemo-/Biosensoren, ...
2. Messgeräte: Spektrometer, Refraktometer, Spannungs-/Strommessgeräte, Kameras, ...
3. Probezuführungs- und -transport-Systeme (Fließinjektions-Analysensysteme, Abk. FIA): Pumpen, Ventile, Autosampler, ...
4. Licht- und andere regelbare Anregungsquellen: Laser, Röntgenröhren, ...
5. Temperierung: Kühlgeräte, Öfen und Heizelemente, usw.
6. Ausgabegeräte: Oszilloskope, EKGs (Elektrokardiogramme), ...
7. Medizinische und diagnostische Systeme: Dialyse, Herz-Lungen-Maschine, Ultraschall, ...

Die Gründe, ebendiese Geräte zu automatisieren sollen wie folgt gezeigt werden.

Sensoren & Messgeräte

Die erste und zweite Gruppe, die Sensoren und Messgeräte, beinhaltet im weitesten Sinne Geräte oder Gerätekomponenten, welche Daten produzieren; unabhängig davon, ob diese Daten als numerische Liste, als Bild oder einfacher Zahlenwert vorliegen. Diese Daten müssen in standardisierter Weise gelesen, aufbereitet, abgespeichert und dem Benutzer dargestellt werden. Für derartige Aufgaben sind Computersysteme prädestiniert.

FIA

Die dritte Gruppe, die Probentransportsysteme, sind Geräte (-komponenten) welche Proben, unabhängig davon, ob es sich um Flüssigkeiten, Feststoffe oder Gase handelt, von einem Ort zu einem anderen transportieren bzw. den zu verwendenden Weg modifizieren. Bei ebendiesen zeigt ein standardisiertes Automatisierungskonzept seine wahren Stärken, da gerade die Transportwege sehr dynamisch sind und ein enges Zusammenspiel aller beteiligten Geräte bedingen.

Auch die nächste Gruppe der Anregungsquellen lässt sich sinnvoll in ein Automatisierungskonzept aufnehmen, beispielsweise zur Steuerung von Laser-Shuttern oder integrierten Monochromatoren, aber auch zur einfachen Einstellung von Strahlungsintensitäten. *Anregungsquellen*

Für die fünfte Gruppe, die Elemente für Temperierung, bietet sich ein Automatisierungskonzept an, wenn beispielsweise bestimmte Temperaturprofile zeitlich genau getriggert werden sollen. *Temperierung*

Ausgabegeräte der sechsten Gruppe sind zur Anzeige von Daten vorgesehen, wofür es wiederum abhängig vom Zweck die verschiedensten Formate gibt. Hier sind Geräte gemeint, welche ein bestimmtes zu messendes Signal in eine ganz bestimmte Ausgabeform umwandeln (z.B. EKG). *Ausgabe*

Die letzte Gruppe ist ein sehr großes und weitreichendes Gebiet, in welcher Geräte in Form von Komplettlösungen enthalten sind. Für derartig integrierte Systeme ist eine Ablaufsteuerung fundamental, da es sich um mehrere Gerätekomponenten in einem engen Zusammenspiel handelt, die aufeinander reagieren und miteinander interagieren müssen. *Diagnostische Systeme*

Die hier getroffene Auswahl ist natürlich nicht vollständig und kann beliebig um weitere Gerätekategorien erweitert werden. Diese Auswahl soll aber genügen, um die Konzepte innerhalb dieser Arbeit nachvollziehen zu können.

Typische Schnittstellen Im Folgenden wird darauf eingegangen, über welche Computer-Schnittstellen die obigen Geräte verfügen. Betrachtet man die Auswahl genauer (und vernachlässigt in erster Näherung die „Messgeräte-Gruppe“), so kommt im Laborumfeld häufig die serielle RS-232 Schnittstelle vor, da meist nur sehr wenig Befehle nötig sind und der Datendurchsatz bei diesen Geräten gering ist. Es genügt in den meisten Fällen eine kleine Liste an Steuerbefehlen bzw. Abfragebefehlen für Geräte-Status, um den kompletten Funktionsumfang ebendieser abzudecken. Dabei ist die sehr einfach gehaltene RS-232 Schnittstelle bestens geeignet, die mit wenig Konfigurationen Daten von und zum Gerät übertragen kann. *RS232*

Betrachtet man die o.g. Gruppe der Messgeräte, erhöht sich v.a. der Datendurchsatz (abhängig von der Art des Geräts). In vielen Fällen ist die RS-232 Schnittstellen nach wie vor ausreichend, so zum Beispiel für die meisten Spektrometer. Sollen jedoch sehr viele Daten in kurzer Zeit vom Gerät empfangen werden, stößt diese mit maximal 115200 Baud schnell an ihre Grenzen. Hier kommen die Vorteile der schnelleren und performan-

teren Schnittstellen wie USB und GPIB zum tragen. Gerade bei Geräten wie Strom-
GPIB oder Spannungsmessgeräten kommt sehr oft die letztgenannte GPIB-Schnittstelle vor. Sie zeichnet sich durch eine relativ einfache Konfiguration und einen hohen Datendurchsatz aus. Darüber hinaus benötigt sie keinen speziellen Gerätetreiber, die Daten werden in den meisten Fällen RAW vom und zum Gerät übertragen. Zusätzlich handelt es sich um ein BUS-System, mit welchem sich mehrere Geräte hintereinander kaskasdieren lassen. Bei maximal 15 Geräten ist aber die Beschränkung der Schnittstelle erreicht, wobei aber in der Praxis dieses Limit selten überschritten wird.

USB Viele Messgeräte verfügen mittlerweile über eine USB-Schnittstelle, welche über einen sehr hohen Datendurchsatz von mehreren GBit verfügt. Hierbei ist von Nachteil, dass für jedes angeschlossene Gerät ein eigener Gerätetreiber benötigt wird. Da die meisten Messgeräte nicht in der Liste der Standard USB-Geräteklassen (s. Abschnitt 3.1.2) aufgenommen sind, muss allein aus Gründen der zum Teil sehr unterschiedlichen Funktionalitäten, zwingend für jedes Messgerät ein Treiber mitgeliefert werden. Dennoch ist es möglich, standardisierte Treiber zu verwenden, um Daten im RAW-Format an das Gerät zu senden oder von diesem zu übertragen.

LAN In der Zwischenzeit finden sich immer häufiger Messgeräte mit einer LAN-Schnittstelle. Einer der größten Vorteile von Netzwerk-Schnittstellen ist die nahtlose Integration in bestehende Firmennetzwerke, mitunter die einfache und standardisierte Adressierung über IP. Gerade diese einfache Integration und umfassende Kompatibilität zu vielen Systemen, bringt aber auch einen großen Nachteil mit sich. Die Kommunikation mit derartigen Geräten über die LAN-Schnittstelle muss immer abgesichert werden, d.h. es muss auf eine verlässliche Autorisation und Authentifikation geachtet werden.

Befehle und Ablaufsequenzen

Befehle Während bei Laborgeräten noch ein Auszug typischer Geräte vorgestellt werden kann, ist eine Verallgemeinerung bei Ablaufsequenzen und Gerätekommandos aufgrund der enormen Vielfalt überhaupt nicht mehr möglich. Aus diesem Grund soll dieses (und auch die beiden nächsten) Kapitel typische Befehle und Abläufe auf Basis der im letzten Kapitel dargelegten Anwendungsbeispiele exemplarisch präsentieren.

Messgeräte Zwei fundamentale Befehle stellen natürlich das „Messen“ und das „Überwachen“ dar, welche ein Messgerät wie ein Spektrometer, Spannungs-/Strommessgerät oder eine Kamera anweisen, entsprechend ein Spektrum, einen Spannungs-/Stromwert oder ein Bild aufzunehmen. Das Überwachen verhält sich dabei ähnlich, außer dass permanent Mes-

sungen innerhalb eines festgelegten Intervalls durchgeführt werden.
Weitere typische Befehle dieser Gruppe sind Konfigurationsbefehle beispielsweise hinsichtlich der Integrationszeit, das Einstellen des Wiederholungsintervalls oder der Wellenlängenbereiche. Wie bereits an dieser Stelle ersichtlich, werden hiermit hauptsächlich Geräte der Gruppe „Messgeräte“ spezifiziert. Sensoren lassen sich im Großen und Ganzen ebenfalls in dieser Gruppe einordnen, da sie im Normalfall auch ein elektrisches Signal als Ausgangsgröße zur Verfügung stellen.

Sensoren

Fließinjektions-Analysensysteme benötigen Befehle, welche einerseits einen Fluss in Bewegung setzen, wie z.B. „Pumpen“, „Aufziehen“ oder „Ablassen“ oder andererseits Befehle um einen Flussweg zu ändern, z.B. „Ventilposition“, „Kanal“, „Öffnen“ oder „Schließen“.

FIA

Anregungsquellen müssen einerseits „Angeschaltet“ oder „Ausgeschaltet“ werden können, andererseits müssen „Intensitäten“ oder Strahlungscharakteristiken wie „Wellenlänge“, „Pulsweite“, „Pulslänge“ oder „Strahlungsprofile“ angepasst werden können.

Anregungsquellen

Temperierungselemente müssen ebenso „Ein-“ und „Ausgeschaltet“ werden können und es müssen Einstellungen hinsichtlich „Temperatur“ oder „Temperaturrampen“ zur Verfügung stehen.

Temperierung

Ausgabegeräte sind die wohl variantenreichste Gruppe. Natürlich muss der Befehl „Ausgabe“ definiert sein, allerdings können Ausgaben auf fast beliebige Art und Weise formatiert und dargestellt werden können, welche auch sehr stark abhängig sind vom betrachteten Ausgabegerät und dessen Funktionsweise.

Ausgabe

Zuletzt ist die Gruppe der diagnostischen Systeme sehr komplex und es lässt sich kaum ein repräsentativer Befehl darstellen, da es sich bei diesen Geräten meist um kombinierte Lösungen handelt, welche aus allen oben genannten Gruppen bestehen. Aus diesem Grund gelten alle bisher genannten Befehle als Ganzes auch für diese Gruppe, die in einer bestimmten Ablaufsequenz das diagnostische Gerät zum Leben erwecken.

Diagnostik

Ablaufsequenzen Beispielhafte Ablaufsequenzen beinhalten mindestens zwei Geräte (Komponenten) aus obigen Gruppen in einem Verbund. An diese soll in einer wohl definierten Art und Weise Befehle gesendet werden könnnen. Hierfür können die in der Bioanalytik geläufigen Testformate herangezogen werden. Wie bereits in den theoretischen Grundlagen besprochen, werden in der Bioanalytik die Interaktionen von biologischen Makromolekülen und ihre Veränderungen untersucht. Darunter fallen Proteine, Nukleotide, Kohlenhydrate, Lipide und Membranen. Die Analytik beruht auf Signalen aus biologischen Interaktion zwischen einem Rezeptor und seinem korrespondierenden

Liganden, wobei verschiedene Testformate zum Einsatz kommen können:

- Direktes Testformat: Hier wird ein Ligand aus dem Probengefäß ohne Zwischenlagerung über den mit Rezeptor belegten Transducer geleitet, vermessen und anschließend verworfen. Befehlssequenz: Pumpe an ≫ Überwachen ≫ Warten ≫ Pumpe aus
- Kompetitives Testformat: Hier wird ein Ligand mit einer wohldefinierten Menge eines markierten *Liganden* gemischt und die Mischung über den mit Rezeptor belegten Transducer geleitet, vermessen und anschließend verworfen. Befehlssequenz: Ventil schalten ≫ Pumpe an ≫ Warten ≫ Pumpe aus ≫ Mischen ≫ Ventil schalten ≫ Pumpe an ≫ Überwachen ≫ Warten ≫ Pumpe aus
- Bindungshemmtest: Hier wird der Ligand über eine gewisse Zeit mit einer wohl definierten Menge an *Rezeptor* vorinkubiert und diese Mischung über den mit Ligand belegten Transducer geleitet, vermessen und anschließend verworfen. Befehlssequenz: Ventil Schalten ≫ Pumpe an ≫ Mischen ≫ Pumpe aus ≫ Warten ≫ Ventil schalten ≫ Pumpe an ≫ Überwachen ≫ Warten ≫ Pumpe aus
- Sandwich: Hier wird ein Ligand über den mit Rezeptor belegten Transducer geleitet und verworfen, anschließend wird ein Sekundär-Rezeptor (meist markiert) über den mit Rezeptor + Ligand belegten Transducer geleitet, vermessen und verworfen. Befehlssequenz: Ventil schalten ≫ Pumpe an ≫ Warten ≫ Pumpe aus ≫ Ventil schalten ≫ Pumpe an ≫ Überwachen ≫ Warten ≫ Pumpe aus

Diese und ähnliche Sequenzen können in Laboratorien außerhalb der Bioanalytik erwartet werden. Je nach Anzahl und Komplexität der Vorbehandlungsschritte, können die Ablaufsequenzen schnell kompliziert und umfangreich werden. Auch müssen die Timings der Wartezeiten sehr präzise auf den jeweiligen Ablauf zugeschnitten sein. Die Aufgabe des Erstellens von derartigen Ablaufsequenzen liegt im Aufgabenbereichs des Assay-Entwicklers oder des Benutzers (nach Abbildung 2.9).

Labormessdaten und deren Auswertung

Labormessdaten hängen vom jeweiligen Gerät ab, außerdem vom betrachteten Labor und dessen Spezialisierung. Gruppen, welche sich mit optischer Spektroskopie beschäftigen, erwarten als Labormessdaten *Spektren*, d.h. Zahlenlisten aus Detektor-Intensitätswerten, die mit einer Liste aus Lichtfrequenzen korreliert werden; der Frequenzbereich und die

Zahlenlisten

Intensitätswerte (inkl. deren Einheit) sind dabei abhängig vom betrachteten Spektralbereich und der zugrunde liegenden Analysentechnik.

Andere Gruppen, die sich überwiegend mit der Mikroskopie beschäftigen, worunter viele biologisch orientierte Gruppen fallen, erwarten als Messdaten *Bilder* in einem der gängigen Bildformate (raw, jpg, png) oder auch proprietären Datenformaten. Viele moderne Mikroskope produzieren anstatt eines Wertes pro Bildpunkt (bzw. 3 für die einzelnen Farben RGB bei Farbmikroskopen) zusätzlich eine spektrale Zerlegung jedes Pixels, wodurch pro Bildpunkt ein komplettes Spektrum gespeichert wird anstatt nur ein bzw. drei Werte.

Bilder

Gruppen in der Elektrotechnik, Ingenieurstechnik, aber auch in der Sensorik, messen häufig Spannungs- bzw. Stromwerte, die zu einem ganz bestimmten Zeitpunkt zwischen zwei Punkten eines Aufbaus anfallen. Weiterhin sind hier sehr oft Spannungs-Strom-Korrelationen von Bedeutung, d.h. wie sich der Betrag des Strom bei Änderung der angelegten Spannung oder die Phase zwischen beiden zu einem gewissen Zeitpunkt bzw. über einen gewissen Zeitraum hinweg verändert. Auch hierbei handelt es sich um Zahlenlisten, bestehend aus Korrelationen zwischen Spannung, Strom und eventuell der Zeit.

Zahlenwerte

Im medizinischen Umfeld findet sich wieder Fülle an Möglichkeiten. Hier können sämtliche Datensätze wie Bilder, elektrische Signale, aber auch Spektren erwartet werden. Als Beispiel seien Ultraschall- oder kernspintomographische Bilder genannt, welche an Datensätze aus der Mikroskopie erinnern.

Die Zahlenlisten, welche aus spektroskopischen oder Strom-Spannungs-Korrelationen erwartet werden, können einerseits statistisch ausgewertet werden, d.h. sie werden statistischen Funktionen (Mittelwertbildung, Standardabweichungen, Basislinienkorrektur, uvm.) unterworfen und werden anschließend ausgewertet. Andererseits können sie direkt oder nach Verrechnung mit anderen Datensätzen grafisch dargestellt werden, worauf (manuelle) grafische Auswertungsroutinen angewendet werden können.

Bilder sind im eigentlichen, informatischen Sinn auch Zahlenlisten, welche allerdings in Zeilen, Spalten und Pixelwerte unterteilt werden, d.h. ein Farbbild enthält eine Liste aus n (Spalten) x m (Zeilen) x rgb (3 Farben pro Pixel) Werten. Typische Auswertungsstrategien sind neben der Darstellung und Speicherung, oft das Ausschneiden bestimmter Bereiche (sog. Region of Interests), Filterungen (Gauß, Median, usw.), Weiß- und Gamma-Wert Korrekturen, uvm.

Arten der Datenspeicherung

Hinsichtlich Datenspeicherung findet sich in Forschungseinrichtungen sehr häufig die sog.
Dropbox *Dropbox-Speicherung*, d.h. Daten werden auf dem lokalen Rechner abgelegt, an welchem das bediente Gerät angeschlossen ist. Im Umkehrschluss bedeutet dies aber auch, dass die Daten immer verstreut auf vielen Rechnern zu finden sind. Über Netzlaufwerke können die Daten zwar anschließend zentral verfügbar gemacht werden, was allerdings oft manuell vom jeweiligen messenden Benutzer gemacht werden muss.

Zentral Automatisierte Datenspeicherung auf FTP oder WebDAV-Servern ist seltener oder nur bei vollständig integrierten Laboranlagen bzw. im industriellen Umfeld zu finden, hauptsächlich weil die meisten Softwareanwendungen dies nicht von Haus aus unterstützen. Der Grund hierfür liegt in der Tatsache, dass für Software, die dediziert für einen bestimmten Systemaufbau entwickelt wird, es meist zu aufwendig ist, eine Vielzahl an externen Ressourcen anzubinden.
Beide oben genannten Servertypen (oder die sicheren Ableitungen davon) bieten die Vorteile der zentralen Datenspeicherung und -ablage, und damit unter anderem automatisierte Datensicherungen, geregelter Zugriff und zuletzt auch ein kontrollierbarer bzw. gut administrierbarer Datenbestand.

Zusammenfassung der Erkenntnisse

Anhand konkreter Beispiele für Geräte, Gerätegruppen, typische Befehle und Abläufe, deren produzierte Daten, Auswertestrategien und Datenspeicherung, kann eine konkrete Vorstellung erlangt werden, was in einem typischen Laborumfeld zu erwarten ist. Anhand dessen kann im folgenden Abschnitt der methodenbasierte Ansatz abgeleitet werden.

2.2.3. Methodenbasierter Ansatz

Abbildung einer Analysenmethode in der Informatik

Die Aufgabe besteht darin, den oben beschriebenen analytischen Methodenansatz in ein informatisch gestütztes Model zu übersetzen. Bei einem analytischen Aufbau handelt es sich um Geräte und Gerätekomponenten mit sehr individuellen Befehlssätzen/-syntaxen. Es muss ein Konzept gefunden werden, Analysenmethoden unabhängig von Geräten gestalten zu können. Dafür werden die sog. *Gerätekategorien* eingeführt, welche Geräte in

Gruppen mit bestimmten Funktionalitäten einteilen. Darüber hinaus sind Ablaufsequenzen (wie z.B. Assay-Abläufe) abhängig von der zugrunde liegenden Analysenmethode. Darauf wird im nächsten Kapitel eingegangen.
Da es sich bei Ablaufsequenzen um rein logische Abläufe handelt, die sequenziell oder parallel Gerätebefehle in einer definierten Reihenfolge abarbeiten, soll ein Konzept vorgestellt werden, alle Zusammenhänge zwischen Geräten und Ablaufsequenzen in eigenständige, vollständig beschriebene Modelle zu übersetzen. Dadurch wird es möglich, Steuerungssoftware ohne Programmierkenntnisse zu konfigurieren anstatt in relativ komplexen und teilweise hardwarenahen Programmiersprachen zu implementieren. Eine Einleitung in dieses Konzept soll Abschnitt 2.2.3 geben.
Die Konfigurationen für eine Analysenmethode gelten auch für andere Systeme und Anlagen mit anderen analytischen Aufbauten, solange es sich bei ihnen um dieselbe Analysenmethode handelt. Das bedeutet, dass auch Ablaufsequenzen einer Analysenmethode unabhängig vom zugrunde liegenden analytischen Aufbau sind. Auf die Übertragbarkeit von Konfigurationen geht Abschnitt 2.2.3 ein.

Ablaufsequenzen und Gerätekategorien

Analysenmethoden bilden hauptsächlich eine Strategie ab, wie ausgehend von einer Probe ein analytisches Ergebnis ermittelt werden kann. Die *Analysentechnik*, also die Generierung von Daten aus dem Zusammenschluss von Geräten und Gerätekomponenten, ist nur ein Teilgebiet davon. Beispielsweise liefert die *Technik* „RIfS“ immer ähnliche Ergebnisse, unabhängig davon ob eine peristaltische Pumpe oder eine Spritzenpumpe verwendet wird, und auch unabhängig davon, welches Assayformat verwendet wird.
Die *Analysenmethode* „RIfS“ ändert sich dahingehend, dass abhängig von Gerätetypen und deren Verschaltung andere Assayformate realisierbar sind. Beispielsweise lässt sich mit nur einer Pumpe zwar ein direktes Testformat messen, ein Sandwich-Assay (s. Abschnitt 2.2.2) allerdings nicht ohne manuellen Zwischenschritt. Dafür wird mindestens ein Ventil mit verschiedenen Proben- und Vorratsgefäßen benötigt. Es ist bereits jetzt erkennbar, dass es für die Analysenmethode „RIfS“ keine Rolle spielt, *welche* Pumpe, *welches* Ventil und *welches* Diodenzeilenspektrometer verwendet wird, solange „gepumpt“, „Spektren aufgezeichnet“ und „Ventile geschaltet“ werden können. Geht man einen Schritt weiter, können für eine bestimmte Analysenmethode *alle* Geräte eingesetzt werden, welche die Operation „Pumpen“, „Spektren aufzeichnen“ und „Ventile schalten“ unterstützen. Außerdem kann in dem analytischen Aufbau „RIfS“ jedes Gerät durch ein äquivalentes

Ablaufsequenzen

Gerät ersetzt werden, ohne dass die Analysenmethode selbst geändert werden muss, solange das Gerät dieselben Operationen unterstützt.

Gerätekategorien

Diese äquivalenten Geräte gehören einer (oder mehrerer) Gruppen an, welche im Folgenden als *Gerätekategorien* bezeichnet werden sollen. Gerätekategorien definieren Operationen und Eigenschaften, welche ein Gerät unterstützen *muss*, damit es durch ein anderes Gerät derselben Kategorie ohne Änderung der Analysenmethode ersetzt werden kann.

Die Definition und der Einsatz von Gerätekategorien hat große Vorteile, da einmal entworfene Ablaufsequenzen, nicht mehr (oder nur noch geringfügig) geändert werden müssen, wenn sie z.B. auf äquivalente Aufbauten übertragen werden sollen. Das ist möglich, solange Eigenschaften und Operationen der Gerätekategorie verwendet werden, anstelle derer aus dem oftmals großen und unübersichtlichen Befehlssatz der Geräte selbst. Auch die grafische Oberfläche muss für Analysenmethoden nicht angepasst werden, da sich die Analysenmethode auch bei vollkommen anderen Geräten und -herstellern nicht ändert.

Konfiguration statt Implementierung

Außer den Ablaufsequenzen müssen ebenso Auswertungsstrategien innerhalb der Analysenmethoden definiert werden. Da die Analysentechnik die Daten aus einem Messaufbau in Rohform liefert, müssen aus diesen mit Hilfe mathematischer Funktionen Informationen extrahiert werden. Wie oben besprochen, werden aus gängigen Messgeräten Datenwerte und Datenlisten erwartet. Die Daten sollen mit Hilfe eines mathematischen Skripts innerhalb eines externen Prozesses berechnet werden. Damit ist gewährleistet, dass die mathematischen Routinen nicht in den Quelltext integriert werden müssen und abhängig von der verwendeten Analysenmethode verändert werden können, ohne die Steuerungssoftware neu kompilieren zu müssen.

Wird darüber hinaus ein Template erstellt, über welches die Informationen adressiert, strukturiert und formatiert werden können, muss auch die Datenausgabe nicht mehr fest in den Quelltext integriert werden. Es muss nur konfiguriert werden, wohin die Daten geschrieben werden sollen und welches Template für die Strukturierung verwenden werden soll (s. Abschnitt 3.3.7).

Diesem Ansatz folgend werden, angefangen bei der Sequenzierung von Abläufen bis zur strukturierten Datenausgabe, keine Programmierkenntnisse benötigt. Dafür muss eine Grammatik entwickelt werden, um Gerätebefehle und deren Antworten zu spezifizieren und das in einer menschenlesbaren Form. Dieser Mechanismus muss kompatibel

mit den Gerätekategorien sein und er muss einfach und ohne Programmierkenntnisse zu realisieren sein. Trotzdem muss er syntaktisch umfangreich und flexibel sein, um die verschiedenen Syntaxen und Strukturen von jedweden Gerätebefehlen und -antworten abzubilden.
Das Gesamtkonzept wird als *Excalibur Methode* und im Folgenden kurz als „Methode“ bezeichnet. Sie kapselt alle Konzepte, angefangen bei Gerätekategorien, der Ablaufsequenz und der Datenformatierung bzw. -ausgabe. Da zum Entwurf einer Methode keine Zeile Code geschrieben werden soll, ist das Resultat eine Verlagerung von einer Software-Implementierung zu einer Software-Konfiguration, also aus Sicht der Zielgruppen eine Verlagerung vom Systemprogrammierer zum Systemintegrator.

Übertragbarkeit von Konfigurationen

Für die Entwicklung einer Steuerungslösung eines analytischen Aufbaus sollen im Idealfall ausschließlich Konfigurationsdaten genügen. Wie gezeigt worden ist, sind die Konfigurationen nicht mehr abhängig von den darunter liegenden Geräten, sondern nur von Gerätekategorien. Alle Geräte können jederzeit durch äquivalente Geräte derselben Kategorie ersetzt werden (z.B. bei Neukauf von Geräten). Das bedeutet, dass eine Methode durch einen beliebigen analytischen Aufbau realisiert werden kann, solange die verwendeten Gerätekategorien alle benötigten Geräte abdecken. Ausgehend davon kann eine Methode auf jedes beliebige System übertragen werden, solange letztere Bedingung erfüllt ist.
Zusätzlich kann eine Methode entworfen und damit auch eine Steuersoftware entwickelt werden, *ohne* die Geräte bereits zu kennen, indem Kategorien mit benötigten Operationen definiert und erst anschließend die Gerätebeschreibungsdateien (s. Treiber in Abschnitt 3.3.3) und Schnittstellendefinitionen (s. Abschnitt 3.3.4) angepasst werden. Das System, an welches Geräte angeschlossen sind, kann demnach eine Methode ausführen, indem es nach der Anweisung in der Methode mit den Geräten kommuniziert. Das bedeutet aber auch, dass ein weiteres Konzept entwickelt werden kann, welche ausschließlich auf Methodenbasis operiert, aber die zugrunde liegenden Geräte nicht kennen muss. Das Resultat ist eine Separierung der Steuerungssoftware in zwei Teile: Einer Hardwaresteuerungskomponente (Server) und einer Konfigurationskomponente (Client), wobei beide über die „Methoden-Schnittstelle“ interagieren. Für die Übertragbarkeit von Konfigurationen bedeutet dies, dass, sobald einer Hardwaresteuerkomponente eine Methode bekannt gemacht wird, er sie an jeden mit dem Server kompatiblen Client ausliefern und

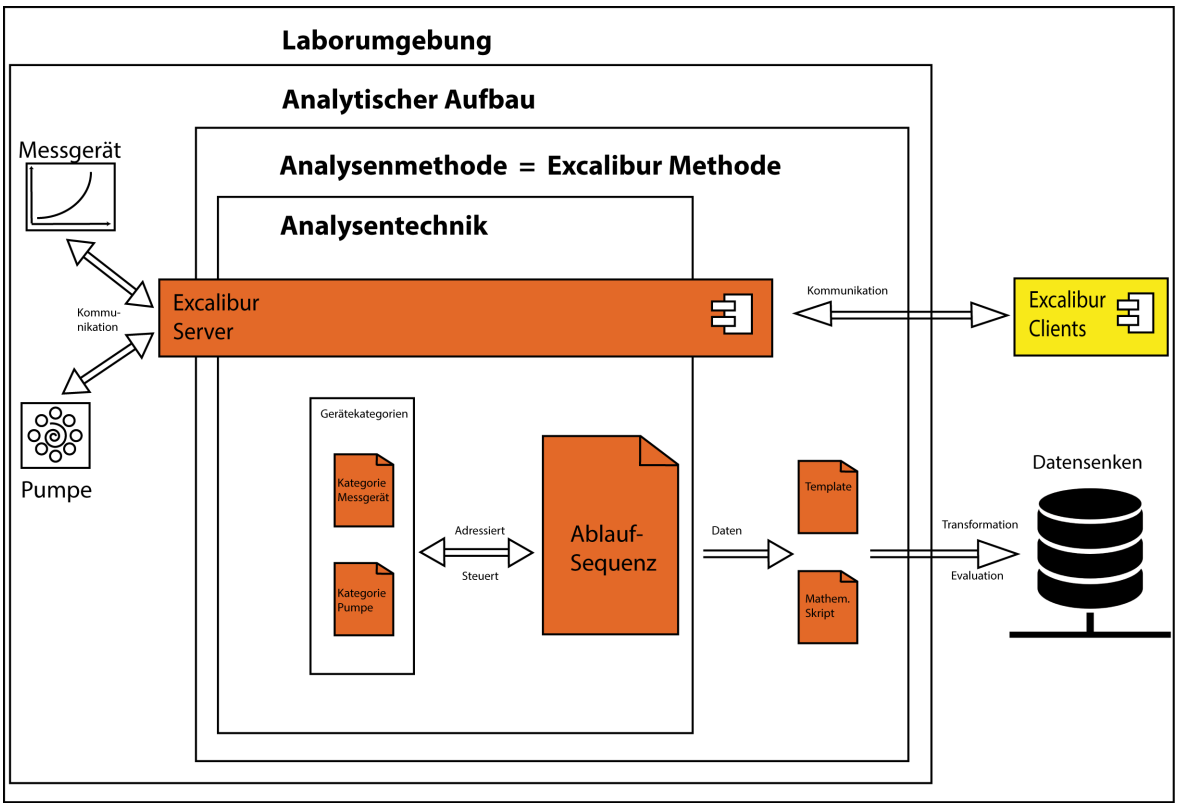


Abbildung (2.11)

Zusammenhang zwischen analytischer Methode und Excalibur-Methode. Alle orangen Symbole stellen Konfigurationsdateien dar und sind Teil des Messrechners des analytischen Aufbaus (mitunter der Excalibur Server). Mit ihnen sollen „reale“ analytische Zusammenhänge beschrieben werden und daraus ein informatisch gestütztes Modell entwickelt werden. Jedes Element der Analysenmethode kann ausgetauscht werden, ebenso wie jedes Element außerhalb, ohne eine Änderungen an einem anderen Element durchführen zu müssen. Elemente innerhalb der Analysentechnik sind aufeinander abgestimmt und weisen Abhängigkeiten auf; sie sind aber vollständig unabhängig von der Laborumgebung und dem analytischen Aufbau. Die Kommunikation mit dem Excalibur Server erfolgt über die Clients. Erklärungen für die einzelnen Elemente, siehe Text.

dieser den analytischen Aufbau und die zugehörige Analysenmethode ausführen kann.

Ergebnis

Aus obigem Ansatz resultieren folgende Konsequenzen:

- Die Hardwaresteuerungskomponente (Server) liefert die Methode an den Client aus, welcher nur die Methodendarstellung implementieren muss
- Der Steuerrechner des analytischen Aufbaus benötigt keine grafische Repräsentation, ausschließlich Konfigurationsdaten, welche von einem Client zur Verfügung gestellt werden
- Die Konfigurationskomponente (Client) kann vollständig ersetzt werden, solange sie der Methodenspezifikation genügt
- Clients können automatisch jede Methode administrieren, da sie nicht auf Geräte-, sondern nur auf Methodenbasis arbeiten, wodurch es möglich wird, nur *eine einzige* Client-Komponente für jeden analytischen Aufbau zu verwenden (Standard Software)
- Gibt es verschiedene Methoden auf verschiedenen Systemen in einem Netzwerk, kommt es zu einem Multi-Server-Multi-Client Netzwerk, da jede Konfigurations- mit jeder Hardwaresteuerungskomponente kommunizieren kann

Die Hardwaresteuerkomponente wird im Folgenden als **Excalibur Server** bezeichnet werden. Ihm ist das gesamte Abschnitt 3 gewidmet, in welchem sein Konzept und seine Funktionalität entwickelt wird.

2.3. Anwendungsbeispiele des Excalibur Servers

Anhand der drei Anwendungsbeispiele aus der Problemanalyse soll die Funktionsfähigkeit des methodenbasierten Ansatzes demonstriert werden. Der Ansatz ist innerhalb einer vollständig implementierten Software-Plattform realisiert, auf welche im nächsten Kapitel aus Sicht der Informatik eingegangen wird. Die Plattform, welche den methodenbasierten Ansatz implementiert, wird im Folgenden unter dem Arbeitstitel Excalibur Server oder einfach Excalibur geführt werden.
Im ersten Fall wird der Excalibur Server dafür verwendet, um ein Fluoreszenz-Spektrometer zu steuern und Messwerte aufzunehmen. Dieses Beispiel soll die Anwendbarkeit des Konzepts für eine einfache Gerätesteuerung demonstrieren, an welcher nur ein einziges Gerät beteiligt ist. Im zweiten Fall wird der Excalibur Server dafür verwendet, eine Steuerungssoftware für die reflektometrische Interferenzspektroskopie zu entwickeln. Dieses Beispiel wird herangezogen, um den Einsatz für mehrere interagierende Geräte zu demonstrieren, die in einer Ablaufsequenz gesteuert werden müssen. Im letzten Fall soll für den komplexen Aufbau einer Kombination aus 1λ-Reflektometrie und elektrochemischer Detektion im Rahmen des EU-geförderten Projekts INSTANT eine Steuerungssoftware entwickelt werden. In diesem letzten Beispiel soll nicht nur die Ablaufsequenz modelliert, sondern zusätzlich (teilweise externe) Plugins geladen und verwendet werden, um daraus ein übergreifendes Gesamtkonzept zu entwickeln.
Anhand der besprochenen Beispiele sollen am Ende des Kapitels noch einmal auf die Zielgruppen eingegangen und deren Rolle innerhalb der Anwendung besprochen werden.

2.3.1. Einsatz in der Fluoreszenz-Spektroskopie

Auf die Funktionsweise eines Fluoreszenz-Spektrometers ist bereits in Abschnitt 2.1.2 eingegangen worden. Für eine mögliche Steuerungssoftware wird neben dem eigentlichen Fluoreszenz-Spektrometer ein Computersystem für die Installation und Datenauswertung benötigt. Auf letzterem wird die Excalibur-Software als Windows®-Dienst installiert. Das Gerät selbst ist das LS50B Fluoreszenz-Spektrometer der Firma Perkin Elmer; das Handbuch mit den Gerätebefehlen liegt vor. Der wichtigste für das Spektrometer benötigte Deskriptor ist der Gerätetreiber, der mit Hilfe des Handbuchs entwickelt werden kann. Er ist in der Auszeichnungssprache XML geschrieben und entspricht deren typischer Syntax[BPSM$^+$06]:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<device xmlns="http://www.excalibur.net/driver">
 <deviceName>Perkin Elmer LS-50B</deviceName>
 <description lang="en">Beschreibung ausgelassen</
      description>
 <supplier>Perkin Elmer</supplier>
 <category>spectrometer</category>
 <command id="measure">
  <description lang="en">Starts a scan</description>
  <token token="$SC [0]">
   <value default="1" format="(1|2)" id="0">
    <constant name="ImmediateStart">1</constant>
    <constant name="AfterStartEvent">2</constant>
   </value>
  </token>
  <response terminator="" token="0000\r\n\x7f50
       .[0],[1][2][3][4],[5],[6],[7].
       [8],[9],[10],[11],[12]\r
   \n0000\r\n[13][14]" timeout="10000">
   <value channel="revision" format="%s" id="0" iterations=
        "1" length="2"/>
   <value channel="monochromator" format="(0|1|2)" id="1"
        iterations="1"/>
   <value channel="monochromator_qualification" format=
        "(0|1)" id="2" iterations="1"/>
   <value channel="luminescence_type" format="(0|1|2)" id=
        "3" iterations="1"/>
   <value channel="filter" format="(0|1)" id="4" iterations=
        "1"/>
   <value channel="excitation_wavelength" format="%5.1f" id=
        "5" iterations="1" type="double"/>
   <value channel="emission_wavelength" format="%5.1f" id=
        "6" iterations="1" type="double"/>
   <value channel="speed" format="%d" id="7" iterations="1"
```

```
        type="int32"/>
25    <value channel="filter_response" format="%d" id="8"
        iterations="1" type="int32"/>
26    <value channel="excitation_slit" format="%4.1f" id="9"
        iterations="1" type="double"/>
27    <value channel="emission_slit" format="%4.1f" id="10"
        iterations="1" type="double"/>
28    <value channel="data_count" format="%3d" id="11"
        iterations="1" type="uint32"/>
29    <value channel="stepsize" format="%3.1f" id="12"
        iterations="1" type="double"/>
30    <value channel="spectrum" format="(?s).*?(?=-9999)" id=
        "13" iterations="1" options="regexp('\d+')" type=
          "uint32list"/>
31    <value channel="null" format="(?s).*0000\x0d\x0a" id="14"
         iterations="1"/>
32   </response>
33   <postReadTime value="100"/>
34  </command>
35  <command id="setAbcissaHighLimit">
36   <description lang="en">Beschreibung ausgelassen</
       description>
37   <resource name="abcissaHighLimit" operation="SET"/>
38   <token token="$AH [0]">
39    <value format="%5.1f" id="0" max="900.0" min="200.0"
        type="double">
40     <constant name="ExcitationPowerOnDefault">800.0</
         constant>
41     <constant name="EmissionPowerOnDefault">900.0</constant>
42    </value>
43   </token>
44   <response token="0000"/>
45   <postReadTime value="100"/>
46  </command>
47  <command id="getAbcissaHighLimit">
```

```
48   <description lang="en">Beschreibung ausgelassen</
         description>
49   <resource name="abcissaHighLimit" operation="GET"/>
50   <token token="$AH"/>
51   <response token="0000\r\n\x7f[0]\r\n0000">
52    <value format="%3.1f" id="0" iterations="1"
          type="double"/>
53   </response>
54   <postReadTime value="100"/>
55  </command>
56
57  ...
58 </device>
```

Dies stellt nur einen Auszug aus dem Treiber für das Fluoreszenzspektrometer dar. Der vollständige Treiber kann unter dem Namen „ls50b.xml“ auf der beiliegenden DVD gefunden werden.
Die Syntax des Treibers wird genauer in Abschnitt 3.3 erklärt werden. Auch wenn der Inhalt des Treibers kompliziert erscheint, ist er logisch strukturiert, flexibel und enthält alle benötigten Gerätebefehle und deren menschenlesbare Übersetzung. Er beginnt mit der Definition eines Gerätenamens, einer Herstellerfirma und einer oder mehrerer Beschreibungen in verschiedenen Sprachen. Die Eigenschaft „category“ legt die Gerätekategorie fest (hier: *spectrometer*) und bestimmt, dass die Kommandos „measure“, „setIntegrationTime“ und „ getIntegrationTime“ vorhanden sein müssen. Betrachtet man die Zeile 9 des Auszugs, so kann der Measure-Befehl gefunden werden, welcher das Spektrometer veranlasst, ein Fluoreszenzspektrum aufzunehmen. Dazu muss vom Benutzer lediglich der Befehl „measure“ aufgerufen werden. Anhand des Treibers kann Excalibur dieses Schlüsselwort in den Gerätebefehl „SC 1“ übersetzen, diese Sequenz an das Gerät senden, die Antwort abwarten und mit den ab Zeile 17 folgenden Informationen die vom Spektrometer zurückgegeben Fluoreszenzdaten interpretieren. Sie werden auf verschiedene Kanäle gelegt (Eigenschaft: „channel“), auf welche in Excalibur wieder Bezug genommen werden kann, z.B. Abspeichern oder Weiterverarbeitung (Mittelwertbildung, Datenextraktion, Konvertierung, usw.) eines Kanals. Auf jeden dieser Kanäle kann nun z.B. innerhalb der Ablaufsequenz und anderen Stellen innerhalb der Excalibur-Plattform zugegriffen werden. Zur Veranschaulichung der Funktionsweise sind zwei weitere Befehle in obigem

Auszug aufgelistet, welche in diesem Fall den oberen Grenzwert für die Abszisse setzen und lesen können. Mit letzteren wird der Maximalwert der Wellenlänge für den Monochromator gesetzt, bis zu welchem eine Spektrometer-Messung erfolgen soll.
Man nehme nun an, der Excalibur Server ist darauf konfiguriert, Anfragen über eine Netzwerkschnittstelle entgegen zu nehmen. Das bedeutet, über einen bestimmten Netzwerkport (z.B. 8080) wird auf einen Verbindungsaufbau durch eine externe Software (z.B. einem Web-Browser) gewartet. Zusätzlich gehen wir davon aus, dass hinter der Netzwerkschnittstelle ein REST-basierter Dienst auf der Adresse „http://localhost:8080/“ lauscht (siehe Glossar und Abschnitt 3.2.3). Um eine Aktion auf dem Fluoreszenzspektrometer auszuführen, wird lediglich der Ressourcenname, Operationsname und eventuelle Parameter benötigt. Mit diesen Informationen kann beispielsweise durch den Aufruf der URI „http://localhost:8080/ls50b/measure“ eine Messung ausgeführt werden[3]. Soll mit Ressourcen des Spektrometers interagiert wie z.B. der obere Grenzwert für die Abszisse gelesen werden, kann die URI „http://localhost:8080/ls50b/abcissaHighLimit“ aufgerufen werden. Die URI „http://localhost:8080/ls50b/abcissaHighLimit/800.0“ kann aufgerufen werden, um ihn auf 800.0 nm zu setzen. Selbstverständlich können die URIs auch von jedem anderen Computer im Netzwerk aufgerufen werden, wobei ausschließlich „localhost“ durch die IP-Adresse des Excalibur Servers ersetzt werden muss.
Man erkennt bereits jetzt, dass allein mit dem Treiber theoretisch eine Aktion auf dem Fluoreszenzspektrometer adressiert und eine Ressource angesprochen werden könnte, ohne dafür zusätzliche Software installieren zu müssen; ein einfacher Web-Browser ist ausreichend. Diese Befehle würden in dieser Form zwar auf dem Excalibur Server eintreffen und in die richtigen Gerätekommandos übersetzt werden, könnten aber ohne weitere Deskriptoren keine Aktionen auf dem „realen“ Spektrometer ausführen. Um dies zu bewerkstelligen, muss dem Excalibur Server mitgeteilt werden, über welche Schnittstelle die Kommunikation erfolgen soll und mit welchem Gerät die Ressource „ls50b“ (siehe URI oben) verknüpft ist.

Die Kommunikation mit dem vorhandenen Perkin Elmer Fluoreszenzspektrometer erfolgt über die RS232-Schnittstelle, wofür ein Deskriptor benötigt wird, der die Schnittstellenparameter definiert und diese einer Ressource zuweist:

[3]Da REST mit Ressourcen und nicht über Operationen arbeitet, auf dem Spektrometer aber eine Operation durchgeführt werden soll, wird in Excalibur die sog. „state“-Ressource definiert und auf „measure“ gesetzt. Eigentlich wird aber die Operation „measure“ ausgeführt. Der Prozess ist sehr vereinfacht dargestellt. Auf sicherheitsrelevante Parameter und korrekt gesetzte HTTP Header Daten wird hier verzichtet

```
1 <RS232 deviceRef="ls50b" driver="ls50b.xml"
2  xmlns="http://www.excalibur.net/interfaces">
3  <port>COM1</port>
4  <baudrate>9600</baudrate>
5  <databits>8</databits>
6  <stopbits>1</stopbits>
7  <parity>NONE</parity>
8 </RS232>
```

Zeile 1 des Listings zeigt bereits wie die Ressource mit einem Treiber verknüpft wird, die restlichen Zeilen setzen die Parameter, welche das Betriebssystem für die Kommunikation mit dem Gerät benötigt (s. Abschnitt 3.1.2). Beide Deskriptoren, Treiber und Schnittstellendefinition, sind nun ausreichend, um über obige Aufrufe im Web-Browser „reale“ Aktionen auf dem Spektrometer durchführen zu können und die Messdaten in einem Array aus Integerwerten im Kanal „spectrum“ (s. Measure-Befehl im Treiber, Zeile 30) zu hinterlegen. Sollen die Daten in einer Datei gespeichert werden, kann in einem weiteren Deskriptor ein FileWriter definiert werden (s. Ein-/Ausgabe in Abschnitt 3.3.5):

```
1 <fileWriter id="FileWriter"
2    xmlns="http://www.excalibur.net/system/io">
3  <channel>spectrum</channel>
4  <basePath>https://webdav.server.de/data/directory/
       fluorescence</basePath>
5 </fileWriter>
```

Auf diese Weise werden die Daten des Kanals „spectrum“ (s. oben) auf dem WebDAV-Server unter der angegebenen Adresse im angegebenen Verzeichnis in einer Datei mit dem Namen „data.xml“ in einem standardisierten XML-Format gespeichert (Die Werte *integer0;integer1;...;integerN* stellen die Intensitätswerte dar):

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <dataset dataType="value" dataFormat="%d" writer=
3    "FileWriter" xmlns="http://www.excalibur.net/core">
4  <set>
```

```
  <data channel="spectrometer" type="uintlist">integer0;
     integer1;integer2;...;integerN</data>
 </set>
</dataset>
```

Sollen die Daten formatiert gespeichert werden, muss ein Template verwendet werden. Dazu wird das obere Listing für den FileWriter folgendermaßen angepasst (s. Template Mechanismus in Abschnitt 3.3.5 und 3.3.7):

```
<fileWriter id="FileWriter"
   xmlns="http://www.excalibur.net/system/io">
 <channel>spectrum</channel>
 <serializerRef>TemplateSerializer</serializerRef>
 <basePath>https://webdav.server.de/data/directory/
      fluorescence</basePath>
</fileWriter>

<templateSerializer id="TemplateSerializer"
   xmlns="http://www.excalibur.net/system/io">
 <template xmlns="http://www.excalibur.net/template">
  <head>
   <style>TEXT</style>
  </head>
  <body basePath="${date}" id="fluorescence_data">
       Fluorescence Fluorescence Measurement<br/><br/>
Date: <datetime id="date"/><br/>
User: <user default="anonymous" id="user"/><br/>
Method: <method default="Fluorescence" id="method"/><br/>
Comment: <comment id="comment"/><br/><br/>
   <table id="spectra">
    <toc>Dateiliste</toc>
    <column id="date">
     <head>Datum</head>
     <time/>
    </column>
    <column channel="spectrum" id="Spektrum%d" interleaved=
```

```
26          "true" type="uint32list" unit="au">
27        <head><title/> [<unit/>]</head>
28        <dataRow/>
29      </column>
30    </table>
31   </body>
32  </template>
33 </templateSerializer>
```

Damit werden die Daten des Spektrometers auf dem WebDAV-Server unter der angegebenen Adresse im Verzeichnis (exemplarisch) „/data/directory/fluorescence/2016-03-15/fluorescence_data0.dat“ gespeichert. Die Datei würde dann folgendermaßen aussehen (Alle Daten sind Beispielwerte), was im *body*-Abschnitt des Templates definiert ist:

```
Fluorescence Measurement

Date: 2016-03-15
User: messknecht
Method: Fluorescence
Comment: Testmessung mit dem Fluoreszenz-Spektrometer

Dateiliste
Datum Spektrum0 [au]
18:00:00.000 125000;125001;125002;125003;125004;...
```

Für eine genaue Definition der einzelnen Schlüsselworte von Templates und deren Funktionsweise soll auf die Dokumentation verwiesen werden.

Mit drei Deskriptoren (Schnittstelle, Treiber, Writer) kann das vorhandene Fluoreszenzspektrometer angesprochen, Messungen aufgenommen und die erhaltenen Daten formatiert auf einen WebDAV Server gespeichert werden, ohne dafür eine Zeile Quelltext zu schreiben. Der ganze Vorgang kann über einen einfachen Web-Browser initiiert werden.

Zwar ist die Steuerung über den Web-Browser eine einfache Möglichkeit, mit Geräten zu interagieren. Ein Benutzer möchte aber im Normalfall auch eine grafische Benutzeroberfläche zur Ausgabe der erzeugten und vom Gerät zurückgegebenen Spektrometer-Daten. Da der Web-Browser nicht wissen kann, wie die erhaltenen Daten zu interpretieren und

darzustellen sind, müssen dafür geeignete grafische Oberflächen eingesetzt werden. Diese sind in Form von Plugins entwickelt worden (s. Abbildung 2.12 und 2.13). Wie das Wort Plugin bereits vermuten lässt, wird zum Laden und Darstellen ein Plugin-Container benötigt. Er stellt dem Benutzer eine Framework zur Verfügung, mit welchem Funktionalitäten zur Gerätesteuerung entwickelt, geladen und eingesetzt werden können. Die Funktionalitäten selbst können von einem Programmierer auf einfache Weise erweitert werden, indem er beispielsweise ein Toolkit verwendet, welches dafür entwickelt worden ist. Damit lassen sich Excalibur-Befehle auf einfache Weise erzeugen und an den Server senden. Zusätzlich sind darin Parsing-Methoden enthalten, welche auch die vom Excalibur Server zurückgesendeten Daten wieder interpretieren können. Da die Beschreibung aller Frameworks, Client-Anwendungen und Hilfs-Applikationen den Umfang dieser Arbeit sprengen würden, sollen ausschließlich ausgewählte Abbildungen von diesen gezeigt werden (s. Abbildung 2.14 und 2.15). Für weitere Informationen soll auf deren Dokumentationen verwiesen werden.
Für eine Anwendung der genannten Deskriptoren ist keines dieser Zusatzpakete obligatorisch. Der Excalibur Server ist ohne grafische Oberfläche lauffähig. Die Deskriptoren können manuell über Standard (XML-)Editoren[4] erzeugt und über jede verfügbare Schnittstelle an ihn übergeben werden. Dass diese Vorgehensweise für den Benutzer nicht komfortabel ist, lässt sich nicht bestreiten, aber für die Erklärung der Ergebnisse und der Funktionsweise des Excalibur Servers soll diese Stellungnahme genügen.
Um eine reale Messung auf dem Spektrometer durchzuführen sind mehr Kommandos nötig, als nur einen Scan zu starten. Für eine zu vermessende Probe müssen dafür Betriebsparameter eingegeben werden können. Einige Betriebsparameter und deren Befehlsname sollen im Folgenden genannt werden (s. Tabelle 2.1). Sie werden für die anschließende Messung von Rhodamin 6G benötigt, einem weit verbreiteten Fluoreszenzfarbstoff. Diese und alle anderen Parameter inklusive Beschreibungen können im Gerätetreibern gefunden werden.

Soll eine Messung für ein Emissionsspektrum gestartet werden, so wird der Anregungs-Monochromator auf die Anregungswellenlänge von 480 nm gesetzt. Dafür wird über *SetMonochromatorSelection* der Anregungs-Monochromator gewählt, danach *SetAbscissaHighLimit* auf 480 nm gesetzt, ebenso wie *SetAbscissaLowLimit* auf denselben Wert gesetzt wird. Da das Emissionsspektrum aufgenommen werden soll, müssen für den Emissions-Monochromator die Anfangswellenlänge und die Endwellenlänge gewählt werden. Dies erfolgt wieder mit den Befehlen *SetMonochromatorSelection* (Emission), *Set-*

[4]Wie beispielsweise <oXygen/>, Eclipse, o.ä.

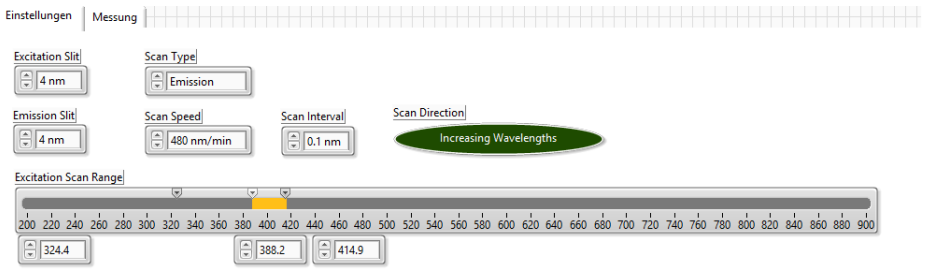


Abbildung (2.12)

Plugin zur Steuerung eines Fluoreszenz-Spektrometers, dieses im Speziellen für das LS50B von Perkin Elmer.

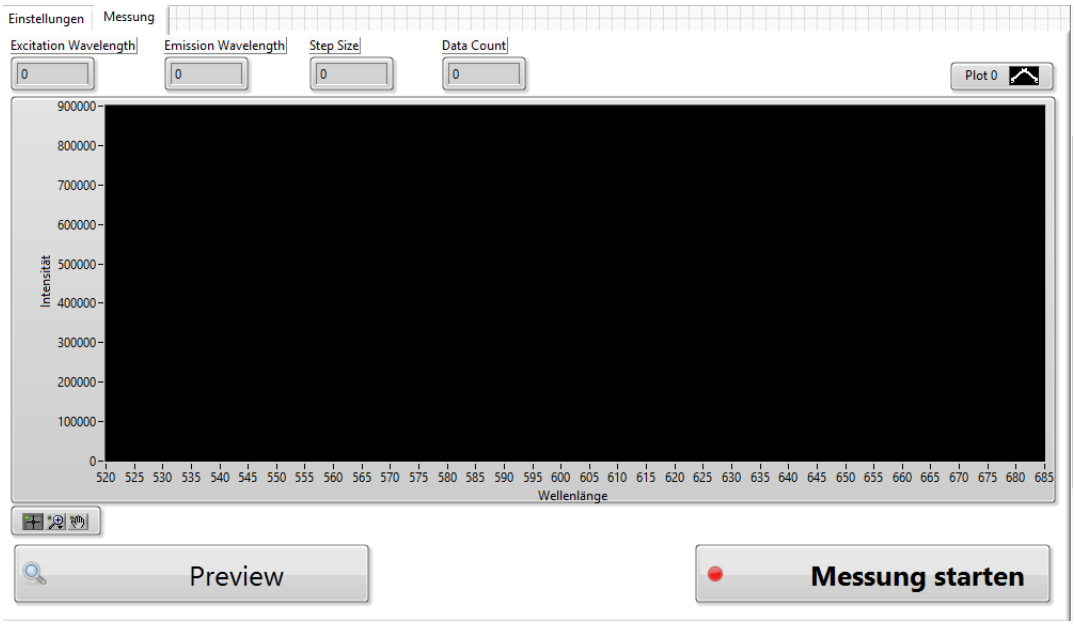


Abbildung (2.13)

Plot zur Anzeige der Messdaten, sobald der Befehl „Scan“ über die Schaltfläche „Messung starten“ an den Excalibur Server gesendet wird (unten).

AbscissaHighLimit (648 nm) und *SetAbscissaLowLimit* (510 nm). Anschließend werden die beiden Spalte über *SetEmissionSlit* und *SetExcitationSlit* auf 4 nm gesetzt. Über *SetScanInterval* wird das Datenausgabeintervall gewählt, nach welcher Anzahl an Nanometern jeweils ein Datenpunkt zurückgegeben wird (hier: 1 nm). Zuletzt wird über *SetScanSpeed* die Geschwindigkeit der Monochromatorbewegung festgelegt (hier: 60 nm/-min für den Emissions-Monochromator).

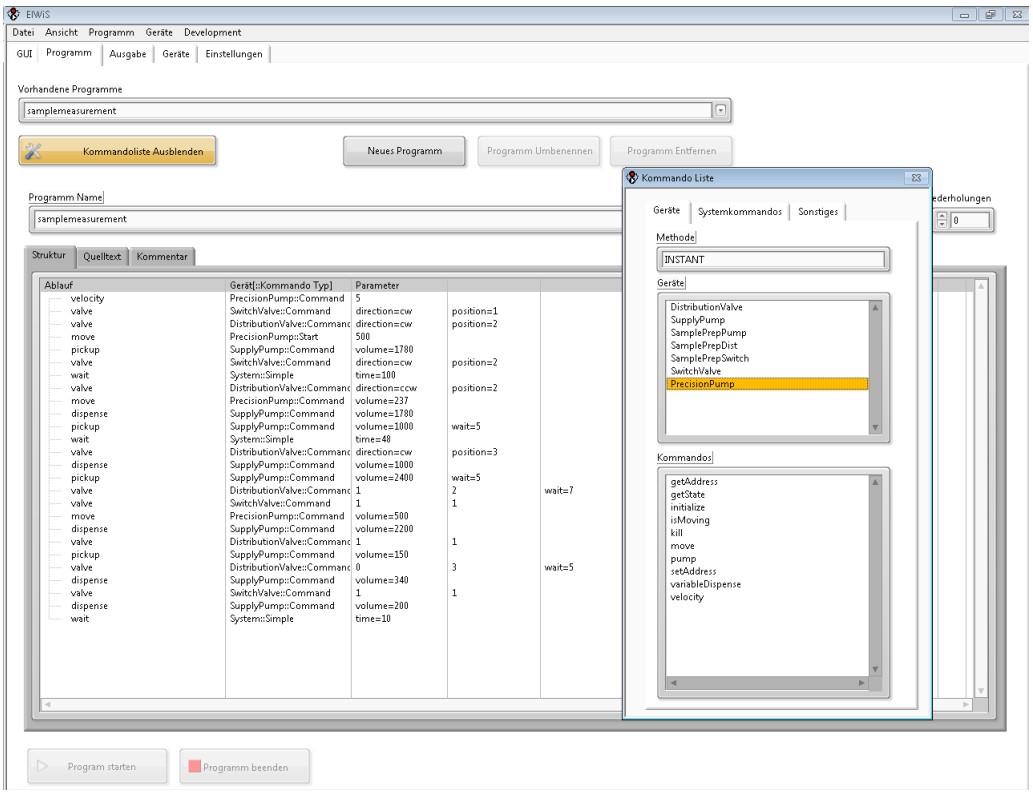


Abbildung (2.14)

Der Plugin-Container ElWiS, welcher verschiedene auf den Anwendungsfall zugeschnittene Plugins laden kann. Es existieren Methoden-, Ablaufsequenz-, Ausgabe- und Geräte-Plugins, hier dargestellt das Standard Ablaufsequenz-Plugin, welches geladen wird, wenn kein benutzerdefiniertes zur Verfügung steht.

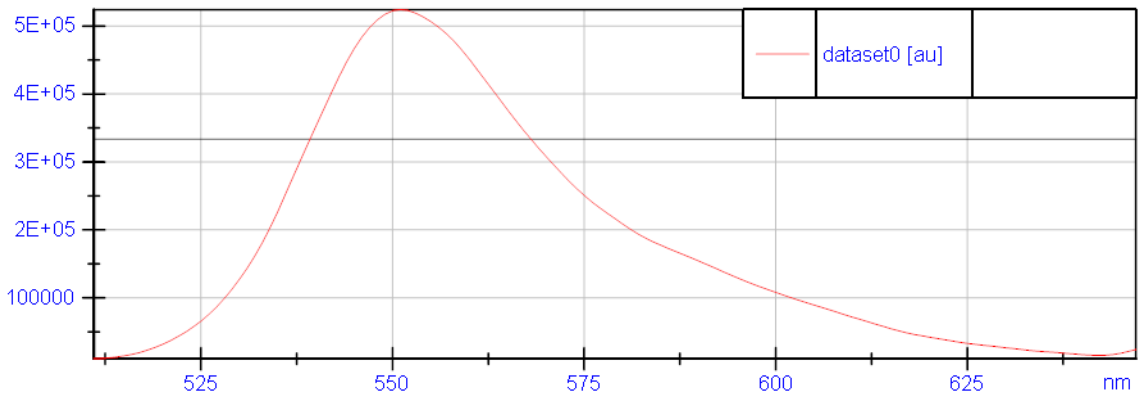
Tabelle (2.1)

Ausgewählte Kommandos des Fluoreszenzspektrometers, die für eine Messung benötigt werden.

Befehlsname	Bedeutung
SetAbscissaHighLimit	Setzt für den gewählten Monochromator ein neues oberes Emissions- bzw. Anregungslimit in einem Bereich zwischen 200.0 und 900.0 nm.

SetAbscissaLowLimit	Setzt für den gewählten Monochromator ein neues unteres Emissions- bzw. Anregungslimit in einem Bereich zwischen 200.0 und 900.0 nm.
SetEmissionSlit	Setzt den Spalt für den Emissions-Monochromator zwischen 2.5 und 15 nm.
SetExcitationSlit	Setzt den Spalt für den Anregungs-Monochromator zwischen 2.5 und 15 nm.
SetMonochromatorSelection	Wählt den Monochromator für den Scan aus mit 0 = Anregung, 1 = Emission.
SetScanInterval	Setzt das Datenausgabeintervall für den Scan zwischen 0.1 und 5 nm.
SetScanSpeed	Setzt die Geschwindigkeit eines Scans zwischen 10 und 1500 (Von niedrigen zu höheren Wellenlängen) bzw. -1500 und -10 nm/min (vice versa).
Scan	Führt einen Scan mit den getroffenen Einstellungen durch.

Für Rhodamin 6G konnte damit per Start des „Measure“-Vorgangs das folgende Spektrum aufgenommen werden:



Fazit

Die Ansteuerung von Einzelgeräten ist auf einfache Weise möglich, wozu der Excalibur Server natürlich in der Lage ist. Will man tatsächlich eine Steuerungssoftware für Einzelgeräte entwickeln, so liegt deren Fokus in einer optimalen Bedienbarkeit, d.h. in grafisch

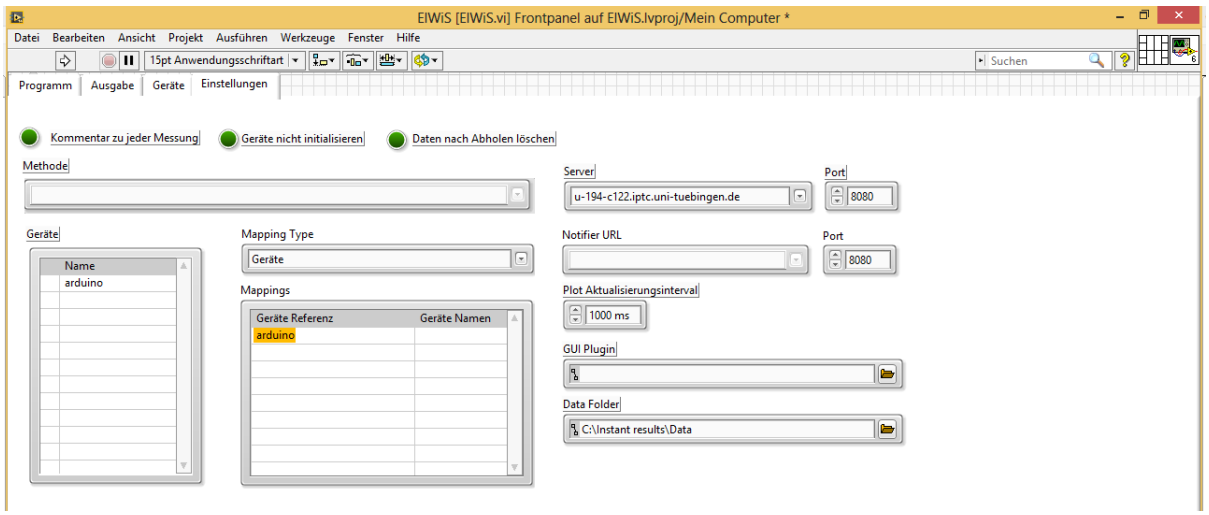


Abbildung (2.15)

Die Einstellungsmaske des ElWiS Plugin-Containers zum Festlegen der Methode, Geräte(-kategorien), der Mappings und zur Auswahl eines Excalibur-Servers. Die Konfiguration der Methode findet nicht hier, sondern in einem anderen Programm, dem Methoden-Konfigurator statt (s. Dokumentation).

ansprechenden und benutzerfreundlichen grafischen Oberflächen. Dies ist explizit nicht der Hauptfokus des Excalibur Servers, weshalb er auch ohne grafische Benutzeroberfläche gestaltet worden ist und nur über Befehle bestimmter Schnittstellen (Queues, Web-Services) gesteuert werden kann. Der Hauptnutzen liegt darin, Abläufe mehrerer Geräte einfacher entwickeln zu können und als Steuerungssoftware für integrierte Systeme zu fungieren.

2.3.2. Einsatz in der Reflektometrischen Interferenzspektroskopie (RIfS)

Bei der RIfS wird die Reflektion von Licht an dünnen Schichten beobachtet, welches auf die Rückseite eines transparenten, optischen Transducers gestrahlt wird. Auf diese Weise können Änderungen auf der Oberfläche des Transducers beobachtet werden, die durch Wechselwirkungen von spezifisch aufeinander abgestimmten, biologischen Systemen verursacht werden.

Für die Beobachtung des reflektierten Interferenzspektrums wird ein Diodenzeilenspektrometer benötigt und für den Transport der biologischen Liganden zum Transducer ein Probentransportsystem, welche alle von Excalibur angesteuert werden sollen. Letzteres besteht aus einem Autosampler, der es ermöglicht nacheinander mehrere Proben zu vermessen, zwei Laborpumpen zum Transport der in Pufferlösung befindlichen biologischen Substanzen durch die Schlauchsysteme und zwei Ventile zum Schalten auf verschiedene Lösungsmittel während eines Messvorgangs. Bei einer RIfS-Messung muss zu jedem Zeitpunkt bekannt sein, welcher Schritt des Messablaufs gerade abläuft. Dies hat den Grund, weil durch die permanente Beobachtung der Transduceroberfläche außer einer Konzentrationsbestimmung des Liganden auch eine zeitabhängige Bestimmung der Wechselwirkungsprozesse möglich ist. Die Geräte müssen demnach innerhalb eines Messablaufs sehr zeitgenau schalten.
Für die Messung der Interferenzspektren wird ein optimales Signal-Rausch-Verhältnis angestrebt, welches durch zwei Arten erreicht werden kann: Man erhöht die Integrationszeit der Detektion oder man bildet den Mittelwert aus vielen Spektren. Beispielsweise hat 1 Spektrum mit 100 μs Integrationszeit ein vergleichbares Signal-Rausch-Verhältnis wie gemittelte 10 Spektren mit jeweils 10 μs. Dennoch können Detektoren nicht unendlich lange Zeit Daten aufnehmen (Sättigung), weshalb zwar der dynamische Bereich eines Spektrometers ausgenutzt werden sollte, aber dennoch niemals die Sättigungsgrenze überschritten werden darf. Aus diesen Werten muss ein optimales Verhältnis aus Integrationszeit und Anzahl Spektren pro Zeiteinheit gefunden werden, was abhängig ist von der reflektierten Lichtintensität, Sättigungsgrenze des Detektors, der möglichen Integrationszeiten, der Prozessierungszeit für Gerätebefehle im Spektrometer und der Datenübertragungsgeschwindigkeit. Für reflektometrische Messungen ist ein Messpunkt jede 5 Sekunden ausreichend. Es haben zwei verschiedene Spektrometer zur Verfügung gestanden. Ein Spekol 1100 der Firma Analytik Jena und ein HR2000+ der Firma Ocean Optics. Die folgende Tabelle soll einen Vergleich der beiden Spektrometer zeigen:

Eigenschaft	Spekol	HR2000+
Dynamischer Bereich	10^5	$2x10^8$
Übertragungsgeschwindigkeit	19600 bps	480 Mbps
Integrationszeit	25 ms - 1000 ms	1 ms - 65 s
Pixelanzahl	256	2048

Das Spekol wird im Normalfall bei Integrationszeiten zwischen 100 und 250 ms betrie-

ben, wodurch jede 5 Sekunden maximal 15 Spektren möglich sind. Der Rest der Zeit wird für die Datenübertragung und die Verarbeitungszeit für die Steuerbefehle benötigt. Das HR2000+ hingegen kann bei einer Integrationszeit von 100 ms fast 25 Spektren jede 5 Sekunden aufnehmen, wodurch das Signal-Rausch-Verhältnis deutlich verbessert wird.

Für jedes unterschiedliche Gerät wird ein Excalibur-Treiber benötigt, d.h. ein Treiber für den Autosampler, einer für das Spektrometer, zwei Pumpentreiber und zwei Ventiltreiber. Wird zur Verdeutlichung der Spekol-Treiber herangezogen, so erkennt man wieder den Befehl „measure“. Dies liegt daran, dass auch das Spekol der Kategorie „spectrometer“ zugeordnet werden kann und somit die Befehle „measure“, „setIntegrationTime“ und „getIntegrationTime“ zur Verfügung stellen muss:

```
<device xmlns="http://www.excalibur.net/driver">
 <deviceName>SPEKOL</deviceName>
 <description lang="de">Beschreibung ausgelassen</
      description>
 <supplier>Analytik Jena</supplier>
 <category>spectrometer</category>
 <command id="preparemeasure">
  <description lang="de">Spektrometer in den Status "Messen"
       bringen</description>
  <visibility>private</visibility>
  <importProperty>integrationTime</importProperty>
  <token token="22"></token>
  <response token="Mxx"/>
  <read>discard</read>
  <postReadTime value="50" />
 </command>
 <command id="domeasure">
  <description lang="de">Messwerte vom Spektrometer
       empfangen</description>
  <visibility>private</visibility>
  <token token="06" encoding="HEX"></token>
  <response token="[0]">
   <value id="0" type="uint32" encoding="HEX" format="%x"
```

```
        length="$DATATYPE_FACTOR*($LAST_DIODE-$FIRST_DIODE
        +1)" options="cut(convert(and(byteorder(LL-LH-HL),
        uint8, 127),3,uint32),7,15)"/>
21   </response>
22  </command>
23  <compositeCommand id="measure">
24   <command ref="preparemeasure" read="discard" />
25   <command ref="domeasure" target="$1">
26    <redirect valueId="0" destination="spectrum"/>
27   </command>
28  </compositeCommand>
29
30  ...
31 </device>
```

Hier wird auch eine Besonderheit des Excalibur-Treibers deutlich gemacht. Im Gegensatz zum oben angeführten Fluoreszenzspektrometer der Firma PerkinElmer, benötigt man zum Erfassen eines Spektrums beim Spekol-Gerät insgesamt zwei Befehle, welche hier mit „preparemeasure“ und „domeasure“ benannt sind. Das letztendliche „measure“-Kommando vereinigt die beiden zu einer Komposition und gewährleistet, dass beide in der richtigen Reihenfolge und mit den korrekten Zeitabständen aufgerufen werden. Darüber hinaus sieht man in Zeile 18, dass der eigentliche Gerätebefehl im Gegensatz zum PerkinElmer Gerät keine menschenlesbare Zeichenkette („SC 1“, siehe oben), sondern eine nicht menschenlesbare Binärsequenz $\backslash x06$ (d.h. die Bitsequenz 0000 0110) sendet und ebenso die Daten in binärer Kodierung empfängt. Mit Hilfe der Anweisung in Zeile 20 wird dem Benutzer aber dieselbe Liste aus Spektren-Daten (Liste aus Fließkommazahlen) auf dem Kanal „spectrum“ (Zeile 27) wie auch beim PerkinElmer Gerät weitergegeben. Werden erneut die oben genannten Einstellungen für den REST-basierten Dienst angenommen, so kann auch hier beim Aufruf der URI „http://localhost:8080/spekol/measure“ eine Messung auf dem Spekol initiiert werden.

Wie man aber bereits jetzt erkennt, hat sich die URI leicht geändert, da die Bezeichnung *ls50b* durch *spekol* ersetzt werden muss. Hier wird nun ein Lösungsansatz angewendet, wie er in Abschnitt 1.5.1 bei der Beschreibung der Modularität angedeutet worden ist. Es wird nun ein sog. *Mapping* eingeführt und in einen Deskriptor geschrieben, welches

die eigentliche Spektrometer-Referenz auf einen anderen Namen abbildet, die aber stellvertretend für die Kategorie „spectrometer“ agiert:

```
<mapping xmlns="http://www.excalibur.net/mapping">
 <deviceMapping>
  <category>spectrometer</category>
  <deviceReference>spekol</deviceReference>
  <deviceName>spectrometer</deviceName>
 </deviceMapping>
</mapping>
```

Dies hat zur Folge, dass das Spekol im Anschluss auch unter der Ressource „spectrometer“ und der Mess-Befehl über die URI „http://localhost:8080/**spectrometer**/measure“ initiiert werden kann. Tauscht man das Mapping aus, indem einfach die „deviceReference“ durch „ls50b“ ersetzt wird, so kann dieselbe URI eingesetzt werden, um eine Messung auf dem Fluoreszenzspektrometer zu initiieren. Durch die Angabe der Kategorie ist dies möglich, da ein Befehl verwendet wird, den beide Geräte *zwingend* definieren müssen. Auf diese Weise ist es sehr einfach möglich, ein Gerät gegen ein anderes auszutauschen, ohne die Befehlssequenz (hier: URI) anpassen zu müssen.

Ähnliche Vorgänge können auch für die restlichen Geräte des RIfS-Laboraufbaus durchgeführt werden. Da sich die beiden Pumpen[5] und die Ventile [6], in einem kombinierten Laborsystem befinden, wird für dieses nur ein einziger Treiber benötigt. Allerdings wird dieses Gerät unter vier verschiedenen funktionalen Einheiten mit unterschiedlichen Namen (Mappings) eingebunden, d.h. es werden vier Mappings angelegt, welche zwei verschiedene Kategorien repräsentieren („pump“ und „valve“), vier verschiedene Namen als Bezeichnungen eingeführt („samplevalve“ => Sampleventil, „distributionvalve“ => Scherventil, „fixpump“ => Fixpumpe, „variopump“ => Variopumpe) und dieselbe Referenz als Verweis verwendet („asia“). Zur Erinnerung: Eine funktionale Einheit ist ein Gerät, welches einer Kategorie zugeordnet und unter einem Gerätenamen angesprochen werden kann. Alle Treiber können unter den Bezeichnungen „asiaas.xml“ (Autosampler), „asia.xml“ (Pumpen und Ventile) und spekol.xml bzw. hr2000.xml (Spektrometer)

[5] Eine Pumpe mit fester Geschwindigkeit [Fixpumpe] und eine mit variabler Geschwindigkeit (Variopumpe)
[6] Eines als Verteiler verwendet (Scherventil) und eines als Probenwahl (Sampleventil)

auf der beiliegenden DVD gefunden werden. Davon werden folgende Befehle für RIfS benötigt und werden auch in den Kategorien „autosampler“, „pump“, „valve“ und „spectrometer“ so definiert:

Kategorie	Befehl	Aktion
autosampler	step	Bewegt den Autosampler zur nächsten Probe
autosampler	buffer	Setzt als aktive Substanz den Puffer
autosampler	sample	Setzt als aktive Substanz die Probe
pump	pump	Aktiviert den Pumpe mit der übergebenen Geschwindigkeit
pump	stop	Stoppt den Pumpprozess
valve	valve	Setzt das aktive Ventil auf die übergebene Position
spectrometer	integrationTime	Setzt die Integrationszeit des Spektrometers auf den übergebenen Wert
spectrometer	measure	Misst ein Spektrum und gibt dieses zurück

Diese Gerätebefehle sind ausreichend, um Ablaufsteuerungen für alle Bio-Assays zu realisieren. Betrachtet man beispielsweise ein direktes Testformat, gilt es folgende Phasen nacheinander abzuarbeiten:
Aufnahme einer Basislinie (P1) ≫ Inkubationsphase (P2) ≫ Dissoziationsphase (P3) ≫ Regneration (P4) ≫ Rückkehr zur Basislinie (P5)
Während aller Phasen nimmt das Spektrometer alle fünf Sekunden ein Spektrum auf (welches aus mehreren Einzelspektren gemittelt wird) und berechnet nach der in Abschnitt 2.1.4 beschriebenen Theorie aus jedem das Minimum der ersten Ordnung. Mit Hilfe von Gleichung 2.19 kann daraus die optische Schichtdicke berechnet werden, welche gegen die Zeitpunkte aufgetragen ein Bindungssignal ergeben.
Jede Phase besteht prinzipiell aus folgenden Einzelschritten:
Schalten des Probenventils auf gewünschte Substanz ≫ Anschalten der Probenpumpe für Transport der Substanz zum Transducer ≫ Schalten des Probenventils auf den Puffer ≫ Transport des Puffers zum Transducer ≫ Abschalten der Probenpumpe
Zu Beginn und zum Ende jeder Phase befindet sich im Transducer der Puffer. Der Unterschied jeder Phase besteht in der *Substanz*, welche in P1 der Puffer selbst ist, in P2 der Analyt (Ligand), in P3 wieder der Puffer, in P4 das Regenerationsmittel (Detergenz) und in P5 wieder Puffer ist.

Ausführung einer Ablaufsequenz

Wie in der Problemanalyse besprochen, steht das Anwendungsszenario RIfS für den Fall, dass ein Gerät durch ein anderes ersetzt werden soll. Dieser Fall hat natürlich Folgen auf die gesamte Ablaufsteuerung, vor allem aber auf die Ablaufsequenz. Wie gezeigt worden ist, kann durch die Einführung von Mappings auf einfache Weise, ein Gerät derselben Kategorie durch ein anderes im Excalibur-Konzept ausgetauscht werden, ohne dabei die Ablaufsteuerung neu kompilieren zu müssen. Im Folgenden soll untersucht werden, wie sich ein derartiger Austausch auf die Ablaufsequenz auswirkt.
Dafür soll die bisherige Steuerungssoftware des RIfS-Laboraufbaus herangezogen werden. Folgend ist ein Ausschnitt aus der Ablaufsequenz der aktuellen Steuerungssoftware mit dem Namen „Measure“ gezeigt, wie sie momentan auf allen RIfS-Laboraufbauten eingesetzt wird:

```
SendPort1 Fixpumpe 1 an
ReceivePort1 Pos
SendPort2 Step
ReceivePort2 Stepped
SendPort1 SampleVentil 1 auf 3
ReceivePort1 Pos
SendPort2 Sample
ReceivePort2 MovedToSample
Enter 12
MultipleSample 5000
SendPort1 Fixpumpe 1 aus
ReceivePort1 Pos
Enter 8
MultipleSample 5000
SendPort1 inject1
ReceivePort1 Pos
SendPort1 Variopumpe 1 auf 40
ReceivePort1 Pos

...
```

In einigen wenigen Sätzen soll das Funktionsprinzip dieses Skripts erklärt werden. Der Befehl „SendPortX“ sendet den darauf folgenden Befehl an die serielle Schnittstelle am Port Nummer „X“, ReceivePortX empfängt einen Wert davon und vergleicht ihn mit der darauf folgenden Variablen. Sowohl Befehlssequenz wie auch die Variablen können innerhalb der „Measure“-Software konfiguriert werden. Auf Port 1 ist das ASIA System (kombiniertes System bestehend aus den beiden Pumpen und den beiden Ventilen, siehe oben) angeschlossen und unter Port 2 der Autosampler. Alle Befehle der oben definierten funktionalen Einheiten werden demnach über Port 1 an das Asia System gesendet (z.B. Fixpumpe, SampleVentil, usw.). Die einzelnen Befehle können aus den Begriffen im Listing abgeleitet werden oder auch in der „Measure“ Dokumentation nachgelesen werden. Einige Worte sollten dennoch den Zeilen beginnend mit „Enter Y“ und „MultipleSample Z“ gewidmet werden. Diese stehen für einen Messvorgang mit dem Spekol-Spektrometer, nämlich indem „Y“ Messungen mit einer Messzeit von „Z“ ms durchgeführt wird, wobei während dieser Zeit permanent Spektren aufgenommen werden (Anzahl abhängig von der Integrationszeit und der Zeit für die Datenübertragung) und diese über den angegebenen Zeitraum „Z“ gemittelt werden.

Es soll nun ein Gerät ausgetauscht werden, nämlich indem das ASIA System durch vier einzelne Geräte anderer Hersteller ersetzt wird (2 Pumpen und 2 Ventile). Zwar können die Befehlssequenzen (z.B. Fixpumpe, Step, usw.) frei gewählt und demnach durch *andere* Zeichenketten ersetzt werden, so müssen aber dennoch alle Ports des Befehls „SendPortX“ innerhalb der Ablaufsequenz auf die Zahlen (für „X“) 1 bis 4 angepasst werden. Problematischer gestaltet sich der Austausch des Spektrometers. Für dieses kann in der Ablaufsequenz nichts angepasst werden, weil es an dieser Stelle nicht adressiert wird. Dieses wird intern über ein Modul angesprochen, welches für 32-bit Windows® Systeme in kompilierter Form in die „Measure“ Software eingebettet ist. Zwar wird über den Befehl „MultipleSample Z“ das Modul herangezogen, um über „Z“ ms Daten empfangen und daraus den Mittelwert zu bilden, so kann dennoch das interne Modul nicht, wie im Fall des ASIA Systems, durch ein anderes ersetzt werden, weil die „Measure“ Software ein Laden von derartig komplexen Modulen nicht unterstützt. Wie bereits oben im Excalibur-Treiber für das Spekol gezeigt worden ist, handelt es sich nicht nur um einfache Zeichenketten (wie „Fixpumpe“ oder „Step“), die gesendet oder empfangen werden müssen, sondern um komplexe binäre Code-Sequenzen.

Im Folgenden soll dieselbe Ablaufsequenz konfiguriert in der Syntax des Excalibur Servers gezeigt werden. Dafür wird wieder angenommen, Excalibur lauscht per REST

auf dem üblichen Netzwerkport und die Sequenz wird im XML-Format an die URI „http://localhost:8080/program“ folgende Ablaufsequenz gesendet[7]:

```
<program name="Binding"
   xmlns="http://www.excalibur.net/program" >
 <start device="spectrometer" name="measure" interval=
      "0"/>
 <command device="fixpump" name="pump" value="on"/>
 <command device="autosampler" name="step"/>
 <command device="samplevalve" name="position" value="3"/>
 <command device="autosampler" name="sample"/>
 <command device="fixpump" name="pump" value="off"/>
 <command device="distributionvalve" name="position" value=
      "1"/>
 <command device="variopump" name="pump" value="40"/>
```

Diese Ablaufsequenz ist nicht nur klarer strukturiert, sondern versteckt auch alle Informationen bezüglich der Geräte-Anschlüsse. In diesem Fall ist sie in XML geschrieben und entspricht deren typischer Syntax[BPSM+06]. Mit dem Schlüsselwort „command“ wird ein einzelnes Gerätekommando eingeleitet, welches an die funktionale Einheit mit dem unter dem Attribut „device“ angegebenen Namen gesendet wird. Das Kommando selbst wird mit dem unter dem Attribute „name“ angegebenen Namen im Treiber gesucht und es werden die unter „value“ angegebenen Werte als Parameter mitgegeben. Ein weiterer Unterschied zur „Measure“-Sequenz von oben liegt im direkten Aufruft des Spektrometers. Mit dem Schlüsselwort „start“ wird ein Parallelprozess gestartet, der den „measure“-Befehl auf der funktionalen Einheit „spectrometer“ so oft wie möglich (Attribut „interval“ ist 0 [ms], abhängig von Integrationszeit) ausführt.
Davon ausgehend können nun auch komplexere Ablaufsequenzen verfasst werden, z.B. um ein direktes Testformat auf einfachste Weise abzubilden. Diese soll hier nur in abgekürzter, exemplarischer Form dargestellt werden. Die vollständige Sequenz befindet sich auf der beiliegenden DVD unter der Methode „RIfS“ > „programs“.

[7] Dies kann nicht einfach über die Adresszeile im Browser eingegeben werden, da noch zusätzliche Informationen (das Programm) mitgesendet werden müssen und diese nicht einfach in die URI geschrieben werden können. Es wird dafür eine Client Software verwendet wie z.B. die ElWiS Software (s. Abbildung 2.14)

```
<program name="Binding"
   xmlns="http://www.excalibur.net/program" >
 <var name="buffer" value="0"/>
 <var name="sample" value="1"/>
 <var name="regeneration" value="2"/>

 <command device="autosampler" name="step"/>

 <start device="spectrometer" name="measure" interval=
      "0"/>
 <call ref="baseline"/>
 <call ref="incubation"/>
 <call ref="dissociation"/>
 <call ref="regeneration"/>
 <call ref="baseline"/>
 <stop device="spectrometer" name="measure"/>

 <function name="baseline"/>
  <command device="samplevalve" name="position" value=
       "$buffer"/>
  <command device="samplepump" name="pump" value="50"/>
  <wait time="120"/>
  <command device="samplevalve" name="position" value=
       "$buffer"/>
  <wait time="120"/>
 </function>

 <function name="incubation"/>
  <command device="samplevalve" name="position" value=
       "$sample"/>
  ...

</program>
```

Mit Hilfe des Schlüsselworts „function“ können sich wiederholende Teile der Ablaufsequenz gruppiert und per „call“ Anweisung, unter Angabe von deren Namen als Referenz, ausgeführt werden.
Die einzelnen Messphasen unterscheiden sich (fast) ausschließlich in den Wartezeiten voneinander. Möchte man sich wiederholende Code-Segmente ersparen, kann man den Prozess auch ein wenig abstrakter gestalten und die Phasen äquivalent behandeln:

```
<program xmlns="http://www.excalibur.net/program" name=
  "Binding">
 <var name="buffer" value="0"/>
 <var name="sample" value="1"/>
 <var name="regeneration" value="2"/>

 <command device="autosampler" name="step"/>
 <start device="spectrometer" name="measure" interval=
    "0"/>
 <call ref="phase" value="$buffer,120"/>
 <call ref="phase" value="$sample,600"/>
 <call ref="phase" value="$buffer,120"/>
 <call ref="phase" value="$regeneration,300"/>
 <call ref="phase" value="$buffer,120"/>
 <stop device="spectrometer" name="measure"/>

 <function name="phase"/>
  <parameter name="valve_position"/>
  <parameter name="waiting_time"/>
  <command device="samplevalve" name="position" value=
     "$valve_position"/>
  <command device="samplepump" name="pump" value="50"/>
  <wait time="$waiting_time"/>
  <command device="samplevalve" name="position" value=
     "$buffer"/>
  <wait time="$waiting_time"/>
 </function>
</program>
```

Neben der Vorteilen der erweiterten Funktionalitäten ist die Ablaufsequenz vollständig unabhängig von den zugrunde liegenden Geräten. Zwar werden in den „command“ und „start“ Befehlen jeweils Gerätenamen innerhalb des *device*-Attributs benannt. Damit werden aber ausschließlich funktionale Einheiten referenziert. Wie oben erwähnt, befinden sich alle Ventile und alle Pumpen in demselben Laborsystem. In der Ablaufsequenz werden sie jedoch als eigene „Geräte“ mit eigenem Namen und eigenem Befehlssatz behandelt.
Würden alle Geräte des RIfS-Aufbaus durch andere Geräte ersetzt (neuer Autosampler, 2 reale Ventile und 2 alleinstehende Pumpen statt eines Laborsystems, HR2000+ statt Spekol Spektrometer), *so ändert sich an dieser Ablaufsequenz nichts*. Die zugrunde liegenden Treiber müssen zwar von denselben Kategorien abgeleitet sein; das Mapping kann aber auf dieselbe Gerätereferenz erfolgen. Die Bezeichnungen für die Kommandos sind durch die Kategorien festgelegt. Müssen die Parameter für die Kommandos in einer anderen Einheit angegeben werden (z.B. Pumpgeschwindigkeit), können sie mit Hilfe von Skripten direkt im Treiber normiert werden.

Als Beispiel soll nun das Spekol durch das HR2000+ Spektrometer ersetzt werden. Dazu soll ein Ausschnitt des Treibers des HR2000+ betrachtet werden:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <device xmlns="http://www.excalibur.net/driver">
3   <deviceName>HR2000</deviceName>
4   <description>Beschreibung ausgelassen</description>
5   <supplier>Ocean Optics</supplier>
6   <category>spectrometer</category>
7   <command id="domeasure">
8    <description lang="de">Messe ein Spektrum</description>
9    <importProperty>integrationTime</importProperty>
10   <token token="09" encoding="HEX"></token>
11   <response token="[0][1]" encoding="HEX">
12    <value format="%x" id="0" length="4096" type="uint16list"
           options="bitflip(byteswitch(),UWORD,13)"/>
13    <value format="%x" id="1" length="1" type="uint8" channel
          ="sync" />
14   </response>
```

```
15  </command>
16  <compositeCommand id="measure">
17   <command ref="domeasure" target="$1">
18    <redirect valueId="0" destination="spectrum"/>
19   </command>
20  </compositeCommand>
21 </device>
```

Auch das HR2000+ ist natürlich der Kategorie „spectrometer“ zugehörig und implementiert deren obligatorische Funktionen. Betrachtet man sich noch einmal die Treiber des Fluoreszenzspektrometers, des Spekols und des HR2000+, so erkennt man sehr deutlich die Unterschiede hinsichtlich der Gerätekommandos (z.B. „measure“), die tatsächlich an das Gerät gesendet werden und auch die Unterschiede in den Antworten, welche die Geräte zurückliefern. Trotz dieser Unterschiede würde sich die Ablaufsequenz für die RIfS Messung um kein Zeichen ändern. Das heißt im Umkehrschluss, dass alle Geräte eines RIfS-Laboraufbaus durch Geräte der gleichen Kategorie ersetzt werden können und alle Ablaufsequenzen würden dabei ohne Änderungen weiter verwendet werden können.

Datenausgabe in Excalibur

In gängigen Programmiersprachen wird die Datenausgabe innerhalb des Quelltextes zusammen mit der Ablaufsequenz definiert. In Excalibur ist sie vollkommen unabhängig davon. Sie wird innerhalb eines Deskriptors konfiguriert und agiert auf Basis von Kanälen, die entweder bereits im Treiber vorgegeben sind oder direkt in der Ablaufsequenz angegeben bzw. modifiziert werden können. Die Kanäle werden über die Geräteantworten mit Werten gefüllt und können anschließend innerhalb von mathematischen Formeln, Formatierungsanweisungen oder zum Routing verwendet werden. Auf diese Weise kann auch hier die Modularität gewährleistet werden, weil Kanäle einfach Daten weiterleiten unabhängig von den funktionalen Einheiten, welche die Daten erzeugen.
Die Datenausgabe soll die vom Spektrometer kommenden Daten auf verschiedene Arten berechnen, formatieren und weitergeben. Für RIfS erfolgt die Berechnung in zwei Stufen: der Mittelwertbildung der Spektren und die Berechnung des Minimums bzw. daraus die optische Schichtdicke. Für beide wird ein sog. Datentransformer verwendet, von welchen der erste die Mittelwertbildung über ein *Formula*-Script durchführt. Letztere Berechnung ist ungleich komplizierter und wird über ein *Octave*-Script durchgeführt. Auf die

genaue Funktionsweise von Transformern und Skripts werden Abschnitt 3.3.5 und 3.5.3 genau eingehen. Wie zu erwarten werden beiden in einen Deskriptor im XML-Format geschrieben. Sie sind im Folgenden exemplarisch für RIfS gezeigt:

Listing (2.1)

Scripts zur Berechnung der gemittelten RIfS-Spektren (oben) und der Minima der gemittelten Interferenzspektren

```
<dataTransformer xmlns="http://www.excalibur.net/system/io"
      id="Averaging">
 <conversionMethod channel="referencedData">
  <sequence xmlns="http://www.excalibur.net/program">
   <var key="#referencedData"/>
   <output key="#referencedData"/>
   <script>
    referencedData = temporal_average(#referencedData,4000)
   </script>
  </sequence>
 </conversionMethod>
</dataTransformer>

<dataTransformer id="MinimumCalc"
   xmlns="http://www.excalibur.net/system/io">
 <conversionMethod xmlns="http://www.excalibur.net/system/io
      ">
  <octave xmlns="http://www.excalibur.net/program">
   <var key="#referencedData"/>
   <var key="wavelength"/>
   <param key="order" type="double" value="1.5"/>
   <param key="minimumBorder_left" type="double"
      value="400"/>
   <param key="minimumBorder_right" type="double"
      value="600"/>
   <output key="#minimumData" type="double"/>
   <script>
    borderLeft = 25; borderRight = 75;
```

```
27      border = struct('left',{borderLeft},'right',{borderRight
            });
28      data = referencedData(border.left:border.right);
29      wavelengths = wavelength(border.left:border.right);
30      border = struct('right',{border.right - border.left},'
            left',{1});
31      min_index = find(data==min(data));
32      min_index = min_index(1);
33      border.right = border.right - min_index;
34      border.left = min_index - border.left;
35      range = [border.right,border.left];
36      dist = min(range) dist = struct('low',{min_index - dist
            }, 'high', {min_index + dist});
37      data = data(dist.low:dist.high);
38      wavelengths = wavelengths(dist.low:dist.high);
39      wavenumbers = 1 ./ wavelengths; q = polyfit(wavenumbers,
            data,4);
40      derivative_q = polyder(q);
41      root_q = roots(derivative_q)(2);
42      minimumData = order * (1/root_q) / 2;
43    </script>
44   </octave>
45  </conversionMethod>
46 </dataTransformer>
```

In den Scripts sieht man deutlich das Funktionsprinzip, wie Variablen über das „var“-Tag in das Script eingeführt werden und über das „output“-Tag wieder vom Skript abgeholt werden. Der Script-Abschnitt enthält die mathematische Ablaufsequenz in Excalibur- (oben) bzw. *GNU Octave*®-Syntax (unten). Variablennamen, die mit einem „#“ beginnen, referenzieren dabei die in den Kanälen transportierten Daten. Eine genauere Beschreibung von Datenein- und -ausgaben wird in Abschnitt 3.3.5 erfolgen.
Die Daten können jetzt, ebenso wie beim Fluoreszenzspektrometer, über Templates formatiert und beispielsweise mit einem FileWriter in Dateien gespeichert werden. Sie können aber auch mit Hilfe eines BufferWriters in Puffern gespeichert werden, um anschließend per Request von einem interessierten Client abgeholt zu werden:

```
1 <bufferWriter xmlns:io="http://www.excalibur.net/system/io"
      id="Referenced-Buffer">
2  <channel>referencedData</channel>
3  <buffer>referenced</buffer>
4 </bufferWriter>
5 <bufferWriter xmlns:io="http://www.excalibur.net/system/io"
      id="Binding-Buffer">
6  <channel>minimumData</channel>
7  <buffer>binding</buffer>
8 </bufferWriter>
```

Auf diese Weise werden die Daten der Kanäle „referencedData“ und „minimumData“ wie sie oben berechnet worden sind (s. Zeilen 7 und 42 in Listing 2.1), in einem Puffer gespeichert (s. Zeilen 5 und 24 in Listing 2.1). Ein Client (wie beispielsweise ein Web-Browser) kann nun die Puffer über ihre Namen abrufen und die Daten weiterverarbeiten oder darstellen. Würde in diesem Fall die URI „http://localhost:8080/buffer/referenced“ aufgerufen, so lieferte der Excalibur Server die referenzierten Spektrometerdaten in Form einer Liste im XML-Format zurück (wie oben beim Fluoreszenzspektrometer gezeigt).

Fazit

Es kann gezeigt werden, dass auch eine Anwendung mit mehrerer Geräten, die mit Hilfe einer Ablaufsequenz in der richtigen Reihenfolge angesteuert werden sollen, mit dem Excalibur Server entwickelt werden kann. Dafür musste bisher keine Zeile Quelltext verfasst werden. Zwar wird für die Übertragung der Ablaufsequenz an den Server ein REST-Client benötigt. Diese sind aber allgemein verfügbar[8] oder es wird der standardisierte ElWiS Plugin-Container (siehe oben) verwendet, wie er im Rahmen dieser Dissertation entwickelt worden ist.

[8] Beispielsweise das Advanced REST client Plugin für den Google Chrome Browser

2.3.3. Einsatz im EU-finanzierten INSTANT Projekt

Dieser Fall behandelt eine Anwendung für das EU-finanzierte Projekt INSTANT, in welchem ein Messsystem basierend auf einer Kombination aus 1λ-Reflektometrie mit elektrochemischer Detektion entwickelt werden soll. Die 1λ-RIfS basiert auf demselben Prinzip wie die spektrale RIfS, mit dem Unterschied dass nur mit einer Wellenlängen statt mit Weißlicht angeregt wird und die Auswertung über Intensität bei dieser Wellenlänge erfolgt, anstatt die spektrale Verschiebung eines Extremums im Interferenzspektrum zu berechnen (s. Abschnitt 2.1.5). Innerhalb des Projekts sollten Nanopartikel in Alltagsprodukten wie Sonnencremes, Orangensaft oder Wein mit Hilfe einer optischen Detektion in Kombination mit elektrochemischen Verfahren nachgewiesen werden. Für erstere wird die 1λ-RIfS gewählt, für letztere die Impedanzspektroskopie wie sie von einem Projektpartner integriert werden soll.

Neben der eigentlichen Detektion zeichnen sich die Anforderungen in diesem Anwendungsszenario durch eine aufwendige fluidische Probenvorbereitung aus. Da Nanopartikel in der Realität in komplexen Matrices wie Cremes, Säften und anderen komplexen Mischungen oder Suspensionen vorliegen, sind die eigentlichen Analyten nicht direkt zugänglich. Sie müssen zunächst mit komplexen Verfahren aus der Matrix extrahiert und in ein standardisiertes Medium überführt werden. Diese Überführung ist einer der wichtigsten Schritte, da nur über reproduzierbare Bedingungen (dasselbe Medium, pH-Wert, chemische Umgebung) die eingesetzten Rezeptoren auf der Transduceroberfläche (s. Abbildung 2.16), welche für den Einsatz im INSTANT Laboraufbau entwickelt worden sind, ein reproduzierbares Signal mit den sich anlagernden Nanopartikeln liefern. Darüber hinaus kann mit Hilfe der Probenvorbereitung eine Aufkonzentration der Nanopartikel erfolgen, wodurch die Nachweisgrenzen des Messsystems verbessert werden können.

Diese Probenvorbereitung ist von einem Kooperationspartner entwickelt worden und liegt in Form eines speziellen Filters mit Extraktionseinheit vor. Die Durchführung der Probenvorbereitung erfolgt erneut mit einem fluidischen Laborsystem bestehend aus mehreren Laborpumpen und -ventilen, die präzise von einer Steuerungssoftware angesteuert werden müssen. Dieser Vorgang stellt einen sehr kritischer Schritt dar, da durch die Präzision der Ansteuerung die Laufzeit und die transportierten Flüssigkeitsmengen

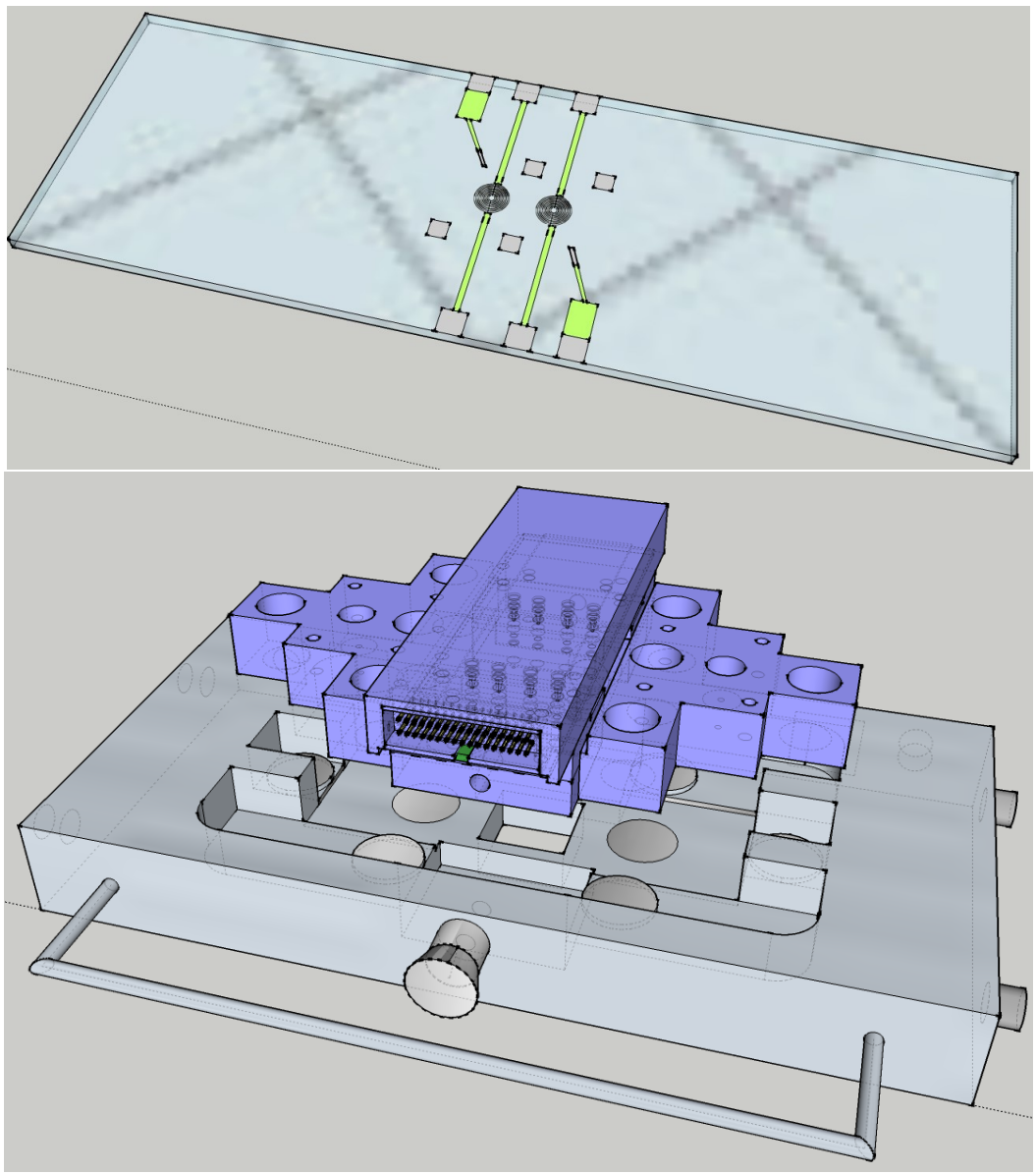


Abbildung (2.16)
INSTANT Transducer (oben) bestehend aus einem RIfS-Glassubstrat mit leitenden ITO Interdigitalstrukturen (grün bzw. grau dargestellt) für die elektrochemische Detektion. Der Transducer wird in die dafür vorgesehene INSTANT Flusszellenhalterung eingesetzt (unten, in grau dargestellt) und kann auf diese Weise mit der Probe überspült werden. Das Substrat ist transparent, kann also mit dem Kamerasystem von unten durch die Öffnung in der Flusszellenhalterung betrachtet werden. Der unten in blau dargestellte Block stellt das Verbindungsstück mit den ITO Elektroden auf dem Transducer dar, indem über die PINs auf der Vorderseite die Verbindung zu den Oszilloskopen hergestellt wird. Das Andocken an die Flusszellenhalterung erfolgt magnetisch.

bestimmt werden. Direkt damit gekoppelt sind die Verdünnungsgrade der aufkonzentrierten Probe und somit auch die Richtigkeit der Konzentrationsbestimmung durch die Detektion.
Durch die Komplexität der Probenvorbereitung und der Detektion sind am INSTANT Laboraufbau mehr Geräte beteiligt als im oben genannten RIfS Laboraufbau. Es müssen in das INSTANT-System insgesamt 1 Peristaltikpumpe (Ismatec RegloICC), 4 Ventile (Hamilton MVP), 1 Dispenserpumpe (VICI M6), 1 Spritzenpumpe (Hamilton PSD), 1 CMOS Kamera (PCO Edge) und 2 Oszilloskope (Tiepie HS5) integriert werden. Diese werden aufgeteilt auf Probenvorbereitung (Peristaltikpumpe, 2 Ventile), Probentransport (Dispenserpumpe, Spritzenpumpe, 2 Ventile) und die Detektion (Kamera, Oszilloskope). Mit Hilfe der Excalibur-Treiber, Kategorisierung und Plugins müssen folgende Geräte in einem Messablauf berücksichtigt werden: 3 *Pumpen*, 4 *Ventile*, 3 *Messgeräte*. Die entwickelten Treiber können auf der beiliegenden DVD gefunden werden. Die Phasen einer Probenmessung lassen sich folgendermaßen gliedern:
Initialisierung (P1) ≫ Probenelution (P2) ≫ Probenanreicherung (P3) ≫ Probeninjektion (P4) ≫ Messung (P5) ≫ Reinigung des Systems (P6)
Da die Messgeräte ausschließlich Bilder (Kamera) bzw. Strom-Spannungskurven kontinuierlich in gleichbleibenden Intervallen während der Phase P5 aufnehmen sollten und außerdem die elektrochemische Detektion durch den Projektpartner erfolgt ist, wird die Detektion selbst von externen Plugins übernommen. Der Excalibur Server steuert in diesem Fall ausschließlich den fluidischen Ablauf und triggert den Start- und den Stopvorgang für die externen Detektionsplugins.
Es gibt prinzipiell zwei Möglichkeiten, den fluidischen Ablauf in der Ablaufsequenz zu strukturieren. Der erste Fall ist angelehnt an die Strategie, wie sie für die reflektometrische Interferenzspektroskopie gewählt worden ist. Die einzelnen Phasen werden in Funktionen (Stichwort „function“) ausgelagert und über einen „call“ im Ablauf der Reihe nach aufgerufen:

```
<program name="INSTANT"
   xmlns="http://www.excalibur.net/program" >
 <call ref="initialization"/>
 <call ref="elution"/>
 <call ref="preconcentration"/>
 <call ref="injection"/>
 <call ref="measurement"/>
 <call ref="clean"/>
```

```
9
10  <function name="initialization"/>
11   <comment>Alle Ventile auf die Initialisierungsposition
          stellen und Pumpen ausschalten</comment>
12   <command device="samplevalve" name="position" value=
          "0"/>
13   <command device="samplepump" name="pump" value="0"/>
14   ...
15  </function>
16
17  <function name="elution"/>
18   <comment>Nanopartikel in Matrix im Solvent lösen und in
          den Filter aufziehen</comment>
19   ...
20
21  <function name="preconcentration"/>
22   <comment>Probe im Durchfluss im Filter anreichern und in
          Laufpuffer lösen</comment>
23   ...
24
25  <function name="injection"/>
26   <comment>Probe aus dem Filter spülen und in den
          Messkreislauf transportieren</comment>
27   ...
28
29  <function name="measurement"/>
30   <comment>Inkubationsphase: Probe mit Laufpuffer über den
          Transducer transportieren</comment>
31   ...
32
33  <function name="clean"/>
34   <comment>Alle Schläuche, Filter und Kanäle des Flussystems
           mit Laufpuffer spülen</comment>
35   ...
36 </program>
```

Die zweite Möglichkeit nimmt die Excalibur-Methode zur Hilfe und definiert die einzelnen Phasen als voneinander unabhängige Prozesse. Da die einzelnen Phasen sich nicht gegenseitig beeinflussen, d.h. keine Informationen zwischen ihnen ausgetauscht werden oder sie aufeinander warten müssen, ist es nicht zwingend notwendig, sie in derselben Ablaufsequenz aufzuführen. Aus diesem Grund kann die Ablaufsequenz folgendermaßen aufgeteilt werden:

```
1  <program name="initialization"
2     xmlns="http://www.excalibur.net/program"/>
3    <comment>Alle Ventile auf die Initialisierungsposition
         stellen und Pumpen ausschalten</comment>
4    <command device="samplevalve" name="position" value=
         "0"/>
5    <command device="samplepump" name="pump" value="0"/>
6    ...
7  </program>
8
9  <program name="elution"
10    xmlns="http://www.excalibur.net/program">
11   <comment>Nanopartikel in Matrix in Solvent lösen und in
         den Filter aufziehen</comment>
12   ...
13
14 <program name="preconcentration"
15    xmlns="http://www.excalibur.net/program"/>
16   <comment>Probe im Durchfluss im Filter anreichern und in
         Laufpuffer lösen</comment>
17   ...
18
19 <program name="injection"
20    xmlns="http://www.excalibur.net/program"/>
21   <comment>Probe aus dem Filter spülen und in den
         Messkreislauf transportieren</comment>
22   ...
23
```

```
24 <program name="INSTANT"
25    xmlns="http://www.excalibur.net/program"/>
26   <comment>Inkubationsphase: Probe mit Laufpuffer über den
         Transducer transportieren</comment>
27   ...
28
29 <program name="clean"
30    xmlns="http://www.excalibur.net/program"/>
31   <comment>Alle Schläuche, Filter und Kanäle des Flussystems
         mit Laufpuffer spülen</comment>
32   ...
```

Die einzelnen Ablaufsequenzen werden nun nicht vom Client (z.B. Web-Browser) der Reihe nach gesendet, sondern innerhalb einer Excalibur-Methode definiert und in einem Ablaufplan strukturiert. Dafür werden die einzelnen Ablaufsequenzen in einzelne Deskriptoren überführt und ein weiterer für die Methode erstellt. In letzterer werden die für die Abläufe benötigten Geräte, die in eigenen Deskriptoren konfiguriert worden sind, ebenso wie die Ablaufsequenzen referenziert. Die oben genannten Geräte werden nun Bezeichnungen zugeordnet. Deren Funktionalitäten werden über die bereits vorgenommene Kategorisierung definiert (s. Tabelle 2.2)

Tabelle (2.2)
Auflistung aller in INSTANT verwendeten Geräte, ihrer Bezeichnungen im Mapping und ihres Einsatzwecks.

Gerät	Bezeichnung	Einsatz
Ismatec RegloICC	samplepump	Pumpe für die Probenvorbereitung, inkl. Filterung, Aufkonzentration, Elution
Hamilton MVP	sampledistributionvalve	Ventil zum Schalten auf die unterschiedlichen Reagenzien in der Probenvorbereitung
Hamilton MVP	sampleswitchvalve	Ventil zum Schalten zwischen Probenvorbereitungs- und Messkreislauf in der Probenvorbereitung

Hamilton PSD4	mixturepump	Spritzenpumpe zum Befüllen und Mischen der Probe in die/der Probenschleife (fluidische Mischungsschleife)
Hamilton MVP	distributionvalve	Ventil zum Schalten auf die unterschiedlichen Reagenzien während des Messzyklus
Hamilton MVP	switchvalve	Ventil zum Schalten zwischen Probenschleife und Sensorkanal
VICI M6	precisionpump	Pumpe zum präzisen Transport der Probe zum Sensor

Mit diesen Informationen lässt sich die vollständige INSTANT Methode definieren:

```
<method xmlns="http://www.excalibur.net/method">
 <name>INSTANT</name>
 <deviceSpecification name="samplepump"/>
 <deviceSpecification name="sampledistributionvalve"/>
 <deviceSpecification name="sampleprepswitchvalve"/>
 <deviceSpecification name="mixturepump"/>
 <deviceSpecification name="distributionvalve"/>
 <deviceSpecification name="switchvalve"/>
 <deviceSpecification name="precisionpump"/>
 <sequence>
  <programSpecification name="initialize"/>
  <programSpecification name="elution"/>
  <programSpecification name="preconcentration"/>
  <programSpecification name="injection"/>
  <programSpecification name="samplemeasurement"/>
  <programSpecification name="sampleprepcleaning"/>
 <sequence>
</method>
```

Die referenzierten Programme können auf der beiliegenden DVD in der Methode INSTANT unter den zugehörigen Dateien (*programname*.xml) gefunden werden.
Der Excalibur Server kann diese Methode verarbeiten, lädt alle spezifizierten Geräte und

Programme, darüber hinaus alle benötigten Mappings. Letztere werden explizit *nicht* in der Methode definiert, da auf unterschiedlichen Systemen unterschiedliche Geräte eingesetzt werden könnten. Die in der Methode referenzierten Namen der „deviceSpecification“ sind also nicht die in der Schnittstellendefinition angegebenen Gerätereferenzen, sondern die in einem Mapping angegebenen Bezeichnungen. Auf diese Weise kann die INSTANT Methode (zumindest die Ablaufsteuerung) ohne Änderungen auf ein anderes System mit unterschiedlichen Geräten (derselben Kategorien) übertragen werden. Die einzelnen Ablaufsequenzen werden nach wie vor fehlerfrei ablaufen.
Die Methode kann nun über den Aufruf der URI „http://localhost:8080/method/INSTANT/start“ gestartet werden und anschließend über „http://localhost:8080/method/-INSTANT/state“ der Zustand abgerufen werden, welcher für die Ablaufverfolgung (siehe unten) benötigt wird. Letzterer liefert zu jedem Zeitpunkt den aktuellen Laufzeit-Status und die Position innerhalb der Ablaufsequenz.
Da auch hier die Steuerung mit dem Browser aufwendig und wenig komfortabel ist, werden wieder der Plugin-Container und mehrere Plugins benötigt, damit Benutzer auf einfache Weise mit dem INSTANT Laboraufbau interagieren können. Konkret werden für die Benutzerinteraktion zwei Plugins benötigt:[9] Eines zur Darstellung der Ablaufverfolgung und Systemkontrolle und eines zur Darstellung der Messsignale (s. Abbildung 2.17). Da im Falle von INSTANT keine Messdaten über die Excalibur-Software verarbeitet werden, müssen nicht wie oben im Falle der RIfS einzelne Writer definiert werden, sondern es werden die Daten direkt von den Plugins verarbeitet, ausgewertet und über die grafische Anzeige ausgegeben.

Die Ablaufverfolgung ist eine grafische Komponente bestehend aus den Repräsentationen der verwendeten Geräte. Jede der einzelnen Repräsentationen zeigt den aktuellen Status des Geräts (z.B. aktuelle Pumpgeschwindigkeit, Pumprichtung, Ventilposition, usw., s. Abbildung 2.17). Anhand dieser kann der Benutzer permanent verfolgen, welches Gerät gerade welche Aktion in welcher Phase durchführt. Da der Ablauf im Messsystem nur sehr schwer, aufgrund der vielen Reagenzien und Kanäle, mit dem Auge verfolgt werden kann, stellt dies ein komfortable Möglichkeit der Systemkontrolle dar.

Das Plugin zur Darstellung der Messsignale ist eine Kombination aus Bildanzeige der Kamera und zwei Graphen zur Anzeige der Impedanzspektren aus den Oszilloskopen. Hier werden permanent die aktuellen Bilder der Kamera angezeigt, welche in einem Hintergrundprozess aufgenommen und auf der Festplatte gespeichert werden. Das Ka-

[9] Zusätzlich zu den einzelnen Geräte-Plugins, mit welchen einzelnen Geräte unabhängig von einer Ablaufsequenz gesteuert werden können

merasystem kann dabei max. 100 Bilder pro Sekunde aufnehmen, die anschließend im Plugin gemittelt werden, interessante Bereiche (welche die Rezeptorschichten zeigen) ausgeschnitten und daraus über die Auswertung der Intensität die Konzentration an Analyt berechnet wird. Ähnliche Prozesse werden auch mit den Oszilloskopen durchgeführt, die bei sich ändernder Wechselspannungsfrequenz den zugehörigen Strom messen und anschließend Amplitude und Phasendifferenz anzeigen. Auch hieraus lassen sich konzentrationsabhängige Signale abhängig vom eingesetzten Analyten und Rezeptorschicht extrahieren.

Anhand dieses Beispiels kann gezeigt werden, dass auch Ablaufsteuerungen mit komplexen fluidischen Abläufen und vielen Geräten konstruiert werden können. Die Wahl, ob alles innerhalb einer Ablaufsequenz oder mehreren nacheinander folgenden, kann frei gewählt werden. Natürlich könnte auch hier jede einzelne fluidische Komponente durch eine andere Gleichartige ersetzt werden.

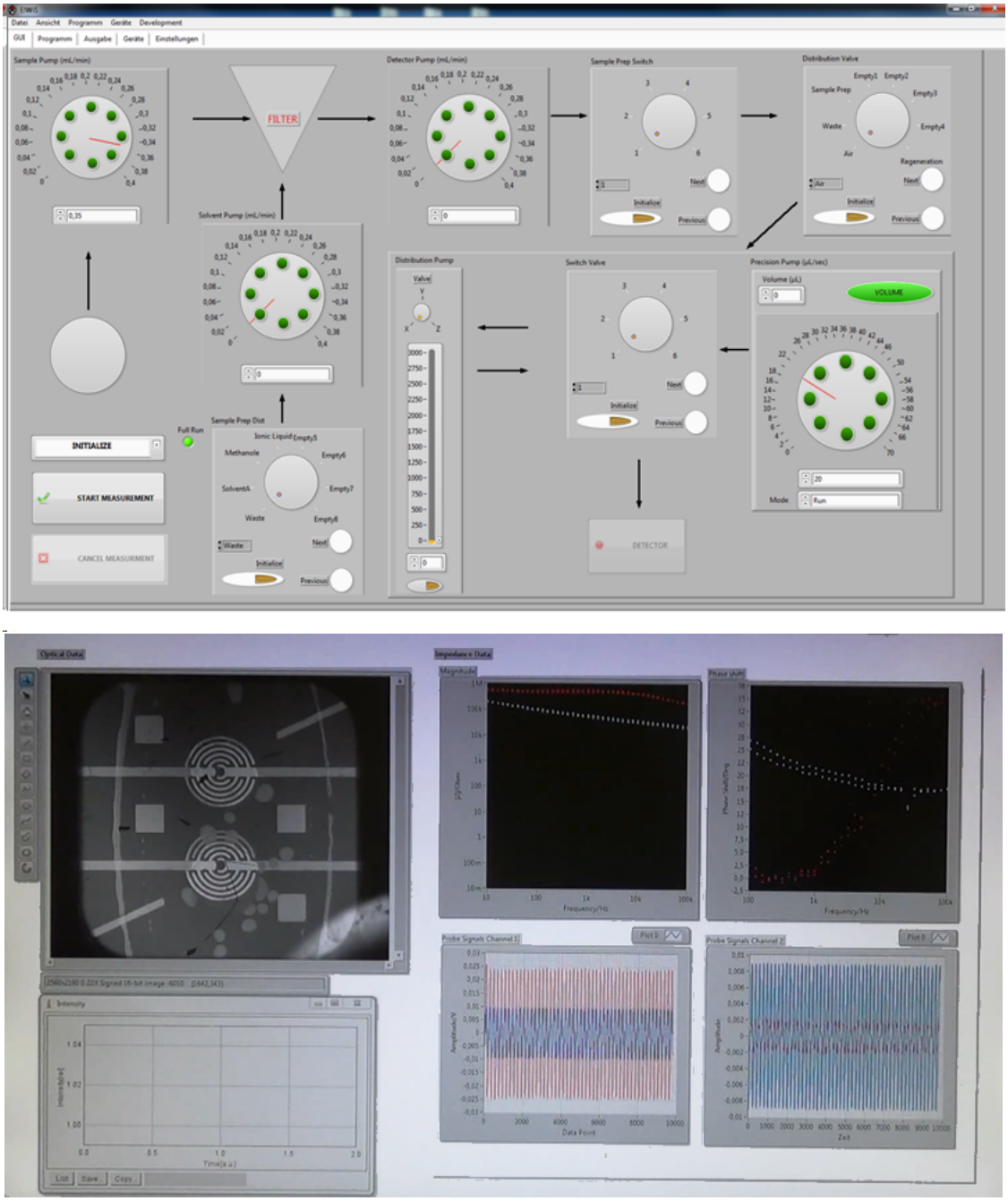


Abbildung (2.17)

Plugin zur Darstellung der Ablaufverfolgung einer INSTANT Messung (oben). Jedes Gerät hat hier seinen eigenen Bereich und zeigt den aktuellen Betriebszustand. Das Bild unten zeigt einen Screenshot der Anzeige der Messdaten von Kamera (links) und den Oszilloskopen (rechts) während einer INSTANT Messung.

Victor Hugo

Die Zukunft hat viele Namen: Für Schwache ist sie das Unerreichbare, für die Furchtsamen das Unbekannte, für die Mutigen die Chance.

3

Entwicklung eines konfigurierbaren Servers für die Laborautomatisierung

3.1. Theorie der Computersysteme und Hardware

3.1.1. Einführung in die Geräteautomatisierung

Im analytischen Labor verwendet man eine Vielzahl analytischer Aufbauten und Laborsysteme, die häufig aus verschiedenen kommerziellen Komponenten wie Pumpen, Detektionssystemen usw. zusammengeschaltet sind. Die Gerätekomponenten werden dazu an Computersysteme (welche dabei zu einem Teil des Messsystems werden) über gängige Schnittstellen angeschlossen. Die eingesetzte Steuerung wird dabei auf demselben Computersystem installiert und mit ihr können nun gerätespezifische Befehle über die Schnittstellen an das Gerät gesendet werden. Die Grammatik und Syntax der Gerätebefehle ist sehr vielfältig, ist aber in erster Linie unabhängig von der verwendeten Schnittstelle (s. Abschnitt 3.3), welche nur die elektronischen Rahmenbedingungen für eine Kommunikation vorgibt. Auf mögliche Schnittstellen und deren Eigenschaften soll dieses Kapitel eingehen.

3.1.2. Geräte und Schnittstellen

Ein Gerät ist ein [beweglicher] Gegenstand, mit dessen Hilfe etwas bearbeitet, bewirkt oder hergestellt wird[DUD06]. Spezieller wird sich innerhalb dieser Arbeit auf *elektrische* Geräte bezogen, welche über eine Kommunikationsschnittstelle verfügen, mit welcher sie sich an ein Computersystem anschließen und steuern lassen. Alle Geräte, welche dieser Definition folgen, lassen sich prinzipiell mit einer Ablaufsequenz automatisieren. Man unterscheidet bei diesen Geräte sog. zeichen- und blockorientierte Geräte. Letztere werden oft für parallele Datenübertragungen verwendet und nutzen Ressourcen des Betriebssystems. Daten werden in Blöcken übertragen und verwenden dafür systemeigene Caching-Mechanismen. Wahlfreier Zugriff, auch Direktzugriff, ist dabei Pflicht, d.h. sie lesen und schreiben in konstanter Zeit beliebige Elemente des Datenspeichers. Die meisten „internen“ Geräte eines Personal Computer (PC), wie Integrated Drive Electronics (IDE) oder Small Computer System Interface (SCSI) Festplatten, Compact Disc Read-Only Memory (CD-ROM)s, aber auch Diskettenlaufwerke zählen zu dieser Gruppe. Für die Laborautomatisierung oftmals interessanter sind die zeichenorientierten Geräte. Sie übertragen pro Zeiteinheit nur ein Zeichen, in der Regel ein Byte. Die Daten werden (meist) nicht gecacht, d.h. sofort übertragen, wodurch sie zur Gruppe der seriellen Datenübertragung zu zählen sind. Im Gegensatz zu blockorientierten Geräten,

bei welchen die Daten asynchron gelesen und geschrieben werden können, verläuft die zeichenorientierte Datenübertragung synchron, d.h. Lese- und Schreibfunktionen kehren nicht zurück, bevor der Vorgang abgeschlossen ist.

Serielle Schnittstellen

Für die Laborautomatisierung sind serielle Schnittstellen oftmals deshalb interessanter, da Laborgerät meistens außerhalb des PCs positioniert werden und über Kabel an den PC angeschlossen werden. Hierbei bedient man sich in den meisten Fällen serieller Protokolle wie RS-232 oder USB, wobei allerdings die Anzahl serieller Standards groß ist. Zu unterscheiden gilt es hier, ob es sich um sog. Punkt-zu-Punkt Verbindungen handelt oder ein Binary Unit System (BUS). Bei der Punkt-zu-Punkt Verbindung ist die Anzahl der Kommunikationspartner auf zwei beschränkt. Daten werden von einem Partner nacheinander entweder gelesen oder geschrieben. Vielseitiger sind die BUS-Systeme: Hier können mehr als zwei Geräte (sog. Knoten) an einem gemeinsamen Übertragungsweg angeschlossen sein. Daten werden dazu immer mit einer Adresse versehen (oftmals über einen eigenen Adressbus) und auf die Busleitung(en) gelegt. Das Bussystem selbst garantiert, dass zu einer gegebenen Zeit, genau ein Knoten Daten auf die Busleitung(en) legt. Der Empfängerknoten erkennt seine Zuständigkeit anhand der Adresse und liest die Daten vom Bus; außer bei speziellen Bussystemen gibt es genau einen Empfänger für einen bestimmten Datenstrom.

RS-232 Viele Laborgeräte verfügen über die RS-232 Schnittstelle. Es handelt sich hierbei um einen sehr alten und sehr einfachen seriellen Standard. Er ist bereits in den früher 1960er Jahren eingeführt worden und wird bis heute verwendet. Allerdings verliert er immer mehr an Bedeutung, da er durch modernere Bussysteme abgelöst wird, die zuverlässiger und schneller arbeiten.
Damit die serielle Schnittstelle verwendbar ist, müssen einige Einstellungen erklärt werden. Die wohl wichtigste Eigenschaft ist der *Port*, welche die Bezeichnung der Schnittstelle angibt, unter welchem das Gerät im Betriebssystem gefunden werden kann. Unter Windows® lauten diese meistens COMX, wobei X eine Zahl beginnend ab 1 darstellt; Unix Systeme verwenden hierfür meist /dev/ttyX. Das Betriebssystem reserviert hierbei einen gewissen Speicherbereich im Hauptspeicher-Adressraum, mit welchen über übliche Speicherzugriffsroutinen gearbeitet werden kann und somit ein vollständiger Zugriff auf die Hardware ermöglicht wird.

Eine weitere sehr wichtige Eigenschaft ist die Geschwindigkeit, welche indirekt über die *Baudrate* festgelegt wird. Diese muss für die konfigurierte Schnittstelle und dem Gerät übereinstimmen, da sonst keine Übertragung möglich ist. Je höher die Baudrate gewählt wird, desto schneller werden Daten über die Leitung gesendet. Allerdings unterstützt nicht jedes Gerät jede Baudrate, da diese abhängig vom verbauten Quarzoszillator ist. Typische Werte sind 75, 110, 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600 und 115200.
Für weitere Informationen und Eigenschaften soll auf Anhang C.1.1 verwiesen werden.

USB USB (Universal Serial BUS) ist ebenso wie RS-232 Schnittstelle eine serielle Schnittstelle. Die Datenübertragung erfolgt symmetrisch, d.h. es werden zwei Datenleitungen (D+ und D-) benötigt, um diese zu transportieren. Der Vorteile der USB Schnittstelle liegen in der hohen Datenübertragungsgeschwindigkeit von über 10 GBit/s (USB 3.0 Standard) und einer Bus-Topologie, wodurch an einem Host-Controller (meist der PC) bis zu 127 Geräte angeschlossen werden können. Ein weiterer Vorteil von USB ist Fähigkeit des Hot plug and play. So können Peripheriegeräte angeschlossen werden, ohne dabei den Host-Controller ausschalten zu müssen.
Für weitere Informationen und Eigenschaften soll auf Anhang C.1.2 verwiesen werden.

Parallele Schnittstellen

Der Unterschied einer parallelen zu einer seriellen Schnittstelle liegt weitestgehend darin, dass gleichzeitig mehrere Bits parallel übertragen werden können. Dazu werden immer mehrere Leitungen benötigt und zwar entsprechend viele, je nach Anzahl der parallel zu übertragenen Bits. Wird heutzutage von Geräten mit paralleler Schnittstelle gesprochen, so ist meist der sog. Druckerport damit gemeint, welcher allerdings nach Einführung von schnelleren seriellen Schnittstellen wie USB als veraltet gilt. Immer noch Verwendung finden parallele Bus-Standards, die in die Peripherie von PCs verbaut werden. Dazu gehören der Accelerated Graphics Port (AGP)-, Peripheral Component Connect (PCI)-BUS, aber auch SCSI und der IEC-625-BUS besser bekannt unter General Purpose Interface Bus (GPIB), welcher vorrangig zur Verbindung von Messgeräten dient.

GPIB Die GPIB Schnittstelle ist 1965 von Hewlett-Packard unter dem Namen (HP-IB) eingeführt worden, um ihre eigenen programmierbaren Instrumente mit Computern

zu verbinden. Aufgrund der hohen Übertragungsraten von 1 MB/s ist der BUS relativ schnell bekannt geworden. Mittlerweile ist der BUS über mehrere Stufen standardisiert worden und wird unter dem IEEE Standard 488-1987 oder IEC-625 geführt, weshalb er manchmal auch als IEC-BUS bezeichnet wird. Heutzutage wird der Name General Purpose Interface BUS (GPIB) am häufigsten verwendet.
Wie auch USB handelt es sich um eine BUS-basierte Schnittstelle, d.h. es können mehrere Geräte hintereinander geschaltet werden, wobei deren Anordnung keine Rolle spielt. Auf diese Weise können bis zu 15 Geräte ausgehend von einem einzigen Computer-Port angeschlossen werden. Durch die 8 Datenleitungen können sehr hohe Datenübertragungsgeschwindigkeiten erreicht werden, weshalb sich diese Schnittstelle für Messinstrumente eignet.
Für weitere Informationen und Eigenschaften soll auf Anhang C.1.3 verwiesen werden.

Indirekter Zugriff auf Laborgeräte Verfügt ein Gerät über eine der oben genannten Schnittstellen, so kann dieses rein prinzipiell gesteuert und/oder geregelt werden. Wird in Geräte allerdings eine proprietäre, das heißt vom Hersteller selbst konzipierte, Schnittstelle verbaut, so kann oft nicht direkt auf das Gerät zugegriffen werden. In diesen Fällen muss der Hersteller eigene Bibliotheken in Form von DLL- (Windows) oder SO-Dateien (Linux) zur Verfügung stellen. Ist dies der Fall, kann nur über Umwege auf die Funktionalität des Gerätes zugegriffen werden. LabVIEW® unterstützt die Verwendungen von Bibliotheken, allerdings wird hierfür die Programmierumgebung benötigt, da Bibliotheken individuell konfiguriert werden müssen. Dies widerspricht auf den ersten Blick der Annahme, das Geräte ohne Neukompilierung hinzugefügt werden können. Nun ist es in LabVIEW® möglich VIs aus einem kompilierten Programm heraus auszuführen, wodurch es indirekt wieder möglich ist aus einer Kollektion an Virtual Instruments (VI)s auf die Dynamic Linked Library (DLL) zuzugreifen. Dafür müssen lediglich Wrapper-VIs konstruiert werden, welche die Funktionalität der entsprechenden Bibliothek abbilden. Auf diese Weise ist es möglich auch auf proprietäre Schnittstellen von Geräteherstellern zuzugreifen und sie in ein vorhandenen Laborautomatisierungskonzept aufzunehmen. Im Hauptteil werden die Konzepte „Treiber“ und „Automatisierung“ näher behandelt, weshalb die Erklärung von Geräten mit nicht standardisierten Schnittstellen an dieser Stelle genügen soll.

3.2. Theorie der Software-Technologien in der Laborautomatisierung

3.2.1. Einführung in die Automatisierung

Im Folgenden soll der Prozess der Automatisierung definiert werden, unter anderem auch eine Einführung in die Laborautomatisierung gegeben werden. Zuletzt sollen in diesem Kapitel die Probleme benannt werden, die aktuell beim Entwicklungsprozess von Ablaufsteuerungen auftreten.

Automatisierungstechnologien

Automatisierung ist nach [Bro87] die

> „Anwendung von technischen Mitteln, mit deren Hilfe ohne Einflussnahme des Menschen Arbeitsmittel teilweise oder ganz nach vorgegebenen Programmen bestimmte Operationen durchführen. Allgemeine Automatisierungsziele sind: Erhöhung der Produktivität, Verbesserung der Produktqualität, höhere Zuverlässigkeit und Sicherheit, wirtschaftlicherer Rohstoff- und Energieeinsatz, Schonung der Umwelt, Humanisierung der Arbeit sowie insbesondere die Ermöglichung von Prozessen, die bei manueller Prozessführung infolge der dem Menschen innewohnenden Unzulänglichkeiten nicht durchführbar sind.“

Die Automatisierung ist ein weitreichendes Gebiet, wie man es bereits an der sehr allgemein gehaltenen Definition erkennen kann. Neben der industriellen Automatisierung, welche den Fokus hat, industrielle Maschinen und Anlagen selbstständig und ohne Mitwirkung von Menschen zu betreiben, gibt es auch andere Lebensbereiche, in dem die Automatisierung Einzug gehalten hat. So verzeichnet man heute einen starken Anstieg in den Entwicklungen der Heim- und Gebäudeautomatisierung bzw. ganz allgemein in elektronischen Alltagsgegenständen. Letzteres wird unter dem Begriff „Internet der Dinge“ propagiert.

Anwendungsfall: Laborautomatisierung Ein weiteres Feld und der Fokus dieser Arbeit ist die Laborautomatisierung. Unter der Laborautomatisierung versteht man die

Automation von Prozessen in chemischen, pharmazeutischen oder medizinischen Laboratorien. Sie stellt Systeme bestehend aus Geräten, Softwarelösungen und den dazugehörigen Automationsprotokollen zur Verfügung.

Zum Thema Laborautomatisierung merkt das Fraunhofer Institut für Produktionstechnik und Automatisierung an:[FIP14]

> "Für die Prozessautomatisierung im Labor gelten [...] andere Anforderungen als in klassischen Produktionen. So ist z.B. die Aufgabe von Flexibilität, die mit der Einführung von starren Automationssystemen verbunden ist, im Labor nicht akzeptabel und verhindert die Einführung von Automationshilfen für Arbeitsprozesse im Labor."

Daraus wird ersichtlich, dass sich die Laborautomatisierung von anderen Sparten, vor allem der von industriellen Automatisierungssystemen (für beispielsweise die Fertigungstechnik), dadurch unterscheidet, dass sich die Arbeitsprozesse innerhalb der Laborumgebungen häufig ändern können. Deshalb müssen auch die Laborautomatisierungskonzepte äußerst flexibel sein und sich relativ einfach an die geänderten Bedingungen anpassen können. Zusätzlich sollen sich Automatisierungskonzepte auch einfach und nahtlos in bestehende Arbeits-, Daten- und Rechnerstrukturen im Labor integrieren lassen, ohne tiefgreifende Änderungen oder Neuanschaffungen vornehmen zu müssen. „Einfach" bedeutet zusätzlich, dass dafür kein speziell geschultes Personal eingestellt und eingelernt werden muss.

Übersicht verschiedener Automatisierungslösungen

Automatisierungslösungen gibt es sehr viele auf dem Markt. Die meisten sind spezialisiert für industrielle Abläufe und aufgrund fehlender Flexibilität nur schwer in Laboratorien einsetzbar. Eine Liste über derzeit am Markt verfügbare Lösungen gibt [nee14]. Es handelt sich hierbei um sog. Software-SPSen. Die SPS ist bereits oben eingeführt worden und sie stellt auf Anlagenebene den derzeitigen Stand der Dinge dar. Dennoch ist der Einsatz von SPSen in den meisten Fällen sehr komplex und bedingt immer speziell geschultes Personal.
Bei einer Software-SPS hingegen dient ein PC mit speziellen Steckkarten und Erweiterung als Gerät, wobei ein Echtzeit-Betriebssystem als Mediator zwischen Automatisierungssequenz und Hardware benötigt wird. Hier wird meist Windows® verwendet. Zwar fällt für letzteres der Einkauf eines speziellen und meist teuren integrierten Gerätes weg,

allerdings bleibt die Komplexität nach wie vor erhalten.

LabVIEW® LabVIEW® (Laboratory Virtual Instrumentation Engineering Workbench) ist eine Systemdesignsoftware, welche von der Firma National Instruments entwickelt wird, um innerhalb kürzester Zeit Anwendungen für Mess-, Steuer- und Regelsysteme zu entwickeln. Es ist speziell für Ingenieure und Wissenschaftler entwickelt und soll ihnen vom Entwurf bis hin zur fertigen Anwendung als Plattform dienen. Es bedient sich hierzu einer grafischen Programmierung genannt "G", welche es den Benutzern ermöglicht den Programmablauf durch erhöhte Lesbarkeit besser zu verstehen; auch die Lernkurve ist sehr flach. Darüber hinaus gibt es eine umfangreiche Treiberbibliothek, um eine sehr große Palette an (Mess-)Geräten einfach einzubinden und mit ihnen direkt zu kommunizieren. Eine weitere Stärke ist der gleichzeitige, implizite Aufbau von grafischen Oberflächen während der Programmierung, da LabVIEW®-Funktionen (sog. VI) immer automatisch aus zwei Teilen bestehen, dem sog. Blockdiagramm und dem Frontpanel. Das Frontpanel wird dadurch immer gleich mitentwickelt. Eine umfangreiche Bibliothek an grafischen Komponenten wie z.B. Plots, Listen, Tabellen, Bäumen, usw. und deren einfache Verknüpfbarkeit mit Datenstrukturen vereinfacht es, Daten auf sehr komfortable Weise direkt grafisch auszugeben. Zusätzlich ist LabVIEW® unter allen gängigen Betriebssystemen Windows, Mac, Sun, Linux, RTTargets und Unix verfügbar und garantiert die Einsetzbarkeit der entwickelten Anwendung über Betriebssystemgrenzen hinweg.
Ist eine Anwendung auf dem Entwicklungssystem programmiert worden, kann sie durch einen integrierten Compiler in eine ausführbare Datei übersetzt und auf ein anderes System übertragen werden. Dieses Zielsystem muss nicht über die LabVIEW® Entwicklungsumgebung verfügen, muss aber dieselbe Architektur aufweisen (dasselbe Betriebssystem und Prozessorarchitektur). Eine frei erhältliche LabVIEW® Runtime Engine erlaubt dann die Ausführung der ausführbaren Datei.

Aufgrund der sehr guten Hardwarekontrolle und der (relativ umfassenden) Plattformunabhängigkeit von LabVIEW® ist die Entscheidung getroffen worden, den Kern des Excalibur Server in LabVIEW® zu entwickeln. LabVIEW® stellt eine vollständig entwickelte Programmiersprache auf Basis eines grafischen Quellcodes dar und enthält den benötigten Funktionsumfang, um die geplante Lösungsstrategie zu entwickeln. Im Gegensatz dazu soll aus der zu entwickelnden Lösung keine neue Programmiersprache entstehen (was LabVIEW® ist), sondern eine auf den Anwendungsbereich „Laborautomatisierung"

zugeschnittene Plattform, die für den Systemintegrator auf einem klar verständlichen und intuitiven Konzept beruht. LabVIEW® allein vereinfacht zwar die Programmierung von Ablaufsteuerungen im Labor im Gegensatz zu tieferen Programmiersprachen wie C/C++, allerdings werden weiterhin tiefe Kenntnisse in Programmierung und Informatik benötigt, auch wenn der Code nicht geschrieben, sondern „gezeichnet“ wird. Darüber hinaus verleitet das „Zeichnen“ zur Generierung von unsorgfältig geplanten Code, der nur noch sehr schwer wartbar ist.

3.2.2. Datenspeicherung

Im Bereich der Sensorentwicklung aber auch in anderen analytisch ausgerichteten Laboratorien, spielt nur nicht die automatisierte Steuerung von Geräte eine große Rolle, sondern auch die Generation von aussagekräftigen Daten. Aus diesen Daten sollen nach der Messung Informationen über das zugrunde liegende Problem getroffen werden können. In dieser Hinsicht ist nicht nur das Generieren der Daten ein wichtiger Fakt, sondern auch deren sichere Datenspeicherung. Unter Datenspeicherung wird das persistente Halten der Daten auf unterschiedlichen Medien verstanden, um sie zu einem späteren Zeitpunkt wieder unverändert abrufen zu können. Dafür werden hier ausschließlich elektronische oder magnetische Speichermedien verstanden, auf welchen Inhalte digital abgelegt werden können.

Lokale Datenspeicherung Obwohl heute in jedem Labor Computernetzwerke anzutreffen sind, werden Messdaten nach wie vor auf dem lokalen Computer generiert und auch gespeichert. Erst im Anschluss werden die Daten manuell an einen anderen Speicherort transportiert, an welchem sie bearbeitet werden. Diese nicht-zentralisierte Speicherung ist insofern problematisch, als dass Daten oft verstreut auf vielen Computern liegen und es zu Synchronisierungsproblemen kommen kann. Ein weiteres Problem ist die Datensicherheit in zweierlei Hinsicht. Zum einen kann beim Ausfall eines lokalen Computers zu Datenverlusten kommen, da oft nicht jeder Messrechner periodisch gesichert wird. Zum anderen sind die Zugriffsberechtigungen auf lokalen Computern oft sehr sporadisch, da viele Benutzer Zugriff auf das lokale System haben müssen, um überhaupt Messungen durchführen zu können. Aus diesem Grund haben sehr viele Benutzer Vollzugriff auf sämtliche Daten, auch solchen, die nicht von ihnen erzeugt worden sind. Für alle genannten Punkte können Gegenmaßnahmen getroffen werden, um die lokale Datenin-

tegrität und -verfügbarkeit zu erhalten. So können natürlich geeignete Datensicherungs- und -synchronisierungskonzepte eingeführt, verschlüsselte Laufwerke und ein geeignetes Benutzermanagement eingeführt werden. Dennoch handelt es sich hierbei wieder um Insellösungen, welche bei Veränderung einer beliebigen Architektur an die geänderten Bedingungen angepasst werden müssen. Dies führt oft zu Problemen.

Zentralisierte Datenspeicherung Eine oftmals bessere Lösung ist die zentralisierte Speicherung von Messdaten und das automatische Ablegen der Daten auf ebendiesen zentralisierten Punkten direkt von der Mess-Software. Bei diesem Ansatz befinden sich die Daten immer an einem (oder mehreren) bekannten, zentralen Punkten, den Daten-Servern. Diese Daten-Server lassen sich problemlos warten und sichern, da es sich um spezialisierte und regelmäßig gesicherte Systeme handelt, wodurch mögliche Datenverluste sehr unwahrscheinlich werden. Darüber hinaus können die Zugriffsberechtigungen von Datenabfrage und -Speicherung unabhängig gestaltet werden. Dies bedeutet, das die Datenspeicherung standardisiert in die Mess-Software integriert werden kann, ohne ein komplexes Benutzer-Organisationssystem (ein Benutzername ist ausreichend) für jede Software-Komponente entwickeln zu müssen, während die Daten-Abfrage von etablierten Benutzer-Organisationssystemen auf dem Daten-Server geregelt werden können. Mit Hilfe dieses Ansatzes kann die Datensicherheit in beiden oben genannten Punkten gewährleistet werden.

Dennoch gibt es bei der zentralisierten Datenspeicherung Nachteile. Es benötigt immer ein funktionsfähiges Computernetzwerk, um die Daten vom Mess-Computer auf das entfernte System zu transferieren. Es muss demnach in der Mess-Software garantiert werden, dass bei einem Netzwerk-Ausfall die Daten weiterhin vorhanden sind und lokal hinterlegt werden, bis das Netzwerk wieder erreichbar ist. Ein zweiter Nachteil ist die Unsicherheit von Netzwerken, da Daten oft sehr viele Zwischenstationen (Switches, Router, usw.) auf dem Weg vom Mess-Computer zum Daten-Server passieren müssen. Dadurch ist ein Abgreifen der Daten von Dritten möglich. Die Messsoftware muss deshalb die Daten über verschlüsselte Verbindungen zum Daten-Server übertragen oder die Daten selbst verschlüsseln. Dies erhöht selbstverständlich die Komplexität der Messsoftware, wofür aber bereits vorhandene Technologien eingesetzt werden können (z.B. Apache Webserver, s. Abschnitt 3.3.10). Bei Betrachtung der Vor- und Nachteile zeigt sich trotzdem, dass zentralisierte Datenhaltung bevorzugt verwendet werden sollte, da gegen die Nachteile (durch geeignete Planung der Messsoftware) Maßnahmen getroffen

werden können. Excalibur hat all die genannten Punkte bereits integriert und erlaubt dem Benutzer beide Möglichkeiten zu verwenden. Es ist sowohl lokale wie auch zentralisierte Datenspeicherung möglich. Im Folgenden wird darauf eingegangen, wie Daten gespeichert werden können und welche Formate unterstützt werden.

Datenbanken

Eine etablierte Möglichkeit bieten Datenbanksysteme, deren wesentliche Aufgabe es ist, große Datenmengen effizient und persistent zu speichern. [Vos12] definiert Datenbanksysteme folgendermaßen: „Datenbanksysteme sind ein weit verbreitetes technisches Hilfsmittel zur effizienten, rechnergestützten Organisation, Speicherung, Manipulation, Integration und Verwaltung großer Datensammlungen. Sie basieren auf der Idee, Daten über die reale Welt, welche von Anwendungsprogrammen verarbeitet werden, als von diesen Programmen unabhängige und integrierte Ressource zu behandeln, und stellen dazu spezifische Funktionalität bereit.“

Die Vorteile von Datenbanksystemen spiegeln die oben genannten Punkte sehr gut wieder. Sie bieten Datensicherheit, Transaktionskontrolle, Datenintegrität und zusätzlich die Fähigkeit auf sie mit mehreren Benutzer gleichzeitig zuzugreifen. Im Gegensatz zur Speicherung von Daten in Dateien, um sie anschließend richtig auf den Datenspeichern zu sortieren, verwenden Datenbanksysteme eigene Zugriffsprozeduren, welche die Daten über eine vom verwendeten Datenbanksystem abhängigen Syntax in ihrer eigenen Struktur ablegen. Die Syntax und die Ablagestruktur sind dabei unterschiedlich für die verschiedenen Produzenten für Datenbanksysteme. Es haben sich aber im Laufe der Zeit bestimmte Abfragesprachen etabliert. Die wohl bekannteste davon ist die Structured Query Language (SQL), welche auf relationaler Algebra beruht. Die Syntax der SQL ist sehr einfach aufgebaut und sie wird von den gängigen Datenbanksystemen durchweg unterstützt, wenn auch die Syntax leicht voneinander abweicht. Diese Standardisierung hat sehr stark dazu beigetragen, eine gewisse Unabhängigkeit der Software von verschiedenen Datenbanksystemen zu erreichen. Für weitere Informationen sei auf einschlägige Fachliteratur verwiesen[Vos08].

SQL

Formale Sprachen, Chomsky-Hierarchie und Backus-Naur-Form

Formale Sprachen Zum Verarbeiten von Daten verwenden Computersysteme formale Sprachen. Gegenüber natürlichen Sprachen haben sie eine *exakte* Definition der zulässigen Ausdrücke und keine Kontextabhängigkeit der Bedeutung. Dadurch können sie von den Computersystemen einfacher verarbeitet werden. Jede Sprache besteht aus Wörtern, die aus einem Alphabet zusammengesetzt werden. Die Syntax legt dabei fest, welche Ausdrücke gebildet werden können und die Semantik definiert, was die einzelnen Ausdrücke bedeuten. Das Alphabet besteht dabei aus einem geordneten Zeichenvorrat aus Symbolen (z.B. Zahlen, Sonderzeichen, alphanumerische Zeichen, usw.), welche zu einer Zeichenkette synthetisiert werden können. Letztere stellt eine endliche Folge von Symbolen aus dem zugrunde liegenden Alphabet dar, die ohne Zwischenraum verkettet werden können. Eine formale Sprache ist letztendlich die Menge an Zeichenketten, welche anhand grammatischer Regeln synthetisiert werden kann. Die Grammatik selbst besteht aus grammatischen Regeln und Komponenten, d.h. Variablen (Nichtterminalsymbole), einem endlichen Terminalalphabet und einer Startvariablen. Ein „Wort“ der formalen Sprache kann anhand des Ableitungsbaums konstruiert werden. Dabei wird an die Wurzel des Baums die Startvariable gesetzt. Ein Blatt des Baums als gegenteiliges Extrem stellt einen Endpunkt dar, der keine Kindelemente hat und damit als Terminalsymbol zu betrachten ist. Ein Knoten des Baums ist ein Nichtterminalsymbol, wenn er Kindelemente enthält, die von links nach rechts markiert werden. Eine Grammatik hat nun die Aufgabe ein Symbol (Terminal oder Nichtterminal) anhand einer Regel in ein anderes/andere zu überführen (linksseitige Symbole werden in rechtsseitige Symbole überführt), d.h. anhand dieser wird jeweils ein Pfad gewählt, der letztendlich immer in einem Blatt des Baums endet. Das Wort ist letztendlich die Folge der (gewählten) Blätter von links nach rechts gelesen.

Chomsky-Hierarchie Der Linguist Noam Chomsky hat formale Sprachen induktiv definiert und anhand ihrer Grammatiken hierarchisch untergliedert. Er unterscheidet dabei 4 Typen an Sprachen: Unbeschränkte (Typ 0), kontextsensitive (Typ 1), kontextfreie (Typ 2) und reguläre Sprachen (Typ 3). Dabei enthalten die tieferen Typen immer die höheren Typen, d.h. eine Typ 2 Sprache ist beispielsweise immer auch eine Typ 3 Sprache. Bei regulären Sprachen resultiert bei Anwendung einer Typ 3-Grammatik auf ein nichtterminales Symbol immer ein terminales. Beispiele hierfür sind reguläre Ausdrücke, auf welche genauer in Abschnitt 3.3.3 eingegangen wird. Bei kontextfreien Sprachen kann

beim Anwenden einer Typ 2-Grammatik auf ein nichtterminales Symbol eine beliebige Folge an nichtterminalen und terminalen Symbolen resultieren. Beispiele hierfür sind gängige Programmiersprachen. Typ 1-Grammatiken verhalten sich ähnlich, außer dass auf der linken Seite eine beliebige Folge aus nichtterminalen und terminalen Symbolen existieren darf, auf welche die Grammatik angewendet wird. Einzige Voraussetzung ist, dass auf der linken Seite mindestens ein nichtterminales Symbol stehen muss. Bei Typ 0-Sprachen gibt es letztendlich keine Beschränkungen mehr. Darunter fallen alle Sprachen, unter anderem auch die natürlichen.

Backus-Naur-Form Die Backus-Naur-Form (BNF) (von John W. Backus und Peter Naur) stellt eine vereinfachte Darstellung kontextfreier Grammatiken dar. Die einzelnen Symbole werden dabei nicht in einer Baumstruktur, sondern in einer Liste angeordnet. Nichtterminale Symbole werden dabei durch < und > eingeschlossen, die Zeichenfolgen *::=* und | sind als Metasymbole für „ist definiert als“ und „oder“ festgelegt. Es werden immer linksseitige nichtterminale Symbole mit Hilfe des *::=* Metasymbols dem/den rechtsseitigen Symbol(en) gegenübergestellt, wobei die Definitionen für die nichtterminalen Symbole nach der Verwendung (in der Liste unterhalb) erfolgen. Weitere Metasymbole innerhalb der Erweiterte Backus-Naur-Form (EBNF) sind (...|...) zur Darstellung von *genau einer* Alternative, [...] zur Darstellung von Optionen (d.h. der Inhalt der Klammer kann einmal oder keinmal vorkommen) und letztendlich den {...} zur Darstellung von Wiederholungen (d.h. Inhalt der Klammer steht mindestens einmal). Zusätzlich seien in dieser Arbeit die Symbole *space* und *vline* definiert, welche für das Leerzeichen und den senkrechten Strich | stehen sollen.

Dateien und Dateiformate

Die wohl gängigste Art der Datenspeicherung findet in Dateien statt. Das Wort „Datei“ ist eine Kombination der der Wörter Daten und Kartei und bezeichnet eine Menge an zusammengehörigen Daten, welche auf einem Datenspeicher hinterlegt sind. Im Gegensatz zu Daten in flüchtigen Speichern liegen die Daten persistent vor, d.h. sie sind auch nach dem Abschalten des Datenspeichers vorhanden. Innerhalb von Dateien können Daten auf fast beliebige Weise zusammengesetzt und formatiert sein. Sie stellen nichts weiteres dar als eine eindimensionale Liste an Bytes, welche durch die Dateigrenzen beschränkt ist. Da diese Liste beliebig zusammengesetzt sein kann, ist eine sinnvolle Interpretation der Daten innerhalb der Datei nur durch eine Software möglich, welche diese

Daten interpretiert. Prinzipiell gibt es zwei Arten von Dateien, Text- und Binärdateien (oder Kombinationen aus beiden), wobei beide Kategorien beliebig viele Untermengen besitzen. Das Datenformat einer Textdatei besteht aus Bytes, die mit Hilfe einer Zeichensatztabelle, wie z.B. ASCII (s. unten), in Zeichen wie Buchstaben und Zahlen, aber auch Steuerzeichen übersetzt werden. Im Gegensatz dazu benötigt man für Binärdateien keine derartigen Tabellen für die Interpretation der enthaltenen Bytes; sie werden von der Software interpretiert.

ASCII/Unicode Dateien Wie oben erwähnt müssen Textdateien mit Hilfe von Zeichsatztabellen interpretiert werden. Dafür hat sich zum einen die American Standard Code for Information Interchange (ASCII)-Tabelle etabliert, welche einzelne Bytes bestimmten Zeichen zuordnet. Der ASCII definiert eine Tabelle, in welcher 128 Zeichen, wovon 33 nicht-druckbar sind, jeweils 7 Bits gegenübergestellt sind. Für das englische Alphabet ist diese Anzahl an Zeichen ausreichend, stößt aber bereits für die deutsche Sprache (z.B. Umlaute) an ihre Grenzen. Die Folge sind einige Änderungen, die z.T. den Zeichensatz erweiterten, z.B. um die griechischen Buchstaben. Letztendlich hat sich ein weiterer Zeichensatz, das Unicode, gebildet, welcher den ASCII Zeichensatz entsprechend erweitert. Es existieren verschiedene „Ausbaustufen“ wie beispielsweise das Universal Character Set - Transformation Format (UTF)-8, welches für die unteren 7 Bits d.h. die Zahlen 0 - 127, den ASCII Zeichensatz abbildet, allerdings für Zahlen größer 127 gängige europäische Zeichen aufgenommen hat. Um auch noch weitere Zeichen wie kyrillische oder chinesische Zeichen abzubilden, ist der UTF-Satz entsprechend vergrößert worden. Weitere Informationen bzgl. ASCII und UNICODE können im Anhang C.2.1 nachgelesen werden.

ASCII

UNICODE

TDM Dateien Um Messdaten, welche fast ausschließlich aus Zahlen bestehen zu speichern, bieten sich Binärdateien an. Diese bieten ein optimales Format, um effizient hinsichtlich Dateigröße und aus Gründen der Performanz Daten auf Datenträgern zu speichern. Der Nachteil ist aber auch direkt offensichtlich. Man benötigt spezialisierte Software, um die Daten wieder zu interpretieren, weil die Struktur der Dateien nicht durch eine Zeichensatztabelle festgelegt ist, sondern durch die Grammatik, nach welcher ein Prozess Bytes in die Datei schreibt; und nur dieser Prozess (oder ein gleich strukturierter) kann diese Daten wieder interpretieren. Die Möglichkeit Dateien zwischen verschiedenen Anwendungen auszutauschen ist aus diesem Grund nur schwer möglich. Zusätzlich ist ein Durchsuchen der Dateien auch nur durch eine Software möglich, welche die Daten

interpretieren kann. Aus diesem Grund hat die Firma National Instruments das Technical Data Management (TDM) bzw. Technical Data Management Streaming (TDMS) entwickelt. Dieses Dateiformat verknüpft alle oben genannten Vorteile für sowohl Text- als auch Binärdateien und reduziert deren Nachteile auf die Verwendung einer Bibliothek. Für nähere Informationen zum TDM Dateiformat soll auf Anhang C.2.2 und auf das Whitepaper bei National Instruments verwiesen werden[Ins13].

Auszeichnungssprache und Datenaustauschformat als Textstrukturierungmöglichkeiten Liegt der Fokus der produzierten Daten nicht auf der effizienten Speicherung, sondern auf Datenaustausch mit anderen Anwendungen und Systemen, sind Binärdateien vollkommen ungeeignet. Selbst strukturierte Textdokumente stoßen sehr schnell an ihre Grenzen, wenn die enthaltenen Daten nicht für den Benutzer gedacht sind, sondern zum Austausch mit anderen Anwendungen (sog. Machine to Machine Communication, M2M). Hier werden standardisierte Strukturierungen benötigt, die in einem wohldefinierten Format vorliegen. Zwar können TDM oder TDMS Dateien hierfür verwendet werden, doch sind sie vor allem im technischen Umfeld verbreitet und haben ein sehr rigide Struktur. Für den Datenaustausch hat sich eine neue Kategorie an Datenformaten entwickelt, die sog. Datenaustauschformate und im weitesten Sinn auch die Auszeichnungssprachen. Es sind unzählige Formate hierfür definiert werden, wobei die bekanntesten wohl das RTF (Rich Text Format), HTML (HyperText Markup Language), XML (eXtensible Markup Language), JSON (JavaScript Object Notation) oder CSV (Comma-Separated Values) gezählt werden können. Auch für Grafiken sind Austauschformate definiert worden, z.B. JPEG (Joint Photographic Expert Group) oder PDF (Portable Document Format). Diese Liste kann noch um unzählige Formate erweitert werden, soll aber für die hier benötigten Ansprüche genügen.

XML Excalibur macht intensiven Gebrauch von der eXtensible Markup Language (XML) als Textstrukturierungsformat. Aufgrund der weiten Verbreitung von XML, vor allem im Umfeld web-basierter Anwendungen, bietet es sehr gute Möglichkeiten standardisierte Formate zu definieren, um sie mit einer Vielzahl an Anwendungen kompatibel zu machen. Darüber hinaus bietet XML durch seine Baumstruktur, der Typisierung und dem Einbinden von externen Dokumenten unzählige Möglichkeiten, Textdokumente zu strukturieren. Die Struktur von XML ist sehr einfach. XML besteht aus Knoten, welche wiederum eine beliebige Anzahl an Knoten enthalten. Zusätzlich kann jeder Knoten eine beliebigen Anzahl an Attributen bestehend aus Schlüssel-Wert Paaren enthalten. Dar-

aus ergibt sich folgende Struktur (beispielsweise für einen Gerätetreiber, vollständiges Listing im Anhang):

Listing (3.1)

Beispiel für ein XML Dokument.

```
<device>
  <name>TestDevice</name>
  <description lang="en">This is a test device</description>
</device>
```

XML ist so vielseitig, dass selbst kommerzielle Produkte wie Microsoft Office® ihre Dateien seit der Version 2007 in einem auf XML basierenden Format speichern.

Allerdings ist XML selbst nur eine kontextfreie Auszeichnungssprache, in welcher es möglich ist, jedwede Art von Struktur abzubilden. Dabei ist es nicht notwendig, dass die Struktur einen inhaltlichen Sinn ergibt, solange der XML Code syntaktisch richtig ist. Die Definition eines Kontexts lässt sich über die sog. XML Schema Definition (XSD) realisieren. Mit Hilfe von XSD lässt sich XML strukturieren, typisieren und zum Teil auch generieren. XSD ist sozusagen eine Schnittstellenbeschreibung, welche Elemente oder Knoten im vorliegenden XML vorkommen müssen/können, welche Kindelemente und Attribute sie enthalten müssen/können und welcher Datentyp damit beschrieben werden soll. Erst mit Hilfe von XSD ist es möglich, XML korrekt zu validieren. Dabei ist XSD selbst in XML geschrieben. Im Folgenden sei eine XSD gezeigt, welche den oben gezeigten XML Block spezifizieren soll:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="www.excalibur.net/driver"
  targetNamespace="www.excalibur.net/driver>
  <element name="device" type="tns:deviceType"/>

  <xs:complexType name="deviceType"
    <xs:sequence>
      <element name="name" type="string"/>
      <element name="description">
        <complexType>
          <simpleContent>
                    <extension base="string">
```

```
13                    <attribute name="lang" type="string"
14                      use="optional" default="de"/>
15                  </extension>
16                </simpleContent>
17          </complexType>
18        </element>
19      </xs:sequence>
20    </xs:complexType>
21 </xs:schema>
```

In einigen wenigen Worten soll dieses Schema erklärt werden. Das Schlüsselwort „element“ gibt vor, dass in einem XML Dokument, welches die Vorgaben dieses Schemas erfüllen soll, ein Element „device“ (in eckigen Klammern < >) enthalten kann (s. Zeile 1 in Listing 3.1). Diese Vorgabe legt das Attribut „name“ fest, außerdem das es sich bei dem Element um den Datentyp „deviceType“ handelt, der ebenso in diesem Schema definiert ist. Letzteres wird durch das Attribut „type“ festgelegt, das auf den Namen „tns:deviceType“ zeigt. Die Zeichenfolge „tns“ sagt dabei aus, das sich der Datentyp im Namensraum „www.excalibur.net/driver“ befinden muss und dieser Namensraum in diesem Dokument definiert ist (Attribut „targetNamespace“ zeigt auf denselben Namensraum). Ebenso gibt die Zeichenkette „xs”vor, dass sich das darauf folgende Schlüsselwort im Namensraum „http://www.w3.org/2001/XMLSchema“ befindet.
Der Datentyp selbst ist im Abschnitt „complexType“ definiert. Dieser legt fest, dass das Element „device“ weitere Elemente mit dem Namen „name“ und „description“ enthalten MUSS. Ersteres ist eine Zeichenkette (Attribut „type“ ist String), letzteres wiederum ein komplexes Element (wie auch „deviceType“), welches wiederum eine Zeichenkette ist. In diesem Fall wird aber der Typ „String“ durch ein optionales Attribute „lang“ erweitert (Attribut „extension“), in welchem der Benutzer wieder eine Zeichenkette setzen kann (hier: die Sprache der Beschreibung). Wendet man dieses Schema auf Listing 3.1 an, so erfüllt der obige XML-Code exakt die in diesem Schema angegeben Kriterien. Jeder Fehler in der Schreibweise oder Anordnung der Elemente hat einen Fehler seitens einer Anwendung zur Folge, welche dieses XML Dokument gegen das Schema validiert. Diese Validierer müssen nicht selbst geschrieben werden, sondern exisitieren bereits für gängige Programmiersprachen (so auch für LabVIEW®). Für weitere Informationen zu XML Schema und deren Schlüsselwörter soll auf [BM04] verwiesen werden.

JSON Die JavaScript Object Notation (JSON) ist ein Datenaustauschformat, welches dafür entwickelt wurde, Objekte in der Skriptsprache JavaScript vollständig zu beschreiben. Damit ist es möglich, JavaScript Objekte zu serialisieren und wieder an anderer Stelle zu deserialisieren, um Daten zwischen Anwendungen austauschen zu können. JSON ist unabhängig von der Programmiersprache und Parser existieren in fast allen auf dem Markt verfügbaren. JSON kann und wird als Alternative zu XML eingesetzt, besonders weil es einen geringeren Overhead mit sich bringt. JSON ist einfacher gestaltet wie XML, wird dadurch aber besonders bei großen Strukturen oft unübersichtlicher. Der größte Vorteil ist jedoch, dass es sich (fast) ausnahmslos um gültiges JavaScript handelt, aus welchem man ohne Umwege mit Hilfe der *eval*() Funktion ein JavaScript Objekt erzeugen kann. Im Folgenden wird der oben gezeigte XML Code in einem äquivalenten JSON Code demonstriert:

```
1  "device" : {
2    "name" : "TestDevice",
3    "description" = {
4      "lang" = "en",
5      "value" = "This is a test device"
6    }
7  }
```

Relevanz für Excalibur Alle relevanten Konfigurationen werden in Excalibur über Text-Dateien realisiert. Als verwendete Auszeichnungssprache wird hierfür XML, u.a. aus dem Grund, da es bereits sehr viele Darstellungsprogramme zum komfortablen Bearbeiten von XML-Dateien auf dem Markt verfügbar sind bzw. auf einfache Weise programmiert werden können (alle benötigten Schema-Dateien sind vorhanden). Befehle an die Excalibur Server werden ebenfalls in Text-Form gesendet. Hierbei kann spezifiziert werden, ob als Format XML oder JSON verwendet werden soll. Auf diese Weise ist es möglich, einfach neue Client-Software zu programmieren, um mit Excalibur zu kommunizieren. Ebenso ist es möglich, bereits auf XML bzw. JSON beruhende Architekturen auf die Excalibur Notation zu erweitern. Für Messungen mit sehr großen Datenmengen bietet sich TDMS als Dateiformat an, u.a. auch da Excalibur in LabVIEW® programmiert wird und deshalb ein sehr einfacher und performanter Zugriff auf die TDM-Technologie möglich ist. Für alle anderen Messungen sind Textdokumente als Dokumentformat ausreichend, die mit Hilfe von Templates (s. unten) sehr genau strukturiert werden können.

3.2.3. Webtechnologien

Der Begriff „Webtechnologien“ ist sehr weitreichend und umfasst rein formal jede Technologie, die sich mit der Kommunikation in Netzwerken befasst. Während in den Anfängen der Netzwerke das Hauptaugenmerk vor allem auf Datenaustausch beruht haben, geht die moderne Technologie noch einige Schritte weiter. Es spielen nicht nur Daten- und Nachrichtenaustausch eine Rolle, sondern auch die Darstellung von Daten, Veränderung ebendieser und sogar die Kommunikation sozialer Aspekte über diese Technologien. Sehr viele sich auf dem Markt befindliche Geräte verfügen über Schnittstellen, um mit verschiedensten Netzwerken zu kommunizieren, seien es lokale Netzwerke über Wireless LAN oder das Internet. Viele Geräte verfügen über eigene netzwerkfähige Anwendungen, um Daten in ihre umliegenden Netzwerke zu publizieren. Bedient wird sich hierbei oft standardisierter Protokolle wie die HyperText Markup Language (HTML) zur Darstellung von multimedialen Inhalten in Browsern oder browserähnlichen Anwendungen. Im Gegensatz zu früheren Technologien sind diese Inhalte jedoch nicht mehr statisch, sondern werden abhängig von den verschiedensten Modellen je nach Anwendungsfall dynamisch generiert. Typischerweise wird dabei noch immer HTML verwendet, um diese dynamischen Inhalte anzuzeigen.

Client-Server Architekturen

Wie bereits erwähnt, ist man über dem Stand der reinen Darstellung von Inhalten jedoch hinaus. Vielmehr lassen moderne Netzwerk-Anwendungen die direkte Bearbeitung der Inhalte zu, welche nicht mehr ausschließlich der Darstellung, sondern vielmehr der Kommunikation mit Mensch und Maschine dienen. In diesem Bereich ist eine Fülle von neuartigen Protokollen definiert werden, um diese Kommunikation zu ermöglichen. Prinzipiell handelt es sich in den meisten Fällen um sog. Client-Server Anwendungen, bei welchen eine Komponente eine Anfrage stellt (Client) während die korrespondierende Komponente auf diese antwortet (Server). Beide Komponenten können dabei innerhalb einer Anwendung realisiert sein oder in mehreren, sie können auf demselben System liegen oder auf verschiedenen. Eine Komponente kann ebenso sowohl für gewisse Aspekte als Client und für andere als Server dienen. Eine typische Client-Server Architektur stellt beispielsweise die Kombination aus Datenbanksystem (Server) und dessen zugehörige Abfrage-Schnittstelle (Client) dar.

Aufbau der Client-Server-Anwendung Server-Anwendungen beruhen typischerweise auf Netzwerkprotokollen wie das Transmission Control Protocol (TCP) oder das User Datagram Protocol (UDP). Im Falle des TCP handelt es sich um eine Ende-zu-Ende Verbindung, welche Daten in Vollduplex überträgt, d.h. Daten können in beide Richtungen zu derselben Zeit ausgetauscht werden. Für eine Verbindung werden aus diesem Grund immer 4 Informationen benötigt: Die Adresse und den Port des lokalen Computers, wie auch die Adresse und den Port des Ziel-Computers. Die Adresse (meist IP) identifiziert dabei die Computer innerhalb eines Netzwerks, die Ports identifizieren die Anwendung innerhalb eines Computersystems. Möchte eine Client-Software Daten von einer Server-Software erhalten, baut er mit diesen Informationen eine Verbindung zur Server-Anwendung auf, überträgt seine Anfrage (Request) und wartet bis der Server seine Antwort (Response) sendet. Die Daten, welche dabei übertragen werden, sind dabei durch die Anwendung spezifiziert. Die Server-Anwendung muss dazu einen Dienst anbieten. Er erzeugt einen Endpunkt (Socket) mit einem Port und seiner eigenen Adresse, auf welchen der Client seine Anfrage stellen kann. Im Gegensatz dazu benötigt das UDP nur die Informationen des entfernten Computers, da nicht auf eine Antwort gewartet wird. Der Prozess ist also nach dem Senden der Nachricht abgeschlossen. In ersterem Fall des TCP handelt es sich um einen synchronen Prozess, denn die Client-Software wartet bis eine Antwort (Information oder Fehler) erhalten worden ist bzw. bricht die Verbindung nach Verstreichen einer gewissen Zeit ab (Timeout). Im anderen Fall handelt es sich bei UDP-Kommunikation um einen asynchronen Prozess, denn eine Antwort des Servers muss nicht zwangsläufig innerhalb einer gewissen Zeitspanne erfolgen bzw. muss überhaupt nicht auftreten. Dies bedeutet, eine Anwendung muss in der Lage sein, ohne die gewünschte Information fortfahren zu können und später auf ein mögliches Eintreffen der Information reagieren können.

TCP

UDP

Anwendungsprotokolle Wie oben beschrieben sind die übertragenen Daten nicht vom TCP- oder UDP-Protokoll spezifiziert und werden von den beteiligten Anwendungen festgelegt. Server und Client müssen dabei die übertragenen Daten „verstehen“, d.h. sich über das verwendete Datenprotokoll einig sein. Von diesen Protokollen gibt es unzählig viele. Bekannte hierunter sind beispielsweise das Simple Mail Transfer Protocol (SMTP)-, File Transfer Protocol (FTP) oder HyperText Transfer Procotol (HTTP)-Protokoll, womit Email-Nachrichten versendet, Dateien übertragen oder Webseiten abgerufen werden. Anhand dieser wenigen Beispiele erkennt man bereits, wie unterschiedlich die Art der übertragenen Daten beschaffen sein kann. Darüber hinaus gibt es zusätzlich sehr viele

proprietäre Protokolle, welche Daten auf unterschiedlichste Art und Weise serialisieren. Wenn nicht nur statische Dokumente wie Emails, Dateien oder Webseiten übertragen werden sollen (Das „T“ in den obigen Protokollen hat immer die Bedeutung „Transfer“), sondern Methoden einer Server-Anwendung auf dem entfernten System ausgeführt werden sollen, werden andere Techniken benötigt als die oben erwähnten. Eine dieser Techniken ist der sog. Remote Procedure Call (RPC). Die Server-Anwendung muss hierfür seine Methoden bei einem Registrar bekannt machen. Der Client sendet an diese Server Anwendung eine Nachricht, welche den Namen der registrierten Methode enthält und zusätzlich die dazugehörenden Parameterwerte (Lookup). Unterstützt der Server diese Methode, kann der Client diese einbinden (Binding), ausführen und so behandeln, als ob sie auf dem lokalen System ausgeführt wird. Hierfür existieren die verschiedensten Implementierungen, wobei zu den bekanntesten CORBA, Java RMI und im weitesten Sinne auch Webservices zu zählen sind. Leider sind die meisten zueinander inkompatibel.

RPC

Webservices

Webservices (auch als Webdienst bezeichnet) werden verwendet, um innerhalb von Computernetzwerken Kommunikation zwischen Maschinen zu realisieren. Damit mit Webservices gearbeitet werden kann, benötigen sie eine eindeutige Adresse in Form eines Uniform Resource Identifier (URI) im vorliegenden Netzwerk, damit ein sog. Service lokalisiert werden kann. Zusätzlich muss bekannt sein, wie mit diesem Service interagiert werden kann. Dafür wird in den meisten Fällen eine Schnittstellendefinition verwendet (z.B. Web Services Definition Language (WSDL) oder Web Application Definition Language (WADL)), in welcher alle Informationen hinsichtlich Anfrage und Antwort spezifiziert werden. Da diese WSDL immer unter dem jeweiligen URI hinterlegt sein sollte, ist es möglich, direkt mit dem jeweiligen Service zu kommunizieren, da sofort bekannt ist, welche Informationen zur fehlerfreien Kommunikation benötigt werden. Die WSDL Datei ist selbst in XML verfasst und wird somit sowohl von Mensch und Maschine einfach verstanden. Der Webservice selbst kann auf unterschiedliche Weise implementiert werden. Eine Möglichkeit ist der oben bereits beschriebene RPC. Eine weitere ist das sog. Simple Object Access Protocol (SOAP), welcher auf XML beruhend mittlerweile als industrieller Standard anerkannt wird. SOAP stellt ein Rahmenwerk (Framework) für Daten zur Verfügung, welche applikationsspezifische Informationen so beschreiben, damit unterschiedliche Anwendungen untereinander diese Daten austauschen können. Für SOAP selbst spielt das zugrunde liegende Übertragungsprotokoll keine Rolle, kann

SOAP

aber in Kombination mit Webservices die zu deklarierenden Methodenrümpfe vollständig beschreiben. Daten können so unproblematisch zwischen zwei Anwendungen auf unterschiedlichen Systemen ausgetauscht werden, da die Kommunikations-Schnittstelle vollständig beschrieben ist. Es ist dabei irrelevant, wie die Daten vor und nach der Übertragung beschaffen sind und wie sie verwendet werden, solange sie während der Übertragung WSDL und SOAP konform sind.

REST Eine weitere Möglichkeit stellen die sog. Representational State Transfer (REST) Webservices dar. Sie stellen eine Architektur dar, welche vor allem für die Maschine-to-Maschine Kommunikation genutzt wird. Im Gegensatz zu den oben erklärten Webservices, die auf Aktionen beruhen, liegt der Fokus bei REST-basierten Webservices auf Ressourcen. Das bedeutet, es wird über einen vorgegebenen Methodensatz (z.B. den HTTP-Methoden GET, POST, DELETE, PUT, usw.[FIG99]) auf die Ressourcen zugegriffen, indem sie gelesen, gesetzt, gelöscht oder verändert werden. RESTful Services müssen dabei folgende Eigenschaften erfüllen: Repräsentation (JSON, XML), Nachrichtenaustausch (HTTP), Addressierbarkeit (URI), einheitliche Schnittstelle (HTTP Methoden) und Zustandslosigkeit (Unabhängigkeit von Anfragen). Sie stellen dabei eine sehr einfache Möglichkeit dar, Ressourcen zu bearbeiten. In vielen Fällen wird dafür ausschließlich eine URI benötigt, sie können aber durch WADL Dateien genau spezifiziert werden. Für nähere Informationen ist auf [RRD07] verwiesen.
REST ist bereits zur Erklärung der Anwendungen in Abschnitt 2.3 herangezogen worden. Die meisten Anwendungen (z.B. ElWiS) und Toolkits, wie sie für Excalibur entwickelt worden sind, basiert jedoch auf SOAP aufgrund der impliziten Unterstützung von XML.

3.2.4. Sicherheit

Wählt man für die Kommunikation eine Netzwerk-basierte Strategie, werden Daten zwischen mindestens zwei verschiedenen Rechnern ausgetauscht. Der Datenaustausch erfolgt über die Netzwerkinfrastruktur des Labors, d.h. jede Station (Router, Switch, usw.) zwischen Start- und Zielrechner ist in der Lage Einfluss auf die ausgetauschten Daten zu nehmen, sei es auch nur zum Auslesen. Diese sicherheitstechnischen Fakten dürfen, vor allem für kritische Systeme, nicht vernachlässigt werden. Man kann sie durch eine geeignete Verschlüsselung der Verbindung zwischen den miteinander kommunizierenden Computersystemen und geeignete Identifikationsverfahren (z.B. digitale Signaturen) abdecken. Genaue Informationen zum Thema Kryptologie können im Anhang C.2.3 nachgelesen werden.

3.3. Analyse der Anforderungen

In diesem Kapitel wird der Lösungsansatz auf Basis seiner Anforderungen untersucht. Daraus sollen gezielt theoretische Ansätze entwickelt werden, auf welche Weise sich daraus ein Software-Konzept realisieren lässt. Es werden für die einzelnen Anforderungen logische Zusammenhänge und Strukturen definiert, aus welchen in den folgenden Kapiteln ein Programmdesign und letztendlich Quelltext generiert werden kann. Die Strukturen werden mit Hilfe der Auszeichnungssprache XML verwirklicht, um eine programmiersprachenunabhängige Darstellung zu präsentieren, welche dennoch sämtliche Zusammenhänge und Definitionen visuell kenntlich machen kann.

3.3.1. Anwendung des Lösungsansatzes auf die Anforderungen

Das wichtigste Kriterium für eine derartige Lösung ist der erwartete Nutzen für die einzelnen Zielgruppen, welche in der Lage sein sollen, auf einfache Weise Steuerungslösungen für einen zugrunde liegenden Messaufbau generieren zu können bzw. die Ausprägung einer Steuerungslösung ohne tiefere Einlernphasen verwenden zu können. Ausgehend von bisherigen und gewohnten Entwicklungen in der Laborautomatisierung müssen aus sich wiederholenden Strukturen *Aspekte* definiert werden, welche schrittweise durch dynamische und nachträglich veränderbare Prozesse ersetzt werden können. Dabei muss darauf geachtet werden, dass bei Änderungen eines der Aspekte, der Quelltext der Steuerungssoftware nicht verändert und die Gesamtsoftware nicht neu kompiliert werden muss. Das bedingt, dass

- die Datenhaltungsmodelle der Aspekte und deren Kommunikationsprotokolle einer offenen Notation gehorchen (Modularität)
- wiederkehrende, komplexe Funktionalität standardisiert bzw. ausgelagert wird (Standardisierung)
- die Ein- und Ausgabe vereinheitlicht wird, wodurch eine Übertragbarkeit gewährleistet und Insellösungen vorgebeugt wird (Datenein- und -ausgabe)
- die Ablaufsteuerung ebenso einer offenen Notation gehorcht und jederzeit, ohne Einfluss auf andere Aspekte, verändert werden kann (Ablaufsteuerung)
- Daten in beliebiger aber konfigurierbarer Formatierung in gängige Datensenken geschrieben werden können (Templating)

3.3.2. Modularität

Unter dem Begriff der Modularität soll hier die Unabhängigkeit von mehreren Teilaspekten der Steuerungssoftware verstanden werden, die ohne Neukompilierung, d.h. ausschließlich durch Konfiguration, auf geänderte Rahmenbedingungen angepasst werden können, ohne dass es einen Einfluss auf alle anderen Teilaspekte hat. Die offene Notation der Datenhaltungsmodelle und der Kommunikationsprotokolle spielen für die Modularität eine fundamentale Rolle, da Änderungen auch *nach* Installation der Software auf das Zielsystem vorgenommen werden können. Hierfür bieten sich textbasierte Auszeichnungssprachen wie XML oder JSON an, da sie von internationalen Vereinigungen standardisiert sind, d.h. eine wohl definierte Notation besitzen, und außerdem bereits unzählige Programmpakete wie Editoren, Parser usw. für sie existieren.
Zur Realisierung werden folgende Randbedingungen definiert:

1. Ein einzelnes Element der Peripherie (s. Abschnitt 3.3.3) inklusive aller Ein- und Ausgaben kann vollständig durch einen Deskriptor beschrieben werden (Vollständigkeit des Peripherie Elements).
2. Ein einzelner Sensor kann vollständig durch einen Deskriptor beschrieben werden.
3. Ein Automationsablauf kann vollständig durch einen Deskriptor beschrieben werden.
4. Die Schnittstellendefinition der Peripherie/Sensor-Anschlüsse kann vollständig durch einen Deskriptor beschrieben werden (Vollständigkeit der Schnittstellendefinition).
5. Jeder Deskriptor kann unabhängig von allen anderen Deskriptoren verändert werden.
6. Die Veränderung eines Deskriptors bedarf keiner Neukompilierung der Steuerungssoftware.

Für einen analytischen Aufbau mit n Peripherie Elementen bzw. Sensoren folgt, dass es mit $n + 2$ *Deskriptoren vollständig beschrieben werden kann*(s. Abbildung 3.1)!

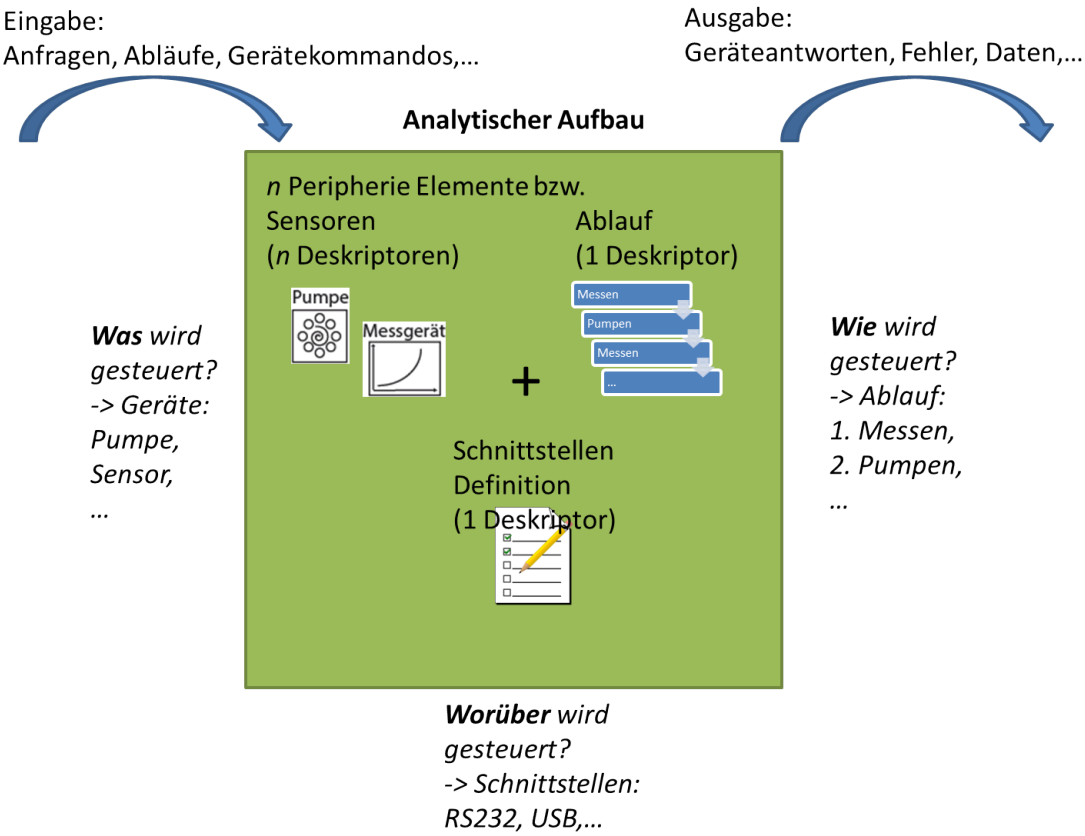


Abbildung (3.1)
Exemplarische Definition des analytischen Aufbaus mit Ein-/Ausgabe, Peripherie, Ablauf und Schnittstellendefinition inkl. der Zuordnung von Deskriptoren.

3.3.3. Peripherie Elemente

Die Peripherie besteht aus einer Menge an Laborgeräten, welche unterschiedliche Aufgaben erledigen. Jede dieser Aufgaben bzw. Aktionen, wie beispielsweise „Messen“, oder „Pumpen“, muss durch ein Gerätekommando angestoßen werden. Nachdem ein Gerätekommando vom Gerät empfangen worden ist (über einen vorhandenen Peripherie-Anschluss), führt es die verlangte Aktion aus und bestätigt bei den meisten Geräten dessen Ausführung oder den Fehlerfall mit einer Geräteantwort. Gerätekommandos und -antworten sind Datenströme und sie weisen eine Vielzahl an Kodierungen und Formate auf, weshalb sie oft nur schwer vom Menschen lesbar sind. Dies ist vor allem der Fall, wenn es sich um Binärcode handelt oder die Ein-/Ausgabe proprietär formatiert ist und

sich nur mit Hilfe des Gerätehandbuchs übersetzen lässt. Da die Möglichkeiten Kommandos zu strukturieren beliebig vielfältig sind, ist es sehr komplex, eine einheitliche Notation zu entwickeln, mit welcher sich *alle* Laborgeräte und all ihre Aktionen eindeutig beschreiben lassen.
Unter der Beschreibung von Laborgeräten versteht man hier einerseits, Gerätekommandos in menschenlesbare Zeichenfolgen zu übersetzen. Zu beachten ist hier, das Gerätekommandos meist einen oder mehrere variable Abschnitte beinhalten, welche durch entsprechend konfigurierbare Übergabeparameter zu füllen sind. Andererseits müssen auch vom Gerät zurückgegebenen Daten übersetzt und mit ihnen ein Anwendungszweck (als Ganzes bzw. mit mehreren Ausschnitten) dieser Antwort verknüpft werden.

Strukturierung von Gerätekommandos

Ein Gerätekommando ist ein Datenstrom in einer vom Gerätehersteller definierten Syntax, welcher vom Gerät interpretiert wird und daraus eine damit verbundene Aktion anstößt. Die Syntax der Kommandos ist vom Hersteller frei wählbar und unterliegt nur wenigen Einschränkungen, wodurch es relativ komplex ist, eine einheitliche Notation zu entwickeln, mit welcher alle erdenklichen Gerätekommandos beschrieben werden können. Dennoch sei angenommen, dass Gerätehersteller *reguläre* Grammatiken einsetzen, da sie sich am einfachsten strukturieren lassen (s. Abschnitt 3.2.2). Um dennoch den Versuch einer Vereinheitlichung zu wagen, können prinzipiell zwei Ansätze gewählt werden. In erstem Ansatz wird eine möglichst große Anzahl verschiedener Geräte und deren Kommandosyntax betrachtet. Die Struktur der Kommandos wird studiert, generalisiert und es wird versucht, eine gemeinsame Schnittmenge zu ermitteln, mit welcher die betrachteten Kommandos beschrieben werden können. Dies hat den Vorteil, dass die Notation sehr nah an der Kommandosyntax der Geräte orientiert ist, die Notation überschaubar und aus diesem Grund auch performant ist. Ein gravierender Nachteil hingegen ist, dass zukünftige Geräte nicht in die gefundene Notation passen können und diese erweitert oder im schlechtesten Fall revidiert werden muss. Betrachtet man hingegen den zweiten Ansatz, so stehen nicht die Geräte im Mittelpunkt, sondern die Möglichkeit, Datenströme auf eine möglichst dynamische Art und Weise zu konstruieren. Dadurch können sehr komplexe Datenfolgen, auf (fast) beliebige Art und Weise konstruiert werden. Die Geräte selbst spielen hierfür nur eine untergeordnete Rolle, da die vom Gerätehersteller gewählte Syntax auf jeden Fall eine Teilmenge der gegebenen Strukturmöglichkeiten darstellt. Der Nachteil hierfür ist, dass komplexe Datenstrukturen auch eine komplexe Notation

bedingen, wie die Datenströme zu konstruieren sind. Daraus resultieren aufwendige Algorithmen, um die Notation in das endgültige Kommando zu übersetzen. Der Hauptvorteil überwiegt jedoch, denn die Notation ist (beinahe) unabhängig von den eigentlichen zu konstruierenden Gerätekommandos, wodurch gewährleistet wird, dass auch zukünftige Geräte reibungslos und ohne Erweiterung in das gewählte Konzept eingebunden werden können.

Für das angestrebte Konzept erscheint der zweite Ansatz zielführender. Allerdings sollte nicht angestrebt werden, eine Notation zu finden, um alle erdenklichen Datenfolgen zu beschreiben. Dies würde mindestens einer kontextsensitiven Grammatik (Chomsky-Hierarchie) entsprechen. Deshalb erscheinen einige generelle Randbedingungen als sinnvoll, darüber hinaus, dass Gerätehersteller für ihre Produkte maximal eine Typ-3-Grammatik (reguläre Grammatik) implementieren würden. Sie werden folgendermaßen definiert:

- Ein Datenstrom besteht aus einem Namen und einer Menge an Tokens, welcher durch einen optionalen Terminator abgeschlossen wird. Zusätzlich können eine beliebige Anzahl Aliase für diesen definiert werden.
- Ein Token ist hier eine Kombination aus Dateneinheiten wie z.B. ASCII-Zeichen oder hexadezimale Zahlenfolgen und sie bildet eine lexikalische Grundeinheit innerhalb des Datenstroms.
- Der Terminator ist eine hexadezimale Zahlenfolge, welche an das Ende des Datenstroms gehängt wird.
- Innerhalb des Datenstroms existiert ein Haupt-Token und eine beliebige Anzahl an Neben-Tokens, welche über Platzhalter mit dem Haupt-Token verknüpft werden.
- Der Haupt-Token besteht aus einer Datenfolge mit einer Kodierung.
- Es gibt drei Arten von Neben-Tokens, alle sind optional: Adress-, Checksummen- und Value-Tokens .
- Adress-Tokens fügen in den Haupt-Token eine gerätespezifische Adresse ein, welche in der Schnittstellendefinitionsdatei (siehe Abschnitt 3.3.4) konfiguriert oder während des Programmablaufs generiert werden kann.
- Über Checksummen-Tokens können, nach üblichen Algorithmen, Checksummen des vorliegenden fertig konstruierten Haupt-Tokens berechnet werden. [1]

[1] Falls der Gerätehersteller keinen üblichen Algorithmus verwendet, kann hierfür auch ein Excalibur-

- Value-Tokens als wichtigstes Strukturelement sind relativ komplex und werden unten erklärt.

Value-Token Value-Tokens bestehen aus einem Format, einer Kodierung und einem Datentyp. Darüber hinaus kann jeder Value-Token eine beliebige Anzahl an Aliase beinhalten und ebenso eine beliebige Anzahl an Konstanten definieren (Weitere Eigenschaften werden in Tabelle B.1 aufgelistet).
Value-Tokens spiegeln dynamische Inhalte des Datenstroms wider bzw. definieren Abschnitte des Haupt-Tokens, welche in einem unterschiedlichem Format zu diesem definiert sind. Zum Beispiel kann der Haupt-Token einen binären Anteil enthalten, während ein zweiter Abschnitt in ASCII kodiert ist. Letzterer kann dazu in einen Value-Token ausgelagert werden. Value-Tokens werden benötigt, damit Benutzereingaben gesetzt, validiert und in ein Gerätekommando eingesetzt werden können, ohne für jede Möglichkeit der Ausprägung ein eigenes Kommando definieren zu müssen. Da die Position von Value-Tokens über einen Platzhalter mit einer ID im Datenstrom gekennzeichnet werden, gibt es zwei Möglichkeiten wie ein Benutzer die Werte an ein Kommando übergeben kann. Einerseits kann er alle Eingaben der Reihe nach, beginnend bei der niedrigsten ID, über eine Werteliste an den Kommandoparser übergeben. Andererseits ist es möglich, einen
Alias oder mehrere *Aliase* für einen Value-Token zu vergeben. In letzterem Fall kann der Benutzer die Werte über einen Namen referenziert an den Parser übergeben.
Als letzte Eigenschaft der Value-Tokens sei genannt, dass für wohl definierte Stati in
Konstanten einem Kommando, wie z.B. „100 % ≙ EIN“ und „0 % ≙ AUS“ eindeutige Konstanten definiert werden können, hier nämlich „EIN“ und „AUS“ (s. Beispiele unten), um die teilweisen komplexen Übergabeparameter über einfache Schlüsselwörter zu adressieren.

Konstruktion eines Gerätekommandos

Fasst man alle Randbedingungen aus dem letzten Abschnitt zusammen, kann man folgendes Schema induktiv ausformulieren:

Es existiert eine Zeichenkette $Z = \{z|z_1, \ldots\}$, mit den einzelnen Zeichen z_i, welche den fertigen Datenstrom widerspiegelt. Z wird aufgebaut aus einer Zeichenkette R, welche eine Vereinigung darstellt aus den *fixen* Zeichenketten $M = \{m|m_1, \ldots, m_{n+1}\}$ und den

Script (siehe Abschnitt 3.3.5) verwendet werden.

Platzhaltern $P = \{p|p_1, \ldots, p_n\}$ (n sei die Anzahl an Platzhaltern), die durch die *variablen* Zeichenketten Q ersetzt werden. Abgeschlossen wird R durch einen Terminator t. Zusammengefasst erscheint $R = m_1 + p_1 + \ldots + m_n + p_n + m_{n+1} + t$. Da alle M, P, t auch leere Zeichenketten sein können, dürfen sie als optional betrachtet werden (R selbst hingegen darf *nicht* leer sein[2]).
$Q = \{q|q_1, \ldots, q_n\}$ seien Zeichenketten, welche aus der Transformation eines Formats F und den Eigenschaften E auf U resultieren, wobei U Zeichenketten aus einer *externen Eingabe* darstellen. Q sind also Teilmengen von Z an den Positionen von P. Handelt es sich bei einem p um die Zeichenkette „[address]“, so wird q durch die Geräteadresse g ersetzt. Handelt es sich bei einem p um die Zeichenkette „[checksum]“, so wird q durch eine Checksumme c ersetzt. In beiden Fällen ist $f = \{0\}$. Handelt es sich bei p um eine Zeichenkette „[0,1,...]“, ist es ein Value-Token mit einem Format F. Letzteres kann dabei entweder ein regulärer Ausdruck oder ein Formatstring sein (siehe Abschnitt 3.3.3), welcher auf u angewendet die resultierende, abgeschlossene Zeichenkette ergibt, die im fertigen Datenstrom Z an der Position p zu finden ist und an das Gerät gesendet wird. Darüber hinaus können die Eigenschaften E auf U bzw. Z wirken, um die Ausgabe Q zu verändern. Können Sie nicht angewendet werden, weil Nebenbedingungen wie z.B. Datentyp oder Kodierung nicht zutreffen, kommt es zu einem Fehler (s. Liste in Tabelle B.1 im Anhang B.3).

Übersetzt man diese Anweisungen in ein XML Schema (s. Erklärung in Abschnitt 3.2.2), sieht die Beschreibung des obigen folgendermaßen aus:

Listing (3.2)

Der Token-Typ zur Abbildung von Gerätebefehlen.

```
1 <complexType name="tokenType">
2  <sequence>
3   <element name="value" maxOccurs="unbounded" minOccurs="0"
        type="tns:tokenValueType"/>
4   <element name="checksum" minOccurs="0" type=
        "checksumType"/>
5  </sequence>
6  <attribute name="token" type="string" use="required"/>
7  <attribute name="terminator" type="hexBinary" use=
8 
```

[2]Read-Only Kommandos dürfen existieren, wobei hier der Token R weggelassen werden muss.

```
      "optional"/>
 <attribute name="encoding" type="common:encodingType" use=
      "optional"/>
</complexType>
```

Listing (3.3)

Der TokenValue-Typ zur Abbildung von dynamischen Abschnitten in Gerätebefehlen.

```
<complexType name="tokenValueType">
 <sequence>
  <element name="alias" maxOccurs="unbounded" minOccurs="0"
       type="string"/>
  <element name="constant" maxOccurs="unbounded" minOccurs=
       "0" type="valueConstantType"/>
 </sequence>
 <attribute name="id" type="unsignedInt" use="required"/>
 <attribute name="format" type="string" use="required"/>
 <attribute name="type" type="tns:driverDataTypeType"/>
 <attribute name="encoding" type="tns:encodingType"/>
 <attribute name="default" type="string"/>
 <attribute name="min" type="double"/>
 <attribute name="max" type="double"/>
</complexType>
```

Die definierten XML Schemas aus Listing 3.2 und 3.3 gelten allgemein, sind aber nur in Kombination mit obiger Beschreibung sinnvoll zu verstehen. Außerdem sind in den Listings nicht alle Eigenschaften dargestellt. Eine vollständige Auflistung befindet sich im Anhang (Seite 274). Die Grammatik in EBNF-Syntax (s. Abschnitt 3.2.2) zur Synthese eines Tokens sieht folgendermaßen aus:

<token> ::= <telements> <terminator>
<telements> ::= {<telement>}
<telement> ::= <xalphas> | <tvalue> | <taddress> | <tchecksum>
<terminator> ::= <void> | <hexdigits>
<tvalue> ::= <sbracko> <value> <sbrackc>
<taddress> ::= <sbracko> <value> <sbrackc>

<tchecksum> ::= <sbracko> <checksum> <sbrackc>
<value> ::= <xalphas>
<checksum> ::= <hexdigits> | <digits>
<xalphas> ::= {<xalpha>}
<xalpha> ::= <alpha> | <digit> | <safe> | <extra> | <national> |
<punctuation> | <escape>
<escape> ::= <hexescape> | <octescape> | <binescape> | <charescape>
<hexescape> ::= \h <hexdigit>
<octescape> ::= \o <octdigit>
<binescape> ::= \b <bindigit>
<charescape> ::= \<cealpha>
<hexdigits> ::= {<hexdigit>}
<hexdigit> ::= <hex> <hex>
<hex> ::= <digit> | a | b | c | d | e | f | A | B | C | D | E | F
<octdigit> ::= <oct> <oct> <oct>
<oct> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
<bindigit> ::= <bin> <bin> <bin> <bin> <bin> <bin> <bin> <bin>
<bin> ::= 0 | 1
<cealpha> ::= a | b | f | n | r | t | v
<alpha> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s |
t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J |
K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<safe> ::= $ | - | _ | @ | . | & | €
<extra> ::= ! | * | ’ | (|) | : | ; | , | space
<national> ::= { | } | vline | \[| \] | \ | ^ | ~
<punctuation> ::= < | >
<sbracko> ::= [
<sbrackc> ::=]
<void> ::=

Mit Hilfe von Beispielen soll die Anwendung von Tokens verdeutlicht werden.

Beispiel 1:
Man betrachte das Kommando einer peristaltischen Laborpumpe. Über ein Kommando

kann diese Pumpe ein- und ausgeschaltet und zusätzlich die Pumpgeschwindigkeit über einen bestimmten Bereich hinweg variiert werden. Die Syntax sei folgendermaßen definiert:
ApXX mit A als Adressparameter zwischen a und z, p als Gerätekommando, um den Status „Pumpen“ zu (de)aktivieren und XX als Pumpgeschwindigkeit zwischen 0 und 99 $\mu L/min$. Abgeschlossen wird das Kommando über einen Zeilenumbruch mit Wagenrücklauf (Carriage Return + Line Feed). D.h. das Kommando *ap50* würde die Pumpe mit der Adresse a anschalten mit einer Pumpgeschwindigkeit von 50 $\mu L/min$.
Möchte man diese Syntax mit obigem Schema ausdrücken, folgt daraus folgendes Muster:

Haupt-Token	
Zeichenkette R	[address]p[0]
Terminator t	0D0A
Value-Token mit Benutzereingabe U	
ID	0
Format f	%2d
Max	99
Min	0

Beispiel 2:
Man betrachte das Setzen der Integrationszeit in einem gedachten Spektrometer. Die Syntax sei folgendermaßen definiert:
0AXXXXX mit *0A* als hexadezimales Gerätekommando zum Setzen der Integrationszeit und *XXXXX* als Integrationszeit in Millisekunden in ASCII Kodierung:

Haupt-Token	
Zeichenkette R	0A[0]
Kodierung	HEX
Value-Token mit Benutzereingabe U	
ID	0
Format f	%5s
Kodierung	ASCII

Aus dieser Auflistung an Eigenschaften und Zeichenketten konstruiert man nun, sich orientierend am XML Schema, ein Kommando in XML Syntax (Exemplarisch an obigen

Beispielen):

Beispiel 1:

Listing (3.4)

XML Beschreibung für einen Token zum Anschalten der Pumpe aus obigem Beispiel 1

```
<token token="[address]p[0]" terminator="0D0A">
 <value id="0" format="%2d" min="0" max="99">
  <alias>speed</alias>
  <constant name="OFF">0</constant>
  <constant name="MAX">99</constant>
 </value>
</token>
```

Beispiel 2:

Listing (3.5)

XML Beschreibung für einen Token zum Setzen der Integrationszeit aus obigem Beispiel 2

```
<token token="0A[0]" encoding="HEX">
 <value id="0" format="%5s" encoding="ASCII"/>
</token>
```

Betrachtet man sich obige Definition, sind beliebige reguläre Datenströme konstruierbar, welche dynamisch mit Benutzereingaben gefüllt werden können. Aus diese Weise ist es möglich, jede Art von Gerätekommando zu generieren und somit jede Art von Herstellersyntax abzubilden.

Strukturierung von Geräteantworten

Eine Geräteantwort ist ein Datenstrom, welcher vom Gerät gesendet wird, nachdem eine Aktion ausgeführt worden ist. Prinzipiell verhält es sich mit diesen ähnlich wie mit den Gerätekommandos und sie liegen in einer vom Gerätehersteller definierten Syntax vor, die natürlich ebenso vielfältig sein kann wie die der Gerätekommandos (Annahme: reguläre Typ-3-Grammatik). Anders als bei Letzteren müssen Antworten nicht konstru-

iert, sondern zerlegt werden, d.h. der Antwort als ganzes bzw. gewissen Teilabschnitten davon, müssen Bedeutungen zugewiesen werden. Um diese Zuweisung bewerkstelligen zu können, muss in erster Linie verstanden werden, welche Art von Antworten von den Geräten erwartet werden kann. Diese „Arten“ werden im Folgenden als Antwortklassen bezeichnet, von welchen der wichtigste der **Kommandostatus** ist und von den meisten Geräten für alle Kommandos implementiert wird. Darunter versteht man eine Geräteantwort, welche dem Aufrufer mitteilt, ob das Kommando eine Aktion ausgelöst hat (Status: „OK“) oder ein Fehler aufgetreten ist (Status: „KO“ mit Fehlermeldung). Weitere verbreitete Antwortklassen sind **Konfigurationen**, bei welcher im Gerät gesetzte Konfigurationsparameter zurückgegeben werden, oder die Klasse der **Daten**, bei welcher Messgeräte Messwert(e) zurückgeben.

Antwortklassen

Status

Konfigurationen

Daten

Eine wichtige Erkenntnis ist, dass ein Datenstrom als Antwort auf ein Gerätekommando folgt. Da diese ähnlich strukturiert sind wie Gerätekommandos, ist es naheliegend, den Antwort-Parser und seine Notation möglichst ähnlich zu strukturieren wie jene des Kommando-Parsers. Eine Antwort kann demnach als Token angesehen werden, welcher oben definierte Eigenschaften erfüllen soll. Einziger Unterschied ist, dass die Zeichenkette Z bereits vorliegt und mit Hilfe von R auf die einzelnen Elemente M und P Rückschlüsse gezogen werden soll[3]. Anschließend können den Elementen m_i und p_i Antwortklassen zugewiesen werden.

Es sind wiederum Randbedingungen für den Datenstrom „Geräteantwort“ zu definieren:

- Ein Datenstrom kann durch eine Menge an Tokens beschrieben und durch einen optionalen Terminator abgeschlossen werden.
- Ein Token ist eine Kombination aus Dateneinheiten wie z.B. ASCII-Zeichen oder hexadezimale Zahlenfolgen und sie bildet eine lexikale Grundeinheit innerhalb des Datenstroms.
- Jeder Token muss durch entweder eine Länge, ein wohl definiertes Format spezifiziert, oder er muss durch einen Terminator abgeschlossen sein.
- Innerhalb des Datenstroms existiert ein Haupt-Token und eine beliebige Anzahl an Neben-Tokens, welche über Platzhalter mit dem Haupt-Token verknüpft werden.
- Der Haupt-Token besteht aus einer Datenfolge mit einer Kodierung (Std: ASCII).

[3] Z ist die Zeichenkette, welche das Gerät selbst verstehen und interpretieren kann. R ist die Zeichenkette, aus welcher Z konstruiert werden kann und in welche die statischen Abschnitte M und die variablen Abschnitte P eingesetzt werden.

- Es gibt zwei Arten von Neben-Tokens, alle sind optional: Checksummen- und Value-Tokens.
- Über Checksummen-Tokens können, nach üblichen Algorithmen, Checksummen des vorliegenden fertig konstruierten Haupt-Tokens berechnet werden.[4]
- Der Value-Token als zweiter Neben-Token ist oben erklärt worden.
- Es darf mehr als ein Haupt-Token definiert werden mit jeweils individuellen Neben-Tokens.

Konstruktion einer Geräteantwort

Es existiert eine Zeichenkette $A = \{a|a_1, \ldots\}$, mit den einzelnen Zeichen a_i, welche die gesendete Geräteantwort widerspiegelt. A wird zerlegt mit Hilfe einer Zeichenkette R, welche eine Vereinigung darstellt aus den *fixen* Zeichenketten $M = \{m|m_1, \ldots, m_{n+1}\}$ und den Platzhaltern $P = \{p|p_1, \ldots, p_n\}$ (n sei die Anzahl an Platzhaltern), die durch die *variablen* Zeichenketten Q ersetzt sind. Abgeschlossen wird R durch einem Terminator t. Zusammengefasst erscheint $R = m_1 + p_1 + \ldots + m_n + p_n + m_{n+1} + t$. Da alle M, P, t auch leere Zeichenketten sein können, dürfen sie als optional betrachtet werden (R selbst hingegen darf *nicht* leer sein). Da auf ein Gerätekommando mehrere verschiedene Antworten denkbar sind (z.B. Daten oder Fehler), existiert eine Menge Y, welche eine Vereinigung aus den Mengen R_i darstellt, also $Y = \sum_{i=1}^{n} R_i$, wobei für ein vorliegendes A nur ein einziges R_i zutreffen darf.
$Q = \{q|q_1, \ldots, q_n\}$ seien Zeichenketten, welche Teilmengen in A an den Positionen P darstellen. Über die Formate F und den Eigenschaften E können die Teilmengen $V \subseteq A$ identifiziert und extrahiert werden, wenn die *Länge* der V eindeutig über einen Formatstring (siehe Abschnitt 3.3.3) oder eine explizite Längenangabe im Format F spezifiziert werden. Ist letzteres nicht der Fall, kann die Geräteantwort A nur iterativ mit den einzelnen transformierten V, also auf welche F und E gewirkt haben, verglichen werden, um den Abschluss jener Zeichenkette zu finden. Stimmen die einzelnen Q nicht mit den transformierten V überein, kommt es zu einem Fehler.
Die einzelnen Eigenschaften sind in Tabelle B.1 gezeigt.

Übersetzt man diese Anweisungen in ein XML Schema, sieht die Beschreibung des obigen

[4] Falls der Gerätehersteller keinen üblichen Algorithmus verwendet, kann hierfür auch ein Excalibur-Script (siehe Abschnitt 3.3.5) verwendet werden.

folgendermaßen aus:

Listing (3.6)

Der Response-Typ zur Abbildung von Geräteantworten in Excalibur.

```
<complexType name="responseType">
 <sequence>
  <element name="value" maxOccurs="unbounded" minOccurs="0"
       type="tns:responseValueType"/>
  <element name="checksum" minOccurs="0" type=
       "checksumType"/>
 </sequence>
 <attribute name="token" type="string" use="required"/>
 <attribute name="terminator" type="hexBinary" use=
      "optional"/>
 <attribute name="encoding" type="common:encodingType" use=
      "optional"/>
</complexType>
```

Listing (3.7)

Der ResponseValue-Typ zur Abbildung von dynamischen Abschnitten in Geräteantworten.

```
<complexType name="responseValueType">
 <sequence>
  <element name="lookup" type="tns:responseValueLookupType"
       maxOccurs="unbounded" minOccurs= "0"/>
 </sequence>
 <attribute name="id" type="unsignedInt" use="required"/>
 <attribute name="format" type="string" use="required"/>
 <attribute name="type" type="tns:driverDataTypeType"/>
 <attribute name="encoding" type="tns:encodingType"/>
 <attribute name="channel" type="string"/>
</complexType>
```

Die Ähnlichkeit zwischen den Listings 3.2 und 3.6 bzw. 3.3 und 3.7 ist deutlich erkennbar. Die Grammatik zur Synthese einer Geräteantwort entspricht der von Tokens, außer

dass das Nichtterminalsymbol „token“ durch „response“ ersetzt werden muss.[5] Auf einige Eigenschaften muss aber näher eingegangen werden.

Geräteantworten und Eigenschaften Während bei Gerätebefehlen eine Zeichenkette konstruiert wird und dafür die Eigenschaften, d.h. die Benutzereingaben, zur Laufzeit festgelegt werden, verhalten sich Geräteantworten in dieser Hinsicht anders, als dass die Eigenschaften bereits vor Laufzeit bekannt sein müssen. Wie bereits erwähnt, handelt es sich bei diesen um Antwortklassen. In Excalibur werden (vorerst) 4 Klassen unterschieden: Gerätestati, Fehler, Konfigurationen und Daten.

Gerätestatus

Unter dem Gerätestatus wird einerseits die Ausführung bzw. Nicht-Ausführung eines Kommandos verstanden, andererseits können im Gerät selbst Gerätestati definiert werden (und unabhängig davon auch in Excalibur). Betrachtet man den ersten Fall, so sendet das Gerät eine „Erfolgsmeldung“, wie z.B. *OK*, * oder eine Wiederholung des gesendeten Kommandos, welcher mit einem definierten Token verglichen und im Fall einer Übereinstimmung anschließend meist verworfen oder möglicherweise geloggt wird. Bei Betrachtung des letzteren Falls, wird der Sachverhalt ein wenig komplizierter, da das Gerät eine Statusmeldung sendet und diese interpretiert werden muss. Die Statusmeldung(en) kann jetzt ebenso wie die Erfolgsmeldung geloggt werden oder mit Hilfe einer „Lookup“-Tabelle (siehe Listing 3.7) in eine Liste an Meldungen übersetzt werden, die anschließend weiterverarbeitet werden kann. Zudem befinden sich Geräte in einem bestimmten Gerätestatus (s. Abschnitt 3.3.3), welcher durch die Ausführung eines Kommandos geändert werden kann. Nach der Ausführung eines Kommandos wird dieser Status in einer Gerätevariablen *immer* neu gesetzt, wenn auch mit demselben Wert, und kann während eines Programmablaufs (s. Abschnitt 3.3.6) abgerufen und auf diesen reagiert werden.

Fehler

Fehler-Stati verhalten sich ähnlich wie Gerätestati. Es gibt vielfältige Möglichkeiten, warum Geräte Fehler produzieren können. Beispielsweise kann sich das Gerät in einem falschen Zustand befinden (s. Abschnitt 3.3.3), die aufzurufende Funktionalität könnte blockiert sein, das Gerät verarbeitet gerade noch ein anderes Kommando, oder es tritt irgendeine Art von Timeout auf. In jedem dieser Fälle wird das Gerät die Ausführung eines Kommandos mit einem Fehler quittieren, welche zum einen eine wohl definierte Zeichenkette sein kein oder zum anderen eine komplexe Datenfolge, welche interpretiert und übersetzt werden muss. Abhängig von der Schwere des Fehlers (s. Tabelle 3.1) kann

[5] Geräteantworten werden nicht synthetisiert, sondern geparst. Das Ergebnis des Parsens muss die synthetisierte „response“ ergeben.

Tabelle (3.1)
Übersicht der Schweregrade von Fehlern und deren Auswirkungen.

Schweregrad	Auswirkung
None	Kein Fehler ist aufgetreten
Warning	Es ist eine Warnung aufgetreten, welche als Fehlermeldung zurückgegeben wird, wenn es durch die Konfiguration so festgelegt wird. Ansonsten verhält sich die „Warning“ wie „None“
Exception	Es ist eine Ausnahme aufgetreten und es wird ein Fehlermeldung generiert und zurückgegeben. Außerdem erfolgt ein Logeintrag
Error	Es ist ein Fehler aufgetreten, der Programmablauf wird sofort beendet
Critical	Es ist ein kritischer Fehler aufgetreten. Die Excalibur Server wird sofort beendet, wenn es nicht durch die Konfiguration verhindert wird. Ansonsten verhält sich „Critical“ wie ein „Error“

dieser einfach ausgegeben und geloggt werden oder es sind gravierendere Folgen für den Programmablauf des Excalibur Servers zu erwarten, z.B. eine direkte Beendigung der Hauptanwendung bei kritischen Fehlern.

Das Verhalten bei einem Fehler ist durch Tabelle 3.1 dargelegt. Die Antwort kann jetzt definiert werden, so dass für gewisse Tokens ganz bestimmte Schweregrade resultieren. Zusätzlich können die Fehlermeldungen frei definiert werden. Beide Eigenschaften werden über das in Listing 3.7 gezeigte „Lookup“ realisiert, welches über die Eigenschaften „Description“ und „Severity“ verfügt und für die gerade besprochenen Fälle gesetzt werden können. Der Aufbau eines Lookup Eintrags ist in Listing 3.3.3 gezeigt.

```
1 <complexType name="responseValueLookupType">
2  <sequence>
3   <element name="description" type="common:descriptionType"
        maxOccurs="unbounded" minOccurs="0"/>
4  </sequence>
5  <attribute name="value" type="string" use="required"/>
6  <attribute name="encoding" type="common:encodingType" use="
       optional" default="ASCII"/>
7  <attribute name="severity" type="common:errorLevelType" use
       ="optional" default="none"/>
```

```
8 </complexType>
```

Werden als Geräteantwort Daten erwartet, muss auf die erwarteten Datenströme anders reagiert werden. In diesem Fall sollen keine Meldungen ausgegeben und Gerätestati gesetzt werden, sondern es sollen Messdaten empfangen, verarbeitet, weitergegeben und möglicherweise gespeichert werden. Da auch die Formate von Messdaten vielfältig sind, wird das beschriebene Token-Konzept auch auf jene angewandt, wobei hier die Eigenschaft „Datentyp“ eine besondere Rolle spielt. Messdaten bestehen meistens aus Zahlen bzw. Zahlenlisten, die aus dem gesendeten Datenstrom extrahiert und in passende Variablen oder Arrays eingefügt und weitergereicht werden sollen. Dieses Weitergeben von Daten wird über sog. Kanäle realisiert und kann durch die Eigenschaft „channel“ (s. Listing 3.7) spezifiziert werden. Das Thema Daten und ihre Verarbeitung wird in Abschnitt 3.3.5 näher besprochen. *Daten*

Konfigurations-Anworten sind ähnlich zu Daten-Antworten und werden entsprechend ähnlich behandelt. Sie spiegeln momentan gesetzte Einstellungen im Gerät wider und deren Rückgabe zielt einzig darauf ab, dem Benutzer mitzuteilen, welche Werte im Moment gesetzt sind. Auch dies wird über Kanäle realisiert, auf welchen explizit gelauscht werden kann (s. Abschnitt 3.3.5). *Konfigurationen*

Obwohl es sich beim Senden von Kommandos und den vom Gerät zurückgesendeten Antworten um unterschiedliche Prozesse handelt, kann gezeigt werden, dass sich beide durch einen ähnlichen Ansatz, nämlich den der Tokens, beschreiben lassen. Mit Hilfe des Token-Konzepts ist es möglich mit einem Textdokument, dem Deskriptor für Peripherie-Elemente, die Herstellersyntax für Geräte in menschenlesbarer Form abzubilden und sie auf variable Weise vom eigentlichen Gerät abzukoppeln. In den folgenden Kapiteln wird dieses Konzept erweitert und somit eine vollständige Auflösung der Abhängigkeit zwischen den Aspekten „Steuerung“ und den Geräten selbst zu realisieren.

Format Syntax von Tokens

Bei dem Token-Format handelt es sich, wie schon oben angedeutet, um eine Zeichenkette, die genau spezifiziert, wie ein gewisser Datenstrom zum/vom Gerät auszusehen hat. Dies gilt für beide Richtungen der Kommunikation mit Geräten, sowohl für den Befehl als auch die Geräteantwort. Um Datenströme, insbesondere Zeichenketten zu konstruieren, existieren auf dem Markt viele syntaktische Sprachen. Zwei davon werden für Excalibur eingesetzt, die Formatstring Syntax und reguläre Ausdrücke, die beide im

Folgenden näher erläutert werden. Da beide einen zugrundeliegenden Vergleich benötigen, um einsetzbar zu sein, gilt es für Gerätekommandos und -antworten dahingehend zu unterscheiden, dass im ersteren Fall das Format auf Benutzereingaben anzuwenden ist und im letzeren Fall auf die vom Gerät stammenden Antwort selbst.

Formatstring Syntax Die Formatstring Syntax ist ursprünglich von der Programmiersprache C für die dort implementierte *printf* Funktion eingeführt worden. Sie beschreibt eine Zeichenkette aus Strings mit Platzhaltern, die mit einem Prozent-Symbol beginnen und in welche Variablen eingesetzt werden. Zwischen dem Prozentzeichen und dem Formatierungsbefehl im Einbuchstabenformat lässt sich die Ausgabe noch genauer über weitere Optionen steuern. Anhand eines Beispiels lässt sich das auf sehr einfache Weise erläutern:

```
1 Spezifikation: printf("Ich moechte %s %d km joggen. Das sind
       %6.3f Meilen.", "gerne", 22, 13.67054);
2
3 Ausgabe: Ich moechte gerne 22 km joggen. Das sind 13.671
      Meilen.
4
5 Spezifikation: printf("Ich moechte %s %d km joggen. Das sind
       %6.3f Meilen.", "keine", 22.00, 13.670539986334001);
6
7 Ausgabe: Ich moechte keine 22 km joggen. Das sind 13.671
      Meilen.
```

Zur Erläuterung: Der Platzhalter %s erwartet eine Zeichenkette, %d eine Zahl, welche in einen Integer konvertiert wird und letztendlich %f formatiert eine Zahl in einen Gleitkommawert, welcher mit der Angabe 6.3 in einen Wert mit maximaler Breite von 6 Zeichen und 3 Nachkommastellen formatiert wird. Zur genaueren Defintion von printf sei auf einschlägige Fachliteratur verwiesen[KR83].
In LabVIEW® wird diese Notation zur Formatierung von Zeichenketten ebenfalls verwendet, die Möglichkeiten hierfür sind aber sehr viel größer. Beispielsweise lassen sich damit auch Zeitstempel formatieren und Einheiten verwenden. Die Vielfältigkeit der Möglichkeiten lässt sich hier nicht vollständig wiedergeben, weshalb auf Hilfe von LabVIEW® Hilfe verwiesen wird oder auf die Webseite von National Instruments[6].

[6] http://zone.ni.com/reference/en-XX/help/371361J-01/lvconcepts/format_specifier_syntax/

Diese Syntax wird direkt von Excalibur mit Hilfe der Eigenschaft „Format“ unterstützt. Pro Value-Token kann allerdings nur ein Prozent-Platzhalter verwendet werden, da pro Value-Token auch nur eine Benutzereingabe bzw. Teilantwort erwartet wird. Ansonsten sind alle Möglichkeiten hinsichtlich der Formatierung gegeben, wie sie auch von LabVIEW® angeboten werden.

Reguläre Ausdrücke Reguläre Ausdrücke beschreiben mit Hilfe von wohl definierten syntaktischen Regeln eine Menge an Zeichenketten. Der Begriff selbst geht auf den Mathematiker Stephen Kleene zurück. Sie beschreiben eine Familie von formalen Sprachen innerhalb der theoretischen Informatik. Sie werden verwendet, um nach den in ihnen definierter syntaktischen Regeln, innerhalb von Zeichenketten ein definiertes Suchmuster wiederzufinden. Es gibt verschiedene Implementierungen für reguläre Ausdrücke, wobei die häufigste, so auch in LabVIEW®, die sog. *Perl Compatible Regular Expressions* (PCRE) verwendet, welche ursprünglich für die Programmiersprache Perl entwickelt worden sind. Eine genaue Beschreibung der Syntax würde den Rahmen dieser Arbeit sprengen, weshalb auf einschlägige Fachliteratur verwiesen werden soll[HU94]. Dennoch soll anhand von wenigen Beispielen die Funktionsweise erläutert werden.

```
1  Eingabe: Ich moechte gerne 22 km joggen. Das sind 13.671
       Meilen.
2
3  Reg-Exp: .*
4  Ausgabe: Ich moechte gerne 22 km joggen. Das sind 13.671
       Meilen.
5
6  Reg-Exp: \d+\.\d{3}
7  Ausgabe: 13.671
8
9  Reg-Exp: \d+.*(sind)(nicht)?
10 Ausgabe: 22 km joggen. Das sind
```

In diesen Beispielen soll einige Konzepte verdeutlicht werden:

- Ein . steht repräsentativ für jedes beliebige Zeichen
- Ein $\backslash d$ steht repräsentativ für jede Zahl zwischen 0 und 9

- Ein \. steht repräsentativ für einen Punkt .
- Die Zeichen *, {x} und ? sind sog. Quantoren und wirken jeweils auf die Zeichenkette, Anweisung bzw. Gruppe davor. Dabei steht * repräsentativ für eine beliebige Anzahl an Vorkommnissen, + für ein oder mehrere Vorkommnissen und {x} für genau x Vorkommnisse.
- Alles innerhalb von (x) wird alle Gruppe gedeutet, auf welche Quantoren wirken können

Die obige Liste stellt bei weitem nicht den kompletten Satz an verfügbare grammatikalischen Elementen für reguläre Ausdrücke dar, allerdings kann anhand der obigen Beispiele bereits die Mächtigkeit erahnt werden, um Zeichenketten in anderen zu suchen und zu extrahieren.

Gerätestatus

Der Gerätestatus ist eine wichtige Eigenschaft. Es ist wichtig für den Benutzer zu wissen, welche Aktion im Moment im Gerät aktiv ist. Zusätzlich ist es eine wichtige Information, ob ein Aktion fehlgeschlagen ist und Fehlermaßnahmen ergriffen werden müssen oder ob eine bestimmte Aktion im aktuellen Status überhaupt durchgeführt werden kann. Ein Gerät, welches sich beispielsweise im Status „Standby“ befindet, muss nicht mehr gestoppt, deaktiviert oder auch initialisiert werden. Excalibur unterstützt explizit Gerätestati und stellt Konfigurationsmöglichkeiten für Gerätekommandos und Statuswechsel zur Verfügung.

Gerätestatus

Aktueller Status und Statuswechsel In Excalibur können im Treiber (siehe unten) beliebig viele Stati definiert und anschließend den Gerätekommandos zugeordnet werden. Dort kann anschließend bestimmt werden, innerhalb welcher Stati jenes Kommando aufgerufen werden darf. Mit Hilfe des Kommandos *getState* kann der aktuelle Status des Gerätes jederzeit abgefragt werden. Falls ein Gerätehersteller eigene Stati, sog. „Betriebsmodi“ definiert, können diese über einen Mapping Mechanismus mit den Excalibur-Stati verknüpft werden. In diesem Fall sollte der Hersteller eine Kommandosequenz definieren, über welche der Betriebsmodus seines Gerätes abgefragt werden kann. So ist es möglich den Status durch Abfrage des Gerätes immer aktuell zu halten. Probleme treten auf, wenn der Hersteller keine Kommandosequenz zur Abfrage des Betriebsmodus definiert

(oder keine Betriebsmodi implementiert hat) und deshalb keine Informationen über den aktuellen Status vorliegen. Dies kann beispielsweise vorkommen, wenn bei einer Geräteaktion ein Fehler aufgetreten ist oder ein Gerät manuell vor Ort bedient worden ist. Da in diesem Fall eine Schätzung des aktuellen Status nicht immer eindeutig möglich ist, wird die Definition genau eines *Idle-State*-Kommandos im Treiber benötigt, welches über das Gerät in einen „Standardzustand“ versetzt (*setIdle*). Jenes Kommando sollte entsprechend sinnvoll gewählt werden, um das vorliegende Gerät tatsächlich in einen Betriebsmodus zu versetzen, der als „Idle“ bezeichnet werden kann.

Idle

Befindet sich das Gerät in einem gewissen Status und es wird ein (in diesem Betriebszustand erlaubtes) Kommando ausgeführt, so kann der aktuelle Status erhalten bleiben oder mit diesem Kommando ein Statuswechsel einhergehen. Für jedes Kommando ist dafür genau ein Zielstatus definierbar. Statuswechsel sind sinnvoll, wenn das Gerät dadurch in einen anderen Betriebszustand versetzt wird (z.B. Pumpe „aus“ in Pumpe „an“), welcher eine andere Menge an Kommandos als zuvor gestattet bzw. eine andere Menge an Kommandos innerhalb des neuen Status verboten ist. Auf diese Weise sollen beispielsweise Beschädigungen an Geräten oder unmögliche Aktionen innerhalb des aktuellen Betriebszustand verhindert werden.

Statuswechsel

Konstruktion des Gerätetreibers

Mit obigen Definitionen ist es möglich, Geräte-Kommandos und ihre Antworten zu definieren. Mit den Stati ist auch ein Prozess erläutert worden, um die Kommandos nur in sinnvollen Betriebszuständen zu erlauben. Es kann nun ein Schritt weiter gegangen und eine Struktur definiert werden, um Laborgeräte und ihre Funktionalität eindeutig zu beschreiben.

Kommandos Das Token-Konzept kapselt die Möglichkeit sowohl Gerätekommandos als auch -antworten zu strukturieren. Allein mit diesem können gesendete und empfangene Datenströme des Gerätes konstruiert bzw. interpretiert werden. Da es sich um eine Notation handelt, welche mit UNICODE-Zeichen ausgedrückt werden kann, bietet es sich an, ebenfalls eine textbasierte Struktur zu entwickeln, um das Token-Konzept zu kapseln. Diese Struktur soll im Folgenden als *Kommandotyp* oder *Excalibur-Kommando* bezeichnet werden. Sein XML-Schema mit den wichtigsten Strukturelementen sieht folgendermaßen aus (Vollständiges Schema befindet sich in Abschnitt B.3):

```
<complexType name="commandType">
 <sequence>
  <element name="description" type="tns:descriptionType"
       maxOccurs="unbounded" minOccurs="0"/>
  <element name="visibility" type="tns:visibilityType"
       minOccurs="0" default="public"/>
  <element name="decimalSeparator" type="tns:delimiterType"
       minOccurs="0"/>
  <element name="token" type="tns:tokenType" minOccurs="0"/>
  <element name="response" type="tns:responseType"
       minOccurs="0" maxOccurs="unbounded" />
  <element name="interrupt" type="tns:interruptType"
       minOccurs="0"/>
  <element name="postWriteTime" minOccurs="0"/>
  <element name="postReadTime" minOccurs="0"/>
  <element name="state" type="tns:commandStateType"
       minOccurs="0"/>
  <element name="alias" type="tns:aliasType" maxOccurs=
       "unbounded" minOccurs="0"/>
 </sequence>
 <attribute name="id" type="string" use="required"/>
</complexType>
```

Außer den Tokens umfasst der Kommandotyp weitere Strukturelemente, die für eine sinnvolle Anwendung wichtig sind. Darunter fallen die *id*, unter welcher ein Name für das Kommando definiert werden kann, und außerdem die selbsterklärenden Eigenschaften *description* und *alias*. Auf diese soll hier jedoch nicht eingegangen werden, sondern erst im Anhang in Abschnitt B.3. Dennoch stechen die oben erklärten wichtigsten Eigenschaft deutlich heraus, vor allem der *Token*, welcher die Definition des Gerätekommandos darstellt und die *Response*, welches die Anweisungen für das Parsen der Geräteantworten definiert.

Kompositionen Während sehr viele Gerätebefehle nach der einfachen Struktur „Anfrage-Antwort“ funktionieren, gibt es dennoch einige Aktionen, welche erst über mehrere

nacheinander folgende Befehle vollständig ausgeführt werden. So wird beispielsweise die Aktion „Messen“ bei den meisten Spektrometern über die Befehlsfolge „Starte Messung“, „Warte Integrationszeit ab“, „Sende Daten“, realisiert. Derartige komplexere Befehlsfolgen können einerseits manuell über den Aufruf der richtigen Excalibur-Kommandos in der richtigen Reihenfolge in der Ablaufsequenz konstruiert werden oder sie werden bereits als eigenständige Kommandos, den sog. Kompositionen realisiert. Sie sind im einfachsten Fall eine Liste an Kommandos, welche in der angegebenen Reihenfolge ausgeführt werden:

```
<complexType name="compositeCommandType">
 <sequence>
  <element name="description" type="common:descriptionType"
       maxOccurs="unbounded" minOccurs="0"/>
  <element name="command" type="tns: commandReferenceType"
       maxOccurs="unbounded" minOccurs="0"/>
  <element name="script" type="program:scriptType"
       maxOccurs="unbounded" minOccurs="0"/>
  <element name="continuity" type="tns:continuityType"
       minOccurs="0"/>
  <element name="alias" type="tns:aliasType" maxOccurs=
       "unbounded" minOccurs="0"/>
 </sequence>
 <attribute name="id" type="string" use="required"/>
</complexType>
```

```
<complexType name="commandReferenceType">
 <sequence>
  <element name="redirect" type="common:redirectType"
       minOccurs="0" maxOccurs="unbounded"/>
 </sequence>
 <attribute name="ref" type="string" use="required"/>
 <attribute name="target" type="string" use="optional" />
 <attribute name="channel" type="string" use="optional" />
</complexType
```

Wie auch im einfachen Kommando finden sich hier die Eigenschaften *id*, *description* oder *alias* wieder, welche ebenso eingesetzt werden wie dort angegeben. Mit der Eigenschaft *command* und dessen Attribut *ref* wird über die Kommandotyp-ID auf das referenzierte Kommando verwiesen. Verwendet werden kann jedes Excalibur-Kommando, vor allem solche, welche *private visibility* (s. *commandType*) angegeben haben und dadurch nicht eigenständig ausgeführt werden können. Wichtig hierbei ist, das sämtliche Benutzereingaben des einfachen Kommandos natürlich auch für die Komposition angegeben werden müssen. Das bedeutet z.B. für eine Komposition aus zwei Excalibur-Kommandos, welche jeweils eine Benutzereingabe erfordern, dass für jene Komposition zwei Benutzereingaben angegeben werden müssen. Um dies zu vereinfachen, soll auf eine Eigenheit des bereits oben eingeführten *Alias* näher eingegangen werden, den sog. *Value-Based Types* (s. Listing B.1 in Anhang B.1), von welchen der *aliasType* ableitet.

Jedem Alias können vordefinierte Werte mitgegeben werden, welche anstelle der Benutzereingabe in das Kommando eingesetzt werden. Die Eigenschaft *value* des *valueBasedTypes* nimmt diesen Wert entgegen, während das Attribut *index* die ID des Value-Tokens (als Zahl) erwartet. Bei Kompositionen wird entsprechend eine Punktnotation verwendet, z.B. soll für eine beliebige Komposition bestehend aus zwei Kommandos mit jeweils zwei Benutzereingaben die zweite Benutzereingabe des ersten Kommandos durch den festen Wert „fest“ ersetzt werden; in diesem Fall wird der *value* auf „fest“ gesetzt und der *index* auf 0.1 (Zählung beginnt bei 0). Auf diese Weise ist es möglich, die Anzahl an Benutzereingaben schon innerhalb des Kommandotyps für bestimmte Spezialfälle zu reduzieren und über den Alias einen Namen zuzuweisen. Sollen alle Benutzereingaben hingegen variabel sein und auch angegeben werden müssen, so wird das Kommando/die Komposition einfach über die ID angesprochen, welche alle Eingaben erwartet[7].
Zusätzlich zum schlichten Abarbeiten von Kommandos, kann auf deren Ausgaben Einfluss genommen werden. Dafür können Skripte über die Eigenschaft *script* definiert werden, die mit den Ausgabekanälen und Zielvariablen der einzelnen Kommandos verbunden werden. Auf Skripte und andere mathematische Operationen bzw. deren Eigenschaften *redirect*, *channel* und *target* wird näher im Abschnitt 3.3.5 eingegangen.

Treiber Werden alle möglichen Aktionen eines Gerätes jeweils in ein Excalibur-Kommando bzw. Kompositionen verpackt, kann man die Menge an diesen in die nächst größere Struktur kapseln. Jene wird im Folgenden als Excalibur-Treiber oder einfach

[7] Über entsprechend gesetzte *Default*-Werte kann ein ähnliches Verhalten auch für das Excalibur-Kommando per se erreicht werden

Treiber bezeichnet. Außer einer Liste an Kommandotypen und Kompositionen, enthält dieser noch einige generelle Eigenschaften, welche global für das Gerät selbst gelten. Da der Treiber selbst nur eine Menge an oben genannten Strukturelementen darstellt, soll hier auf eine detaillierte Beschreibung verzichtet und auf Anhang B.3 verwiesen werden.

3.3.4. Schnittstellendefinition

Der Treiber mit seinen Kommandotypen und Kompositionen definiert die Datenströme, welche zum Gerät gesendet bzw. von diesem empfangen werden. Er definiert hingegen nicht, auf welche Weise die Datenströme zum Gerät gelangen. Ob Daten über die RS232- oder USB-Schnittstelle zum Gerät übertragen werden, ist aus Sicht des Treibers irrelevant. Transportprotokolle und -technologien werden über einen anderen Mechanismus realisiert, der ausschließlich für die Konfiguration von Geräteschnittstellen verantwortlich sein soll.
Durch die Trennung der Geräteschnittstellen-Deskriptoren von den Geräte-Treibern wird ein hohes Maß an Flexibilität erreicht, da es für den Messablauf oder etwaige Gerätekonfigurationen irrelevant ist, über welche Schnittstelle die Kommando- und Konfigurationsdaten zum Gerät übertragen werden.

Konfiguration der Schnittstellen

Im Folgenden wird eine Auflistung der von Excalibur unterstützten Schnittstellen gezeigt inkl. ihrer Konfigurationsparameter. Die genaue technische Funktionsweise ist in Abschnitt 3.1.2 besprochen worden.

Ausgehend vom Verständnis für die einzelnen Konfigurationsparameter können jetzt Strukturen entworfen werden, um die einzelnen Schnittstellen zu kapseln. Für jede der einzelnen Schnittstellen wird eine eigene Struktur gebildet, den sog. Schnittstellentypen, welche über alle notwendigen und hinreichenden Eigenschaften der jeweiligen Schnittstelle verfügt, d.h. eine Struktur für RS232, USB, GPIB und die Library. Die exakte Struktur soll hier nicht gezeigt werden (vollständige Struktur siehe Abschnitt 3.1.2). Auf eine weitere Schnittstelle soll aber zusätzlich noch eingegangen werden, welche eine wichtige Rolle für die Integrationsphase spielt:

Tabelle (3.2)
Übersicht der von Excalibur unterstützten Schnittstellen und ihrer wichtigsten Konfigurationsparameter.

Schnittstelle	Übertragung	Konfigurationsparameter
RS232	Seriell	Port, Baudrate, Datenbits, Stopbits, Parität, Flusskontrolle
USB	Seriell	Vendor ID, Produkt ID, Seriennummer, Schnittstellen Nummer, Endpunkte
GPIB	Parallel	Schnittstellen ID, Primäre Adresse, Sekundäre Adresse, EA Timeout, EOS Byte, Auto Polling, Bus Timing
Library	Hauptspeicher	Pfad
LAN	Ethernet/Wireless	Netzwerk-Adresse, Subnetzmaske, Gateway, DNS-Server

Testschnittstelle Die Testschnittstelle wird für die Integrationsphase benötigt, bei welcher der Geräte-Aufbau noch nicht vollständig erfolgt ist. Hier kann der Treiber des zukünftigen Gerätes eingesetzt und in der zu entwickelnden Umgebung getestet werden, um alle anderen Aspekte anhand eines konkreten Gerätes (mit simulierter Funktionalität) zu entwickeln. An die Testschnittstelle können alle Befehle des angeknüpften Treibers gesendet werden und sie antwortet immer mit der ersten definierten Geräteantwort, indem sie die Platzhalter mit Zufallsdaten im angegebenen Format bestückt.

Verknüpfung der Schnittstellen mit den Gerätetreibern

Sind die Geräteschnittstellen über die jeweiligen Elemente konfiguriert, werden sie in einer Struktur gekapselt, welche im Folgenden als *Schnittstellenkonfiguration* bezeichnet werden soll. Sie wird aus einer Menge aus oben definierten konkreten Schnittstellentypen gebildet. Alle leiten von einem allgemeinen Schnittstellentyp ab, der fundamentale Eigenschaften einer jeden Schnittstellendefinition definiert. Die beiden wichtigsten Eigenschaften sind die *deviceRef* und der *driver*. Unter der *deviceRef* wird ein Name für eine Schnittstellendefinition angegeben, welcher auch für das angeschlossene Gerät innerhalb von Excalibur verwendet wird. Mit dieser Referenz kann nun von jeder Stelle innerhalb des Excalibur-Servers auf das zugrundeliegende Gerät Bezug genommen werden. Die Angabe des *driver* erfolgt über den Namen der XML-Datei, welche im Verzeichnis

„driver“ (oder einem seiner Unterverzeichnisse) zu finden ist und eine korrekte Struktur nach Vorschrift der XSD-Datei enthält. Mit Hilfe der Gerätereferenz ist es nun möglich ein Gerät zu identifizieren, dessen Befehlssatz mit seinen Kommando und Kompositionen zu finden und ebenso die Schnittstelle, über welche die Befehle gesendet bzw. deren Antworten empfangen werden sollen.
Die Definition des allgemeinen Schnittstellentyps lautet wie folgt:

Listing (3.8)

XML Schema des Interface Types

```
1 <complexType name="interfaceType">
2  <attribute name="deviceRef" type="string" use="required"/>
3  <attribute name="driver" type="string" use="required"/>
4  <attribute name="address" type="string" use="optional"/>
5 </complexType>
```

Eine weitere Eigenschaft ist die *address*, auf welche bereits in Abschnitt 3.3.3 verwiesen worden ist. Es handelt sich um eine eindeutige Geräteadresse, welche für das angeschlossene Gerät angegeben werden kann. Dies kann beispielsweise eine Identifikation darstellen, welche per Software oder Jumper eingestellt wird, falls mehrere dieser Geräte angeschlossen und über ein proprietäres Bussystem miteinander verbunden sind, d.h. dass sie über dieselbe Schnittstelle miteinander vernetzt sind. Diese Adresse kann natürlich zur Laufzeit überschrieben werden.

3.3.5. Datenein- und -ausgabe

Software ist in den meisten Fällen nur sinnvoll, wenn die Verarbeitungsroutine des Softwarekerns durch externe Eingaben beeinflusst werden kann. Ebenso muss die Verarbeitungsroutine Mitteilungen und Daten wieder nach Extern zurückgeben können. Im Allgemeinen wird dies als EVA-Prinzip (s. Glossar) bezeichnet. Das EVA-Prinzip *EVA*
wird im Excalibur Server durch das Reader-Writer-Konzept realisiert, unter welchem ein transportmechanismus-basierter Ansatz verstanden wird. Dabei wird für jeden systeminternen Prozess ein abstraktes Modell definiert, um sämtliche Eigenschaften des Prozesses abzubilden. Dieses Modell wird anschließend durch sog. *Deserialisierer* mit Werten bestückt und dessen Operationen werden an einen Transportmechanismus delegiert. Die Ausgabe erfolgt entsprechend in die Gegenrichtung und geschieht über die

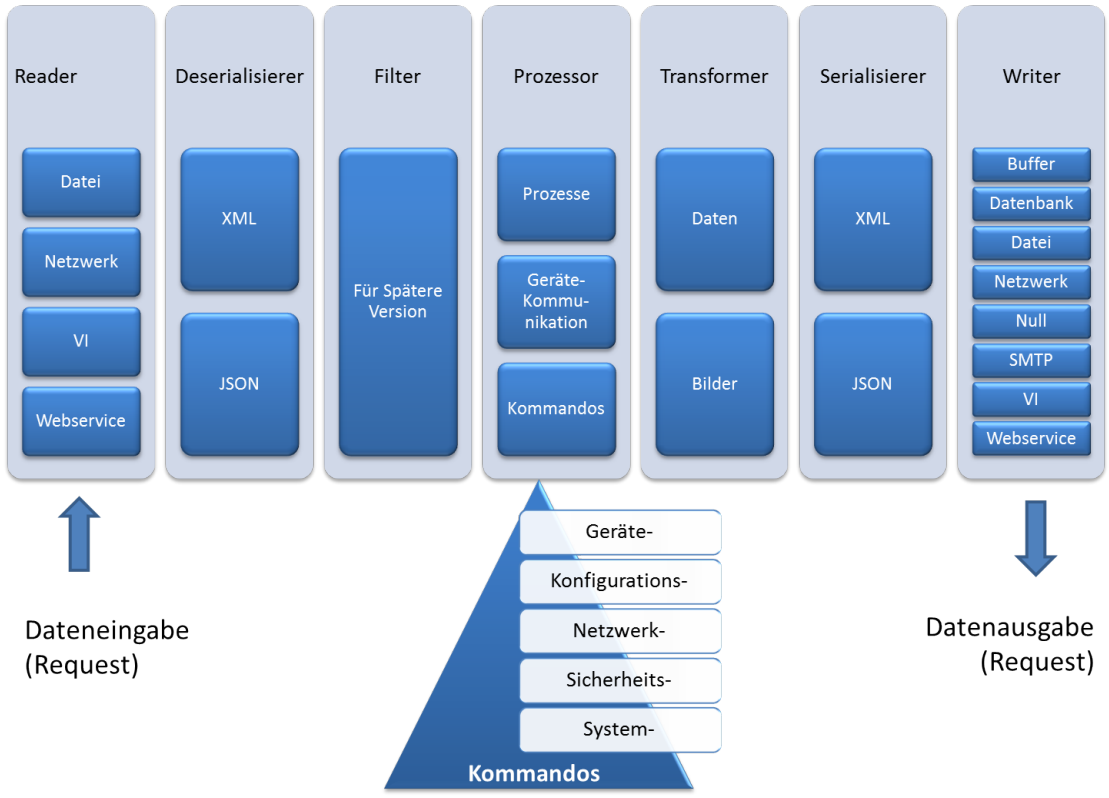


Abbildung (3.2)
Abfolge der einzelnen Ein- und Ausgabe-Elemente des Excalibur EVA-Prozesses. Der Vorgang wird ausgelöst durch das Eintreffen eines Requests, durchläuft anschließend die gesamte EVA-Kaskade und endet in einer Datenausgabe.

Definition von sog. *Serialisierern*. Die Parameter des Modells werden anschließend dazu verwendet, um eine Kommandostruktur aufzubauen, welche inkrementell vom Softwarekern abgearbeitet wird.

Um im Detail zu verstehen, wie das Konzept funktioniert, sollen die wichtigsten Punkte schrittweise erklärt werden und sind folgendermaßen auf die nächsten Unterkapitel aufgeteilt (s. Abbildung 3.2):

1. Zuerst erfolgt eine Auflistung aller möglichen Eingabe-Strukturen von außen nach innen (Reader -> Deserialisierer -> Filter)

2. Anschließend wird beschrieben, wie aus den einzelnen Strukturen ein Prozess konstruiert wird
3. Nun wird auf die verschiedenen Prozesse und die daraus generierten Kommandos eingegangen
4. Im nächsten Abschnitt erfolgt wieder eine Auflistung aller möglichen Ausgabe-Strukturen von innen nach außen (Transformer -> Serialisierer -> Writer)
5. Dann wird darauf eingegangen, wie aus einem Prozess Daten erzeugt und über interne Strukturen weitergegeben werden
6. Und zuletzt wie diese Strukturen in Ausgabedaten übersetzt und extern ausgegeben werden

Reader, Deserialisierer und Filter

Reader, Deserialisierer und Filter definieren die Struktur und die Schnittstellen von Eingaben für den Excalibur Server. Dadurch sind sie ausschließlich für Systemintegratoren interessant, welche die Konfigurationseinstellungen für den Excalibur Server durchführen.

Reader - Übersicht und Funktionsbeschreibung Unter einem Reader wird in Excalibur ein Transportmechanismus verstanden, mit welchem Daten in die Software eingegeben werden können. Es werden 4 verschiedene Reader angeboten, welche jeweils Implementierungen für verschiedene Transportmechanismen darstellen:

- File-Reader
- Network-Reader
- VI-Reader
- Webservice-Reader

Ein Reader stellt dabei ausschließlich den Transportmechanismus selbst zur Verfügung und lässt keine Interpretation der Daten zu. Rein prinzipiell lassen sich dadurch beliebige Daten an den Reader senden, solange das zugrunde liegende Transport-Protokoll eingehalten wird, welches für die einzelnen Reader separat konfiguriert werden kann. Im Folgenden soll eine kurze Einführung der implementierten Reader gegeben werden.

Eine vollständige Auflistung der Eigenschaften befindet sich in Anhang B.5. Alle Reader arbeiten im Polling-Modus, d.h. ein separater Prozess ruft permanent die Lesefunktion des Readers auf und überprüft, ob Daten vorhanden sind. Über das sog. *readInterval* kann die Wartezeit zwischen den einzelnen Lesevorgängen definiert werden.

Tabelle (3.3)
Übersicht der von Excalibur unterstützten Readern und ihrer wichtigsten Eigenschaften.

Reader Name	Beschreibung
File-Reader	Beobachtet Verzeichnisse auf der lokalen Festplatte und deren enthaltene Dateien. Sobald eine Datei mit einer enthaltenen Anfrage erkannt wird (z.B. indem sie über einen externen Prozess in das beobachtete Verzeichnis geschoben wird), liest der Prozess die Datei aus, führt das enthaltene Kommando aus und löscht die Datei anschließend.
Network-Reader	Lauscht auf einem TCP- oder UDP-Port auf das Eintreffen eines Datenpakets, mit welchem wie oben verfahren wird. Das verwendete Standardprotokoll ist HTTP, aber es wird auch das ni.dex-Protokoll unterstützt, welches die Daten als sog. LabVIEW® Netzwerk Streams behandelt; es stellt ein von National Instruments entwickeltes Netzwerkprotokoll dar[8]. Ist TCP als Protokoll konfiguriert, wird automatisch mit dem Reader auch ein *Network-Writer* (s. Abschnitt 3.3.5) angelegt, da auf eine Anfrage immer auch eine notwendige Antwort folgen muss (Protokoll ist verbindungsorientiert).
VI-Reader	Ausgehend von einer vordefinierten Struktur, kann ein LabVIEW®-VI konstruiert werden, welches von selbst geschriebenen LabVIEW®-Programmen eingebunden werden kann, um an den Excalibur-Prozess Befehle zu senden.

[8] http://www.ni.com/white-paper/12267/en/

Webservice-Reader	Der *Webservice-Reader* stellt einen Webservice zur Verfügung, über welchen mit SOAP-Requests (s. Simple Object Access Protocol)[9] Daten an den Excalibur Server übermittelt werden können. Über eine zugrundeliegende WSDL-Datei können dabei alle möglichen Befehle und ihre Konfigurationen automatisiert abgerufen und realisiert werden. Wie auch beim *Network-Reader* wird bei der Kommunikation über das verbindungsorientierte TCP automatisch ein *Webservice-Writer* erzeugt (s. Abschnitt 3.3.5).
Composite-Reader	Der *CompositeReader* spielt eine spezielle Rolle und stellt ein Strukturelement dar, welche keine eigene Funktionalität erfüllt. Er ist dafür vorgesehen, alle anderen Reader zu kapseln, um Baumstrukturen zu realisieren. Dies ist nützlich, damit reader-implementierende Strukturen nicht immer mehrere Transportmechanismen abarbeiten müssen, sondern nur einen (Parent-)Composite beauftragen, alle Eingaben gebündelt einzulesen und eine Liste an Anfragen weiterzugeben.

Deserialisierer - Übersicht und Funktionsbeschreibung Der Deserialisierer ist für die Interpretation der am Reader eintreffenden Rohdaten zuständig. Um dies zu ermöglichen, *muss* für jeden Reader *genau ein* Deserialisierer definiert werden, der von diesem Zeitpunkt an festlegt, welches Datenformat an der Eingabeschnittstelle erwartet wird. Jede Anwendung, welche mit dieser Schnittstelle kommunizieren will, muss das zugrunde liegende Protokoll des Transportmechanismus beherrschen und die Daten im erwarteten Format senden. Soll eine andere Schnittstelle oder ein anderes Datenformat verwendet werden, muss (wenn möglich) eine weitere Reader-Deserialisierer Kombination eingerichtet werden. Im Moment existieren zwei Deserialisierer, der *JSON-* und der *XML-Deserialisierer*.
Der prinzipielle Aufbau ist für beide Datenformate ähnlich und lautet wie folgt:

- Es existiert *genau* ein Hauptelement, dessen Name auch den anzustoßenden Prozess spezifiziert
- Als erstes Unterelement wird immer eine Session-ID erwartet[10] (welche beim Öffnen

[10] Außer beim AccessCtrl-Prozess, durch welchen eine neue Session geöffnet und eine Session-ID zurückgegeben wird

einer Excalibur-Session erzeugt wird)

- Alle weiteren Geschwister-/Kind-Elemente listen die Parameter in einer für den angefragten Prozess wohldefinierten Struktur auf, mit dem Ziel ebendiesen vollständig zu konfigurieren

Beim *JSON-Deserialisierer* wird die Struktur als im JSON-Austauschformat definierter Datenstrom erwartet. Die Identifikation des Prozesses erfolgt über den Namen des ersten Objekts, unter welchem die dafür benötigten Parameter angegeben werden.

Listing (3.9)

Struktur eines Requests im JSON Format

```
{
 requestName: {
  sessionId: random_hexadecimal_characters,
  param0: value0,
  object0: { ... }
  ...
 }
}
```

Der *XML-Deserialisierer* erwartet einen XML-Datenstrom, wobei die Identifikation des Prozesses über das Root-Element erfolgt.

Listing (3.10)

Struktur eines Requests im XML Format

```
{
 <requestName>
  <sessionId>random_hexadecimal_characters</sessionId>
  <param0>value0</param0>
  <object0>
   ...
  </object0>
  ...
 </requestName>
}
```

Ein letzter Strukturtyp der Eingabeprozedur ist noch nicht besprochen worden: Der Filter. Während der Deserialisierer aus einem externen Datenformat wie XML oder JSON eine interne Excalibur-Struktur, d.h. Objekte konstruiert, soll mit den Filtern die interne Datenstruktur überprüft und gegebenenfalls verworfen bzw. angepasst werden. Das Konzept des Filters ist bislang ausschließlich exemplarisch eingeführt, ist aber noch nicht realisiert worden. Es wird erst in einer zukünftigen Excalibur-Version implementiert, wird hier aber dennoch aus Gründen der Vollständigkeit genannt. Die exakte Konfiguration der Deserialisierer und Filter ist individuell vom Reader-Typ abhängig und wird ausführlich in Abschnitt B.5 besprochen.

Von der Anfrage bis zum Prozess Damit eine Kommunikation mit dem Excalibur Server möglich ist, müssen dafür eine beliebige Anzahl an Readern, deren Deserialisierer und (optional) Filter definiert werden. Dies geschieht über die zentrale Konfigurationsdatei (s. Abschnitt B.2), wobei auch nachträglich zur Laufzeit, alle EA-Komponenten hinzugefügt, gelöscht und ausgetauscht werden können. An jede der über den Reader-Typ definierten Schnittstellen können nun Anfragen gesendet werden, während die enthaltenen Deserialisierer das Format der Anfrage bestimmen und es in ein von Excalibur lesbares Format übersetzen.

Sobald eine Anfrage an einen der konfigurierten Reader eintrifft, wird der in Abbildung 3.3 gezeigte Ablauf angestoßen. Beginnend bei dem durch den Reader definierten Transportmechanismus, werden die Daten byteweise gelesen und der Datenstrom inkl. aller im Transportmechanismus mitgelieferten Metainformationen (meist im Header angegeben) gespeichert. Der Deserialisierer versucht nun mit der ihm zugrundeliegenden Grammatik, die Daten zu interpretieren. Dazu liest er das Hauptelement, wählt eine geeignete Datenstruktur aus und setzt den ihr zugrundeliegenden Prozess als Parameter. Anschließend erfolgt die Überprüfung der SessionId (außer im Falle des AccessCtrl Requests) und es wird am Rechte Management des Systems angefragt, ob es der referenzierten Session gestattet ist, den angefragten Prozess auszuführen. Falls es diesem nicht gestattet ist, wird der Request verworfen und die Access-Denied Routine gestartet (s. Abschnitt 3.3.10).
Nach Überprüfung der Rechte versucht der Deserialisierer den akzeptierten Datenstrom vollständig auf die Datenstruktur abzubilden. Sind alle obligatorischen Eigenschaften gesetzt, wird die Datenstruktur den Filtern übergeben, welche abhängig von ihren Fil-

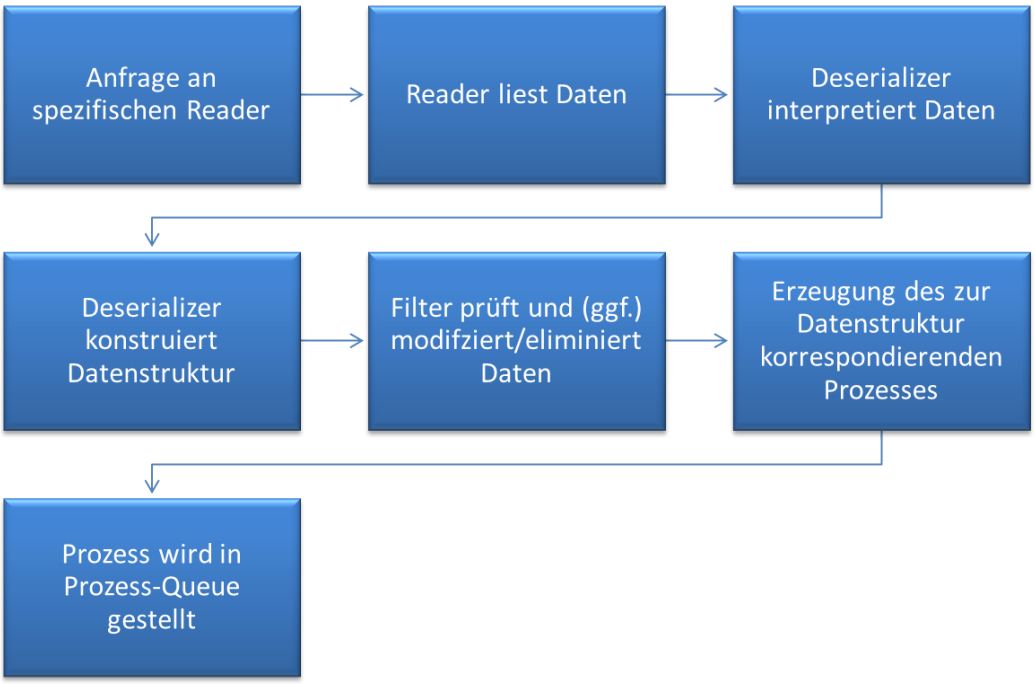


Abbildung (3.3)
Genereller Ablauf des Anfrage-Mechansimus des Excalibur Servers. Nach dem Eintreffen des Datenstroms am Reader, werden die Daten mit Hilfe des Deserialisierers intepretiert, über den Filter aussortiert/modifiziert und anschließend ein interner Prozess erzeugt und in die Prozess-Queue zur Verarbeitung gestellt.

terregeln die Daten modifizieren, verwerfen oder unverändert weiterleiten.[11] Sobald die gefüllte Datenstruktur diesen Punkt passiert hat, greift sie der Excalibur-Prozessor auf und erzeugt daraus einen Prozess, der die Datenstruktur speichert, einen Zeitstempel setzt, eine eindeutige Prozess-Id, -kategorie (s. Tabelle 3.4) und -priorität zuweist und sie dem Prozess-Manager übergibt. Jener setzt den Status (s. unten) des Prozesses auf „New“ und stellt ihn in eine Warteschleife, von welcher ausgehend die einzelnen Prozess-Stati durchlaufen werden. Der angestoßene Prozess ist ab jetzt aktiv und wird erst aus dem Prozess-Manager entfernt, wenn er entweder abgearbeitet oder durch einen Fehler abgebrochen wird.

[11] Im Moment werden alle Daten von den Filtern ungeprüft weitergeleitet

Tabelle (3.4)
Auflistung in Excalibur definierter Prozess-Kategorien und ihre Bedeutung.

Name	Bedeutung
Default	In der Standardkategorie werden ohne Abweichungen alle Prozess-Stati durchlaufen
Error	Falls während der Verarbeitung innerhalb eines Prozess-Status ein Fehler auftritt, wird die Kategorie des Prozesses automatisch auf diese gesetzt und es wird eine Fehlerbehandlungsroutine aktiviert
Program	Der Prozess enthält eine Ablaufsequenz und wird in einem separaten Thread abgehandelt. Dabei wird für jeden Sequenzschritt ein neuer Prozess eingeleitet und der Status „Run“ gesetzt
System Control	Der Prozess soll Veränderungen am Excalibur-System vornehmen oder Informationen von diesem beziehen (z.B. Zeit abfragen, Konfigurationen verändern, usw.).

Prozess-Stati Nachdem bereits einige Prozess-Stati genannt worden sind, soll im Folgenden kurz darauf eingegangen werden, was unter diesen hier verstanden werden soll. Während des Lebenszyklus eines Prozesses durchläuft er mehrere Schritte in einer wohl definierten Reihenfolge. Die Identifikation des aktuellen Schritts wird über den Prozess-Status kenntlich gemacht und jeder Prozess-Schritt setzt den Status des nächsten Schritts. Damit ähnelt es dem informatischen Modell des Zustands-Automaten. Abbildung 3.4 zeigt exemplarisch anhand eines Prozesses der *Default*- und der Program-Kategorie, welche Prozesse dies sind und in welcher Reihenfolge sie durchlaufen werden. Nachdem der Status „finish“ erreicht ist, erfolgen finale Aufräumvorgänge und der Prozess wird abschließend gelöscht.

Vom Prozess zum Kommando Der Prozess selbst verrichtet prinzipiell keine Arbeit am System; er hält nur eine Datenstruktur zur Bearbeitung bereit. Erst über den Mechanismus der *Kommandos* werden direkte Aktionen am System durchgeführt. Gelangt ein Prozess in die Prozess-Warteschleife, ist dies das Zeichen für den Prozess-Manager, anhand der im Prozess hinterlegten Datenstruktur, ein oder mehrere Kommandos zu konstruieren und sie einem Arbeiter-Thread zu übergeben. Kommandos sind die eigentlichen Akteure des Excalibur-Systems, welche sämtliche Aufgaben inkl. der Gerätesteuerung übernehmen. Zwar werden alle Kommandos von einer gemeinsamen Basis

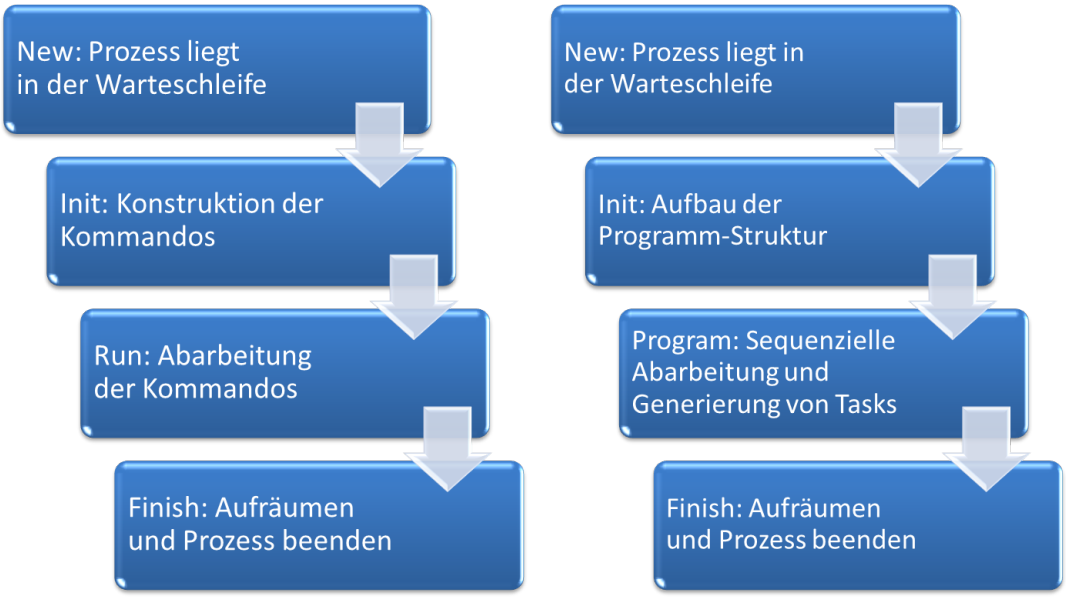


Abbildung (3.4)
Lebenszyklus eines Prozesses, die einzelnen Schritte und deren Reihenfolge, in welcher sie abgearbeitet werden. Links: Default-Prozess, Rechts: Program-Prozess

abgeleitet, so hat aber dennoch jede Aktion ein eigenes korrespondierendes Kommando (s. nächstes Kapitel und Abschnitt 3.4.3).

Kommandos und Kompositionen Es werden 5 verschiedene Kommandotypen unterschieden:

- Gerätekommandos
- Konfigurationskommandos
- Netzwerk-Kommandos
- Sicherheitskommandos
- Systemkommandos

Jede Aktion kann einem dieser Typen zugeordnet werden und bildet einen eigenen Sub-Typ, welcher direkt vom korrespondierenden Elter abgeleitet wird. Auf die einzelnen (Haupt-)Typen soll im Folgenden eingegangen werden.

Gerätekommandos Die wichtigsten für eine Steuerungssoftware sind eindeutig die *Gerätekommandos*. Sie kapseln sämtliche Aktionen, welche einen Einfluss auf die angeschlossenen Geräte haben, darunter das *Schreiben* von Datenströmen, ebenso wie das *Lesen*. Eine weitere Aktionen ist das *Verbinden* mit dem Gerät und damit die Verknüpfung von ebendiesem mit einer Schnittstellendefinition, einem Treiber und einer Konfiguration; zusätzlich wird damit dem Betriebssystem die Schnittstellenkonfiguration mitgeteilt und als „geöffnet“ markiert. Wenn eine Aktion *Verbinden* definiert ist, existiert zwangsläufig auch die Aktion *Trennen*, welche die Verbindung zum Gerät „schließt“ und sämtliche benötigte Ressourcen wieder freigibt.
Die Kommunikation mit Laborgeräten erfolgt über das Schreib- und das Lesekommando in genau dieser Reihenfolge. Alle Geräte verbleiben im Normalfall im aktuellen Zustand, es sei denn, über ein Schreibkommando wird dieser Zustand geändert (s. Abschnitt 3.3.3). Erst anschließend wird das Gerät auf den Vorgang antworten und die Antwort kann mit Hilfe eines Lesekommandos abgerufen werden. Lesekommandos ohne ein vorheriges Schreibkommando sollten seltener in Geräten vorkommen. Ein Beispiel hierfür sind Event-basierte Geräte, welche der aufrufenden Steuerungssoftware Zustandsänderungen über Callback-Daten zur Verfügung stellen, welche diese zu einem beliebigen Zeitpunkt über ihre Hardware-Schnittstelle senden können. Das heißt in anderen Worten, Lesen ohne vorigen Schreibbefehl ist vor allem für asynchrone Gerätekommunikation wichtig.

Schreib- und Lesekommandos repräsentieren im übertragenen Sinn eine Kapselung der bereits oben eingeführten fertig konstruierten Tokens aus Excalibur-Kommandos, wobei näheres hierzu unter Abschnitt 3.3.6 zu finden ist. Das Verbinden und Trennen von Geräten spielt ebenfalls eine fundamentale Rolle. Dabei werden wie bereits angedeutet, verschiedenste Verknüpfungen des Systems mit den Geräten vorgenommen. Zum einen werden, wie auch für das Excalibur-System selbst, verschiedene Writer, Serialisierer und Transformer konfiguriert (s. dazu Abschnitt 3.3.5). Zum anderen wird den Geräten eine individuelle Konfiguration mitgeteilt, mit welcher benötigte Konstanten, Variablen und Initialisierungskommandos gesetzt werden können, wie z.B. Berechnungen von Adressen, initialisierende Gerätebefehle oder allgemeine Betriebsparameter. Um letztendlich mit Geräten kommunizieren zu können, muss deshalb zuerst eine Verbindung aufgebaut werden.

Konfigurationskommandos Zentrale und auch umgebungsbedingte Konfigurationen des Servers sollen von den Client-Anwendungen auf den Server übertragen werden, damit nicht jede Client-Anwendung individuell konfiguriert werden muss, sondern die Ein-

stellungen von einer zentraler Stelle abgerufen werden können. Dafür sind die Konfigurationskommandos zuständig, welche Funktionalitäten für die Persistenz von Konfigurationen zur Verfügung stellt. Das Thema Persistenz wird weiter unten in Abschnitt 3.3.8 erklärt werden.

Netzwerk Kommandos Netzwerk-Kommunikation im Allgemeinen ist meist nur über den Aufbau komplexer Netzwerkarchitekturen und -Hardware (Router, Switches, usw.) realisierbar, wodurch es beim Ausfall einiger zentraler Komponenten vorkommen kann, dass eine Datenübertragung nicht mehr möglich ist. Auch ist bei Netzwerken mit hohem Durchsatz oft mit einer Reduktion der Übertragungsgeschwindigkeit zu rechnen und damit auch mit einem Überlaufen der Datenpuffer am sendenden Medium bzw. mit Timeouts. Aus diesem Grund werden Aktionen auf Netzwerkebene in Kommandos gekapselt und innerhalb einer separaten Warteschleife abgearbeitet. Im Falle eines Netzwerkausfalls, werden die Daten gespeichert und bei Verfügbarkeit erneut gesendet, ohne den gesamten Excalibur Server zu blockieren. Darunter fallen beispielsweise alle Datenspeicherungsroutinen (FTP, WebDAV, usw.).

Sicherheitskommandos Damit keine ungewollten Zugriffe auf den Excalibur Server erfolgen, gilt es diese bereits im Vorfeld durch geeignete Sicherheitsmechanismen zu unterbinden. Dadurch können Datenverluste, Datenspionage oder gar Hardwareschäden verhindert werden. Zur Verhinderung derartiger Zugriffe werden Mechanismen für Authentifizierung und Autorisierung benötigt, was Aufgabe der Sicherheitskommandos ist. Das Thema Sicherheit wird weiter unten in Abschnitt 3.3.10 erklärt werden.

Systemkommandos Sollen Aktionen direkt am Excalibur-System durchgeführt werden, sind hierfür die Systemkommandos zu verwenden. Sie sind die Akteure, um beispielsweise zentrale Einstellungen lesen und schreiben zu können, laufende Prozesse zu beeinflussen, Zeitsynchronisationen zu bewerkstelligen, usw. Es existiert eine Vielzahl an Systembefehlen, welche direkt den Excalibur Server per se beeinflussen; eine vollständige Übersicht befindet sich in Abschnitt B.9.

Kompositionen Wie gezeigt worden ist, erscheint ein Lese- ohne korrespondierenden Schreibvorgang nicht sinnvoll. Solche und ähnlich eng zusammengehörige Aktionen, werden innerhalb von Excalibur zu einer einzigen Kompositions-Struktur zusammengefasst

und nacheinander abgearbeitet. Auf diese Weise können Prozesse komplexe Kommando-Kaskaden konstruieren, welche in einer wohl definierten Reihenfolge, zusammengehörige Aktionen gebündelt ausführen können. Zusätzlich kann bei der Konstruktion entschieden werden, ob die Kommandos Abhängigkeiten voneinander aufweisen, d.h. das ein darauffolgendes „abhängiges“ Kommando nur bei fehlerfreier Abarbeitung des ersten ausgeführt oder ansonsten verworfen wird. Derartige Abhängigkeiten sind über die Prozess-Struktur selbst nicht abbildbar, da sich verschiedene Prozesse „immer“ vollständig unabhängig voneinander verhalten.

Writer, Serialisierer und Transformer

Nach der Erzeugung aller für den zugrundeliegenden Prozess benötigten Kommandos, werden diese nun schrittweise abgearbeitet und führen Aktionen auf dem Excalibur-System oder auf den Geräten aus. Nach der Abarbeitung aller Kommandos muss jeder Prozess dem Aufrufer/System mitteilen, ob er seine Aufgabe erfüllt hat oder ob ein Fehler produziert worden ist. Dafür gibt es Kommandos, welche diese Antworten generieren und sie dem System zur Ausgabe wieder zur Verfügung stellen. Die Ausgabe selbst wird über die sog. *Writer* realisiert. Sie sind das Gegenstück der *Reader*, und sie stellen wie diese einen Transportmechanismus dar. Die Formatierung der Daten für den Transport wird von den sog. *Serialisierern* übernommen und sie übersetzen interne Datenstrukturen in das gewünschte Ausgabeformat.
Writer, Serialisierer und Transformer definieren die Struktur und die Schnittstellen von Ausgaben für den Excalibur Server aber auch für die Geräte. Dadurch sind sie ausschließlich für Systemintegratoren interessant, welche die Konfigurationseinstellungen für den Excalibur Server durchführen oder die Geräte des zu entwickelnden analytischen Aufbaus konfigurieren. Zwar können Benutzer die Geräteausgaben innerhalb der Ablaufsequenzen umleiten und zwischenspeichern, wodurch aber die Konfiguration der einzelnen Writer/Serialisierer/Transformer nicht verändert wird.

Writer - Übersicht und Funktionsbeschreibung Es werden 8 verschiedene Writer angeboten, welche jeweils Implementierungen für verschiedene Transportmechanismen darstellen:

- Buffer-Writer
- Datebase-Writer

- File-Writer
- Network-Writer
- Null-Writer
- SMTP-Writer
- VI-Writer
- Webservice-Writer

Tabelle (3.5)
Übersicht der von Excalibur unterstützten Writern und ihrer wichtigsten Eigenschaften.

Writer Name	Beschreibung
Buffer-Writer	Stellt eine Warteschleife zur Verfügung, womit genierte Daten gespeichert und zu einem späteren Zeitpunkt wieder abgerufen werden können. Auf diese Weise ist es möglich, auf dem Excalibur Server einen Prozess zu starten, sich selbst überlassen, und Daten asynchron abzufragen.
Database-Writer	Leitet Daten an eine Datenbank-Schnittstelle (SQL-basierend) weiter und versucht für jeden ankommenden Datensatz Einträge in Datenbank-Tabellen hinzuzufügen. Diese Tabellen müssen im Writer konfiguriert werden und der Spaltentyp muss für die aus dem Serialisierer stammenden Ausgabedatentypen übereinstimmen. Eine bessere Möglichkeit, die Ausgabedaten strukturiert in Datenbank-Tabellen zu speichern, bieten die Templates, die in Abschnitt 3.3.7 besprochen werden.

File-Writer	Ist für die Speicherung der Ausgabedaten in Dateien zuständig. Als Konfiguration benötigt er nichts weiteres als einen Ordner auf der lokalen Festplatte, in welche der *File-Writer* Dateien schreibt, sie mit den aus dem Serialisierer stammenden Daten füllt und in den angegebenen Ordner speichert. Werden mehrere Messungen durchgeführt, wird für jede einzelne eine neue Datei angelegt und eine Zahl angehängt. Auch hier bieten Templates (s. Abschnitt 3.3.7) viel bessere Strukturierungsmgöglichkeiten. Der *File-Writer* bietet darüber hinaus die Möglichkeit, die Dateien nicht nur lokal, sondern auch im Netzwerk zu hinterlegen; mögliche Protokolle zur Datenübertragung sind FTP und WebDAV.
Network-Writer	Sollen Daten über das Netzwerk gesendet werden, bietet der *Network-Writer* die Möglichkeit über TCP- bzw. UDP-Ports Ausgabedaten vom Serialisierer zu senden. Wie auch beim *Network-Reader* kann zwischen HTTP und dem ni.dex-Protokoll gewählt werden.
Null-Writer	Soll gekennzeichnet werden, dass Daten eines bestimmten Geräts explizit verworfen werden sollen, kann dafür ein *Null-Writer* verwendet werden. Bei dessen Verwendung werden Daten direkt aus dem Speicher entfernt.
SMTP-Writer	Das SMTP-Protokoll wird für die Übertragung von Mitteilungen über Netzwerke verwendet (z.B. Emails). Sollen gewisse Status-Mitteilungen das Ende von Ablaufsequenzen oder Fehler bekannt gemacht werden, können sie per SMTP direkt an den Benutzer adressiert werden. Auch ganze Datenausgaben können per SMTP übertragen werden, entweder unformatiert wie mit Hilfe des Serialisierers erzeugt oder mit Hilfe des Templating-Mechanismus, um die Daten möglichst detailliert zu strukturieren (s. Abschnitt 3.3.7).

VI-Writer	Ist das Gegenstück zum *VI-Reader*, d.h. es mit dessen Hilfe können VIs mit LabVIEW® erzeugt werden, um die Ausgabedaten mit spezialisierten VI aufzunehmen und zu verarbeiten. Dafür kann natürlich auch dasselbe VI wie für den Reader verwendet werden.
Webservice-Writer	Stellt einen Port für die Ausgabe zur Verfügung, um auf einkommende Webservice-Anfragen zu antworten. Im Moment kann der *Webservice-Writer* keine Webservice-Anfrage initiieren, um Daten beispielsweise an einen fremden Webservice-Provider weiterzuleiten[12].
Composite-Writer	Ist wie auch sein korrespondierender Reader ein Strukturmechanismus. Es können somit Ausgabekaskaden konstruiert werden, um Daten gezielt zu modifizieren und die veränderten Daten mit Hilfe von Kanälen (s. Abschnitt 3.3.5) in unterschiedliche Ausgabemedien delegieren.

Serialisierer - Übersicht und Funktionsbeschreibung Der Serialisierer wandelt erzeugte Ausgabedaten in ein anhand einer, abhängig vom gewählten Serialisierer, wohl definierten Grammatik definiertes Format um, welches als Ausgabe des Systems nach Extern weiter- oder zurückgegeben werden soll. Für jeden Writer *muss* genau *ein* Serialisierer definiert werden, der die interne Datenstruktur verstehen und umwandeln kann. Es sind 3 verschiedene Serialisierer definiert:

- JSON-Serialisierer
- Template-Serialisierer
- XML-Serialisierer

Wie auch bei den Deserialisierern wird mit Hilfe des *JSON-* und des *XML-Serialisierers*, JSON bzw. XML Code erzeugt. Die Inhalte der Ausgaben sind für alle möglichen Ausgaben wohl definiert, wobei die Ausgabestruktur der beiden Serialisierer ähnlich ist:

[12]Kommunikation mit externen Webservice-Providern sind erst in zukünftigen Versionen geplant.

Listing (3.11)

Struktur einer Response im JSON Format

```
{
  responseName: {
    param0: value0,
    object0: { ... }
    ...
  }
}
```

Listing (3.12)

Struktur einer Response im XML Format

```
<responseName>
  <param0>value0</param0>
  <object0>
    ...
  </object0>
  ...
</responseName>
```

Ein Spezialfall stellt der *Template-Serialisierer* dar, welcher die interne Datenstruktur mit Hilfe einer Strukturierungsanleitung in eine konfigurierbare Datenstruktur übersetzt. Dieser Serialisierer-Typ muss mit einer speziellen Formatierungsanleitung bestückt werden, nach wessen Vorschriften eine Datenausgabe sehr variabel formatiert werden kann. Der *Template-Serialisierer* erzeugt mit Hilfe der Kombination aus Strukturierungsanleitung und den Ausgabedaten eine spezielle Ausgabestruktur in einer Syntax, welche nur von einigen der Writer interpretiert und ausgegeben werden kann (Database-, File- und SMTP-Writer). Jene Writer können diese Syntax anschließend in das von ihnen repräsentierte Ausgabeformat übersetzen. Das Funktionsprinzip wird detailliert in Abschnitt 3.3.7 abgehandelt werden.

Transformer - Übersicht und Funktionsbeschreibung Ein letztes Strukturelement steht für die Ausgabe noch zur Verfügung, die sog. Transformer. Im Übereinstimmung mit den Filtern der Eingabe, kann auch während der Ausgabe der Zugriff auf die Daten erfolgen und diese ggf. verändert oder verworfen werden. Jedem Writer kann eine beliebi-

ge Anzahl an Transformern angehängt werden, wobei es zwei Typen gibt, welche beliebig oft und in beliebiger Reihenfolge eingesetzt werden können: der *Data-Transformer* und der *Image-Transformer*. Ersterer stellt einen Wrapper für mathematische Berechnungen (s. Abschnitt 3.3.5) dar, wodurch während des Ausgabeprozesses Einfluss auf generierte Daten genommen werden kann, bevor sie zur Speicherung abgelegt werden. Der zweite Transformer ist ebenfalls ein Wrapper, der Funktionalitäten für die Bildbearbeitung bereitstellt (s. Abschnitt 3.5.4). Auf diese Weise kann Einfluss auf Bilder genommen werden, wie beispielsweise das Extrahieren von Bereichen oder automatisierte Filterung von Bilddaten.

Das Kommando und dessen Ausgabe Die Ausgabe der Prozesse werden über Kommandos generiert. Wie in Abschnitt 3.3.3 eingeführt, gibt es für Geräte verschiedene Antwortklassen: Kommandostatus, Fehler, Konfiguration und Daten. Auch das System beherrscht dieselben Klassen, wobei alle die Gemeinsamkeit haben, dass sie über die System-Writer ausgegeben werden. Gerätekommandos hingegen können ihre Antworten, entweder über die System- oder über eigene Writer ausgeben[13], je nachdem, ob die Antwort einen Einfluss auf das System hat (z.B. Gerätestatus, Konfigurationen) oder nicht (Daten).
Zusätzlich sei zu beachten, dass nicht alle Kommandos antworten müssen, aber dennoch *jeder* Prozess *genau ein* Kommando für die Antwort besitzen muss, da ansonsten der vom Benutzer gesendete Request nicht direkt beantwortet würde und somit entweder ein Timeout auftritt bzw. die Anfrage absichtlich abgebrochen werden muss (z.B. bei TCP als Transportprotokoll). Übernimmt nun ein Kommando die Rolle des Beantwortens, fällt bei Betrachtung eines Prozesses auf, dass zweierlei Ausgaberouten möglich sind: Eine Ausgabe über das System an den Benutzer oder eine Ausgabe über Geräte. Da der Ablauf in beiden Fällen derselbe ist (vgl. Abbildung 3.5), ist es naheliegend, das Kommando entscheiden zu lassen, über welche Route eine Antwort ausgegeben werden soll, und nicht den auslösenden Prozess; denn dieser weiß letztendlich nicht, von wem die aktuell abgerufene Information stammt und an wen sie adressiert ist. Dies hat zur Folge, dass der Excalibur Server selbst als Gerät definiert wird und er dieselben Eigenschaften besitzt wie ein typisches Gerät. Auf diese Weise kann das Kommando Antworten zur Verfügung stellen und nur über den Empfänger festlegen, wohin sie letztendlich weiter-

[13] Es sollten allerdings immer Systemwriter verwendet werden, da ansonsten andere Prozesse keine Möglichkeit haben, Daten über die Geräte-Writerkaskade weiterzugeben. Nur das Gerät, welches seine eigenen Writer definiert, hat Zugriff auf diese.

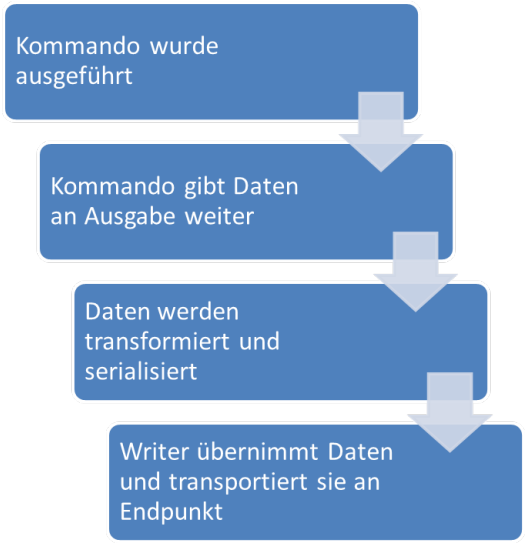


Abbildung (3.5)
Ausgabe eines Kommandos über Serialisierung, Transformation und Transport innerhalb des Systems.

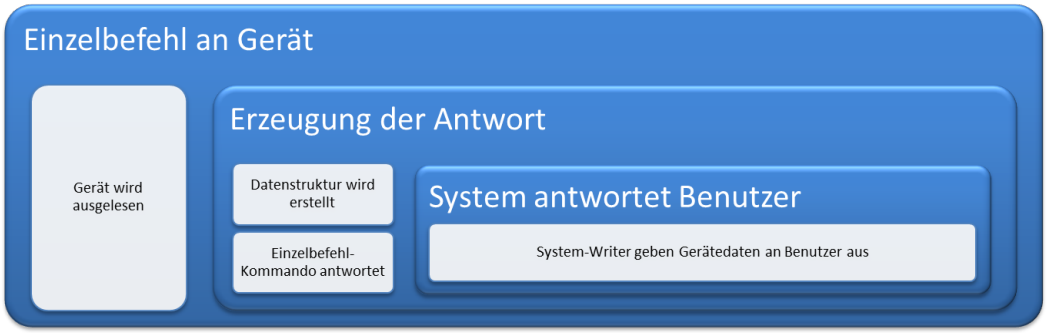


Abbildung (3.6)
Die Route der Antwort ausgehend von einem Geräte-Einzelkommando. Das Kommando wird auf dem Gerät ausgeführt, Daten werden vom Gerät erzeugt und diese an die System-Writern übergeben, damit diese den Benutzer die Daten weitergeben.

geleitet werden sollen. Die Datenstruktur selbst wird dabei nicht verändert und auch nicht das Datenflussmodell, welches einfach für beide Routen unterschiedliche *Writer*-Kaskaden zur Verfügung stellt.

Dies soll mit Abbildung 3.6 und 3.7 erläutert werden. Wird ein Geräte-Einzelkommando

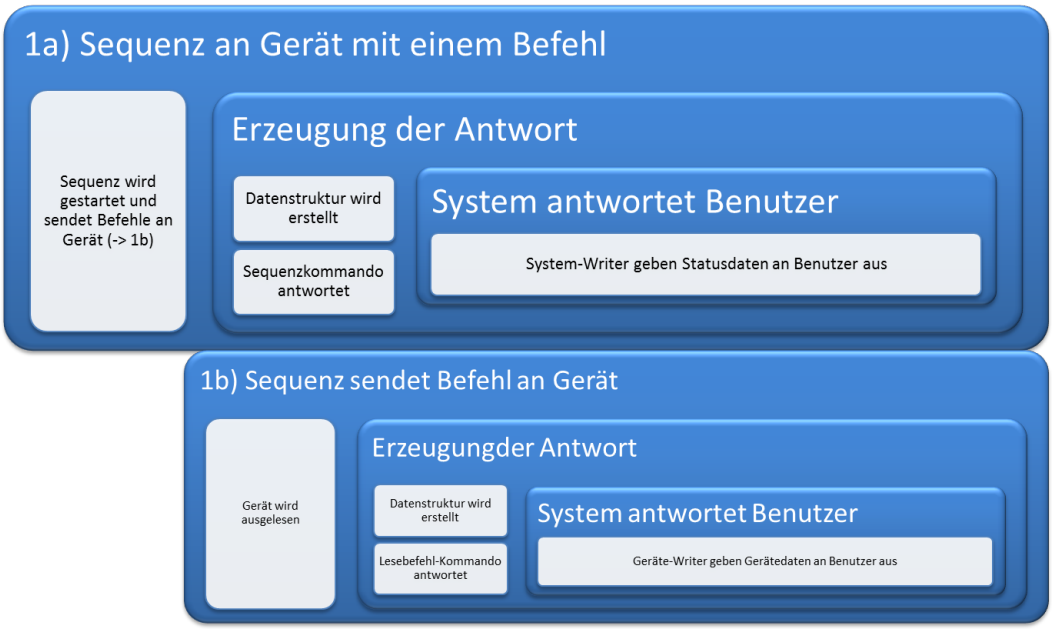


Abbildung (3.7)
Die Route der Antwort ausgehend von einem Sequenz-Kommando mit einem enthaltenen Geräte-Kommando. Das Sequenz-Kommando startet eine Sequenz, erzeugt die Daten (dass sie momentan ausgeführt wird) und übergibt sie den System-Writern, damit diese den Benutzer von der Ausführung informieren. Währenddessen erzeugt die Sequenz Geräte-Kommandos (hier: eines), welche auf dem Gerät ausgeführt werden, die vom Gerät erzeugten Daten übernimmt und an die Geräte-Writer übergibt. Diese speichern die Daten nun wie in deren Writern angegeben.

an Excalibur gesendet, wird das korrespondierende Prozess-Kommando (eine Komposition aus einem Schreib- und einem Lesekommando) auf dem Gerät ausgeführt. Die dabei produzierten Daten werden vom Prozess-Kommando in einer Datenstruktur gespeichert und diese mit dem System als Empfänger in eine Antwort-Warteschleife gestellt. Ein Thread liest diese Warteschleife aus, lädt die Writer des Empfänger-Geräts (hier: das System) und ruft deren Schreib-Befehl auf, welche anschließend die Daten transformieren, serialisieren, über die den Writern repräsentierenden Transportmechanismen ausgeben und den Benutzer somit die Daten des Gerätes zurücksenden.
Wird hingegen ein Sequenz-Kommando gesendet[14] (welches hier im einfachsten Fall ein

[14]Sequenzen sind Abläufe bestehend aus Geräte- und System-Befehlen, die der Reihe nach (oder auch parallel) ausgeführt werden. Ein Sequenz-Kommando ist entsprechend ein auszuführender Befehl

identisches Geräte-Kommando enthalten soll wie das Geräte-Einzelkommando oben), wird das korrespondierende Prozess-Kommando (eine Programm-Kaskade wie in Abschnitt 3.3.6 erklärt wird) erzeugt und in einem separaten Thread ausgeführt. Das Prozess-Kommando speichert die Daten, dass die Sequenz nun ausgeführt wird in obiger Datenstruktur und stellt diese mit dem System als Empfänger in eine Antwort-Warteschleife. Ein Thread liest nun diese Warteschleife aus, lädt die Writer des Empfänger-Geräts (hier: das System) und ruft deren Schreib-Befehl auf, welche anschließend die Daten transformieren, serialisieren und über die den Writern repräsentierenden Transportmechanismen ausgeben und den Benutzer somit mitteilen, dass die Sequenz ausgeführt wird. Die Sequenz führt nun sequentiell alle Teile der Programm-Kaskade aus. Jedes Geräte-Kommando wird nun auf dem Gerät ausgeführt und die dabei entstehenden Daten werden in einer Datenstruktur gespeichert, welche mit dem *Gerät* als Empfänger in dieselbe Antwort-Warteschleife gestellt wird. Derselbe Thread liest die Warteschleife aus, lädt die Writer des Empfänger-Geräts (hier: das ausführende Gerät) und ruft deren Schreib-Befehl auf, usw.
Es ist deutlich zu erkennen, dass unabhängig davon, auf welche Weise ein Prozess oder dessen Kommandos ausgeführt werden, die anfallenden Antwortdaten immer in derselben Warteschleife eingestellt und anschließend von demselben Prozess weiterverarbeitet werden. Da das Excalibur-System und die Geräte dieselben Eigenschaften und Operationen implementieren, ist nur der Adressat ausschlaggebend, welche Route für die Antwortdaten eingeschlagen wird.

Die Ausgabe über Antworten Im letzten Kapitel ist die Route von Antworten eingeführt worden. Zwar kann prinzipiell jede produzierte Antwort(klasse) über jede Route umgeleitet werden, wobei allerdings nur gewisse Kombinationen Sinn ergeben. Dies gilt natürlich nicht nur für Antworten der Klasse „Daten“, sondern ebenso für die Klassen „Konfiguration“ und „Status“, wobei die beide letzteren fast ausschließlich über die System-Writer ausgegeben werden. Aus diesem Grund soll im Folgenden auf die System-Route eingegangen werden. Unabhängig von der Antwortklasse befinden sich die Daten nun in einem Writer, welcher die Antworten über einen Thread aus der Antwort-Warteschleife erhalten hat. Da jeder Writer einen Transportmechanismus repräsentiert, wird am Ende eine Antwort über ebendiesen nach Extern zur Verfügung gestellt. Der Serialisierer bestimmt dabei, in welchem Format die Daten das Excalibur-System verlassen (z.B. XML, JSON, formatiertes ASCII, usw.), weshalb für ihn die Antwortklasse eine

innerhalb der Sequenz.

fundamentale Rolle spielt und jene nur auf bestimmte Klassen sinnvolle Formatierungen generieren können.

Kategorien: Konfigurationen & Status

Für sowohl JSON- als auch XML-Serialisierer sind vor allem Konfigurationen und Statusklassen sehr gut verarbeitbar. Da es sich bei Konfigurationen um klar strukturierte, meist im ASCII/UNICODE-Format vorliegende Datensätze handelt, können diese auf sehr einfache Weise mit XML/JSON Tags eingehüllt und zurückgegeben werden. Dasselbe gilt für auch Status-Antworten, zu denen auch die Fehler zählen, welche z.B. bei ungültigen Anfragen oder Gerätefehlern auftreten können. Letztere werden immer über die System-Route ausgegeben und unterbinden in schwerwiegenden Fällen sogar die Route über die Laborgeräte.
Gerätedaten sind mit den beiden auf Text basierenden Serialisierer schwerer verarbeitbar, vor allem, wenn es sich bei den generierten Daten um binäre Formate handelt. Beide Sprachen unterstützen nativ keine binäre Daten. Es gibt aber indirekte Möglichkeiten, Binärdaten mit Hilfe dieser beiden Sprachen zu übertragen. Zum einen lassen sich die Daten kodieren, z.B. über Base64 oder BCD(Binary coded decimals), was allerdings die Datenmenge stark vergrößert. Zum anderen gibt es die Möglichkeit sog. MIME-Boundaries zu verwenden, womit innerhalb von Excalibur ausschließlich gearbeitet wird (s. Glossar). Es ist naheliegend, die Verwendung von XML- und JSON-Serialisierern für den Transport von Gerätedaten einzuschränken, da es sich um textbasierende Notationen handelt und binäre Daten nur über Umwege verwendet werden können.[15] Die Stärken der beiden Serialisierer liegen vor allem im Transport von Konfigurationen und Statusmitteilungen.

Datenausgabe in Geräten Datenausgabe über die Geräteroute verhält sich prinzipiell gleich wie die Ausgabe über die System-Route, außer dass normalerweise andere Writer/-Serialisierer zum Transport herangezogen werden. Für die Geräteroute spielen vor allem

Kategorie: Daten

Antworten der Klasse „Daten“ eine große Rolle und damit des Template-Serialisierers. Er ist dafür entwickelt worden, um Gerätedaten anhand von einer vom Benutzer definierten Formatierungsanweisung zu strukturieren und mit zusätzlichen Metadaten zu bestücken. Statusantworten bzw. Konfigurationen werden nicht über die Formatierungsanweisung erfasst; sie führen sogar im Fall des Status „Fehler“ zu einem Aussetzen der Serialisierung. Der Template-Serialisierer ist deshalb eigentlich nicht für System-Writer bestimmt und sollte dort auch nicht eingesetzt werden.

[15] Teilweise wird dennoch auf Gerätedaten im Rahmen des Einzel-Gerätekommandos und beim Abruf von Daten aus Puffern mit dem Mime-Mechanismus gearbeitet

Umgekehrt ergeben für die Geräteroute JSON- bzw. XML-Formatierungen nur Sinn, wenn es mehrere Excalibur Server in einem Netzwerk gibt (s. Ausblick in Abschnitt 4.2). Um Daten formatiert in Datensenken abzulegen, sind diese Serialisierer eher ungeeignet, da der Anteil an Nutzdaten im Vergleich zu den sprachlich bedingten Notations-Daten klein ist und somit ein hoher Overhead resultiert, der in den meisten Fällen für eine permanente Speicherung nicht relevant ist. Sollen dennoch JSON oder XML Daten über Geräte beispielsweise in eine Datei geschrieben werden, sollte auch hierfür ein Template-Serialisierer verwendet werden (s. Templates in Abschnitt 3.3.7).
Gerade für Daten spielen aber noch andere Konstrukte eine große Rolle: die Transformer. Abhängig davon, welcher Datentyp erwartet wird und welche Transformation durchgeführt werden soll, können dem Writer, wie oben erklärt, eine beliebige Anzahl Transformer angehängt werden. Anhand ihrer Konfiguration wenden sie nun Skripte, mathematische Operationen oder Bildverarbeitungsroutinen auf die Daten an. Dabei kann sowohl die Datenstruktur als auch -Menge stark verändert werden. Transformationen ergeben allerdings nur Sinn, wenn sie für jeden Ablauf dieselbe Aktion durchführen müssen z.B. Mittelwertbildung, Bildausschnitte extrahieren, Referenzierungen, Differenzierungen usw. Im Gegensatz dazu sollten statistische Berechnungen oder andere Operationen, wessen Ausführung von den Daten selbst abhängen, erst im Anschluss offline erfolgen. Im Umkehrschluss bedeutet es, dass Transformationen nur mit automatisierbare Datenverarbeitungroutinen zu verwenden sind, bei welcher während eines Messablaufs keine Benutzereingabe erforderlich sein darf.

Datenkanäle und -verarbeitung

Es ist gezeigt worden, dass Geräteantworten von Kommandos erzeugt und über eine von zwei Routen weitergegeben werden. Treffen diese Daten nun auf Writer, Transformer und Serialisierer, werden sie nach der dort hinterlegten Konfiguration modifiziert und an den korrespondierenden Endpunkt weitergeleitet. Bei näherer Betrachtung dieses Prozesses fällt auf, dass alle (!!) Antworten dieselbe Behandlung erfahren, d.h. zwei Geräteantworten für zwei unterschiedliche Befehle (z.B. eine Messung und eine Parameterabfrage) folgen denselben Weg durch die Writer-Kaskade. Dieses Vorgehen kann bereits aus Gründen des Datentyps zu Problemen führen (1. Fall: Liste aus Zahlen für z.B. ein Spektrum, 2. Fall: Eine einzige Zahl für beispielsweise die Integrationszeit).
In anderen Worten kann der Writer die Daten nicht interpretieren und muss über derer Zweck und Bestimmungsort aufgeklärt werden. Für diese und ähnliche Fälle werden

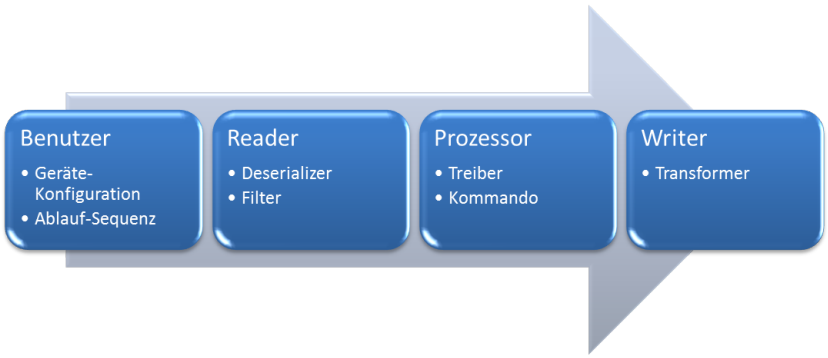


Abbildung (3.8)
Abfolge der Sub-Prozesse, welche Veränderungen an den Kanälen eines Datenstroms nehmen können.

Datenflussanweisungen benötigt, um die Datenströme und deren Wege regeln; sie sollen im Folgenden als *Kanäle* bezeichnet werden. Ein Kanal ist nicht anderes als ein *Tag*, welcher an einen Datenstrom angehängt wird und ihn somit für einen bestimmten Pfad kennzeichnet. Jeder Datenstrom kann sich in beliebig vielen Kanälen befinden (wobei nur eine Kopie der Daten vorliegt). Jeder Writer, Transformer und Serialisierer lauscht nur auf gewisse Kanäle, welche für jede einzelne Instanz individuell konfiguriert werden können. Ist kein Kanal konfiguriert, lauscht die jeweilige Instanz auf allen Kanälen. Auf diese Weise können Kaskaden an Writern/Transformern/Serialisierern konstruiert werden, wobei ein bestimmter Datenstrom mit Hilfe der Kanäle auf ganz bestimmte Pfade durch die Hierarchie gelenkt werden kann und somit auch die Ausgabe geregelt an einen bestimmten Endpunkt transportiert wird. Um dies zu bewerkstelligen, kann jeder Sub-Prozess, welchen der Datenstrom passiert, Kanäle setzen, umleiten oder löschen (s. Abbildung 3.8).

Datenkanäle und Zielvariablen Datenströme sind dynamische Datenflüsse, welche entweder beim Reader oder am Gerät (per Lese-Kommando) erzeugt werden. Einmal erzeugt, werden sie durch alle Prozesse ohne Zwischenspeicherung bis zur Ausgabe transportiert. Das bedeutet, dass ein Sub-Prozess (z.B. ein Skript), welcher Daten aus einem Datenstrom extrahieren möchte, dies zu dem Zeitpunkt tun muss, wenn er gerade diesen Sub-Prozess passiert. In anderen Worten bedeutet es, dass eine Verrechnung der Daten aus zwei Datenströmen miteinander per se nicht möglich ist, da sie einen Sub-Prozess zu unterschiedlichen Zeitpunkten passieren. Derartige Berechnungen sind hingegen wich-

tig, da Geräteantworten durchaus zu einem späteren Zeitpunkt wiederverwendet werden müssen: Exemplarisch anhand des Fluoreszenz Spektrometers aus Abschnitt 2.1.2 soll ein Referenzspektrum aufgenommen werden und alle folgenden aufzunehmenden Spektren durch diese Referenz geteilt werden. Dieser Fall kann allein mit Kanälen nicht abgedeckt werden, da die Referenz sehr viel früher aufgenommen worden ist als das Spektrum. Für diesen und ähnliche Fälle gibt es die sog. *Zielvariablen*, auf welche überall innerhalb von Excalibur zugegriffen werden kann. Sie sind nichts weiteres als Speicherbereiche, welche in einem Pool abgelegt werden und von dort jederzeit wieder abrufbar sind. Dies eröffnet eine dritte Route, um Daten weiterzugeben: Das Ablegen der Daten innerhalb von einem Cache zur späteren Wiederverwendung.

Datenverarbeitung Die eingeführten Datenausgabe-Mechanismen führen zu sehr flexiblen Möglichkeiten, von Geräten erzeugte Daten zu organisieren. Mit ihrer Hilfe ist es möglich, Daten geregelt zu bearbeiten, dem Benutzer zu Verfügung zu stellen und sie in Datensenken abzulegen. Über die verschiedenen Serialisierer können Datenklassen organisiert und für verschiedene Ausgabe-Endpunkte vorbereitet werden, je nachdem welcher Einsatzzweck mit den erzeugten Daten verbunden werden soll. Über die Datenkanäle können die Ausgabewege spezifiziert werden und die konfigurierten Endpunkte unabhängig voneinander gewählt werden, während mit Hilfe der Zielvariablen Daten zwischengespeichert werden können, um sie nachträglich in Skripten oder Templates wiederverwenden zu können.
Durch diese vielseitigen Möglichkeiten können Daten aber nicht nur durch den Excalibur-Prozess geleitet oder abgelegt werden. Auch grafische Oberflächen lassen sich auf diese Weise einfach mit Werten füllen, indem Steuerelemente einfach auf verschiedene Kanäle gelegt werden. Mit entsprechenden Befehle können anschließend die Daten geändert wieder an den Excalibur Server zurückgesendet werden.

Skripte und mathematische Berechnungen

Daten werden von den Geräten meist in einer nicht aufgearbeiteten Rohform geliefert und sind auf diese Weise nur bedingt direkt einsetzbar. Damit ist gemeint, dass, wie zu Beginn in der Problemanalyse eingeführt, Sensoren innerhalb einer Sensorumgebung nur sinnvoll erscheinen, wenn die von ihnen gelieferten Daten einem mathematischen Modell unterworfen werden, welches die zu messende Änderung der physikalischen/chemischen Eigenschaft beschreibt. Dies ist als Abhängigkeit zwischen Sensor und Steuerungssoftwa-

re bezeichnet worden. Das Lösen dieser Abhängigkeit soll in diesem Kapitel besprochen werden. Unabhängig vom mathematischen Modell können Daten (vor allem Binärdaten) auch in falschen bzw. nicht interpretierbaren Datenformaten vorliegen (z.B. falsche Bytereihenfolgen, invertierte Bits, unpassende Byte-Anzahl, usw.) und sie müssen vor der eigentlichen Verarbeitung zuerst in ein passendes Datenformat konvertiert werden.
Für diese und ähnliche (automatisierbare) Fälle können die im Folgenden als *Skripte* bezeichneten mathematischen Routinen verwendet werden. Skripte spielen für alle Zielgruppen eine wichtige Rolle. Systemprogrammierer können Skripte verwenden, um komplexe mathematische Routinen während der Plugin- oder grafischen Oberflächenentwicklung auf Daten anzuwenden. Systemintegratoren können Skripte zur Bearbeitung von Gerätendaten innerhalb von Treibern oder Transformern einsetzen, während letztendlich der Benutzer Skripte für Datenbearbeitungen innerhalb von Ablaufsequenzen einsetzen kann. Weiter unter wird jedoch genauer auf die Anwendbarkeit von Skripten für die einzelnen Zielgruppen eingegangen.
Das nächste Unterkapitel gibt eine Übersicht der vorhandenen Möglichkeiten wieder, das darauf folgende geht auf die Einsatzpunkte und -möglichkeiten von Skripten ein.

Übersicht und Funktionsbeschreibung Es werden 5 Arten von Skripten unterschieden (in Klammer: Befehlsfolge):

- Formeln (formula)
- Sequenzen (sequence)
- Octave (octave)
- Python (python)
- VI (vi)

Skripte funktionieren alle auf eine ähnliche Weise: Es gibt eine Skript-Struktur, über welche textuell eine Abfolge an Skript-Befehlen angegeben werden kann. Innerhalb der Berechnungen kann auf Kanäle, Zielvariablen und lokale Variablen zugegriffen, sie können zur Berechnung herangezogen und es können Daten wieder in ihnen gespeichert werden. Dazu wird eine universelle Struktur für Skripts definiert, die einige fundamentale Eigenschaften hat (s. XML Schema für Skripts in Anhang B.6.2). Die Anwendung des Skripting-Mechanismus ist bereits anhand der reflektrometrischen Interferenzspektroskopie in Abschnitt 2.3.2 gezeigt worden.

Im Folgenden wird kurz auf die einzelnen Skript-Arten eingegangen:
Die *Formeln* entsprechen weitestgehend den LabVIEW®-Formelknoten und bilden einen Wrapper für diese. Vorteil von ihnen ist eine sehr performante Verarbeitung aber auf Kosten der Funktionalität. Hiermit können nur die wirklich fundamentalsten Berechnungen durchgeführt werden. Ein genaue Beschreibung der Funktionlität kann unter[16] nachgelesen werden.
Sequenzen sind zur Verrechnung aufeinander folgender Daten derselben Quelle eingeführt worden. Während Formeln auf Datenströme „online“ angewendet werden, d.h. jeden Datensatz einmalig verarbeiten und dann weiterleiten (z.B. Glättungen, Referenzierungen, usw.), stellen Sequenzen Funktionalitäten zur Verfügung, um verschiedene nacheinander folgender Datenströme miteinander zu verrechnen oder zu reduzieren (z.B. Mittelwertbildung, Median, usw.).
Sollen komplexere mathematische Berechnungen durchgeführt werden, kann auf die Funktionalität von externen mathematischen Bibliotheken zugegriffen werden. Dafür ist das *Octave*-Skript eingeführt, welches ein vom Benutzer definiertes Skript an einen externen GNU Octave®-Prozess weitergibt. Nähere Informationen zu GNU Octave® finden sich in Abschnitt 3.5.3.
Noch komplexere Berechnungen, welche eine vollständige Programmierumgebung und komplexe Datenverarbeitungsroutinen bedingen, können entweder mit Hilfe des *Python*- oder *VI*-Skripts (und damit LabVIEW®-VIs realisiert werden). Hiermit kann auf den vollen Funktionsumfang der Programmiersprachen Python bzw. LabVIEW® zugegriffen werden.

Einsatzgebiete von Skripten Skripte können auf von Geräten produzierte Daten wirken. Abhängig vom Anwendungsfall (aber auch von der Zielgruppe), gibt es mehrere mögliche Positionen innerhalb des Excalibur Server, an welchen Skripte eingesetzt werden können. Da es sich in vielen Fällen beispielsweise beim Schreiber eines Treibers um eine andere Person handelt als beim Schreiber einer Ablaufsequenz, gibt es Möglichkeiten für beide Personengruppen, mit eigenen Skripten auf die Daten einzuwirken.
Ein Skript im Treiber stellt für die spätere Anwendung Basisberechnungen zur Verfügung (s. Listing 3.3.3). Mit Hilfe der Kanäle und Zielvariablen können dann verschiedene Berechnungen über unterschiedliche Routen, dem System zur Verfügung gestellt werden. Damit sind beispielsweise Formatumwandlungen, Referenzierungen oder Änderungen der

[16] http://zone.ni.com/reference/en-XX/help/371361J-01/gmath/eval_formula_node/
und http://zone.ni.com/reference/en-XX/help/371361J-01/lvhowto/formula_node_and_xpress/

Byte-Reihenfolge direkt nach dem Auslesen des Gerätes durchführbar, ohne welche auf die benötigte Information innerhalb dieser Daten möglicherweise nicht zugegriffen werden kann. Systemintegratoren und Benutzer können Skripte in Ablaufsequenzen einsetzen (s. Abschnitt 3.3.6), um Daten für die Ausgabe vorzubehandeln (z.B. Glättungen, Datenreduktionen, usw.). Diese Position ist relevant, da die Ablaufsequenz jederzeit für jede Zielgruppe veränderbar ist und Skripte die Daten abhängig vom vorliegenden Assay anpassen können. Das gilt nicht für die Datentransformer, welche im Normalfall vom Systemintegrator konfiguriert werden. Die Definition der Skripte im Datentransformer sollte der bevorzugte Weg sein, sobald Skripte allgemeingültig für eine Methode, aber unabhängig vom zu messenden Assay eingesetzt werden sollen. Die Ablaufsequenz verliert dadurch an Komplexität und kann beliebig verändert werden, solange die zugrunde liegende Messmethode und deren Auswertung die gleiche bleibt. Auf diese Weise wird eine Trennung zwischen Ablauf und Datenverarbeitung realisiert.

3.3.6. Ablaufsteuerung

Dieses Kapitel wird sich mit der Struktur und Syntax von Ablaufsequenzen beschäftigen. Abläufe spielen die wohl wichtigste Rolle für eine Steuerungssoftware, da hiermit die Reihenfolge des Aufrufs und die Ausführungsdauer von Gerätekommandos festgelegt werden. Auf den ersten Blick scheint hier auch der Hauptteil der Komplexität zu finden zu sein, was allerdings nur bedingt richtig ist. Bei näherer Betrachtungsweise ist zwar die interne Prozessierung komplex, die äußere Darstellung aus Sicht des Benutzers ist aber sehr einfach gehalten. Doch diese Einfachheit ist gewünscht, denn die Konfiguration von Abläufen soll durch einen Benutzer oder Systemintegratoren erfolgen und nicht durch Systemprogrammierer mit fundierten Programmierkenntnissen. Ziel ist es, dass sich ein Entwickler von Sequenzen rein prinzipiell nur folgende Fragen stellen muss, um Ablaufsequenzen zu verfassen:

- Welches Gerät soll welche Aktivität ausführen?
- Wann soll diese Aktivität ausgeführt werden?
- Wie lange soll sie ausgeführt werden?
- Wie oft soll sie ausgeführt werden?

Wer diese Fragen eindeutig beantworten kann, ist prinzipiell in der Lage, eine Ablaufsequenz zu erstellen, zumindest wenn ein grundsätzliches Verständnis für den vorliegen-

den Messaufbau vorhanden ist, auf welchen die Ablaufsequenz angewendet werden soll. Wie in den vorangegangenen Kapiteln gezeigt worden ist, spielt die Datenspeicherung, -verarbeitung und die -ausgabe hierbei keine Rolle.[17] Wie es die *Methoden* in Abschnitt 3.3.8 zeigen werden, spielen für die Ablaufsequenzen nicht einmal die zugrunde liegenden Geräte eine große Rolle, solange ihr Funktionsprinzip dem System bekannt gemacht wird. Anhand des folgenden Beispiels sei eine kleine Ablaufsequenz vorgestellt:

Listing (3.13)

Minimaler Messablauf für ein direktes Testformat, bei welchem eine Probe aus einem Vorrat direkt und ohne Zwischenschritt über einen Sensor geleitet wird

```
1 Ventil: schalte auf Position 'Sample'
2 Peristaltikpumpe: schalte an mit 35 µL/s
3 Spektrometer: starte Messung jede 100 ms
4 Warte 60 s
5 Ventil: schalte auf Position 'Buffer'
6 Warte 300 s
7 Spektrometer: stoppe Messung
8 Peristaltikpumpe: schalte aus
```

Listing 3.13 zeigt den Minimalfall einer Messung eines Assay im direkten Testformat, d.h. eine Probe wird direkt aus einem Vorrat über einen Sensor geleitet und ein Signal aufgenommen. Das gezeigte Beispiel bedarf eigentlich keiner Erklärung, denn die dort gezeigte Syntax ist zwar proprietär, aber dennoch verständlich. Es gilt jetzt diese „umgangssprachliche“ Ausdrucksweise zu standardisieren und in eine etwas allgemeingültigere Syntax zu überführen, aber dabei die Lesbarkeit nicht zu gefährden. Dazu wird jede Zeile in einem ersten Schritt in Pseudo-Code (Python Stil) überführt:

Listing (3.14)

Meta Syntax für Listing 3.13, nach welcher direkt in eine Auszeichnungssprache übersetzt werden kann

```
1 def program:
2  sample=1
3  buffer=2

5 _____
```

[17] Es sei denn für den zu messenden Assay sollen ganz spezielle Auswertungsroutinen innerhalb der Ablaufsequenz definiert werden (was in den seltensten Fällen zutrifft)

```
4  valve(position=sample)
6  peristaltic(speed=35)
7  start(spectrometer(interval=0.1))
8  wait(60)
9  valve(position=buffer)
10 wait(300)
11 stop(spectrometer())
12 peristaltic(off)
```

In einem nächsten Schritt wird dieser Pseudo-Code in eine Auszeichnungssprache[18] übersetzt:

Listing (3.15)

Listing 3.13 in XML, welches als Anfrage an den Excalibur Server gesendet werden kann

```
1  <program>
2   <var name="sample" value="1"/>
3   <var name="buffer" value="2"/>
4   <command device="valve" name="valve" position="$sample"/>
5   <command device="peristaltic" name="pump" speed="35"/>
6   <start device="spectrometer" name="measure"
7     interval="0.1"/>
8   <wait value="60"/>
9   <command device="valve" name="valve" position="$buffer"/>
10  <wait value="300"/>
11  <stop device="spectrometer name="measure"/>
12  <command device="peristaltic" name="pump" value="off"/>
13 </program>
```

Wie in Listing 3.15 ersichtlich, ist die Lesbarkeit der Sequenz erhalten geblieben (auch wenn der Overhead durch die XML Notation gestiegen ist), wodurch die Syntax sehr einfach zu erlernen ist. Zu sehen ist auch, dass nur die Reihenfolge und Ausführungsdauer festgelegt werden muss, die Ausgabe der Geräte in der Ablaufsequenz hingegen keine Rolle spielt. Darüber hinaus hat XML (und ggf. auch JSON) den Vorteil, dass

[18] Hier werden die Resultate exemplarisch nur in XML gezeigt, das JSON-Format wird entsprechend gebildet.

es sich sehr einfach parsen bzw. generieren lässt und sich auf diese Weise, sehr einfach (automatisiert) Codegerüste in beliebigen Programmiersprachen entwickeln lassen. Anhand dieses Prinzips wird in den folgenden Kapiteln auf die Einzelelemente und deren Funktionsweise eingegangen. In Abschnitt 2.3.2 und 2.3.3 sind bereits umfangreichere Beispiele gezeigt worden.

Gerätekommandos und Kompositionen

Zur sprachlichen Unterscheidung der Sequenz-Kommandos von den Kommandotypen des Treibers, wird im Folgenden die Bezeichnung ***Ablaufkommando*** *für erstere verwendet.*
Wie es Listing 3.15 zeigt, werden Ablaufkommandos mit dem Schlüsselwort *command* in XML eingeleitet; für JSON gilt ähnliches. Dies gilt für alle Positionen innerhalb von Excalibur, an welchen Gerätebefehle gesendet werden können. Über dieses Schlüsselwort weiß der Excalibur-Prozessor, dass *genau ein* Befehl aus dem Treiber an ein Gerät gesendet werden soll. Welches Gerät angesprochen wird, spezifiziert die Eigenschaft *device* der *command*-Struktur; welches Kommando aus dem Treiber referenziert wird, spezifiziert die Eigenschaft *name*.[19]

Kommandos

Wie in Abschnitt 3.3.3 erklärt, können in jedem Kommandotyp des Treibers Value-Tokens spezifiziert werden, welche als Platzhalter für Benutzereingaben dienen. Die erwarteten Werte hierfür werden über die Eigenschaft *value* Ablaufkommando übergeben; mehrere Werte werden entsprechend mit einer komma-separierten Liste übergeben. Da die Reihenfolge der Parameter eine wichtige Rolle spielt, kann dies vor allem bei vielen Eingabeparametern zu Komplikationen führen. Aus diesem Grund wurden die bereits besprochenen Aliase für Value-Tokens im Treiber eingeführt, welche eine Möglichkeit darstellen, Eingabeparameter mit einem Namen zu versehen und somit als Eigenschaft für das Ablaufkommando zur Verfügung zu stellen (s. Eigenschaft „position“ im „valve“-Kommando in Listing 3.15). *Kompositionen* wie sie in Abschnitt 3.3.3 eingeführt worden sind, unterscheiden sich innerhalb der Ablaufsequenz nicht von den einfachen Kommandos. Einzig die Anzahl der Eingabeparameter müssen an alle Unterkommandos einer Komposition angepasst werden, wobei sich die Einführung entsprechender Aliase an dieser Stelle lohnt.

Kompositionen

[19] Soll eine Zeichenkette wie angegeben an ein Gerät gesendet werden, ohne dass ein Kommandotyp im Treiber definiert ist, so wird die Eigenschaft *raw* anstelle von *name* verwendet.

Start- und Stop-Kommandos Ein Excalibur-Kommando, wie es in Abschnitt 3.3.3 definiert worden ist, wird bei einem Aufruf genau ein einziges Mal ausgeführt. Soll es innerhalb einer Sequenz wiederholt ausgeführt werden, müssen auch mehrere Aufrufe des Kommandos erfolgen. Für Kommandos mit Statuswechsel ist diese Vorgehensweise sehr sinnvoll. Sollen hingegen Messwerte über Polling abgefragt werden, müssen Kommandos wiederholt aufgerufen werden. In diesem Fall ist der Ansatz relativ aufwendig, da der Kommandoaufruf in einer Schleife oder einem Parallelprozess erfolgen muss. Für diesen Fall, steht ein mächtigeres Werkzeug zur Verfügung:

Listing (3.16)

XML Schema des Continuity Types

```
1  <complexType name="extendedContinuityType">
2   <sequence>
3    <element name="interval" type="float" minOccurs="0" />
4    <choice minOccurs="0">
5     <element name="start" type="tns:valueBasedType"/>
6     <element name="startCommand" type="tns:tokenType"/>
7    </choice>
8    <choice minOccurs="0">
9          <element name="stop" type="tns:valueBasedType"/>
10         <element name="stopCommand" type="tns:tokenType"/>
11   </choice>
12  </sequence>
13 </complexType>
```

Durch ein anderes Schlüsselwort kann veranlasst werden, dass ein Kommando automatisch in einem Parallelprozess nach Ablauf eines bestimmten Intervalls wiederholt aufgerufen werden kann, wenn es gewünscht ist; das Intervall kann dabei direkt im Treiber über den sog. Continuity-Abschnitt (s. Listing 3.16) oder zur Laufzeit in der Ablaufsequenz festgelegt werden. Der Unterschied zum einfachen Ablaufkommando-Aufruf per „command“ ist einzig das Schlüsselwort „start“ (s. Listing 3.15), wodurch automatisch ein Parallelprozess gestartet wird, der solange ausgeführt wird, bis der Prozess durch den „stop“-Aufruf wieder angehalten wird. Darüber hinaus ist es möglich, im Continuity-Bereich verschiedene Aufrufe für Start- und Stop-Prozess zu definieren, d.h. es können unterschiedliche Gerätekommandos zu Beginn, während der Wiederholung und zum Ende an das Gerät gesendet werden. Ist kein Continuity-Abschnitt im Treiber definiert und

gleichzeitig auch kein Intervall in der Ablaufsequenz angegeben, so wird das Kommando wie im Falle des Aufrufs über „command“ genau *einmal* ausgeführt (Intervall ist unendlich).
Für den Entwurf von Ablaufsequenzen existieren weitere Ablaufkommandos, von welchen auf die wichtigsten detaillierter eingegangen wird. Auf deren exakte Datenstruktur soll aber hier verzichtet werden. Sie wird mit Hilfe von XML Schemas im Anhang wiedergegeben (s. Anhang B.6.1).

Datenhaltungsstrukturen

Um während eines Ablaufs Daten zu speichern und wieder zur Bearbeitung abzurufen, können Variablen mit Hilfe der „programVariableGroup“ definiert werden. Direkt gespeichert werden können alle einfachen Datentypen und ihre zugehörigen Arrays, indem über die Eigenschaften „Name“, „Datentyp“ und „Wert“ alle benötigten Informationen direkt in der Ablaufsequenz spezifiziert werden. Eine andere Möglichkeit der Variablendefinition kann über die Eigenschaft *target* in einfachen oder „start“-Ablaufkommandos über die Angabe eines Variablennamens erfolgen. Jeder Variablenwert kann überall in der Ablaufsequenz über die Notation *$Variablenname* wieder abgerufen werden.

Ablaufstrukturen

Konditionale Strukturen Während eine Ablaufsequenz ohne weitere Angaben angefangen beim ersten Sequenzkommando bis zum letzten durchläuft, können über die konditionalen Strukturen der „programConditionGroup“ alternative Ablaufwege definiert werden. Dies wird beispielsweise über das Schlüsselwort „choose“ erreicht (s. Anhang B.6.1). Innerhalb eines „choose“ kann eine beliebige Anzahl „if“-Blöcken definiert werden, in welchen eine Bedingung angegeben werden und mit Hilfe der boole'schen Algebra die Ausführung oder Nicht-Ausführung eines Blocks gesteuert werden kann. Als letzte Alternative kann bei Nicht-Eintreffen aller If-Bedingungen ein Standard in Form eines „Else“-Blocks definiert werden.

Wiederholungsstrukturen Ablaufsequenzen und alle darin enthaltenen Blöcke werden ohne weitere Angaben ein einziges Mal ausgeführt. Sollen gewisse Abschnitte wiederholt werden, so kann dies mit Hilfe von Wiederholungsstrukturen in Form einer „program-

LoopGroup“ erfolgen. Dafür existieren zwei Möglichkeiten: Der „for“- und der „while“-Block. Sie verhalten sich äquivalent zu anderen Programmiersprachen, d.h. for-Schleifen werden solange ausgeführt bis eine vorher definierte Zähler-Variable einen Schwellwert erreicht und while-Schleifen bis eine vorher definierte Bedingung „TRUE“ wird (s. Anhang B.6.1).

Funktionen Zur Wiederverwendbarkeit von Blöcken an mehreren Stellen einer Ablaufsequenz ist die Möglichkeit gegeben, mit Hilfe der „programFunctionGroup“ Code-Abschnitte zu kapseln. Dadurch können sich wiederholende Abschnitte mit unterschiedlichen Werten aufgerufen werden, indem Übergabeparameter an die Funktion übergeben werden. Auch diese Funktionalität ist äquivalent zu anderen Programmiersprachen zu betrachten. Zu beachten ist, dass vorher auf definierte Variablen der „programVariableGroup“ (s. Abschnitt 3.3.6) nicht zugegriffen werden kann, es sei denn, sie sind mit der Eigenschaft „global“ gekennzeichnet (s. Anhang B.6.1). Funktionen sind bereits in Abschnitt 2.3.2 und 2.3.3 eingesetzt worden.

Verarbeitungsstrukturen

Daten können unabhängig davon, ob sie von Geräten über die „target“-Eigenschaft oder direkt über Variablen gesetzt worden sind, wieder innerhalb der Ablaufsequenz gelesen und weiterverwendet werden. Zusätzlich gibt es die Möglichkeit innerhalb einer Ablaufsequenz, auf die Daten einzuwirken und mit ihnen Berechnungen durchzuführen. Handelt es sich um Zeichenketten, können Stringverarbeitungs-Routinen auf sie angewendet werden. Die typischen Methoden werden in Anhang B.6.1 gezeigt.
Für Zahlen und Listen gibt es weiterhin die Möglichkeit Berechnungen mit Hilfe eines Scripts (s. Abschnitt 3.3.5) durchzuführen. Einfachere Berechnungen über Zuweisungen von Variablen, d.h. mit den LabVIEW®-Formelknoten berechenbaren, können mit Hilfe der „programVariableGroup“ (s. oben) vorgenommen werden.

3.3.7. Templating

Eine der Hauptaufgaben für ein Messsystem ist die Datenausgabe. Einerseits ist ein permanentes Monitoring wichtig, damit ein Benutzer der Steuerung seine Messung zur Laufzeit verfolgen und gegebenenfalls bei Problemen direkt reagieren kann. Anderer-

seits sollen Daten auch gespeichert werden, damit sie im Anschluss einer Messung (oder auch von anderen Prozessen während einer Messung) ausgewertet bzw. weiterverwendet werden können. Der Prozess der Datenspeicherung ist jedoch nicht einfach ein „Ablegen“ von unformatierten Daten in einer beliebigen Datensenke. Messdatenspeicherung bedeutet vielmehr, eine geregelte und formatierte Datenablage und das Anhängen von Metainformationen an die eigentlichen Datensätze. Da ein Messsystem aber nicht allein entscheiden kann, welche Metainformationen benötigt und wie bzw. wohin die Daten abgelegt werden sollen, muss von Fall zu Fall über eine individuelle Datenstruktur entschieden werden. Aufgrund verschiedener Steuerungslösungen für die einzelnen analytischen Aufbauten eines Labors und die unterschiedlichen Ansichten der Programmierer von Steuerungslösungen, können diese Datenformatierungs-Strukturen sehr unterschiedlich ausfallen und somit eine Auswertung seitens der Benutzer erschweren, da sie mit unterschiedlichen Datenformaten konfrontiert werden, teilweise sogar für dieselbe analytische Methode.
Um den Prozess der Datenausgabe und -formatierung zu vereinheitlichen wird hier ein Templating-System eingeführt und erklärt, womit Ausgabestrukturen individuell konfiguriert und die Datenformate auf die Laborumgebung zugeschnitten werden können. Dadurch kann eine einheitliche Datenstruktur definiert werden, die sich an die äußeren Bedingungen anpassen, ohne Änderungen an der Software vornehmen zu müssen. Verändern sich die äußeren Bedingungen z.B. durch die Einführung eines neuen Laborinformations-Managementsystems, können die definierten Templates einfach an die geänderten Umstände angepasst werden, ohne die Steuerungssoftware neu kompilieren zu müssen. Die Templates werden einfach in den in Abschnitt 3.3.5 eingeführten Template-Serialisierer eingebunden, womit erreicht werden kann, dass in unterschiedliche Datensenken mit unterschiedlichen Datenformaten geschrieben werden kann. Darüber hinaus können in den Templates Metainformationen definiert werden, z.B. Datum, Benutzer, Kommentare, Zielvariableninhalte usw., um die Daten für Suchalgorithmen verfügbar und auffindbar zu machen.
Das Template ist eine einfache XML-Datei, welche dem „template“-Schema gehorcht (s. Anhang B.7). Es ist wie eine HTML-Datei aufgebaut und besteht aus einem *Head* und einem *Body*. Im Head werden grundlegende Ausgabekriterien wie Ausgabestil (s. nächstes Kapitel) oder auch der Datenspeicherungstyp (s. Abschnitt 3.3.7) festgelegt. Im Body werden die Formatierungsbefehle erfasst, wobei für gewisse Metainformationen spezielle XML-Nodes existieren, wie z.B. user, method, comment oder text, usw. Gerätedaten werden in Tabellen abgelegt, welche über die folgende Syntax im Body spezifiziert

werden können.

Ausgabestile

Unter dem Ausgabestil soll die Notation verstanden werden, in welche die Daten inklusive deren Metadaten konvertiert werden. Zum einen ist eine reine Textausgabe möglich, um beispielsweise CSV-Daten oder ähnliche Klartext-Formate zu realisieren. Andererseits sind gemischte Text-Binär-Strukturen über das sog. TDMS Format realisierbar. Zuletzt kann in eine weiterverarbeitbare Notation wie XML oder SQL konvertiert werden.

Textausgabe Die einfachste Ausgabeform ist die Textausgabe. Dabei werden alle XML-Nodes des Templates durch ihren referenzierten Wert ersetzt und in die Ausgabe geschrieben. Auf diese Weise lassen sich die gängigsten Datenausgabeformate wie CSV- oder ähnliche Formate realisieren. Innerhalb des Templates werden alle Zeichen außerhalb der XML-Nodes genauso übernommen wie angegeben, auch alle Textabschnitte. Zeilenumbrüche müssen über das korrespondierende *
*-Tag angegeben werden. Innerhalb der Tabellen können über bestimmte XML-Attribute Eigenschaften wie Spalten- und Zeilenseparatoren, Datums- oder Zeitformate festgelegt werden. Wichtig hierbei ist, dass nur *vollständige Datensätze* gespeichert werden können, d.h. Daten einer Zeile sind abhängig und wenn eine Tabelle mit beispielsweise 5 Spalten mit verschiedenen Werten definiert wird, müssen erst alle Spalten einer Zeile mit Werten gefüllt werden, bevor die nächste Zeile angefangen wird. Unvollständige Datensätze führen zu einer nicht definierten Datenausgabe, sobald eine Messung beendet ist, da dem Template-Prozessor nicht bekannt ist, welche(r) Wert(e) ausgelassen worden ist/sind.

TDMS Ausgabe Unter TDMS wird ein von National Instruments eingeführtes Dateiformat verstanden[20], welches Daten sowohl in ASCII als auch binär speichert. Dadurch werden viele Vorteile von Datenspeicherungsoptionen gängiger Dateiformate in einem einzigen vereinigt (s. Abschnitt 3.2.2), unter anderem hohe Datenspeicherungsraten.

Für das TDMS-Dateiformat werden einige XML-Nodes des Templates mit vordefinierten Attributen von TDMS korreliert, während alle anderen Eigenschaften als benutzerdefinierte Attribute gespeichert werden. Template-Tabellen werden in den TDMS-Channel-Groups gespeichert, während die Spalten als TDMS-Channels eingefügt werden. Da-

[20] http://www.ni.com/white-paper/3727/de/

durch werden Daten in ihrem nativen Datenformat binär gespeichert. Für TDMS spielt die Vollständigkeit der Datensätze keine Rolle, da jede Spalte einen eigenen Kanal innerhalb der Channel-Group besitzt und ausschließlich zu dieser eine Abhängigkeit hat, aber nicht zu anderen Datensätzen anderer Spalten.

XML Ausgabe Sollen bestimmte Daten in Form von HTML oder XML ausgegeben werden, kann diese Ausgabeform verwendet werden. Über die Eigenschaft *id* des XML-Nodes des Templates können XML-Node-Names für die Ausgabe definiert werden. Auf diese Weise können interne Excalibur-Daten externen auf XML basierten Prozessen zur Verfügung gestellt werden. Handelt es sich bei den angegebenen Nodes um HTML-Nodes, kann die Ausgabe direkt als Webseite in einem beliebigen Browser dargestellt werden.

SQL Ausgabe Es ist auch eine Ausgabe als SQL-Befehlssatz möglich, bei welcher die Daten als SQL-insert Befehle gekapselt werden. Dabei muss beachtet werden, dass für jede Template-Tabelle eine eigene SQL-Tabelle angelegt wird, ebenso eine weitere für alle anderen Zeichen, welche außerhalb der Tabellen spezifiziert sind (Haupt-SQL-Tabelle). Es werden nur Daten innerhalb von XML-Nodes beachtet, wobei für jedes einzelne der Reihe nach Spalten im angeführten Datentyp erzeugt werden; alle Zeichen außerhalb von XML-Nodes werden ignoriert. Wie auch für die Textausgabe müssen die Datensätze vollständig sein, da SQL-basierte Datenbanken zeilenweise arbeiten. Jede Zeile enthält die ID der Haupt-SQL-Tabelle als Referenz zur Zuordnung.

Datenspeicherungstyp

Unter dem Datenspeicherungstyp kann angegeben werden, wie die einzelnen Tabellen über die im Writer[21] spezifizierte Datensenke verteilt werden. Es wird zwischen einer einfachen Datenausgabe, einer externen und einer Ausgabe in mehrere Datensenken unterschieden. Dabei ist es wichtig zu verstehen, dass es eine Haupt-Datensenke gibt, in welche alle Daten geschrieben werden, welche *nicht* in einer Tabelle spezifiziert sind, d.h. der Typ wirkt sich vor allem auf Tabellen innerhalb des Templates aus.

Einfache Datenausgabe Hierbei werden alle Daten in dieselbe Datensenke geschrieben, d.h. Informationen außerhalb der Tabellen, wie auch die Tabelle selbst, werden so

[21] Beim SQL-Ausgabestil wird diese Option ignoriert

geschrieben, wie im Template spezifiziert.

Auslagerung in externe Datensenke Bei externen Tabellen werden Daten nicht in die Haupt-Datensenke geschrieben, sondern es wird eine neue Datensenke für die Tabelle angelegt. Dies entspricht weitestgehend der Speicherung der SQL-Ausgabe, in welcher Haupt- und Datentabellen getrennt sind.

Auslagerung in mehreren Datensenken Die Auslagerung in mehreren Datensenken entspricht im Großen und Ganzen derjenigen der Auslagerung in externen Tabellen. Hierbei wird jedoch nicht für jeden (vollständigen) Datensatz eine neue Zeile angelegt, sondern eine neue Datensenke geöffnet und die letztere geschlossen. Das ist nützlich, wenn jeder Datensatz beispielsweise Bildern oder ähnlich (großen) eigenständigen Datensätzen entspricht.

Einfluss der Templates auf die Writer Ausgabe

Jedes Template wird einem Template-Serialisierer angehängt, welcher allerdings als Serialisierer für jeden Writer dienen kann. Allerdings interpretieren die verschiedenen Writer Templates auf unterschiedliche Weise.

Ausgabe des File-Writers Für den File-Writer gelten alle obigen Erklärungen für Templates. Als Datensenken dienen Dateien auf der lokalen Festplatte. Über den Ausgabestil kann festgelegt werden, welche Art von Datei erzeugt werden soll und über den Datenspeicherungstyp, wie die Daten über eine bzw. mehrere Dateien verteilt werden sollen. Wird als Stil „SQL“ festgelegt, werden die Daten natürlich nicht in einer Datenbank gespeichert, sondern die SQL-insert Befehle ausformuliert und in eine einzige Datei geschrieben (Speicherungstyp wird ignoriert). Über die Eigenschaften *fileName* und *pattern* kann ein Dateiname und eine Dateinamensstruktur festgelegt werden. Für letzteres können Platzhalter für Zählungen festgelegt werden, falls der Dateiname bereits existiert. Wird beispielsweise *fileName* auf „datensatz“ und das *pattern* auf %name_%d.dat gesetzt, so wird für jede neue Messung eine Datei nach der Struktur datensatz_0.dat, datensatz_1.dat,... erzeugt.

Ausgabe des SMTP-Writers Die Ausgabe des SMTP-Writers entspricht der des File-Writers, außer dass Messages über das SMTP-Protokoll versendet werden und keine Daten auf die lokale Festplatte geschrieben werden.

Ausgabe des Database-Writers Für den Database-Writer ist es sinnvoll als Ausgabestil SQL anzugeben, wobei dies nicht obligatorisch ist. Die Daten werden so interpretiert wie oben angegeben und es werden Datensätze in die angegeben Datenbank-Tabellen geschrieben. Im Template selbst können sowohl im Body-Attribut *table* wie auch im Attribut *id* einer jeden Tabelle ein Name für die anzulegende SQL-Tabelle angegeben werden. Über das Attribut *ref* kann in jeder Tabelle eine entsprechende Spalte für den Foreign-Key festgelegt werden, wobei ohne diese Angabe einfach der Haupt-Tabellenname mit angefügtem *_id* verwendet wird.

3.3.8. (Analytische) Methoden als zentrales Strukturelement

Bei der Entwicklung analytischer Aufbauten wird das Konzept der analytischen Methoden benötigt wie es in Abschnitt 2.2 eingeführt worden ist. Wie bereits erklärt besteht eine analytische Methode aus einer Analysentechnik und einer Auswertung, die Excalibur-Methode aus Geräten und Abläufen (Analysentechnik), außerdem Skripts (Auswertung). Wichtig hierbei ist, dass alle drei Aspekte voneinander unabhängig sein sollen.

In Methoden können sowohl Geräte als auch Gerätekategorien gekapselt werden (die vollständigen Schema Dateien für Methoden, Geräte, Ein-/Ausgabe-Typen und Programme befinden sich im Anhang). Wie aus der Analyse der Problematik hervorgeht, sollte immer mit Gerätekategorien gearbeitet werden, um Abhängigkeiten der konkreten Geräte von der Methode zu vermeiden (s. unten).
Darüber hinaus werden alle Ablaufsequenzen gekapselt, die Aktionen auf Geräten/Kategorien dieser Methode ausführen. Methoden enthalten demnach alle Informationen, die benötigt werden, um einen Messablauf für eine bestimmte analytische Methode durchzuführen. Die Ein- und Ausgaben werden letztlich über die ebenfalls in den Methoden definierten korrespondierenden Elemente (Writer, Transformer, Serialisierer) spezifiziert (s. unten).

Gerätereferenzen und -namen

Ein Gerät wird innerhalb von Excalibur mit einer Schnittstellendefinition, einer Konfiguration und einem Befehlssatz (Treiber) repräsentiert. Durch diese Unterteilung ist eine hohe Flexibilität und Unabhängigkeit der einzelnen Unteraspekte gegeben. Die Schnittstellendefinition spezifiziert den Kommunikationsweg über die Hardware, die Konfiguration die Geräteeigenschaften (Ein-/Ausgaben, Initialisierung usw.) und letztendlich der Treiber definiert die Kommandos für die Geräteaktionen. Den Zusammenhang zwischen den einzelnen Deskriptoren bilden die *Gerätenamen*. Im Gegensatz zur Gerätereferenz (s. Abschnitt 3.3.4), welche eine im System einzigartige Bezeichnung für ein Gerät im Sinne einer Hardware-Schnittstelle darstellt, sind die Gerätenamen eine Bezeichnung für eine funktionale Einheit, unter welchem eine Methode (oder Methodenteile wie Ablaufsequenzen, Plugins, u.ä.) ein Gerät oder einen Gerätebestandteil adressiert. In anderen Worten kann man Hardware-Schnittstellen mehrere Namen zuweisen, mit welchen auf einen ganz bestimmten Funktionssatz (Aktionen und Eigenschaften) des zugrunde liegenden Gerätes zugegriffen werden kann. Besteht beispielsweise eine Multifunktions-Laboranlage aus sowohl einer Pumpe und einem Ventil, so kann mit Hilfe zweier Namen ebendiese Anlage in zwei Geräte unterteilt werden, welche für eine Ablaufsequenz unabhängig voneinander adressiert werden können. So ist es möglich diese eine Anlage durch zwei neue Geräte zu ersetzen (einer Pumpe und einem Ventil von z.B. zwei verschiedenen Herstellern), ohne dass die Ablaufsequenz geändert werden muss. Dieses Vorgehen ist bereits am Szenario der reflektometrischen Interferenzspektroskopie in Abschnitt 2.3.2 demonstriert worden.

Gerätenamen werden über ein Mapping festgelegt, welches unabhängig von der zugrunde liegenden Methode, den Gerätereferenzen Namen zuordnet. (Mindestens) Ein Mapping existiert für jede Methode und kann über den entsprechenden Mapping-Request (s. Anhang) vom Excalibur Server abgerufen und gesetzt werden. Im Umkehrschluss bedeutet dies, dass Methoden die zugrunde liegenden (Hardware-)Geräte nicht kennen und Geräte ausschließlich über den Namen der jeweiligen funktionalen Einheit referenzieren.

Gerätekategorien

Das Verwenden von funktionalen Einheiten birgt die Gefahr, dass bei Austausch eines Gerätes gegen ein Äquivalentes, die Ablaufsequenzen dennoch unbrauchbar werden, ausgelöst durch Unterschiede in den Treibern. Zwar ist es verständlich, dass beim Austausch

von z.B. einem Spektrometer gegen ein neueres eines anderen Herstellers, das Gerät nach wie vor Spektren aufnimmt. Dennoch ist es möglich, dass der Excalibur-Treiber des Gerätes nicht konform ist und der „Mess-Befehl“ zwar vorhanden, aber unterschiedlich benannt worden ist.
Es gibt zwei Möglichkeiten, dieses Problem zu lösen: Entweder es wird im Treiber ein Alias für den „Mess-Befehl“ erzeugt (s. unten) oder es wird der Identifikationsname des Kommando-Typs geändert. Beide Möglichkeiten sind äquivalent. Um diesen Prozess zu verallgemeinern, gibt es die Gerätekategorien. Eine Gerätekategorie ist ein Deskriptor mit Bezeichnungen für Kommando-Typen und deren Übergabeparameter, um einen wohl definierten Kommandosatz zu gewährleisten. Ein Treiber wird dabei einer/mehreren Kategorie(n) zugeordnet, während der Parser des Treibers überprüft, ob der Kommandosatz mindestens die in allen Kategorien spezifizierten Kommandotypen enthält. Auf diese Weise kann durch die ausschließliche Verwendung von Kategorien innerhalb der Methodendefinition immer gewährleistet werden, dass alle Geräte, welche von den benötigten Kategorien abgeleitet werden, auch für die zugrunde liegende Excalibur-Methode und damit auch analytische Methode verwendet werden können.

Kommandos und Aliase

Wie für jede Geräte-Referenz Aliase in Form von Gerätenamen vergeben werden können, kann auch für jeden Kommando-Typ ein Alias vergeben werden. Wie bereits angedeutet, können dabei einerseits gewisse Eingabeparameter bereits vorbelegt werden, um den Aufruf zu vereinfachen. Andererseits ist es möglich, generische Treiber z.B. aus einer Treiberdatenbank zu verwenden und die einzelnen Kommandotypen konform zu in der Laborumgebung verwendeten Gerätekategorien zu machen. Zuletzt muss eine Möglichkeit existieren, Multifunktions-Laborgeräte den Methoden verfügbar zu machen, welche zu doppelten IDs in den Kommandotypen führen würden. Dies ist der Fall, wenn beispielsweise eine Laboranlage über zwei Einzelpumpen verfügt und diese der Methode über zwei unterschiedliche Namen zur Verfügung gestellt werden. Weist man die beiden Pumpen jeweils einer Kategorie zu, müssten beide Pumpen über denselben Befehlssatz verfügen (beide Pumpem müssen „pumpen“). Dies ist aber nicht möglich, weil das System unterscheiden muss, welche der beiden Einzelpumpen adressiert werden soll. Für diesen Fall können Aliase für Kommandotypen in Kombination mit dem Gerätenamen eingeführt werden, wodurch zwar nicht der Befehlssatz eindeutig wird, aber die Kombination aus Gerätebefehl mit dem Gerätenamen.

Wie auch die Gerätenamen sind die Kommando-Aliase unabhängig von der Methode. Auch sie werden über ein Mapping zur Verfügung gestellt und können über den Mapping-Request (s. Anhang) abgerufen und verarbeitet werden. Für jede Methode können mehrere Mappings angelegt werden.

Ein- und Ausgabeelemente

Ein- und Ausgaben sind ein fundamentaler Aspekt der Datenverarbeitung. Das in Abschnitt 3.3.5 besprochene EVA-Konzept ist die in Excalibur eingesetzte Lösung, um auf externe Ressourcen und Eingaben zu reagieren. Um eine maximale Flexibilität zu gewährleisten, können Ein- und Ausgaben in Form von Writern nicht nur für Geräte, sondern auch für Methoden definiert werden. So können abhängig von der Methode, Daten unterschiedlich interpretiert, evaluiert und auch abgelegt werden.
Darüber hinaus ist die Datenein- und -ausgabe nicht mehr abhängig von den Geräten, Kategorien[22] oden den Systemen, auf denen sie laufen. Die Datenwege werden dabei ausschließlich über Kanäle gesteuert. Somit ist die Datenausgabe unabhängig davon, an welcher Stelle und auf welche Weise die Daten erzeugt worden sind.

Zusammenführung aller Merkmale

Aus diesen Merkmalen kann ein Datenmodell erzeugt werden, das mit Hilfe eines XML Schemas dargestellt werden kann (s. Anhang B.8). Über einen eindeutigen Namen kann die Methode definiert werden. Die besprochenen Merkmale werden nun wie in Listing B.10 definiert, der Reihe nach die Geräte, Kategorien, Ablaufsequenzen, ein Frameset und die Ein- bzw. Ausgabeelemente. Das Frameset ist dabei eine Beschreibungsdatei, welche der Methode Plugins zuweist und diese konfiguriert. Die kompletten Spezifikationen der einzelnen Typen befinden sich im Anhang.

Lösung

Fasst man die Elemente so zusammen, kann eine Methode ohne Änderung von einem System auf ein anderes transferiert werden. Es müssen lediglich die Mappings für Geräte und Aliase auf dem neuen System angelegt werden.

[22] Solange sich die Datentypen nicht ändern.

Persistenz

Unter Persistenz wird das Halten und Abrufen von Information mit möglicher zeitlicher Unterbrechung der Datenverbindung verstanden. Methoden, wie auch die enthaltenen Gerätekonfigurationen und Ablaufsequenzen (Programme) werden vom Excalibur Server interpretiert und verwendet. Die Erstellung und Bearbeitung kann jedoch nicht mit diesem erfolgen, da er über keine grafische Oberflächen für derartige Formulare verfügt. Aus diesem Grund können diese Daten zwar auf dem Excalibur Server hinterlegt werden und mit Hilfe des ConfigurationRequests (s. Anhang) abgerufen werden; die Daten müssen aber von einem externen Programm geladen, verändert und wieder abgelegt werden, welches über die entsprechenden Formulare und eine grafische Oberfläche verfügt.

Speicherung von Gerätekonfigurationen, Ablaufsequenzen und Methoden Unter der Gerätekonfiguration werden Initialisierungsvorschriften, Variablendefinitionen oder auch Verarbeitungsparameter und -Skripte verstanden. Sie müssen beim Verbinden des Gerätes angegeben werden und werden dann einmalig für das zu verbindende Gerät bis zum erneuten Verbinden ausgeführt bzw. gesetzt. Zusätzlich können für das Gerät für spezialisierte Ausgaberouten Elemente wie Writer, Transformer und Serialisierer spezifiziert werden. Sie werden dann im verbundenen Gerät hinterlegt und bilden die oben eingeführte „Geräteroute“ für die Ausgabe.

Konfigurationen

Betrachtet man sich den Verbindungsprozess genauer, müssen beim Verbinden eines Gerätes alle Gerätekonfigurationen angegeben werden. Dies bedeutet im Gegenzug, dass dem verbindenden Client diese Konfigurationen lokal zur Verfügung stehen müssen, entweder auf der lokalen Festplatte oder indem sich der Client die Konfigurationen vom Excalibur Server abholt. Um ein Laden vom Server und erneutes Zurücksenden zu vermeiden, wie es bei diesem Szenario des Verbindens der Fall ist, kann mit Hilfe der Persistenz der Vorgang abgekürzt werden. Da die Gerätekonfigurationsdaten bereits auf dem Server abgelegt sind, kann direkt auf diese Daten verwiesen werden ohne sie erst vom Server laden zu müssen. Sie werden einfach über den korrespondierenden Gerätenamen referenziert und der Excalibur Server übernimmt das Laden der Konfiguration und setzt die Werte während des Verbindungsvorgangs.

Ablaufsequenzen

Ähnliches gilt auch die Ablaufsequenzen, welche in Form von „Programmdateien“ auf dem Server hinterlegt sind. Anstatt das Programm auf den lokalen Client zu laden, werden Ablaufsequenzen direkt über den Programm-Namen aufgerufen und ausgeführt.

Methoden

Da Methoden letztendlich Gerätekonfigurationen und Programme kapseln, ist es nahe-

liegend auch diese auf dem Server zu hinterlegen. So ist es nun möglich über spezielle Methodenbefehle, direkt die Geräte zu konfigurieren, verschiedene (oder alle) Ablaufsequenzen zu administrieren und die Daten geregelt zu verarbeiten und abzulegen, ohne auf dem Client dafür komplexe Parser implementieren zu müssen. Die Entwicklung von Clients wird dadurch sehr vereinfacht, abhängig davon ob Methoden bearbeitet oder ausschließlich aufgerufen werden sollen. Vor allem hinsichtlich mobiler Anwendungen kann der Entwicklungsaufwand deutlich reduziert werden.

3.3.9. Dezentrale Steuerung

Die meisten Steuerungslösungen basieren auf einer lokalen Gerätesteuerung, d.h. Geräte sind an demselben System angeschlossen, an welchem auch die Steuerungssoftware installiert ist. Dieser Ansatz birgt einige Nachteile. Zum einen wird für jede Laboranlage ein eigenes Computersystem mit Ein- und Ausgabegeräten benötigt. Darüber hinaus muss das Messsystem ständig vor Ort überwacht werden, um im Fehlerfall direkt eingreifen zu können. Dies stellt vor allem in kritischen Anwendungen, bei welchem das Analysensystem örtlich getrennt vom Benutzer liegen *muss*, ein Hindernis dar.
Eine dezentrale Steuerung löst die oben genannten Probleme durch eine Trennung der Gerätekommunikation von der Benutzer-Interaktionsebene. Zwar entstehen durch diese Trennung sicherheitstechnische Hürden, welchen aber durch geeignete Authentifizierungs- und Autorisierungsstrategien entgegengewirkt werden kann (s. Abschnitt 3.3.10). Die Interaktion zwischen den beiden Komponenten erfolgt ausschließlich über Anfragen (s. unten). Zuletzt ist es durch diesen Ansatz möglich, den Client für andere Anwendungen wiederzuverwenden bzw. bei Plugin-basierten Clients, mit demselben Client mehrere Anlagen gleichzeitig zu betreiben (s. Abschnitt 3.3.9).

Request-Response Mechanismus

Der Excalibur Server ist ein Dienst, der ohne grafische Oberfläche als Hintergrundprozess permanent läuft. Um mit dem Server kommunizieren zu können, werden Kommunikationsschnittstellen benötigt, um Aktionen auf dem Dienst durchzuführen. Dafür sind die oben erklärten Eingabeschnittstellen in Form von Readern eingeführt worden, welche mit Daten bestückt interne Datenstrukturen befüllen können und damit Prozesse auslösen. Wie diese Daten auszusehen haben, wird durch die Deserialisierer entschieden, z.B. XML oder JSON. Es können verschiedene Reader mit mehreren Deserialisierern

definiert werden, an welche Anfragen (Requests) gestellt werden können. Durch den gewählten Ansatz ist Excalibur per se nur dezentral steuerbar, d.h. über den Reader kann gewählt werden, auf welche Weise die Anfragen an den Dienst gestellt werden sollen: Über das Netzwerk, lokale Dateien, einem Webservice oder theoretisch über andere kabellose Schnittstellen wie Bluetooth oder Mobilfunk.
Die Requests selbst müssen von externen Programmen, den Clients, selbst konstruiert werden, beispielsweise über Factories (s. Abschnitt 3.4), um die für den aufzurufenden Prozess benötigten Informationen zu enthalten. Für diese Prozesse sind Bibliotheken und verschiedene Toolkits entwickelt worden (s. Abschnitt 3.4), um auf einfache Weise Requests im richtigen Format konstruieren zu können. Wird der Webservice-Reader in Kombination mit einem XML-Deserialisierer als Eingabeschnittstelle verwendet, ist die Eingabe bereits durch eine WSDL oder WADL Datei standardisiert und es kann mit geeigneten, frei verfügbaren Clients, daraus vollständige Klassenstrukturen generiert und direkt eingesetzt werden.[23]
Nach Ausführung des angeforderten Prozesses wird in jedem Fall eine Antwort an den Aufrufer zurückgegeben. Dies erfolgt über die entsprechenden Writer inkl. deren konfigurierte Serialisierer, welche die Antworten in der gewünschten Notation zurückliefern. Ist während des Prozesses ein Fehler aufgetreten, wird in der entsprechenden Notation eine Fehlermeldung generiert und dem Aufrufer gemeldet.

Publish-Subscribe Mechanismus

Nicht immer werden Prozesse nur ausgeführt, wenn zuvor eine Anfrage gestellt worden ist. Zum einen können im Rahmen von Backup-Prozessen oder beispielsweise bei Überwachung von Verzeichnissen von einem File-Writer, Mitteilungen an den Benutzer auszuliefern sein. Zum anderen senden Ablaufsequenzen Statusmitteilungen während sie ausgeführt werden.
Die bei selbständig laufenden Prozessen produzierten Mitteilungen, Programm-Statusmitteilungen oder die anfallenden Fehler können nicht an den Benutzer ausgeliefert werden, da keine Anfrage gestellt worden ist. Um diese Meldungen als Benutzer dennoch zu erhalten, gibt es zwei mögliche Lösungsansätze. Einerseits können die Meldungen in einer Queue hinterlegt werden und der Benutzer kann sie in festgelegten Intervallen abrufen; diesen Prozess nennt man *Polling* und er läuft *synchron*. Andererseits kann auch die gegenteilige Variante gewählt werden und der Server liefert die Meldungen an

Synchron

[23] Beispielsweise http://membrane-soa.org/soap-client/

Asynchron

den Client aus. Dieser Prozess wird als *eventbasiert* bezeichnet und er läuft *asynchron*. Letzteres bedingt, dass der Server den Client bzw. seine Adresse kennen muss, damit der Meldungen an diesen ausliefern kann. Dazu muss sich der Client im Vorfeld registrieren (Subscribe). Fällt nun eine interne Meldung an, kann der Excalibur Server sich entscheiden, ob die auszuliefernde Meldung durch eine synchrone Anfrage ausgelöst worden ist oder durch einen bereits laufenden Prozess. In letzterem Fall durchläuft er eine Liste registrierter Clients und sendet die Meldung an deren hinterlegte Adressen (Publish).
Mit dem eventbasierten Mechanismus kann der Server den Client permanent über interne Prozesse auf dem aktuellen Stand halten, ohne dass der Client ständig Status-Anfragen senden muss. Zusätzlich kann der Server ständige Angaben senden, inwieweit laufende Ablaufsequenzen fortgeschritten oder bereits beendet sind. Dadurch weiß der Benutzer bzw. dessen verwendete Client-Software, an welcher Stelle sich die Ablaufsequnez gerade befindet.

Multi-Server/Multi-Client Ansatz

Ein Client-Server Ansatz im eigentlichen Sinn sieht vor, dass zwei getrennte Softwarekomponenten über ein Netzwerk-basierte Schnittstelle mit Hilfe eines gemeinsamen Protokolls kommunizieren. Der Vorteil dieser Vorgehensweise ist die Möglichkeit, den Server und den Client unabhängig voneinander entwickeln zu können. Die Verständigung der beiden Komponenten bleibt dabei erhalten, solange sich das Kommunikationsprotokoll nicht ändert. Meist wird dabei eine Server-Komponente eingesetzt, auf welche mit verschiedenen Clients zugegriffen werden kann. In anderen Worten existiert eine Adresse, auf welcher ein Service angeboten wird (z.B. ein Such-Service wie auf www.google.de), und es sollen mit verschiedensten Komponenten von einem anderen Ort (z.B. ein Browser wie Firefox oder Chrome) Informationen bezogen werden.
Excalibur verhält sich dahingehend anders, als dass auf jeder Laboranlage eine eigene Instanz des Excalibur Server installiert wird, da der zugehörige Dienst direkt mit den Hardware-Schnittstellen und damit auch Geräten kommuniziert. Das bedeutet, es existieren in einem Labor unter Umständen mehrere Server. Die Administration und Steuerung eines Servers hingegen erfolgt hingegen über die Clients, welche sich nicht zwangsläufig auf demselben Computersystem befinden müssen wie die zu konfigurierende Serverkomponente. Weil hingegen die Konfigurationen für die Geräte und Methoden auf dem Server hinterlegt sind, kann über denselben Client-Computer auf alle Server zugegriffen werden. In anderen Worten können alle Clients mit allen im Netzwerk ver-

fügbaren Servern kommunizieren. Die Kommunikation wird über standardisierte Notationen ermöglicht, wodurch auch verschiedenste Clients auf verschiedenen Systemen (Betriebssysteme, mobile Geräte, integrierte Systeme, usw.) eingesetzt werden können. Wird beispielsweise als Client das Web-Schnittstelle eingesetzt (s. Dokumentation), ist es ausreichend die Netwerk-Adresse der Messrechner zu kennen. Wird darüber hinaus die Repository-Komponente eingesetzt (s. Dokumentation), muss sogar ausschließlich die Adresse von diesem bekannt sein, da diese alle weiteren Server im Netzwerk kennt und Anfragen weiterleiten kann.

3.3.10. Sicherheit - Autorisierter Zugriff mit Sessions

Wählt man zur Kommunikation mit dem Excalibur Server das Netzwerk, werden Daten zwischen mindestens zwei verschiedenen Rechnern ausgetauscht. Der Datenaustausch erfolgt wie in den theoretischen Grundlagen erläutert (s. Abschnitt 3.2.4) über die Netzwerkinfrastruktur des Labors, d.h. jede Station (Router, Switch, usw.) zwischen Start- und Zielrechner ist in der Lage Einfluss auf die ausgetauschten Daten zu nehmen. Darüber hinaus kann jeder Client Steuerbefehle an den Server senden, wodurch prinzipiell das gesamte Internet und jedes darin befindliche Computersystem als Client dienen kann. Diese sicherheitstechnischen Fakten dürfen, vor allem für kritische Systeme, nicht vernachlässigt werden. Man kann sie durch eine geeignete Verschlüsselung der Verbindung zwischen den miteinander kommunizierenden Computersystemen und geeignete Identifikationsverfahren (z.B. digitale Signaturen) abdecken.

Sessions

Ein Datenaustausch zwischen dem Excalibur Server und einem Endgerät (Client) über das Netzwerk (oder andere Funkstandards) wird in erster Linie vom Server abgelehnt. Zuvor muss mit dem Server ausgehandelt werden, welches System (oder Person) Daten mit dem Server austauschen möchte und ob es/sie dazu berechtigt ist (Authentifizierung und Autorisierung). Diese erfolgt über einen SecurityRequest (s. Anhang). Bevor jedoch die Authentizität überprüft werden kann, wird anhand eines zertifikatbasierten Verfahrens eine verschlüsselte Verbindung mit dem Endgerät aufgebaut. Dabei wird vom Server bei Anfrage durch den Client ein digitales Zertifikat ausgeliefert, welches über enthaltene Attribute den Server identifizieren kann und zusätzlich einen öffentlichen Schlüssel enthält. Der Client überprüft nun die digitale Unterschrift dieses Zertifikat anhand einer

Zertifizierungsstelle, ob das Zertifikat vertrauenswürdig ist. Diese Zertifizierungsstelle (z.B. Symantec, Comodo, u.ä.) hat im Vorfeld das Server-Zertifikat validiert und mit einer digitalen Unterschrift signiert (s. oben). Gilt das Zertifikat als vertrauenswürdig, wird eine verschlüsselte Verbindung zwischen den beiden Computersystemen aufgebaut (s. SSL in [Res00]). Die Standard-Zertifizierungsstellen, welche unter anderem in jedem Web-Browser standardmäßig als vertrauenswürdig eingestuft sind, werden vom Excalibur Server abgelehnt. Um Kommunikation zu gewährleisten muss entweder die Excalibur-Zertifizierungsstelle in jedem Client als vertrauenswürdig gekennzeichnet werden oder man kann selbst als Zertifizierungsstelle dienen.
Sobald die Authentizität überprüft und das anfragende System autorisiert ist, erstellt der Excalibur Server eine Session, über welche Transaktionen abgewickelt werden können. Dazu sendet der Server über die verschlüsselte Verbindung eine Session-ID, welche das anfragende System bei jeder Anfrage angeben muss und welche zusammen mit der Absenderadresse bei jeder Anfrage vom Server überprüft wird. Diese Session ist solange gültig, bis keine Anfragen über einen gewissen (einstellbaren) Zeitraum eintreffen (Timeout) oder die Session per SecurityRequest (Anhang) wieder geschlossen wird.

Remote User Authentifizierung

Alle oben genannten Vorgängen bedingen eine sehr komplexe sicherheitstechnische Struktur des Excalibur Server. Zur Überprüfung der Authentizität des anfragenden Systems werden Datenbanken mit Benutzerinformationen benötigt. Darüber hinaus werden Datenbanken mit Benutzerrechten, sog. ACLs (Access Control Lists) benötigt, damit anhand dieser bestimmt werden kann, welche Anfragen bzw. Aktionen das anfragende System auf dem Server durchführen darf. Eine derartige Infrastruktur zu implementieren erscheint sehr aufwendig, vor allem aus dem Gesichtspunkt betrachtet, dass sicherheitskritische Module selbst zu implementieren, ein Sicherheitsrisiko in sich darstellt, falls dabei Fehler unterlaufen (s. Heartbleed Bug[24]). Es ist daher sinnvoll auf bereits vorhandene, etablierte Bibliotheken oder Softwarepakete zurückzugreifen, welche diese Infrastruktur implizit unterstützen. In Excalibur wird aus diesem Grund ein Apache Webserver[25] als Proxy eingesetzt, welcher implizit die Verschlüsselungsalgorithmen, Authentifizierungs- und Autorisierungsmodule bereits unterstützt. Zusätzlich wird der Apache Webserver im Rahmen den Python Client Bibliothek (s. Dokumentation) als Darstellungsmaske ver-

[24] http://heartbleed.com/
[25] https://httpd.apache.org/

wendet. Sämtliche oben besprochene Überprüfungen und Abfragen werden somit vom Apache Webserver durchgeführt, ebenso wie die Bereitstellung der Zertifikate und der Aufbau der verschlüsselten Netzwerk-Verbindung. Die Kommunikation mit dem Excalibur Server erfolgt auf diese Weise nur indirekt, indem der Webserver als Proxy fungiert, d.h. dem aufrufenden System vorspielt, es würde direkt mit dem Excalibur Server kommunizieren, aber die Anfragen nur an diesen weiterleitet. Dazu werden in dem gesendeten Datenpaket bestimmte Attribute gesetzt, welche Auskunft geben, wer gerade eine Anfrage gestellt hat und ob er dazu berechtigt ist. Ohne näher darauf einzugehen, wird im Header des korrespondierenden HTTP-Requests das sog. *Remote User Attribut* gesetzt. Nähere Informationen können unter dem Stichwort Single Sign-On gefunden werden.[26]

3.3.11. Anwendung der Ergebnisse auf Problemanalyse

Durch die Trennung der Ablaufsequenzen von den Geräten durch die Einführung von Kategorien, können Geräte und größere Probleme in einem Messsystem durch eine anderes gleichartiges ausgetauscht werden. Dadurch ändern sich die Ablaufsequenzen, mathematische Berechnungen und auch die Ein- und Ausgabe nicht. Innerhalb des Excalibur Servers werden analytische Methoden durch Excalibur-Methoden abgebildet, welche eine Zusammenstellung aus diesen Kategorien, Ablaufsequenzen, Skripten und Ein-/Ausgabeelemente darstellen. Die Verknüpfung dieser Aspekte erfolgt über Namen, Aliase, Zielvariablen und Kanäle. Namen und Alias werden über eine Mapping-Schicht definiert, während Zielvariablen und Kanäle, auf welche an (fast) beliebiger Stelle innerhalb von Excalibur zugegriffen werden kann und Bearbeitungen möglich sind.

Dadurch löst sich das Problem der „Abhängigkeiten der Aspekte einer Sensorumgebung“. ***Lösung***

Durch die Abbildung einer analytischen Methode auf die Excalibur-Methode kann die Steuerung der Laboranlagen im Vorfeld geplant und umgesetzt werden. Die Software-Systemintegration kann dadurch parallel zur Hardware-Systemintegration erfolgen, indem benötigte Gerätekategorien, geeignete Excalibur-Treiber und die Test-Schnittstellendefinitionen verwendet werden und damit Ablaufsequenzen, Skripte, Ein- und Ausgabeelemente definiert und getestet werden. Nach der Hardware-Integration werden nur die Test-Schnittstellen-definition durch die richtige Hardware-Schnittstellendefinition ersetzt, um das System lauffähig zu machen. Für all diese Prozesse muss keine Zeile Quelltext in jedweder Programmiersprache geschrieben werden.[27] Es werden deshalb keine

[26] http://www.opengroup.org/security/sso/sso_intro.htm
[27] Unter der Vorraussetzung, dass mathematische Skripte nicht als Quelltext verstanden wird.

Systemprogrammierer benötigt. Ebenso entfällt das aufwendige Software-Testen; ausschließlich die finalen Integrationstests müssen durchgeführt werden, aber keine White- und Blackbox Tests.

Lösung Dadurch minimiert sich das Problem des „Zeit-/Kosten-Bedarfs durch Komplexität der Steuerung“.

Alle Aspekte können wiederverwendet werden, angefangen bei den Treibern, Kategorien, Methodendefinitionen, bis hin zu den Ein- und Ausgabeelementen zur Formatierung der Datenausgabe. Für ähnliche Methoden können teilweise auch die Ablaufsequenzen wiederverwendet werden.

Lösung Durch diesen Ansatz wird das wiederholte Programmieren von wiederkehrenden Funktionalitäten vermieden.

Die Kommunikation mit dem Excalibur Server erfolgt ausschließlich über Requests über standardisierte Notationen. Auch die zu übertragenen Daten sind standardisiert und in Schemas definiert (z.B. XSD, WSDL). Dadurch können immer dieselben Clients oder ähnliche Endgeräte eingesetzt werden, wodurch sich die Bediener nicht in ständig neue Software-Designs einarbeiten müssen.

Lösung Dadurch löst sich das Problem der „fehlenden Übertragbarkeit“.

Die favorisierte Kommunikationsschnittstelle mit dem Excalibur Server ist Netzwerk-Schnittstelle, insbesondere der Webservice. Die Kommunikation ist dadurch implizit dezentral, wodurch sich die Clients nicht zwangsläufig auf demselben System befinden müssen wie der Server. Die Überwachung des Mess-Prozesses kann daher prinzipiell von überall erfolgen, wodurch ein schnelleres Eingreifen im Fehlerfall ermöglicht wird. Zusätzlich können, ohne zusätzlichen Aufwand kritische Systeme realisiert werden, bei welchem sich das Messsystem zwangsläufig in einer vom Benutzer getrennten örtlichen Umgebung befinden muss.

Lösung Dadurch löst sich das Problem der „Lokalen Steuerung und Datenspeicherung“.

Durch die hier durchgeführte Analyse sind die Spezifikationen für eine Steuerungssoftware entwickelt worden, um alle in der Einleitung dargelegten Probleme zu lösen. In den folgenden Kapiteln wird aus dieser Spezifikation ein Software-Design entwickelt und Strategien für deren Implementierung aufgezeigt.

3.4. Software-Design

Das Softwaredesign bzw. der Softwaredesignprozess ist eine Vorgehensweise, eine Softwarelösung zu planen und zu strukturieren. Dazu werden Anforderungen definiert, welche aus Sicht eines *Auftraggebers*, die die von ihm gewünschten Funktionalitäten beinhalten sollen, und dem *Auftragnehmer* helfen, mögliche Lösungen des Softwareprojekts zu definieren, um die gewünschten Funktionalitäten zu realisieren (*was* wird entwickelt). Diese Anforderungen werden im sog. *Lastenheft* zusammengefasst und sollen den Entwicklern der Softwarelösung unterstützen, die Struktur und damit die Komplexität abzuschätzen und mögliche Fehlentwicklungen zu vermeiden. Auf der Grundlage des Lastenhefts kann der Auftragnehmer anschließend konkrete Lösungsstrategien entwickeln und daraus einen grundsätzlichen Aufbau für die zu entwickelnde Softwarelösung definieren (*wie* wird entwickelt). Diese Informationen wird anschließend im sog. *Pflichtenheft* festgelegt, welches zusammen mit dem Auftraggeber bewertet, ggf. angepasst und abgenommen wird. Beide Dokumente sollen allgemein verständlich die Anforderungen und möglichen Lösungen darstellen.
Die Anforderungen an den Excalibur Server sind im Kapitel 3.3 festgelegt worden, welches somit als Vorlage für ein Lastenheft dienen kann. In diesem Kapitel soll aus den dort festgelegten Anforderungen ein Softwaredesign entwickelt werden, mit dessen Hilfe eine Softwarelösung implementiert werden kann (s. Abschnitt 3.5). Dieses Kapitel kann somit als Vorlage für ein Pflichtenheft dienen.

3.4.1. Anwendungsfälle (Use-Cases)

Zur Festlegung der Anforderungen einer Softwarelösung bieten sich die sog. *Anwendungfälle* an. Sie beschreiben in kurzen Stichworten die zu implementierende Funktionalität und verknüpfen sie mit den auslösenden und den realisierenden Akteuren. Ein Akteur kann hierbei eine Person, das Gesamtsystem oder ein Teilsystem der Softwarelösung darstellen. Zusätzlich kann in den Anwendungsfällen auf benötigte (*includes*) oder optionale (*extend*) Zusatz-Funktionalität verwiesen werden. In anderen Worten beschreiben Anwendungsfälle, was die Umwelt von einem System erwartet.

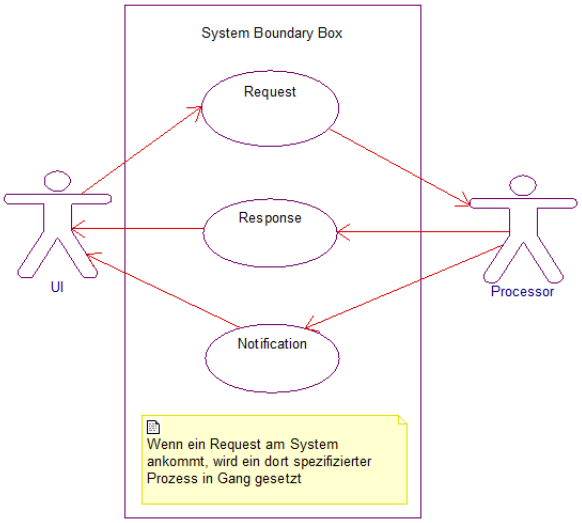


Abbildung (3.9)

Generelle Kommunikationsstruktur für ein UserInterface bzw. einem Benutzer (Akteur „UI“) mit dem Excalibur-Server (Akteur „Processor“). Daten können über Requests, Responses und Notifications ausgetauscht werden.

Übersicht der Requests und ihrer Responses

Ein Request ist eine *Anfrage* an den Excalibur Server, um einen Prozess und damit eine Aktion anzustoßen. Wie in Abschnitt 3.3 besprochen, wird aus einer Anfrage ein Prozess und daraus Kommandos generiert, welche anschließend die gewünschte Aktion durchführen und entsprechend Antworten in Form von *Responses* zurückgeben. Requests (s. Abbildung 3.10) können unterteilt werden in SessionBased, welche nach dem Aufbau einer gültigen Session Aktionen durchführen können, und SessionLess-Requests, welche keine gültige Session benötigen. Hinter ersteren verbergen sich die Haupt-Funktionalitäten des Excalibur Server.[28] Hinter letzteren die Requests, welche für das Session-Management verantwortlich sind und einem TestRequest, der zur Identifikation eines Excalibur Server herangezogen werden kann.

[28] Die dargestellten Requests sollen nur exemplarisch die möglichen Anfragen an den Excalibur Server wiedergeben und nicht die exakte Funktionalität jeder Anfrage beschreiben. Eine genaue Beschreibung jedes Requests mit allen Parametern kann in der Dokumentation nachgelesen werden.

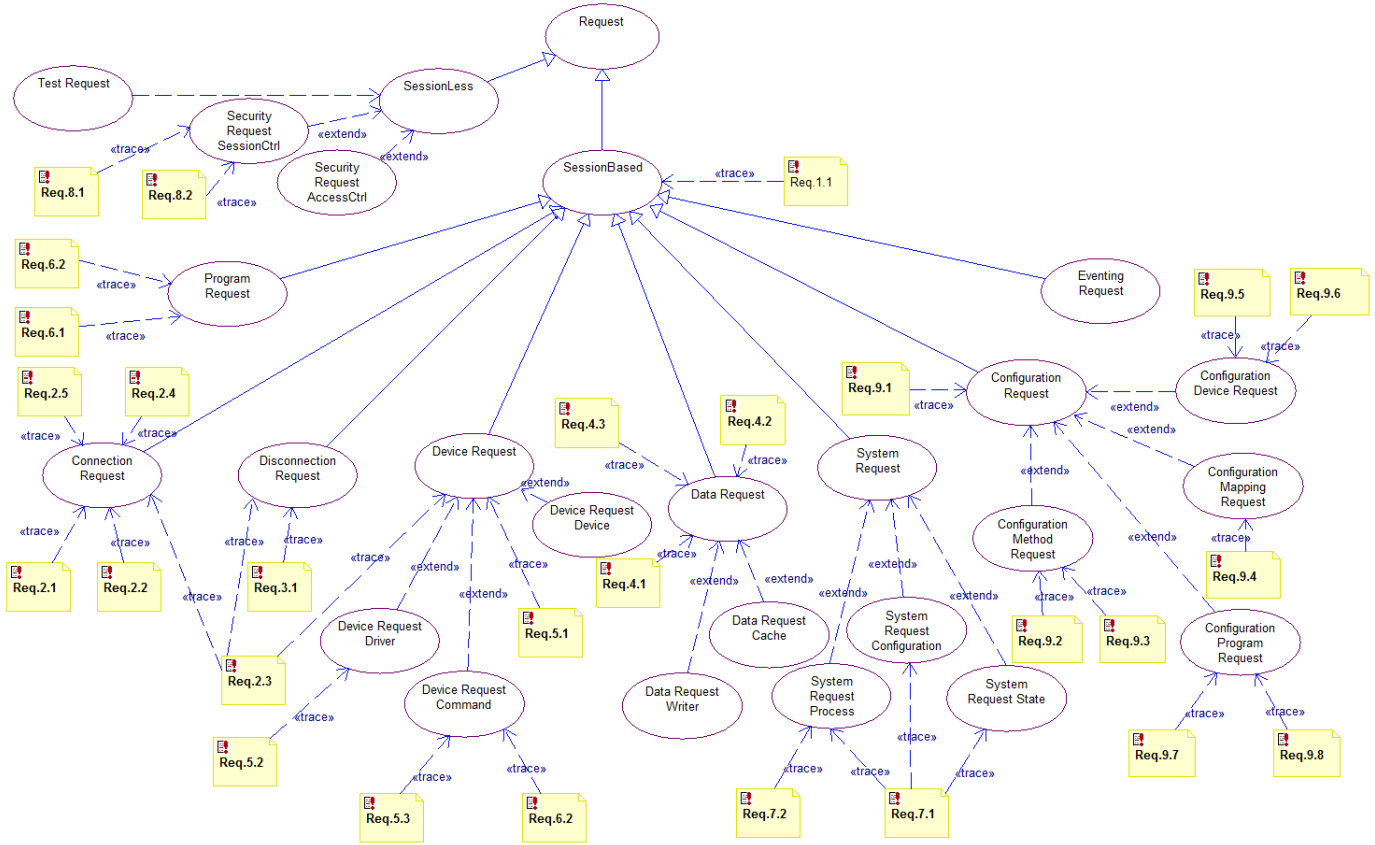


Abbildung (3.10)

Alle verfügbaren Requests des Excalibur Servers, welche entweder die SessionBased-Schnittstelle implementieren und für alle Aktionen des Excalibur Servers verantwortlich sind, oder diese nicht implementieren und für das Sicherheits-Management benötigt werden. Der TestRequest stellt einen Sonderfall dar und gibt ausschließlich die gesendeten Daten wieder formatiert zurück. Auf diese Weise können laufende Instanzen des Excalibur Servers identifiziert werden. Die gelben Boxen definieren die sog. Requirements, d.h. obligatorische Voraussetzungen, die gegeben sein müssen, um den Request auszuführen.

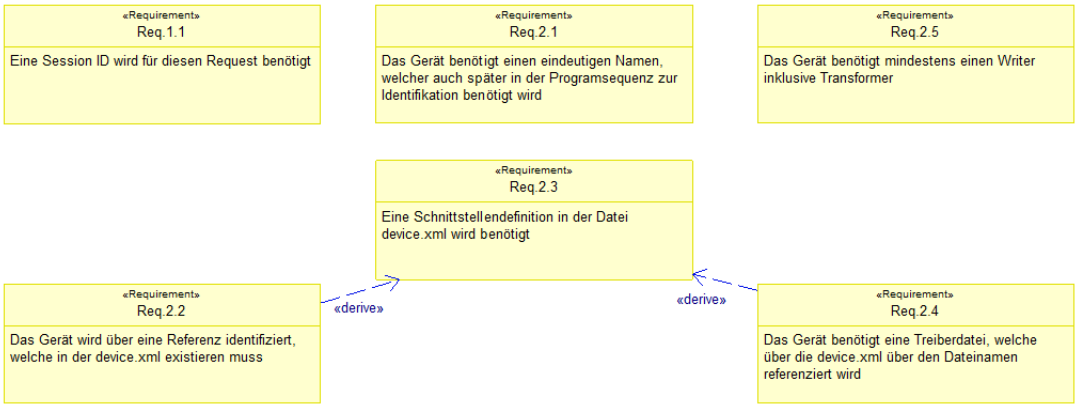


Abbildung (3.11)
Die Requirements des ConnectionRequests.

Für die meisten Requests müssen bestimmte Voraussetzungen erfüllt sein. Dies wird über die sog. *Requirements* (Anforderungen) definiert und sie sind in Abbildung 3.10 mit Hilfe der gelben Boxen dargestellt. Exemplarisch seien in Abbildung 3.11 die Anforderungen des ConnectionRequest aufgeführt. Alle weiteren Anforderungslisten befinden sich in Abschnitt A.1.1 im Anhang.

Auf jeden Request an den Excalibur Server folgt eine Antwort oder eine Fehlermeldung (s. Abbildung 3.12). Sie werden von den einzelnen Kommandos (s. Abschnitt 3.4.3) erzeugt und an den auslösenden Akteur zurückgegeben.[29]

Übersicht der Notifications

Die letzte Möglichkeit der Interaktion zwischen dem auslösenden Akteur und dem Excalibur-System erfolgt über die sog. *Notifications*, d.h. unabhängig von einer Anfrage zurückgegebene Antworten. Notifications werden vom Excalibur Server an die Clients gesendet, was bedingt, dass er diese auch kennen muss. Dies verhält sich dahingehend anders als bei den Requests, bei welchen über die Anfrage des Clients an den Server, letzterer bereits weiß, *wohin* er die Antwort zurücksenden muss. Um diese Informationen auch für die Notifications zu erhalten, muss mit Hilfe eines vorangegangenen Requests, des *EventingRequests*, die Adresse des auslösenden Clients im Server hinterlegt werden. Auf diese Weise können zu jeder Zeit Antworten in Form von Statusmitteilungen und

[29] Auch die möglichen Responses sollen mit Abbildung 3.12 nur exemplarisch wiedergegeben werden und können in der Dokumentation detailliert nachgelesen werden.

Tabelle (3.6)
Übersicht der Requests mit kurzer Erklärung

Request	Erklärung
ConfigurationRequest	Administration der Konfigurationen von Methoden, Gerätekonfigurationen und Programmen. Über diesen Request können deren Konfigurationsobjekte auf dem Server persistent gehalten werden.
ConnectionRequest	Mit diesem Request werden Geräte mit dem Server verbunden, d.h. es wird ein Treiber geladen, die Gerätekonfiguration abgefragt, die Schnittstelle verknüpft und dem Betriebssystem mitgeteilt, dass ein Gerät über diese Schnittstelle kommunizieren möchte.
DataRequest	Abfrage von Daten aus Puffern, welche über gewisse Datenausgabemechanismen (Array-Writer) hinterlegt worden sind. Außerdem können auf diese Weise Caches ausgelesen werden, die z.B. über Skripte oder Zielvariablen geschrieben worden sind.
DisconnectionRequest	Gegenteilige Aktion vom ConnectionRequest, d.h. Trennen des Geräts vom Server.
DeviceRequest	Ausführen von Aktionen auf einem bestimmten Gerät, z.B. Abfrage von Treibern, Schnittstellendefinitionen, aber auch die Ausführung von einzelnen Gerätekommandos.
EventingRequest	Registrierung/Abmelden und eines Clients am/vom Server, damit dieser per Notifications Daten übermitteln kann.
ProgramRequest	Ausführung von Ablaufsequenzen.
SecurityRequest	Authentifizierung und Autorisierung am Excalibur Server.
SystemRequest	Administration des Excalibur Servers, z.B. Setzen/Auslesen von Konfigurationsparametern, Prozess-Steuerung, Zeitsynchronisation, usw.

Fehlermeldungen empfangen werden, um geeignete Maßnahmen treffen zu können, ohne den Server in regelmäßigen Intervallen abfragen zu müssen (s. Abbildung 3.13).

3.4.2. Architektur

Nach [Bal11] beschreibt eine Softwarearchitektur „eine strukturierte oder hierarchische Anordnung der Systemkomponenten sowie Beschreibung ihrer Beziehungen“. Unter der Softwarearchitektur wird also das Funktionsprinzip von Computerprogrammen verstan-

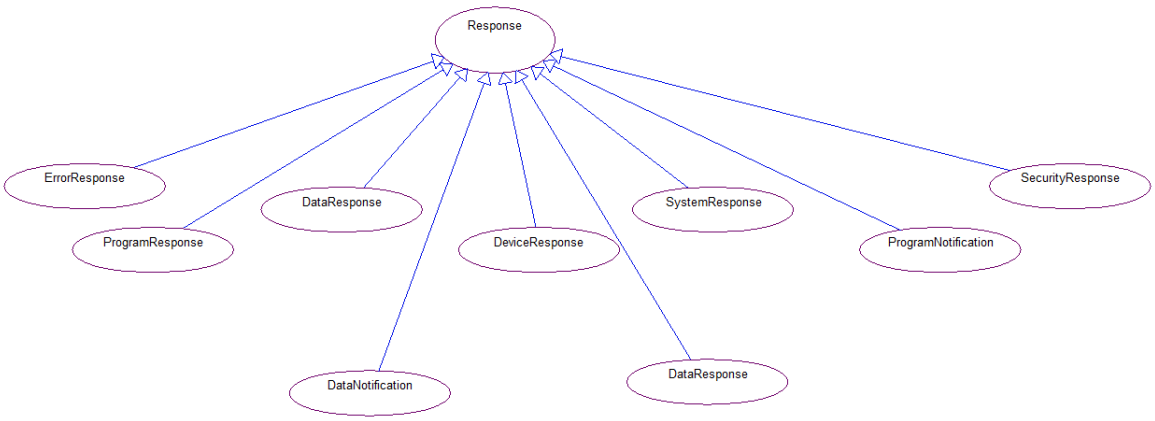


Abbildung (3.12)
Alle verfügbaren Responses des Excalibur Servers, welche auf eine Anfrage zurückgegeben werden können.

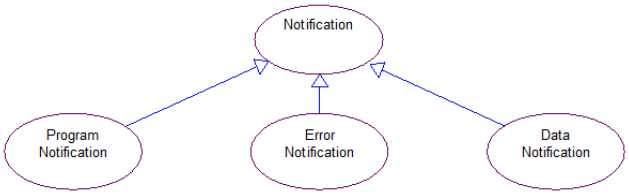


Abbildung (3.13)
Alle verfügbaren Notifications des Excalibur Servers, welche unabhängig von einer Anfrage zurückgegeben werden können. Dazu muss sich der anfragende Akteur beim Excalibur Server registrieren.

den, dazu die Einzelkomponenten, aus denen sie bestehen, und das Zusammenspiel der Einzelkomponenten untereinander.
Zur Verwendung des Excalibur Servers (im Folgenden als *Messserver* bezeichnet) werden Komponenten benötigt, um ihn in einen geeigneten, gebrauchsfähigen Kontext zu setzen. Dazu gehören beispielsweise grafische Editoren zur Strukturierung von Ablaufsequenzen, Komponenten zur Lokalisation und Identifikation von Messservern in einem Netzwerk oder Framework zur einfachen Konstruktion/Interpretation von Anfragen bzw. Antworten, usw. Auf die Architektur dieses Kontextes wird zunächst in Abschnitt 3.4.2 eingegangen. Davon stellt die Einzelkomponente „*ExcaliburServer*“ letztendlich den tatsächlichen Messserver mit dem Namen *Excalibur Server* dar.

Der Messserver selbst besteht aus Einzelkomponenten, auf welche detailliert in Abschnitt 3.4.3 und 3.4.4 eingegangen wird und welcher für die eigentliche Gerätekommunikation, -administration, usw. zuständig ist.

Komponenten und Teilsysteme

Zunächst soll der gesamte Excalibur-Kontext gezeigt werden, innerhalb von welchem eine benutzerfreundliche Verwendbarkeit von einem/mehreren Messserver(n) ermöglicht wird (s. Abbildung 3.14). Ohne diese Komponenten können die Messserver zwar ohne Einschränkungen betrieben werden. Die Kommunikation, Lokalisation und Konfiguration muss allerdings manuell erfolgen; auch grafische Oberflächen zur Gerätesteuerung müssten vollständig neu implementiert werden.

Der „*ExcaliburServer*“ ist letztendlich für die Gerätekommunikation verantwortlich. Dennoch müssen die Anfragen, welche an den Server gesendet werden, in einem gewissen Format vorliegen und, abhängig von der anzustoßenden Aktion, eine wohl definierte Informationsmenge enthalten. Zwar können verschiedene Eingabeformate konfiguriert werden (Deserialisierer), dennoch muss die Informationsmenge geeignet aufbereitet werden (z.B. XML oder JSON). Dafür existieren verschiedene Toolkits, welche zur Verwendung für unterschiedliche Programmiersprachen entwickelt worden sind (*LabVIEWToolkit*, *PythonToolkit* und *JavaToolkit*). Von diesen abgeleitet, existieren mehrere Module zur komfortablen Administration gewisser Teilaspekte oder Deskriptoren.
Der *DriverBuilder* erlaubt eine visuelle Erstellung und Bearbeitung von Excalibur-Treibern und basiert auf dem Java-Toolkit. Damit lassen sich alle Eigenschaften und Routinen der Treiber komfortabel über eine lokale Applikation vom Messserver laden, bearbeiten und wieder speichern.
Der *SequenceAdministrator* erlaubt eine einfache Erstellung von Ablaufsequenzen, deren Ablaufkontrolle und Visualisierung. Der *InterfaceConfigurator* erlaubt eine komfortable Konfiguration der Geräteschnittstellen. Letztere beiden Toolkits basieren auf Python und können als Modul in ein auf „*Django*“ basierendes Framework geladen werden.[30]
Das *DataManagement* ist eine Webapplikation, um Messdaten zu administrieren und dem Benutzer zentral zur Verfügung zu stellen. Dadurch haben Benutzer unter einer einzigen Adresse Zugriff auf alle erzeugten Daten aller Methoden.
Das *Repository* ist eine Webapplikation basierend auf Django und stellt eine Liste aller

[30] Django ist ein auf Python basierendes Web-Framework, welches geeignete Hilfsmittel zur einfachen und schnellen Implementierung von Webapplikationen bereitstellt.

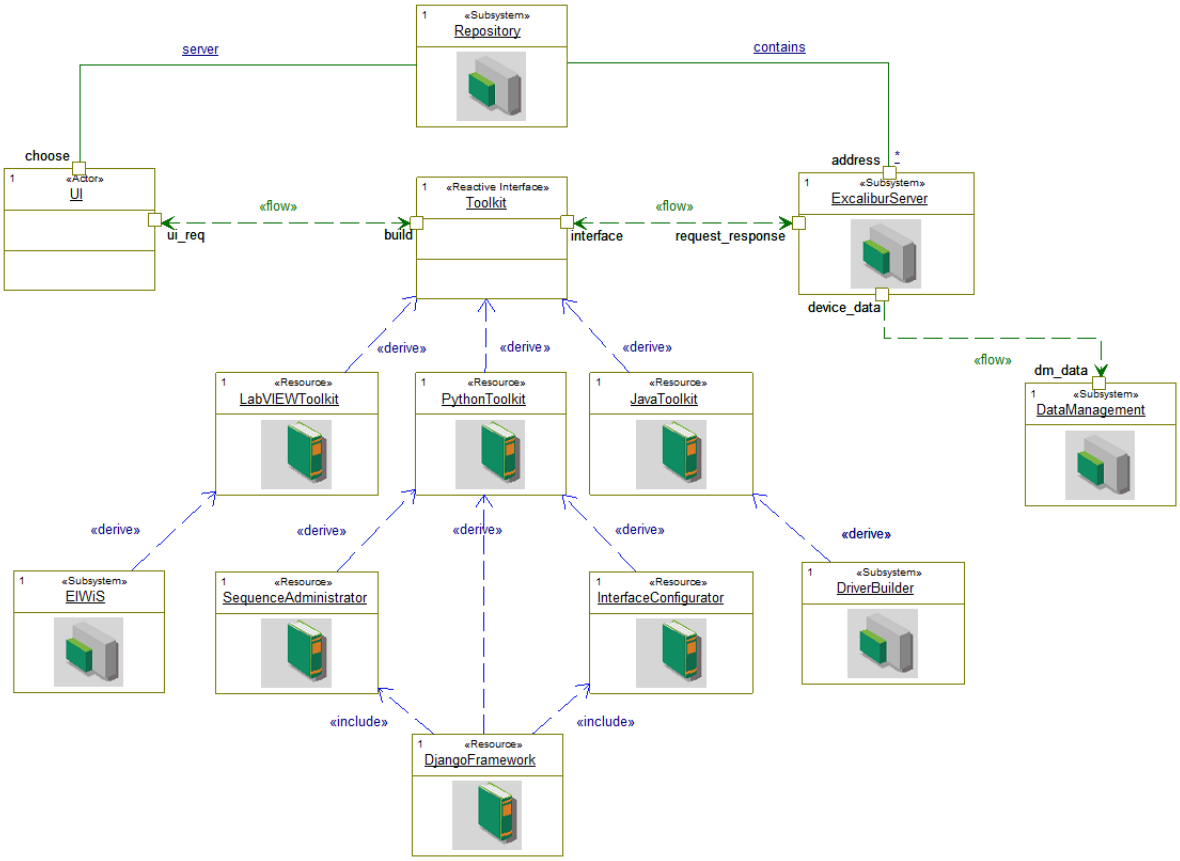


Abbildung (3.14)
Der gesamte Excalibur-Kontext mit allen Einzelkomponenten. Weitere Informationen: siehe Text.

im lokalen Netzwerk verfügbaren Messserver zur Verfügung, inklusive deren Standort und Erreichbarkeit. Wiederum muss der Benutzer nur eine Adresse wissen, um Zugriff auf alle im Netzwerk verfügbaren Geräte zu erhalten.
ElWiS letztendlich ist eine vollständige Client-Implementierung, um auf einfache Weise analytische Methoden zu entwerfen, administrieren und Messungen auf allen im lokalen Netzwerk verfügbaren Servern ausführen zu können. Es handelt sich hierbei um einen Plugin-Container, welcher um benötigte Funktionalitäten wie Ablaufsequenz-Administration, Daten-Visualisierung oder spezialisierte grafische Oberflächen erweitert werden kann. Darüber hinaus stellt er für alle Funktionalitäten des *LabVIEWToolkits* vorgefertigte grafische Oberflächen zur Verfügung.
Auf die Module und alle Komponenten außerhalb des „ExcaliburServer“ (s. Abbildung

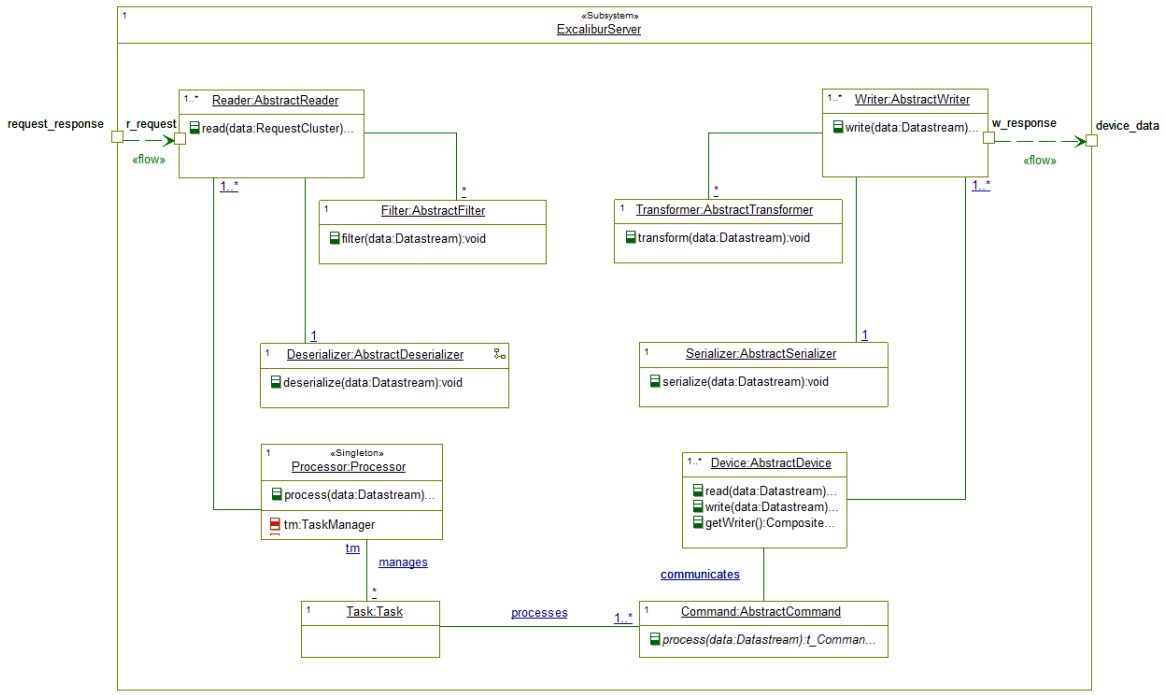


Abbildung (3.15)

Der Excalibur Server mit seinen Ein- und Ausgabekomponenten: Reader, Filter, Deserialisierer, Writer, Transformer, Serialisierer.

3.14) wird hier nicht näher eingegangen. Die Architektur des Servers selbst ist in Abbildung 3.15 dargestellt.
Sobald ein Request am Eingang eintrifft, wird die Kaskade angefangen beim Reader (über Deserialisierer und Filter) angestoßen, anschließend im Prozessor ein Task und Kommandos generiert, über die Geräte[31] Daten produziert und diese letztendlich über die Writer (über Transformer und Serialisierer) wieder ausgegeben.

3.4.3. Statische Struktur des Excalibur Servers

Ausgehend von der Architektur (s. Abbildung 3.15) des Messservers soll im Folgenden auf die statische Struktur der einzelnen Komponenten eingegangen werden, angefangen bei der Ein- bzw. Ausgabe. Anschließend wird der Kernprozess mit der Task- und Kommandostruktur näher betrachtet und letztendlich auf die Struktur einzelner durch die

[31] Der Excalibur Server selbst ist auch ein Gerät und kann entsprechend Daten (als Response) ausgeben.

Kommandos ausgelöster Aktions-Kaskaden. In der statischen Struktur werden viele Entwurfsmuster eingesetzt und können direkt übernommen oder leicht verändert in den jetzt folgenden Klassendiagrammen wiedergefunden werden. Unter einem Entwurfsmuster versteht man Lösungen für wiederkehrende programmatische Problemstellungen, welche als Vorlage für den vorliegenden Fall wiederverwendet werden können. In diesem Kapitel werden ausschließlich Erzeugungs- und Strukturmuster verwendet[GHJV98, Gra98].

Ein- und Ausgabe

Die Eingabe erfolgt über Reader, welcher eine Repräsentation eines Transportmechanismus darstellt (s. Abbildung 3.16). Anfragen können über jeden konfigurierten Transportmechanismus gestellt werden. Die gesendete Anfrage wird im Anschluss durch den konfigurierten Deserialisierer interpretiert und über die verschiedenen Filter modifiziert. Deserialisierer interpretieren die am Reader erreichten Daten und erzeugen daraus ein Datenmodell, das der Messserver verwerten kann (s. Abbildung 3.17).
Reader werden mit Hilfe der IOFactory generiert und anschließend in den Hauptprozess des Messservers eingehängt. Darüber hinaus muss für jeden Reader ein Deserialisierer erstellt werden und es können beliebig viele Filter eingebunden werden. Der Hauptprozess ruft nun in regelmäßigen (und durch das „interval-Attribut“ konfigurierbare) Abständen die *read*-Funktion auf, welche auf dem zugrunde liegenden Transportmechanismus Daten empfängt und sie mit Hilfe des Deserialisierers in ein Excalibur-eigenes Datenmodell übersetzt. Jeder Reader-Typ muss auf eine bestimmte Art und Weise konfiguriert werden: Der FileReader benötigt mindestens ein Verzeichnis auf einem lokalen Datenträger, auf welchem er lauschen soll; der WebserviceReader benötigt eine WSDL Spezifikation, der VIReader den Pfad zu einem kompatiblen LabVIEW®-VI zum Auslesen der Daten und letztendlich der NetworkReader eine URL, über die Daten in einem bestimmten Netzwerk-Protokoll abgerufen werden.
Die über die Reader empfangenen Daten liegen in einem bestimmten Format vor, welches von den Deserialisierern interpretiert werden kann. Daten können dabei entweder in XML oder in JSON vorliegen. Filter sollen letztendlich die Daten modifizieren oder gar aussortieren, sind aber im Moment nur für zukünftige Anwendungen reserviert und es existiert keine konkrete Klasse.

Für die Writer gilt entsprechendes, d.h. sie stellen einen Transportmechanismus dar, welcher Daten aus dem Excalibur Server an die Umgebung weitergeben soll (s. Abbildung 3.18). Dazu gehört einerseits der Benutzer, aber auch Datensenken wie Datenbanken,

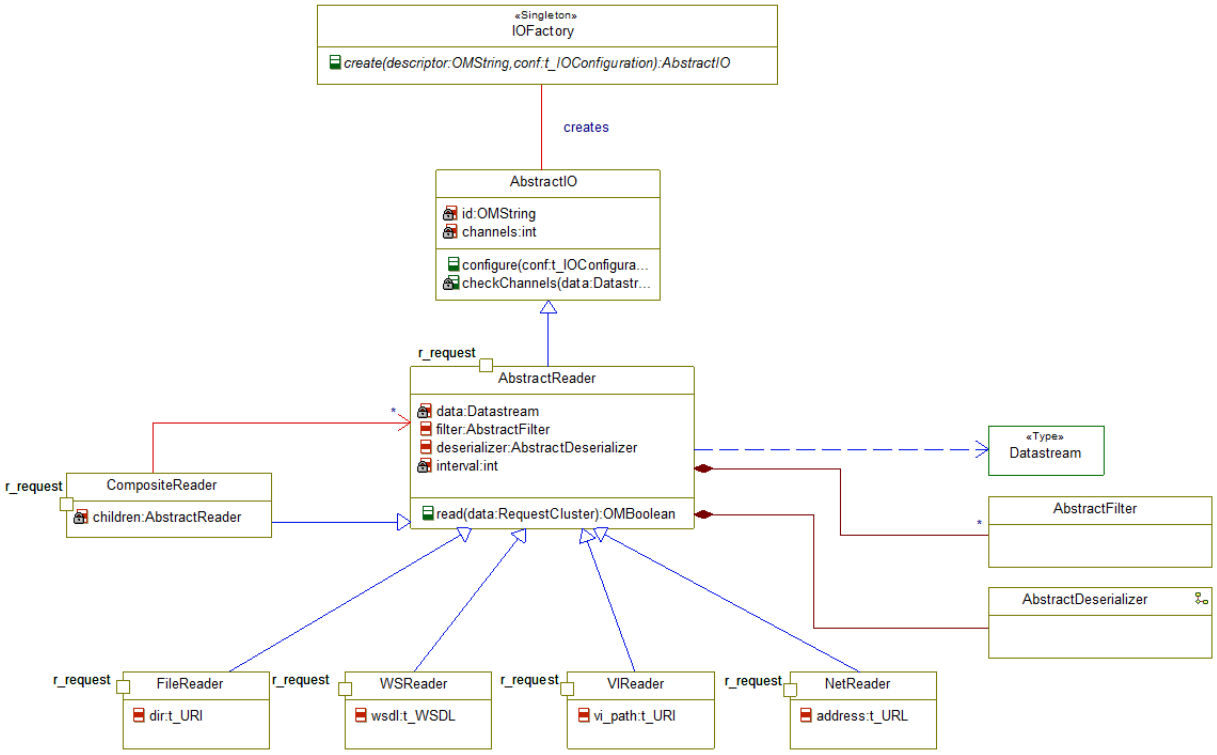


Abbildung (3.16)

Klassendiagramm aller verfügbaren Reader und deren Komposition. Nähere Informationen: siehe Text.

Dateimanagement-Systeme oder spezialisierte Serversysteme (FTP, SMTP, Webserver, usw.). Um die Daten in das richtige Ausgabeformat zu übersetzen, werden Deserialisierer konfiguriert. Zusätzlich können Daten über Transformer mit Hilfe von Skripten oder Prozessanweisungen vorbehandelt werden (s. Abbildung 3.18).
Writer können mit Hilfe der IOFactory generiert und anschließend in den Hauptprozess des Messservers eingehängt werden. Dort werden dann permanent über den Hauptprozess eintreffende Daten mit der *write*-Funktion der einzelnen Writer geschrieben. Für jeden Writer muss jeweils ein Serialisierer konfiguriert und es können zusätzlich beliebig viele Transformer eingebunden werden. Werden Daten von bestimmten Kommandos in die Ausgabe-Queue des Hauptprozesses gelegt, können diese über die konfigurierte Kaskade abgearbeitet werden. Anhand der gesetzten Kanäle im Datenstrom wissen die einzelnen Writer, die alle in Kompositionen (einer baumartigen Struktur) vorliegen, ob sie für die zu schreibenden Daten verantwortlich sind. Der Datenstrom selbst ist eine Struktur, welche die konkreten Datenmodelle aus den einzelnen zu schreibenden Responses

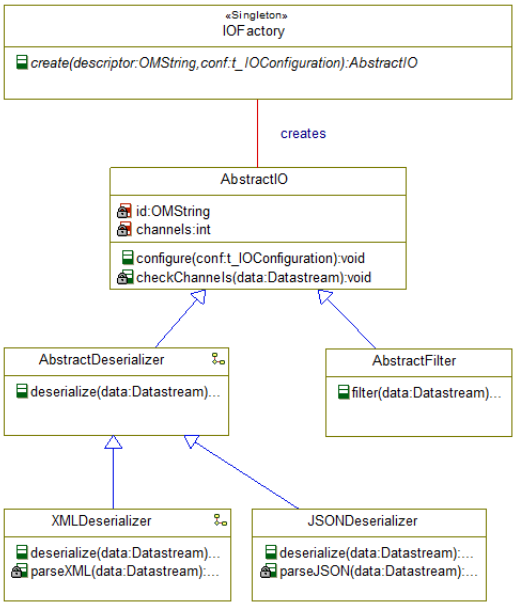


Abbildung (3.17)
Klassendiagramm aller verfügbaren Deserialisierer und Filter. Weitere Informationen: siehe Text.

und außerdem benötigte Prozessparameter, welche bereits während des Leseprozesses gesetzt worden sind, transportieren (z.B. Prozessname, HTTP-Header, TCP-Handles, uvm.). Mit Hilfe des Serialisierers werden die Daten nun in das erwartete Datenformat konvertiert und mit den Transformern nach den gewünschten Vorgaben transformiert. Diese Vorgaben können sehr flexibel gestaltet werden, wenn der zugrunde liegende Writer die *TemplateBased*-Schnittstelle realisiert. In diesem Fall können enthaltene Daten nach einer in Templates (s. Abschnitt 3.4.3) vorgegebenen Strukturanweisung formatiert ausgegeben werden.
Jeder Writer-Typ muss auf eine bestimmte Art und Weise konfiguriert werden: Der FileWriter benötigt mindestens ein Verzeichnis auf einem lokalen Datenträger, in welches er Daten in Form von Dateien schreiben soll; der WebserviceWriter benötigt eine WSDL Spezifikation, der VIWriter den Pfad zu einem kompatiblen LabVIEW®-VI zum Ablegen der Daten, der BufferWriter einen Namen unter welchem er Daten im Arbeitsspeicher puffert und sie zum späteren Abruf zur Verfügung stellt (ggf. werden die Daten auf der Festplatte zwischengespeichert). Der NetworkWriter benötigt eine URL, über die Daten in einem bestimmten Netzwerk-Protokoll gesendet werden, die SMTP- und Datenbank-Writer jeweils Spezifikationen, an welches System die Daten übermittelt werden sollen

(SMTP-Server oder Datenbank-Management-System (DBMS)), und letztendlich dient der NullWriter dazu, Daten explizit zu verwerfen.
Die Serialisierer konvertieren die Daten in das gewünschte Ausgabeformat. Es existieren wie auch bei der Eingabe, eine formatierte Ausgabe als XML und JSON. Zusätzlich können Daten mit Hilfe des TemplateSerialisierers anhand einer vorgegebenen Templatesyntax sehr dynamisch für gewisse Transportmechanismen formatiert werden. Beispielsweise können Daten mit Hilfe eine Templates in Datenbank-Tabellen eingetragen werden inklusive Verlinkung mittels Primary- und Foreign-Keys (s. Abbildung 3.4.3).
Mit Hilfe der Transformer können Daten anhand mathematischer Routinen (Skripte) konvertiert und vorbehandelt werden (DataTransformer) oder gängige Bildbearbeitungsfunktionen angewendet werden (ImageTransformer).

Kernprozess

Im Kernprozess läuft ein separater Thread[32] kontinuierlich ab und liest in regelmäßigen Intervallen alle Reader aus. Sie liefern eine Datenstruktur zurück, welche in einer Queue im Prozessor hinterlegt werden. Sequenziell wird aus jeder Request-Datenstruktur (RequestCluster) ein Task erzeugt und einem *Task-Manager* übergeben. Dieser ist aus mehreren separaten Threads aufgebaut, welche über Queues miteinander Daten austauschen. Abhängig von der Art und seinem momentanen Status durchläuft ein Task mehrere dieser Threads, bis er abgearbeitet ist. Die statische Struktur jener Prozesse ist in Abbildung 3.20 gezeigt.
Abhängig von den enthaltenen Daten erzeugt ein Task verschiedene Kommandos, welche anschließend auf dem System die angeforderten Aktionen ausführen. Alle möglichen Kommandos sind in Abbildung 3.21 gezeigt. Im Normalfall werden die Kommandos eines Tasks sequenziell abgearbeitet. Unter bestimmten Voraussetzungen müssen bestimmte Kommandos aber auch parallel abgearbeitet werden können. Dazu können die Kommandos in ein Composite-Kommando eingefügt und in diesem das *parallel*-Flag gesetzt werden. Dadurch werden alle enthaltenen Kommandos in separate Threads aufgeteilt und, abhängig von der Prozessor-Architektur, (möglichst) gleichzeitig abgearbeitet.
Jedes Kommando kann prinzipiell eine Antwort liefert. Auf einen am Reader eingetroffene Request kann aber im Normalfall nur eine Antwort erfolgen, da der Client nur auf eine einzige Antwort wartet. Manche Transportmechanismen (z.B. HTTP) erlauben auch

[32] Für Grundbegriffe modernen Betriebssysteme wie Thread, Prozess, Task, Queues, usw. soll auf einschlägige Fachliteratur der Informatik verwiesen werden.[Tan09]

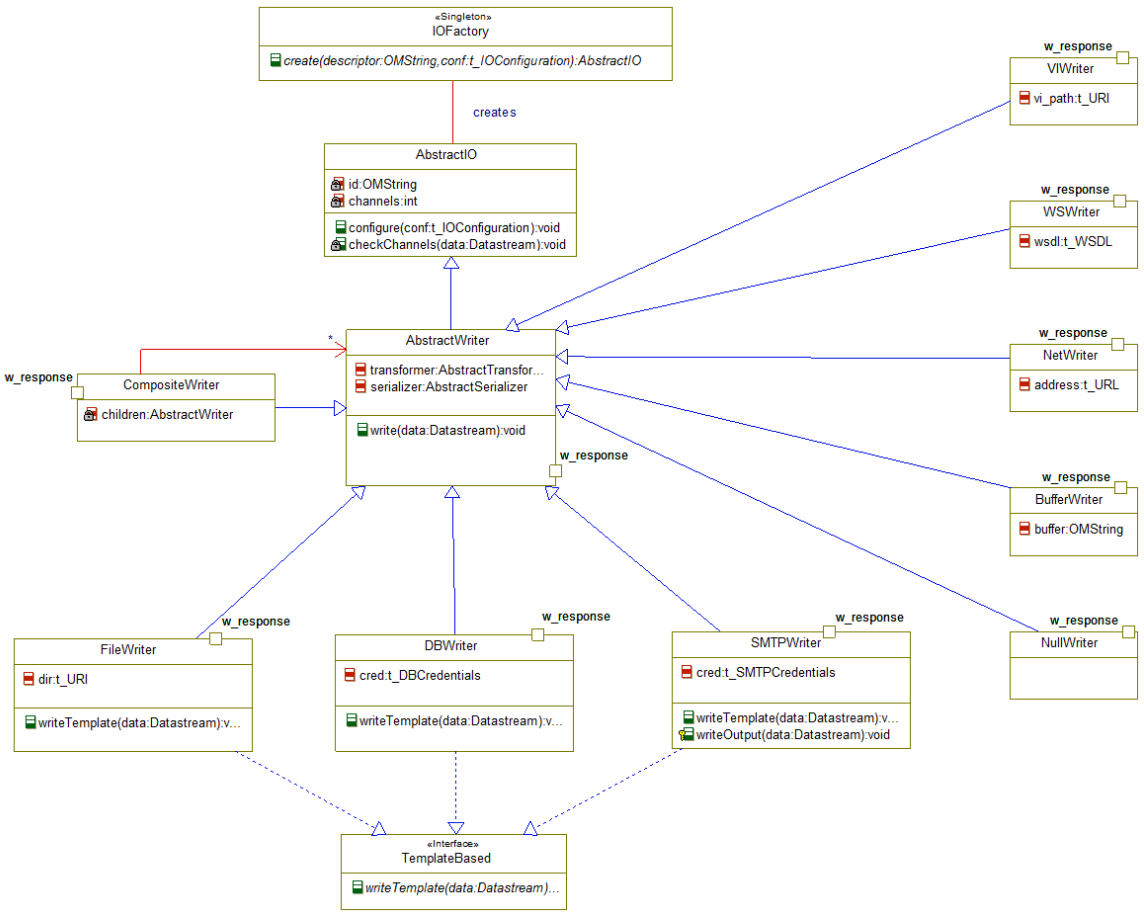


Abbildung (3.18)

Klassendiagramm aller verfügbaren Writer und deren Komposition. Nähere Informationen: siehe Text.

nur den Empfang einer Antwort (HTTP-Response oder HTTP-Error). Der Messserver hat nun die Möglichkeiten entweder die Antworten aller Kommandos zu bündeln oder nur eine ausgewählte Kommandoantwort zurückzugeben. Bedingt durch die enthaltene Informationsmenge der Kommandos, ist letztere Möglichkeit gewählt worden. Welche Kommandoantwort weitergeleitet wird und darüber hinaus, über welchen Weg dies erfolgt (Ausgabe über System- oder Geräte-Writer, s. Abschnitt 3.3.5), ist die Aufgabe des Kommando-Builders (s. Abbildung 3.22).

Der Kommando-Builder ist eine abstrakte Klasse, welche über eine Prozess-Anweisung einen konkreten Builder wählt (DeviceCommandBuilder, NetworkCommandBuilder, usw.). Jener konstruiert, abhängig vom Prozess, die benötigte Kommando-Kaskade und kon-

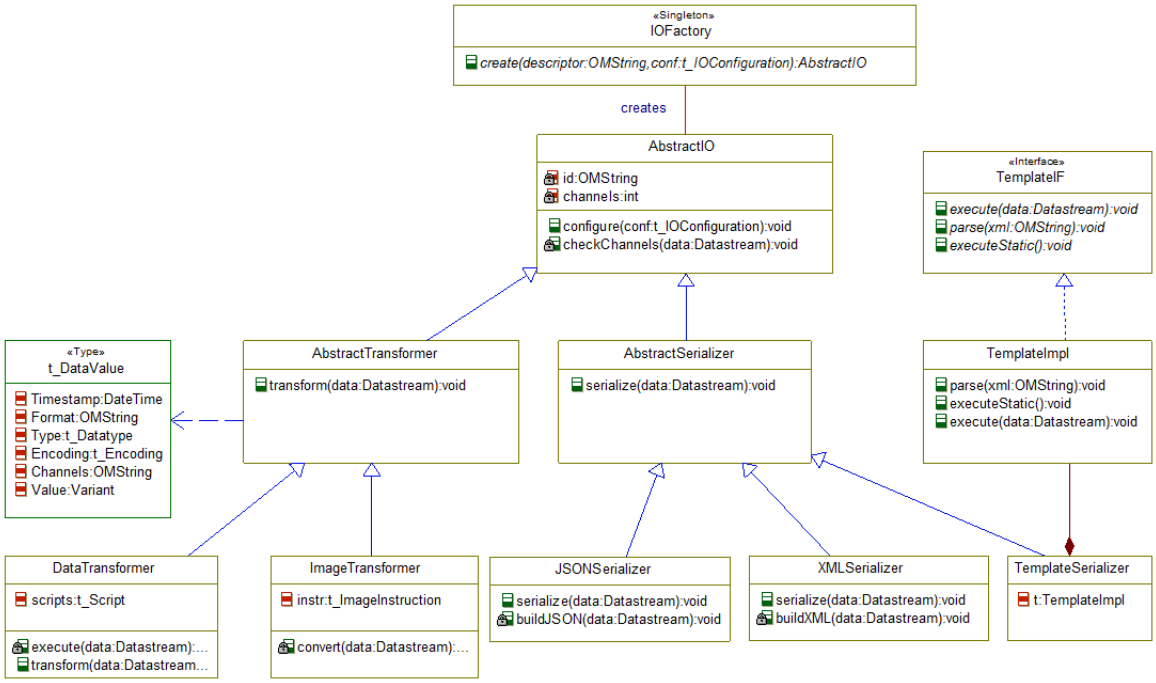


Abbildung (3.19)

Klassendiagramm aller verfügbaren Transformer und Serialisierer. Nähere Informationen: siehe Text.

figuriert die Kommandos anhand der im übergebenen Datenstrom transportierten Modelle. Zusätzlich legt der Builder die Ausgaberoute (System, Gerät) fest und bestimmt das Kommando, wessen Antwort weitergeleitet wird.

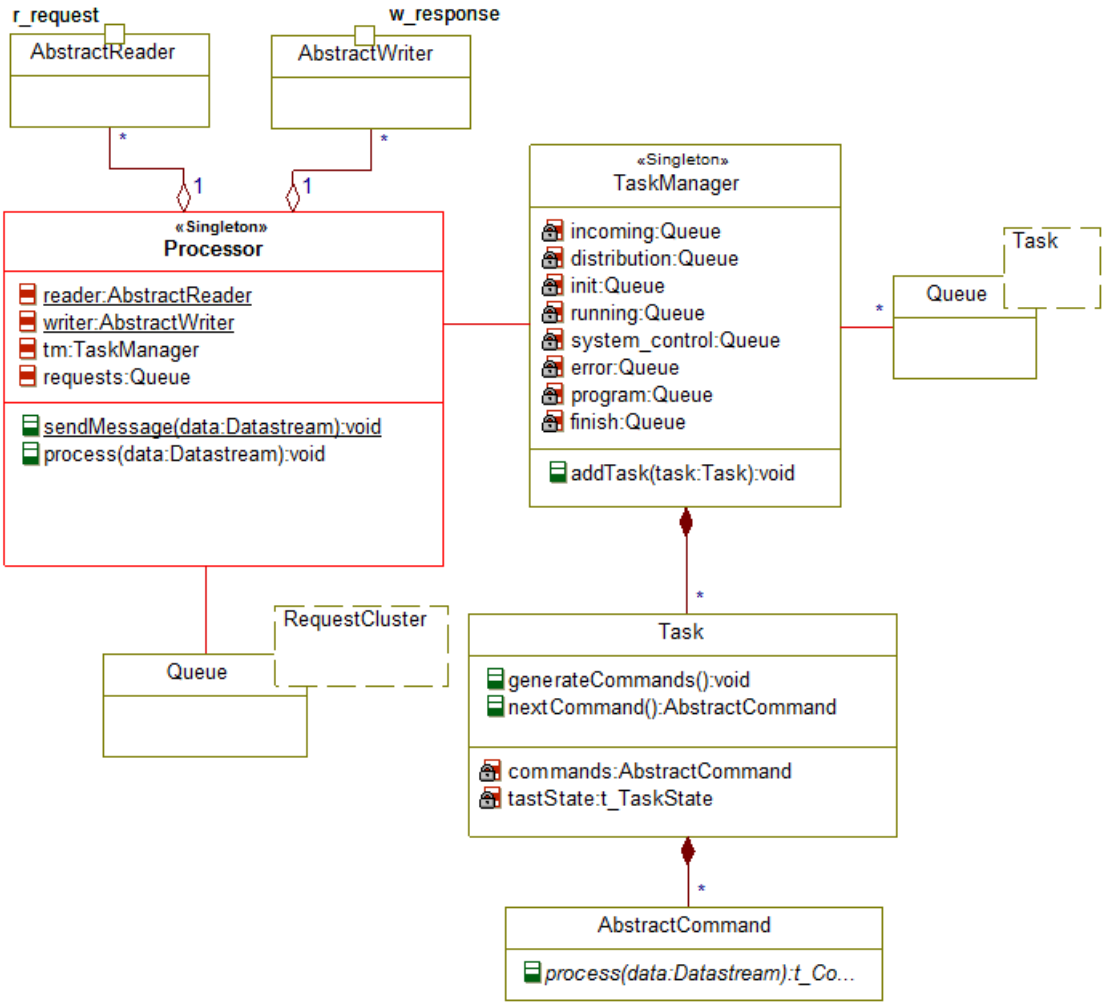


Abbildung (3.20)

Klassendiagramm der Komponenten des Kernprozesses. Der Prozessor kapselt alle Writer und Reader, ebenso hält er eine Instanz des TaskManagers, welcher sequenziell alle ankommenden Requests abarbeitet, indem er Kommandos generiert und diese ebenso sequenziell ausführt.

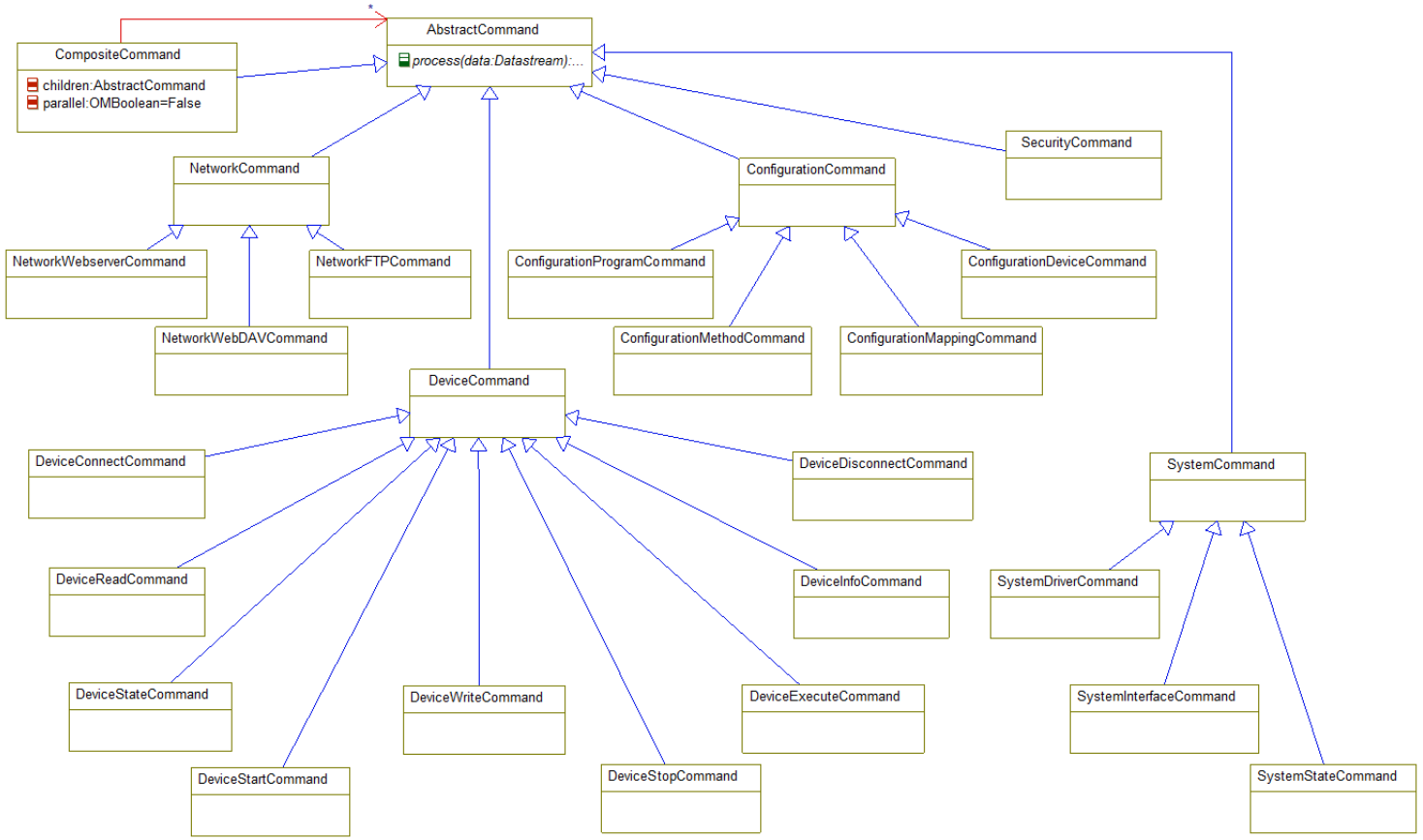


Abbildung (3.21)

Alle verfügbaren Kommandos, die aus einer Anfrage an den Excalibur Server generiert werden können. Jedes Kommando ist exakt für eine Server-Aktion verantwortlich und wird entsprechend ausgeführt (sequenziell). Sollen Kommandos eines Tasks parallel ausgeführt werden, können sie in einem CompositeCommand gekapselt und das parallel*-Flag gesetzt werden.*

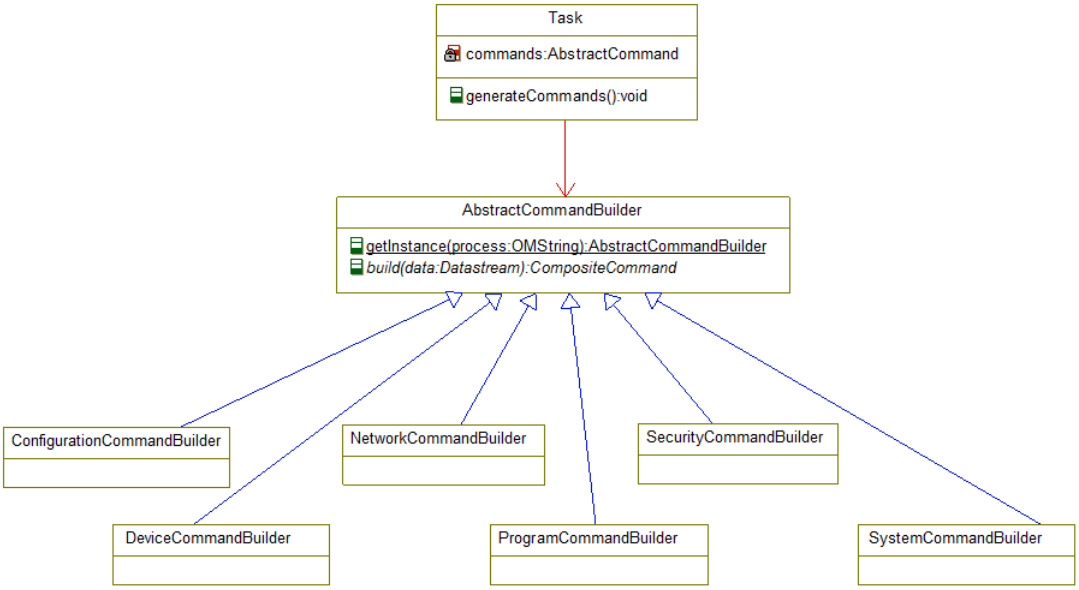


Abbildung (3.22)
Klassendiagramm der Builder zur Konstruktion von Kommandokaskaden.

Schnittstellendefinition

Gerätekommunikation erfolgt über Computerschnittstellen, an denen die Geräte angeschlossen sind. Die Konfiguration der Schnittstellen wird über eine XML-Schnittstellen-Definition realisiert, welche für jeden Schnittstellentyp (RS232, USB, usw.) die benötigten Eigenschaften vorgibt (s. Abschnitt B.4). Die Verknüpfung der Schnittstelle mit dem Gerät, dem Betriebssystem und dem Treiber erfolgt über den „Connection Request“ (s. Abbildung 3.10) und dem zugehörigen DeviceConnectCommand (s. Abbildung 3.21). Dadurch wird die Device-Klasse (s. Abbildung 3.23) mit Werten gefüllt. Zu diesen Werten gehören unter anderem eine Gerätereferenz und eine beliebige Anzahl an Aliases, womit das Gerät jederzeit vom Hauptprozess identifiziert und gefunden werden kann. Zusätzlich wird das Gerät innerhalb des Verbindungsprozesses konfiguriert, d.h. individuelle Geräteparameter, aber auch Initialisierungskommandos festgelegt, welche beim „Verbinden“ einmalig ausgeführt werden sollen. Zuletzt existiert für jedes Gerät ein sog. *Listener*, welcher die Daten von den einzelnen Gerätekommandos (s. unten) zwischenspeichert und sie dem Hauptprozess zur Verfügung stellt. Alle an diesem Prozess beteiligten Klassen sind in Abbildung 3.24 dargestellt.

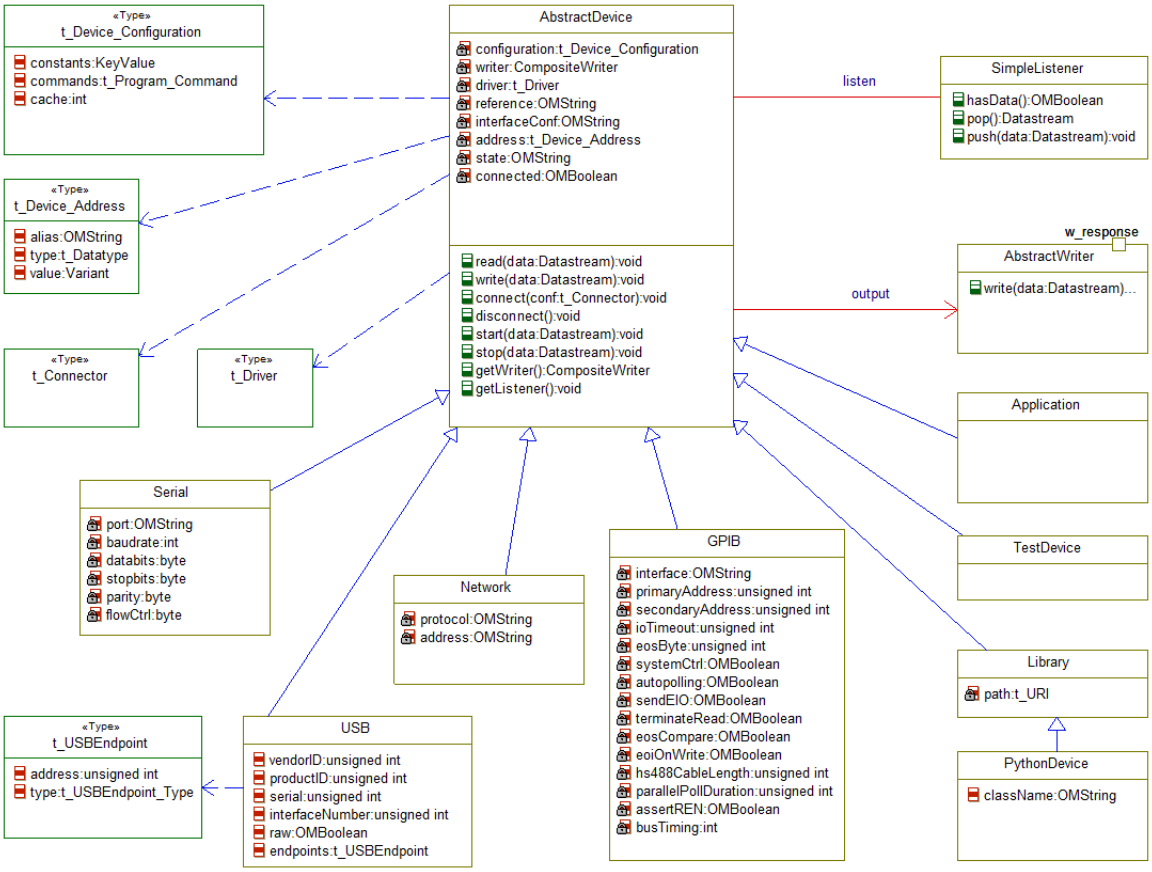


Abbildung (3.23)

Klassendiagramm eines Gerätes mit allen Attributen und Operationen.

Nach dem Aufbau der Verbindung kann Kommunikation mit diesem Gerät erfolgen, d.h. Daten an das Gerät gesendet und von diesem empfangen werden. Bereits während der Kommunikation können die Daten mit Hilfe von Skripten verarbeitet werden. Abbildung 3.25 zeigt alle am Kommunikationsprozess beteiligten Klassen.

Peripherie Deskriptor - Der Gerätetreiber

Der Gerätetreiber selbst ist eine (relativ große) Struktur, damit alle Elemente der Definition (s. Abschnitt B.3 im Anhang) gekapselt werden können. Interessanter als die Struktur selbst sind die Methoden, welche den Treiber verwenden. Dazu gehören auch einige Elemente, die aus Gründen der Übersichtlichkeit noch nicht in obigen Abbildungen 3.21 und 3.22 gezeigt worden sind. Eine einfache Kaskade, die gut zur Demonstration

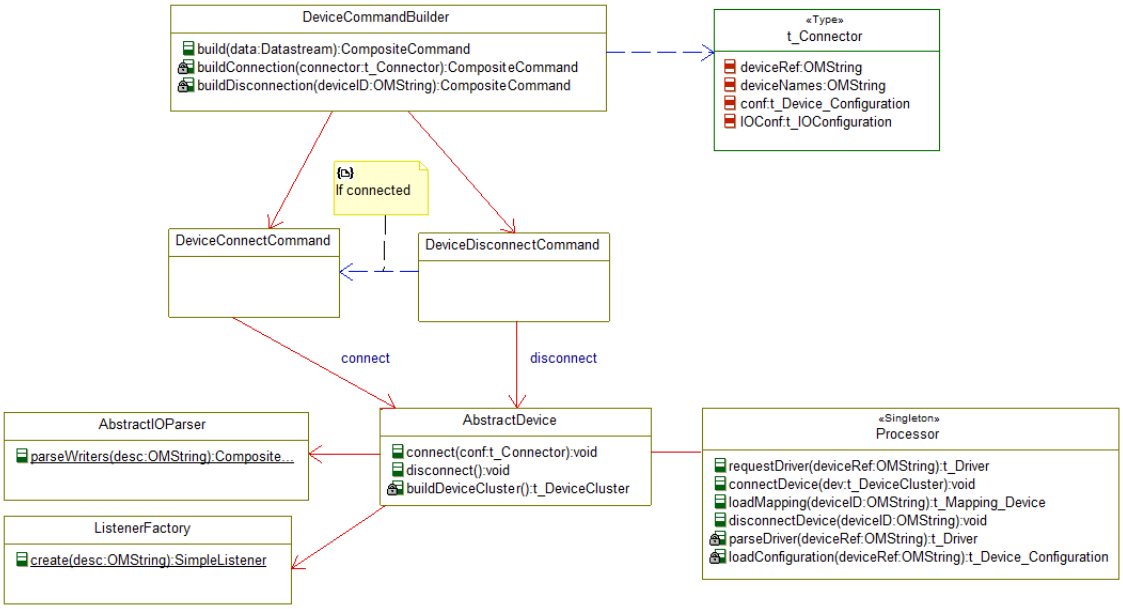


Abbildung (3.24)
Klassendiagramm der an der Verbindungsprozess beteiligten Modelle.

der Abhängigkeiten geeignet ist, wird durch den „Device Request Command“ (s. Abbildung 3.10) repräsentiert. Die Kaskade führt ein einzelnes Gerätekommando auf dem Gerät aus und liefert dessen resultierende Geräteantwort ohne Zwischenschritte an den Aufrufer (Benutzer) zurück. Dabei wird ein „DeviceExecuteCommand“ (s. Abbildung 3.21) erzeugt, welches einen Befehl an das Gerät sendet und im Anschluss die Antwort wieder abfragt. Letztere wird anschließend über die Systemroute an den Benutzer zurückgeliefert.
Eine weitaus komplexere Kaskade stellt die Ablaufsequenz dar und sie wird weiter unten in Abschnitt 3.4.3 besprochen.
Auf das Treibermodell wird mit Hilfe des DeviceCommandBuilders zugegriffen (s. Abbildung 3.26), welcher für die nachfolgende Gerätekommunikation die benötigten „DeviceCommands“ generiert (z.B. das obige DeviceExecuteCommand). Die dafür benötigten Informationen sind immer ein eindeutiger Gerätename (Gerätereferenz oder -alias), ein Kommandoname (oder -Alias) und die individuellen Kommandoparameter. Mit diesen Informationen kann der DeviceCommandBuilder den richtigen Treiber wählen, darin das angefragte Kommando suchen und den Token extrahieren. Die angegebenen Kommandoparameter werden anhand der im Treiber angegebenen Syntax überprüft, in den korrekten Datentyp übersetzt und ggf. neu kodiert (ASCII, Binär, usw.). Anschließend

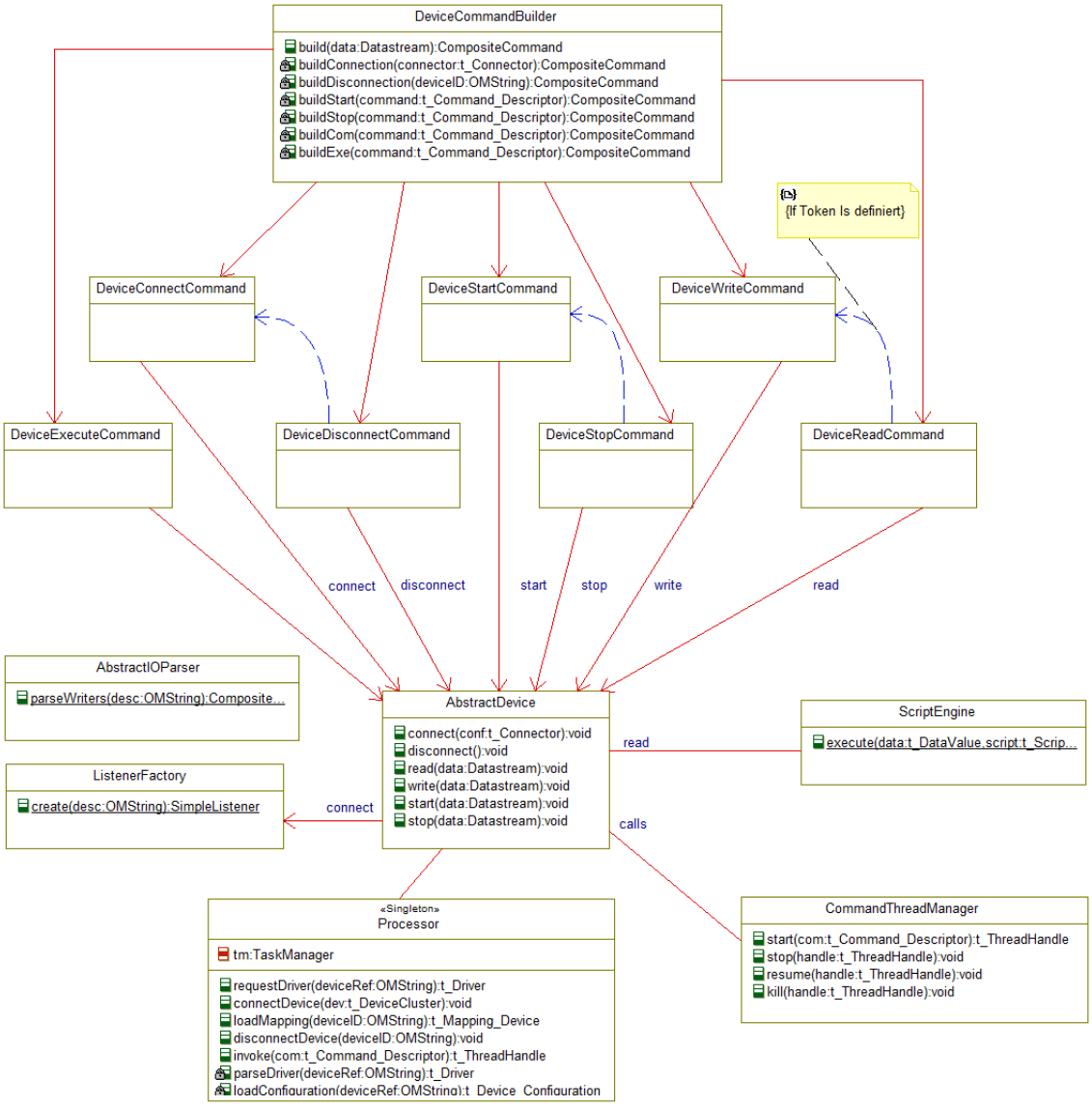


Abbildung (3.25)
Klassendiagramm der an der Gerätekommunikation beteiligten Modelle.

werden sie in die Platzhalter des Tokens eingefügt, nachdem dieser dieselbe Prozedur durchlaufen hat.

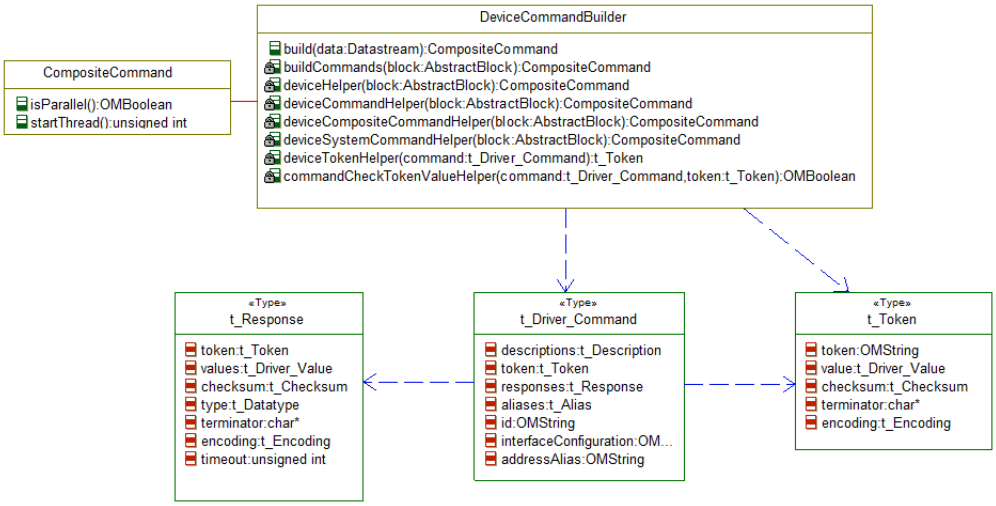


Abbildung (3.26)
Klassendiagramm des DeviceCommandBuilders zum Zugriff auf die Funktionalitäten des Treibers.

Templates

Writer, welche die TemplateBased-Schnittstelle (Marker Interface) implementieren, können Daten des Excalibur-Template Datentyps handhaben. Diese Daten resultieren, wenn eine spezielle XML-basierte Syntax (s. Abschnitt B.7 im Anhang) auf Gerätedaten angewendet wird. Abhängig vom verwendeten Writer kann über Templates auf die Formatierung der Daten und die Aufteilung jener auf die einzelnen Datensenken eingewirkt werden (s. Abbildung 3.27). Über diesen Mechanismus ist es möglich, auf einfache Weise Daten von verschiedenen Geräten für unterschiedliche Datensenken zu formatieren.

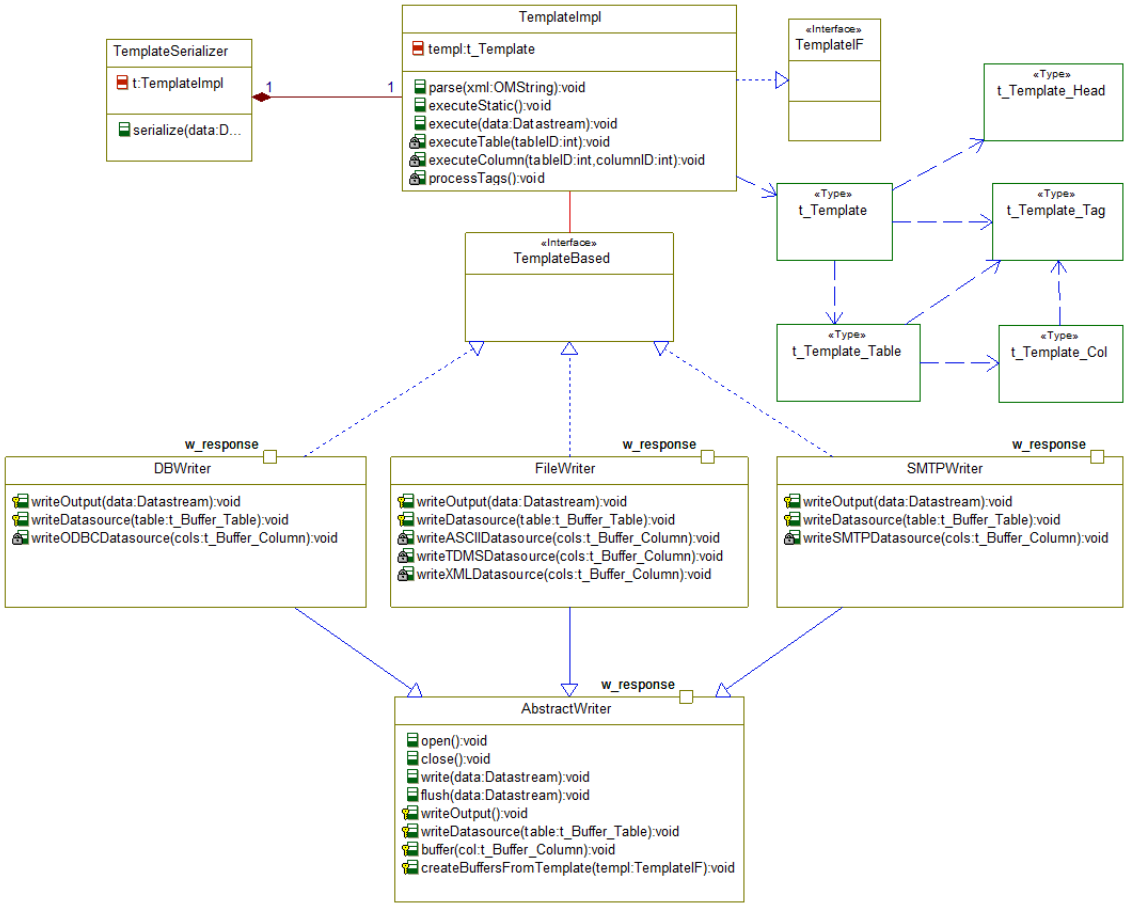


Abbildung (3.27)
Klassendiagramm der Elemente, die am Templating Mechanismus beteiligt sind.

Ablaufsequenz

Die Ablaufsequenz stellt die Kernkomponente für die Gerätekommunikation dar. Der zugehörige Prozess hat im Task-Manager einen eigenen Thread. Das hat den einfachen Grund, dass Ablaufsequenzen im Normalfall eine lange Prozessdauer haben, oftmals sogar eine endlose. Dieser Thread arbeitet schrittweise alle Anweisungen der Ablaufsequenz ab und erzeugt für jede einzelne davon einen neuen Task, welcher erneut die normale Kernprozesskette (Abschnitt 3.4.3) durchläuft. Wann die nächste Anweisung der Ablaufsequenz durchgeführt wird, ist abhängig von der Art der vorhergegangenen Anweisung. Sie kann direkt gestartet werden oder auf den vorhergegangenen Prozess warten. Im Normalfall wartet die aktuelle Anweisung bis die alte abgearbeitet ist (sequenzieller

Ablauf), woher auch der Name „Ablaufsequenz“ stammt (s. Abbildung 3.28). Im Falle von Parallelprozessen[33] wartet der Prozess nicht. Da dies aber vom Benutzer aufwendig zu synchronisieren ist, gibt es eine komfortablere Möglichkeit, parallele und wiederkehrende Anweisungen zu konstruieren. Mit Hilfe des „DeviceStartCommand“ kann jede Gerätekommando-Kaskade in einem weiteren separaten Thread gestartet werden. Dabei wird das enthaltene Gerätekommando in festgelegten Intervallen erneut in den Kernprozess (Abschnitt 3.4.3) einfügt und wieder ausgeführt.
Der zweite Kommandotyp, die *Systemkommandos*, werden im Normalfall schnell ausgeführt. Zu diesen zählen unter anderem alle Berechnungen von ablaufrelevanten Variablen (Zähler, Parameterumrechnungen und -anpassungen, usw.), Umformungen von Zeichenketten, aber auch das „Warten“ (bevor die nächste Anweisung ausgeführt wird). Gerade letzteres Systemkommando kann aber zu einem „Anhalten“ der Ablaufsequenz und auch zu einem Verlust des Zugriffs auf den Task führen. Falls hierfür ein typisches „Pausieren“ von Threads verwendet würde, wäre der Thread nicht mehr ansprechbar bis die Wartezeit abgelaufen ist. Aus diesem Grund wird ein anderes Vorgehen gewählt, bei welchem der Thread nicht pausiert wird. Hierbei wird ein Zeitstempel gesetzt und der Task-Status auf „Waiting“ gesetzt. Er durchläuft nun solange einen Waiting-Prozess, bis seine Zeit abgelaufen ist; anschließend wir der Status wieder auf „Running“ gesetzt. Diese Vorgehensweise kostet zwar mehr Prozessorlast und ein wenig zeitliche Präzision, da eine gewisse Zeit bis zur nächsten Zeitabfrage und dem Vergleich vergeht. Allerdings ist ein permanenter Zugriff auf den Task möglich und er kann beispielsweise bei Fehlkonfiguration der Ablaufsequenz (endlose Wartezeit) abgebrochen werden, ohne den Messserver abbrechen zu müssen.

[33] Wie es beispielsweise für den Fall gilt, dass mehrere Messgeräte existieren, die an verschiedenen Messpunkten zur selben Zeit eine Messung durchführen sollen

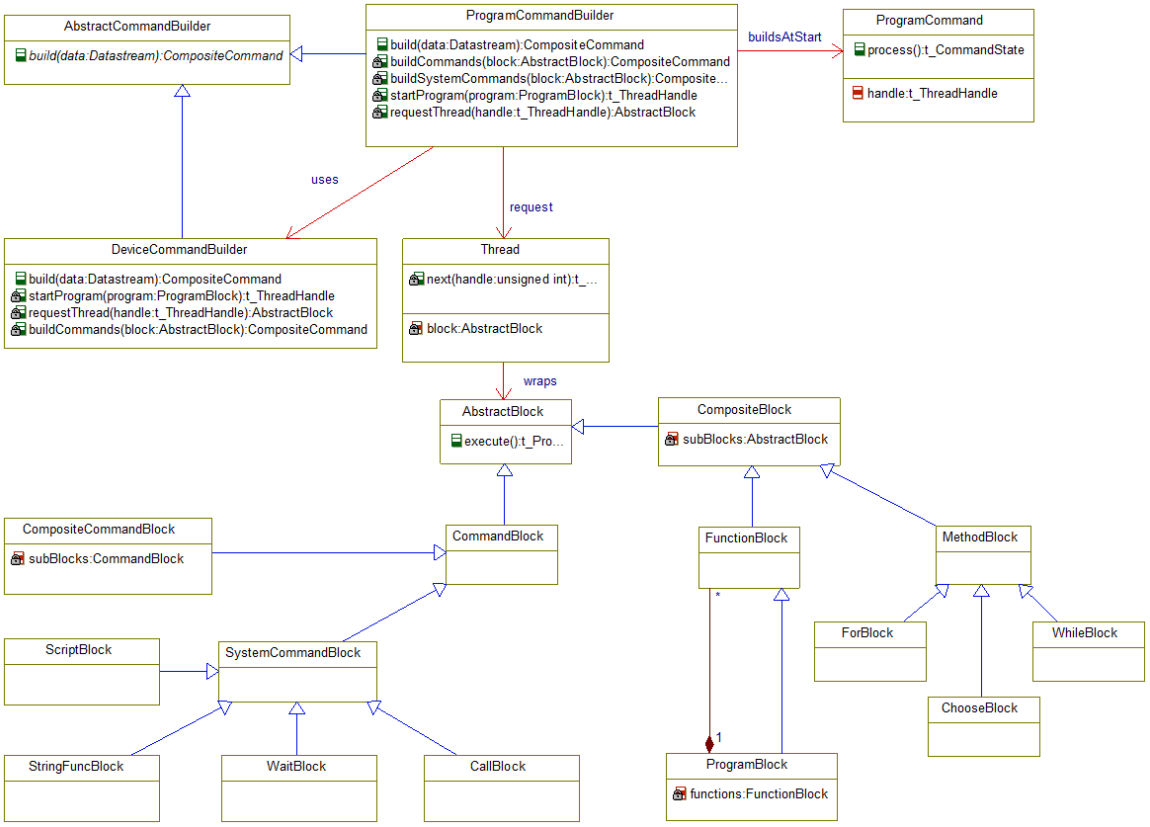


Abbildung (3.28)

Klassendiagramm der an der Ablaufsequenz beteiligten Elemente.

3.4.4. Dynamische Struktur des Excalibur Server

Die dynamische Struktur einer Software gibt die Interaktionen und den Ablauf der einzelnen Strukturelemente wieder. Dabei werden die oben eingeführten Klassen wiederverwendet und in eine logische Abfolge gebracht. Angefangen wird wieder beim Eingang einer Anfrage über die Reader, wobei die Serialisierer ein Modell erzeugt und dieses mit Werten gefüllt wird (s. Abbildung 3.29). Der Eingabeprozess wird hier anhand der abstrakten Klassen bzw. deren Objekten dargestellt und gilt repräsentativ für alle konkreten Instanzen der Klassen, die von obigen abgeleitet sind (AbstractReader, AbstractSerializer und AbstractFilter). In Worten ausgedrückt bedeutet dies, dass beim Erkennen einer Anfrage über eine konfigurierte, konkrete Reader-Instanz, die Daten gelesen werden, über den konfigurierten Serialisierer interpretiert und durch die Filter bearbeitet werden. Daraus wird im nächsten Schritt ein Task generiert, eine TaskID, eine Priori-

tät und ein Typ festgelegt und diese Eigenschaften in einem Datenstrom gesetzt. Der Datenstrom enthält nun alle Informationen, um die korrekten Kommandos für die auszuführenden Aktionen erstellen zu können.
Zum Verständnis der Kommunikation zwischen verschiedenen Threads, auf die im Weiteren eingegangen wird, gelten folgende Regeln. Da Threads parallele Prozesse darstellen, die unabhängig vom Hauptprozess der Applikation ausgeführt werden, erfolgt die Kommunikation asynchron. Dies erfolgt mit Hilfe von Queues, welche über eine Methode *push* zum Ablegen von Daten in die Queue und über eine Methode *pop* zum Abfragen (und Löschen) des (zuerst) abgelegten Datensatzes verfügen (FIFO-Prinzip, First-In-First-Out). Da dies in Excalibur immer der Fall ist, werden in Sequenzdiagrammen zur Bewahrung der Übersichtlichkeit ausschließlich die Threads gezeigt und die Methoden *push* und *pop* dem Thread zugeordnet, auch wenn die Kommunikation in der Realität über die Queues erfolgt.

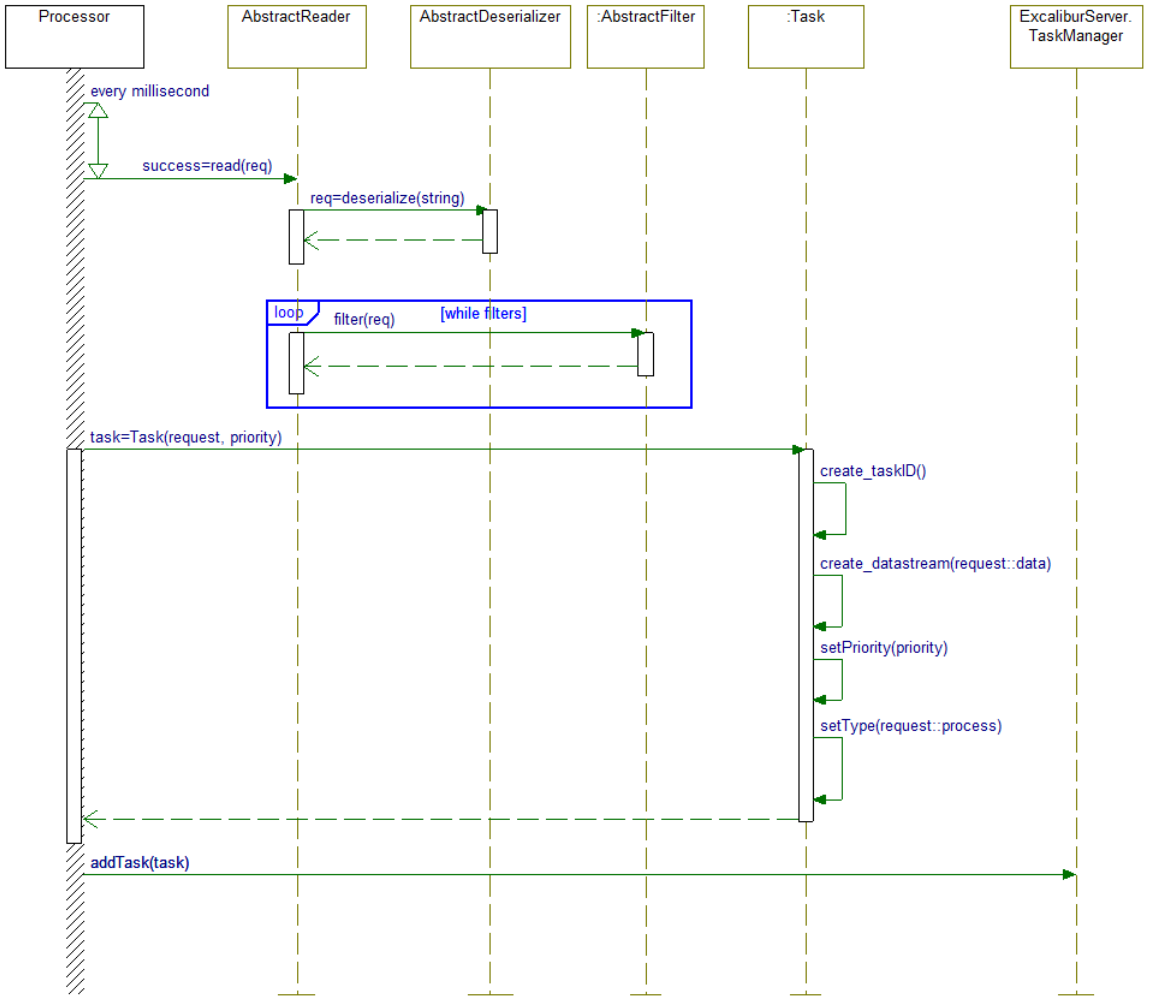


Abbildung (3.29)

Sequenzdiagramm zur Darstellung des Prozesses der Modellerzeugung bei Anfrage durch einen Benutzer.

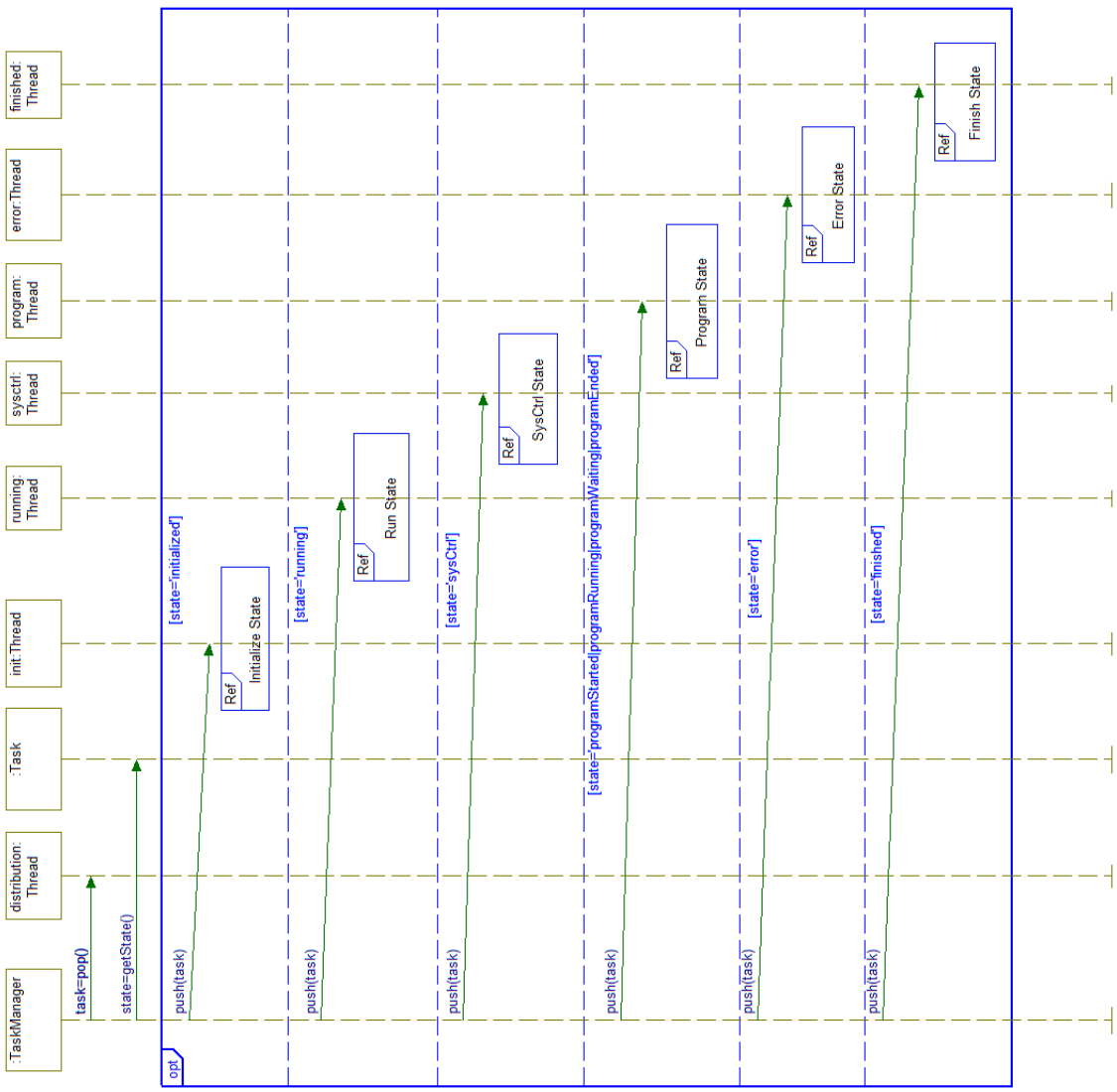


Abbildung (3.30)
Sequenzdiagramm zur Darstellung des Ablaufs eines Tasks innerhalb des Task-Managers.

Von der Anfrage bis zur Antwort

Im ersten Schritt nach dem Lesen wird ein Task generiert, welcher im Anschluss vorgegebene Stati des Task-Managers durchläuft (s. Abbildung 3.30). Die einzelnen Stati sind aber im Gegensatz zu einem typischen Zustandsautomaten nicht exklusiv, sondern liegen im Excalibur Server als separate Threads vor. Im Task-Manager können demnach mehrere Tasks in verschiedenen Stati gleichzeitig abgearbeitet werden. Die Kommunikation zwischen den einzelnen Threads erfolgt über Queues, wobei die erste Queue (bzw. der erste Thread) immer die *Initialisierung* darstellt. Sie ist in obiger Abbildung im ersten Abschnitt (*state=='initialized'*) innerhalb der referenzierten Interaktion mit dem Namen „Initialize State“ zu finden. Auf andere Interaktionen dieses Zustandsautomaten wird weiter unten eingegangen, wobei immer auf diese Abbildung verwiesen wird.
Während dieses Initialisierungsschritts erfolgt die Generierung von Kommandos über die korrespondierenden Builder, die anhand des im Datenstrom angegebenen Prozesses vom AbstractCommandBuilder ausgewählt werden (s. Abbildung 3.31). Den AbstractBuilder kann man demnach als *Factory* ansehen, um die einzelnen konkreten Builder zu instanzieren. Die Funktionsweise des konkreten Builders ist abhängig vom zugrunde liegenden Prozess. Sie soll exemplarisch anhand des DeviceCommandBuilders in Abbildung 3.32 dargelegt werden, weil dieser als einer der fundamentalen Konstruktionsprozesse innerhalb des Messservers gilt und er für die Konstruktion aller für die Geräteaktionen zuständigen Kommandos verantwortlich ist. Alle anderen konkreten CommandBuilder verhalten sich entsprechend und sollen hier nur erwähnt bleiben.
Der Initialisierungsvorgang für Gerätekommandos wird durch fünf verschiedene Prozesse ausgelöst: *connect*, *disconnect*, *execute*, *start* und *stop*. Prinzipiell ist der Vorgang immer derselbe, weshalb auch nur der Konstruktionsprozess des DeviceCommandBuilder näher gezeigt werden soll. Der Parameter „process“ im Datenstrom gibt an, welcher CommandBuilder (hier der DeviceCommandBuilder) und welche Methode innerhalb des CommandBuilder ausgeführt werden soll. Alle für die Konstruktion der Gerätekommandos benötigten Daten stammen aus dem Datenstrom und können direkt extrahiert werden.
Im Connect-Prozess beispielsweise werden meistens mehrere Kommandos konstruiert, zumindest aber immer das „DeviceConnectCommand“, welches die tatsächliche Verbindung mit den bereits mehrfach erwähnten Aktionen durchführen soll. Weiterhin werden der Reihe nach die Schnittstellendaten geladen und eine passendes Schnittstellenobjekt erzeugt (s. Abbildung 3.23), eine Treiberinstanz erzeugt und dem Objekt angefügt, die

Konfiguration geparst und dem Objekt angefügt und ggf. Gerätekommandos erzeugt, welche bei der Initialisierung ausgeführt werden sollen. Alle Kommandos werden zusammen in ein Kompositionskommando gekapselt und zurückgegeben.
In den anderen Fällen gestaltet sich der Ablauf einfacher, da meistens nur ein (maximal zwei) Kommandoobjekte erzeugt und konfiguriert werden müssen. Unter der Konfiguration versteht man hierbei (wie auch im Falle aller Kommandos des Initialisierungsprozesses) das Setzen einer Zieladresse (die Gerätereferenz) und das Anfügen des Datenstroms.

Der nächste Schritt, der *Run*-State, arbeitet nun der Reihe nach alle Kommandos ab (s. Abbildung 3.33). Dieser Prozess erfolgt sequentiell. Da es sich bei den einzelnen Kommandos aber immer um ein (abgewandeltes) *Kommando-Pattern* in Kombination mit einem *Kompositum-Pattern*[GHJV98, Gra98] handelt, können die Kommandos von den konkreten CommandBuildern beliebig geschachtelt werden. Der Aufruf erfolgt dann vom Task-Manager auf der höchsten Hierarchie-Ebene, indem die *process*-Methode aufgerufen wird. Je nachdem, ob es sich um ein (weiteres) Kompositum oder ein Blatt-Element handelt, wird der Aufruf entweder an die Kindelemente weitergereicht oder die zugrunde liegende Funktionalität aufgerufen. Darüber hinaus kann im Kompositum das *parallel*-Flag gesetzt werden, so dass alle Blatt-Elemente des Kompositums parallel ausgeführt werden. Exemplarisch sei dies anhand des parallelisierten Auslesens von Geräten in Abbildung 3.34 gezeigt.
Der konkrete CommandBuilder legt auch fest, welche der Kommandos die Geräteantwort zurückliefern sollen und über welche Route dies erfolgen soll (System oder Gerät). Dabei werden die während des Ausführens der Kommandos erzeugten Daten in einen Datenstrom verpackt und per System-Event an den Hauptprozess gesendet. Der Hauptprozess durchläuft permanent alle Geräte (mitunter auch den als Gerät geltenden Excalibur Server selbst) und durchsucht sie nach neuen Daten (*Dirty Flag* gesetzt). Im Falle von empfangenen Daten werden diese den korrespondierenden Writern zugeführt (s. Abbildung 3.35).
Zuletzt werden die Tasks immer im Finished- oder Error-Thread abgelegt. Ersterer gibt dabei verwendete Ressourcen wieder frei und löscht alle am Task beteiligten Objekte und Referenzen. Der Error-Thread interpretiert den aufgetretenen Fehler anhand eines gesetzten Fehlerstatus (*Warning, Exception, Error, Critical*) und gibt dem Benutzer den Fehler in Form einer *ErrorResponse* zurück (per Response oder Notification). Dabei werden gestartete Ablaufsequenzen bei den Stati *Warning* und *Exception* weitergeführt, bei den restlichen abgebrochen. Zusätzlich kann in der Excalibur-Konfiguration festgelegt

werden, ob beim Auftreten des Status *Critical* der Excalibur Server beendet wird.

Steuerung von Laborgeräten

Unter der Steuerung von Laborgeräten versteht man das Senden und Empfangen von Daten an/von den Geräten. Dazu müssen die Excalibur-Treiber einbezogen werden, welche die Befehle kapseln, mit denen die Gerätekommunikation erfolgen kann. Außerdem legen sie über eine vorgegebene Grammatik fest, wie die empfangenen Gerätedaten interpretiert werden müssen.
Die Kommunikation kann auf zwei Arten erfolgen. Einerseits kann dies über den in Abbildung 3.21 dargestellten Prozess genau ein Kommando ausgeführt werden, welches direkt die Antwort wieder über die Systemroute an den Benutzer zurück liefert. Andererseits kann die Gerätekommunikation im Rahmen von Ablaufsequenzen erfolgen, bei welchen die Daten (meistens) über die Geräteroute weitergegeben werden und gegebenenfalls mit Hilfe von Transformern verarbeitet und mit Hilfe verschiedener Writer unterschiedlichen Datensenken zugeführt werden können. In den folgenden Kapiteln wird auf letzteren Prozess eingegangen.
Voraussetzung für die Ablaufsequenz ist es, dass ein ProgramRequest (s. Abbildung 3.10) über die Excalibur-Eingabe eintrifft. Der korrespondierende Deserialisierer konstruiert aus diesem Request das Datenmodell für einen Ablauf, bestehend aus den *Block*-Klassen (s. Abbildung 3.28). Die einzelnen Klassenbezeichnung lassen bereits auf die Bedeutung der Funktionalität schließen. Eine kurze Erklärung der einzelnen Blöcke gibt Tabelle 3.7. Die erzeugten Block-Objekte werden im Initialisierungsprozess des TaskManagers innerhalb eines ProgramCommand gespeichert, wofür der ProgramCommandBuilder verantwortlich ist. Dies erfolgt im TaskManager im Program-Thread. Legt der TaskManager einen Task in der ProgramQueue ab, wird dieser in einen weiteren Zustandsautomaten bestehend aus drei Stati weitergeleitet (s. Abbildung 3.36 und Abbildung 3.37). Der erste Status erzeugt einen weiteren Thread, den Ablauf-Thread, in welcher die Ablaufsequenz gespeichert wird. Das erfolgt über den ProgramCommandBuilder mit der Erzeugung eines ProgramCommand. Letzteres speichert eine Referenz (Handle) auf den erzeugten Thread, außerdem dessen enthaltene Kommunikations-Queue. Der Thread wird nun gestartet und er läuft so lange im Hintergrund, bis die Ablaufsequenz abgearbeitet ist oder durch einen Fehler abgebrochen wird. Nach Ausführung dieser Prozesse wird das ProgramCommand im Task gespeichert und der nächste Status, *ProgramRunning*, gesetzt. Ab diesem Zeitpunkt, wird die Ablaufsequenz Zeile für Zeile, d.h. Block für Block, ab-

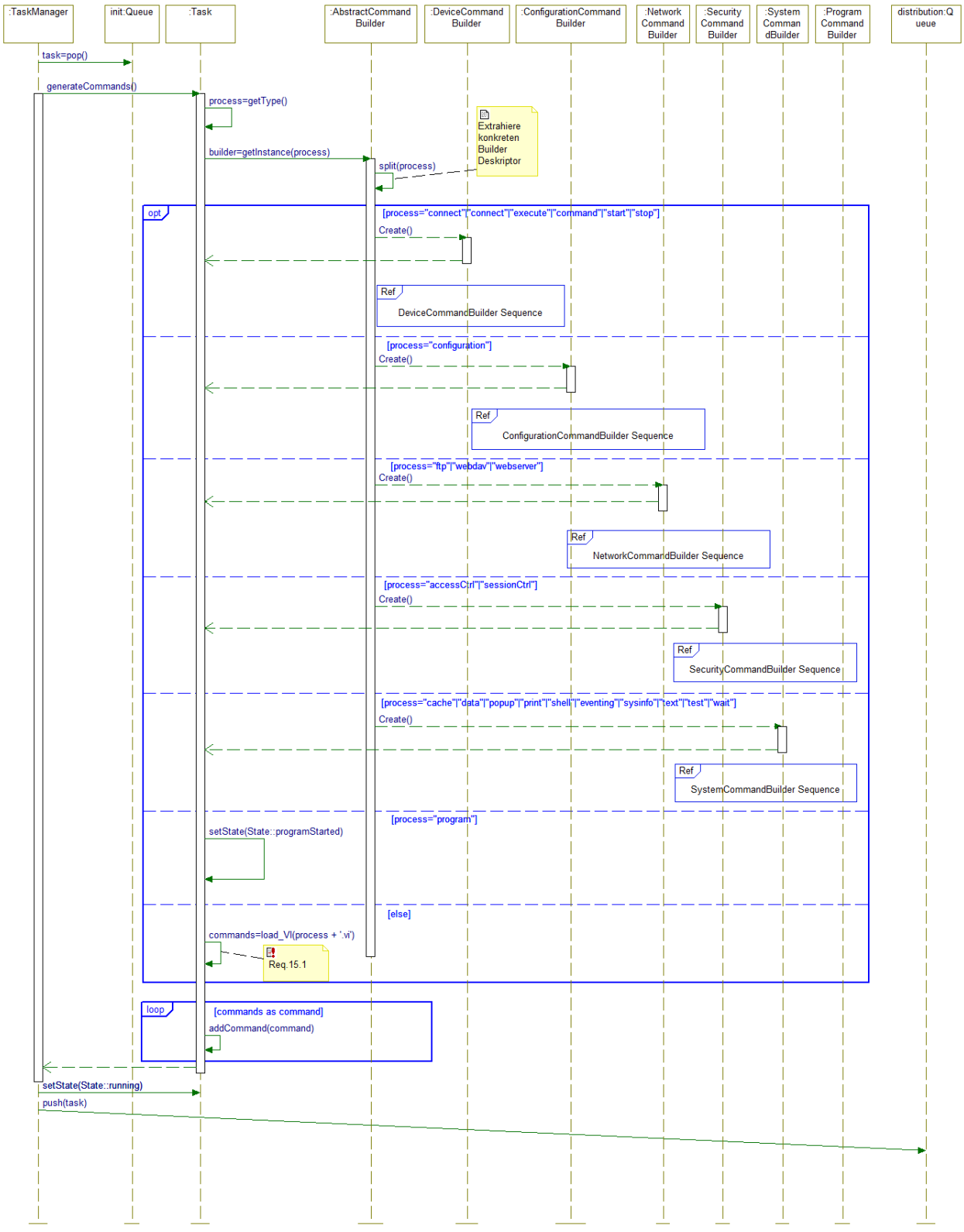


Abbildung (3.31)

Sequenzdiagramm zur Darstellung des Initialisierungs-Prozesses, bei welchem Kommandos über geeignete Builder generiert werden.

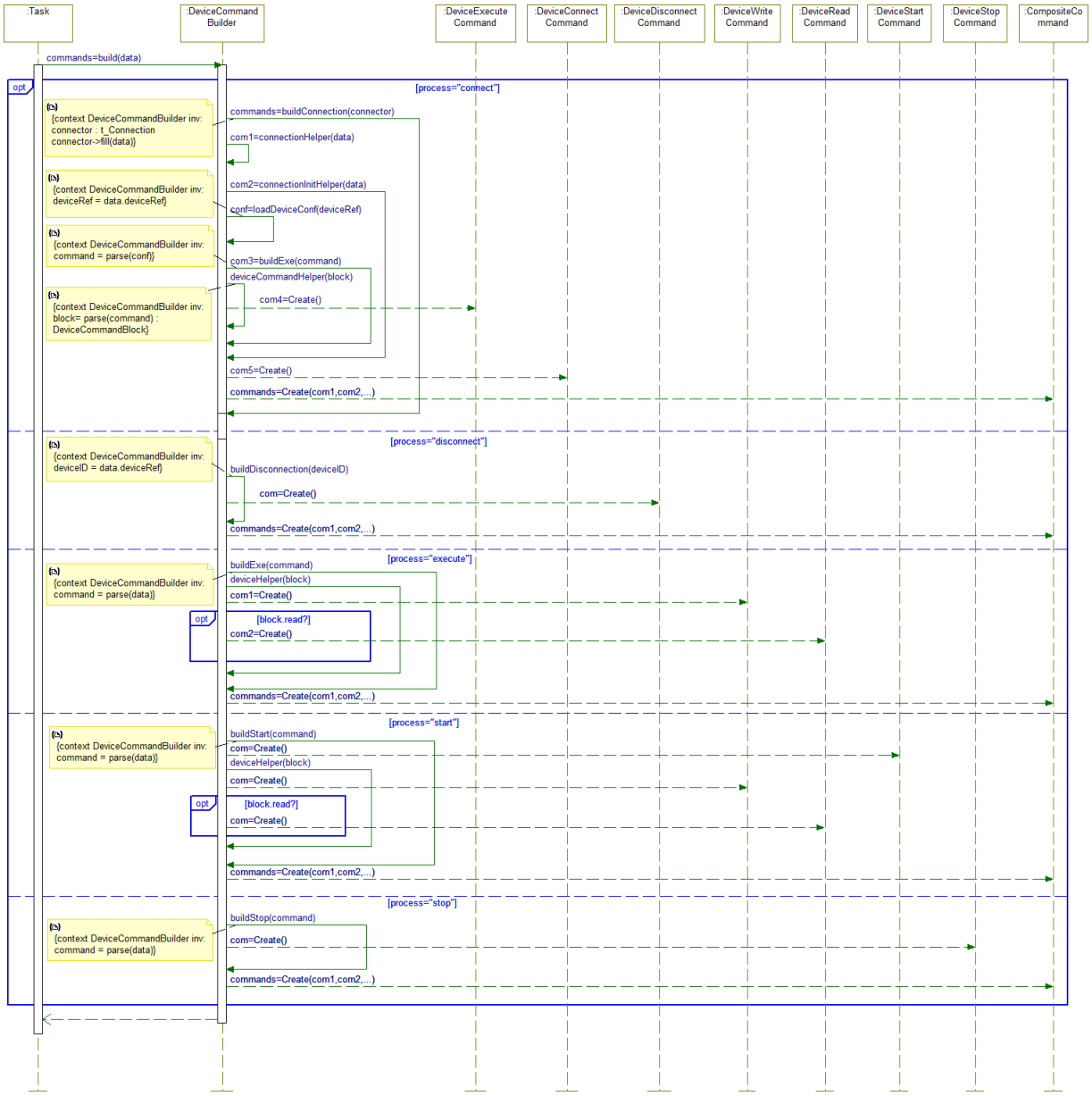


Abbildung (3.32)

Sequenzdiagramm zur Darstellung des DeviceCommandBuilder Prozesses, welcher die Kommandos für die Kommunikation mit Geräten konstruiert und initialisiert.

gearbeitet. Dabei beginnt der Ablauf-Thread beim obersten Element des Block-Baums und sucht das erste Blatt-Element. Existieren auf dem Pfad aufzulösende Bedingungen (z.B. innerhalb eines WhileBlocks), versucht der Thread-Prozess diese zu entscheiden.

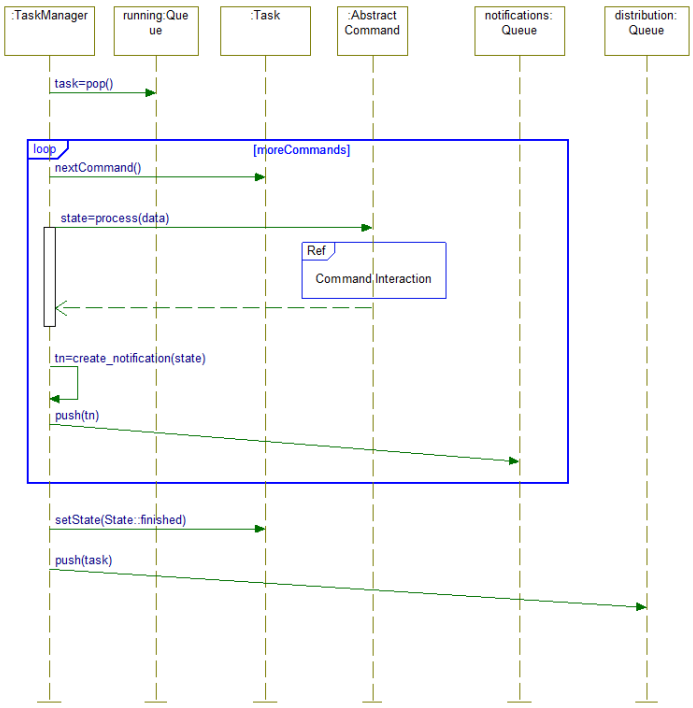


Abbildung (3.33)

Sequenzdiagramm zur Darstellung des Run-Prozesses, in welchem alle Kommandos des Tasks abgearbeitet werden.

Sind alle benötigten Information vorhanden und eine Entscheidung kann getroffen werden, wird der Pfad weiter verfolgt. Kann eine Entscheidung auf dem Pfad nicht mit den vorhandenen Informationen getroffen werden oder wird das Blatt-Element erreicht, setzt der Ablauf-Thread an dieser Position einen Ablaufsequenz-Zeiger. Hier bleibt der Thread stehen, bis der Program-Thread eine Anfrage startet. Bei jeder Anfrage versucht der Thread, eventuell nicht getroffene Entscheidungen von Bedingungen zu lösen und gibt dem Program-Thread entweder den Status *ProgramRunning* oder *ProgramWaiting* zurück, falls die Entscheidung weiterhin getroffen werden muss.
Währenddessen befindet sich der Program-Thread im Zustand *ProgramRunning*. Bei jedem Durchlauf sendet er eine Anfrage an den Ablauf-Thread, um den aktuellen Block zu erhalten. Dies erfolgt so lange, bis der Ablauf-Thread den Program-Status *ProgramEnded* zurückgibt. Solange gültige Blöcke zurückgegeben werden, gibt es zwei mögliche Szenarien: Das Block kann direkt ausgeführt werden oder der Block wird gestartet.
Im ersten Fall wird der Block dem ProgramCommandBuilder übergeben, welche den Kommandotyp anhand einer Lookup Tabelle vergleicht. Daraus wird entschieden, ob es

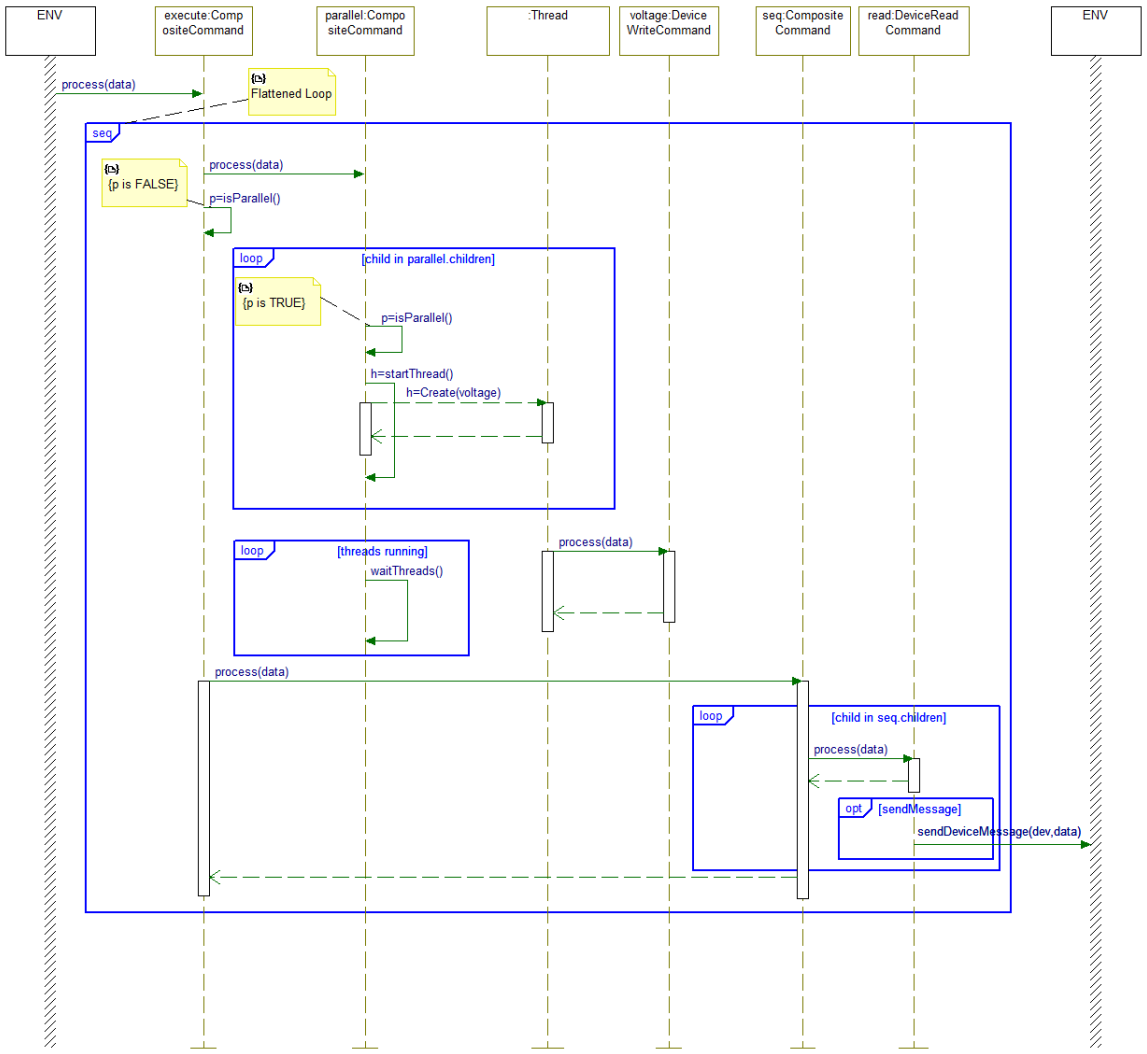


Abbildung (3.34)

Sequenzdiagramm zur Darstellung von parallelen und sequentiellen Prozessen.

sich um ein Geräte- oder ein System- bzw. Strukturkommando handelt.
In letzterem Fall wird der Block in einen weiteren Thread abgelegt, der in regelmäßigen Intervallen ausgeführt wird.
Mit diesen beiden Fällen werden sich die nächsten beiden Abschnitte beschäftigen.

Geräte- und Systemkommandos Hat der Program-Thread einen gültigen Block empfangen und das zugehörige Kommando soll direkt ausgeführt werden, entscheidet der

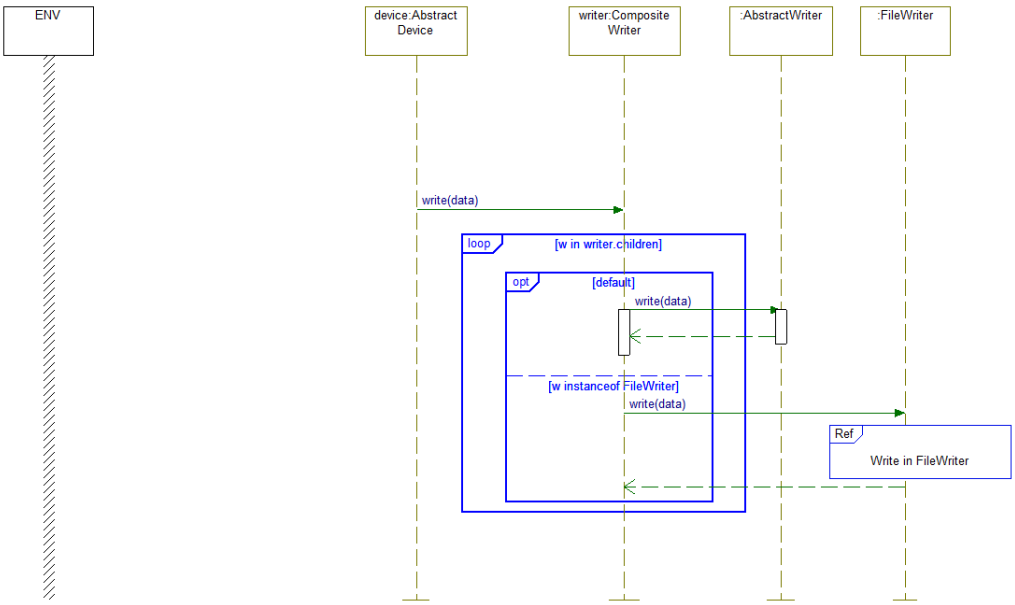


Abbildung (3.35)

Sequenzdiagramm zur Darstellung des Schreibprozesses in die Writer der Geräte (inkl. Excalibur Server)

Blocktyp, über welchen CommandBuilder der Konstruktionsprozess abläuft. Handelt es sich um einen CommandBlock, wird der DeviceCommandBuilder herangezogen (s. Abbildung 3.38). Dabei werden alle Informationen des CommandBlocks extrahiert, u.a. der Gerätename (bzw. dessen Alias). Anhand des Gerätenamens (oder indirekt über den Alias) kann der zugehörige Excalibur-Treiber gefunden werden und, über die restlichen Informationen des Blocks, das angefragte Kommando und dessen Übergabeparameter. Letztere werden vor dem Einsatz durch eine Plausibilitätskontrolle überprüft und gegebenenfalls mit Standardwerten vervollständigt. Daraus können im Anschluss die benötigten Schreib- und Lesekommandos erzeugt werden.
In allen anderen Fällen übernimmt der SystemCommandBuilder die Konstruktion der Kommandos. Handelt es sich um ein SystemCommandBlock (oder einen Abkömmling) davon, wird anhand des Block-Typs auf das korrekte SystemCommand geschlossen. Eventuelle Übergabeparameter werden wieder über eine Plausibilitätskontrolle überprüft und im SystemCommand gesetzt. Handelt es sich hingegen um einen CompositeCommandBlock (oder einem Abkömmling), wird der Prozess rekursiv wiederholt bis keine Kompositionen mehr vorhanden sind. Unter letztere fallen u.a. alle MethodBlocks wie der WhileBlock, der ChooseBlock oder der ForBlock.

Tabelle (3.7)
Übersicht der einzelnen Block-Klassen einer Ablaufsequenz.

Klassenname	Bedeutung
CommandBlock	Kapselt ein Gerätekommando und alle dazugehörigen Eigenschaften, darunter den Gerätenamen (oder -Alias), den Kommandonamen (oder -Alias) und die benötigten Übergabeparameter.
CompositeCommandBlock	Kapelt Geräte-Kompositionskommandos, darunter alles referenzierten Gerätekommandos (s. CommandBlock) und Skripte, die angewendet werden sollen.
CompositeBock	Kann mehrere andere Blöcke kapseln.
FunctionBlock	Stellt eine Funktion gängiger Programmiersprachen dar. Sie enthalten andere Blöcke und können mit Hilfe des CallBlocks in einem Ablauf referenziert und ausgeführt werden.
MethodBlock	Stellt eine Überstruktur für die internen Methoden ForBlock, WhileBlock und IfElseBlock dar.
ForBlock	Enthält andere Blöcke und wiederholt deren Ausführung (in der angegebenen Reihenfolge) für eine angegebene Anzahl an Durchläufen.
WhileBlock	Entspricht dem ForBlock, außer dass die Ausführung solange erfolgt, bis eine angegebenen Bedingung *wahr* wird.
ChooseBlock	Enthält andere Blöcke, die abhängig von einer (oder mehrerer) Bedingungen ausgeführt werden können. Er kann beliebig viele Bedingungen und eine (optionale) Alternativbedingung enthalten. Dieser Block entspricht den If-Else-Strukturen gängiger Programmiersprachen dar.
CallBlock	Referenziert einen FunctionBlock im Ablauf.
ScriptBlock	Enthält Excalibur-Skripte, welche auf Daten innerhalb einer Ablaufsequenz angewendet werden sollen.
StringFuncBlock	Enthält gängige Stringbearbeitungsfunktionalitäten zum Bearbeiten von Zeichenketten (s. Dokumentation).
WaitBlock	Stellt einen Wartemechanismus für den Ablauf dar. Hiermit kann die Ablaufsequenz für den angegebenen Zeitraum gestoppt werden.

Startbare Kommandos Soll das über den ProgramCommandBuilder konstruierte Gerätekommando in einem bestimmten Intervall wiederholt ausgeführt werden, wird das Kommando anstatt es in einen neuen Task zu verpacken, in einen weiteren Thread, dem

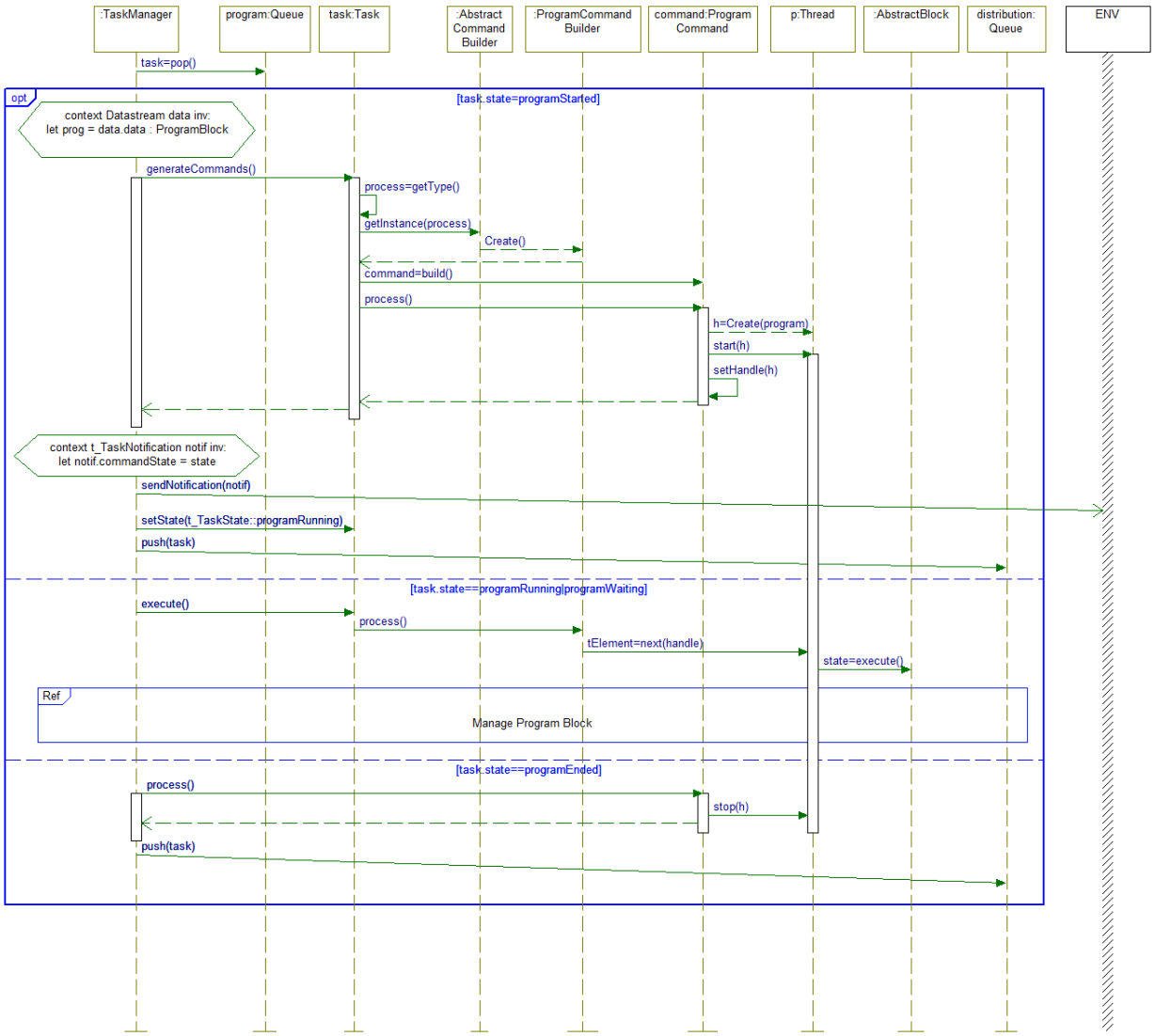


Abbildung (3.36)

Sequenzdiagramm zur Darstellung des ProgramThreads und dessen Zustandsautomaten.

Started-Thread, verschoben (s. auch Abbildung 3.37). Der TaskManager fragt nun permanent diesen Thread ab und vergleicht die Ausführungszeiten mit dem angegebenen Intervall. Ist das Intervall erreicht, wird das Kommando wiederum in einen Task verpackt und zur Bearbeitung dem Running-Thread übergeben (Abbildung 3.39).

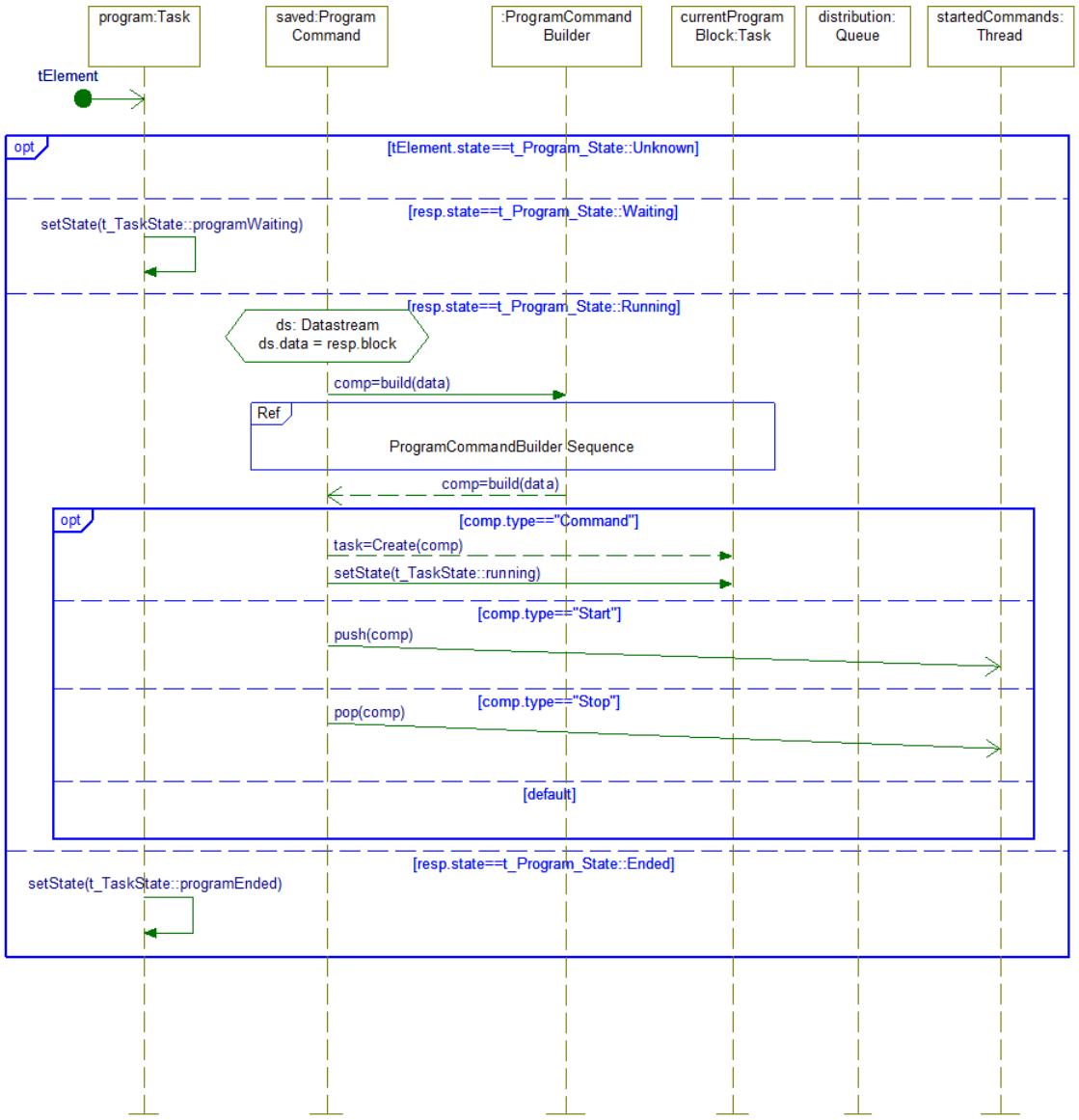


Abbildung (3.37)

Sequenzdiagramm zur Darstellung des Ablaufs eines direkt ausführbaren Blocks.

Abschließende Bemerkungen Die vorgestellten Abläufe sollen nur exemplarisch den Hauptprozess des Excalibur Server wiedergeben. Spezielle Blöcke wie der ScriptBlock oder den StringFuncBlock sind hier nicht besprochen worden, können aber in der Dokumentation nachgelesen werden. Das vorgestellte Konzept ist modular aufgebaut und kann jederzeit um weitere Block-Klassen erweitert werden, solange sie von einer der Basis-

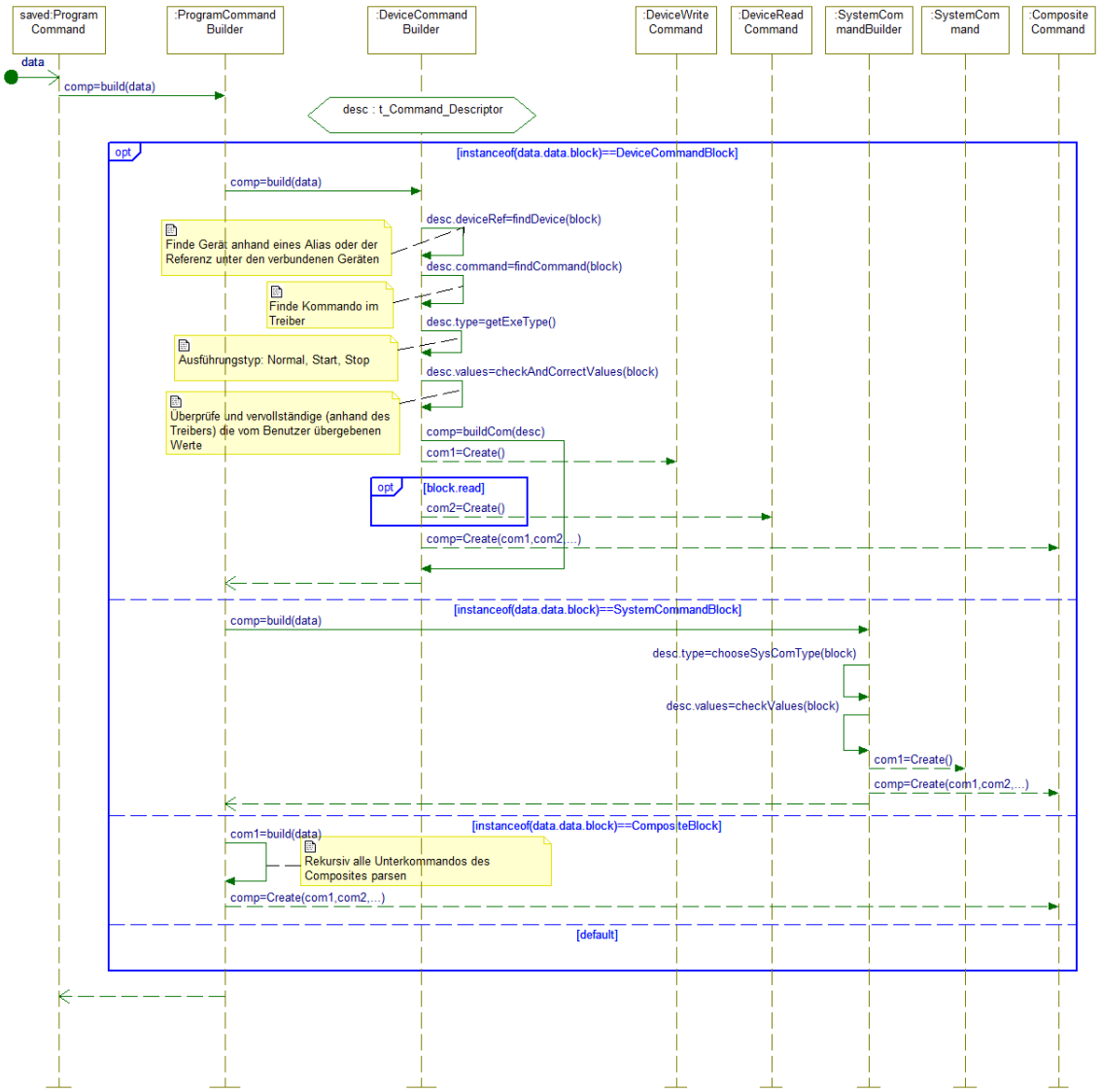


Abbildung (3.38)

Sequenzdiagramm zur Darstellung des ProgramCommandBuilders und des Konstruktionsprozesses für Ablaufkommandos.

klassen (CommandBlock, SystemCommandBlock, CompositeBlock) abgeleitet werden. Dadurch lässt sich die Funktionalität des Excalibur Server um weitere Strukturelemente erweitern.

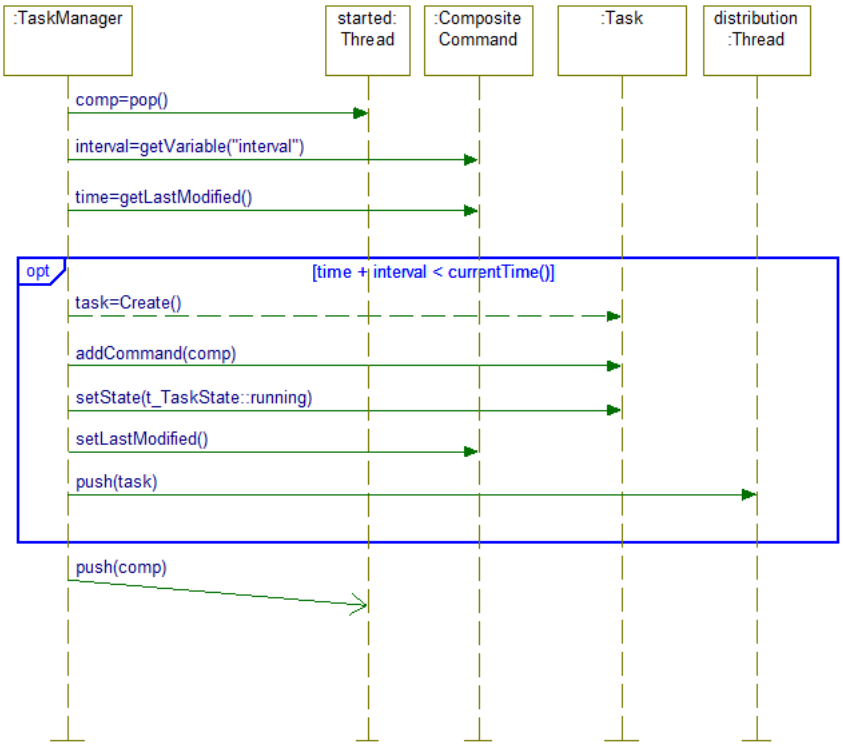


Abbildung (3.39)

Sequenzdiagramm zur Darstellung des Konstruktionsprozesses für startbare Kommandos.

Datenausgabe

Die Datenausgabe über die Writer ist bereits in Abbildung 3.35 eingeführt worden. Sie soll nun exemplarisch anhand eines CompositeWriters mit enthaltenem FileWriter erfolgen. Dies entspricht in obiger Abbildung 3.35 der Interaktion „Write in FileWriter“, welche beispielhaft den Schreib- und Transformationsprozess anhand von Dateioperationen demonstrieren soll. Die Datenausgabe soll deshalb über einen DataTransformer erfolgen und über einen TemplateSerializer serialisiert werden.
Erreicht in diesem Szenario ein Datensatz den CompositeWriter, generiert von einem beliebigen Gerät, ruft der Prozessor die *write*-Methode des im korrespondierenden Gerät enthaltenen Writers auf. Dieser durchläuft alle seine Kindelemente und ruft deren *write*-Methode auf. Anschließend übergibt der Writer die empfangenen Daten an seine Transformer, welche der Reihe nach durchlaufen und deren *transform*-Methode aufgerufen werden. Nachdem alle Transformer die Daten entweder bearbeitet oder sie verworfen

haben (falls auf den im Datenstrom gesetzten Kanälen nicht gelauscht werden soll), serialisiert der gesetzte Serialisierer die Daten in das gewünschte Ausgabeformat. Im Falle des hier angenommenen FileWriter wird ein enthaltener DataTransformer und ein TemplateSerializer angenommen. Auf deren Interaktionen wird in den nächsten Kapiteln eingegangen.
Die formatierten Daten werden zuletzt in die konfigurierte, vom Writer-Typ abhängige Datensenke geschrieben (hier: eine Datei, s. Abbildung 3.40). Der Schreibprozess selbst läuft für jeden Writer-Typ ähnlich ab und wird mit Hilfe eines Zustandsautomaten realisiert, der im AbstractWriter definiert ist. Zu Beginn einer Ablaufsequenz wird die Verbindung zur zugrunde liegenden Datensenke hergestellt (hier: Datei geöffnet). Mit jedem Aufruf des *write*-Befehls erfolgt nun die Ausgabe über die geöffnete Verbindung, abhängig vom Datenübertragungsprotokoll. Wird die Ablaufsequenz beendet (Letztes Kommando abgearbeitet oder durch Fehler abgebrochen), so werden auch alle Verbindungen zur Datensenke getrennt (hier: Datei geschlossen, s. Abbildung 3.41).

Daten Evaluation Die in die Writer zu schreibenden Daten sind in den meisten Fällen Rohdaten, d.h. unverändert von den Geräte weitergeleitet worden. Um aus den Daten Informationen zu gewinnen, müssen mathematische Operationen auf diese angewendet werden. Allgemeiner gesprochen heißt dies, dass Daten transformiert werden können. Hierfür sind die *Transformer* verantwortlich, die in den Writern konfiguriert werden können. Es können beliebig viele davon konfiguriert werden, was im Umkehrschluss bedeutet, dass alle über dieselbe Ein- und Ausgabestruktur verfügen müssen, nämlich einen Excalibur-DataValue (s. Abbildung 3.19). Abhängig vom Transformer-Typ werden verschiedenen Operationen auf die Daten angewendet, welche über die Eigenschaften der jeweiligen Transformer, meist in Form von Scripting-Anweisungen, konfiguriert werden können. Im Falle des hier betrachteten DataTransformers können mathematische Skripte definiert werden, die über die ScriptEngine ausgewertet werden (s. Abbildung 3.42 mit statischer Struktur in Abbildung 3.43). Dazu werden die im Datenstrom enthaltenen Daten, welche von den Geräten in einen Excalibur-DataValue gekapselt worden sind, mit Hilfe textbasierter Skripts ausgewertet. Diese Script-Typen enthalten Eingabe- und Ausgabevariablen, die über verschiedene Mechanismen gesetzt werden können, z.B. aus dem Excalibur-Cache, Kanälen oder direkt über Parameter bestehend aus Schlüssel-Wert-Paaren (Key-Value-Pairs). Innerhalb der „Skript-Texte“ der Typen werden diese Variablen über die SkriptEngine in das Eingabeformat der durch die ScriptGateways repräsentierten Skriptsprache (Matlab-Script, Octave-Script oder Python Quelltext) übersetzt.

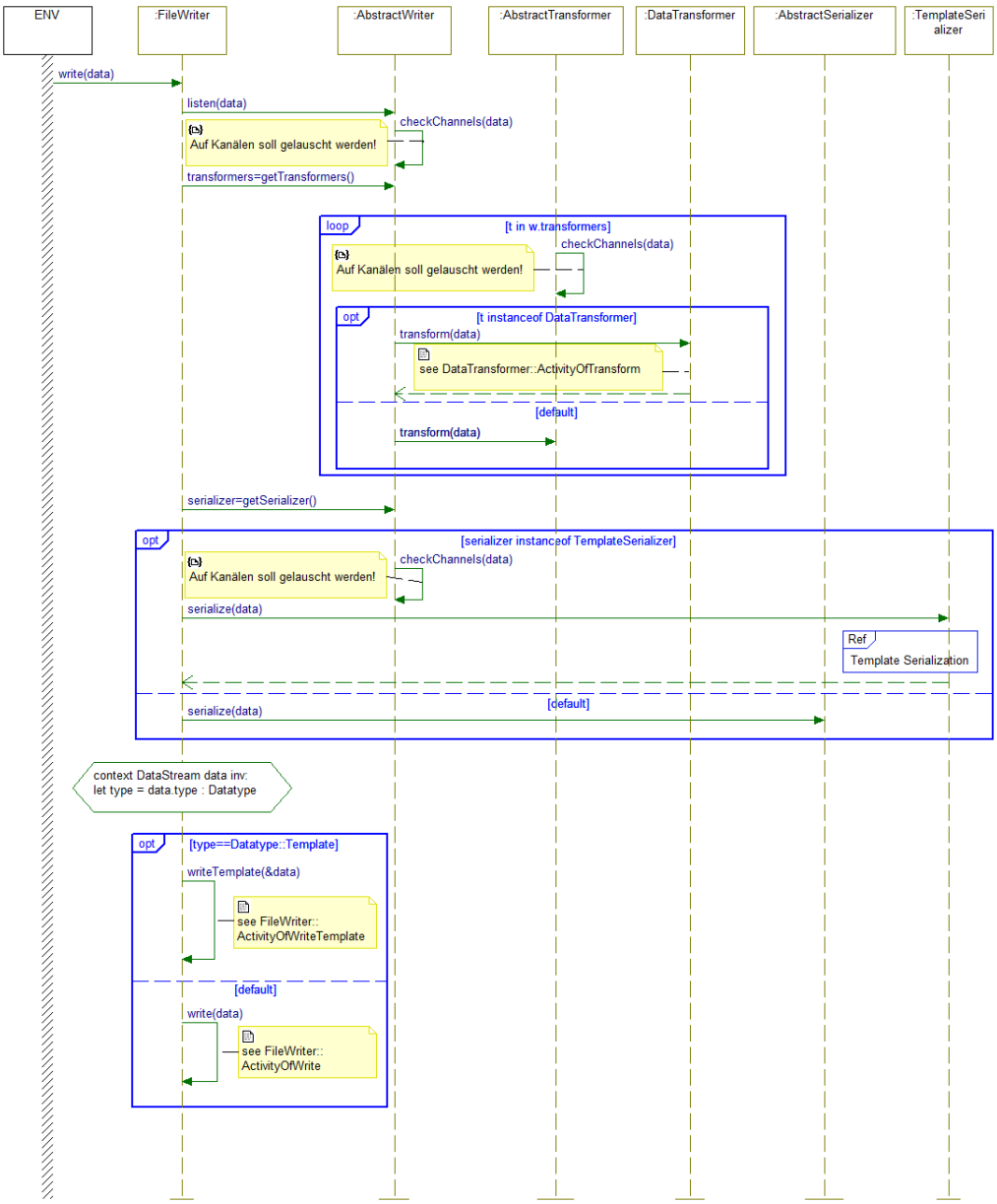


Abbildung (3.40)

Sequenzdiagramm zur Darstellung des Schreibprozesses von Writern anhand des konkreten Beispiels eines FileWriters.

Nach der Auswertung über den Gateway wird das Ergebnis in die ebenso im Script-Typ definierten Ausgabevariablen gespeichert und wieder im Excalibur-DataValue hinterlegt. Gegebenenfalls wird danach das nächste, im DataTransformer vorhandene, Skript ausgeführt.

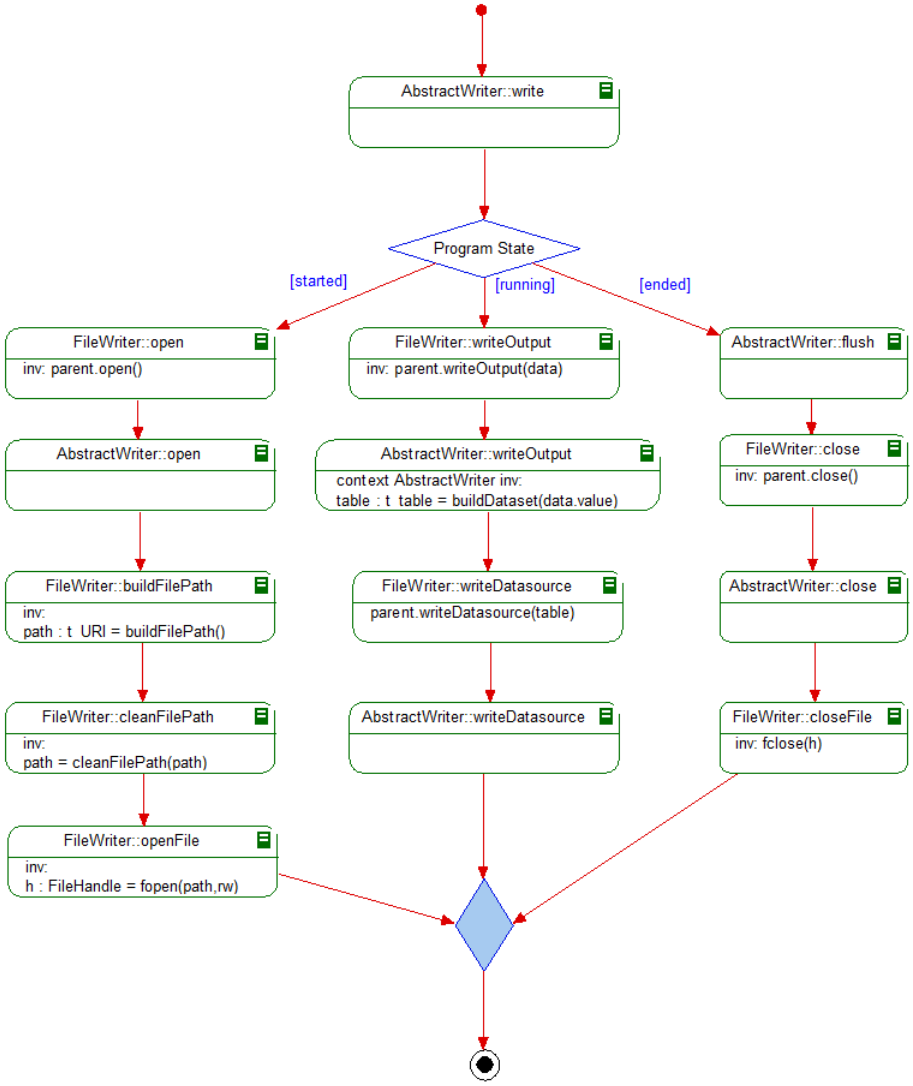


Abbildung (3.41)
Aktivitätsdiagramm zur Darstellung des Schreibprozesses eines FileWriters.

Templates und die formatierte Ausgabe Der letzte, wichtige dynamische Prozess betrifft die Template-Engine, mit welcher Daten (genauer Excalibur-DataValues) in verschiedene Ausgabeformate konvertiert werden. Dafür wird ein spezieller Serialisierer, der TemplateSerializer, verwendet. Templates haben eine eigene, an HTML angelehnte Syntax. Sie kann über die Template-Klasse geladen und interpretiert werden. Für spezielle Elemente gibt es eigene Tags, d.h. in „<“ und „>“ eingeschlossene Schlüsselwörter, um bestimmte, vordefinierte Eigenschaften zu setzen (z.B. Benutzer, Datum, Methoden-

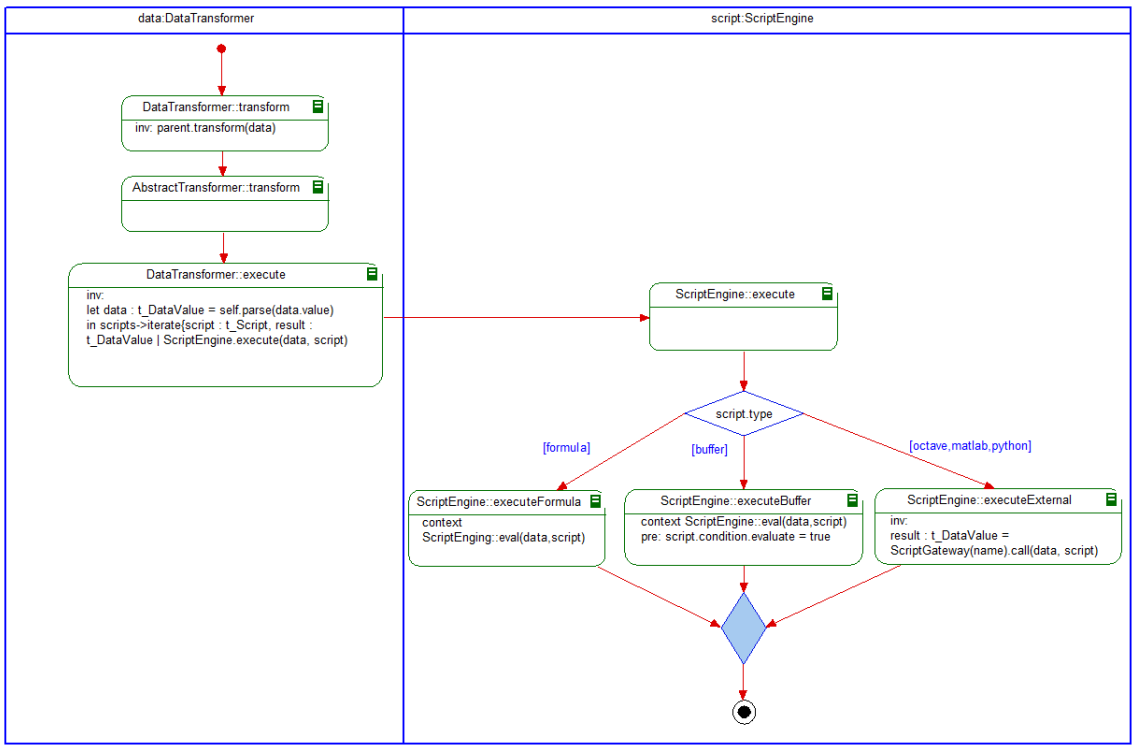


Abbildung (3.42)

Aktivitätsdiagramm zur Darstellung des Transformationsprozesses eines DataTransformers.

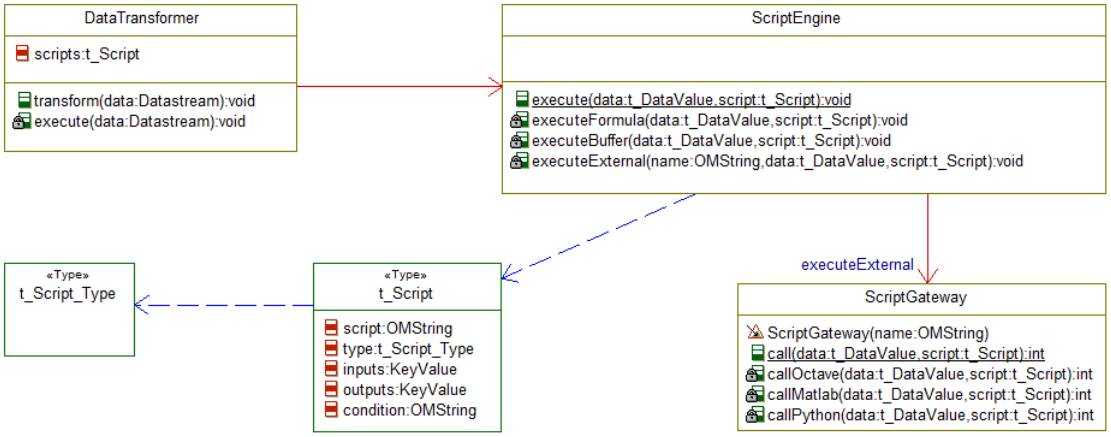


Abbildung (3.43)

Klassendiagramm aller zugehörigen Modelle, die für das Skripting benötigt werden.

namen, usw.). Eine vollständige Liste hierzu kann in der Dokumentation nachgelesen werden.
Soll ein Datensatz mit Hilfe von Templates serialisiert werden, so wird der Datensatz auf das Template „angewendet“. Das bedeutet, im Datensatz muss vermerkt sein, wohin die Daten im Template geschrieben werden sollen. Es muss aber nicht vermerkt werden, wie die Daten geschrieben (formatiert, konfiguriert, usw.) werden sollen. Dafür gibt es die Tabellendefinition, wobei die einzelnen Spalten Datensätzen von Geräten zugeordnet werden können. Die Identifikation der zugehörigen Spalte wird über die Kanäle vorgenommen. Für jede „Zeile“ der Tabelle gilt, dass diese *vollständig* sein muss. Es soll eine Beispieltabelle mit drei Spalten betrachtet werden, von welchen die erste die Messzeit (des ersten eingetroffenen Wertes dieser Zeile), die zweite ein Spannungswert (eines angenommenen Messgerätes) und die dritte ein Stromwert (desselben Messgerätes) darstellen soll. Ist der erste (durch den Kanal identifizierte Wert) eine Spannung, so wird der nächste zu schreibende Spannungswert erst geschrieben, nachdem auch ein Stromwert eingetroffen ist. Bis dies der Fall ist, wird der Spannungswert zwischengespeichert. Kommt er niemals an, so werden beim *flush*-Aufruf nach Beendigung der Ablaufsequenz alle zwischengespeicherten Werte dennoch geschrieben, während fehlende mit der jeweils für den zu schreibenden Datentyp üblichen Notation abgelegt wird (z.B. NaN für Zahlen, ” für Zeichenketten, usw.). Deshalb ist bei Tabellen zu beachten, dass nicht korrelierte Daten nicht in dieselbe Tabelle geschrieben werden dürfen; Datensätze müssen zeilenweise *vollständig* sein.
Außerhalb von Tabellen kann beliebiger Fließtext mit beliebig vielen oben genannten Tags angegeben werden. Je nach Datensenke, wird dieser unterschiedlich behandelt. Für Textdateien wird er so geschrieben, wie angegeben. Im Fall von TDMS-Dateien werden alle Tags als Metadaten gesetzt (doppelte Tags werden immer überschrieben). Bei XML-Dateien werden die Tags ausgefüllt übernommen (es handelt sich bei XML auch um Tags). Bei Datenbanken letztendlich, muss eine Tabelle für die Metadaten existieren und jeweils eine für jede angegebene Tabelle. Unter den Metadaten sollen hier alle Informationen verstanden werden, die Informationen zu den Gerätedaten geben, aber nicht die Gerätedaten selbst (z.B. Benutzername, Datum, Uhrzeit, usw.). Innerhalb der Metadaten-Tabelle muss die Spaltenbezeichnung (und der Datentyp) der Tag-ID entsprechen. Weitere Informationen bzgl. Templates, Notationen, Tags und Anwendung auf Messdaten können in der Dokumentation nachgelesen werden.
Der Vorgang der Tag-Suche ist für den Fließtext, Tabellenköpfe, Spaltenköpfe und den Spalteninhalt identisch. Wird die Methode *execute* des TemplateSerializers aufgerufen,

durchläuft dieser der Reihe nach den Fließtext (einmal zu Beginn einer Ablaufsequenz), die Tabellen und zuletzt alle Spalten in den Tabellen. Dabei wird nach Tags gesucht, mit deren Namen ein Wert verbunden ist, z.B. mit „date“ ein Datum, mit „time“ eine Zeit, mit „data“ ein Datensatz, mit „user“ der aktuelle Benutzer, usw. Die zugehörigen Werte werden dann vom Messserver abgefragt, entweder aus dem Cache, dem passierenden Kanal des Datenstroms oder direkt übergebenen Parametern. Die Tags werden anschließend in eine Struktur gekapselt und gespeichert, während der ursprüngliche Tag im Template-Text durch einen Platzhalter mit Identifikation ersetzt wird (s. Abbildung 3.44).
Ein zu Templates kompatibler Writer kann diese gekapselten Strukturen im Anschluss interpretieren. Je nach Datensenke, werden die enthaltenen Daten unterschiedlich extrahiert und formatiert. Exemplarisch am FileWriter erklärt, ergeben sich mehrere Möglichkeiten, Dateien zu schreiben. Dafür existieren die Dateitypen *ASCII*, *TDMS* und *XML*. Jeder interpretiert und schreibt die Daten auf eine andere Weise, aufgrund der unterschiedlichen Kodierungen und Notationen der Dateiformate (s. Abbildung 3.45). Rein prinzipiell läuft der Prozess des Schreibens aber ähnlich ab. Zunächst wird der Lauftext außerhalb von Tabellen geschrieben, indem zunächste alle Platzhalter wieder durch die Tags ersetzt werden. Jene müssen zunächst nach der für den zugrunde liegenden Dateityp formatiert werden. Als nächstes erfolgt dasselbe für die Tabellenköpfe, welche in ASCII-Dateien direkt, in TDMS-Dateien als Metadaten in die Tabellenköpfe und in XML-Dateien als XMLNode (hier nur die Tags selbst) geschrieben werden. Zuletzt erfolgt ein äquivalenter Prozess für die Spaltenköpfe und die Spaltendaten.

Abschließende Bemerkungen

Die dynamische Struktur der wichtigsten Abläufe innerhalb des Excalibur Server ist beschrieben worden. Auf diese Weise können die Hauptprozesse verstanden werden, welche den Hauptteil der Laufzeit ausmachen. Erreicht keine Anfrage den Messserver, läuft er im Idle-Modus und wartet nur auf Anfragen. Im nächsten Kapitel soll nun auf die konkrete Implementierung und die dafür benötigten Anforderungen beschrieben werden.

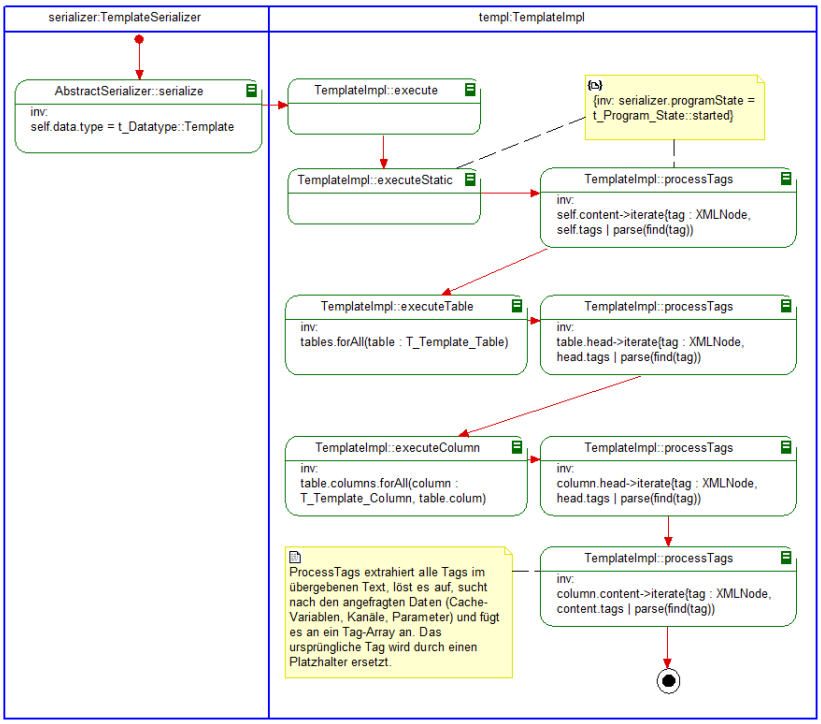


Abbildung (3.44)

Aktivitätsdiagramm zur Darstellung des Templating-Mechanismus eines TemplateSerializers.

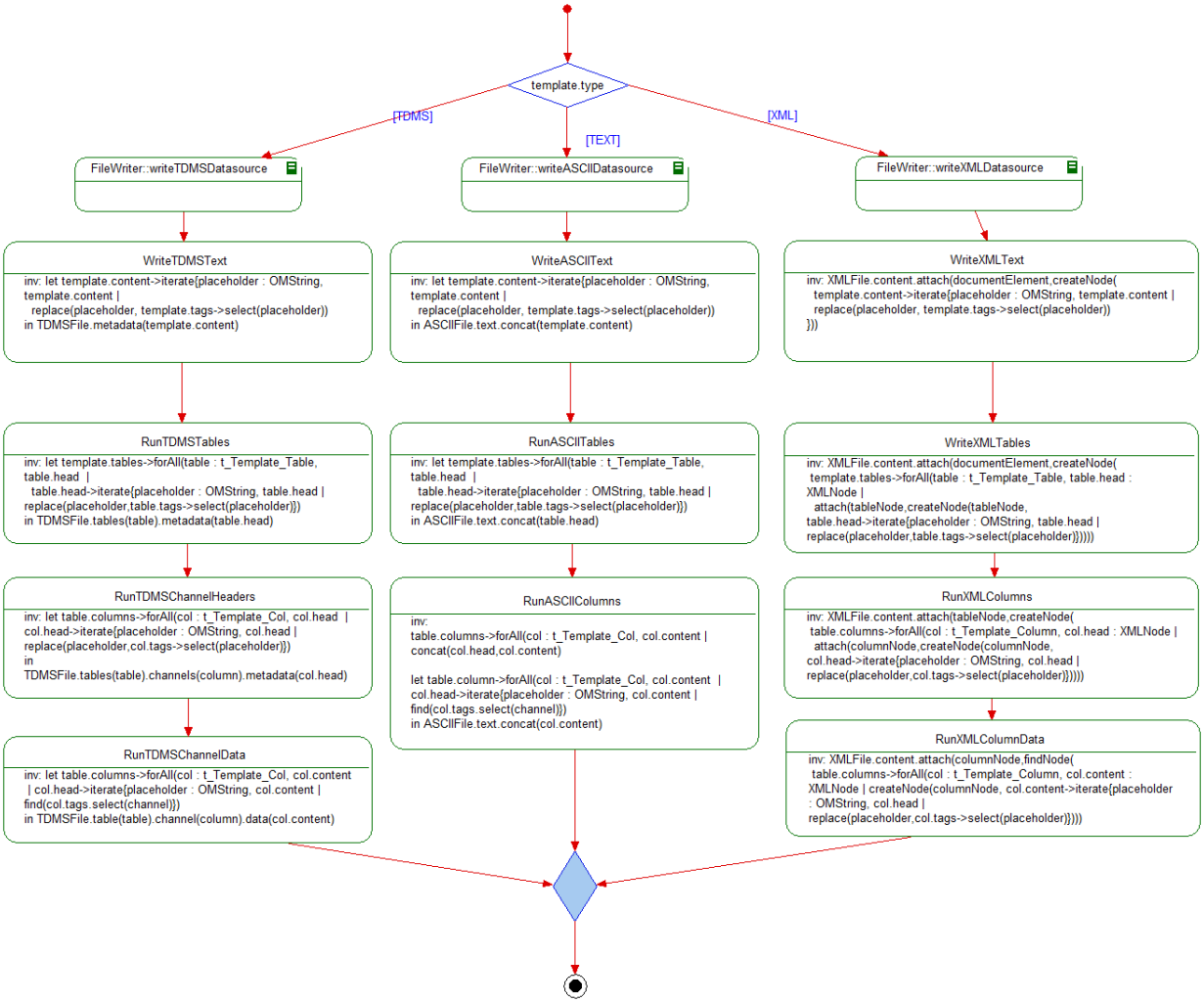


Abbildung (3.45)

Aktivitätsdiagramm zur Darstellung Schreibprozesses eines Templates mit einem kompatiblen Writer.

3.5. Implementierung des Excalibur Server

Dieses Kapitel beschäftigt sich mit der Implementierung des Excalibur Server. Anhand des im letzten Kapitel dargelegten Designs können hierfür konkrete Module entwickelt werden. Die dafür gewählte Programmiersprache muss für diesen Zweck einige Anforderungen erfüllen. Darunter fallen:

- Eine einfache Möglichkeit, über eine Vielzahl an Computerschnittstellen mit Geräten zu kommunizieren.
- Netzwerkfähigkeit - Diverse Netwerkprotokolle müssen unterstützt werden.
- Gängige Bibliotheken (extern oder nativ) müssen unterstützt werden (Webservices, Mathematische Berechnungen, Bildbearbeitung, Protokolle wie FTP oder WebDAV, usw.).
- Objektorientierte Programmierweise muss erlaubt sein, da sonst ein anderes Software-Design entwickelt werden muss.
- Threading und Multiprozessorunterstützung muss gegeben sein.
- Betriebssystemunabhängigkeit, damit die Steuerungssoftware auf möglichst vielen Endgeräten eingesetzt werden kann.
- Parser für typische Auszeichnungssprachen müssen enthalten sein (XML, JSON).

3.5.1. LabVIEW® - Möglichkeiten und Grenzen

Die Programmiersprache LabVIEW® erfüllt alle genannten Anforderungen. Eine Einführung und Beschreibung dieser Programmiersprachen mit -umgebung ist bereits in Abschnitt 3.2.1 erfolgt. Aus diesem Grund ist die Wahl für die Implementierung auf LabVIEW® gefallen.
Dennoch sind dieser Programmiersprache Grenzen gesetzt, wovon einige vor allem umfangreiche Entwicklungen verkomplizieren können. Einer der Vorteile von LabVIEW®-Programmen ist zwar die Tatsache, dass syntaktisch korrekter Code immer „direkt“ ausgeführt werden kann, weil der Code zu jeder Zeit in kompilierter Form vorliegt und deshalb ohne Zeitverlust getestet werden kann. Das ist aber auch gleichzeitig der Haupt-Nachteil, wenn es sich um sehr umfangreiche Projekte handelt. Falls Änderungen im Code vorgenommen werden müssen, die viele VIs umfassen, dauern diese Änderungen

sehr lange, weil alle geänderten VIs neu kompiliert werden müssen. Je umfangreicher das Projekt ist und je mehr VIs durch eine Änderung betroffen sind, desto länger ist die Wartezeit, bis weiter gearbeitet werden kann (mitunter sind mehrere Minuten keine Seltenheit).
Weitere Probleme ergeben sich durch die begrenzte Testfähigkeit von LabVIEW®-Code. Zwar existiert ein Unit-Testframework, welches sich aber für das vorliegende umfangreiche Projekt als nicht einsetzbar herausgestellt hat. Die Einführung eines Testfalls muss über die LabVIEW®-Projektmanager durchgeführt werden, wofür beim letzten Versuch 15 min pro Testfall benötigt worden sind. Bei mehreren hundert Testfällen ist dieser Zeitverlust nicht hinnehmbar. Aus diesem Grund erfolgt das Testen mit einem selbst erstellten Testframework, welches dafür aufwendig implementiert werden musste.
LabVIEW® Code ist betriebssystemunabhängig, solange für das Zielsystem eine Laufzeitumgebung existiert. Diese werden von der Firma National Instruments für Windows®, Linux und OS X® basierte Betriebssysteme zur Verfügung gestellt. Der LabVIEW®-Code kann über einen Build-Prozess (National Instruments Application Builder) in eine ausführbare Datei (Applikation) übersetzt werden, für welche anschließend keine LabVIEW®-Entwicklungsumgebung mehr benötigt wird. Allen Laufzeitumgebungen liegt zugrunde, dass sie auf X86- oder AMD64-basierten Prozessorarchitekturen angewiesen sind. Aus diesen Gründen können LabVIEW®-Applikationen (die kompilierten Code-Dateien) nicht auf mobilen Endgeräten wie Android® oder iOS®, ebenso wenig wie auf allen ARM-basierten Prozessorarchitekturen ausgeführt werden.
Der Excalibur Server ist in LabVIEW® entwickelt worden und liegt sowohl als kompilierte Version für Windows® als auch als unkompilierte Version in Form von VIs innerhalb eine LV-Projekts auf der beigelegten DVD vor.

3.5.2. VISA - Der standardisierte Zugriff auf Hardware

Virtual Intrument Software Architecture (VISA) ist ein Industriestandard und stellt ein Application Programming Interface (API) für die Kommunikation mit Messgeräten dar. Es wird teilweise als Kommunikationstreiber bezeichnet und erlaubt die Entwicklung von Programmen, die BUS-unabhängig sind. Die Verwendung von VISA Bibliotheken erlaubt die Kommunikation für viele Computerschnittstellen wie GPIB, USB oder Ethernet. LabVIEW® stellt ein Implementierung dieser Bibliotheken zur Verfügung, um die Gerätekommunikation zu erleichtern. Für einige Geräteschnittstellen innerhalb von Excalibur findet sie aus diesem Grund Verwendung.

3.5.3. Excalibur Skripte - Mathematische und Logische Operationen

Teilaufgabe eines Mess- und Steuersystems ist die Aufnahme und Bewertung von Daten. Ziel von Excalibur ist es gewesen, die einzelen Teilaspekte (wie im ersten Kapitel erklärt) voneinander unabhängig zu machen. Unter anderem sollten Ablaufsequenzen keine mathematischen Operationen durchführen, welche auf die von den Geräten stammenden Daten Einfluss nehmen. Dies ist wichtig, da sich Gerätedaten in ihren Datentypen voneinander unterscheiden und somit beim Austausch eines Gerätes durch ein kategorisch gleichwertiges, die Ablaufsequenzen möglicherweise unbrauchbar werden. Diese mathematischen Operationen sollen in die Excalibur-Treiber oder die Ausgabe (Transformer) verlagert werden, wenn es möglich ist.
Innerhalb von Excalibur sind verschiedene Methoden für mathematische Operationen verwendbar. Wie bereits in Abschnitt 3.3.5 besprochen gibt es dafür verschiedene Skripting-Anweisungen, welche je nach Einsatzzweck eingesetzt werden können. LabVIEW® selbst unterstützt zwar einen sehr großen Funktionssatz an mathematischen Methoden, aber es stellt diesen in Form von VIs zur Verfügung. Das bedeutet, dass zur Änderung von mathematischen Modellen immer eine Entwicklungsumgebung vorhanden sein muss. Excalibur als Software soll aber ohne eine Entwicklungsumgebung eingesetzt werden können, weshalb der Einsatz von VIs nicht ohne Umwege möglich ist. Ein Möglichkeit die mathematischen VIs in einer kompilierten Applikation zu verwenden wäre es, alle (!) mathematischen VIs bzw. deren Bibliotheksaufruf (unter Windows® den DLL-Aufruf) in eine eigene Bibliothek zu kapseln und dafür eine Domain Specific Language (DSL) zu entwickeln. Diese Bibliothek ließe sich dann kompilieren, über die selbst entwickelte Grammatik ansprechen und damit in einer Applikation einsetzen. Außerdem scheint es nicht sinnvoll, LabVIEW®-Bibliotheken in eigene Bibliotheken zu übersetzen und damit LabVIEW®-Funktionalität zu „kopieren“.
Ein anderer Weg sieht vor, externe Mathematikprogramme zu verwenden, die bereits eine eigene DSL mitbringen. Beispiele hierfür sind GNU Octave® oder das kommerziell erhältliche Matlab®. Diese sind bereits weit verbreitet; außerdem arbeiten sie sehr performant.

GNU Octave® Mathematikumgebung

Wie zuletzt erwähnt bietet sich für komplexe Berechnungen eine mathematische Umgebung wie GNU Octave® an. Es ist eine interpretierte Hochsprache, die für numerische

Berechnungen verwendet wird. Damit lassen sich von linearen/nicht-linearen Problemen bis hin zu komplexen Matrizenverarbeitungen viele mathematische Berechnungen durchführen. Die Sprache ist ähnlich zur kommerziell verfügbaren Mathematiksoftware Matlab®, weshalb sich dort entwickelte Skripte einfach portieren lassen.
In LabVIEW® wird GNU Octave® nicht direkt unterstützt, weshalb dafür ein Gateway verwendet werden muss. Dafür wird die GOLPI-Schnittstelle (Gnu Octave to Labview Pipes Interface)[34] verwendet, welches als API nachinstalliert werden kann. Diese Schnittstelle erlaubt es nun, über die installierten VIs auf einen GNU Octave®-Prozess zuzugreifen, Variablen zu übermitteln und Berechnungen über Skripte durchzuführen. GNU Octave® muss dafür vorher auf dem verwendeten Betriebssystem installiert werden.

Python

Eine weitere Möglichkeit, mathematische Operationen durchzuführen bietet die Programmiersprache Python. Hierbei handelt es sich im Gegensatz zu GNU Octave® oder Matlab® um eine vollwertige Programmiersprache. Ähnlich zwischen allen ist, dass es sich um eine interpretierte Hochsprache handelt, auf die über ein Gateway zugegriffen werden kann. Dafür wird die *OpenG Python Library* verwendet, die über die mitgelieferten VIs einen Python-Prozess öffnet und wie auch im Fall von GNU Octave® dazu in der Lage ist, Variablen zu übermitteln und Berechnungen in Form von Python-Skripten durchzuführen. Diese Bibliothek kann über das National Instruments-Tools-Framework mit Hilfe des VI Package Manager®, der bei der Installation von LabVIEW® mitgeliefert wird, nachinstalliert werden. Auch hier muss vorher Python (Version 2.x) auf dem verwendeten Betriebssystem installiert werden.

3.5.4. Bildbearbeitung in Excalibur

Geräte wie Kameras oder andere bildgebende Systeme, liefern Bilder als Messdaten zurück. Die Auswertung von Bilddaten ist äußerst komplex und rechenintensiv, weil meist schon sehr große Datenmengen bereits in einem einzigen Bild vorhanden sein können. Wie auch bei den mathematischen Skripten bietet LabVIEW® eine Fülle an Bildbearbeitungs-VIs an. Leider tritt hier ein ähnliches Problem auf, wie bereits oben angemerkt. Die Bildbearbeitungs-VIs müssten in einer eigenen Bibliothek gekapselt werden

[34] Siehe unter http://kaero.wz.cz/golpi.html

und mit Hilfe einer DSL eingebunden werden. Auch mögliche Lösungswege sind dahingehend denen der Mathematikumgebungen ähnlich. Ein weiteres Problem von LabVIEW® ist hier, das die erweiterten Bildbearbeitungsfunktionalitäten in einer separaten Laufzeitumgebung enthalten sind. Diese muss für jedes Zielsystem separat erworben werden und kostet hohe Lizenzgebühren.

Der ImageTransformer - Zugriff auf den ImageMagick®-Prozess

Mit der Software ImageMagick® existiert ein auf einer Konsolenanwendung basiertes Bildbearbeitungssystem. Es ist für alle gängigen Betriebssysteme erhältlich und verfügt über eine Vielzahl an Bibliotheken für gängige Programmiersprachen. LabVIEW® unterstützt das Einbinden von Bibliotheken in allen erhältlichen Laufzeitumgebungen (DLL-Bibliotheken unter Windows, SO-Bibliotheken unter Unix-basierten Systemen). Auf diese Weise ist es möglich, die Bibliotheksaufrufe in einer eigenen Klasse zu kapseln. Auf diese Weise kann der komplette Funktionsbestand von ImageMagick® abgebildet und in Excalibur verwendet werden. Von dieser Funktionalität macht bislang ausschließlich der ImageTransformer Verwendung, um von Geräten produzierte Bilddaten zu konvertieren oder Bildbereiche auszuschneiden.

3.6. Ergebnisse und abschließende Betrachtungen

Es ist gelungen ein Software-Konzept für die Laborautomatisierung zu entwickeln, um die in der Einführung genannten Probleme zu lösen. Anhand der Analyse dieses Konzeptes, können die Rahmenbedingungen für ein mögliches Software-Design abgeschätzt werden.
Ein wichtiger Punkt dafür ist die Modularität, denn sie erlaubt es, Aspekte für eine softwarebasierte Steuerung unabhängig voneinander zu verändern, ohne den Gesamtprozess anpassen zu müssen. Dazu sind die Aspekte Gerätekommunikation, Ablaufsteuerung und die Datenein- bzw. -ausgabe voneinander getrennt worden und die Interaktion zwischen ihnen verläuft über Mappings, d.h. einer eindeutigen Zuordnung über Namen mit klar definierten Schnittstellen.

3.6.1. Ergebnis

Treiber

Mit Hilfe der Treiberdefinition ist es gelungen, typische Laborgeräte und deren Kommandos eindeutig zu beschreiben. Da es sich bei den Treibern um Textdokumente mit UNICODE-Kodierung handelt, lassen sie sich auf sehr einfache Weise erstellen, modifizieren und ersetzen, wodurch gewährleistet wird, dass sich Geräte ohne Umstände durch andere Geräte in Excalibur ersetzen lassen, ohne die gesamte Software neu übersetzen zu müssen. Dafür wird eine Notation basierend auf XML verwendet, welches als standardisierte Auszeichnungssprache eine gute Lesbarkeit, Bearbeitbarkeit und langfristige Kompatibilität gewährleistet.
Darüber hinaus ist es mit Hilfe des Alias Mechanismus möglich, verschiedene Namen für Kommandos zu definieren, welche in Ablaufsequenzen eingesetzt werden können, diese kompatibel zu verschiedenen Geräten zu machen[35] und somit den Treiber unabhängig von der Definition der Ablaufsequenzen zu halten.

Schnittstellendefinition

In diesem Kapitel ist Spezifikation der Hardwareschnittstelle beschrieben worden, über welche Geräte an Computersysteme angeschlossen sind. Mit ihrer Hilfe ist es möglich, die Konfiguration der Schnittstelle vollständig zu beschreiben, um über sie mit den Geräten kommunizieren zu können. Zusätzlich ist diese Spezifikation der Verknüpfungspunkt zum Gerätetreiber, welche über die Gerätereferenz gegeben ist. Auf diese Weise ist es möglich, nur über diese Bezeichnung eine Verbindung beider herzustellen und sowohl

[35] Ein besserer Mechanismus wird über die Einführung von Kategorien in Abschnitt 3.3.8 besprochen

Treiber oder Schnittstellendefinition auszutauschen, solange die Referenzbezeichnung beibehalten wird. Dadurch wird ein hohes Maß an Flexibilität erreicht, denn bei Änderung der Schnittstelle eines Gerätes (beispielsweise für Geräte mit sowohl RS232- und USB-Schnittstelle), müssen keinerlei Änderungen an jedweden Deskriptoren durchgeführt werden außer an diesen Schnittstellendefinitionen, wenn auf die andere Computerschnittstelle gewechselt wird[36].

EVA Prinzip mit Readern und Writern

Es ist der vollständige Excalibur Request-Response-Prozess besprochen worden, angefangen bei einer Benutzeranfrage, der Umwandlung ebendieser in eine interne Datenstruktur, die Prozessierung und letztendlich die Datenausgabe. Es gibt verschiedene Möglichkeiten, die Eingabe und das Transport-Protokoll zu konfigurieren. Bisher nicht implementierte Transport-Protokolle und Datenaustauschformate bzw. Auszeichnungssprachen sind ohne Probleme integrierbar, da sie von einer gemeinsamen Datenstruktur ableiten. Die Prozessierung der Anfrage wird über Kommandos realisiert, welche auch ohne Schwierigkeiten erweiterbar sind, da es sich um ein sehr dynamisches *Verhaltens-Pattern* (s. Abschnitt 3.4.4) handelt. Zuletzt erlaubt die sehr dynamisch konfigurierbare Ausgabe mit Hilfe der Kanäle, Datenflüsse geregelt über verschiedene Austauschformate/Auszeichnungssprachen und Transport-Mechanismen an unterschiedliche Endpunkte zu leiten. An Schlüsselpositionen kann mit Hilfe von Skripten Einfluss auf die Daten genommen werden, je nachdem ob die Skripting-Mechanismen für die Geräte, die Methode oder den gerade zu messenden Assay konfiguriert worden sind.

Kanäle und Skripte

Die Kombination aus Kanälen und Skripten löst eines der in der Problemanalyse dargelegten Problematiken: Die Abhängigkeit zwischen Sensor und Steuerung, da Auswertungsstrategien unabhängig von den Geräten, den Sensoren und den darauf wirkenden Ablaufsequenzen verändert werden können.

Ablaufsequenzen

Mit Hilfe von Ablaufsequenzen ist es möglich, die Reihenfolge und die Frequenz von Gerätekommandos zu definieren, indem mit einem Namen und optionalen Parameterwerten auf Hardwarebefehle innerhalb des Excalibur-Treibers verwiesen wird. Dabei ist es irrelevant, ob mit der Kommando-ID oder einem seiner Aliase auf das Kommando verwiesen wird. Durch den Einsatz von Datenhaltungs-, Ablauf- und Verarbeitungsstrukturen kann Einfluss auf den Ablauf der Sequenz bzw. der generierten Daten genommen werden.

Templates

Über den Template-Prozessor ist es möglich, Daten auf sehr vielseitige Weise zu formatieren und an die entsprechenden Datensenken weiterzugeben. Dadurch kann mit relativ wenig Aufwand eine eigene und übergreifende Dateiformatierung innerhalb der Labor-

[36] Unter der Voraussetzung, dass die Gerätehersteller denselben Befehlssatz für beide Schnittstellen verwenden

umgebung eingeführt werden. Sollte diese einmal im Rahmen einer Umstrukturierung geändert werden, so ist dies auch ohne Änderung der Mess-Software möglich, einfach indem ein anderes Template verwendet wird.
Darüber hinaus handelt es sich bei den Templates um XML-Dokumente, d.h. es können beliebige XML-Editoren[37] dafür verwendet werden. Beim Einbinden der korrespondieren XSD-Datei kann der XML-Editor unterstützen, die Template-Syntax auf eine sehr einfache Weise zu konstruieren. Darüber hinaus ist es möglich, grafische Editoren auf Basis von WYSIWYG (What you see is what you get) zu implementieren, mit deren Hilfe das Schreiben von Template-Dateien vereinfacht werden kann.

Methoden

Mit Hilfe des Methodenmodells ist es möglich, analytische Methoden in Excalibur abzubilden. Die Planung der Methode kann ohne konkreten technischen Aufbau realisiert werden, da allein das Wissen über die Art der Geräte genügt. Auf diese Weise können Laboraufbauten unabhängig von den darunter liegenden Systemen gestaltet werden.

Synchrone und asynchrone Kommunikation

Die Kommunikation mit dem Excalibur Server kann sowohl über einen synchronen Request-Response Mechanismus als auch über einen asynchronen, Event-basierten Mechanismus erfolgen. Abhängig von der gewählten Ein- und Ausgabekonfiguration über die Reader und Writer, können die Kommunikationsstrategie auf das vorliegende Laborumfeld angepasst werden. Durch die standardisierten Notationen können verschiedene Clients eingesetzt und auch entwickelt werden. Durch diesen Multi-Server/Multi-Client Ansatz kann abhängig vom Umfeld die passende Infrastruktur für die Laboranlagen gewählt werden, ausgehend von einem einzelnen Computersystem, der sowohl den Excalibur Server auch auch den favorisierten Client enthält, bis hin zu einer komplexen Netzwerk-Infrastruktur mit mehreren Mess-Servern, verschiedenen administrativen Endgeräten, Daten-Servern und Verzeichnisdiensten.

Authentifizierung und Autorisierung

Authentizität, Integrität und Vertraulichkeit müssen für ein über das Netzwerk interagierendes System unbedingt gewährleistet sein. Nur so kann garantiert werden, dass die übertragenen Daten ausschließlich demjenigen zur Verfügung stehen, welcher dazu berechtigt ist. Darüber hinaus muss immer garantiert werden, dass nur berechtigte Personen und Systeme tatsächlich Aktionen auf dem Excalibur Server ausführen und damit auf Laborgeräte und -anlagen zugreifen dürfen. In Excalibur wird die Sicherheitsinfrastruktur über eine externe, etablierte Software, dem Apache Webserver, bereitgestellt, welche Anfragen an den Serverdienst über einen Proxy realisiert. Dem Client fällt dabei nicht auf, dass er nicht direkt mit dem Excalibur Server kommuniziert (sondern über eine Zwischenschicht).

[37] Beispielsweise <oXygen/> oder oder Eclipse

3.6.2. Bewertung der Ergebnisse

Durch die Lösung der in der Einführung genannten Probleme, ist es für *Systemintegratoren* einfacher, für ihre entwickelten analytischen Aufbauten, Steuerungssoftware zu entwickeln. Dafür muss nicht mit Hilfe eines Systemprogrammierers aufwendig Programmcode in gängigen Programmiersprachen entwickelt werden, sondern es wird eine Steuerung anhand eines methodenbasierten Ansatzes modelliert. Für Systemintegratoren ist diese Perspektive auf Steuerungen einfacher nachvollziehbar, weil sie die Denkweise des vorliegenden Aufbaus widerspiegelt und nicht anhand von Paradigmen aus der Informatik.
Der *Benutzer* kann selbst in den Softwareentwicklungsprozess eingreifen, da die Konfiguration über textbasierte Deskriptoren mit einer leicht verständlichen Notation vorgenommen wird. Darüber hinaus kann er Ablaufsequenzen, ohne eine Zeile Quelltext verfassen zu müssen, auf einfache Weise verändern oder erweitern.
Der *Systemprogrammierer* muss am eigentlichen Steuerungsprozess keine Arbeit investieren, sondern kann sich auf Komponenten wie die grafische Benutzerinteraktion oder neue Client-Funktionalitäten konzentrieren. Durch diesen Zeitgewinn kann dort mehr Entwicklungsaufwand investiert werden.

Albert Einstein

> Falls Gott die Welt geschaffen hat, war seine Hauptsorge sicher nicht, sie so zu machen, dass wir sie verstehen können.

4

Zusammenfassung und Ausblick

In diesem Kapitel sollen alle Ergebnisse zusammengetragen und aufgezeigt werden. Zuletzt soll ein kleiner Ausblick gegeben werden, welche Erweiterungen der Steuerungslösung zukünftig sinnvoll erscheinen. In einem kurzen Fazit sollen zum Schluss einige allgemeine Worte zur entwickelten Anwendung folgen.

4.1. Zusammenfassung

Innerhalb dieser Thesis werden die Problematiken während des Entwicklungsprozesses von Steuerungssoftware für Laborsysteme erläutert. Auf diese Problematiken wird im darauf folgenden Kapitel genauer eingegangen und es ist versucht worden, dafür mögliche Lösungsansätze zu finden. Daraus wird eine Strategie entwickelt, auf die mit Hilfe allgemeiner Software-Designprozesse eingegangen wird. Zunächst werden alle Lösungen analysiert und in Abschnitte unterteilt, welche im Anschluss beleuchtet werden.
Beachtet werden müssen, dass die Einzelaspekte *Peripherie*, *Schnittstelle*, *Datenein-/-ausgabe* und *Ablaufsteuerung* separiert und mit Hilfe von Deskriptoren „vollständig“ beschrieben werden müssen. Dafür sind $n + 2$ Deskriptoren ausreichend (n ist Anzahl an unterschiedlichen Geräten). Die Interaktionen zwischen diesen Einzelaspekten wird über Mappings realisiert, d.h. Abbildungen eines Aspekts auf den anderen über gemeinsam genutzte Schnittstellen und Bezeichnungen (Namen und Aliase in Geräten, Kommandos und Parametern). Um eine Aspekt-unabhängige Formatierung der Ausgabe zu

realisieren, werden *Templates* eingeführt, welche über *Kanäle* Daten benutzerdefiniert „schreiben“ können. Für eine *netzwerkbasierte* Strategie, ist es nötig sowohl die Gerätesteuerung von der Benutzerinteraktion zu trennen, als auch „Geräteschnittstellen“ einzuführen, welche auf Netzwerktechnologien beruhen. Eine Trennung der eigentlichen Gerätesteuerung von der Benutzereingabe kann aus diesem Grund als weiterer Aspekt des Resultats angesehen werden. Um diese Trennung zu ermöglichen, ist die Verbindung der beiden über gängige Transportmechanismen auf Basis von Netzwerkprotokollen naheliegend (HTTP, Webservices) und wird auch auf diese Weise realisiert. Diese Wahl erlaubt es, gängige Netzwerkinfrastrukturen zu verwenden, die in den Einsatzbereichen meist schon vorhanden sind. Folgt man diesem Ansatz, müssen Datensicherheit im Rahmen von Authentizität und Autorisierung gewährleistet sein, wofür aber bereits etablierte Sicherheitsstandards und -technologien existieren.

Auf dieser Basis werden im Anschluss Randbedingungen und Anforderungen der einzelnen Themen dargelegt. Aus dieser Analyse wird ein Softwaredesign entwickelt, womit die Anforderungen mit Hilfe informatischer Modelle abgebildet werden kann. Zunächst wird diesbezüglich die Gesamtarchitektur des Excalibur-Kontexts eingeführt und beschrieben. Für den konkreten Fall des Excalibur Servers werden dann sowohl wichtige statische Modelle über Klassendiagramme erläutert, als auch die Interaktionen zwischen diesen über Sequenz- und Aktivitätsdiagramme.

Mit deren Hilfe sind letztendlich die Auswahl einer geeigneten Programmiersprache und eines -paradigma erfolgt, inklusive aller verwendeten externen Bibliotheken. Letztere sind notwendig, weil nicht alle benötigten Einzelteile des Gesamtsystems selbst programmiert können.

4.1.1. Betrachtung des Gesamtprozesses

Betrachtet man sich die genannten Anwendungsbeispiele von Abschnitt 2.3 zeigt sich, dass mit Hilfe des Excalibur Servers innerhalb kürzester Zeit ohne Programmieraufwand Steuerungssoftware entworfen werden kann.[1] Die Anwendung ist sehr dynamisch und

[1] Die INSTANT Software ist innerhalb von 3 Wochen inklusive entwickelt worden, mit allen Plugins, Ablaufsequenzen und fehlenden Treibern

kann effizient auf sich ändernde Bedingungen angepasst werden. Änderungen im analytischen Aufbau können sehr einfach in der Steuerungssoftware abgebildet werden und eine Übertragbarkeit der geänderten Bedingungen auf die zugrunde liegende (oder andere) Methode ist ohne Probleme möglich. Auch sehr komplexe Ablaufsequenzen mit vielen Geräte (bzw. funktionalen Einheiten) können realisiert werden. Darüber hinaus ist jede Anwendung immer netzwerkfähig und die Steuerung kann von entfernten Computersystemen erfolgen wie auch die Kommunikation mit netzwerkbasierten Geräten erfolgen kann. Grafisch ansprechende Oberflächen können auf diese Weise unabhängig vom zugrunde liegenden analytischen Aufbau realisiert werden.

Sicht des Systemprogrammierers

Systemprogrammierer haben bislang die Hauptaufgabe während des Softwareentwicklungsprozesses gehabt, Steuerungssoftware für analytische Aufbauten zu implementieren. In dem hier vorgestellten Ansatz verschiebt sich ihr Einsatzgebiet von der Implementierung der Steuerung (des Servers) zur Implementierung der Benutzerinteraktion (des Clients).

Sicht des Systemintegrators

Der Systemintegrator ist für die Planung und die Entwicklung des analytischen Aufbaus verantwortlich. Ist kein Systemprogrammierer vorhanden, fällt diesem ebenso die Aufgabe der Implementierung der Steuerungssoftware zu. Da dies nicht seiner Fachkompetenz entspricht, ist die Wahrscheinlichkeit sehr hoch, dass daraus fehlerhafte oder ungenügend getestete Software entsteht. Die Alternative ist es, den Softwareentwicklungsprozess zu hohen Kosten an externe Anbieter abzugeben. Mit dem hier vorgestellten Konzept kann der Systemintegrator fehlerfreie Steuerungssoftware selbst generieren, weil der Ansatz der Excalibur-Software auf einem methodenbasierten Ansatz beruht, welcher der Denkweise bei der Entwicklung eines analytischen Aufbaus entspricht. Die internen Prozesse der Software, welche eine tiefes informatisches Fachwissen benötigen, sind ihm bereits abgenommen worden.

Sicht des Benutzers

Der Benutzer profitiert vom Komfort von gleichbleibenden grafischen Oberflächen, da durch den Einsatz des hier vorgestellten Ansatzes, wiederverwendbare Komponenten für die Benutzerinteraktion geschaffen werden können. Standardprozeduren wie Datenspeicherung oder Prozesskontrolle werden dem Benutzer durch die internen Excalibur-Prozesse immer auf die gleiche Art und Weise zur Verfügung gestellt. Darüber hinaus kann der Benutzer alle Geräte von seinem Arbeitsplatz (oder zu Hause) administrieren, da die Steuerungssoftware immer über das Netzwerk agiert.

4.1.2. Durch Lösungsansatz gelöste Probleme

Folgende Probleme sind mit der vorliegenden Implementierung gelöst worden:

- Eine Trennung der Aspekte Gerätekommunikation, Ablaufsteuerung, Datenein-/-ausgabe und der grafischen Oberfläche ist erfolgt.
- Das eingeführte Treiberkonzept erlaubt es, Geräte eines analytischen Aufbaus ohne Änderungen der Ablaufsequenz oder gar Neukompilierung der Steuerungssoftware durch gleichwertige zu ersetzen, wodurch ein hohes Maß an Flexibilität gegeben ist.
- Mit Hilfe der Test-Schnittstelle ist es möglich, die Entwicklung der Steuerungssoftware vorzunehmen, ohne dass die Hardware-Integration bereits erfolgt ist.
- Das zugrunde liegende System kann mit geringfügigen Änderungen von einem System auf ein anderes übertragen werden (anderes Betriebssystem, andere Geräte, andere Datenein-/ausgabe).
- Bereits existierende Module müssen nicht neu entwickelt, sondern über die verschiedenen Deskriptoren und Mappings nur neu verknüpft werden.
- Die Steuerungssoftware muss nicht programmiert, sondern ausschließlich über XML-Deskriptoren konfiguriert werden, wodurch sehr viel Entwicklungszeit eingespart werden kann.
- Die Steuerungssoftware basiert auf einem Multi-Server/Multi-Client Ansatz und ist immer rein netzwerkbasierend, d.h. die Geräteinteraktion ist immer getrennt von der Benutzerinteraktion.

- Datenspeicherung wird über Deskriptoren konfiguriert, ist sehr flexibel über Templates formatierbar und kann auf gängige Medien erfolgen (Datei, Datenbank, WebDAV Verzeichnis, FTP).

4.1.3. Bestehende Probleme

Es sind nicht für alle Deskriptoren grafische Oberflächen vorhanden, weshalb manche Deskriptoren manuell konfiguriert werden müssen. Dafür muss nach wie vor ein Verständnis für die Prozesse und die einzelnen Parameter vorhanden sein. Zusätzlich sind Regelprozesse nach wie vor nicht berücksichtigt, wodurch Ausgangsdaten von Geräten keinen Einfluss auf interne Prozesse nehmen können. Gerade dieses wird auch die zukünftige Hauptaufgabe sein.

4.2. Ausblick

Für eine umfassendere Verwendbarkeit sollen geeignete Frameworks für weitere Programmiersprachen (Python und Java Frameworks existieren bereits) realisiert werden. Weitere Erweiterungen und Vereinfachungen seitens der Ablaufsteuerungen sollen realisiert werden. Darunter fallen eine benutzerfreundliche Konfiguration von Threads, Zeitablaufsteuerung (Scheduling), Regelketten und Zustandsautomaten. Zuletzt soll eine Verfügbarkeit des Excalibur Server auf miniaturisierte Systeme (Barebones) angestrebt werden.

4.3. Fazit

Während bisherige Softwarelösungen für Gerätesteuerung in gängigen Programmiersprachen von Systemprogrammierern implementiert worden sind, erfolgt beim Einsatz dieser Architektur eine Verschiebung des Entwicklungsprozesses auf eine andere Zielgruppe. Die Konfiguration muss dabei nicht mehr zwangsläufig vom Systemprogrammierer durchgeführt werden, sondern kann auf einfache Weise vom Systemintegrator oder gar einem Benutzer abgenommen werden. Sollen keine aufwendigen grafischen Oberflächen generiert werden, so ist diese Aussage ohne weiteres möglich.
Um netzwerkbasierte Ablaufsteuerungen zu entwickeln, soll ausschließlich die Installation der Excalibur-Software auf dem Zielsystem nötig sein. Der Rest kann prinzipiell über

eine grafische Weboberfläche konfiguriert werden.[2] Auf diese Weise muss zwar (natürlich) das Verständnis für den zugrunde liegenden analytischen Aufbau vorhanden sein, um eine Steuerungssoftware zu entwickeln. Es werden aber keine tief gehenden Programmierkenntnisse oder fundiertes informatisches Fachwissen benötigt.

[2]Das ist mit dem Django Framework möglich

Glossar

Ablaufsteuerung

Als *Ablaufsteuerung* wird das logische Modell verstanden, um die Aspekte eines Laboraufbaus in einen sinnvollen Zusammenhang zu bringen. Dazu zählen neben dem eigentlichen Steuern der Geräte auch die Interaktionen zwischen den Geräten, die Datenauswertung bzw. deren Speicherung und auch die grafische Repräsentation für Darstellung und Überwachung der Geräte.

Antwortklasse

Im Kontext des Excalibur Server stellt die Antwortklasse die Zuordnung einer Geräteantwort zu einer Kategorie dar, welche unterschiedlich behandelt werden. Es gibt die Klassen Status, Fehler, Daten und Konfiguration.

Auszeichnungssprache

Eine Auszeichnungssprache (engl.: markup language) ist eine formale (nicht-natürliche) Sprache, die es ermöglicht, unterschiedliche Bestandteile eines Textes als solche zu kennzeichnen. Durch eine Auszeichnungssprache können beliebigen Textelementen auf deklarative Weise Eigenschaften zugewiesen werden, wodurch deren Bedeutung ausgedrückt werden kann[EDW14].

Automationsablauf

Bei Automationsablauf oder eine Autmatisierungssequenz handelt es sich um eine definierbare Abfolge von Geräteebefehlen, welche von einem Benutzer definiert werden kann. Er legt fest, in welcher Reihenfolge einzelne Geräte angesprochen werden.

Automatisierung

Anwendung von technischen Mitteln, mit deren Hilfe ohne Einflussnahme des Menschen Arbeitsmittel teilweise oder ganz nach vorgegebenen Programmen bestimmte Operationen durchführen. Allgemeine Automatisierungsziele sind: Erhöhung der Produktivität, Verbesserung der Produktqualität, höhere Zuverlässigkeit und Sicherheit, wirtschaftlicherer Rohstoff- und Energieeinsatz, Schonung der Umwelt, Humanisierung der Arbeit sowie insbesondere die Ermöglichung von Prozessen, die bei manueller Prozessführung infolge der dem Menschen innewohnenden Un-

zulänglichkeiten nicht durchführbar sind[Bro87].

Base64

Hierbei handelt es sich um ein Kodierungsverfahren für 8-bit Werte, um beliebige Daten in eine Folgen von Zeichen umzuwandeln, die in jedem Schriftsatz enthalten sind. Damit werden nach wie vor beispielsweise Email-Anhänge bei der Übertragung kodiert.

Baudrate

Die Baudrate oder Schrittgeschwindigkeit gibt die Zahl der gesendeten Zeichen bzw. Kennzustände pro Sekunde an. Die Einheit ist $1/s$. Erfolgt die Datenübertragung binär, d.h. mit zwei Kennzuständen, so lässt sich daraus die Übertragungsgeschwindigkeit nach $v = v_B \times \log^2 n = v_B$ (für n = 2) mit v = Übertragungsgeschwindigkeit, n = Anzahl Kennzustände, v_B = Schrittgeschwindigkeit.

BCD

Bei diesem Kodierungsverfahren (Binary-Coded Decimals) werden die einzelnen Stellen einer Dezimalzahl mit einzelnen Halbbytes kodiert, d.h. die Ziffer jeder Dezimalposition wird jeweils einzeln kodiert. Sie wird vor allem dort verwendet, wenn höchste Präzision bei Gleitkommazahlen benötigt wird und die 32-bit Gleitkommadarstellung nicht ausreicht.

Bibliothek

Bibliotheken in Form von DLL- (Windows) oder SO-Dateien (Linux) stellen Funktionalitäten zur Verfügung, wie es auch ausführbare Dateien tun würden. Der Vorteil liegt darin, dass sie nur einmal in den Speicher geladen werden müssen und verschiedene Anwendungen können auf sie zugreifen und deren Funktionalitäten verwenden. Dadurch kann man den Speicherplatz sowohl auf der Festplatte als auch im Hauptspeicher reduzieren. Bibliotheken können dabei sowohl Programmcode, Daten als auch Ressourcen enthalten.

Datenaustauschformat

ein einfaches Dateiformat für den Datenaustausch, das oft auf dem ASCII-Code beruht. Man benutzt es, um z. B. Texte, Tabellen oder Datenbanken in eine Form zu bringen, in der sie auf einem anderen Rechner mit anderen Programmen oder

sogar Betriebssystemen weiterverarbeitet werden können[UL13].

Deskriptor

Unter einem Deskriptor versteht man innerhalb dieser Arbeit ein Textdokument in einer Auszeichnungssprache wie XML oder JSON, welches einen Teilaspekt der Steuerungssoftware (Geräte, Programmabläufe, Templates, usw.) eindeutig beschreibt. Es bedient sich hierbei einer in der jeweiligen Auszeichnungssprache definierten Notation. Jeder Deskriptor ist vollkommen unabhängig von allen anderen Deskriptoren. Referenzen zwischen den Deskriptoren werden über Mappings und Mapping-Dateien realisiert.

Entwurfsmuster

Entwurfsmuster sind wiederverwendbare Lösungen für Probleme, welche innerhalb des Software Entwicklungsprozesses auftreten.

EVA-Prinzip

Das EVA-Prinzip (Eingabe-Verarbeitung-Ausgabe-Prinzip) ist ein Grundprinzip der Datenverarbeitung. Er kann entweder räumlich oder zeitlich betrachtet werden. In ersterem Fall werden verschiedene Bereiche der DV-Anlage auf die drei Bereiche Dateineingabe, Verarbeitung und Ausgabe verteilt, wobei diese Unterteilung sowohl für die Organisation der Hard- und/oder Software gültig ist. In letzterem Fall wird der Prozess als abgeschlossener Ablauf angesehen werden, wobei zuerst alle Eingaben erfasst werden, anschließend die Verarbeitung jener Eingabe erfolgt und zuletzt die Ausgabe in Form von Ergebnissen durchgeführt wird. Weiterhin sei bei zeitlicher Betrachtungsweise das *Streaming*-Konzept eingeführt. *Streaming*
Dabei können Eingaben kontinuierlich erfolgen, während die Abarbeitung der bereits eingetroffenen Daten gleichzeitig und die Ausgabe der "'älteren Daten"' direkt anschließend erfolgt. Für eine detaillierteren Beschreibung des EVA-Konzepts sei auf Fachliteratur der Informatik verwiesen[Fri06].

Fließinjektions-Analysesystem

Unter einem FIA versteht man einen fluidischen Laboraufbau bestehend Geräten zum Probentransport. Die Proben liegen dabei in flüssiger Form vor (wässrige Lösungen) und werden mit Hilfe von Laborpumpen, Ventilen, Autosamplern usw. aus Probenreservoirs entnommen, möglicherweise gemischt und zu einem Zielort

transportiert.

Framework

Ein Framework stellt in der Programmierung eine Rahmenstruktur zur Verfügung, innerhalb welcher ein Programmierer Software-Anwendungen erstellen kann. Es ist noch kein fertiges Programm, sondern stellt für Applikationen eine wiederverwendbare Struktur zur Verfügung. Beispielsweise stellt der Firefox-Browser ein Framework zur Verfügung um Plugins zu entwickeln. Sie können im Anschluss mit dem Firefox-Browser geladen und eingesetzt werden.

FTP

Das File Transfer Protokoll ist ein Datenübertragungsprotokoll, um Dateien zwischen verschiedenen Systemen auszutauschen. Dabei erfolgt der Austausch über IP-basierten Netzwerken. Oft wird FTP synonym für die Serveranwendung verwendet, die auf dem Computersystem läuft, auf welchem die Dateien hinterlegt werden sollen und mit welchem kommuniziert werden soll.

Handshake

Bei einem Handshake (Three-Way-Handshake) handelt es sich um ein Verfahren, um eine verlustfreie Datenübertragung zwischen zwei Kommunikationspartnern aufzubauen. Dabei überträgt ein Partner eine Anfrage, die von Partner zwei bestätigt und mit einer weiteren Anfrage quittiert wird. Partner 1 bestätigt im Anschluss die quittierte Anfrage. Durch diesen Ablauf ist gewährleistet, dass beide Anfragen von beiden Partnern empfangen worden sind. Der Handshake wird vor allem in der Netzwerktechnik (TCP) oder für asynchrone Busse eingesetzt.

Hash

Ein Hash in eine Abbildung einer großen Eingabemenge auf eine kleine Zielmenge, wobei es verschiedene Hashfunktionen gibt, um dies zu bewerkstelligen. Gute Hashfunktionen bilden verschiedene Eingabedaten auf verschiedene Hashwerte ab. Eine Kollision entsteht, wenn derselbe Hashwert mehreren Eingabedaten zugeordnet werden kann. Sie werden oft als Prüfsummen zur Identifikation von Datenübertragungsfehlern verwendet, aber auch in der Kryptologie, um beispielsweise Identifikationsdaten (Passwörter) nicht in Klartext zu speichern.

IP

Das "'Internet Protokoll"' IP ist ein Netzwerkprotokoll. Auf Basis einer IP-Adresse können dabei Computer innerhalb eines Netzwerks identifiziert werden. Mit Hilfe der sog. Subnetz-Maske können zusätzlich Netzwerke in Subnetzwerke gruppiert werden, zwischen welchem mit Routing Daten ausgetauscht werden können.

JavaScript

JavaScript ist eine Skriptsprache, die typisiert und objektorierntiert ist. Sie ist ursprünglich für Webbrowser im Rahmen von dynamischem HTML eingeführt worden. Für eine genauere Definition und Anwendungsgebiete von JavaScript sei auf einschlägige Fachliteratur der Informatik verwiesen.

JSON

JSON (JavaScript Object Notation) ist ein schlankes Datenaustauschformat, das für Menschen einfach zu lesen und zu schreiben und für Maschinen einfach zu parsen (Analysieren von Datenstrukturen) und zu generieren ist. Es basierend auf einer Untermenge der JavaScript Programmiersprache[JS:99][JSO13]. Bei JSON handelt es sich um ein Textformat, das komplett unabhängig von Programmiersprachen ist, aber vielen Konventionen folgt, die Programmieren aus der Familie der C-basierten Sprachen (inklusive C, C++, C#, Java, JavaScript, Perl, Python und vielen anderen) bekannt sind. Diese Eigenschaften machen JSON zum idealen Format für Datenaustausch[EIJ14].

Laboraufbau

Der *Laboraufbau* oder *analytische Aufbau* stellt einen Zusammenschluss an Peripherieelementen dar, mit welchem sich ein (oder mehrere) analytische Fragestellungen lösen lassen. Betrachtet man beispielsweise die Fragestellung: "'Abbildung und Charakterisierung von Pflanzenzellen"', so wäre ein möglicher analytischer Aufbau ein geeignetes Linsensystem mit Beleuchtungseinheit. Daraus lässt sich über den Prozess der Systemintegration ein Messsystem entwickeln oder, im größeren Maßstab, eine Laboranlage. In diesem Beispiel wäre der Laboraufbau ein Lichtmikroskop.

Laborgerät

Unter einem Laborgerät oder allg. Laborgeräten werden per Definition alle Gefäße, Werkzeuge und Hilfsmittel verstanden, welche in chemischen Laboratorien zur Durchführung von Synthesen und Analysen verwendet werden. In dieser Arbeit wird diese Definition erweitert auf: Alle Geräte, welche in chemischen Laboratorien zur Durchführung von Synthesen und Analysen verwendet werden und über eine Schnittstelle zum Anschluss an ein prozessorgesteuertes Messsystem verfügen.

LAN

Das Local Area Network ist ein Computernetzwerk, welches Computer in einem abgeschlossenen Bereich miteinander vernetzt. Typische Vertreter hierfür sind Ethernet oder WLAN.

LIMS

Ein Laborinformations- und Managementsystem (LIMS) ist eine IT-Applikation, die den Laborbetrieb in Bezug auf die administrativen und koordinativen Aufgaben der Probenbearbeitung sowie hinsichtlich der Erfassung und Auswertung ermittelter Analysendaten unterstützt[LIM14].

Mapping

Unter einem Mapping versteht man allgemein das Abbilden von Datenelementen zwischen verschiedenen Datenmodellen. Speziell in dieser Arbeit wird darunter eine Abbildung von Element(en) eines Deskriptors auf Element(e) eines anderen verstanden.

MIME

Bei MIME werden innerhalb des Transportmechanismus im Bereich der Nutzdaten mehrere Bereiche definiert, die sog. MIME-Boundaries, und mit einem MIME-Type versehen. Genau eine davon, die sog. Start-Boundary, enthält Textdaten, welche über Hyperlinks auf weitere Daten wie beispielsweise Binärdaten referenzieren. Dies referenzierten Daten befinden sich in den folgenden Boundaries, die jeweils über eine "'Content-ID"' identifiziert werden kann und einen anderen Content-Type bzw. ein anderes Content-Transfer-Encoding besitzen. Diese Vorgehensweise ist bereits standardisiert und z.B. mit XOP realisiert worden.[GMNR05] Weitere Informationen bzgl. MIME kann ich den korrespondierenden RFCs nachgelesen

werden[KC93, Fre96a, Fre96b, Moo96, Lev98]
.

Name-Wert-Paar

Ein Name-Wert-Paar ein eine Datenstruktur, die einen definierbaren Namen eine frei setzbaren Wert zuordnet.

Parser

Ein Parser ist ein Software-Modul, das Textdokumente in einem definierten Datenformat interpretieren und sie in das zugehöriges Datenmodell überführen kann. Ein XML-Parser beispielsweise kann ein XML Dokument lesen, verarbeiten und dem Benutzer in Form eines Datenobjekts zur Verfügung stellen, auf welches der Benutzer mit dort definierten Methoden zugreifen kann.

Peripherie

Unter der Peripherie werden hier Geräte oder Gerätekomponenten verstanden, die eine Rolle für einen zu entwickelnden Aufbau spielen. Dazu zählen alle Komponenten, die direkt oder indirekt miteinander kommunizieren können, wie z.B. Messgeräte, Lichtquellen, Verstärker, Probentransportsysteme wie mikrofluidische Komponente (Pumpen, Ventile) oder Autosampler.

Polling

Polling ist eine Strategie in der Datenverarbeitung, bei welcher in festgelegten Zeitintervallen eine Ressource abgefragt wird. Beispielsweise soll jede Sekunde ein Messwert von einem Strommessgerät abgerufen werden. Dazu muss eine Wiederholungsstruktur (Schleife) in regelmäßigen Abständen einen bestimmte Lesebefehl aufrufen, blockiert aber währenddessen den Hauptprozess. Eine andere Strategie stellen Events dar, bei welcher Ressource den aktualisierten Wert per Callback selbständig mitteilt.

Queue

Eine Queue ist in der Informatik eine Datenstruktur in Form einer Warteschlange und dient der Zwischenspeicherung von Daten in der Reihenfolge, in welcher sie eingegangen sind (FIFO-Prinzip = First-In-First-Out).

Request

Anfrage eines Clients an ein Server-System oder eine Server-Anwendung einer Client-Server-Architektur.

Response

Antwort des Server-Systems oder der Server-Anwendung einer Client-Server Architektur nach erfolgtem der Client-Anwendung.

REST

REST ist das Akronym für REpresentational State Transfer und stellt eine Kommunikationsarchitektur für verteilte Systeme dar. REST basiert auf dem Konzept der Ressourcen und verarbeitet diese. Im Gegensatz zu anderen Architekturen wie SOAP, welches Operationen definiert, werden über eine URI Ressourcen adressiert, auf welche dann mit typischen (meist HTTP basierten) Operationen zugegriffen wird. Zum Beispiel wird über die URI http://example.com/spectrometer/integrationTime auf die Ressource "'integrationTime"' des Geräts "'spectrometer"' zugegriffen. Über die HTTP Operation POST kann diese gesetzt werden, über GET empfangen. Die genaue Spezifikation von REST kann hier nicht widergegeben werden, da sie den Umfang sprengen würde[RRD07].

Serialisierung

Unter der Serialisierung von Daten (Gegenteil ist die Deserialisierung) wird die Überführung eines Datenmodells in eine (meist binäre) Datei verstanden. Damit kann der Inhalt des Datenmodells persistent gehalten werden und auch nach Abschalten und Wiedereinschalten des Systems erneut (durch Deserialisierung) abgerufen werden, um das Datenmodell wieder aufzubauen.

Simple Object Access Protocol

SOAP stellt ein Rahmenwerk (Framework) für Daten zur Verfügung, welche applikationsspezifische Informationen so beschreiben, damit unterschiedliche Anwendungen untereinander diese Daten austauschen können. Für SOAP selbst spielt das zugrunde liegende Übertragungsprotokoll keine Rolle, kann aber in Kombination mit Webservices die zu deklarierenden Methodenrümpfe vollständig beschreiben.

Speicherprogrammierbare Steuerung

Eine SPS ist ein Gerät, welches zur digitalen Automation d.h. Steuerung und Regelung von Maschinen eingesetzt wird.

Steuerungssoftware

Das Produkt der implementierten Ablaufsteuerung stellt letztendlich die *Steuerungssoftware* dar. Die Begriffe Ablaufsteuerung und Steuerungssoftware werden im Folgenden synonym betrachtet, wobei der Fokus des ersteren das theoretische Modell und des letzteren auf dem implementierten Produkt liegt.

Tag

In Auszeichnungssprachen werden Tags als Kürzel verwendet, die in spitzen Klammern eingeschlossen werden. Damit werden Daten zu klassifizieren und zu strukturieren. Ganz allgemein ist ein Tag eine Auszeichnung eines Datensatzes mit zusätzlichen Informationen.

Toolkit

Unter einem Toolkit wird in der Programmierung eine Sammlung an Funktionen, Schnittstellen und Bibliotheken in einer bestimmten Programmiersprache verstanden und soll einen Programmierer dabei die Entwicklung von Applikationen zu erleichtern.

Treiber

Ein Treiber ist eine Software und steuert die Interaktion mit Geräten, indem er über eine Hardware-Schnittstelle Daten zwischen einem Gerät und dem Betriebssystem austauscht. In Excalibur wird unter einem Treiber die Übersetzungsdatei zwischen Gerätebefehlen und menschenlesbaren Befehlen verstanden.

URI

Der Uniform Resource Identifier ist eine Zeichenfolge und dient zur Identifizierung einer physischen Ressource, z.B. identifiziert "'www.google.de"' die Webseite der Suchmaschine Google, "'name@uni-tuebingen.de"' die Email-Adresse des Benutzers "'name"' an der Universität Tübingen.

WebDAV

Web-based Distributed Authoring and Versioning ist ein Standard im Internet, um Daten für andere Personen bereitzustellen. Es ist eine Erweiterung von HTTP und ermöglicht es Personen, Verzeichnisse im Internet wie eine (lokale) Festplatte zu verwenden.

Webservice

Ein Webservice ist eine Software, welche über eine Netzwerkschnittstelle Dienste bereitstellt, die eindeutig identifizierbar sind (URI) und über eine klar definierte Schnittstellenbeschreibung festlegt, wie mit dem Dienst zu kommunizieren ist.

XML

XML (Abkürzung von engl.: extensible markup language) ist eine Metasprache für die Definition von anwendungsspezifischen Auszeichnungssprachen, die vom W3C normiert wird. Mittels XML können somit Auszeichnungssprachen für beliebige Anwendungsbereiche maßgeschneidert werden und anwendungsspezifische Datenstrukturen beschrieben werden[EDW14].

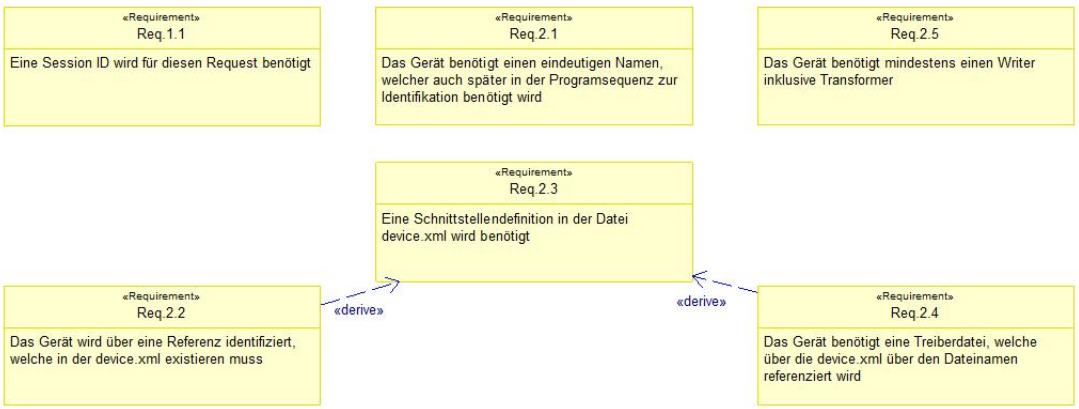
A

UML Struktogramme

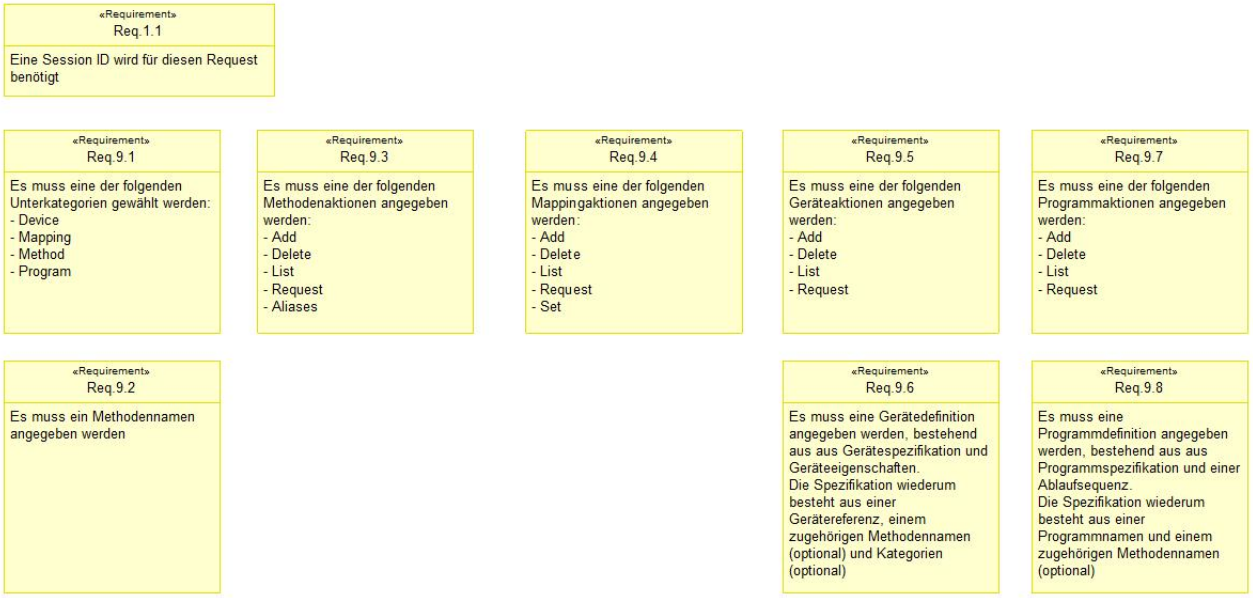
A.1. Anwendungsfälle

A.1.1. Requirements

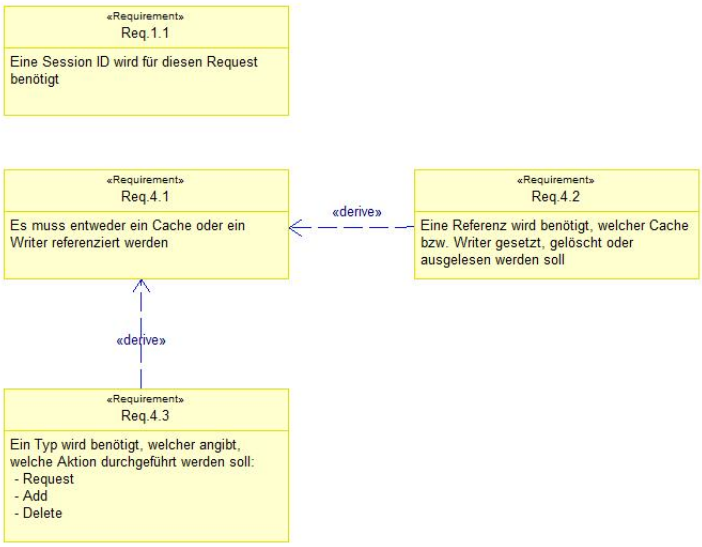
Requirements des ConnectionRequests



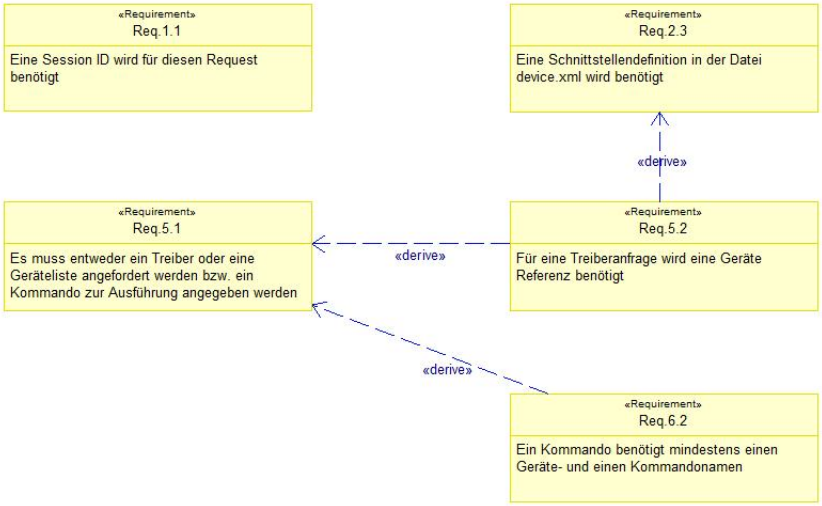
Requirements des ConfigurationRequests



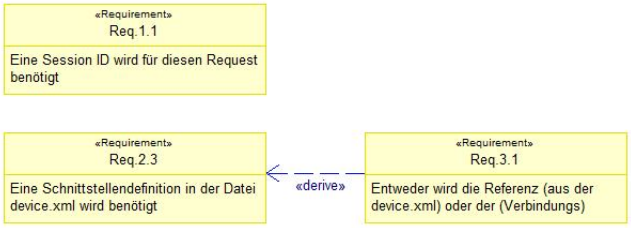
Requirements des DataRequests



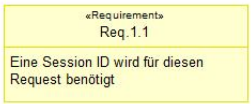
Requirements des DeviceRequests



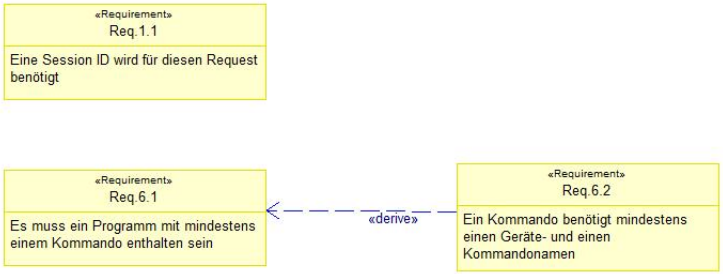
Requirements des DisconnectionRequests



Requirements des EventingRequests



Requirements des ProgramRequests



Requirements des SystemRequests

B

XML/XSD Dokumente

B.1. Commons

In der Commons.xsd sind generelle Definitionen zu finden, welche an verschiedenen Stellen wieder benötigt werden. Darunter fallen beispielsweise Datentypen, Einheiten, Länderkürzel oder Fehlerlevel.
Das XML Schema der Commons ist sehr umfangreich und kann deshalb hier nicht dargestellt werden. Es kann im „Schema“-Ordner unter „common.xsd“ auf der DVD gefunden werden.

B.2. Zentrale Excalibur Konfiguration

Das XML Schema der Excalibur Konfiguration ist sehr umfangreich und kann deshalb hier nicht dargestellt werden. Es kann im „Schema“-Ordner unter „configuration.xsd“ auf der DVD gefunden werden.

B.3. Excalibur Treiber

Das XML Schema des Treibers ist sehr umfangreich und kann deshalb hier nicht dargestellt werden. Es kann im „Schema“-Ordner unter „driver.xsd“ auf der DVD gefunden werden.

Eine kurze Definition der wichtigsten Elemente soll aber dennoch gegeben werden:

Die in der driver.xsd gezeigten Eigenschaften sind ausreichend, um ein Gerät vollständig zu charakterisieren. Die Eigenschaften *devicename*, *description* und *supplier* spezifizieren Eigenschaften, welche vom Gerätehersteller festgelegt werden und genau so übernommen werden können.
Über *include* können weitere weitere Treiberdateien eingebunden werden. Dabei sollte beachtet werden, dass Kommandos mit gleichen IDs bzw. Aliases immer von Kommandos der Hauptdatei überschrieben werden. Gleiche IDs in derselben Treiberdatei (ohne Include) werden nicht toleriert und mit einem Fehler quittiert. Dafür ist in der XSD (s. Abschnitt B.3) ein Unique-Tag gesetzt.
In den Eigenschaften *terminator* bzw. *responseTerminator* wird ein global gültiger Kommando- bzw. Antwort-Terminator angegeben. Sie können im Kommandotyp individuell überschrieben werden, falls für eine bestimmte Aktion kein oder ein anderer Terminator gültig sein sollte. Der Antwort-Terminator hat eine eigene Bedeutung. Er wird benötigt, falls vom Gerät eine Antwort unbekannter Länge gesendet wird, von welcher das Ende erst durch eine Abschlusssequenz kenntlich gemacht wird.
Der *decimalSeparator* legt global das Trennzeichen für Gleitkommazahlen fest. Auch er kann individuell von jedem Kommandotyp angepasst werden.
Mit der *category* wird dem Gerät eine Kategorie mit vordefinierten Kommandosätzen zugewiesen, die vom Gerät auf jeden Fall implementiert werden müssen. Näheres zu Kategorien wird in Abschnitt 3.3.8 beschrieben.
Über *declare* und *define* können Variablen festgelegt werden, auf welche vom Treiber zugegriffen werden kann. Sie können an fast jeder beliebigen Stelle im Treiber verwendet werden und werden jeweils zur Laufzeit aufgelöst. Der Unterschied zwischen *declare* und *define* liegt im Zeitpunkt der Zuweisung seiner Werte. Während *define* eine direkte Zuweisung eines Wertes im Treiber erwartet, müssen „declared“ Variablen durch ein Gerätekommando zur Laufzeit gesetzt werden. Dafür muss eine Komposition definiert

werden, wovon eines der zugrundeliegenden Excalibur-Kommandos ein *target* definieren muss, welches denselben Namen trägt wie die „declared“ Variable. Dessen Kommando ID muss anschließend der *declared*-Eigenschaft unter dem Attribut *for* kenntlich gemacht.
In den Eigenschaften *state* und *idleState* werden Gerätestati definiert und das Idle-State Kommando festgelegt. In ersterem kann hierbei der Status-Namen, den möglichen zugehörigen herstellerspezifischen Betriebsmodi und eine beliebige Anzahl an Aliasen festgelegt werden. In letzterem wird ein Kommando (referenziert über seine ID) festgelegt, welches ausgeführt werden soll, um in den Status „Idle“ zu gelangen.
Mit *response* kann eine Liste an Standard Antwort-Tokens definiert werden. Auf diese kann innerhalb jedes Kommandotyps über die Eigenschaft *response* und dessen Attribut *ref* referenziert werden.
Letztendlich werden in *command* und *compositeCommand* die Liste der Kommandotypen und Kompositionen definiert, wie oben beschrieben.

Im Folgenden sollen die wichtigsten Elemente der Gerätekommandos im Treiber definiert werden:

Außer den Tokens umfasst der Kommandotyp weitere Strukturelemente, die für eine sinnvolle Anwendung wichtig sind. Sie sollen im Folgenden erklärt werden. Da einzelne Eigenschaften über komplexe Typen realisiert werden, welche nicht hier sondern erst in Abschnitt B.3 vollständig gelistet sind, soll von diesen komplexen Typen hier nur eine Erklärung der wichtigsten Eigenschaften genügen.

Das Attribut *id* definiert einen eindeutigen Namen für das Excalibur-Kommando, unter welchem es dem System bekannt gemacht wird. Anhand der ID kann das Kommando z.B. in Ablaufsequenzen identifiziert werden. *Dies ist die einzige obligatorische Eigenschaft des Excalibur-Kommandos.*
In *description* können Beschreibungen und das Funktionsprinzip des Kommandos dargelegt werden. Dafür kann zusätzlich die Eigenschaft *lang* also „Sprache“ gesetzt und für jede Sprache eine eigene Beschreibung zur Verfügung gestellt werden. Abhängig von der in Excalibur konfigurierten Systemsprache wird während der Laufzeit die richtige (oder die Standard-Sprache *en*) gewählt und angezeigt.
Mit der *visibility* kann entschieden werden, ob ein Kommando direkt aufgerufen werden kann (= *public*) oder nur über Kompositionen ein indirekter Zugriff erlaubt wird (= *private*). Näheres zu Kompositionen wird weiter unten erläutert.

Der *decimalSeparator* gibt das Trennzeichen für Gleitkommazahlen an.
Mit der *continuity* werden die sog. *Start*-Kommandos definiert. Näheres zu diesem Kommandotyp wird in Abschnitt 3.3.6 erläutert.
Unterbrechungen in Ablaufsequenzen können mit Hilfe des *interrupt* definiert werden. Muss, z.B. bedingt durch manuelle Benutzereingaben am Gerät, der Ablauf bis zum Ausführen ebendieser bzw. Bestätigung einer Mitteilung unterbrochen werden, können hier Mitteilungen (in verschiedenen Sprachen) definiert werden. Der Ablauf wird anschließend so lange unterbrochen, bis über einen bestimmten Request, den *Resume-Request*, das Zeichen zum Wiederaufnehmen des Ablaufs gegeben wird.
Mit Hilfe der Eigenschaften *postWriteTime* und *postReadTime* können Wartezeiten definiert werden (nach dem Schreib- bzw. anschließenden Lesevorgang), bevor die nächste Aktion durchgeführt werden soll.
Der *state* definiert den weiter oben in Abschnitt 3.3.3 eingeführten Gerätestatus.
Mit einem (oder mehreren) *alias* können alternative Namen für die Kommando-ID spezifiziert werden, mit welchen das Kommando ebenfalls in Ablaufsequenzen identifiziert werden kann.
Wie bereits in Abschnitt 3.3 angedeutet, können mit Hilfe von *Tokens* Werte für die Gerätebefehle- und -antworten definiert werden. Sie haben mehrere mehrere Eigenschaften, die gesetzt werden können (s. Tabelle B.1).

Tabelle (B.1)

Eigenschaft	Werte	Wirkung	Beispiel
Funktionale Eigenschaften			
Datentyp	siehe Abschnitt B.1	u liegt im angegebenen Datentyp vor und wird aus dem vorliegenden String in diesen umgewandelt	Ist $u = 5$, so ist $q = 0x05$ mit „int32“ und $q = 0x35$ mit „string“
Kodierung	(ASCII|UTF-8|UTF-16|BIN|OCT|DEC| HEX), Std: ASCII	u liegt in der angegebenen Kodierung vor und wird in dieser in q umgewandelt	Ist $u = A$, so ist $q = 0x0A$ mit „HEX“, $q = 0x41$ mit „ASCII“ und ERROR mit „DEC“
Default	Abhängig von Datentyp und Kodierung	Legt einen Standard Wert für q fest, falls u leer ist	Ist Default „20“ (DEC) und u leer oder nicht vorhanden, dann ist $q = 0x14$
Min	Zahl	Prüft, ob Zahl größer/gleich Min (*wenn Datentyp numerisch*), sonst ERROR)	Ist Min „5“ und $u < 5$ wird ERROR geworfen
Max	Zahl	Prüft, ob Zahl kleiner/gleich Max (*wenn Datentyp numerisch*), sonst ERROR)	Ist Max „5“ und $u > 5$ wird ERROR geworfen
Organisatorische Eigenschaften			
Alias	Zeichenkette	Weist dem Value-Token einen Namen zu, mit welchem ein Benutzer auf diesen referenzieren kann	Für den Befehl „pump“ kann im Rahmen einer Spritzenpumpe auch ein Alias „position“ gesetzt werden für eine genaue Positionierung der Spritze
Constant	Name-Wert-Paar	Weist einem Namen einen gewissen Wert q zu, wobei jener ohne Transformation in Z verwendet wird	Für das letzte Beispiel der Spritzenpumpe können die Konstanten „FULL“ und „EMPTY“ gesetzt werden

Listing (B.1)

XML Schema des Value-based Types

```
<complexType name="valueBasedType">
 <sequence>
  <element name="value" minOccurs="0" maxOccurs="unbounded">
   <complexType>
    <simpleContent>
     <extension base="string">
      <attribute name="index" default="0" use="optional">
       <simpleType>
        <restriction base="string">
         <pattern value="[0-9]+(\.[0-9]+)?" />
        </restriction>
       </simpleType>
      </attribute>
     </extension>
    </simpleContent>
   </complexType>
  </element>
 </sequence>
</complexType>
```

B.4. Definition der Schnittstellen

Das XML Schema der Schnittstellendefinition ist sehr umfangreich und kann deshalb hier nicht dargestellt werden. Es kann im „Schema“-Ordner unter „interfaces.xsd“ auf der DVD gefunden werden.

B.5. Ein- und Ausgabe

Das XML Schema der Ein-/Ausgabe ist sehr umfangreich und kann deshalb hier nicht dargestellt werden. Es kann im „Schema“-Ordner unter „systemio.xsd“ auf der DVD

gefunden werden.

B.6. Programme und Skripte

Das XML Schema der Ablaufsequenzen und Skripte ist sehr umfangreich und kann deshalb hier nicht dargestellt werden. Es kann im „Schema“-Ordner unter „program.xsd“ auf der DVD gefunden werden.

Auf einige Bereiche wird im Folgenden Bezug genommen, da sie für das Verständnis einiger Abschnitte der Thesis relevant sind.

B.6.1. Programme

Konditionale Strukturen

Listing (B.2)

XML Schema des Choose Types. Dies entspricht einer Switch-Case bzw. If-Else-Struktur der gängigen Programmiersprachen.

```
<complexType name="programChooseType">
 <sequence>
  <choice maxOccurs="unbounded">
   <element name="if">
    <complexType>
     <sequence>
          <group ref="tns:programBlockGroup" minOccurs="0"
               maxOccurs="unbounded" />
     </sequence>
     <attribute name="test" use="required" type="string" />
    </complexType>
   </element>
  </choice>
  <choice minOccurs="0" >
   <element name="else" type="tns:programEmptyType" />
  </choice>
```

```
16  </sequence>
17 </complexType>
```

Wiederholungsstrukturen

Listing (B.3)

XML Schema des For Types. Dies entspricht einer For-Schleife in gängigen Programmiersprachen.

```
1  <complexType name="programForType">
2   <sequence>
3    <group ref="tns:programBlockGroup" minOccurs="0"
4      maxOccurs="unbounded" />
5   </sequence>
6   <attribute name="initialization" type="string" use="
        optional" />
7   <attribute name="test" type="string" use="optional" />
8   <attribute name="condition" type="string" use="optional" />
9   <attribute name="iteration" type="string" use="optional" />
10 </complexType>
```

Listing (B.4)

XML Schema des While Types. Dies entspricht einer While-Schleife in gängigen Programmiersprachen.

```
1  <complexType name="programWhileType">
2   <sequence>
3    <group ref="tns:programBlockGroup" minOccurs="0"
4      maxOccurs="unbounded" />
5   </sequence>
6   <attribute name="test" use="required" type="string" />
7  </complexType>
```

Funktionen

Tabelle (B.2)

Übersicht der Stringverarbeitungs-Routinen mit kurzer Begriffserklärung

Methode	Erklärung
charat	Extrahiert ein Zeichen an einer bestimmten Position
concat	Fügt zwei Zeichenketten zusammen
indexof	Suche das erste Vorkommen eines Zeichens und gibt die Position des Zeichens zurück
regexp	Wendet einen regulären Ausdruck auf die Zeichenkette an und gibt das Ergebnis zurück
strlen	Gibt die Länge der Zeichenkette zurück
substr	Extrahiert eine Zeichenkette zwischen zwei Positionen einer Zeichenkette und gibt diese zurück

Listing (B.5)

XML Schema des Function Types. Dies entspricht einer Methode in gängigen Programmiersprachen.

```
<complexType name="programFunctionType">
 <sequence>
  <element name="parameter" minOccurs="0" maxOccurs=
    "unbounded" type="tns:programFunctionParameterType"/>
  <group ref="tns:programBlockGroup" minOccurs="0"
    maxOccurs="unbounded"/>
 </sequence>
 <attribute name="name" use="required" type="string"/>
</complexType>
```

Stringverarbeitung

B.6.2. Skripte

Im Folgenden wird die Datenstruktur von Skripten mit Hilfe eines XML Schemas wiedergegeben:

Listing (B.6)

Struktur eines Skripts mit Ein- und Ausgabevariablen

```
<complexType name="scriptType">
 <sequence>
  <element name="script" type="string"/>
  <element name="var" minOccurs="0" maxOccurs="unbounded">
   <complexType>
    <attribute name="key" type="string" use="required"/>
   </complexType>
  </element>
  <element name="param" minOccurs="0" maxOccurs="unbounded">
   <complexType>
    <attribute name="key" type="string" use="required"/>
    <attribute name="type" type="common:dataTypeType" use="
         optional"/>
    <attribute name="value" type="string" use="optional"/>
   </complexType>
  </element>
  <element name="output" minOccurs="0" maxOccurs=
   "unbounded">
   <complexType>
    <attribute name="key" type="string" use="required"/>
    <attribute name="type" type="common:dataTypeType" use=
      "optional" default="double"/>
   </complexType>
  </element>
 </sequence>
</complexType>
```

Mit Hilfe der *var*-Eigenschaft können Zielvariablen an das Skript übergeben werden; mit Hilfe der Notation „*#key*“ wird der Kanal mit dem Namen *key* an das Skript übergeben. Mit Eigenschaft *param* können Konstanten definiert werden, in welchen mit *key* der Name, *type* der Datentyp und mit *value* ein Wert definiert wird. Zuletzt kann über den *output* festgelegt werden, wie die Ausgabe des Skripts die Daten wieder zurück an das Excalibur-System gibt. *key* definiert wieder den Namen der Ausgabe und *type* den

Datentyp; wie auch der *var*-Eigenschaft wird mit dem #-Zeichen am *key* ein Kanal referenziert.
Das erste zu definierende Element *script* gibt letztendlich in textueller Form die Berechnungsanweisungen in der jeweils gewünschten mathematischen Auszeichnungssprache wider, worin mit Hilfe der unter *key* definierten Variablen auf deren Werte zugegriffen werden kann. Wie immer definieren linksseitige Werte „Ausgaben“ und rechtsseitige Werte „Eingaben“. Das folgende Beispiel soll die Funktionweise anhand der einfachsten Skript-Art vorstellen: *Beispiel*

Listing (B.7)

Beispiel eines einfachen Skripts zur Berechnung der Summe aus einer Zielvariablen, einer Konstanten und eines Kanals. Target und channel müssen dabei bereits durch einen anderen Prozess definiert worden sein.

```
1 <formula>
2  <script>result = target + param + channel</script>
3  <var key="target"/>
4  <var key="@channel"/>
5  <param key="param" type="int32" value="5"/>
6  <output key="result" type="int32"/>
7 </formula>
```

Es werden in diesem Beispiel die Variablen *target* und *param* addiert, anschließend der Kanal *channel* dazugezählt und die Summe in der Ausgabevariablen *output* hinterlegt. *target* ist eine Zielvariable und wird über *var* in das Skript importiert (ebenso wie *channel* einen Kanal über *var* importiert), *param* ist eine Konstante und wird über die Eigenschaft *param* definiert. Zuletzt wird die Ausgabevariable *result* über *output* exportiert.

B.7. Templates

Das XML Schema des Templates ist sehr umfangreich und kann deshalb hier nicht dargestellt werden. Es kann im „Schema“-Ordner unter „template.xsd“ auf der DVD gefunden werden.

Auf einige Bereiche wird im Folgenden Bezug genommen, da sie für das Verständnis einiger Abschnitte der Thesis relevant sind.

Listing (B.8)

XML Schema des Datatable Types. Ähnlich wie in HTML wird hiermit eine Tabelle im Body abgebildet, in welche Daten strukturiert eingefügt werden können.

```
<complexType name="bodyDataTableType">
 <sequence>
  <element name="column" maxOccurs="unbounded" type=
     "tns:bodyDataTableColumnType"/>
 </sequence>
 <attribute name="storage" type="tns:bodyStorageType"
    use="optional"/>
 <attribute name="style" type="tns:bodyStyleType"
    use="optional"/>
</complexType>
```

Listing (B.9)

XML Schema des Datatable Column Types. Dies entspricht einer Spalte der obigen Tabelle (im Gegensatz zu HTML, wo mit Zeilen gearbeitet wird).

```
<complexType name="bodyDataTableColumnType">
 <sequence>
  <element name="head" minOccurs="0" type=
     "tns:bodyDataTableHeadType"/>
   <choice maxOccurs="unbounded">
        <group ref="tns:tableColumnTagGroup"/>
        <element name="dataRow" type="tns:bodyDataRowType"/>
   </choice>
 </sequence>
 <attribute name="channel" use="optional" type="string"/>
 <attribute name="unit" type="string" use="optional"/>
 <attribute name="type" type="common:dataTypeType"
    use="optional"/>
</complexType>
```

Über die Eigenschaft *channel* im „bodyDataTableColumnType“ kann der gesuchte Kanal des Datenstroms angegeben werden, um Daten in die vorgesehene Spalte zu schreiben. Mit Hilfe der Eigenschaft *type* kann der Datentyp angegeben werden, wenn in Datentypsensitiven Datensenken (z.B. Datenbank) geschrieben werden soll. Über die *unit* kann zusätzlich eine Einheit für den zu schreibenden Wert spezifiziert werden. Weiterhin können die im Head definierten Eigenschaften in jeder Tabelle überschrieben werden. Auf eine genauere Erklärung und eine vollständige Auflistung aller Eigenschaften sei auf Anhang B.7 verwiesen.

B.8. Methoden

Das folgende XML Schema bildet das Datenmodell für Methoden ab:

Listing (B.10)

XML Schema des Method Types. Die Methode ist die Kapselung von Geräten, Kategorien und Ablaufsequenzen in Excalibur. Sie sollen die programmatische Repräsentation der analytischen Methode darstellen.

```
1  <complexType name="methodType" xmlns="http://www.excalibur.
       net/method">
2   <sequence>
3    <element name="name" type="string"/>
4    <element name="deviceSpecification"
5      type="deviceSpecificationType" minOccurs="0"
6      maxOccurs="unbounded"/>
7    <element name="categorySpecification"
8      type="deviceCategorySpecificationType" minOccurs="0"
9      maxOccurs="unbounded"/>
10   <element name="programSpecification"
11     type="programSpecificationType" minOccurs="0"
12     maxOccurs="unbounded"/>
13   <element ref="fs:frameset" minOccurs="0"/>
14   <group ref="io:readerGroup" minOccurs="0"
15     maxOccurs="unbounded"/>
```

```
16   <group ref="io:filterGroup" minOccurs="0"
17     maxOccurs="unbounded"/>
18   <group ref="io:deserializerGroup" minOccurs="0"
19     maxOccurs="unbounded"/>
20   <group ref="io:writerGroup" minOccurs="0"
21     maxOccurs="unbounded"/>
22   <group ref="io:transformerGroup" minOccurs="0"
23     maxOccurs="unbounded"/>
24   <group ref="io:serializerGroup" minOccurs="0"
25     maxOccurs="unbounded"/>
26  </sequence>
27 </complexType>
```

Listing (B.11)

XML Schema des Category Types. Eine Kategorie stellt eine Funktionsbeschreibung für Gruppen von Geräten dar.

```
1 <complexType name="deviceCategorySpecificationType" xmlns="
      http://www.excalibur.net/method">
2  <sequence>
3   <element name="name" type="string"/>
4   <element name="command" type="deviceCategoryCommandType"
        maxOccurs="unbounded"/>
5  </sequence>
6 </complexType>
```

B.9. Core

Das XML Schema des Core ist sehr umfangreich und kann deshalb hier nicht dargestellt werden. Es kann im „Schema“-Ordner unter „core.xsd“ auf der DVD gefunden werden.

C

Detaillierte Informationen

C.1. Hardware Schnittstellen

C.1.1. RS232

Bei RS-232 handelt es sich um eine Spannungsschnittstelle, bei welcher Daten d.h. binäre Zustände über zwei unterschiedliche elektrische Spannungspegel realisiert werden. Dabei wird auf den Datenleitungen (TxD, RxD) mit negativer Logik gearbeitet. Eine logische „0“ wird durch eine Spannung zwischen -3 V und -15 V dargestellt, eine logische „1“ durch eine Spannung zwischen +3 V und +15 V. Übertragen werden vollständige Datenwörter, die asynchron von jedem der beiden Teilnehmer zu jedem beliebigen Zeitpunkt (sobald die Leitung „frei“ ist) auf die Leitung gelegt werden können. Da es sich bei RS-232 um eine Punkt-zu-Punkt Verbindung handelt und kein gemeinsamer Takt existiert, müssen sich die beiden Teilnehmer im Vorfeld auf die Übertragungsgeschwindigkeit, Daten- und Stop-Konditionen einigen. Die Übertragungsgeschwindigkeit, die Datenrate, muss dazu manuell vom Benutzer auf beiden Seiten festgelegt werden. Eingestellt wird dafür die sog. Baudrate, welches eine Maßeinheit der Schrittgeschwindigkeit (s. Baudrate) ist, die hier aber wegen der binären Übertragungsweise der Übertragungsgeschwindigkeit entspricht (Standard Baudrate: 9600, weitere s. unten). Eng gekoppelt mit der Übertragungsgeschwindigkeit ist die Kabellänge, da die Signalqualität mit zunehmender Leitungslänge wegen Leitungsreflexionen abnimmt. Reflexionen ergeben sich aufgrund des Wellenwi-

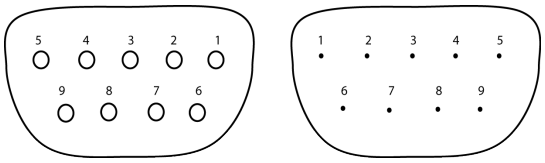


Abbildung (C.1)
Bild des 9-poligen RS-232 Steckers - weiblich (links) und männlich (rechts) mit der entsprechenden Nummerierung der Pins

Tabelle (C.1)
Buchsenbelegung der RS-232 Schnittstelle

Pin/Buchse	Bedeutung
1	RTS (Request to Send)
2	RxD, RX, RD (Receive Data)
3	TxD, TX, TD (Transmit Data)
4	DTR (Data Terminal Ready)
5	GND (Ground)
6	DSR (Data Set Ready)
7	RTS (Request to Send)
8	CTS (Clear to Send)
9	RI (Ring Indicator)

derstandes in Leitern. Daraus ergibt sich für unterschiedliche Übertragungsgeschwindigkeiten ein maximale Leitungslänge, bei der die Reflexionen begrenzt werden (Für 9600 sind das ca. 150 m, für 115200 unter 2 m).
Sollen nun Daten übertragen werden, so muss der sendende Teilnehmer ein sog. Startbit, eine logische 0, setzen. Dadurch wird dem Empfänger mitgeteilt, dass die folgenden Bits Datenbits sind. Anschließend sendet er nacheinander bitweise, in der festgelegten Baudrate, die zu übertragenden Daten. Auch diese Anzahl an Datenbits muss im Vorfeld festgelegt werden und beträgt im Normalfall 7 oder 8 Datenbits per Zyklus. Zuletzt erfolgt das sog. Stopbit, welches über 1 bis 1,5 Schritten den Pegel der Leitung auf eine logische „0“ zieht.
Ein Bild der Steckverbindung der RS-232 Schnittstelle ist in Abbildung C.1 gezeigt mit der entsprechenden Funktionalität der einzelnen Leitungen (Pins/Buchsen). Die Leitung TxD, RxD und GND sind obligatorisch für jede Anwendung, da es sich dabei um die Sendeleitung (TxD) und Empfangsleitung (RxD) für Daten und bei letzterem um die Masse handelt, gegen welche die Spannungen gemessen werden. Für eine Erklärung der restlichen Leitungen sei auf entsprechende Fachliteratur verwiesen[EK:14, Kai97].

Damit die serielle Schnittstelle verwendbar ist, müssen noch weitere Einstellungen erklärt werden. Die wohl wichtigste Eigenschaft ist der *Port*, welche die Bezeichnung der Schnittstelle angibt, unter welchem das Gerät im Betriebssystem gefunden werden kann. Unter Windows® lauten diese meistens COM*, wobei $*$ eine Zahl beginnend ab 1 darstellt; Unix Systeme verwenden hierfür meist /dev/tty*. Das Betriebssystem reserviert

hierbei einen gewissen Speicherbereich im Hauptspeicher-Adressraum, mit welchen über übliche Speicherzugriffsroutinen gearbeitet werden kann und somit ein vollständiger Zugriff auf die Hardware ermöglicht wird.
Eine weitere sehr wichtige Eigenschaft ist die Geschwindigkeit, welche indirekt über die *Baudrate* festgelegt wird (s. oben). Diese muss für die konfigurierte Schnittstelle und dem Gerät übereinstimmen, da sonst keine Übertragung möglich ist. Je höher die Baudrate gewählt wird, desto schneller werden Daten über die Leitung gesendet. Allerdings unterstützt nicht jedes Gerät jede Baudrate, da diese abhängig vom verbauten Quarzoszillator ist. Typische Werte sind 75, 110, 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600 und 115200.
Mit der Eigenschaft *databits* wird die Anzahl Bits festgelegt, welche pro Sende-/Empfangsvorgang (Informationswort) nacheinander übertragen werden; 5 - 9 Bits kann als Wert hierfür gewählt werden, obwohl die meisten modernen Geräte 8 Bits verwenden (also genau 1 Byte). Nach dem Senden der Daten wird das sog. *stopbit* gesetzt, welches entweder über eine Zeit von einem, 1.5 oder 2 Bits (pro BAUD) den Pegel auf „0“ zieht.
Über die *parity* können letztendlich Übertragungsfehler detektiert werden. Dabei werden im einfachsten Fall die Anzahl an logischen „1“ gezählt und im Falle einer „EVEN“ *parity* wird als Paritätsbit bei einer geraden Anzahl eine logische „0“ an das Datenwort gehängt, bei einer ungeraden Anzahl eine logische „1“; bei „ODD“ ist dies genau umgekehrt. Wird „NONE“ gesetzt, so wird kein Paritätsbit an das Datenwort gehängt. „MARK“ und „SPACE“ sind eher unüblich und sollen hier nicht weiter behandelt werden.
Letztendlich kontrolliert die Datenflusssteuerung *flowcontrol*, dass eine möglichst kontinuierliche Datenübermittlung vor allem bei langsam arbeitenden Geräten erfolgt. Da bei letzteren eine Datenüberlastung die Folge wäre, wenn die Datenübertragung nicht zeitweise und kontrolliert ausgesetzt würde, kann über diesen Parameter das Verfahren eingestellt werden, um diese Engpässe zu organisieren. Mögliche Einstellungen hierfür sind XON/XOFF, RTS/CTS und DTR/DSR, wobei die genaue Funktionsweise den Rahmen dieser Arbeit sprengen würde und auf einschlägige Fachliteratur verwiesen sei.

C.1.2. USB

USB (Universal Serial BUS) ist ebenso wie RS-232 Schnittstelle eine serielle Schnittstelle. Die Datenübertragung erfolgt symmetrisch, d.h. es werden zwei Datenleitungen (D+ und D-) benötigt, um diese zu transportieren. Dabei werden auf einer Leitung die

Datenwörter übertragen, während auf der zweiten Leitung die identischen Datenwörter invertiert vorliegen. Hierdurch können durch Differenzbildung beim Empfänger die Störungen der Leitungen herausgerechnet werden. Zwei weitere Leitungen werden für die Stromversorgung verwendet, wobei eine eine Spannung von 5 V ± 5% zur Verfügung stellt (VBUS) und die letzte die Masse (GND) darstellt gegen welche gemessen wird. Der Vorteile der USB Schnittstelle liegen in der hohen Datenübertragungsgeschwindigkeit von über 10 GBit/s (USB 3.0 Standard) und einer Bus-Topologie, wodurch an einem Host-Controller (meist der PC) bis zu 127 Geräte angeschlossen werden können. Die USB-Spezifikation sieht vor, dass ein ebensolcher Host-Controller vorhanden ist, welcher die „Administration“ des Busses übernimmt. Die Übertragung erfolgt nach dem Token-Ring Prinzip. Der Host-Controller sendet hierbei in einem festgelegten Rhytmus alle in ihm registrierten Geräte einen Token, welches unter anderem die Adresse im 7-bit Format enthält. Durch Vergleich mit seiner eigenen Adresse findet ein Gerät heraus, ob ihm aktuell die Aufmerksamkeit des Host-Controllers gewidmet ist. Ist dies der Fall, sendet er ihm entweder Daten zurück (zwischen 8 und 255 Datenwörter) oder teilt ihm mit, dass keine Daten vorhanden sind. Ist dies nicht der Fall, legt er den Token wieder auf den Bus, damit ein anderes Gerät die Überprüfung vornehmen kann. Ist das Gerät identifiziert und die Datenübertragung eingeleitet, findet ein Datenfluss zwischen dem Host-Controller und einem bestimmten Endpunkt des USB-Gerätes statt. Jedes Gerät kann verschiedene Endpunkte haben, wobei jeder unabhängig voneinander operiert und eindeutig nummeriert ist. Es können maximal 16 Endpunkte in *normalen* Geräten definiert sein dürfen, für *langsame* Geräte maximal zwei. Jeder dieser Endpunkte kann durch seine Nummer und die Geräte-Adresse korrekt per Software angesprochen werden. Eine Verbindung zwischen einer Software (über den Host-Controller) und dem Endpunkt auf der Peripherie wird als Pipe bezeichnet, welche einen Datenfluss darstellt. Es gibt zwei Arten von Pipes: Stream-Pipes und Message-Pipes, die sich in der Datenstruktur und der direktionalen Basis unterscheiden. Während Stream-Pipes immer unidirektional sind und USB-unspezifische Datenstrukturen überträgt, gilt für Message-Pipes das genaue Gegenteil (bidirektional und USB-spezifisch). Zwar arbeiten letztere bidirektional, allerdings unterstützen sie nur Kontroll-Transfers. Für erstere gibt es nun verschiedene Transfer-Arten, welche unterschiedlichen Spezifikationen genügen. Hier sei aber auf die einschlägige Fachliteratur verwiesen.[Red06]
Ein weiterer Vorteil von USB ist Fähigkeit des Hot plug and play. So können Peripheriegeräte angeschlossen werden, ohne dabei den Host-Controller ausschalten zu müssen. Sobald ein Gerät angschlossen werden, erkennt er dies anhand einer Spannungsänderung

Abbildung (C.2)
Typischer männlicher USB-Stecker (Typ-A).

auf den Datenleitungen D+ und D- und leitet dann die Initialisierungssequenz ein. Dabei weist der Host-Controller dem Gerät kurzzeitig die (reservierte) Adresse „0“ zu, beginnt intern eine Aufzählungsprozedur, indem er alle bereits registrierten Geräte nach ihren Adressen abfragt. Ist dieser Vorgang abgeschlossen, teilt er dem neu zu registrierenden Gerät seine entgültige, frei Adresse zu, welches sich anschließend beim Host-Controller identifiziert und seine internen Merkmale mitteilt.

C.1.3. GPIB

GPIB Geräte kommunizieren mit anderen GPIB Geräten über geräteabhängigen Nachrichten und Schnittstellen-Nachrichten über das eigene Schnittstellen System. Dabei handelt es ich bei den Geräte-abhängigen Nachrichten um Daten, welche darüber hinaus gerätespezifische Informationen wie Programmierbefehle, Messresultate, Maschinen-Stati, aber auch Daten-Dateien transportieren. Im Gegensatz dazu organisieren Schnittstellen-Nachrichten den BUS selbst. Gewöhnlicherweise werden sie als Kommandos oder Kommando-Nachrichten bezeichnet, welche bestimmte Funktionalitäten wie Initialisierung, Adressierung und Änderungen von Geräte-Modi übernehmen. Die GPIB Schnittstelle besteht aus insgesamt 16 Leitungen, wovon 8 Datenleitungen sind; 3 weitere dienen als Handshake, die restlichen 5 dienen der Schnittstellen-Organisation. Betrachtet man den Stecker (s. Abbildung C.3), so erkennt man weitere 8 Leitungen, welche alle auf Masse geschalten sind. Weiterhin erkennt man, dass die Stecker stapelbar sind, wodurch man mehrere

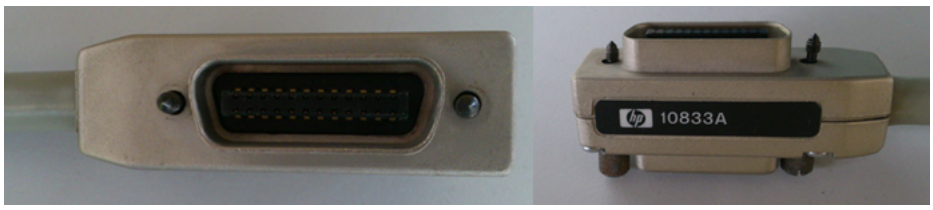


Abbildung (C.3)
Bild des GPIB Steckers 10833A von Hewlett-Packard, links: Draufsicht mit Darstellung der einzelnen Pins, rechts: Seitenansicht mit Darstellung der Möglichkeit die Stecker nacheinander zu schalten.

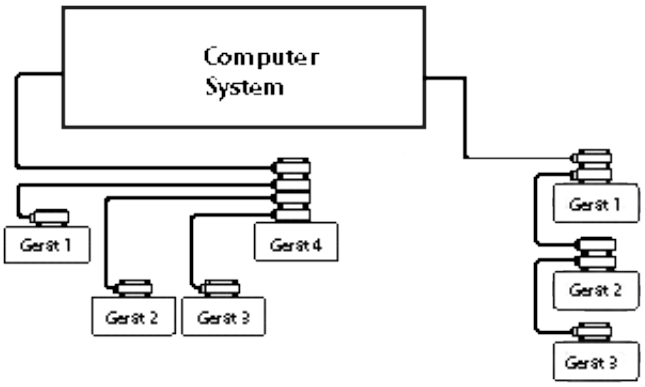


Abbildung (C.4)
Anordnungsmöglichkeiten von GPIB-Geräte entweder in linearer Weise (links) oder in Sternform (rechts). Auch eine Kombination aus beiden ist möglich.

Geräte nacheinander schalten kann, jedoch maximal 15 [1]. Die Anordnung der Geräte spielt dabei keine Rolle (s. Abbildung C.4)

Die 8 Datenleitungen (s. Tabelle C.2) transportieren sowohl die geräteabhängigen wie auch die Schnittstellen-Nachrichten. Die ATN-Leitung entscheidet dabei über den Typ der gesendeten Nachricht. Alle Schnittstellen-Nachricht wie auch die meisten geräteabhängigen Nachrichten verwenden eine 7-bit ASCII oder ISO Zeichenkodierung, wobei die Leitung DIO8 entweder nicht benötigt wird oder als Paritätsbit Verwendung findet. Die 3 Handshake-Leitungen werden als asynchrone Kontrollleitungen verwendet,

[1] Auch wenn die Schnittstelle rein formal 30 Geräte unterstützen würde

Pin/Buchse	Bedeutung	Kategorie
1 - 4	DIO1-4	DT
5	EOI (End or Identify)	SOL
6	DAV (Data Valid)	HL
7	NRFD (Not Ready for Data)	HL
8	NDAC (Not Data Accepted)	HL
9	IFC (Interface Clear)	SOL
10	SRQ (Service Request)	SOL
11	ATN (Attention)	SOL
12	GND (Ground)	Masse
13 - 16	DIO5-8	DT
17	REN (Remote Enable)	SOL
18 - 24	GND (Ground)	Masse

Tabelle (C.2)
Pinbelegung von GPIB-Steckern. Es wird dargestellt, welcher Pin welche Funktionalität zur Verfügung stellt und welcher Kategorie er zugeordnet ist. Die Kategorien sind wie folgt: DT entspricht dem Pin für eine Datenleitung, SOL dem für eine Schnittstellen-Organisations Leitung und letztendlich HL für eine Handshake-Leitung. GND ist ein Pin, der auf Masse gelegt wird.

um zu garantieren, dass Nachrichten ohne Übertragungsfehler gesendet oder empfangen werden. Die 5 Schnittstellen-Organisations Leitungen haben speziellere Bedeutungen, welche hier nicht erklärt werden. Dafür sei auf einschlägige Literatur bzw. die Definition der Schnittstelle verwiesen[IEE03, IEE04].

C.2. Software

C.2.1. ASCII und UNICODE

Entsprechend werden durch den Ersatz von ASCII durch UNICODE natürlich auch die Datensätze größer, da sich entsprechend die Anzahl an Bytes pro Zeichen abhängig vom Datensatz vergrößern. Vorteile von Text-Dokumenten sind natürlich die Lesbarkeit. Man benötigt ausschließlich eine Software, welche Zeichensatztabellen auf die zugrunde liegenden Daten anwendet. Jedes Betriebssystem bietet dafür seine eigenen Editoren, die

mehr oder weniger zusätzliche Funktionalitäten bieten. Dadurch, dass Text-Dokumente „lesbar“ sind, ergibt sich auch die Möglichkeit der Indizierung, wodurch ein Durchsuchen erleichtert wird. Auf diese Weise können Dokumente anhand ihrer Datenbestände auf einem Datenträger einfach gefunden werden. Nachteile sind die bereits erwähnte Größe der Dateien. Um Zeichen wie Buchstaben des Alphabets darstellen ist es durchaus sinnvoll, die Datensätze direkt zu interpretieren. Enthalten Dateien allerdings Zahlen oder Zahlenlisten, so werden beträchtliche Ressourcen benötigt, um diese Zahlen zu kodieren. Betrachtet man beispielsweise die Zahl 128, so benötigt man bei binärer Datenspeicherung dafür 1 Byte = 8 Bit, d.h. die Zahl $2^7 d = 01000000b = 80h$ (Die Abkürzungen d, h und b stehen für dezimale, hexadezimale und binäre Schreibweise). Speichert man die Zahl in ASCII, so benötigt man dafür 3 Byte nämlich jeweils eines zur Kodierung der einzelnen Zeichen „1“,“2“ und „8“, d.h. $31h32h38h$.

C.2.2. TDMS

Messdaten bestehen zwar im Großen und Ganzen aus Zahlen, allerdings werden meiste zusätzliche Informationen gespeichert, welche die Daten spezifizieren. Für Messwerte ist beispielsweise nicht nur der absolute Wert interessant, sondern auch, wann und von wem dieser Wert aufgenommen wurde, mit welcher Methode er aufgenommen worden ist und in welcher Einheit der Wert vorliegt. Zusätzlich sind Benutzerkommentare hilfreich, um zu verstehen, in welchem Zusammenhang das erfolgt ist. Diese Attribute oder Metadaten müssten in reinen Binärdateien auch binär hinterlegt werden und umständlich von einer Anwendung interpretiert werden. Besonders Sequenzen mit variabler Länge wie Kommentare sind sehr schwer zu interpretieren. Das TDM-Dateiformat ist deshalb als eine gemischte Datei aus Text-Abschnitten bestehend aus Schlüssel-Wert Paaren und Kanälen, in welchen Daten binär gespeichert werden, entwickelt worden. Darüber hinaus sind die Daten in Tabellen organisiert, wobei jede Datei, jede Tabelle und jeder Kanal einer jeden Tabelle mit eigenen Text-Abschnitten versehen werden können. TDM Dateien verbinden aus diesem Grund die Vorteile von Text- und Binär-Dateien, außerdem von Datenbank-Tabellen (s. Abbildung C.5), d.h. sie können einfach durchsucht werden, sind lesbar und können sehr performant und speichereffizient mit Daten beschrieben werden.

	ASCII	Binary	XML	Database	TDMS
Exchangeable	✓		✓		✓
Small Disk Footprint		✓			✓
Searchable				✓	✓
Inherent Attributes			✓		✓
High-Speed Streaming		✓			✓
NI Platform Supported	✓	✓	✓	✓*	✓

Abbildung (C.5)
Bild zur Erklärung der Datenspeicherungsoptionen des TDMS Datenformates im Vergleich zu anderen Dateiformaten. Entnommen aus NI-Whitepaper 3727.

C.2.3. Kryptologie

Verschlüsselung

Für die Verschlüsselung existieren verschiedenste Verschlüsselungsalgorithmen und -strategien und sie sind als Teilgebiet der Kryptographie angesiedelt. Dabei wird zwischen symmetrischen und asymmetrischen Verfahren unterschieden. Bei ersterem werden die Daten vom Sender mit Hilfe eines Schlüssels chiffriert, während der Empfänger denselben Schlüssel benötigt, um die Daten wieder zu dechiffrieren (z.B. Passwort). Somit können Computersysteme zwischen den beiden Endpunkten die Daten nicht interpretieren, solange der Chiffrier-Schlüssel für sie unbekannt ist. Dabei ist zu bedenken, dass die Schlüssel im Vorfeld zwischen den beiden Kommunikationspartnern ausgetauscht werden müssen (was manuell erfolgen muss). Beim asymmetrischen Verfahren muss im Vorfeld ein sog. Schlüsselpaar erzeugt werden, bestehend aus einem öffentlichen und einem privaten Schlüssel. Der öffentliche Schlüssel wird anschließend dazu verwendet, die zu sendenden Daten zu chiffrieren, wobei dieser Schlüssel nicht mehr in der Lage ist, die Daten zu dechiffrieren. Dies vermag ausschließlich der private Schlüssel, der auf dem Zielsystem liegt und dieses nicht verlassen sollte. Vorteil dieser Methode ist, dass der öffentliche Schlüssel über eine unverschlüsselte Leitung (manuell oder automatisiert) verteilt werden kann. Solange der private Schlüssel geheim gehalten wird, kann der Datenbestand als vertraulich und ungelesen angesehen werden.

Signierung

Dabei ist allerdings die Authentizität des Senders noch nicht gelöst, d.h. jeder kann diese Daten rein prinzipiell gesendet haben, wenn er entweder den Chiffrierschlüssel oder den öffentlichen Schlüssel besitzt (was einfach möglich ist, da der öffentliche Schlüssel meist in öffentlichen Verzeichnissen zum Abruf zur Verfügung steht). Im symmetrischen Fall kann allein über den Schlüssel zwar die Vertraulichkeit der Daten gesichert werden, allerdings nicht die Authentizität. Dies kann über die Kombination aus einer Identifikation (z.B. Benutzername) und dem Schlüssel gewährleistet werden, welcher nur dem Besitzer der Identifikation bekannt sein darf; diese Kombination kann in einer Datenbank hinterlegt werden. Auch im asymmetrischen Fall ist Authentizität nicht ohne ein weiteres Merkmal möglich, auch wenn es direkter zugänglich ist. Um die Authentizität und auch Integrität der Daten zu gewährleisten wird aus den Daten ein sog. Hashwert gebildet und mit dem privaten Schlüssel „signiert“. Auch der Sender benötigt hierzu ein Schlüsselpaar, von welchem der Empfänger den öffentlichen Schlüssel besitzt. Der Empfänger kann anschließend mit Hilfe des öffentlichen Schlüssels die Signatur überprüfen und somit gewährleisten, dass die Daten unverändert sind und vom gewünschten Empfänger stammen. Der Vorgang ist dementsprechend umgekehrt zur Verschlüsselung, bei welcher der der öffentliche Schlüssel des Empfängers zum Verschlüsseln verwendet wird, während hier der private Schlüssel des Senders zur Signierung angewendet werden muss. Auch in Excalibur wird auf Authentizität, Integrität und Vertraulichkeit geachtet. Zur Datenkommunikation wird dabei auf ein zertifikatbasiertes System zurückgegriffen, um Vertraulichkeit zu gewährleisten (Server Zertifikate). Mit Hilfe von sowohl symmetrischen (Benutzername und Passwort) als auch mit einem zertifikatbasierten Verfahren (Client Zertifikate) soll die Authentizität und Integrität gewährleistet werden. Mehr zum Thema Kryptologie kann in entsprechender Fachliteratur der Informatik nachgelesen werden.[Eck09, BSW01]

Literaturverzeichnis

[APB+06] Atkins, P.W. ; Paula, J. de ; Bär, M. ; Schleitzer, A. ; Heinisch, C.: *Physikalische Chemie*. Wiley, 2006 https://books.google.de/books?id=rSX3AAAACAAJ. – ISBN 9783527315468

[Bal11] Balzert, Helmut: *Lehrbuch der Software-Technik [2]*. Heidelberg [u.a.] : Spektrum, Akad. Verl., 2011. – ISBN 9783827417060 3827417066

[BM04] Biron, P ; Malhotra, A: *XML Schema Part 2: Datatypes Second Edition*. Version: 2004. https://www.w3.org/TR/xmlschema-2/

[BPSM+06] Bray, T ; Paoli, J ; Sperberg-McQueen, C. M. ; Maler, E. ; Yergeau, F.: *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. 2006 https://www.w3.org/TR/2006/REC-xml-20060816

[Bro87] Brockhaus: *Neunzehnte völlig überarbeitete Auflage*. Bd. 19: *Brockhaus. Enzyklopädie in vierundzwanzig Bänden*. Mannheim, 1987

[BSW01] Beutelspacher, Albrecht ; Schwenk, Jörg ; Wolfenstetter, Klaus-Dieter: *Moderne Verfahren der Kryptographie : von RSA zu Zero knowledge*. Braunschweig; Wiesbaden : Vieweg, 2001. – ISBN 3528365900 9783528365905

[Cam01] Cammann, Karl: Instrumentelle Analytische Chemie. In: *Spektrum-Verlag Heidelberg* (2001)

[DUD06] *Der Duden. Das Standardwerk zur deutschen Sprache*. 24. Mannheim : Bibliographisches Institut, 2006

[Eck09] Eckert, Claudia: *IT-Sicherheit Konzepte - Verfahren - Protokolle*. München : Oldenbourg, 2009. – ISBN 9783486589993 3486589997

[EDW14] *Enzyklopädie der Wirtschaftsinformatik*. Version: 2014. http://www.enzyklopaedie-der-wirtschaftsinformatik.de/

[EIJ14] *Einführung in JSON*. Ecma International, 2014 http://http://json.org/json-de.html

[EK:14] *Serielle Schnittstelle (RS232 / V.24 / COM).* Version: 1997-2014. `http://www.elektronik-kompendium.de/sites/com/0310301.htm`

[FIG99] Fielding, R ; Irvine, UC ; Gettys, J: *Hypertext Transfer Protocol – HTTP/1.1.* Version: 1999. `https://tools.ietf.org/html/rfc2616`

[FIP14] *Fraunhofer Institut für Produktionstechnik und Automatisierung.* Version: 2014. `http://www.bioproduktion.com/`

[Fre96a] Freed, N.: *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.* Version: 1996. `https://tools.ietf.org/html/rfc2045`

[Fre96b] Freed, N.: *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types.* Version: 1996. `https://tools.ietf.org/html/rfc2046`

[Fri06] Frielingsdorf, Herbert: *Einfache IT-Systeme.* Troisdorf : Bildungsverl. Eins, 2006. – ISBN 3823711407 9783823711407

[GHJV98] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design patterns : elements of reusable object-oriented software.* 20. printing. Reading, Mass. : Addison-Wesley, 1998 (Addison-Wesley professional computing series). – xv, 395 sider S. – ISBN 0201633612 (hardback). – kr. 412,38 Erich Gamma ... [et al.]. ill. Bibliography: side 375-381.

[GMNR05] Gudgin, Martin ; Mendelsohn, Noah ; Nottingham, Mark ; Ruellan, Hervé: *XML-binary Optimized Packaging.* Version: 2005. `http://www.w3.org/TR/xop10/`

[Gra98] Grand, Mark: *Patterns in Java, volume 1: a catalog of reusable design patterns illustrated with UML.* John Wiley
amp; Sons, Inc., 1998. – 467 S. – ISBN 0–471–25839–3

[GZ94] Göpel, Wolfgang ; Ziegler, Christiane: *Struktur der Materie.* Stuttgart : Teubner, 1994 (Grundlagen, Mikroskopie und Spektroskopie). – 668 S. `http://www.bsz-bw.de/cgi-bin/ekz.cgi?SWB03879081http://www.gbv.de/dms/ilmenau/toc/147829372.PDF`. – ISBN 3–8154–2110–1

[Hec99] Hecht, Eugene: *Optik.* 2., durchges. Aufl. München ; Wien : Oldenbourg, 1999. – x, 717 s. S. – ISBN 3486251864

[HU94] Hopcroft, John E. ; Ullman, Jeffrey D.: *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Bonn; Paris [u.a.] : Addison-Wesley, 1994. – ISBN 3893197443 9783893197446

[IEE03] IEEE Standard for Higher Performance Protocol for the Standard Digital Interface for Programmable Instrumentation. In: *IEEE Std 488.1-2003 (Revision of IEEE Std 488.1-1987)* (2003), S. 1–143. http://dx.doi.org/10.1109/IEEESTD.2003.94413. – DOI 10.1109/IEEESTD.2003.94413

[IEE04] Standard Digital Interface for Programmable Instrumentation - Part 2: Codes, Formats, Protocols and Common Commands (Adoption of (IEEE Std 488.2-1992). In: *IEC 60488-2 First edition 2004-05; IEEE 488.2* (2004), S. 1–5. http://dx.doi.org/10.1109/IEEESTD.2004.95390. – DOI 10.1109/IEEESTD.2004.95390

[Ins13] Instruments, National: *NI-TDMS-Dateiformat*. Version: 2013. http://www.ni.com/white-paper/3727/de/pdf

[JS:99] *ECM99 ECMA Standard 262: ECMAScript Language Specification*. 3. Ecma International, 1999 http://www.ecma-international.org/publications/standards/Ecma-262.htm

[JSO13] *Standard ECMA-404: The JSON Data Interchange Format*. Version: 2013. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[Kai97] Kainka, Burkhard: *Messen, Steuern, Regeln über die RS-232-Schnittstelle : Messdatenerfassung und Prozesssteuerung mit dem PC; mit 20 Tabellen*. München : Franzis, 1997. – ISBN 3772360580 9783772360589

[KC93] Klensin, J. ; Chair, WG: *SMTP Service Extension for 8bit-MIMEtransport*. Version: 1993. https://tools.ietf.org/html/rfc6152

[KR83] Kernighan, Brian W. ; Ritchie, Dennis M.: *Programmieren in C : mit dem Reference-Manual in deutscher Sprache*. Mu¨nchen; Wien : Hanser, 1983. – ISBN 3446138781 9783446138780

[Lev98] Levinson, E.: *The MIME Multipart/Related Content-type*. Version: 1998. https://tools.ietf.org/html/rfc2387

[LIM14] *Das Portal für Laborinformations- und Management-Systeme.*

Version: 2014. http://www.lims.de

[Moo96] MOORE, K.: *Multipurpose Internet Mail Extensions (MIME) Part Three: Message Header Extensions for Non-ASCII Text.* Version: 1996. https://tools.ietf.org/html/rfc2047

[nee14] NEED.DE i: Software Steuerungen. In: *SPS Magazin* 7 (2014), S. 43–47

[Red06] REDEMANN, Bernhard: *Steuern und Messen mit USB : Hard- und Softwareentwicklung mit dem FT232BL, FT245BL und FT2232L ; [VCP- und D2XX-Treiber, viele Beispielprogramme, I2C- und Web-Projekt, Grundlagen USB].* [Berlin, Albertinenstr. 20] : B. Redemann, 2006. – ISBN 3000178848 9783000178849

[Res00] RESCORLA, Eric: *SSL and TLS : building and designing secure systems.* Harlow : Addison-Wesley, 2000. – ISBN 0201615983 9780201615982

[RRD07] RICHARDSON, Leonard ; RUBY, Sam ; DEMMIG, Thomas: *Web-Services mit REST : [frischer Wind für Web-Services durch REST].* Beijing; Cambridge; Farnham; Köln; Paris; Sebastopol; Taipei; Tokyo : O'Reilly, 2007. – ISBN 9783897217270 3897217279

[Tan09] TANENBAUM, Andrew S.: *Moderne Betriebssysteme.* München; Boston [u.a. : Pearson Studium, 2009. – ISBN 9783827373427 3827373425

[UL13] *Universal Lexikon.* Version: 2000-2013. http://universal_lexikon.deacademic.com

[Vos08] VOSSEN, G.: *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme.* Oldenbourg, 2008 http://books.google.de/books?id=_ZODTXSJHfwC. – ISBN 9783486275742

[Vos12] VOSSEN, Gottfried: *Datenbanksystem.* Version: 2012. http://www.enzyklopaedie-der-wirtschaftsinformatik.de/

[WF12] WEDLER, G. ; FREUND, H.J.: *Lehrbuch der Physikalischen Chemie.* Wiley-VCH, 2012 https://books.google.de/books?id=2p8tBAAAQBAJ. – ISBN 9783527329090