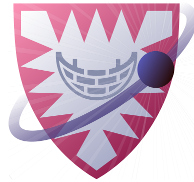
Christian-Albrechts-Universität zu Kiel

Bachelorarbeit

Synthese von Datenflussdiagrammen aus annotierten C-Programmen

Sven Gundlach

29. März 2012



Institut für Informatik
Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden

betreut durch:

Dipl.-Inf. Miro Spönemann

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel, 29. März 2012

____

Zusammenfassung

Datenflussdiagramme sind ein Mittel zur Modellierung von Daten verarbeitenden Prozessen, wie sie zum Beispiel in der Signalverarbeitung vorkommen. Ziel dieser Arbeit ist es, ein Verfahren zu entwickeln, das mit einem gut überschaubaren technologischen Aufwand Datenflussdiagramme aus Annotationen in C-Quellcode erzeugen kann. Dazu werden gängige Dokumentationswerkzeuge genutzt, um C-Quellcode in Datenstrukturen zu überführen, die mit Parsern einfach weiterverarbeitet werden können. Anschließend wird ein Vorschlag zum Aufbau von Annotationen gemacht, aus denen ein Datenflussdiagramm hierarchisch definiert werden kann.

Inhaltsverzeichnis

1. Einführung 1

1.1. Aufgabenstellung 1
1.2. Aufbau dieser Arbeit 2

2. Grundlagen 5

2.1. KiRAT 5
2.2. Softwarearchitektur-Rekonstruktion 7
2.3. Datenflussdiagramme 8
2.4. Dokumentationswerkzeuge 11
2.5. Parser 12

3. Verwendete Technologien 17

3.1. Doxygen 17
3.2. XPath 19
3.3. pugixml 21
3.4. KIELER 21
3.5. KAOM 22

4. Implementierung 25

4.1. Aufbau der Annotationen 25
4.2. Programmstruktur 27
4.3. Erstellen der XML-Dokumente 28
4.4. Filtern der Annotationen 29
4.5. Erstellen der KAOM-Modelle 31

5. Zusammenfassung und Ausblick 35

A. Bedienungsanleitung 47

B. C++-Code für C2KAOM 51

B.1. Beschreibung 51
B.2. Der Code 54

Verzeichnis der Akronyme

BSD	Berkeley Software Distribution
C++/CLI	C++ Common Language Infrastructure
C1X	C standard revision
C2KAOM	C to kaom
CAU	Christian-Albrechts-Universität
CHM	Microsoft Compiled HTML Help
DOM	Document Object Model
DSS	Gruppe für Digitale Signalverarbeitung und Systemtheorie
DTD	Document Type Definition
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
Gnu LGPL	GNU Lesser General Public License
GPL	GNU General Public License
GUI	Graphical user interface
HTML	HyperText Markup Language
ICC	Inter-car communications
ID	Identification
IDL	Interface Description Language
JS	JavaScript
KAOM	KIELER Actor Oriented Modeling
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KiRAT	Kiel Realtime Audio Toolkit
MIT	Massachusetts Institute of Technology
MSXML	Microsoft XML Core Services
PDF	Portable Document Format
PL/SQL	Procedural Language/Structured Query Language
PHP	PHP: Hypertext Preprocessor

RTF Rich Text Format
SAX Simple API for XML
TMF Textual Modeling Framework
VB Visual Basic
VBScript Visual Basic Scripting Edition
W3C World Wide Web Consortium
XLST Extensible Stylesheet Language Transformations
XPath XML Path Language
XSL Extensible Stylesheet Language

1. Einführung

Programme mit messtechnischen Aufgaben und der Verarbeitung messtechnischer Ergebnisse sind meistens ein Ablauf, in dem Ausgaben einer Stelle an anderer Stelle als Eingabe verwendet und weiterverarbeitet werden. Beispielsweise könnte bei einer Sprachkonferenzsoftware eine Komponente die akustischen Signale digitalisieren. Eine weitere Komponente könnte die digitalen Signale filtern und manipulieren. Eine dritte Komponente könnte für die Übertragung an andere Teilnehmer zuständig sein. Und wieder eine andere Komponente könnte für die akustische Wiedergabe der digitalen Signale vorgesehen sein.

Die Verarbeitung von Daten gibt in einem solchen Programm einen guten Überblick über den Ablauf des Programms. Um die Verarbeitung von Daten in einem Prozessablauf dem Entwickler sichtbar zu machen, können *Datenflussdiagramme* genutzt werden. In Datenflussdiagrammen werden die Algorithmen durch einfache Verzweigungen und Schleifen der Signalflüsse zwischen den einzelnen Aktoren des Algorithmus dargestellt. Das Datenflussdiagramm zeigt so, welche Komponenten im Algorithmus miteinander kommunizieren und welche Informationen sie dabei senden. Es ist so möglich, eine anschauliche Übersicht der Algorithmen zu geben. Diese Übersicht entspricht der gewohnten Darstellung von Prozessen in der Signalverarbeitung.

Dokumentationswerkzeuge bieten Entwicklern die Möglichkeit, ausführliche Dokumentationen durch das Hinzufügen von Annotationen und Kommentaren zum Quellcode automatisch zu erzeugen. Gängige Dokumentationswerkzeuge bieten allerdings keine direkte Unterstützung, um Datenflussdiagramme automatisch aus dem Quellcode zu erstellen. Dokumentationswerkzeuge können aber dazu genutzt werden, um den Quellcode zur Weiterverarbeitung in andere Datenstrukturen zu überführen. Aus solchen Datenstrukturen können dann, mit entsprechenden darin enthaltenen Informationen, auf einfachere Weise Datenflussdiagramme erstellt werden.

1.1. Aufgabenstellung

Die Ausgangshypothese ist, dass das Erstellen von Datenflussdiagrammen aus Annotationen im Quellcode durch die Verwendung vorhandener Technologien stark vereinfacht werden kann. Dazu wird das Problem in Teilprobleme unterteilt, um durch die Kombination verschiedener ausgereifter Technologien zu einer überschaubaren Lösung des Problems zu kommen. Die Teilprobleme ergeben sich aus den einzelnen Phasen zur halbautomatischen Erstellung eines Datenflussdiagrammes. Als Programmiersprache vom Quellcode wird C oder C++ vorausgesetzt.

Als Teilprobleme wurden folgende identifiziert:

1. Das Entwickeln von Annotationen, aus denen ein Datenflussdiagramm erstellt werden kann.
2. Das Einlesen des Quellcodes mit der Möglichkeit, die Annotationen im Quellcode zu erkennen. Dafür können Dokumentationswerkzeuge mit Mechanismen, die ein individuelles Anpassen der Dokumentation möglich machen, eingesetzt werden.
3. Das Erkennen der Annotationen und Aktoren in der Dokumentation. Dazu muss die Dokumentation automatisch verarbeitbar sein. Es kann vorausgesetzt werden, dass die Syntax der Annotationen korrekt ist.
4. Das Filtern der zur Darstellung des Datenflussdiagrammes erforderlichen Informationen aus den Annotationen und das Überführen in ein Format zur Darstellung des Datenflussdiagrammes. Als Format zur Darstellung wird *KIELER Actor Oriented Modeling* (KAOM) genutzt und die damit verbundenen Funktion vom KIELER Projekt. Dazu muss ein Softwarewerkzeug implementiert werden, das die Annotationen auswerten und ins KAOM Format schreiben kann.

Diese Arbeit ist explorativ. Sie konzentriert sich darauf, mit einem überschaubaren technologischen Aufwand ein Datenflussdiagramm halbautomatisch zu erstellen. Dabei werden nur die grundlegenden funktionalen Aspekte eines Datenflussdiagrammes, also Datenfluss, Funktionen und deren Eingänge und Ausgänge berücksichtigt. Speziellere Ausprägungen werden ausgelassen.

Das entwickelte Softwarewerkzeug *C to kaom* (C2KAOM) ist ein Prototyp und bietet daher nur einen eingeschränkten Umfang. Die Visualisierung, die grafische Sicht und das Layout der KAOM Modelle wird durch KIELER ausgeführt. Auch wird die Fehlerbehandlung des Softwarewerkzeugs nicht alle Konventionen, die in Annotationen vorausgesetzt werden, überprüfen.

Der Einsatz für andere Quellsprachen als C oder C++ ist unter Umständen möglich, aber nicht gewährleistet.

1.2. Aufbau dieser Arbeit

In Kapitel 2 werden Grundlagen, die zum Verständnis der Arbeit notwendig sind, erklärt bzw. vorgestellt. Dabei wird auf die der Implementierung vorangegangene Recherche eingegangen und diese auch vorgestellt.

Kapitel 3 befasst sich mit den in der Arbeit verwendeten und bereits vorhandenen Technologien. Die Technologien werden in der Reihenfolge behandelt, in der sie auch später im Programmablauf vorkommen.

Das Kapitel 4 befasst sich mit dem Aufbau der Annotationen und der eigentlichen Implementierung des prototypischen Softwarewerkzeugs zur Erstellung der Datenflussdiagramme. Es werden die Struktur der Annotationen und des Softwarewerkzeugs vorgestellt. Danach wird auf die einzelnen Phasen im Programmablauf eingegangen.

Zum Schluss werden in Kapitel 5 die gewonnen Erkenntnisse zusammengefasst und es wird auf mögliche Erweiterungen des Softwarewerkzeugs hingewiesen, die in zukünftigen Arbeiten behandelt werden können.

2. Grundlagen

In diesem Kapitel werden Grundlagen zu den mit der Arbeit verbundenen Themen gegeben. Es wird zuerst KiRAT vorgestellt, für das das prototypische Softwarewerkzeug C2KAOM entwickelt wurde.

Die Synthese der Datenflussdiagramme stellt faktisch eine Rekonstruktion der Programmarchitektur mit Datenfluss-Sicht dar. Daher wird zuerst ein kurzer Überblick über die Softwarearchitektur-Rekonstruktion gegeben.

Danach wird der Nutzen und Aufbau von Datenflussdiagrammen beschrieben. Diese sind in eingeschränkter Weise später die Ausgabe von C2KAOM.

Zur Erzeugung von Datenflussdiagrammen werden hier Dokumentationswerkzeuge genutzt. Daher wird anschließend die Funktionsweise von Dokumentationswerkzeugen beschrieben und eine Auswahl von hierfür nutzbaren Programmen sowie deren Vorteile und Nachteile angegeben.

Der letzte Abschnitt befasst sich mit Parsern. Es werden im speziellen XML-Parser behandelt, sowie verschiedene Entscheidungskriterien, die für oder gegen bestimmte Arten von Parsern sprechen, aufgeführt, sowie auf die jeweiligen Vor- oder Nachteile eingegangen. Auch hier wird eine Auswahl von Programmen sowie deren Vor- und Nachteile angegeben.

2.1. KiRAT

Das *Kiel Realtime Audio Toolkit* (KiRAT) ist ein Softwarewerkzeug bzw. Programmiergerüst, das von der Gruppe *Digitale Signalverarbeitung und Systemtheorie* (DSS) an der Technischen Fakultät der Christian-Albrechts-Universität (CAU) entwickelt wird. Es ist die Basis für Audio- und Sprachforschung am Institut.

Eine Übersicht über die Struktur von KiRAT gibt die Abbildung 2.1. KiRAT wird zur Entwicklung von Software für Echtzeitanwendungen im Audio- und Sprachbereich eingesetzt und stellt dafür eine Reihe von Algorithmen/Modulen bereit. Allgemein kann KiRAT für die Softwareentwicklung zu Themen in der digitalen Signalverarbeitung genutzt werden, da das Programm auf die Bedürfnisse der Signalverarbeitung zugeschnitten ist.

In KiRAT wird zur Parametrisierung und Visualisierung der Ausgaben eine grafische Schnittstelle (GUI) bereitgestellt. Diese wurde mit dem Qt Framework implementiert und ist dadurch plattformübergreifend nutzbar.

Das Programm dient als Grundlage für spezielle Anwendungen im Echtzeitbereich. Dazu werden die Möglichkeiten von KiRAT mit Erweiterungen vergrößert. So unterstützt KiRAT

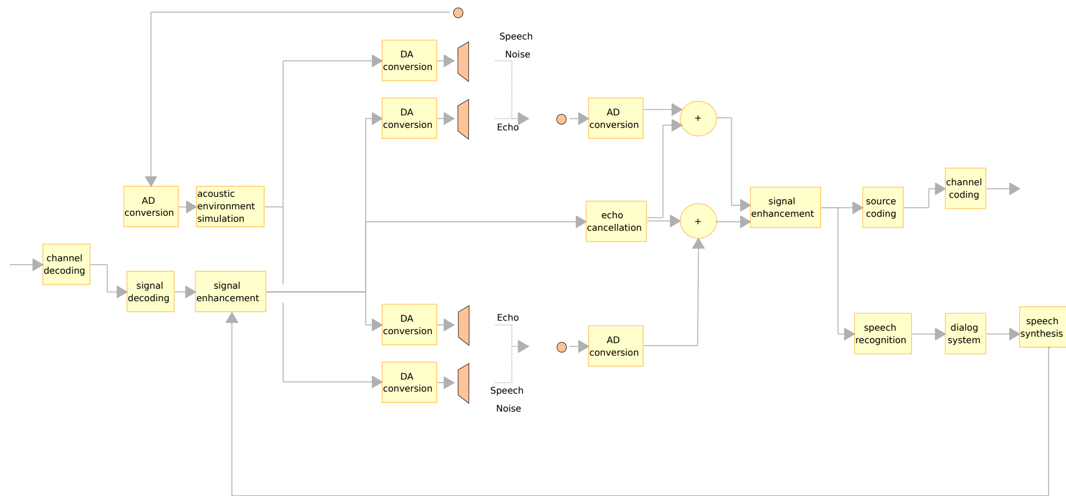


Abbildung 2.1.: Programmaufbau von KiRAT

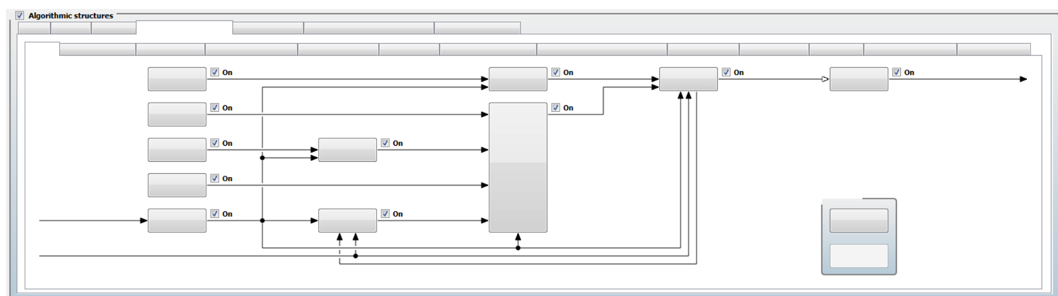


Abbildung 2.2.: Eine Ansicht der KiRAT-GUI

z.B. Fahrzeuginnenraumkommunikation (ICC), automatische Evaluierung von ICC-Systemen, akustische Umgebungssimulation, Lautsprecherentzerrung, automatische Mikrofonkalibrierung und Anbindung von Software an Hardware, wie z.B. verschiedene AD/DA Wandler oder Modems, aber auch Roboterarme.

Bei der Anwendung in der Fahrzeuginnenraumkommunikation werden mit KiRAT Algorithmen entwickelt, die Gesprochenes über Mikrofone aufnehmen, Störgeräusche heraus filtern, analysieren und dann das Gesprochene über Lautsprecher möglichst verständlich wiedergeben.

2.2. Softwarearchitektur-Rekonstruktion

Die von C2KAOM erstellten Datenflussdiagramme sind faktisch eine Darstellung der Programmarchitektur mit Datenfluss-Sicht. Im Allgemeinen muss man eine Rekonstruktion der Programmarchitektur durchführen, um Datenflussdiagramme aus dem Quellcode zu erstellen.

Die Softwarearchitektur-Rekonstruktion besteht grundsätzlich aus vier Schritten.

Zuerst werden die grundsätzlichen Architektur-Konzepte des Systems bestimmt. Dabei wird auch ermittelt, wie die architektonischen Konzepte zum Code zugeordnet werden können. Auf den in diesem Schritt ermittelten Konzepten baut die weitere Softwarearchitektur-Rekonstruktion auf.

Im zweiten Schritt werden die Architektur-Konzepte mit allen relevanten Informationen gefüllt, die zur Beschreibung der Programmarchitektur notwendig sind. Die richtige Auswahl der Architektur-Konzepte ist entscheidend dafür, ob dieser Schritt erfolgreich ist. Da das Erfassen von relevanten Informationen hauptsächlich Datenerfassung im Code ist, kann es gut automatisiert werden. Dazu werden dynamische und statische Analysen durchgeführt. Diese können eine Simulation des Programms oder auch eine Analyse des Quellcodes sein. Für bestimmte Informationen wie z.B. Datenflüsse kann es jedoch aufgrund bestimmter Datenstrukturen wie Zeigern sehr aufwendig werden, relevante Informationen automatisch zu extrahieren.

Der dritte Schritt ist die Abstraktion der Informationen, die im vorherigen Schritt ermittelt wurden. Diese Informationen haben meistens ein sehr niedriges Abstraktionsniveau und eignen sich nicht direkt für Modellbildungen. Durch die Abstraktion werden die Informationen aufbereitet und für die Darstellung der architektonischen Konzepte nutzbar gemacht.

Der letzte Schritt ist die Darstellung der gewonnen Informationen. Dazu muss eine passende Visualisierung gewählt und präsentiert werden [8].

Alle vier Schritte müssten für die Synthese von Datenflussdiagrammen durchgeführt werden. Man kann jedoch relevante Informationen, wie sie im dritten Schritt ermittelt würden, bereits in den Quellcode schreiben, Beispielsweise durch Annotationen in den Kommentaren. Dadurch werden die einzelnen Schritte wesentlich vereinfacht und eine einfachere Lösung möglich.

2.3. Datenflussdiagramme

Datenflussdiagramme stellen eine spezielle Sicht auf einen Prozessentwurf dar. Sie geben dabei den Datenfluss eines Prozesses oder allgemein einer Tätigkeit wieder. Den Prozess bilden sie durch die Kommunikation zwischen den beteiligten Aktoren ab. Ein *Aktor* ist ein Element, das Eingaben in andere Ausgaben transformiert und eventuell dabei Seiteneffekte hervorruft. Datenflussdiagramme sind eine graphische Darstellung von Aktor-orientierten Modellen. Sie abstrahieren aber vollständig vom Kontrollfluss.

Aktor-orientierte Modelle sind eine effektive Methode für Software-System-Entwürfe. Unter anderem modellieren sie die Kommunikation zwischen Aktoren aufgrund von Daten-Pipelines. Aktoren haben dabei eine fest definierte Schnittstelle, mit der ihr internes und externes Verhalten abstrakt spezifiziert wird. Für das externe Verhalten werden sogenannte Ports definiert, über die ein Aktor mit seiner Umgebung kommuniziert. Für das interne Verhalten gibt die Schnittstelle Parameter vor, mit denen die Funktionsweise eines Aktors eingestellt werden kann. Ein Aktor kommuniziert nicht direkt mit anderen Aktoren, sondern er kommuniziert über festgelegte Daten-Pipelines mit der Umgebung. Die Daten-Pipelines werden von einer übergeordneten Instanz vorgegeben.

Aktor-orientierte Modelle sind hierarchisch aufgebaut. Ein Modell kann daher wieder ein Aktor in einem anderen Modell sein und hat daher auch eine feste Schnittstelle. Diese Schnittstelle enthält *externe ports* und Parameter, die von den Schnittstellen der inneren Aktoren unabhängig sind [4].

Die verwendete Notation in Datenflussdiagrammen ist unterschiedlich. Grundsätzlich werden für Datenflussdiagramme aber folgende Elemente verwendet:

Datenfluss: Zwischen anderen Elementen wird der Datenfluss als gerichteter Pfeil von einer Ausgabe zu einer Eingabe dargestellt. Der Datenfluss entspricht einer Pipeline, die Informationen transportiert. Es können je nach Granularität auch mehrere Pfeile für verschiedene Zusammenhänge zwischen den Aktoren angegeben werden. Datenflüsse können sich verzweigen oder wieder vereinigen.

Funktionen: Funktionen werden meistens als Rechtecke dargestellt. Funktionen sind Aktoren, die eingehende Daten transformieren oder neue generieren. Eine Funktion hat einen oder mehrere Ein- und Ausgänge.

Schnittstellen: Schnittstellen werden als Rechtecke dargestellt. Sie haben entweder Eingänge oder Ausgänge und sind spezielle Aktoren, über die Daten in den Prozess einfließen oder aus dem Prozess ausgegeben werden können. Durch sie kann ein Prozess mit seiner Umgebung interagieren. Schnittstellen an denen Daten einfließen heißen *Quellen* und Schnittstellen an denen Daten ausgegeben werden heißen *Senken*. Wird der Prozess als Aktor in einem anderen Datenflussdiagramm verwendet, dann können Schnittstellen als Quadrate an die Darstellung des Prozesses angehängt werden.

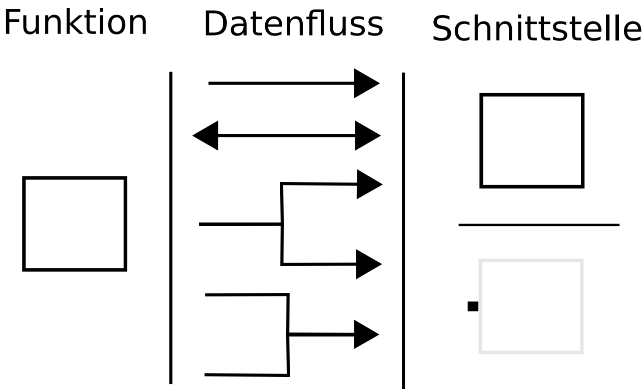


Abbildung 2.3.: Notationen für Datenflussdiagramme in der digitalen Signalverarbeitung

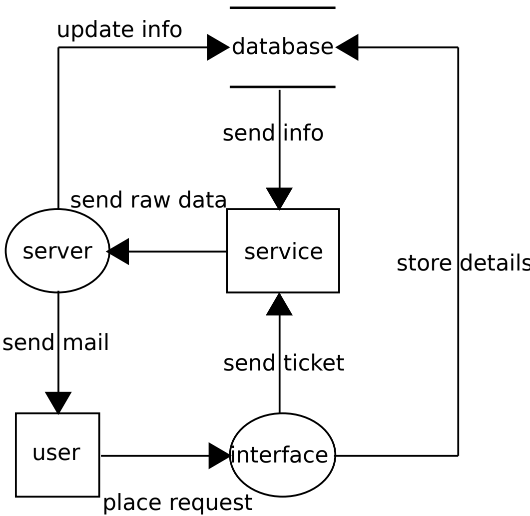


Abbildung 2.4.: Einfaches Datenflussdiagramm aus der strukturierten Analyse

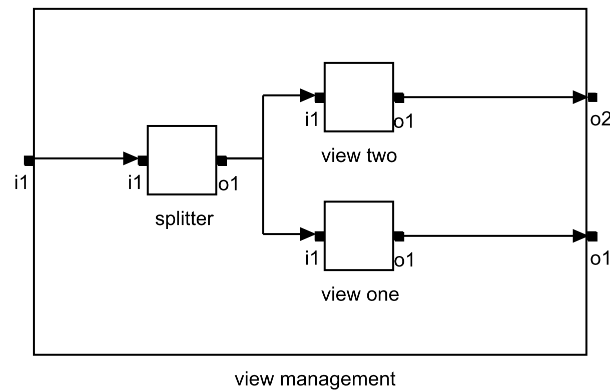


Abbildung 2.5.: Einfaches Datenflussdiagramm mit hierarchischen Aufbau aus der digitalen Signalverarbeitung

Man kann Prozesse in anderer Weise auch konkreter darstellen, indem man Aktoren als beliebige geschlossene Struktur darstellt. So kann die Funktion der Aktoren durch ihre Darstellung beschrieben und das Diagramm übersichtlicher gestalten werden.

Wie auch in Aktor-orientierten Modellen haben diese Funktionen eine feste Schnittstelle bzw. *Interface*, die ihr externes und internes Verhalten bestimmt. Für das externe Verhalten legt die Schnittstelle die Aus- und Eingänge fest. Für das interne Verhalten werden alle weiteren enthaltenen Elemente festgelegt.

Datenflussdiagramme, die ein Prozess vollständig beschreiben, können schnell sehr komplex und unübersichtlich werden. Es ist aber möglich, Datenflussdiagramme als hierarchischen Graph aufzubauen. Ein Aktor eines Datenflussdiagrammes kann wiederum als Datenflussdiagramm in der darunterliegenden Ebene dargestellt werden. Auf diese Weise kann das Datenflussdiagramm eines komplexen Prozesses den Ablauf übersichtlich darstellen.

Bei großen Prozessen ist es daher nützlich, verschiedene Ebenen eines Prozesses einzeln darzustellen und so die Diagramme überschaubar zu halten. Es wird, wie in Aktor-orientierten Modellen, dafür ein Hierarchiekonzept verwendet, das den Prozess schrittweise in Ebenen verfeinert. In automatischen Softwarewerkzeugen für Datenflussdiagramme kann dieses Konzept umgesetzt werden, indem entweder Aktoren als ein komplett neues Diagramm dargestellt oder diese innerhalb des Diagramms expandiert werden.

Beispiele zur Anwendung finden sich im Software Engineering in der strukturierten Analyse oder in der digitalen Signalverarbeitung. Auch hier wird ein Hierarchiekonzept verwendet. Zur strukturierten Analyse werden jedoch weitere Elemente verwendet, mit denen mehr auf die Funktionen der Elemente eingegangen wird. Ein Beispiel dazu zeigt die Abbildung 2.4.

Anwendungsbeispiel in der digitalen Signalverarbeitung: In der digitalen Signalverarbeitung produzieren und konsumieren Aktoren *token*. Diese werden über den Datenfluss

an *ports* anderer Aktoren geleitet. Jeder Aktor hat dabei eine *firing function*, die eingehende *token* auf ausgehende *token* abbildet. Damit *token* produziert werden können, muss eine *firing rule* erfüllt sein. Diese Vorschrift spezifiziert, auf welchen *port* wie viele *token* vorhanden sein müssen, damit neue produziert werden können. Die Aktoren können in den darunterliegenden Ebenen durch weitere Datenflussdiagramme verfeinert werden, bis sich die Aktoren nicht mehr weiter sinnvoll zerteilen lassen. Auf unterster Ebene können Aktoren dann mit anderen Spezifikationen festgelegt werden [5].

2.4. Dokumentationswerkzeuge

Dokumentationen gehören zu den grundlegenden Prozessen von Softwaretechnik. Sie ermöglicht die Wartung, Erweiterung und leichtere Nutzung von Programmen. Deshalb müssen Dokumentationen aktuell, verständlich, komplett und übersichtlich sein. Damit die Dokumentation aktuell ist, sollte sie direkt mit dem Quellcode geschrieben werden.

Dokumentationen im Text-Format der Quellcodes sind aber meistens unübersichtlich und in ihrer Darstellung schwer zu lesen. Gute Dokumentationswerkzeuge können hier in einigen Teilen Abhilfe schaffen. Zum einen erlauben sie die automatische Erzeugung von unterschiedlichen Dokumenten in verschiedenen Formaten (z.B. HTML, CHM, RTF, PDF, XML) aus dem Quellcode. So halten sie die Dokumentation aktuell und konsistent. Zum anderen erlauben sie durch spezielle Schlüsselwörter die erzeugte Dokumentation in ihrer Struktur und Darstellung zu verbessern und damit leichter lesbar zu machen.

Eine Voraussetzung für Dokumentationswerkzeuge ist meisten das Hinzufügen von speziell formatierten Kommentaren in den Quellcode. Diese Kommentare werden meistens durch festgelegte Zeichen eingeleitet und anschließend mit Schlüsselwörtern strukturiert. Ein Dokumentationswerkzeug liest nicht nur diese Kommentare ein, sondern analysiert den gesamten Code mit Vererbungsabhängigkeiten, Parametern, Bezeichnern von Methoden und dem Zusammenhang der einzelnen Quelldateien. Diese Informationen werden mit den hinzugefügten Kommentaren verknüpft und zu einer Dokumentation im Zielformat verarbeitet. Eine weitere notwendige Voraussetzung ist daher, dass das Dokumentationswerkzeug die Quellsprache und das Zielformat unterstützt und selbstverständlich auf dem ausführenden System lauffähig ist.

Für diese Arbeit sind die Quellsprachen C oder C++ und das Zielformat KAOM vorgegeben. Da es aber kein Dokumentationswerkzeug gibt, das das KAOM Format unterstützt, wird stattdessen XML oder HTML als Zwischenformat genommen. Dabei eignet sich XML besonders gut zum Speichern von strukturierten Daten. Als Betriebssystem werden gängige Betriebssysteme wie Unix, Linux oder Windows angenommen.

Eine Auswahl an Dokumentationswerkzeugen ist in Tabelle 2.1 zusehen. Am besten eignet sich demnach Doxygen, da es XML, C und C++ unterstützt und unter verschiedenen Betriebssystemen lauffähig ist. Außerdem können hinzugefügte Kommentare durch eigene Schlüsselwörter erweitert werden, und es lässt sich über eine Konsole ausführen. Zusätzlich steht es unter der *GNU General Public License* (GPL) und ist dadurch frei verfügbar.

Name	Lizenz	Betriebssystem	Quellsprache	Zielformat
Doc-O-Matic	Proprietary	Windows	C/C++/C#, Java, VB/VBScript, Delphi, Pascal, IDL, PHP, JS	HTML, CHM, RTF, PDF, XML
Document! X	Proprietary	Windows	C++/CLI only/ C#, VB/VBScript, IDL, PL/SQL	HTML, CHM
Doxygen	GPL	Linux, Mac OS, Windows, NSD, Unix	C/C++/C#, Java, IDL, Fortran, PHP, Python	HTML, CHM, RTF, PDF, LaTeX, PostScript, man pages, XML
Imagix 4D	Proprietary	Linux, Unix, Windows	C/C++, Java	HTML, RTF
TwinText	Proprietary	Windows	Jede mit Kommentaren	HTML, CHM
Universal Report	Free Education, Proprietary	Windows	Jede mit Kommentaren	HTML, RTF, PDF, LaTeX, PostScript

Tabelle 2.1.: Auswahl an Dokumentationswerkzeugen für diese Arbeit

2.5. Parser

Parser sind Programme zur Umwandlung von Daten aus beliebigen Dokumenten in andere Strukturen.

Allgemein analysieren Parser die Syntax einer Eingabe und transformieren diese zur Weiterverarbeitung in ein anderes Format, das in darüber liegenden Schichten der Anwendung genutzt wird. Parser werden daher in vielen Anwendungen eingesetzt, z.B. werden sie im Compilerbau dazu eingesetzt, die syntaktische Analyse durchzuführen und einen abstrakten Syntaxbaum zu erstellen.

Für XML-Code gibt es XML-Parser. Diese erzeugen einen Zwischencode, der die enthaltenen Informationen (Tags, Attribute, Namen, Werte, Hierarchie) zur Verfügung stellt. Auf diesem Zwischencode können andere Programme über standardisierte Schnittstellen weiterarbeiten. Dafür gibt es eine Reihe von Schnittstellen z.B. DOM, SAX oder SAX2. Außerdem validieren einige XML-Parser die Eingabe, andere prüfen nur, ob die Eingabe der XML-Spezifikation folgt und wohlgeformt ist. SAX2 bezeichnet die aktuellere Version von SAX mit mehr Erweiterungen. Daher lassen sich XML-Parser in vier Varianten unterteilen:

- validierender SAX-Parser
- validierender DOM-Parser
- nicht validierender SAX-Parser

- nicht validierender DOM-Parser

Das *Document Object Model* (DOM): DOM ist ein durch das *World Wide Web Consortium* (W3C) definierter Standard für den lesenden und schreibenden Zugriff auf Daten in HTML- oder XML-Format.

DOM nutzt eine Baumstruktur, durch die das XML-Dokument im Speicher abgelegt wird. Auf diese Baumstruktur kann beliebig zugegriffen werden. Jedes XML-Element wird als Knoten dargestellt. Der Zugriff auf spezielle Knoten erfolgt über Verwandtschaftsbeziehungen der einzelnen Knoten ausgehend vom Wurzelelement, dem *Root-Element* des XML-Dokumentes.

Der Vorteil von DOM ist, dass es so einen vollständigen Zugriff auf das XML-Dokument ermöglicht und durch einen festen Standard die Portabilität von Code zwischen XML-Parsern garantiert ist. Daher eignet es sich besonders für interaktive Anwendungen [9].

Die *Simple API for XML* (SAX): SAX ist ein Industriestandard, der von der XML-Community für den lesenden Zugriff auf Daten im XML-Format entwickelt wurde.

Im Mai 1998 wurde die erste Version veröffentlicht. Nachdem sich abzeichnete, dass verschiedene Anwendungen mehr Information benötigten, wurde im Mai 2000 die aktuelle Version SAX 2.0 (SAX2) veröffentlicht.

SAX liest das XML-Dokument stückweise ein und führt auf den aktuell gelesenen Daten für definierte Ereignisse bestimmte Aktionen aus. Daher ist es im Gegensatz zu DOM ereignisorientiert. Es sind keine Aktionen definiert, um direkt in Daten zu schreiben oder auf zukünftige bzw. vorangegangene Daten zuzugreifen. Es ist aber möglich, eigene Funktionen zu schreiben, die Daten abspeichern und so den späteren auch schreibenden Zugriff möglich machen.

Der Vorteil einer ereignisorientierten Vorgehensweise ist, dass man sofort mit der Analyse des XML-Dokumentes beginnen kann und nicht erst das gesamte Dokument einlesen und strukturieren muss. Das ist besonders bei sehr großen XML-Dokumenten von Vorteil. Hier kann eine vollständige Abbildung im Speicher zu extremen Laufzeiten führen, obwohl eventuell nur ein kleiner Teil der Eingabe überhaupt von Interesse ist. SAX eignet sich daher für Anwendungen, die ein XML-Dokument nur lesen und sofort auf die Eingaben reagieren können [6].

SAX ist daher meistens schneller als DOM, und Nutzer kommen auch schneller zu ersten Ergebnissen. Dafür bietet DOM mehr Möglichkeiten, das Dokument zu manipulieren. Außerdem bietet DOM gegenüber SAX die Möglichkeit, den Datenfluss beliebig und nicht nur von oben nach unten zu lesen. Einige Anwendungsaufgaben lassen sich so wesentlich leichter lösen.

Für beide Schnittstellen gibt es von verschiedenen Herstellern für fast alle gängigen Programmiersprachen teilweise frei verfügbare XML-Parser.

Da in dieser Arbeit C++ verwendet wird und es vorausgesetzt wird, dass die Annotationen korrekt sind, kann ein nicht validierender Parser eingesetzt werden. Durch den Aufbau der

Name	Programmiersprache	Schnittstelle	Lizenz	validierend
Arabica	C++	DOM, SAX2, XPath	BSD style	Ja
libxml++	C++	DOM, SAX	Gnu LGPL	Ja
MSXML	C++, COM interface	DOM, SAX2	Proprietary	Ja
PugiXML	C++	DOM, XPath	MIT License	Nein
RapidXml	C++	DOM	Boost 1. 0, MIT License	Nein
TinyXml	C++	DOM	zlib license	Nein
Xerces	C++	DOM, SAX2	Apache License, Version 2. 0	Ja
XMLParser library from Frank Vanden Berghen	C++	DOM	Aladdin Free Public License	Nein

Tabelle 2.2.: Auswahl an Parsern für diese Arbeit

Annotation ist es außerdem notwendig, auf den Inhalt vorangegangener Tags zuzugreifen bzw. das Dokument mehrfach zu lesen. Daher wird ein DOM-Parser verwendet.

Ein kurze Übersicht einiger XML-Parser ist in Tabelle 2.2 zu sehen.

Im Detail ergibt sich im Zusammenhang mit dieser Arbeit für die einzelnen Parser folgendes:

- Arabica ist selbst kein richtiger Parser sondern verwendet selber andere Parser wie Xerces oder MSXML, bietet aber eine gute Schnittstelle zum Parsen.
- libxml++ bindet libxml2, den Gnome-XML Parser, in C++ ein und nutzt daher sehr stark die GNU Bibliotheken.
- MSXML ist der XML-Parser von Microsoft und nur proprietär nutzbar.
- PugiXML ist ein kleiner und schneller XML-Parser, der zusätzlich als einziger kleiner Parser noch XPath unterstützt.
- RapidXml ist ein sehr schneller und kleiner XML-Parser, der auf unnötige Extras verzichtet.
- TinyXml ist ein einfacher und langsamerer XML-Parser, der zudem mehr Speicher verbraucht.
- Xerces ist einer der größten und umfangreichsten XML-Parser, den es für C++ gibt, aber er verbraucht auch entsprechend viel Ressourcen.
- Der XML-Parser von Frank Vanden Berghen ist ein einfach zu bedienender XML-Parser.

Die *XML Path Language* (XPath) ist keine Schnittstelle sondern eine Abfragesprache für XML mit, der sich Mengen von XML-Elementen einfach bestimmen lassen.

Grundsätzlich können alle aufgeführten XML-Parser verwendet werden. Die validierenden XML-Parser sind aber sehr groß und verursachen einen unnötigen Overhead.

Von den nicht validierenden kleinen XML-Parsern zeichnet sich PugiXML durch die Unterstützung von XPath und RapidXml durch besonders hohe Effizienz beim Parsen aus. Da in dieser Arbeit der Aufwand gering gehalten werden soll, eignet sich am besten PugiXML in Verbindung mit XPath.

3. Verwendete Technologien

In diesem Kapitel wird ein Überblick über die Technologien gegeben, die direkt von C2KAOM zur Synthese der Datenflussdiagramme verwendet oder im Anschluss zur Visualisierung genutzt werden können. Der Überblick konzentriert sich auf eine kurze Zusammenfassung und die genutzte Funktionalität.

3.1. Doxygen

Doxygen ist ein umfangreiches Dokumentationswerkzeug. Wie im Kapitel 2.4 beschrieben, kann Doxygen zur Dokumentation von Quellcode benutzt werden.

Doxygen steht unter der GPL Lizenz und ist auf allen gängigen Betriebssystemen lauffähig. Neben den Programmiersprachen C, C++ und C# unterstützt Doxygen auch Java, IDL, Fortran, PHP und Python. Durch Anpassungen der *scanner* ist es auch möglich, andere Programmiersprachen, die nahe bei C liegen, zu benutzen. Mit Input-Filtern kann Doxygen auch noch für anders aufgebaute Sprachen verwendet werden, wie z.B. Javascript oder Visual Basic.

Doxygen wird über die Kommandokonsole oder eine eigene grafische Oberfläche gestartet. Beide Möglichkeiten bieten dieselben Einstellungen. Getroffene Einstellungen können in einer Konfigurationsdatei abgespeichert und später wieder geladen werden.

Für die Dokumentation von Quellcode analysiert Doxygen einerseits den Quellcode automatisch und nutzt andererseits speziell formatierte Kommentare, die vom Programmierer eingefügt werden. Aber auch ohne diese Kommentare kann Doxygen aus der automatischen Analyse eine Dokumentation mit z.B. Vererbungsbeziehungen, verwendeten Klassen und Namesbereichen erstellen.

Für detailliertere Dokumentation benötigt Doxygen Kommentare in C-Stil-Kommentar-Block. Sie sind entweder im JavaDoc-Stil und beginnen mit einem Sternchen (*) oder im Qt-Stil und beginnen mit einem Ausrufezeichen (!). In beiden Fällen öffnet man zuerst einen C-Kommentar. Alternativ kann man auch einzelne Zeilen auskommentieren. Diese Zeilen haben dann nach dem Öffnen des Kommentars ein Slash oder ein Ausrufezeichen.

Doxygen unterteilt Kommentare in drei Kategorien.

- Die Kurzbeschreibung, meistens ein Einzeiler.
- Die detaillierte Beschreibung, eine ausführlichere Beschreibung der Objekte.
- Für Methoden und Funktionen eine verteilte Beschreibung, die als verteilter Kommentar in den Rumpf geschrieben wird.

```
/**
 ... JavaDoc style ...
 */

/*!
 ... Qt style ...
 */

/// ... brief description ...

///
/// ... detailed description ...
///
```

Auflistung 3.1: Beispielkommentare für Doxygen

Kurze und detaillierte Beschreibungen werden normalerweise vor die Deklaration oder Definition der Objekte geschrieben, können aber auch durch Markierungen an andere Stellen geschrieben werden. Ein Kommentar wird von Doxygen vorwiegend als detaillierte Beschreibung verwendet. Ausnahme sind hier einfach auskommentierte Zeilen. Einzelne Zeilen, die auskommentiert sind, werden als kurze Beschreibung und mehrere auskommentierte Zeilen als detaillierte Beschreibung verwendet.

Die Kommentare passen durch verschiedene Schlüsselwörter die erzeugte Dokumentation an. Alle Schlüsselwörter beginnen mit einem Backslash oder einem @-Zeichen. Eine Übersicht aller Schlüsselwörter gibt es unter der Internetadresse: `http://www.stack.nl/~dimitri/doxygen/commands.html`. Doxygen ermöglicht es zudem, neue Schlüsselwörter über den Eintrag `ALIASES` in der Konfigurationsdatei zu definieren. Die Definition geschieht über bereits vorhandene Schlüsselwörter und kann diese überladen aber nicht überschreiben. Es ist möglich, neu definierte Schlüsselwörter zu benutzen und Argumente zuzuweisen. Eine detaillierte Anleitung dazu gibt es unter der Internetadresse: `http://www.stack.nl/~dimitri/doxygen/custcmd.html`.

Kurze Beschreibungen beginnen mit dem Schlüsselwort *brief*. Alternativ kann in der Konfiguration `JAVADOC_AUTOBRIEF` eingeschaltet werden, wodurch Kommentare im JavaDoc-Stil oder mehrere einfach auskommentierte Zeilen als kurze Beschreibung verwendet werden.

In XML-Ausgaben erzeugt Doxygen für jede Datei ein XML-Dokument und für jede Funktion ein XML-Element, das in weiteren XML-Elementen die entsprechende Dokumentation enthält. Unter anderem werden Name, Parameter, Typ, die detaillierte Beschreibung und die kurze Beschreibung gespeichert. Die kurze Beschreibung wird als ein XML-Element ohne weitere XML-Elemente gespeichert und lässt sich somit leicht auslesen. ,

3.2. XPath

Eine häufige Aufgabenstellung für Programme, die XML-Dokumente verarbeiten, besteht darin, Teilmengen mit bestimmten Kriterien dieser XML-Dokumente zu erstellen. Dazu ist es möglich, Funktionen zu schreiben, die das XML-Dokument bezüglich der Kriterien durchlaufen. In bestimmten Fällen ist es aber einfacher, einen datengesteuerten Ansatz zu nutzen. Zum Beispiel falls Kriterien nicht zu Beginn feststehen oder die zur Verfügung stehende Schnittstelle nur eine umständliche Lösung bietet. XML Path Language (XPath) verfolgt diesen Ansatz.

XPath ist eine vom W3C entwickelte Abfragesprache für XML-Dokumente. Es ermöglicht eine URL-ähnliche Notation, um über Pfade in der hierarchischen Struktur eines XML-Dokumentes zu navigieren und Teile des Dokumentes zu selektieren. XPath wurde für die Verwendung in XLST im Rahmen von XLS entwickelt, aber es kann auch als Abfragesprache für XML-Datenbanken genutzt werden.

XPath modelliert ein XML-Dokument als Baumstruktur. Darin werden XML-Elemente durch XPath-Ausdrücke adressiert. XPath-Ausdrücke bestehen aus zwei Teilen: sogenannten Achsen und optionalen Prädikaten. Die Achsen werden mit Knoten durch Eltern-Kind-Beziehungen gebildet. Sie bestimmen den Pfad zu einem XML-Element. Die Prädikate bestehen aus Relationen und Funktionen.

Ein XPath-Ausdruck hat daher die Form:

$$Achse :: Kontext[\textit{Prädikat}]$$

Der Kontext ist dabei eine Menge von Knoten, auf die die Achse reduziert wird. Das Prädikat wird danach auf den Kontext angewendet und reduziert die Menge der Knoten weiter. Eine Auflistung aller Achsen eines Knoten gibt die Abbildung 3.1.

XML-Elemente werden in mehrere Knoten unterschieden, unter anderem in Element-Knoten für XML-Elemente, Attribut-Knoten für Attribute und Text-Knoten für Inhalte von XML-Elementen. Attribut- und Text-Knoten sind daher Kinder von Element-Knoten.

Pfade werden universell durch zwei führende Backslash, absolut mit einem führenden Backslash oder relativ ohne führenden Backslash angegeben. Teilpfade werden, ähnlich wie in der URL, durch einen Backslash voneinander getrennt.

Für Achsen und Kontext gibt es mehrere Abkürzungen. Z.B. wenn die Achse nicht angegeben wird, wird standartmässig die *child* Achse genommen. Durch zwei Backslash wird die *descendant-or-self* Achse für jeden Knoten gleich welcher Art genommen.

Prädikate ermöglichen es durch Funktion oder Relation Knotenmengen zu spezifizieren, welche sich nicht allein durch ihren Pfad unterscheiden lassen. Prädikate sind ein beliebiger boolescher oder arithmetischer Ausdruck. Zusätzlich gibt es eine Reihe von vordefinierten Funktionen, die im Prädikat für den aktuellen Knoten berechnet werden.

Die vollständige Spezifikation von von XPath mit einer Auflistung aller Funktionen und Knoten-Kategorien gibt es auf der Internetseite des W3C [10].

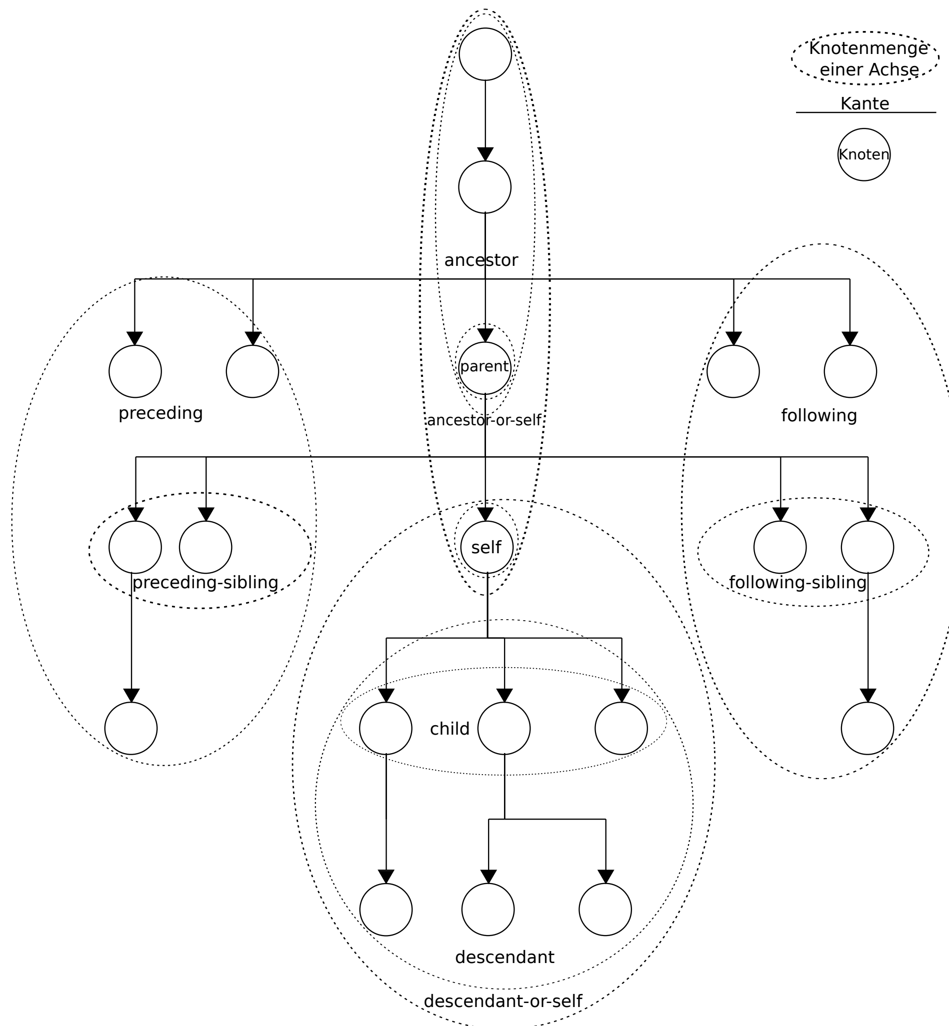


Abbildung 3.1.: Mögliche Achsen für den Kontext eines Knoten in XPath

3.3. pugixml

pugixml ist eine schlanke C++ Bibliothek zur Verarbeitung von XML-Dokumenten. Sie besteht aus einem sehr schnellen und speichereffizienten XML-Parser mit DOM-Schnittstelle. Sie hat volle Unicode-Unterstützung und ermöglicht Konvertierungen zwischen verschiedenen Unicode Kodierungen. Der Quellcode besteht aus nur zwei Header- und einer Source-Datei und ist dadurch sehr einfach und plattformunabhängig nutzbar. pugixml hat XPath 1.0 implementiert. Dadurch sind auch für komplexe XML-Dokumente einfache Abfragen möglich.

Da pugixml einen DOM Parser verwendet kann sie keine XML-Dokumente verarbeiten, die nicht in den Arbeitsspeicher geladen werden können. Sie unterstützt auch keine Validierung gegen Schemata oder DTD. Sie akzeptiert einige fehlerhafte XML-Dokumente und ist daher nicht ganz konform zum W3C Standard.

pugixml ist unter der MIT Lizenz veröffentlicht. Sie eignet sich sowohl für Open Source Projekte als auch für proprietäre Anwendungen. Sie wurde 2006 als Nachfolger von pugxml erstellt und wird seitdem durch Arseny Kapoulkine gepflegt.

Zum parsen von XML-Dokumenten wird der Quellcode in ein vorhandenes Projekt eingefügt und über den Header `pugixml.hpp` genutzt oder als eigenständige Bibliothek gebaut. Danach kann das XML-Dokument über eine Reihe von Funktionen geladen werden. pugixml unterstützt dafür XML-Daten, C++ iostreams und memory buffer. Für geladene Daten stellt pugixml Funktionen bereit, mit denen auf die Daten zugegriffen und diese verändert werden können. Der Zugriff kann direkt über das Adressieren bestimmter Knoten erfolgen oder durch rekursiven Abstieg. Zusätzlich bietet pugixml noch den Zugriff auf Daten über Xpath-Ausdrücke.

3.4. KIELER

Das *Kiel Integrated Environment for Layout Eclipse Rich Client* (KIELER) ist ein Open Source Projekt, das von der Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme an dem Institut für Informatik der Christian-Albrechts-Universität (CAU) entwickelt wird.

KIELER kann sowohl als Rich Client Applikation als auch durch Plugin-Mechanismen in einer vorhandenen Eclipse Installation genutzt werden.

Das KIELER Projekt beschäftigt sich mit der grafischen Modellierung komplexer Systeme im Bereich der Echtzeitsysteme und eingebetteten Systeme. Sein Schwerpunkt ist das automatische Layout von Modellen. KIELER unterstützt verschiedene Modellierungssichten sowie das Synthetisieren, strukturbasierte Editieren und Simulieren bestimmter Modelle.

Dazu implementiert KIELER eine Reihe von Konzepten und Technologien. Unter anderen nutzt KIELER EMF, GMF, TMF, Xpand und Xtend.

Die Konzepte zur Darstellung basieren auf einer automatischen Gestaltung.

- Dargestellte Informationen werden gefiltert, selektiv zusammengefasst und hervorgehoben.

- Beschriftungen werden automatisch platziert.
- Einzelne Bereiche der Ansicht lassen sich fokussieren. Dadurch werden diese Bereiche detaillierter angezeigt und der Kontext wird in den Hintergrund gerückt.
- Die Darstellung wird der Semantik angepasst, bzw. vom Benutzer bestimmt für einzelne Bereiche ausgewählt.
- In der Ansicht wird die Positionierung von Objekten durch Algorithmen für die einzelnen Ansichten berechnet. Dazu werden verschiedene Algorithmen angeboten.

Bei der Modellierung werden grundsätzlich zwei Konzepte angewendet.

- Modelle können aus anderen Modellen durch Muster oder Skripte synthetisiert werden.
- Für Änderungen wird strukturbasiertes Editieren genutzt. Der Nutzer wählt dazu eine Aktion aus einer Palette aus und gibt an, wo sie im Modell hinzugefügt werden soll. Die Darstellung wird automatisch aktualisiert.

Eine umfangreiche Dokumentation zu KIELER befindet sich auf der Internetseite der Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme [2].

Außerdem werden noch Konzepte zur Simulation genutzt auf die hier nicht weiter eingegangen wird [3].

3.5. KAOM

KIELER Actor Oriented Modeling (KAOM) ist eine Komponente von KIELER, um Aktororientierte Modelle zu beschreiben und in KIELER zu visualisieren.

Eine Übersicht über KAOM gibt das Metamodel in Abbildung 3.2. Dadurch können Aktororientierte Modelle implementiert und in KiVi, einer weiteren KIELER Komponente, visualisiert werden. Eine Grammatik zur textuellen Darstellung von KAOM-Modellen ist in Tabelle 3.1 zu sehen [7].

Ptolemy: KAOM wird in KIELER beispielsweise angewendet, um Modelle aus Ptolemy zu verwenden. Ptolemy ist ein Open Source Projekt zur Modellierung von Echtzeitsystemen und eingebetteten Systemen. Es wird an der Universität von Kalifornien, Berkeley entwickelt. Sein Schwerpunkt liegt bei der Simulation von nebenläufigen Modellen [1].

TopEntity	→	(*impAnn*)* EntityHead
EntityHead	→	((*ann*)* 'entity' *id* (*string*)? EntityBody)?
EntityBody	→	({ SubEntity } | ';')
SubEntity	→	(Entity | Link | Port | Relation)
Entity	→	(*ann*)* 'entity' *id* (*string*)? EntityBody
Link	→	(*ann*)* 'link' (*string*)? Reference 'to' Reference ';'
Reference	→	(Linkable | *id*)
Linkable	→	(Entity | Port | Relation)
Port	→	(*ann*)* 'port' *id* (*string*)? ';'
Relation	→	(*ann*)* 'relation' *id* (*string*)? ';'

Tabelle 3.1.: Grammatik von KAOM

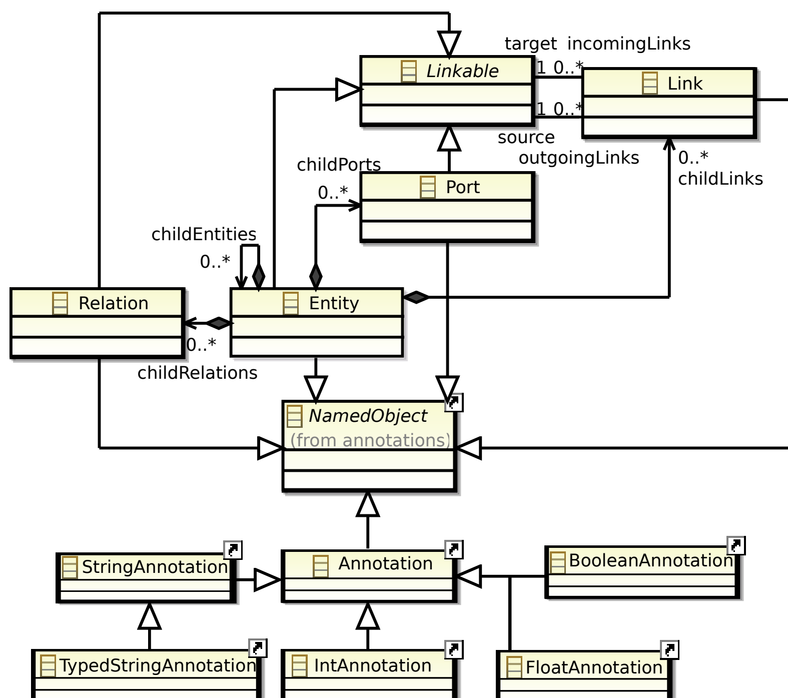


Abbildung 3.2.: Metamodel von KAOM

4. Implementierung

Dieses Kapitel befasst sich mit der Entwicklung von Annotationen zum Quellcode, aus denen Datenflussdiagramme synthetisiert werden können und der Implementierung eines Prototypen, der Annotationen in KAOM-Modelle überführt. Es stellt den Hauptteil dieser Arbeit dar. Im Kapitel wird zuerst der Aufbau der Annotationen vorgestellt und dann anschließend auf den Ablauf des Programms eingegangen. Dazu werden die im Kapitel 3 eingeführten Technologien verwendet.

4.1. Aufbau der Annotationen

Die Annotationen sollen dazu verwendet werden, ein Datenflussdiagramm, wie es in Abschnitt 2.3 beschrieben wurde, zu erstellen. Das Diagramm wird abstrakt dargestellt. Damit die Annotationen das Programm nicht beeinflussen, werden sie in Kommentare geschrieben. Funktionen werden einheitlich als Aktoren verwendet. Daher wird die Darstellung keine Strukturen für bestimmte Aktoren enthalten und nicht auf spezielle Funktionen von Aktoren eingehen. Datenflüsse, die sich verzweigen, zusammenfließen oder ungerichtet in beide Richtungen verlaufen, werden als mehrere Datenflüsse dargestellt, sodass es nur noch einfach gerichtete Datenflüsse gibt. Die Darstellung von mehreren Datenflüssen als Hyperkante wird der Visualisierung überlassen.

Die Annotation eines Aktors spiegelt seine Schnittstelle wieder. Für die interne Darstellung enthält sie alle Informationen über seine internen Datenflüsse. Außerdem enthält die Annotation alle zur Identifikation der enthaltenen Aktoren notwendigen Informationen sowie deren Bezeichner. Für die externe Darstellung werden nur die Ein- und Ausgänge festgelegt. Alle weiteren Informationen zur externen Darstellung werden durch Annotationen übergeordneter Aktoren festgelegt. Die einzige Ausnahme ist, wenn es keine übergeordneten Aktoren gibt. Dann werden alle Informationen zur externen Darstellung durch die Annotation des Aktors der obersten Ebene festgelegt.

Jede Annotation wird in einen C-Kommentar geschrieben und beginnt mit dem Schlüsselwort `@kaom`. Außerdem wird der Kommentar mit `/*!` für den Qt-Stil oder `/**` für den JavaDoc-Stil eingeleitet.

Eine Annotation wird durch Schlüsselwörter in fünf Kategorien strukturiert:

- “`input:`” steht vor der Auflistung aller Eingänge.
- “`output:`” steht vor der Auflistung aller Ausgänge.
- “`link:`” steht vor der Auflistung aller Datenflüsse.

```
/*! @kaom
input:  ID_1 "Bezeichner", ID_2 "Bezeichner", ... ;
output: ID_1 "Bezeichner", ID_2 "Bezeichner", ... ;
link:   Quelle -> Kontext.Ziel, Kontext.Quelle -> Kontext.
   Ziel, Quelle -> Ziel, ... ;
content:Funktion_1:ID "Bezeichner", Funktion_2:ID "
   Bezeichner", ... ;
toplevel:ID "Bezeichner";
*/
```

Auflistung 4.1: Abstraktes Beispiel für eine Annotation

- “`content:`” steht vor der Auflistung aller enthaltenen Aktoren.
- Mit “`toplevel:`” werden Aktoren oberster Ebene gekennzeichnet.

Eingänge und Ausgänge werden allgemein durch eine in der Annotation eindeutige ID und optional mit einem beliebigen Bezeichner angegeben.

Datenflüsse werden durch die IDs ihrer Quelle und ihres Zieles angegeben. Werden die Quelle oder das Ziel in einem anderen Aktor als der Datenfluss definiert, dann muss zusätzlich noch die ID des Aktors, in dem Quelle oder Ziel definiert sind, angegeben werden. Die ID des entsprechenden Aktors wird vor die ID von Quelle oder Ziel geschrieben und mit einem Punkt abgetrennt. Quelle und Ziel werden durch einen Pfeil nach rechts, der die Flussrichtung angibt, getrennt. Es ist nicht zulässig, dass sich Quelle und Ziel um mehr als eine Ebene unterscheiden.

Enthaltene Aktoren werden auch durch eine ID und einen optionalen Bezeichner angegeben. Zusätzlich muss noch der Name der Funktion, die sie im Code repräsentieren, angegeben werden. Der Funktionsname wird vor die ID geschrieben und mit einem Doppelpunkt abgetrennt.

Für Aktoren oberster Ordnung wird eine ID und ein optionaler Bezeichner angegeben. Für andere Aktoren wird diese Kategorie weggelassen.

In allen Kategorien steht ein Komma zwischen Aufzählungen. Alle Kategorien werden durch ein Semikolon abgeschlossen. Bezeichner werden in Anführungszeichen geschrieben.

Beispiel zu Abbildung 2.5: Für das Beispiel können die Bezeichner der Eingänge und Ausgänge als IDs verwendet werden. Die inneren Aktoren könnten ebenfalls durch ihre Bezeichner eindeutig angegeben werden. Die Bezeichner würden dann auch zur Definition der Datenflüsse genutzt werden. Außerdem würde der äußere Aktor als Aktor oberster Ebene fungieren. Eine mögliche Annotation für den äußeren Aktor zeigt Auflistung 4.2.

```
/*! @kaom
input: i1 ;
output: o1, o2 ;
link: i1 -> splitter.i1, splitter.o1 -> view one.i1, view
   one.o1 -> o1, splitter.o1 -> view two.i1, view two.o1
   -> o2;
content: splitter:splitter, view:view one, view:view two;
toplevel: view management;
*/
```

Auflistung 4.2: Annotation für den Aktor oberster Ebene aus Abbildung 2.5

4.2. Programmstruktur

In diesem Abschnitt wird die Programmstruktur von C2KAOM vorgestellt. Grundsätzlich werden die Quelldateien durch Doxygen in XML-Dokumente überführt. Die XML-Dokumente werden durch pugixml mit XPath gefiltert und in interne Strukturen überführt. Aus diesen Strukturen wird dann ein KAOM-Modell durch Iteration über die enthaltenen Aktoren erstellt.

Alle Schritte werden im Programm jeweils durch eine spezielle Klasse repräsentiert. Das Programm ist daher in drei Klassen unterteilt.

Das Programm startet am Anfang mit der Funktion Main, die für den grundsätzlichen Ablauf des Programms zuständig ist. Als Eingabe verwendet die Main Funktion das Quell- und Zielverzeichnis. Die Main Funktion konfiguriert und startet zuerst das Programm Doxygen. Dies geschieht durch die Klasse `BuilderXml`. Die Eingabe dieser Klasse sind Quell- und Zielverzeichnis aus der Eingabe der Main-Funktion. Die Ausgabe ist eine Liste aller XML-Dokumente, die von Doxygen für Quelldateien erstellt wurden.

Die nächsten beiden Schritte werden für jedes XML-Dokument einzeln ausgeführt. Im ersten Schritt wird das XML-Dokument gefiltert. Dieser Schritt wird durch die Klasse `FilterXml` repräsentiert. `FilterXml` nutzt den Parser pugixml, um alle relevanten Informationen mit XPath aus dem XML-Dokument heraus zu filtern. Die Eingabe ist ein XML-Dokument. Falls das XML-Dokument eine Quelldatei repräsentiert, ist die Ausgabe eine Warteschlange. Die Warteschlange enthält jeweils ein Element zu jeder enthaltenen Funktion mit allen relevanten Informationen zu dieser Funktion. Falls das XML-Dokument keine Quelldatei repräsentiert, werden die relevanten Informationen über eine globale Struktur für alle XML-Dokumente verfügbar gemacht. Außerdem wird ermittelt, ob Annotationen im XML-Dokument gefunden wurden. Wenn Annotationen gefunden wurden, wird die Ausgabe an die Klasse `BuilderKaom` weitergegeben. Diese Klasse erstellt aus der Warteschlange eine `.kaot`-Datei, aus der ein Datenflussdiagramm erstellt werden kann. Eine `.kaot`-Datei enthält eine textuelle Beschreibung eines KAOM-Modells. Die Beschreibung eines KAOM-Modells folgt der Grammatik wie sie in Tabelle 3.1 beschrieben ist.

4.3. Erstellen der XML-Dokumente

Das Überführen von Quellcode nach XML wird durch die Klasse `BuilderXml` realisiert. Für das Erstellen der XML-Dokumente wird von der Eingabe das Quell- und Zielverzeichnis übernommen und in die Konfigurationsdatei von Doxygen geschrieben. Danach wird Doxygen mit der modifizierten Konfigurationsdatei über die Kommandokonsole automatisch gestartet.

Die Konfigurationsdatei ist so voreingestellt, dass Doxygen keine Nachrichten oder Warnungen auf der Kommandokonsole ausgibt, es sei denn Doxygen findet Fehler. Fehler werden in einer Logdatei gespeichert und führen zur Abbruch des Programms. Alle XML-Dokumente werden von Doxygen in das Unterverzeichnis `xml` des Zielverzeichnisses geschrieben. Außerdem sind alle Endungen möglicher Quelldateien in der Konfigurationsdatei unter `FILE_PATTERNS` aufgeführt.

Doxygen durchsucht das Quellverzeichnis und dessen Unterverzeichnisse nach Dateien mit entsprechenden Endungen. In diesen Dateien sucht Doxygen nach Kommentaren, wie sie in Abschnitt 3.1 beschrieben sind.

In den Kommentaren markiert das Schlüsselwort `@kaom` Annotationen, die für Datenflussdiagramme verwendet werden sollen. Diese Informationen verwendet Doxygen als Kurzbeschreibung der entsprechenden Funktion. Zu jeder Funktion erzeugt Doxygen ein XML-Element im XML-Dokument. Das XML-Element enthält den Typ, den Namen, die Definition, die Argumente, die Kurzbeschreibung und die detaillierte Beschreibung der Funktion. Außerdem wird für jede Funktion der Pfad zur Quelldatei und die Zeilenangaben in der Datei abgespeichert. Für Objekte, die nur deklariert werden, wie *structs* oder Klassen, werden gesonderte XML-Dokumente angelegt. Diese XML-Dokumente enthalten für das gesamte Objekt ein XML-Element, in dem unter anderem der Objektname, die Attribute, die Kurzbeschreibung und die detaillierte Beschreibung enthalten sind. Für die Dateien, in denen die Objekte deklariert wurden, wird im entsprechenden XML-Dokument ein XML-Element hinzugefügt, das auf das XML-Dokument des Objektes verweist. Neben den XML-Dokumenten für die Quelldateien erzeugt Doxygen auch eine Reihe von Daten, die für eine umfassende Dokumentation den Zusammenhang der XML-Dokumente angeben.

Wenn Doxygen alle XML-Dokumente erzeugt hat, wird im Unterverzeichnis `xml` nach allen XML-Dokumenten gesucht, die nicht mit "`dir_`" anfangen oder nicht "`index.xml`" sind. XML-Dokumente, deren Dateiname mit "`dir_`" anfängt, enthalten nur Informationen über die Verzeichnisstruktur des Quellverzeichnisses. Die Datei "`index.xml`" enthält eine Zusammenfassung des Programms, jedoch keine weiteren relevanten Details. Daher werden diese Dateien nicht weiter beachtet. Alle anderen XML-Dokumente können relevante Informationen enthalten. Daher werden deren Dateinamen in eine Warteschlange geschrieben und an die Main-Funktion zurückgegeben.

```
<compoundname>view.cpp</compoundname>
...
 <memberdef ...>
 <type>void</type>
 <definition>void management</definition>
 <argsstring>()</argsstring>
 <name>management</name>
 <briefdescription>
 <para>input: ... ; output: ... ; link: ... ; content: ... ;
    toplevel: ... ; </para>
 </briefdescription>
 <detaileddescription>
 </detaileddescription>
 <inbodydescription>
 </inbodydescription>
 <location .../>
 </memberdef>
...
```

Auflistung 4.3: XML-Dokument mit Auschnitt vom XML-Element für view management aus Abbildung 2.5

4.4. Filtern der Annotationen

Der Schritt zum Filtern der Annotationen aus XML-Dokumenten wird durch die Klasse `FilterXml` realisiert. `FilterXml` muss für jedes XML-Dokument einzeln aufgerufen werden. In `FilterXml` werden die Annotationen, die von Doxygen als Kurzbeschreibung abgespeichert wurden, zusammen mit den Objektnamen und dem Dateinamen extrahiert und in eine Warteschlange überführt. Dafür werden die XML-Dokumente durch pugixml eingelesen und mit XPath gefiltert.

Zuerst wird von pugixml ein DOM-Baum für das entsprechende XML-Dokument erzeugt. Danach wird aus diesem DOM-Baum ein Teilbaum durch XPath erstellt.

Dafür werden fünf XPath-Ausdrücke verwendet, die den Dateinamen, die Funktionsnamen, die Kurzbeschreibungen und verwiesene Objekte heraus filtern. Da die Kurzbeschreibungen mit den Annotationen je nach XML-Dokument an zwei Stellen stehen können, werden zwei XPath-Ausdrücke verwendet. Je nach Fundstelle der Kurzbeschreibung, werden die gefilterten Informationen global für alle XML-Dokumente oder nur für das aktuelle XML-Dokument genutzt.

Namen von Objekten, die durch das XML-Dokument repräsentiert werden, befinden sich im XML-Element `<compoundname>` und können daher durch den Ausdruck `//compoundname` selektiert werden. Namen sind entweder Dateinamen oder Namen von Objekten im Quellcode.

```
//memberdef/briefdescription/para | //memberdef/
    briefdescription/para/preceding::name[1] | //
    compoundname | //compounddef/innerclass
```

Auflistung 4.4: XPath-Ausdrücke zum Filtern lokal verwendeter Informationen

```
//compounddef/briefdescription/para | //compoundname
```

Auflistung 4.5: XPath-Ausdrücke zum Filtern global verwendeter Informationen

Kurzbeschreibungen befinden sich für alle enthaltenen Elemente innerhalb von memberdef-Elementen. Für Elemente, die durch das XML-Dokument repräsentiert werden, befinden sie sich im compounddef-Element. Innerhalb von compounddef- oder memberdef-Elementen befinden sich die Kurzbeschreibungen im XML-Element `<briefdescription>`. Ihr Inhalt wird in einem inneren Element `<para>` gespeichert. Der XPath-Ausdruck ist daher entweder `//memberdef/briefdescription/para` oder `//compounddef/briefdescription/para` und ist analog zum vorherigen Ausdruck aufgebaut.

Verweisende Objekte befinden sich in innerclass-Elementen innerhalb des compounddef-Elementes. Daher ist der XPath-Ausdruck `//compounddef/innerclass` ebenfalls analog zum ersten Ausdruck aufgebaut.

Die Funktionsnamen stehen im XML-Element `<name>` direkt vor dem XML-Element `<briefdescription>`. Daher können sie über zwei Wege gefunden werden. Erstens kann, analog zu den ersten beiden Ausdrücken, das gesamte Dokument nach name-Elementen innerhalb von memberdef-Elementen durchsucht werden. Entsprechend wäre der Ausdruck einfach `//memberdef/name`. Dieser Ausdruck würde aber auch name-Elemente innerhalb von memberdef-Elementen ohne Annotationen finden, da alle Funktionen standardmäßig ein name-Element haben. Um nur die memberdef-Elemente mit Annotationen zu finden, kann man folgenden Sachverhalt nutzen. Das name-Element steht vor dem briefdescription-Element und das briefdescription-Element enthält nur dann ein inneres para-Element, wenn eine Annotation vorhanden ist. Daher kann man jedes erste name-Element in der preceding-Achse der para-Elemente suchen. Der XPath-Ausdruck dafür ist `//memberdef/briefdescription/para/preceding::name[1]`.

Die einzelnen XPath-Ausdrücke können durch eine Aufzählung mit `|` zu einem XPath-Ausdruck konkateniert werden. Um globale oder lokale Informationen zu filtern, werden die XPath-Ausdrücke entsprechend zu zwei XPath-Ausdrücken zusammengesetzt. Global werden Informationen zu Annotationen genutzt, die zum Objekt gehören, das von einem XML-Dokument repräsentiert wird. Lokal verwendete Informationen werden durch innere XML-Elemente repräsentiert. Für die Suche nach global verwendeten Informationen wird daher der XPath-Ausdruck 4.5 genutzt und für die Suche nach lokal verwendeten Informationen der Ausdruck 4.4.

Diese XPath-Ausdrücke ordnen die gefundenen XML-Elemente in derselben Reihenfolge an,

in der sie im XML-Dokument angeordnet waren. Dadurch wird eine komplizierte Rekonstruktion der Anordnung vermieden.

Das XML-Dokument wird zuerst nach globalen Informationen durchsucht. Wenn es solche enthält, kann es keine lokal verwendbaren Informationen enthalten und repräsentiert damit auch keine Quelldatei, sodass dafür auch kein KAOM-Modell erstellt wird. Wenn keine globalen Informationen gefunden wurden, werden lokale Informationen gesucht und gegebenenfalls daraus ein KAOM-Modell erstellt.

Die gefundenen Kurzbeschreibungen können außer den Annotationen Teile enthalten, die nicht zur Darstellung des Datenflussdiagrammes gebraucht werden. Daher werden die Kurzbeschreibungen nach relevanten Informationen durchsucht. Diese werden durch die Schlüsselwörter `input:`, `output:`, `link:`, `content:` und `toplevel:` eingeleitet und durch Semikolons abgeschlossen. Auf diese Weise können die Annotationen von irrelevanten Informationen getrennt werden.

Könnte ein KAOM-Modell aus den vorhandenen Informationen erstellt werden, wird ermittelt, ob außer dem Dateinamen noch weitere Elemente gefunden wurden. Wurden keine anderen Elemente gefunden, wird dies an die Main-Funktion zurückgegeben und aus dem entsprechenden XML-Dokument wird kein KAOM-Modell erstellt. Die Kurzbeschreibungen werden dann nicht nach Annotationen durchsucht, da keine Kommentare in der Datei als Annotation markiert waren. Ansonsten werden die einzelnen Annotationen herausgesucht und mit dem entsprechenden Funktionsnamen zu Elementen zusammengesetzt. Diese Elemente werden dann zur Weiterverarbeitung in eine Warteschlange geschrieben.

4.5. Erstellen der KAOM-Modelle

Der letzte Schritt, das Erzeugen der KAOM-Modelle aus den gefilterten Annotationen, wird durch die Klasse `BuilderKaom` repräsentiert. `BuilderKaom` wird nur aufgerufen, falls beim vorausgegangenen Aufruf von `FilterXml` Annotationen ermittelt wurden. Ist dass der Fall, wird das Ergebnis von `FilterXml` zusammen mit dem Pfad zur Zieldatei an `BuilderKaom` übergeben.

Zuerst wird der Kopf des KAOM-Modells mit einem leeren Rumpf und einer leeren priorisierten Liste erzeugt. Das KAOM-Modell wird dann stückweise um die Aufrufe der Aktoren oberster Ebene erweitert. Dazu werden die Elemente der Warteschlange einzeln geladen und in KAOM-Vorlagen für die jeweiligen Funktionen überführt.

Eine KAOM-Vorlage stellt ein abstraktes Modell einer konkreten Funktion dar, sodass daraus KAOM-Code einfach erzeugt werden kann. Die konkrete Modellierung wird durch Parameter, die beim Aufruf der Vorlage von der aufrufenden Vorlage übergeben werden, festgelegt. Auf diese Weise kann dieselbe Vorlage für eine Funktion überall im Modell verwendet werden.

Aus diesen Vorlagen wird das KAOM-Modell iterativ erzeugt, indem jede Vorlage Aufrufe für innere Funktionen hat.

Um die Elemente der Warteschlange in Vorlagen zu überführen, werden die Annotationen in den Elementen gemäß der Schlüsselwörter in Kategorien zerlegt. Für jedes Element der

```
@portConstraints Free entity <ID@lias> " <@lias> " {}
```

Auflistung 4.6: Vorlage einer Funktion mit Kopf und leeren Rumpf

```
@portConstraints Free entity <ID@lias> " <@lias> "
{
port <ID@lias>_i1 "i1";
port <ID@lias>_o1 "o1";
port <ID@lias>_o2 "o2";
link <ID@lias>_i1 to <ID@lias>_splitter_i1;
link <ID@lias>_splitter_o1 to <ID@lias>_viewone_i1;
link <ID@lias>_viewone_o1 to <ID@lias>_o1;
link <ID@lias>_splitter_o1 to <ID@lias>_viewtwo_i1;
link <ID@lias>_viewtwo_o1 to <ID@lias>_o2;
repl@ce splitter:splitter "label";
repl@ce view:view one;
repl@ce view:view two;
}
```

Auflistung 4.7: Vorlage für view management aus Abbildung 2.5

Warteschlange wird eine neue Vorlage erzeugt. Diese Vorlage stellt zunächst nur den Kopf der Funktion mit einem leeren Rumpf dar. Dafür wird der Code 4.6 erstellt und entsprechend der einzelnen Kategorien durch jeden Eintrag folgendermaßen um Code erweitert:

- input: und output: → port <ID@lias>_*id* “*label*”;

  id und label werden durch die ID und den Bezeichner des Eintrags ersetzt.

- link: → link <ID@lias>_*sourceId* to <ID@lias> _*targetId*;

  sourceId und targetId werden durch die ID von Quelle und Ziel des Eintrags ersetzt.

- content: → repl@ce *function*:*id* “*label*”;

  function wird durch den Funktionsnamen, id durch die ID und label durch den Bezeichner ersetzt.

Die zusammengesetzte Vorlage wird schließlich unter dem Funktionsnamen, der im Quellcode repräsentierten Funktion, in die priorisierte Liste geschrieben.

Für Einträge in der Kategorie toplevel: wird der Rumpf des KAOM-Modells um den Aufruf repl@ce *function*:*id* “*label*”; erweitert.

Nachdem alle Vorlagen erstellt, gespeichert und der Rumpf des KAOM-Modells erweitert wurde, wird das KAOM-Modell iterativ aufgebaut.

Der Aufbau wird in zwei Schritten durchgeführt. Zuerst wird das vorderste `repl@ce` durch die entsprechende Vorlage ersetzt. Dann werden alle `<ID@lias>` und `<@lias>` bis zum nächsten `repl@ce` durch die Parameter `id` und `label` ersetzt. Für `<ID@lias>` werden neben der übergebenen ID auch alle ID aller Vorgänger der Vorlage eingesetzt. Vorgänger einer Vorlage sind alle textuell umfassenden Vorlagen. Dadurch ist die ID im gesamten Modell eindeutig, solange sich die ID der Vorgänger zweier Vorlagen oder die ID der Vorlagen selbst um mindestens eine ID unterscheiden. Diese Bedingung ist dadurch erfüllt, dass eine ID in dem Kontext, in dem sie definiert wurde, eindeutig ist. Dieses muss jedoch durch den Entwickler eingehalten werden, da es nicht vom Programm überprüft wird. Die Schritte zum Aufbau des Modells werden wiederholt bis alle `repl@ce`, `<ID@lias>` und `<@lias>` ersetzt wurden.

Das komplette KAOM-Modell wird abschließend im Zielverzeichnis unter dem Dateinamen der Quelldatei als `.kaot`-Datei abgespeichert.

Der Aufbau und die Verarbeitung der Vorlagen ist unabhängig von der Synthese relevanter Informationen, die bisher aus den Annotationen entnommen werden. ID und Bezeichner könnten z.B. alternativ aus den Funktionsnamen generiert werden. Enthaltene Aktoren oder Datenflüsse könnten über Ausdrücke oder Anweisungen bestimmt werden. Problematisch bei Aktoren oder Datenflüsse ist aber die große Vielfalt an Seiteneffekten, wie sie z.B. bei Zeigern und Referenzen auftauchen.

5. Zusammenfassung und Ausblick

Datenflussdiagramme sind eine effiziente Methode zur Modellierung von Abläufen in der Signalverarbeitung. KIELER bietet eine große Anzahl an Möglichkeiten zur Modellbildung und Visualisierung von Modellen. Es bietet sich daher an, Datenflussdiagramme mit KIELER automatisch erstellen zu lassen.

Diese Arbeit hat untersucht, wie man in Datenflussdiagrammen enthaltene Informationen sinnvoll und kompakt aufschreiben kann, sodass diese automatisch verarbeitet werden können ohne den Programmablauf zu beeinflussen. Das Ergebnis ist ein Vorschlag, der den Aufwand durch Einsatz vorhandener Technologien überschaubar hält.

Es wurden Annotationen entwickelt, die die Schnittstelle der Aktoren in Aktor-orientierten Modellen widerspiegeln. Die Strukturierung durch Schlüsselwörter macht die entworfenen Annotationen nicht nur besser lesbar, sondern lässt die Annotationen auch effizient und automatisch verarbeiten.

Mit dem Programm Doxygen wurde eine robuste Möglichkeit gefunden, beliebige C, C++ oder C- nahe Quelldateien zu analysieren, ohne auf spezielle Eigenschaften von den Programmiersprachen einzugehen. Zusätzlich ermöglicht Doxygen die individuelle Anpassung des Analyseergebnisses. Dadurch ist eine effizient zu verarbeitende Ausgabe in XML möglich. C2KAOM benutzt zur Weiterverarbeitung den Parser pugixml. pugixml ermöglicht den Einsatz von XPath, wodurch die Analyse von XML-Dokumenten erheblich vereinfacht wird. Durch XPath wurde eine komplizierte Verarbeitung der XML-Dokumente vermieden.

Da eine direkte Erzeugung von Modellen, wie sie in KIELER verwendet werden können, nicht effizient ist, wurde eine interne Datenstruktur zur Erzeugung von Modellen entworfen. Die Datenstruktur ist einerseits unabhängig vom Aufbau der Annotationen und kann andererseits einfach in ein KAOM-Modell überführt werden.

Im KAOM-Modell wurden nur wenige Annotationen verwendet und Hyperkanten ganz weggelassen. Dadurch wurde die Visualisierung des Modells möglichst wenig eingeschränkt.

Insgesamt konnte der Aufwand, um die Annotationen zu verarbeiten, durch zusätzliche Technologien überschaubar gehalten werden. Es ist aber noch zu klären, wie die Verarbeitung effizienter gestaltet werden kann.

Die folgenden Abschnitte befassen sich mit Punkten, die noch im Programm verbessert oder im Anschluss daran durchgeführt werden können. Sie bieten einen Überblick über die Erweiterungsmöglichkeiten oder Themen für zukünftige Arbeiten.

Einbau in KiRAT: Der Ausgangspunkt dieser Arbeit ist KiRAT. Innerhalb von KiRAT sollten Entwickler die Möglichkeit haben, Datenflussdiagramme automatisch zu generieren.

C2KAOM ermöglicht dies nur in KIELER. In einer Erweiterung oder anschließenden Arbeit könnte daher eine Visualisierung der Datenflussdiagramme für KiRAT entwickelt werden.

Hierfür müssen Programme bereitgestellt werden, die die aufwändigere Darstellung wie z.B. in der Abbildung 2.2 ermöglichen. Außerdem müssen KAOM-Modelle in das KiRAT-Format überführt werden. Probleme können hierbei zusätzliche Elemente sein, die nicht in KAOM vorkommen (Spezialknoten). Außerdem bietet KiRAT Möglichkeiten zur Interaktion mit den Diagrammen, die speziell behandelt werden müssten.

Zusammenfassend ist zu klären, wie KAOM-Modelle effizient in ein Format, das in KiRAT verwendet werden kann, überführt werden können.

Kontextfreie Grammatik: Die Fehlerbehandlung der Eingabe kann durch moderne Programmiermethoden wie z.B. reguläre Ausdrücke oder kontextfreie Grammatiken erheblich verbessert werden. Diese Möglichkeiten waren bisher nur über spezielle Bibliotheken in C++ verfügbar. Seit Dezember 2011 gibt es einen neuen C-Standard (*C1X*). Dieser Standard ermöglicht es, viele moderne Programmiermethoden einzusetzen. Es wäre daher denkbar, dass durch den Einsatz von C1X eine effizientere Lösung des Problems möglich ist. Zur Klärung müsste daher der C1X-Standard umfassend betrachtet werden.

Datenstruktur für Aktoren: Bisher werden für die Darstellung von Aktoren Paare von Funktionsnamen und Vorlagen verwendet. Durch spezielle Objekte könnte die Synthese der KAOM-Modelle vereinfacht bzw. verallgemeinert werden. Dadurch könnten auch abstraktere Ergebnisse der Analyse zur Synthese genutzt werden. Zum Beispiel könnte es möglich sein, vollständig automatisch generierte Ergebnisse zu verarbeiten. Der Aufbau einer solchen Datenstruktur wäre zu klären.

Formatierung und Grafische Visualisierung: In dieser Arbeit wurde die Visualisierung der Datenflussdiagramme möglichst nicht eingeschränkt. Das erstellte Modell wurde so allgemein gehalten, dass die standardmäßige Visualisierung brauchbare Ergebnisse liefert. Die Darstellung kann durch verschiedene Möglichkeiten verbessert werden. Z.B. ist die textuelle Ausgabe unformatiert und die Formatierung kann bisher nur manuell in KIELER durchgeführt werden. Diese Formatierung könnte durch einen *Formatter* auch automatisch durchgeführt werden. Außerdem gibt es in KAOM-Modellen die Möglichkeit, Hyperkanten darzustellen oder durch Annotationen die Darstellung weiter zu beeinflussen. Darüber hinaus könnte auch ein speziellerer Editor zur Visualisierung entwickelt werden. Hier gibt es noch viele Richtungen, in die weiter geforscht werden könnte.

Hilfsdateien entfernen: C2KAOM verwendet Doxygen und pugixml. pugixml lässt sich direkt in den Quellcode von C2KAOM integrieren, aber Doxygen muss unabhängig von C2KAOM installiert werden. Außerdem werden Hilfsdateien, die von Doxygen erzeugt werden, nicht weiter behandelt. Diese Hilfsdateien können jedoch die Ergebnisse späterer Ausführungen beeinflussen. Es ist daher einerseits zu klären, wie Hilfsdateien entfernt werden

können, andererseits, wie C2KAOM weiterentwickelt werden kann, sodass Doxygen nicht mehr benötigt wird.

Eine andere Möglichkeit wäre aus Doxygen direkt KAOM-Modelle zu erzeugen. Doxygen ermöglicht es über Erweiterungen komplett neue Dokumentationen zu erstellen. Dazu müssten Erweiterungen für Doxygen implementiert werden.

Einsatz für andere Programmiersprachen: C2KAOM ist für die Verarbeitung von C- oder C++-Quelldateien entworfen worden. Der Aufbau der Annotationen und die einzelnen Überlegungen zur Verarbeitung der Annotationen lassen sich auf andere Programmiersprachen übertragen. Aufbauend auf diesen Überlegungen könnte man C2KAOM weiterentwickeln, sodass C2KAOM nicht mehr auf C oder C++ beschränkt ist und sondern auch für andere Programmiersprachen verwendet werden kann.

Zusammensetzen einzelner Datenflussdiagramme: Bisher wird für jede einzelne Quelldatei ein Datenflussdiagramm erzeugt. Vollständige Programme bestehen jedoch aus mehreren Quelldateien. Für eine komplette Ansicht eines Programms ist es daher passend, ein Datenflussdiagramm für ein gesamtes Programm aus mehreren Dateien zu erhalten. Die in dieser Arbeit gemachten Überlegungen, Annotationen einer Datei zusammenzusetzen, könnten dazu auf das Zusammensetzen von Annotationen aus mehreren Dateien übertragen werden.

Literaturverzeichnis

[1] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[2] Hauke Fuhrmann and Reinhard von Hanxleden. The Kiel Integrated Environment for Layout for the Eclipse RichClientPlatform (KIELER) Homepage, 2009. `http://www.informatik.uni-kiel.de/rtsys/kieler/`.

[3] Hauke Fuhrmann and Reinhard von Hanxleden. On the pragmatics of model-based design. In *Foundations of Computer Software. Future Trends and Techniques for Development—15th Monterey Workshop 2008, Budapest, Hungary, September 24–26, 2008, Revised Selected Papers*, volume 6028 of *LNCS*, pages 116–140, 2010.

[4] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers (JCSC)*, 12(3):231–260, 2003.

[5] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. Lulu, 2011.

[6] David Megginson. SAX 2.0: The simple API for XML, 2000. `http://www.megginson.com/SAX/index.html`.

[7] Christian Motika, Miro Spönemann, Hauke Fuhrmann, Christoph Krüger, John Julian Carstens, and Reinhard von Hanxleden. KIELER Actor Oriented Modeling (KAOM). Poster presented at 9th Biennial Ptolemy Miniconference (PTCONF'11), Berkeley, CA, USA, February 2011.

[8] Arie van Deursen and Claudio Riva. Software architecture reconstruction. *Proceedings of the 26th International Conference on Software Engineering*, pages 745–746, May 2004.

[9] W3C DOM IG. W3C Document Object Model, 2007. `http://www.w3.org/DOM/`.

[10] World Wide Web Consortium (W3C). XPath homepage. `http://www.w3.org/TR/xpath`.

Tabellenverzeichnis

1. Einführung **1**

2. Grundlagen **5**
2.1. Auswahl an Dokumentationswerkzeugen für diese Arbeit 12
2.2. Auswahl an Parsern für diese Arbeit 14

3. Verwendete Technologien **17**
3.1. Grammatik von KAOM 23

4. Implementierung **25**

5. Zusammenfassung und Ausblick **35**

A. Bedienungsanleitung **47**

B. C++-Code für C2KAOM **51**

Abbildungsverzeichnis

1. Einführung **1**

2. Grundlagen **5**

2.1. Programmaufbau von KiRAT 6
2.2. Eine Ansicht der KiRAT-GUI 6
2.3. Notationen für Datenflussdiagramme in der digitalen Signalverarbeitung . . 9
2.4. Einfaches Datenflussdiagramm aus der strukturierten Analyse 9
2.5. Einfaches Datenflussdiagramm mit hierarchischen Aufbau aus der digitalen Signalverarbeitung 10

3. Verwendete Technologien **17**

3.1. Mögliche Achsen für den Kontext eines Knoten in XPath 20
3.2. Metamodel von KAOM 23

4. Implementierung **25**

5. Zusammenfassung und Ausblick **35**

A. Bedienungsanleitung **47**

B. C++-Code für C2KAOM **51**

Verzeichnis der Auflistungen

1. Einführung 1

2. Grundlagen 5

3. Verwendete Technologien 17

3.1. Beispielkommentare für Doxygen 18

4. Implementierung 25

4.1. Abstraktes Beispiel für eine Annotation 26
4.2. Annotation für den Aktor oberster Ebene aus Abbildung 2.5 27
4.3. XML-Dokument mit Auschnitt vom XML-Element für view management aus Abbildung 2.5 29
4.4. XPath-Ausdrücke zum Filtern lokal verwendeter Informationen 30
4.5. XPath-Ausdrücke zum Filtern global verwendeter Informationen 30
4.6. Vorlage einer Funktion mit Kopf und leeren Rumpf 32
4.7. Vorlage für view management aus Abbildung 2.5 32

5. Zusammenfassung und Ausblick 35

A. Bedienungsanleitung 47

A.1. Allgemeines Beispiel für eine Annotation 48
A.2. Allgemeines Beispiel zur Angabe von Eingängen 48
A.3. Allgemeines Beispiel zur Angabe von Ausgängen 48
A.4. Allgemeines Beispiel zur Angabe von Datenflüssen 49
A.5. Allgemeines Beispiel zur Angabe von enthaltenen Aktoren 49
A.6. Allgemeines Beispiel zur Angabe von Aktoren oberster Ebene 49
A.7. Allgemeines Beispiel zur Angabe eines speziellen Pfades 50

B. C++-Code für C2KAOM 51

A. Bedienungsanleitung

Um mit C2KAOM arbeiten zu können, muss als erstes ein Quell- und Zielverzeichnis als Argument übergeben werden. C2KAOM erzeugt dann im Zielverzeichnis für jede annotierte Quelldatei aus dem Quellverzeichnis oder einem Unterverzeichnis davon eine `.kaot`-Datei. Außerdem wird ein Unterverzeichnis *xml* im Zielverzeichnis erzeugt, das wieder gelöscht werden muss.

Allgemeine Angabe von Annotationen: Um Annotationen allgemein anzugeben, müssen diese in einen C-Kommentar vor das entsprechende Objekt gesetzt werden. Dieser Kommentar wird durch `/*!` eingeleitet und mit `*/` abgeschlossen. Vor dem Beginn der einzelnen Inhalte muss einmal das Schlüsselwort `@kaom` geschrieben werden. Danach folgen die einzelnen Inhalte. Ein allgemeines Beispiel einer kompletten Annotation ist in Auflistung A.1 zu sehen.

IDs dürfen nur aus einem bestimmten Zeichensatz bestehen. Das sind alle Zeichen des lateinischen Alphabets in Groß- und Kleinschreibung (`a-z, A-Z`), sowie indische Ziffern in europäischer Schreibweise (`0-9`). Außerdem sind Leerzeichen und Unterstriche (`_`) erlaubt. Andere Zeichen werden intern durch Unterstriche ersetzt. Das betrifft auch die diakritischen Zeichen *ä, ö, ü*, sowie die Ligatur *ß*. Doppelpunkte (`:`) oder rechtsgerichtete Pfeile (`->`) dürfen nicht in IDs geschrieben werden. Genauso dürfen auch die Schlüsselwörter `input:`, `output:`, `link:`, `content:` und `toplevel:` nicht in IDs geschrieben werden. Nach ASCII sind die Zeichen 32, 48-57, 65-90, 95 und 97-122 in IDs erlaubt.

In Bezeichnern dürfen alle Zeichen außer Anführungszeichen, Kommata, Semikola, At-Zeichen, Sternchen mit Backslash und Backslash mit Sternchen (‘‘, `,`, `;`, `@`, `*/`, `/*`) stehen.

Die Objektnamen müssen mit den Objektnamen im Quelltext identisch sein.

Es wird nicht geprüft ob verwendete IDs genutzt oder definiert werden.

Angabe von Eingängen: Die Angabe der Eingänge eines Aktors werden mit dem Schlüsselwort `input:` eingeleitet. Danach folgt die Aufzählung der Eingänge. Für jeden Eintrag muss eine im Aktor eindeutige ID angegeben werden. Optional kann hinter der ID ein beliebiger Bezeichner in Anführungszeichen gesetzt werden. Weitere Einträge werden mit Kommata (`,`) voneinander getrennt. Die Aufzählung aller Eingänge muss mit einem Semikolon (`;`) abgeschlossen werden. Ein allgemeines Beispiel zur Angabe der Eingänge ist in Auflistung A.2 zu sehen.

Angabe von Ausgängen: Die Angabe der Ausgänge eines Aktors werden mit dem Schlüsselwort `output:` eingeleitet. Danach folgt die Aufzählung der Ausgänge. Für jeden

```
/*! @kaom
input:   ID_1 "Bezeichner", ID_2 "Bezeichner", ... ;
output:  ID_1 "Bezeichner", ID_2 "Bezeichner", ... ;
link:    Quelle -> Kontext.Ziel, Kontext.Quelle -> Kontext.
    Ziel, Quelle -> Ziel, ... ;
content:Funktion_1:ID "Bezeichner", Funktion_2:ID "
    Bezeichner", ... ;
toplevel:ID "Bezeichner";
*/
```

Auflistung A.1: Allgemeines Beispiel für eine Annotation

```
input:   ID_1 "Bezeichner", ID_2 "Bezeichner", ... ;
```

Auflistung A.2: Allgemeines Beispiel zur Angabe von Eingängen

Eintrag muss eine im Aktor eindeutige ID angegeben werden. Optional kann hinter der ID ein beliebiger Bezeichner in Anführungszeichen gesetzt werden. Weitere Einträge werden mit Kommata (`,`) voneinander getrennt. Die Aufzählung aller Ausgänge muss mit einem Semikolon (`;`) abgeschlossen werden. Ein allgemeines Beispiel zur Angabe der Ausgänge ist in Auflistung A.3 zu sehen.

Angabe von Datenflüssen: Die Angabe der Datenflüsse eines Aktors werden mit dem Schlüsselwort `link:` eingeleitet. Danach folgt die Aufzählung der Datenflüsse. Für jeden Eintrag muss die ID der Quelle und die ID des Ziels angegeben werden. Quelle und Ziel werden mit einen rechtsgerichteten Pfeil, der die Richtung des Datenflusses angibt, getrennt. Wenn Quelle oder Ziel in einen anderen Aktor als der Datenfluss definiert sind, muss die ID des Aktors von Quelle oder Ziel mit angegeben werden. Die ID des Aktors wird direkt vor die ID von Quelle oder Ziel gesetzt und mit einen Punkt abgetrennt. Aktoren, in denen Quelle und Ziel definiert sind, dürfen sich nicht um mehr als eine Hierarchieebene unterscheiden. Weitere Einträge werden mit Kommata (`,`) voneinander getrennt. Die Aufzählung aller Datenflüsse muss mit einem Semikolon (`;`) abgeschlossen werden. Ein allgemeines Beispiel zur Angabe der Datenflüsse ist in Auflistung A.4 zu sehen.

Angabe von enthaltenen Aktoren: Die Angabe der enthaltenen Aktoren eines Aktors werden mit dem Schlüsselwort `content:` eingeleitet. Danach folgt die Aufzählung der enthaltenen Aktoren. Für jeden Eintrag muss die repräsentierte Funktion im Code durch den Funktionsnamen angegeben werden. Danach folgt eine im Aktor eindeutige

```
output:  ID_1 "Bezeichner", ID_2 "Bezeichner", ... ;
```

Auflistung A.3: Allgemeines Beispiel zur Angabe von Ausgängen

```
link:    Quelle -> Kontext.Ziel, Kontext.Quelle -> Kontext.
    Ziel, Quelle -> Ziel, ... ;
```

Auflistung A.4: Allgemeines Beispiel zur Angabe von Datenflüssen

```
content:Funktion_1:ID "Bezeichner", Funktion_2:ID "
    Bezeichner", ... ;
```

Auflistung A.5: Allgemeines Beispiel zur Angabe von enthaltenen Aktoren

ID, die vom Funktionsnamen mit einem Doppelpunkt abgetrennt wird. Optional kann hinter der ID ein beliebiger Bezeichner in Anführungszeichen gesetzt werden. Weitere Einträge werden mit Kommata (`,`) voneinander getrennt. Die Aufzählung aller enthaltenen Aktoren muss mit einem Semikolon (`;`) abgeschlossen werden. Ein allgemeines Beispiel zur Angabe von enthaltenen Aktoren ist in Auflistung A.5 zu sehen.

Angabe von Aktoren oberster Ebene: Die Angabe der Aktoren oberster Ebene eines Datenflussdiagrammes werden mit dem Schlüsselwort `toplevel:` eingeleitet. Die Angabe muss in die Annotation der repräsentierten Funktion gesetzt werden. Der Eintrag besteht aus einer im Aktor eindeutigen ID. Optional kann hinter der ID ein beliebiger Bezeichner in Anführungszeichen gesetzt werden. Der Eintrag muss mit einem Semikolon (`;`) abgeschlossen werden. Ein allgemeines Beispiel zur Angabe von Aktoren oberster Ebene ist in Auflistung A.6 zu sehen.

Spezialfall Es ist möglich, entgegen den allgemeinen Vorgaben Datenflüsse zu erzwingen. Das ist vor allem dann der Fall, wenn es mehrere Aktoren oberster Eben gibt. Die Angabe von Datenflüsse zwischen solchen Aktoren wird zu den Angaben der anderen Datenflüsse gesetzt. Für die Angabe müssen alle Vorgänger des Aktors und der Aktor selber, in dem Quelle bzw. Ziel definiert sind, durch IDs angegeben werden. Für die Angabe müssen der Pfad zur Quelle und zum Ziel angegeben werden. Beide werden durch einen rechtsgerichteten Pfeil getrennt. Ein Pfad besteht aus der ID der Quelle (bzw. Ziel) und der ID des Aktors, in dem sie definiert ist, sowie den IDs aller textuell umfassenden Aktoren. Die ID des definierenden Aktors wird vor die ID der Quelle (bzw. Ziel) gesetzt und durch einen Doppelpunkt abgetrennt. Die IDs der textuell umfassenden Aktoren werden, absteigend sortiert, vor die ID des definierenden Aktors gesetzt. Die IDs aller Aktoren werden im Pfad durch Unterstriche voneinander getrennt. Quelle und Ziel dürfen sich dabei um mehr als eine Hierarchieebene unterscheiden. Alle Zeichen, die in einer ID durch Unterstriche ersetzt werden, müssen bei der Angabe ebenfalls durch Unterstriche ersetzt werden. Ein allgemeines Beispiel zur Angabe von

```
toplevel:ID "Bezeichner";
```

Auflistung A.6: Allgemeines Beispiel zur Angabe von Aktoren oberster Ebene

```
link:    Quelle -> ID0_ID1_ID2_..._ID:Ziel, ... ;
```
Auflistung A.7: Allgemeines Beispiel zur Angabe eines speziellen Pfades

speziellen Datenflüssen ist in Auflistung A.7 zu sehen.

B. C++-Code für C2KAOM

B.1. Beschreibung

`Main.cpp`: In der Datei `Main.cpp` ist die Funktion `main` beschrieben. Die Funktion `main` realisiert den Ablauf von C2KAOM und erzeugt Instanzen aller anderen Klassen. Sie besteht aus einer Abfrage der Argumente, dem Aufruf von Doxygen und einer `while`-Schleife. Die Argumente enthalten das Quell- und Zielverzeichnis. In der `while`-Schleife werden pugixml und die Synthese der Datenflussdiagramme aufgerufen. Mit der Abfrage, ob der Wert von der Methode `filter` gleich null ist, wird überprüft, ob Annotationen in der Datei gefunden wurden.

`BuilderXml.hh`: In dieser *Header*-Datei sind die Klasse `BuilderXml` definiert und ihre Methoden deklariert. Außerdem sind hier die globalen Variablen deklariert, die zur Ausführung von Doxygen benötigt werden. Zusätzlich sind *Getter* für die privaten Variablen definiert.

`BuilderXml.cpp`: In der Datei `BuilderXml.cpp` sind die Methoden der Klasse `BuilderXml` definiert. Die Klasse `BuilderXml` realisiert den Aufruf von Doxygen und damit die Überführung von Quellcode in XML. Sie besteht aus einem Konstruktor, der bekanntlich genauso heißt wie seine Klasse, und den Methoden `loadDoxyfile`, `fillDoxyfile`, `callDoxygen` und `buildFileQueue`.

Die Methode `loadDoxyfile` lädt die Konfigurationsdatei von Doxygen.

Die Methode `fillDoxyfile` modifiziert die Konfigurationsdatei von Doxygen, indem sie das Quell- und Zielverzeichnis einfügt.

Die Methode `callDoxygen` ruft Doxygen mit der Konfigurationsdatei auf.

Die Methode `buildFileQueue` speichert alle Dateinamen von XML-Dateien in einer *Queue*. Über die Abfrage, ob der Dateiname bestimmte Zeichen enthält, wird sichergestellt, dass keine irrelevanten Dateinamen gespeichert werden.

`FilterXml.hh`: In dieser *Header*-Datei sind die Klasse `FilterXml` definiert und ihre Methoden deklariert. Außerdem sind hier die Variablen deklariert, die zur Ausführung von pugixml und XPath benötigt werden. Zusätzlich gibt es eine globale Variable, über die Daten für allen Instanzen der Klasse `FilterXml` verfügbar gemacht werden. *Getter* für die privaten Variablen sind direkt in der Klasse definiert.

`FilterXml.cpp`: In der Datei `FilterXml.cpp` sind die Methoden der Klasse `FilterXml` definiert. Die Klasse `FilterXml` realisiert die Ausführung von XPath und damit die

Überführung von XML zu den gefilterten Annotationen. Zusätzlich werden Informationen zur Quelldatei und den enthaltenen Objekten gesammelt. Sie besteht aus einem Konstruktor, der bekanntlich genauso heißt wie seine Klasse, und den Methoden `filter`, `loadFile`, `xpathCompound`, `xpathMember` , `xpathtoQueue` , `xpathtoMap` und `buildResult`.

Die Methode `filter` realisiert den grundsätzlichen Ablauf des Filterns. Zuerst werden globale Informationen gefiltert und falls keine gefunden wurden danach lokale Informationen. Über die Abfrage, ob mehr als ein Elemente gefunden wurde, wird sichergestellt, dass mindestens eine Annotation gefunden wurde.

Die Methode `loadFile` lädt das XML-Dokument.

Die Methode `xpathCompound` realisiert die Ausführung von XPath zum Filtern globaler Informationen durch den XPath-Ausdruck, wie er in Kapitel 4.3 beschrieben wurde.

Die Methode `xpathMember` realisiert die Ausführung von XPath zum Filtern lokaler Informationen.

Die Methode `xpathtoQueue` überführt die Ausgabe von XPath in eine *Queue*. Über die `while`-Schleife wird in der globalen *Map* nach allen verweisenden Einträgen gesucht und diese zur Ausgabe hinzugefügt. In der `for`-Schleife werden die restlichen Knoten der Ausgabe lokal abgearbeitet. Durch die enthaltenen Abfragen wird jeweils die entsprechende Kategorie extrahiert und zur Annotation hinzugefügt. Die Annotation wird dann der *Queue* hinzugefügt.

Die Methode `xpathtoMap` extrahiert wie `xpathtoQueue` die entsprechenden Kategorien. Die Kategorien werden zu einer Annotation zusammengesetzt und zu der globalen *Map* hinzugefügt. Über die enthaltenen `for`-Schleifen werden die Inhalte der Kind-Knoten und Enkel-Knoten zu einem Wert zusammengesetzt.

Die Methode `buildResult` extrahiert die relevanten Informationen aus dem Inhalt eines Knoten.

`BuilderKaom.hh`: In dieser *Header*-Datei sind die Klasse `BuilderKaom` definiert und ihre Methoden deklariert. Außerdem sind hier die globalen Variablen deklariert, die zur Synthese der Datenflussdiagramme benötigt werden.

`BuilderKaom.cpp`: In der Datei `BuilderKaom.cpp` sind die Methoden der Klasse `BuilderKaom` definiert. Die Klasse `BuilderKaom` realisiert die Synthese der Datenflussdiagramme als KAOM-Modell. Außerdem realisiert sie, dass die Datenflussdiagramme als Kaot-Dateien im Zielverzeichnis abgespeichert werden. Sie besteht aus einem Konstruktor, der bekanntlich genauso heißt wie seine Klasse, und den Methoden `replaceSpecChar`, `deleteBlank`, `buildPattern`, `buildKaom`, `buildArgsQueue`, `buildResult` und `saveKaom`.

Die Methode `replaceSpecChar` ersetzt alle Sonderzeichen einer Zeichenkette durch Unterstriche. Sie besteht aus einer `for`-Schleife, über die die Zeichenkette abgearbeitet

wird. Die einzelnen Abfragen stellen sicher, dass nur Sonderzeichen nach ASCII-Zeichensatz ersetzt werden.

Die Methode `deleteBlank` löscht aus allen Elementen der *Queue* `input_` Leerzeichen, die vor oder nach einem Komma, Semikolon, Doppelpunkt, Punkt oder rechtsgerichteten Pfeil stehen. Sie besteht aus einer `for`-Schleife und drei `while`-Schleifen, die ineinander verschachtelt sind. Über die `for`-Schleife wird die *Queue* abgearbeitet. Die erste `while`-Schleife realisiert, dass solange gesucht wird, bis in der restlichen Zeichenkette des aktuellen Elements keine gesuchten Zeichen mehr vorkommen. Die beiden anderen `while`-Schleifen realisieren die Suche nach Leerzeichen vor und nach einem gefundenen Zeichen.

Die Methode `buildPattern` überführt die Einträge der einzelnen Kategorien in abstrakten KAOM-Code. Sie besteht aus einer `switch`-Anweisung, über die entsprechend nach Kategorien die Codeauswahl getroffen wird. Die Codeauswahl besteht aus Abfragen, die sicherstellen, dass nur die aktuelle Kategorie verarbeitet wird. Für Kategorien, die eine Aufzählung von Einträgen enthalten können, besteht die Codeauswahl zusätzlich aus einer `while`-Schleife, mit der die Aufzählung abgearbeitet wird. Außerdem wird in der Codeauswahl der Kategorie entsprechend der erzeugte Code definiert und zur aktuellen Vorlage hinzugefügt.

Die Methode `buildKaom` realisiert die eigentliche Synthese des Datenflussdiagrammes als KAOM-Modell. Sie besteht aus zwei `do-while`-Schleifen und einer `while`-Schleife. Über die beiden `do-while`-Schleifen wird das KAOM-Modell iterativ aufgebaut. Die `while`-Schleife ersetzt Maskierungen, die beim Aufbau des KAOM-Modells eingefügt werden. Zusätzlich enthält `buildKaom` eine innere `while`-Schleife, über die aktuelle IDs mit den IDs der Vorgänger verbunden werden. Mit den Abfragen werden die zu ersetzenden Schlüsselwörter selektiert und ersetzt.

Die Methode `buildArgsQueue` ordnet die Kategorien entsprechend ihrer Position in einer Liste. Dadurch kann sichergestellt werden, dass auch beim Fehlen eines Semikolons nicht über das Ende der Kategorie hinaus gelesen wird.

Die Methode `buildResult` realisiert den grundsätzlichen Ablauf der Synthese und Speicherung des Datenflussdiagrammes als KAOM-Modell. Sie besteht aus zwei `while`-Schleifen und den Aufrufen der einzelnen Methoden. In den `while`-Schleifen werden die `buildArgsQueue` und `buildPattern` Methoden für jedes Element der Eingabe aufgerufen.

Die Methode `saveKaom` speichert das KAOM-Modell in einer Kaot-Datei im Zielverzeichnis. Mit der Abfrage, ob Datei und Ergebnis gültig sind, wird sichergestellt ob das Ergebnis in die Datei geschrieben werden kann.

`Templates.hh`: In dieser *Header*-Datei sind *Templates* definiert. Speziell ist hier das *Template* `PairComparator` definiert. Dieses wird benötigt um Paare miteinander zu vergleichen und in geordnete Strukturen zu überführen.

B.2. Der Code

Der Quellcode von C2KAOM kann als Open Source Projekt auf der Internetseite der Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme eingesehen werden [2].