Fortschritt-Berichte VDI

VDI

Reihe 8

Mess-,
Steuerungs- und
Regelungstechnik

Nr. 1268

Dipl.-Ing. Holger Jeromin,
Verl

Explizites Modell für Benutzungsschnittstellen im gesamten Lebenszyklus einer leittechnischen Anlage

ACPLT
AACHENER
PROZESSLEITTECHNIK

Lehrstuhl für
Prozessleittechnik
der RWTH Aachen

"Explizites Modell für Benutzungsschnittstellen im gesamten Lebenszyklus einer leittechnischen Anlage"

Von der Fakultät für Georessourcen und Materialtechnik
der Rheinisch-Westfälischen Technischen Hochschule Aachen

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

genehmigte Dissertation

vorgelegt von **Dipl.-Ing.**

Holger Jeromin

aus Düsseldorf

Berichter: Univ.-Prof. Dr.-Ing. Ulrich Epple
Prof. Dr.-Ing. Leon Urbas

Tag der mündlichen Prüfung: 03. September 2019

Fortschritt-Berichte VDI

Reihe 8

Mess-, Steuerungs- und Regelungstechnik

Dipl.-Ing. Holger Jeromin, Verl

Nr. 1268

Explizites Modell für Benutzungsschnittstellen im gesamten Lebenszyklus einer leittechnischen Anlage

ACPLT
AACHENER PROZESSLEITTECHNIK

Lehrstuhl für Prozessleittechnik der RWTH Aachen

Jeromin, Holger
Explizites Modell für Benutzungsschnittstellen im gesamten Lebenszyklus einer leittechnischen Anlage
Fortschr.-Ber. VDI Reihe 08 Nr. 1268. Düsseldorf: VDI Verlag 2019.
84 Seiten, 25 Bilder, 0 Tabellen.
ISBN 978-3-18-526808-3 ISSN 0178-9546,
€ 38,00/VDI-Mitgliederpreis € 34,20.

Für die Dokumentation: HMI – Human-Machine Interface – Bedienoberflächen – Modellierung – Prozessleittechnik – Prozesstechnik – PLT – Softwaredesign – Automatisierungstechnik

Diese Arbeit schlägt ein neues Konzept für die Erstellung, Wartung und den Gebrauch von Benutzungsschnittstellen für prozesstechnische Anlagen vor. Die gesamte Darstellung wird als HMI-Modell hinterlegt. Dafür wurden nicht nur für alle Grafikelemente (Text, Rechteck, Kreis …), sondern auch für die gesamte Interaktion mit dem Prozess und dem Bediener Modellbausteine (als Metamodellbausteine) definiert. Dies erleichtert die automatische Erstellung und Veränderung der gesamten Darstellung. Dieses Metamodell ist für größte Zukunftssicherheit technologieunabhängig definiert. Um ein solches HMI-Modell einer Anlage darzustellen wird ein Anzeigesystem benötigt, welches die wenigen definierten Metamodellbausteine zur Laufzeit interpretiert. Dieses Anzeigesystem kann bei Bedarf im Laufe der Lebensdauer der technischen Anlage in neuen Technologien implementiert werden.

Bibliographische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Daten sind im Internet unter www.dnb.de abrufbar.

Bibliographic information published by the Deutsche Bibliothek
(German National Library)
The Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliographie (German National Bibliography); detailed bibliographic data is available via Internet at www.dnb.de.

D82 (Diss. RWTH Aachen University, 2019)
Tag der mündlichen Prüfung: 03. September 2019

ISSN 0178-9546

ISBN 978-3-18-526808-3

Vorwort

Die vorliegende Dissertation entstand während meiner Tätigkeit am Lehrstuhl für Prozessleittechnik der RWTH Aachen University. Ich möchte mich an dieser Stelle bei allen bedanken, die geholfen haben, diese Arbeit erfolgreich abzuschließen. Mein besonderer Dank gilt dabei Herrn Professor Dr.-Ing. Ulrich Epple als Doktorvater und auch als Vorgesetzten. Die Gespräche und Diskussionen mit ihm waren von Weitblick und tiefen Einblick in die Automatisierungstechnik geprägt und haben damit maßgeblich zum Erfolg dieser Dissertation beigetragen. Gleichzeitig hat er am Lehrstuhl eine sehr angenehme Arbeitsatmosphäre gepflegt und viel Vertrauen in seine Mitarbeiter gezeigt.

Weiterhin möchte ich mich herzlich bei Prof. Dr.-Ing. Leon Urbas, Inhalber der Professur für Prozessleittechnik an der TU Dresden, bedanken für die Übernahme der Rolle des Zweitgutachters. Sein tiefes Verständnis in der Thematik der Modellierung von Benutzungsschnittstellen hat mir sehr geholfen.

Alle Kollegen des Lehrstuhls haben durch ihre Hilfsbereitschaft und unterschiedliche Expertisen ihren Anteil an dieser Arbeit geleistet. Besonders möchte ich jedoch Stefan Schmitz danken, der mich in seinen Jahren am Lehrstuhl immer unterstützte und die Basis meiner Arbeit am Lehrstuhl legte. Weiterhin möchte ich Lars Evertz danken mit dem ich oft gemeinsam auf der Suche nach der technisch optimalen Lösung war. Auch Tina Mersch lieferte entscheidene Anregungen in meiner Promotion.
Auch bei meinen ehemaligen Studenten Christian Nick und Yannik Rocks möchte ich mich für die konstruktive Mitgestaltung der erstellten Software bedanken.

Schließlich danke ich meiner Familie, angefangen bei meinen Eltern Lutz und Christa die immer an mich glaubten und mich in allen Entscheidungen unterstützen. Weiterhin bedanke ich mich bei meinen Kindern Laura und Vera die mein Leben sehr bereichern. Mein wichtigster Dank gebührt jedoch meiner Frau Sabine, welche mich immer unterstützt und mit unendlicher Geduld motiviert hat die Arbeit zu einem erfolgreichen Ende zu führen.

Verl, im November 2019 *Holger Jeromin*

> Sollen sich auch alle schämen, die gedankenlos sich der Wunder der Wissenschaft und Technik bedienen und nicht mehr davon geistig erfasst haben als die Kuh von der Botanik der Pflanzen, die sie mit Wohlbehagen frisst.

Albert Einstein (Eröffnungsansprache der 7. Großen Deutschen Funkausstellung und Phonoschau, Berlin, Haus der Rundfunkindustrie, 22. August 1930)

Inhaltsverzeichnis

Kurzfassung

Prozesstechnische Anlagen sind sehr komplex und erfordern eine ausgefeilte Steuerung. Leider „nehmen Kompetenz und Qualifikation auf der Anwender- und Bedienerseite ab“. Dies gaben jedenfalls 56 % von rund 1800 befragten Mitglieder im Verband Deutscher Maschinen- und Anlagenbau (VDMA) in einer Umfrage an [Sch12]. Damit Bediener die Steuerung gerade auch in kritischen Situationen bedienen können, ist eine leistungsfähige angepasste Benutzungsschnittstelle nötig. Diese Schnittstellen sind jedoch sehr aufwendig bei der Erstellung.

Um diese Kosten zu senken, bieten sich zwei Möglichkeiten an. Erstens können Kosten eingespart werden, indem möglichst viele Teile der Benutzungsschnittstelle aus vorhandenen Planungsdaten automatisch erstellt werden. Dies hat zudem den Vorteil, dass das endgültige Ergebnis früher bereitsteht. Weiterhin können sich weniger Fehler bei wiederkehrenden Parametrierungsaufgaben bei der Erstellung einschleichen, was insgesamt die Qualität erhöht. Zweitens lassen sich Kosten durch eine möglichst lange Nutzungszeit der Benutzungsschnittstelle reduzieren. In prozesstechnischen Anlagen ist eine Lebensdauer von 30 Jahren nicht ungewöhnlich. Die Steuerungstechnik und erst Recht die Visualisierungstechnologie verwenden jedoch immer mehr Standard-Komponenten der IT-Branche, welche einem schnelleren Wandel unterliegen.

Diese Arbeit schlägt daher ein neues Konzept für die Erstellung, Wartung und den Gebrauch von Benutzungsschnittstellen vor. Die gesamte Darstellung wird als HMI-Modell hinterlegt. Dafür wurden nicht nur für alle Grafikelemente (Text, Rechteck, Kreis . . .), sondern auch für die gesamte Interaktion mit dem Prozess und dem Bediener Modellbausteine (als Metamodellbausteine) definiert. Dies erleichtert die automatische Erstellung und Veränderung der gesamten Darstellung. Dieses Metamodell ist für größte Zukunftssicherheit technologieunabhängig definiert. Für sehr komplexe Aufgaben existiert jedoch zusätzlich eine Erweiterung um per HTML und JavaScript frei zu programmieren. Diese Erweiterung ist dabei so entwickelt worden, dass sie stark verzahnt ist mit der Modellwelt und zwischen beidem ein einfacher Informationsaustausch möglich ist.
Um ein solches HMI-Modell einer Anlage darzustellen wird ein Anzeigesystem benötigt, welches die wenigen definierten Metamodellbausteine zur Laufzeit interpretiert. Dieses Anzeigesystem kann bei Bedarf im Laufe der Lebensdauer der technischen Anlage in neuen Technologien implementiert werden.

Als Prototyp wurde ein Anzeigesystem mit Webtechnologie realisiert. Diese Technologie hat den großen Vorteil, dass für unterschiedlichste Betriebssysteme leistungsfähige Webbrowser existieren. Damit ist der Prototyp selbst plattformunabhängig nutzbar.

Abstract

Process plants are very complex and require a sophisticated control system. Unfortunately "competence and qualification on the user and operator side are decreasing". At any rate, 56 % of around 1800 members surveyed in the German engineering association VDMA gave this result in a survey [Sch12]. In order for operators to be able to operate the plant even in critical situations, a powerful adapted user interface is required. However, these interfaces are very complex to create.

There are two evident ways to reduce these costs. First, costs can be saved by automatically creating as many parts of the user interface as possible from existing planning data. This also has the advantage that the final result is available earlier. Furthermore, fewer errors can creep in during recurring parametrization tasks during creation, which increases overall quality. Secondly, costs can be reduced by using the user interface as long as possible. In process plants, a service life of 30 years is not unusual. The control technology and especially the visualization technology, however, use more and more standard components from the IT industry, which are subject to a faster change.

Therefore, this thesis proposes a new concept for the creation, maintenance and use of user interfaces. The entire representation is stored as an HMI model. Therefore, not only for all graphic elements (text, rectangle, circle ...), but also the entire interaction with the process and the operator model elements (as meta model elements) were defined. This facilitates the automatic creation and modification of the entire representation. This meta model is defined as technology-independent for maximum future security. For very complex tasks, however, there is an additional extension to freely program via HTML and JavaScript. This extension was developed in such a way that it is strongly interlocked with the model world and between both a simple information exchange is possible.
In order to display such an HMI model of a plant, a display system is required that interprets the few defined meta model elements at runtime. If required, this display system can be implemented in new technologies during the life cycle of the technical plant.

As a prototype a display system with web technology was realized. This technology has the great advantage that powerful web browsers exist for all modern operating systems. This means that the prototype itself can be used platform-independently.

1 Einleitung

Im gesamten Lebenszyklus einer technischen Anlage werden verschiedene Benutzungsschnittstellen (oft auch Bedienoberfläche oder Human Machine Interface, kurz HMI genannt) benötigt. Am wichtigsten ist diese Schnittstelle während des Betriebs, da die meisten Informationen über den Zustand der Anlage hierüber abgerufen werden können. Zusätzlich werden alle Eingriffe in den Prozess über diese Schnittstelle vorgenommen. Dies gilt sowohl für den Normalbetrieb, als auch für eine Störung des bestimmungsgemäßen Betriebs. Daher werden Benutzungsschnittstellen aufwändig an den Anwendungszweck sowie an die Wünsche der Anwender angepasst. Nur so ist eine spätere Akzeptanz zu gewährleisten.

Eine Anlage hat teilweise einen jahrzehntelangen Lebenszyklus, in der die Steuerungsaufgabe erfüllt werden muss. In diesem Zeitraum gibt es oft Veränderungen, da die Anlage umgebaut oder erweitert wird. Änderungen, welche über den Austausch baugleicher Bauteile hinausgehen, erfordern dabei meist eine Anpassung der Benutzungsschnittstelle. Solche Eingriffe erfordern Expertenwissen sowohl der Richtlinien, als auch des Prozessleitsystems, da die Anzeigen meist sehr komplex programmiert sind.

Ein anderer Fall bei dem Anpassungen der Benutzungsschnittstelle notwendig werden ist der Austausch des gesamten Leitsystems. Da die Hersteller ihre Bedienoberflächen meist sehr unterschiedlich realisieren, wird dabei teilweise auch eine komplette Neuentwicklung nötig.

Obwohl das Haupteinsatzgebiet für Benutzungsschnittstellen weiterhin die Leitwarte mit dedizierten Computern bleiben wird, werden zusätzliche Anzeige- und Bedienmöglichkeiten immer wichtiger. Der Trend geht aktuell zu mobilen Endgeräten, welche vom Bediener direkt in der Anlage mitgeführt werden können. Weiterhin ist von Anlagenbetreibern oft ein direkter Einblick aufs Prozessleitsystem gewünscht. Da schließlich auf mobilen Geräten oder Office-Computern die komplexe Software zur Anlagensteuerung nicht installiert werden kann beziehungsweise soll, wird hierfür eine separate Zugangstechnologie (zum Beispiel Webtechnologie) benötigt, die installiert und gewartet werden muss.

Parallel zu den Benutzungsschnittstellen der Prozessführung gibt es einen immer größeren Bedarf nach Assistenten und Zusatzwerkzeugen, welche zum Beispiel ein zusätzliches Monitoring oder spezielle Kennzahlen darstellen. Auch diese Softwarewerkzeuge benötigen Benutzungsschnittstellen. Hier ist jedoch eine Installation von viel Zusatzsoftware nicht gewünscht.

Je mehr Zugangsarten (zum Beispiel stationärer und mobiler Bediener, Einblick durch Führungspersonen) genutzt werden, desto aufwendiger sind Änderungen oder Neuentwicklungen der Benutzungsschnittstellen. Dabei sind meist umfassende Programmierkenntnisse erforderlich. Die Benut-

zungsschnittstellen des Prozessleitsystems Honeywell Experion PKS sind zum Beispiel mit JScript und VisualBasic frei programmierbar. Weiterhin werden Kenntnisse von Normen und Richtlinien zur optimalen Gestaltung benötigt; beispielsweise die Richtlinie VDI/VDE 3850 [VDI02] für die Fertigungsindustrie oder die Richtlinie VDI/VDE 3699 [VDI13] für die Prozessindustrie).

All diese Herausforderungen machen klar, dass die Erstellung und Pflege von Benutzungsschnittstellen sehr aufwändig und damit teuer ist. Da die Wartung bei den HMI-Herstellern unterschiedlich komplex ist, hat die Festlegung auf einen Hersteller einen großen Einfluss auf den späteren Betrieb.

Ein herstellerunabhängiger Ansatz für Benutzungsschnittstellen wurde bereits am Lehrstuhl unter der Bezeichnung ACPLT/HMI [Sch10] entwickelt. Er erfordert jedoch für nicht triviale Aufgabenstellungen eine Programmierung in Hochsprache, sodass jeder Änderungswunsch eine Änderung im C-Code notwendig macht.

Eine Alternative wäre eine vollständige Modellierung von Benutzungsschnittstellen inklusive der Interaktion. Dabei sollten Änderungen zur Laufzeit ohne tiefe Programmiererfahrung einfach für die Benutzer zu realisieren sein. Durch eine technologieneutrale Definition könnte der Wert der Benutzungsschnittstelle im gesamten Lebenszyklus gesichert werden, da bei einem Plattformwechsel die bisherige Applikation ohne teure Neuimplementierung übernommen werden kann.

Die vorliegende Arbeit analysiert daher einen neuen Ansatz eines expliziten Modells zur Beschreibung einer Benutzungsschnittstelle, welche sich aus wenigen, vorher definierten Elementarbausteinen zusammensetzt (siehe Abbildung 1.1). Ziel ist es zu zeigen, dass dieses Konzept für viele unterschiedliche Benutzungsschnittstellen im gesamten Lebenszyklus einer leittechnischen Anlage nutzbar ist und die Wartung vereinfacht wird.

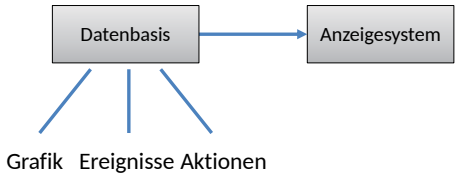

Abbildung 1.1: Grundkonzept der vorliegenden Arbeit

Zu Beginn der Arbeit beschäftigt sich Kapitel 2 mit dem aktuellen Stand der Technik von Benutzungsschnittstellen. Es werden verschiedene Standards aus der IT und Automatisierungstechnik zur Modellierung von Bedienoberflächen vorgestellt. Anschließend geht Kapitel 3 auf das Konzept der vorgestellten Lösung ein, bevor dessen prototypische Implementierung in Kapitel 4 detailliert vorgestellt wird. Daraufhin wird in Kapitel 5 durch Beispielanwendungen geprüft, ob das vorgestellte Modell für unterschiedlich komplexe Benutzungsschnittstellen nutzbar ist. Die Arbeit schließt mit einer Diskussion und Ideen für zukünftige Forschungsgebiete in Kapitel 6 ab.

2 Hintergrund und Motivation

Wie in der Einleitung erwähnt hat eine leittechnische Anlage teilweise eine jahrzehntelange Laufzeit. Die Entität der Anlage beginnt jedoch schon ab der Planung und endet mit dem Rückbau oder anderweitigen Nutzung der physischen Anlage. Dieser Zeitraum wird in DIN 40912 [DIN14] als Lebenszyklus bezeichnet und definiert als „die Folge von Prozessen, die eine Entität während ihrer Existenz durchläuft.“. Diese Prozesse sind nicht eingeschränkt auf die Nutzzeit der Anlage, da selbst die Alterung als Prozess angesehen werden kann.

Hier seien exemplarisch einige Phasen des Lebenszyklus aufgelistet:

- Konzeption
- Planung
- Errichtung
- Inbetriebnahme
- Betrieb und Nutzung
- Umbau/Umrüstung
- Rückbau

Innerhalb der Phasen des Lebenszyklus werden unterschiedliche Bedienoberflächen eingesetzt. So wird in der Planungsphase beispielsweise das verfahrenstechnische Fließbild und der Elektroplan erstellt. Zur Erstellung beider Pläne wird in der Industrie meist jeweils ein gesondertes Werkzeug genutzt. Diese sind teilweise mit unterschiedlichen Technologien realisiert und haben damit unterschiedliche Bedien- und Gestaltungskonzepte.

Die Bedienoberfläche, welche während des Betriebs genutzt wird, hat noch eine größere Bedeutung, da sie von weniger qualifizierten Bedienern und täglich viele Stunden genutzt wird. Durch die jahrzehntelange Lebensdauer ist die softwaretechnische Realisierung von großer Bedeutung. Die richtige Wahl der Softwareinfrastruktur sorgt dafür, dass eine erfolgreiche Bedienoberfläche über diese lange Zeit kosteneffizient nutzbar bleibt.

Auf dem Markt der Prozessleittechnik und erst recht allgemein der Computertechnik haben sich unterschiedliche Ansätze zur Modellierung von Bedienoberflächen ausgebildet. Daher sollen in

diesem Kapitel einige wichtige vorhandenen Modellierungstechniken aus diesen beiden Domänen vorgestellt werden.

Der Hauptfokus in diesem Kapitel liegt in der Modellierung der allgemeinen Darstellung und der Interaktion.

2.1 Stand der Technik

Modelle von Bedienoberflächen gibt es von verschiedenen Herstellern und Forschungseinrichtungen. Daher werden in den nachfolgenden Unterkapiteln einige Standards von Industriesteuerungs-HMI und angrenzender Standards vorgestellt und deren Eigenschaften in Bezug auf automatische Erstellung und allgemein der Anwendungsmöglichkeiten in der Prozesstechnik geprüft. Zusätzlich werden erfolgreiche Modelle aus dem Konsumerbereich vorgestellt um einige Ansätze auf deren Eignung in der Prozesstechnik zu analysieren. Die weiteren vorgestellten Modelle stammen aus dem Bereich der Industrie.

Auch mobile Geräteklassen wie Mobiltelefone oder Tablets haben eine Bedienschnittstelle. Diese haben einen anderen Anwenderkreis und insbesondere andere Aufgaben als die Anzeigen in der Prozessleittechnik. Trotzdem wird hier die sehr erfolgreich genutzte Technik kurz vorgestellt.

2.1.1 iPhone/Android Programierung

Android von Google ist das führende Betriebssystem für Smartphones und Tablets.[1] Es ist seit seiner Veröffentlichung 2008 auf Touchbedienung und unterschiedliche Gerätemodelle ausgelegt. Die Anwendungen (Apps genannt) müssen mit einer Vielzahl von Bildschirmgrößen und Pixelanzahl nutzbar sein, daher sind die Bedienbilder meist relativ zur Bildschirmdimension definiert. Die Breite eines Knopfes ist zum Beispiel halb so breit wie der Bildschirm und in der Mitte positioniert.

Die Darstellung aller solcher auf dem Bildschirm sichtbaren Objekte wird über zwei Basisklassen realisiert. Ein *View* Objekt bietet ein sichtbares Objekt, eine *ViewGroup* ist dagegen nur ein (unsichtbarer) Container. Ein Element der Klasse View kann keine weiteren Elemente aufnehmen, eine ViewGroup kann jedoch beliebig viele View und ViewGroup Elemente aufnehmen. Aus dieser Verschachtelung wird daraus eine Baumstruktur. Die ViewGroup ist nur eine Basisklasse, welche über verschiedene Layouts realisiert wird. So gibt es zum Beispiel LinearLayout für hintereinander gehängte Views (in einer langen Spalte oder Reihe), RelativeLayout für relativ zueinander positionierbare Views (Positionierung ist relativ zum Vater- oder Nachbarelement möglich). Auch komplexe, fertig zur Verfügung gestellte Komponenten wie eine vollwertige Anzeige für HTML Inhalte wie WebView sind als ViewGroup realisiert.

[1] Marktanteil 86,1 % Quelle Gartner, Stand Mai 2017 `https://www.gartner.com/newsroom/id/3725117` (abgerufen am 28.7.2018)

Es existieren zwei Möglichkeiten eine Darstellung in Android zu realisieren. Das Layout kann vollständig und statisch in einem XML-Dokument definiert werden. Alternativ können alle Elemente (also View- und ViewGroup-Objekte) einzeln per Programmcode erzeugt werden und so die Applikation dynamisch aufgebaut werden.[2] Die XML-Datei verwendet als XML-Namensraum die URI `http://schemas.android.com/apk/res/android`, es ist jedoch kein formales Schema für diese XML-Dateien verfügbar. Dies liegt daran, dass das erstellte XML Abhängigkeiten zu beliebigen Fremdbibliotheken hat, welche nicht in einer zentralen Schemadatei erfasst werden können.

Alle Interaktion wird über Java-Programmcode definiert. So hat jedes View oder ViewGroup Element eine Entsprechung in einem Java-Objekt auf welchem sogenannte Listener (wie OnItemClickListener oder OnItemLongClickListener) registriert werden können. Dieser Java-Programmcode wird daraufhin bei einem einfachen (bzw. langem) Klick auf dieses Objekt aufgerufen.

Der größte Konkurrent von Android ist das Mobilbetriebssystem iOS von Apple, welches auf mobilen Apple Geräten wie iPhone und iPad läuft.[3] Die Programmierung erfolgt ähnlich wie bei Android mit dem Unterschied, dass eine manuelle Zusammenstellung der Anzeige meist nicht erfolgt. Stattdessen wird auf die umfangreiche Hilfe des *Interface Builders* zurückgegriffen. Weiterhin werden hier die einzelnen Ansichten (Scenes genannt) nicht voneinander unabhängig erstellt, sondern sie bilden eine Einheit unter dem Dach eines *Storyboards*. Sie werden mit sogenannten Segues verknüpft. Dies sind festgelegte Übergänge zu anderen Scenes bei der Benutzung einer Schaltfläche. Diese Art der High-Level Verknüpfung verringert die Zahl der frei programmierten Logik.[4] Da es keine Dokumentation über die dahinterliegenden Datenmodelle gibt und weiterhin eine Entwicklung von iOS Applikationen nur auf macOS Computern möglich ist, ist eine Übertragung der Modellierung dieser Programmierung in die Prozesstechnik nicht sinnvoll.

2.1.2 Field Device Tool/Device Type Manager (FDT/DTM)

Das offene System Field Device Tool/Device Type Manager hat es sich zur Aufgabe gemacht eine herstellerunabhängige Konfiguration und Parametrierung von Feldgeräten zu ermöglichen. Mit diesem Konzept muss ein Gerätehersteller keine eigene vollständige Software erstellen, sondern liefert eine Device Type Manager-Datei (DTM), welche von einem Interpreter (der FDT Rahmenapplikation) dem Benutzer präsentiert wird (siehe Abbildung 2.1). Diese Applikation kann ein separates Tool (zum Beispiel das kostenlose PACTware[5]) sein oder in ein Leitsystem (wie beispielsweise das System 800xA von ABB) integriert sein.

[2] Android Entwickler Dokumentation: `https://developer.android.com/guide/topics/ui/declaring-layout` (abgerufen am 28.7.2018)

[3] Marktanteil 13,7 % Quelle Gartner, Stand Mai 2017 https://www.gartner.com/newsroom/id/3725117 (abgerufen am 28.7.2018)

[4] iOS Entwickler Dokumentation `https://developer.apple.com/xcode/interface-builder/` (abgerufen am 27.7.2018)

[5] `http://www.pactware.com/` (abgerufen 27.7.2018)

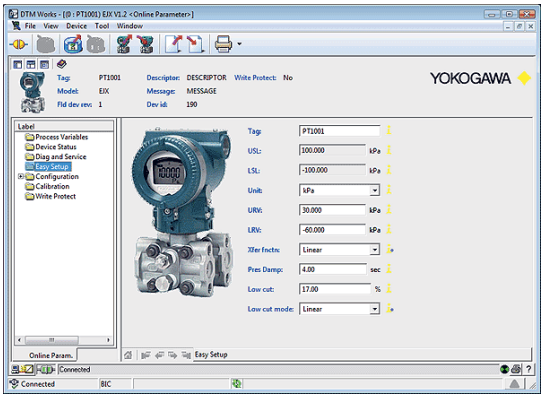

Abbildung 2.1: FDT/DTM Konfiguration eines Yokogawa EJX110A Differenzdruck-Messumformer ©Yokogawa

Das System ist gedacht als eine Schnittstelle für Techniker und nicht als Visualisierung für einen vollständigen Prozess[6]. Die Darstellung innerhalb der DTM-Dateien kann als reine grafische Darstellung von Gerätebeschreibungsdateien, Electronic Device Description (EDD), ausgeführt sein oder als komplexe Applikation. Solch eine Applikation wird mithilfe der .NET-Technologie von Microsoft (siehe Kapitel 2.1.9.2) implementiert und ist daher aktuell auf anderen Betriebssystemen als Windows oder Mobilgeräten nur über Proxylösungen wie Remote FDT Server[7] von der M&M Software GmbH möglich. Einheitliches Aussehen wird über einen DTM-Style Guide erreicht, welcher auch für eine Zertifizierung eingehalten werden muss. Eine eigene Modellierung einer grafischen Bedienoberfläche für den Industrieeinsatz bietet FDT/DTM daher nicht.

2.1.3 Siemens SIMATIC WinCC, Honeywell Experion PKS

In diesem Abschnitt werden zwei HMI Systeme aus dem Bereich der Prozessindustrie vorgestellt, welche in den Grundzügen eine identische Philosophie verfolgen. Sowohl die Bedienoberfläche von Siemens SIMATIC WinCC als auch das Honeywell Experion PKS bieten eine Möglichkeit einer dynamischen Anpassung der Darstellung [Sie13, Hon14]. Beide liefern einen grafischen Editor zur einfachen Erstellung. Die Dynamik ist bei beiden Herstellern nur über eine Freitext-Programmierung möglich. Dies erfolgt bei WinCC unter dem Stichwort *Dynamisierungen* über einen „Dynamik-Dialog“ (grafisches Werkzeug um eine einfache WENN/DANN Logik zu programmieren), ANSI-C oder Visual Basic Script (VBS) und bei Experion per Visual Basic Script oder JavaScript.

[6] `http://www.abb.com/cawp/seitp202/847374139ddb1f73c1257dd9004b1740.aspx` - ABB präsentiert das erste FDI-gestützte Gerätemanagement-Tool (abgerufen 27.7.2018)
[7] `https://mm-software.com/de/fdt-services` (abgerufen 27.7.2018)

2.1.4 Beckhoff TwinCAT 3 HMI

TwinCAT 3 HMI (TcHmi) von Beckhoff Automation ist eine Bedienoberfläche hauptsächlich für Maschinensteuerungen. Erstellt wird die Oberfläche grafisch über eine Extension von Beckhoff in der Entwicklungsumgebung Visual Studio von Microsoft. Dargestellt wird die Bedienoberfläche per Webbrowser über eine Webseite mit JavaScript. Alle darzustellenden Elemente werden zu eigenständigen Einheiten (Controls) wie zum Beispiel Container, Kreis, Rechteck, Knopf, Auswahlliste zusammengefasst. Auch eigene Controls können per HTML und JavaScript erstellt werden.

Die konkrete Nutzung und Verschachtelung der gewünschten Controls einer Anzeige werden in einer HTML-ähnlichen Beschreibungssprache gespeichert. Dabei wird jedoch nur der generische Container von HTML (`<div></div>`) genutzt und mit TwinCAT spezifischen Attributen parametriert. Das HTML-Attribut `data-tchmi-type="tchmi-button"` legt beispielsweise fest, dass in der späteren Darstellung dieser generische `<div>`-Container durch einen Button ersetzt werden soll. Weitere HTML-Attribute legen sowohl die Position sowie Größe der Controls, als auch die Interaktion mit dem Bediener oder den Prozess-/Maschinendaten fest.

Es gibt bei allen Controls sehr viele Attribute zur Anpassung der Darstellung. Die Definition dieser Attribute lehnt sich sehr stark an die genutzte Darstellungsplatform HTML an. Der Fokus liegt auf maximaler Flexibilität und Erweiterbarkeit durch den Anwender.[8]

2.1.5 ACPLT/HMI

Am Lehrstuhl wurde schon vor dieser Arbeit mit dem modellbasierten Ansatz ACPLT/HMI [SE07, Sch10] gearbeitet. Hierbei liefert jede Komponente (Bausteintypicals genannt) seine aktuelle Darstellung als Scalable Vector Graphics (SVG, [FJF03]). Diese Darstellung wird im Automatisierungssystem generiert und zyklisch vom Anzeigesystem neu abgefragt und dargestellt (siehe Abbildung 2.2).

Die Anzeige wird ergänzt um Interaktions-Hinweise. Diese werden interpretiert und somit werden unter anderem Klick, Doppelklick, Texteingabe und Drag&Drop ermöglicht. Das Anzeigesystem ist jedoch extra simpel gehalten und gibt eine erkannte Interaktion (genannt Gesten, beispielsweise ein Klick) nur an das Automatisierungssystem weiter. Für die wirkliche Aktion der Gesten muss dort eine Programmierung hinterlegt worden sein. So kann eine beliebig komplexe Reaktion auf diese Interaktion durchgeführt und, bei Bedarf, die Darstellung geändert werden.

Da bisher keine Sitzungen verwaltet werden, zeigen alle Anzeigesysteme stets das exakt gleiche Bild an. Als Technologiedemo wurden alle Basisformen von SVG wie Rechteck, Kreis, Text und weitere erstellt. Auch Gesten wie ein Farbwechsel oder eine Positionsänderung wurden implementiert. Beides kann einfach im Automatisierungssystem instanziiert werden. Es existiert jedoch keine Beschreibungssprache für die Darstellung oder Gesten. Jede nicht triviale Aufgabenstellungen erfordert daher auch eine Programmierung in Hochsprache.

[8] `http://beckhoff.de/te2000/` (abgerufen 27.7.2018)

Die Anzeige der SVG-Darstellung ist webbasiert und bietet daher eine Plattformunabhängigkeit [Jer08] für unterschiedliche Geräteklassen (Mobil, Desktop) ohne dass eine Installation auf den Endgeräten nötig wäre.

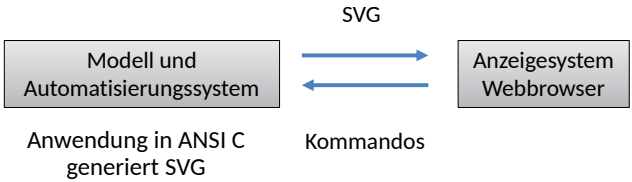

Abbildung 2.2: Grundstruktur von ACPLT/HMI

2.1.6 NAMUR Module Type Package

Das Konzept „DIMA - Dezentrale Intelligenz für modulare Anlagen“ wurde von der WAGO Kontakttechnik GmbH & Co.KG zusammen mit der Professur für Automatisierungstechnik, Helmut-Schmidt-Universität Hamburg und der Professur für Prozessleittechnik, Technischen Universität Dresden entwickelt, um eine vollständige Automatisierung von modularen Anlagen zu ermöglichen.[HLW+16] Das Konzept wird aktuell als NAMUR Empfehlung (VDI/VDE/NAMUR 2658) standardisiert.[BHH+16]

Diese Module werden jeweils im Format Module Type Package (MTP) beschrieben, welche die Prozedursteuerung, Control-, I/O-Ebene und auch die Visualisierung beinhaltet. Als Beschreibungssprache für die Bedienoberfläche wird mittlerweile AutomationML[IEC10b] genutzt.

Die Visualisierung wird bei jeder Änderung der Anlagentopologie neu zusammengestellt. Um ein einheitliches Aussehen bei unterschiedlichen Herstellerkomponenten zu gewährleisten, arbeitet MTP mit Referenzen der Klassifizierungsbibliothek eCl@ss (inklusive einer Position und Rotation auf dem Bildschirm).[OHU+15] Diese werden daraufhin vom endgültigen Programm auf dem Bildschirm erstellt, wobei alle genutzten Rollen dem System bekannt sein müssen um die Module fehlerfrei darstellen zu können. Die Datenverbindung der Module werden über eine Liste von Datenpunkten des Moduls festgelegt. Somit ist eine Interaktion mit dem Prozess möglich.

2.1.7 automotiveHMI

Das Format *automotiveHMI* ist ein domänenspezifisches Austauschformat für die (verteilte) Entwicklung von Infotainmentsystemen im Automobilbereich. Es wurde finanziert vom Bundesministerium für Wirtschaft und Technologie (BMWi) als Verbundprojekt unter der Koordination des Deutschen Forschungszentrums für Künstliche Intelligenz (DFKI) in Kaiserslautern[9] mit elf Automobilherstellern und Zulieferern. Die Spezifikation steht frei unter der MIT-Lizenz (kommerzielle Nutzung

[9] `https://www.dfki.de/web/news/detail/News/projekt-automotive-hmi-austauschformat/` (abgerufen 27.7.2018)

möglich) als Download[10] zur Verfügung. Es ist modellbasiert und bietet im Sprachkern eine integrierte Versionierung.

Durch die Beschränkung auf ein Infotainmentsystem ist die hauptsächliche Interaktionsform dialoggestützt. Von einem Hauptbildschirm wird über Transitionen zu unterschiedlichen Dialogen gewechselt. Diese Transitionen werden über eine einfache Zustandsmaschine gesteuert und werden über Ereignisse sowie Konditionen (*guardCondition*) definiert.

automotiveHMI spezifiziert auch Pop-ups im Kern, welche andere Anzeigen überlagern können. Diese Pop-ups sind priorisierbar, so kann ein Pop-up für eine Ölstandwarnung eine Warnung zur niedrigen Außentemperatur überlagern.

Weiterhin bietet der Standard auch ein Templatesystem zur Wiederverwendung von beliebigen Teilobjekten.

2.1.8 MOVISA

Ein sehr interessantes HMI-Modell ist MOVISA von Stefan Henning [HB11, Hen12]. Es wird aktuell auch kommerziell genutzt in der „MONKEY WORKS Suite“ der ELCO Industrie Automation GmbH. Das HMI-Modell der Applikation wird vom Anwender über eine abstrakte Modellierungsschnittstelle (high fidelity) programmiert, bietet für automatische Engineeringaufgabe jedoch auch einen direkten Zugriff auf die Interna (low fidelity). Eine Möglichkeit der Verifikation auf diesen Modellen ist direkt integriert. Die Verifikationsregeln können generisch sein, lassen sich über Java-Programme jedoch auch vom Kunden erweitern.

Aus diesen Modellen wird daraufhin in einem separaten Schritt die gewünschte native Anwendung generiert. Diese Transition kann auch Mobilgeräte als Ziel haben. Für die unterschiedlichen Anforderungen der Eingabegeräte und Bildschirmgrößen können zusätzliche Vorschriften für die Transition definiert werden. So können Teile der Darstellung für Mobilgeräte beispielsweise versteckt werden.

Das Modell selbst ist eine „Domain Specific Language“ für Produktionsautomatisierung. Sie beschreibt neben einfachen grafischen Objekten, auch viele sogenannte „Common interaction widgets“. Darunter fallen: Buttons, Slider, Combobox, Listbox, Checkbox, Radiobuttons, Eingabefelder. Weiterhin sind sogar domänenspezifische Elemente als „Automation specific widgets“ definiert: Dies ist beispielsweise ein „Alarm control widget“, Trend Chart und ein Drehzeigerdiagramm. Auch sehr komplexe Darstellungselemente wie eine Tabelle und eine Baumansicht sind als „Complex widgets“ beschrieben.

Eine Besonderheit des MOVISA-Modells ist die explizite Modellierung der Interaktion außerhalb von freier Textprogrammierung. So ist das Lesen und Schreiben von Prozessdaten direkt über eine grafische Notation möglich. Diese wird jedoch ergänzt durch eine textuelle Syntax.

[10] `https://sourceforge.net/projects/automotivehmi/` (abgerufen am 27.7.2018)

2.1.9 IT HMI Standards

Da die Modellierung der Bedienoberfläche im Fokus dieser Arbeit ist, werden im Folgenden einige Standards zu Bedienoberflächen aus der Informationstechnologie beschrieben.

2.1.9.1 OpenLaszlo und Apache Flex

Die beiden Produkte Apache Flex (ehemals Adobe Flex)[11] und das ältere OpenLaszlo[12] sind konzeptionell ähnlich aufgestellt. Beide definieren die reine Bedienoberfläche in einem eigenen XML-Dialekt (*MXML* bei Flex beziehungsweise *LZX* bei OpenLaszlo). Die Anwendungslogik wird mithilfe an ECMAScript angelehnter Programmierung implementiert.

Aus diesen Ressourcen wird eine Binärdatei erstellt, welche vom Browserplugin Adobe Flash beim Anwender auf den Bildschirm dargestellt wird. Bei OpenLaszlo ist zusätzlich eine Webseite ohne Pluginbenutzung als Export vorgesehen.

Durch die Nutzung einer Freitextprogrammiersprache mit spezieller API zu den Grafikelementen ist eine Technologieunabhängigkeit nicht gegeben.

2.1.9.2 Extensible Application Markup Language (XAML)

Die deklarative Sprache Extensible Application Markup Language (XAML) wurde von Microsoft im Jahre 2006[13] für das Grafikframework .NET entwickelt. XAML selbst definiert nur die statische Darstellung. Die Programmlogik wird in einer klassischen imperativen Programmiersprache wie `C#`, `Visual Basic` oder auch `JavaScript` implementiert.

Eine große Stärke von XAML sind die umfangreichen Steuerelemente (Controls) und die sehr gute Toolunterstützung (Visual Studio und Microsoft Blend) welche die Entwicklung der Applikation sehr beschleunigen.

XAML kann in fünf unterschiedlichen Microsoft Architekturen genutzt werden. Dies sind die *Windows Presentation Foundation* (WPF, für Windows-Desktop-Anwendungen), *Universal Windows Platform* (UWP, für Windows 10 Anwendungen), *Silverlight* für Windows Phone Anwendungen, *Silverlight* innerhalb eines Webbrowser-Plug-in und letztendlich noch für iOS, Android und Windows Phone der Sprachdialekt *Xamarin Forms*. Alle diese Architekturen bringen einen unterschiedlichen Satz an Steuerelementen mit. Dies hat zur Folge, dass eine per XAML erstellte Anwendung nicht ohne weiteres auf anderen Plattformen läuft. Eine Initiative von Microsoft eine einheitliche Definition unter dem Namen `XAML Standard` läuft nur sehr schleppend an.[14]

[11] `http://flex.apache.org` (abgerufen am 27.7.2018)
[12] `http://www.openlaszlo.org` (abgerufen am 27.7.2018)
[13] `http://download.microsoft.com/download/0/A/6/0A6F7755-9AF5-448B-907D-13985ACCF53E/%5BMS-XAML%5D.pdf` Xaml Object Mapping Specification 2006 (PDF), Microsoft, June 2006
[14] Holger Schwichtenberg, heise developer, „Kommentar: Kann Microsoft mit XAML Standard die Abwanderung von Entwicklern stoppen?“ `https://heise.de/-3712263` (abgerufen am 27.7.2018)

2.1.9.3 XML User Interface Language (XUL)

Die Beschreibungssprache *XML User Interface Language* wurde vom Mozilla-Projekt entwickelt, um eine betriebssystemunabhängige Beschreibung für die Bedienoberflächen des Browsers Mozilla zu erhalten. Zur Unterstützung eines Betriebssystems muss nur der XUL Interpreter angepasst werden.

XUL beschreibt die Darstellung ausschließlich mit hoher Abstraktion mit Hilfe von sogenannten Controls. Diese Controls werden per Cascading Style Sheets (CSS) an das Aussehen des Betriebssystems angepasst. Zur Auswahl stehen beispielsweise Button, Checkboxen, Datumswähler, Listen und Texteingabefelder. Es fehlen jedoch Basiselemente wie Kreis/Ellipse, Rechteck da diese komplett freie Darstellung für die Bedienoberfläche der Software nicht benötigt wird.

Die Interaktion mit den grafischen Elementen erfolgt wie bei einer Webseite ausschließlich durch freie Programmierung durch JavaScript.

2.1.9.4 USer Interface eXtensible Markup Language (UsiXML)

Das USer Interface eXtensible Markup Language (UsiXML) erhebt den Anspruch nicht nur das endgültige Aussehen einer Bedienoberfläche zu modellieren, sondern die gesamte Entwicklung dieser. So definiert dieses Format vier Abstraktionslevel.

„Task & Concepts“ auf der höchsten Ebene beschreibt hier beispielsweise die Aufgabe eine Datei über einen Trigger herunterzuladen. Die nächstniedrigere Ebene „Abstract User Interface (AUI)“ legt fest, dass hierfür ein Bedienelement gebraucht wird. Dieses wird in der Ebene „Concrete User Interface (CUI)“ beispielsweise mit drei Möglichkeiten implementiert: Hardware-Taster, 2D-Button mit einem normalen Bildschirm oder 3D-Button innerhalb einer Virtuellen Realität. Erst im „Final User Interface (FUI) wird daraus ein Knopf einer Webseite oder eines nativen Windows- oder Linuxprogramms.

Durch diese Beschreibungsschichten kann mit dem Format eine extrem große Vielfalt von Plattformen (beispielsweise Telefon, Tablet, Kiosk, Laptop, Desktop), Interaktionsmodi (Maus, Touch-Bildschirm, Tastatur, Spracheingabe) und sowohl grafische Interaktion, Sprachinteraktion, 3D Interaktion oder auch Interaktion innerhalb von Virtual Reality beschrieben werden.

UsiXML erlaubt sowohl die Verallgemeinerung als auch die Spezialisierung per Graphtransformation vorzunehmen und so beispielsweise (einmal Verallgemeinerung und wiederum Spezialisierung in eine andere Richtung) aus einer konkreten Bedienoberfläche eines Desktop-Computers eine Bedienoberfläche für mobile Endgeräte zu generieren. Diese Transformationen müssen vom Anwender meist selbst definiert werden. Sie werden jedoch für eine weitere Benutzung gespeichert, sodass eine Änderung auf einer Abstraktionsebene in die anderen Ebenen überführt werden kann.[LVM+05]

Alle Interaktion wird über ein Task Modell beschrieben. Dieses ist wie die Grafikbeschreibung in verschiedenen Abstraktionen unterteilt. So ist eine allgemeine Beschreibung beispielsweise die Aufgabe „Erfassung von Bestellungen“, welche weiter spezifiziert wird zu „Kundendaten erfassen / Kundendatensatz aufrufen, Liste der Produkte und danach Versandart und Bezahlarten erfragen“. Die Unteraufgaben können weiterhin für eine gute Benutzerführung über verschiedene Realisierungen laufen, also Suche über ID, Name oder Adresse.[Pri06]

2.1.9.5 QML

QML wurde von Nokia im Jahre 2010 als universelle Beschreibungssprache für Mobil- und Desktop-Anwendungen innerhalb der Qt Infrastruktur vorgestellt. Die Entwicklung wird mittlerweile von *The Qt Company*[15] weiter betrieben.

QML erlaubt eine Kombination aus deklarativer und imperativer Beschreibung einer Bedienoberfläche. Jedes grafische Element wird über eine einfache Textsyntax hierarchisch beschrieben.

Deklarativ ist beispielsweise eine Kopplung der Breite mit der Höhe über `height: 2 * width` möglich. Jede Änderung der Breite führt so automatisch zu einer Aktualisierung der Höhe. Auch eine rein imperative Programmierung per JavaScript ist möglich.

Da die meisten QML Visualisierungen gemeinsam mit einem Programmkern in C++ benutzt werden ist eine direkte Koppelung des Qt Eventsystems (Signal & Slot) möglich.[16] Somit wird eine Reaktion auf Benutzereingaben über JavaScript oder C++ realisiert.

2.2 Gemeinsamkeiten und allgemeine Struktur von Bedienoberflächen

Auch wenn die einzelnen vorgestellten Systeme sehr unterschiedlich sind, so lassen sich immer wiederkehrende Komponenten erkennen. So gibt es vielfältige technische Lösungen eine Bedienoberfläche zu modellieren, zu speichern und auf einem Bildschirm aufzubauen. In modernen Modellierungstechniken wird die gewünschte Applikation in Einzelelemente zerlegt. Dieser Vorgang kann je nach Zielsetzung unterschiedlich weit gehen.

So kann beispielsweise eine einfache Applikation nur aus komplexen, fertigen Komponenten zusammengesetzt werden. In diesem Fall ist die Entwicklung der Gesamtlösung schneller möglich, die Flexibilität jedoch eingeschränkt. Eine Möglichkeit diese wieder zu erhöhen ist eine Parametrierbarkeit der Teilkomponenten. So kann der Anwendungsentwickler im gewissen Rahmen Einfluss auf das spätere Aussehen und/oder die Funktionalität nehmen.

[15] `https://www.qt.io/` (abgerufen am 27.7.2018)
[16] `http://doc.qt.io/qt-5/qtqml-syntax-signals.html` (abgerufen am 27.7.2018)

Volle Kontrolle hat der Entwickler im entgegengesetzten Extrem. Hierbei wird die Auftrennung bis herunter zu den Grundformen (Text, Kreis, Rechteck ...) getrieben. Hier ist alles auf den Anwendungszweck abstimmbar. Der Nachteil ist jedoch eine wesentlich aufwändigere Entwicklung.

Die gleiche Bandbreite der Abstraktion ist auch bei der Modellierung der Interaktionsmöglichkeiten (Beispiel: Verhalten nach einem Klick ...) zu finden. Hier kann eine komplexe, festgelegte Interaktion hinterlegt sein oder der Anwendungsentwickler muss die gesamte Logik selbst implementieren.

In der Praxis wird meist ein Mittelweg genutzt. So werden mehrfach genutzte Komponenten wiederverwendet und der Rest einmalig implementiert.

Ist die gewünschte Modellierungstiefe festgelegt, so muss die modellierte Bedienoberfläche anschließend gespeichert werden. Hierzu gibt es sehr viele Möglichkeiten, deren Wahl jedoch wesentlich weniger Einfluss auf Flexibilität hat als die Modellierung selbst.

Ansätze der modellgetriebenen Architektur erlauben es eine Software nicht in Freitext-Quelltext zu pflegen, sondern die Logik als abstraktes und vor allem zugreifbares Modell zu hinterlegen. Dies ermöglicht es beispielsweise Teile der späteren Software automatisch zu generieren. Diese Generierung kann endgültig sein oder auch nur als eine Art Rohfassung zur späteren händischen Optimierung dienen.

Diese Modelle werden danach in normalen Code transformiert um sie auf dem Zielsystem nutzen zu können.[MPV11]

2.3 Automatische Erstellung von Bedienoberflächen

Im Bereich der Prozesstechnik gibt es viele Teile der Darstellung welche mehrfach vorkommen. Die Anzahl der Prozessbedienbildern einer durchschnittlichen Chemieanlage beträgt 130-500 welche 2500 bis 7500 EMSR (Elektrisches Messen, Steuern, Regeln)-Stellen darstellen.[Kir07] So benötigt beispielsweise jede Pumpe ein Repräsentanz in der Anzeige, häufig zusätzlich mit einem Faceplate für Detailinformationen. Für andere Anlagenteile gilt ähnliches, sodass viele Darstellungen umfangreiche Konfigurationsarbeit erfordern.

Hier ist eine automatische Erstellung der Darstellung eine Erleichterung für diese monotone Arbeit. Ist der Regelsatz einmal fehlerfrei implementiert, so ist gewährleistet, dass dann beispielsweise die Verknüpfung zu allen Anlagenteilen korrekt ist. Weiterhin ist die Applikation schneller und damit kostengünstiger zu erstellen.

Schon 2007 wurde am Lehrstuhl für Prozessleittechnik in Aachen ([SE07]) die automatische Erstellung von Bedienoberflächen angedacht.

Eine Voraussetzung für eine einfache automatische Erstellung ist die modellbasierte Speicherung der Bedienoberfläche. Hier ist ein offenes Datenformat oder eine offene Schnittstelle von Vorteil, damit nicht nur der Hersteller der Bedienoberfläche solche Leistungen anbieten kann.

Simatic PCS 7 von Siemens besitzt hierfür die Funktionalität „Bildbausteine erzeugen“. Dieses erzeugt Bausteine basierend aus der Steuerungsinformation der Ventile, Motoren und Ablaufsteuerungen. Allerdings fehlen hier unter anderem noch die Sensorik, Behälter und Rohrleitungen. Weiterhin ist in den Steuerungsinformationen keine Positionierungsinformation, weshalb nur die Erstellung und Verknüpfung der Bedienelemente mit dem Steuerungssystem möglich ist.[DDFU11] Grobe Positionierungsdaten bietet das R&I-Fließschema, welches in der Siemenssoftware COMOS vorhanden ist. Diese Information kann über den Standard CAEX exportiert werden. Dies nutzt beispielsweise *autoHMI* der TU Dresden [DDFU11, DU11, UHH+11]. Hier wird die Positionierung der vorhanden Bausteine aus dem R&I-Fließschema/CAEX-Daten korrigiert. Weiterhin werden aus diesen Daten die fehlenden Elemente (Sensoren, Behälter, Rohrleitungen) extrahiert und in der Bedienoberfläche ergänzt.

2.4 Fazit

Zusammenfassend lässt sich festhalten, dass es viele Modelle gibt, welche eine automatische Erstellung von Bedienoberflächen erlauben.

Jedoch existieren keine Modelle, welche gleichzeitig auch die Interaktion mit dem Bediener und dem Prozess explizit modellieren. Dies wäre jedoch für eine einfache Änderung durch Nicht-Spezialisten sinnvoll. Weiterhin hilft diese Technologieneutralität bei der Sicherstellung der Zukunftssicherheit.

3 Explizites Modell für Benutzungsschnittstellen leittechnischer Funktionen

Aus dem vorangegangenen Kapitel 2 wird deutlich, dass aktuell keine Modelle der Prozessleittechnik oder Informationstechnologie zur expliziten vollständigen technologieneutralen Beschreibung einer Bedienoberfläche bestehen. Dies ist aufgrund der langen Lebensdauer einer Anlage jedoch wünschenswert.

Ziel der Arbeit ist es daher ein neuartiges Modell für Bedienoberflächen für leittechnische Funktionen zu entwickeln.

3.1 Anforderungen

Das Modell soll nicht nur für eine spezifische Applikation entwickelt werden. So ist eine Bedienoberfläche zu einer technischen Anlage oder auch ein Engineeringwerkzeug denkbar. Diese stellen jedoch unterschiedliche Anforderungen an eine Visualisierungsinfrastruktur. Für das erste Beispiel benötigt das System Zugriff auf Aktualwerte der Anlage. Dies kann beispielsweise der Messwert eines Temperatursensors sein. Neben diesen lesenden Eingriffen ist auch schreibender Zugriff zwingend nötig. So muss eine solche Applikation alle Arten von Aktoren schalten können.

Ein Engineering-Werkzeug hat dagegen komplexere Anforderungen. So muss die Struktur des Automatisierungssystems analysiert werden können, um beispielsweise für jede Komponente einen passenden Anzeigeteil bereitstellen zu können. Diese Struktur muss auch verändert werden können, um neue Elemente im Automatisierungssystems zu erstellen oder auch mehrere zueinander logisch zu verknüpfen.

Entsprechend werden die Kommunikationsformen „Wert schreiben“, „Wert lesen“, „Strukturen auflisten“, „Objekte erstellen“, „Objekt umbenennen“ und „Objekte löschen“ benötigt. Je nach gewählter Infrastruktur ist auch „Assoziation erstellen“ und „Assoziation löschen“ erforderlich.

Das Modell muss neben einfachen auch für komplexe Applikationen nutzbar sein. Es ist zu prüfen, ob ein solches Darstellungsmodell ausreichend mächtig erstellt werden kann, dass auch komplexe Anwendungen realisierbar sind.

Das Ziel ist ein einfaches System, welches im Lebenszyklus ohne spezielle Programmierkenntnisse angepasst werden kann. Hat ein Schichtführer die benötigten Rechte, so soll die Anpassung (je nach Standortpolitik) auch von diesem direkt erledigt werden können.

3.2 Grobstruktur des Modells

Das erstellte Modell für Bedienoberflächen fügt sich in die Modelle der ACPLT-Landschaft des Lehrstuhls für Prozessleittechnik in Aachen ein. Es trägt den Namen „Client Side Human Maschine Interface“, abgekürzt ACPLT/csHMI.

Für eine einfache Änderung ohne spezielle Programmierkenntnisse ist die Nutzung einer textbasierten Programmiersprache wie C/C++ nicht geeignet. Die Anwender kennen aus dem Arbeitsalltag mit Continous Function Chart (CFC) jedoch konfigurierbare Funktionsbausteine nach IEC 61131-3 ([IEC03]). Diese werden parametriert und arbeiten im Betrieb ihre Funktion ab, ohne dass der Anwender hier den genauen Quelltext einsehen kann oder gar will. Stichwort „Parametrieren statt Programmieren“.

Ähnlich wie viele in Kapitel 2.1 vorgestellte Modelle werden auch in csHMI Grafikelemente (wie Rechteck oder Text) als einzelne Bausteine modelliert. Aus den Bausteinen dieses allgemeinen Metamodells wird das spätere Modell der Applikation zusammengesetzt. Da eine Bedienoberfläche oft hierarchisch aufgebaut wird, bietet das Metamodell eine nicht zyklische, gerichtete Graphstruktur in die die Elementarbausteine eingehangen werden können. So soll beispielsweise ein Container alle grafischen Kind-Elemente auch in der Baumstruktur des Modells unter sich gruppieren.

Anders als bei anderen Technologien ist die Modellierung der Interaktion mit dem System und dem Benutzer gelöst. Diese wird äquivalent zu den Grafikelementen auch mit Elementarbausteinen modelliert. Unterhalb von jedem Grafikelement kann ein Ereignis-Baustein (wie Klick) erstellt werden. Dieser hat wiederum als Kindelement ein Aktions-Baustein der beispielsweise einen Motor startet. Die Gesamtheit aller Grafik-, Ereignis- und Aktions-Bausteine definiert das Modell der gewünschten Applikation.

Es gibt Applikationen, welche durch diese explizite Modellierung nicht vollständig effektiv beschrieben werden können. Zur Unterstützung dieser ist ein komplexer weiterer Baustein namens Blackbox definiert. Dieser erweitert das Modell um eine freie Programmierung, um die Nachteile bei grafisch sehr komplexen Anwendungen umgehen zu können. Trotzdem wurde diese Erweiterung nicht als „Fremdkörper“ der Philosophie des Gesamtkonzeptes gestaltet, sondern spielt die Vorteile geschickt aus. Im Idealfall wird ein Anwender einer solchen Erweiterung (wie bei einem CFC) nicht die Notwendigkeit sehen, in die Interna eintauchen zu müssen.

Die soeben erwähnte Modellierung beschreibt nur die Bedienoberfläche selbst. Zur Nutzung wird noch die Schnittstelle zum Benutzer sowie der zu steuernde Anlage benötigt. Das Gesamtkonzept sieht daher diese drei Komponenten (siehe Abbildung 3.1) vor:

- Datenbasis zur Speicherung des Modells der Applikation.

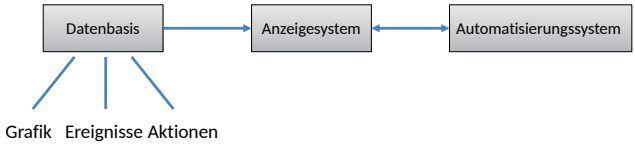

Abbildung 3.1: Grundstruktur des Konzeptes

- Anzeigesystem: Mit diesem interagiert der Bediener direkt. Im Normalfall ist dies ein Programm das auf seinem Computer oder Mobilgerät ausgeführt wird.
- Automatisierungssystem dessen Status und/oder Struktur angezeigt und/oder manipuliert werden soll.

Das Anzeigesystem wird vom Benutzer aufgerufen, lädt das Applikationsmodell und erstellt daraus die Darstellung auf den Bildschirm. Ist in der Applikation eine Interaktion (zum Beispiel Werte lesen oder schreiben) mit einem Automatisierungssystem erforderlich, so kommuniziert das System direkt mit diesem.

Mit welcher Kommunikationstechnologie das Automatisierungssystem angesprochen wird ist im Metamodell nicht festgelegt. Hier wird eine textbasierte Adressierung festgelegt, so dass beispielsweise OPC/UA [IEC10a] als `opc.tcp://427C-AS-RTX:4840` oder ACPLT/KS [Alb03] als `acpltks://427C-AS-RTX/fb_lbo_ProcessControl` adressierbar ist.

Ein Automatisierungssystem kann jedoch nicht nur Daten liefern, sondern auch einen Teil seiner Bedienoberfläche als Teilmodell selbst mitbringen. In die Hauptansicht kann daraufhin dieser dezentral gespeicherte Teil integriert werden. Siehe Abbildung 3.2:

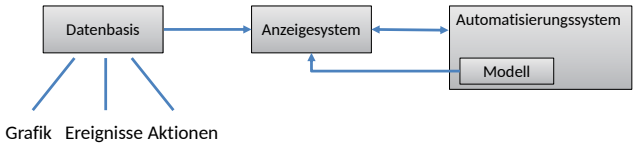

Abbildung 3.2: Erweiterte Grundstruktur des Konzeptes

3.3 Modellierungsebenen

Das Beschreibungskonzept besteht aus verschiedenen Ebenen.

- So existiert eine **generische Ebene**, welche alle benötigten grafischen Primitive und Ereignisse einer Bedienoberfläche beschreibt.
- Die zweite Ebene ist die **abstrakte Ebene**, welche die genaue Modellierung der vorgestellten Elemente der generischen Ebene inklusive der Definition der Aktionen festlegt.

- Weiterhin existiert die **technologische Ebene**, welche die wirkliche Implementierung des Anzeigesystems in einer bestimmten Technologie beschreibt. Dieses kann beispielsweise in C# oder Java erstellt worden sein.

Diese strikte Trennung hat den Vorteil, dass eine Anwendung in der abstrakten Ebene definiert wird. Somit kann man die technologische Ebene auch nach Erstellung vieler Anwendungen beliebig verändern. Ein Technologiewechsel ist durch einen Export aus der alten und einen anschließenden Import in die neue Technologie einfach möglich.

3.4 Komponenten des Modells

Um ein explizites Modell einer Applikation zu spezifizieren, wird eine begrenzte Auswahl von Elementarbausteinen sowohl für die Darstellung und als auch die Interaktion benötigt. Zur Auswahl wird auf die Gemeinsamkeiten der vorhandenen Modelle (siehe Kapitel 2.2) zurückgegriffen. Neben den Bausteinen werden jeweils deren wichtigste Attribute vorgestellt.

3.4.1 Darstellung

Jede grafische Darstellung besteht aus einer gewissen Anzahl von grafischen Primitiven oder Grundformen. Viele komplexe Bedienoberflächen sind nur aus wenigen Grundtypen zusammengestellt (siehe [Dam96, Sch10]). Die meist verwendeten Elemente sind zum Beispiel das *Rechteck* und ein *Text*. Eine hierarchische UML-Darstellung der im Folgenden erwähnten Primitive findet sich in Abbildung 3.3.

Diese Elemente sind Teil der **generischen Ebene des Gesamtkonzepts** (siehe Kapitel 3.3), da sie in allen Bedienoberflächentechnologien so oder ähnlich benötigt werden.

Zu diesen Form-Elementen gehören zusätzlich zu den beiden Genannten noch der *Kreis*, *Ellipse* sowie der *Polygonzug* und das *Polygon* (dies ist ein geschlossener Polygonzug). Mithilfe eines *Pfad*-Form-Elements ist es möglich komplexe Darstellungen wie Kurven oder Kreisbögen zu erstellen. Diese können über eine Beschreibung von Stiftbewegungen, welche die benötigte Darstellung liefern würde, definiert werden (vergleiche die *path*-Elemente in den Vektorgrafikstandards SVG [Fer01], PGML [FD98] und VML [BD98]). Auch eine Anzeige von Raster- und Vektorgrafiken wird oft benötigt.

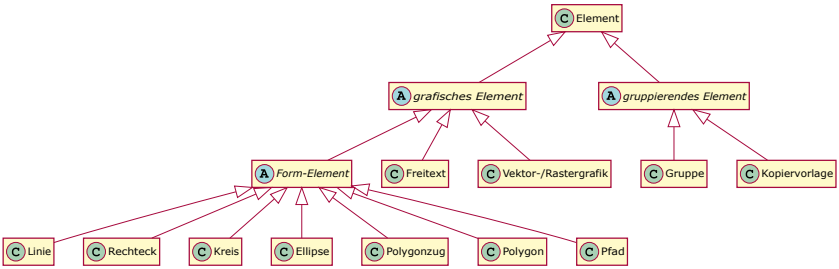

Abbildung 3.3: UML-Grundstruktur aller Elemente

Alle grafischen Elemente erhalten zur universellen Nutzung einige gemeinsame Attribute: *Strichfarbe*, *Füllfarbe*, *Rotation*, *Transparenz* und die *Sichtbarkeit*. Eine *Strichstärke* benötigen dagegen nur alle Form-Elemente, nicht jedoch zum Beispiel Text.

Allen grafischen Elementen gemeinsam ist die Überlappung. Das Konzept sieht vor, dass die Elemente in einer festgelegten Reihenfolge in der Hierarchiestufe gespeichert sind. In genau dieser

Reihenfolge werden sie auf dem Bildschirm gezeichnet. Frühe Elemente in der Hierarchie werden von anderen im Zweifelsfall überdeckt.

Alle weiteren Attribute sind spezifisch zu den Elementen zu definieren und im Folgenden aufgelistet:

- Eine Linie benötigt eine Spezifizierung der Start/End-Koordinaten (zum Beispiel `x1`, `x2`, `y1`, `y2`).
- Ein Rechteck kann über vier Punkte oder einfacher mit zwei Koordinaten und Breite und Höhe definiert werden (`x`, `y`, `width`, `height`).
- Ein Kreis ist über den Mittelpunkt und den Radius festgelegt (`cx`, `cy`, `r`).
- Eine Ellipse benötigt zwei Radien jeweils für die große und kleine Halbachse (`cx`, `cy`, `rx`, `ry`).
- Der Polygonzug und das Polygon kann mit der gleichen Reihe von Eckpunkten aufgebaut werden (`points`). Das Polygon wird jedoch bei der Anzeige geschlossen.
- Das Pfad-Element kann man als Aneinanderreihung von Stiftbewegungen definieren. Ein Beispiel wäre: Bewege den Stift auf Koordinate x:10, y:10; Zeichne von dort eine gerade Linie zur Position x:130, y:31; Bewege den Stift (ohne zu zeichnen) zu x:20, y:20 und führe von hier einen Halbkreis mit Radius 30 nach unten. Dieses Element benötigt zur Definition daher eine Auflistung aller Stiftbewegungen in einer speziellen Syntax (zum Beispiel im Attribut `shape`).
- Freitext benötigt den gewünschten Inhalt, den Ankerpunkt, Ausrichtungsanweisung (rechts-, linksbündig, mittig), Schriftgestaltungshinweise wie Schriftgröße oder Schriftart (`content`, `x`, `y`, `horAlignment`, `verAlignment`, `fontSize`, `fontStyle`, `fontWeight`, `fontFamily`).
- Raster-/Vektorgrafiken benötigen einen Ankerpunkt, eventuell eine Größe sowie eine Einbindung der gewünschten Darstellung, sei es als Referenz oder direkte Einbindung (`x`, `y`, `width`, `height`, `Vektorcontent`, `Bitmapcontent`).

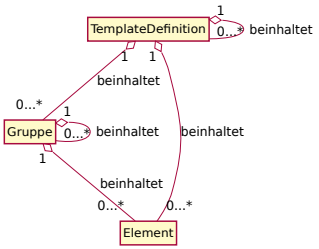

Abbildung 3.4: Erlaubte Assoziation von Gruppe und Element

Weiterhin ist ein gruppierendes Element sinnvoll, so dass grafische Elemente logisch zusammengefasst werden können. Dies erleichtert auch das gemeinsame Positionieren zum Beispiel innerhalb

eines Pop-ups. Diese `Gruppe` benötigt ähnlich wie Elemente eine *Position*, *Ausdehnung*, *Rotation*, *Transparenz* und die *Sichtbarkeit*.

Die Regeln, welche Elemente und Gruppen wie ineinander verschachtelt werden dürfen, sind in Abbildung 3.4 festgelegt. So darf eine Gruppe eine beliebige Menge an Elementen und auch Gruppen enthalten. Elemente dürfen jedoch selbst keine weiteren Elemente oder Gruppen als Kindelemente beinhalten. Auch braucht ein Element zwingend eine Gruppe als „Vater“ innerhalb der Baumstruktur.

Gruppen stellen somit auch den Einstiegspunkt einer Darstellung dar. Hat eine solche Gruppe keine andere Gruppe als „Vater“ so ist der Baum den sie aufspannt eine gültige Anzeige.

3.4.2 Kopiervorlagen

Zur Wiederverwendung von Elementen und Gruppen ist eine Art Kopiervorlage sinnvoll. Diese `TemplateDefinition` können von den normalen Gruppen referenziert werden und ergänzen dessen eigene Darstellung. Sie sind äquivalent zu Klassen in einer objektorientierten Programmierung zu sehen. So werden auch sie einmal zentral definiert und können beliebig oft und unterschiedlich parametriert verwendet werden. Das Konzept sieht vor, dass jedes gruppierende Element genaue eine Kopiervorlage referenzieren und beliebig viele eigene Kindelemente enthalten kann. Kopiervorlagen besitzen selbst nur eine Größe, nicht jedoch Position.

Verschiedene „Instanzen“ der gleichen `TemplateDefinition` sollen verschiedene Funktionen bereit stellen können. So ist es sinnvoll eine Vorlage für Darstellungen und Bedienung von Pumpen zu erstellen und dieser eine *Referenz* zu der jeweiligen Pumpeninstanz mitgeben zu können.

Es scheint sinnvoll die Adressierung als Text auszulegen, damit das Modell sehr flexibel in der Wahl des Kommunikations- und damit des Automatisierungssystems ist. Die meisten Systeme in der Prozessleittechnik bieten aufgrund der IEC61131 Sprachen [IEC03] als Text adressierbare Objekte mit zugehörigen Variablen. Diese Grundstruktur wurde beispielsweise auch in das Informationsmodell des Protokolls OPC UA [IEC10a] übernommen.

Dieses Konzept der Objekte mit Variablen hat zur Folge, dass drei verschiedene Parameterarten gebraucht werden:

1. Keine oder genau eine Objektreferenz (`FBReference`): Über diese Referenz ist ein komplexer Baustein erreichbar. Dies kann beispielsweise ein Motorkontrollbaustein sein, aber auch ein Additionsbaustein. Ein Nutzer dieser Information kann beispielsweise eine oder mehrere Variablen des Bausteins direkt auslesen. Dazu muss er den Aufbau dieses Bausteins kennen.

2. Keine oder mehrere Variablenreferenzen (`FBVariableReference`): Diese Referenz mit einem eindeutigen Namen zeigt auf genau eine Variable. Durch den Namen ist es hiermit möglich einer Kopiervorlage mehrere unterschiedliche Variablennamen zu übergeben.

3. Keine oder mehrere Konfigurationswerte (`ConfigValues`): Hier können beliebig viele Variablen übergeben werden. Diese haben jeweils einen eindeutigen Namen und einen statischen Wert. Dies kann beispielsweise der Maximal-Wert eines Sensors oder ein Beschriftungstext sein.

Auf diesem Konzept basieren auch die „Custom Properties“ der „Dynamic Shapes“ der Operatorstation HMIWeb des Experion PKS von Honeywell Process Solutions (siehe [Hon14]).

3.4.3 Ereignisse

Ein komplexerer Bereich ist die Interaktion zwischen dem Benutzer und beliebigen Daten. Die für die Darstellung relevanten Daten können innerhalb des Anzeigesystems selbst oder im Automatisierungssystem liegen. Die Interaktionen auslösenden Ereignisse können verschiedener Natur sein. Beispielsweise zyklisch, zeitgesteuert oder „beim Laden“. In Benutzungsoberflächen wird ein Ereignis oft ausgelöst durch eine Benutzeraktion, wie ein Klick.

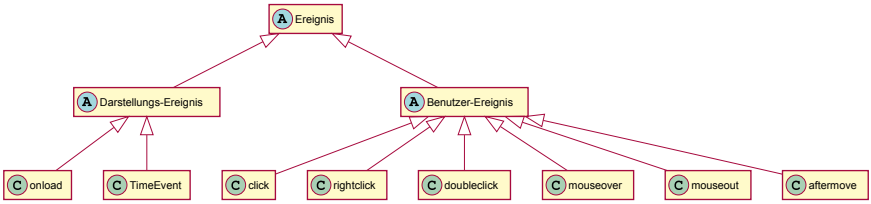

Abbildung 3.5: UML-Grundstruktur aller Ereignisse

Auch diese Ereignisse sind Teil der **generischen Ebene des Gesamtkonzepts** (siehe Kapitel 3.3), da sie in allen Bedienoberflächentechnologien benötigt werden.

Eine allgemeine Darstellungstechnologie sollte die gleichen Aktionen bei unterschiedlichen Ereignissen ausführen können, sodass eine Trennung der Ereignisse von den auszuführenden Aktionen sinnvoll ist. Abbildung 3.5 bietet eine Liste von häufig innerhalb einer Darstellung genutzten Ereignissen. Einem grafischen oder gruppierenden Element können im Konzept dieser Dissertation beliebig viele Ereignisse zugeordnet werden (siehe Abbildung 3.6).

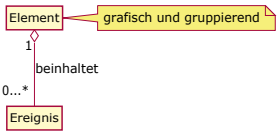

Abbildung 3.6: Erlaubte Assoziation von Element und Ereignis

Das wichtigste Ereignis ist `onload`. Dieses wird aktiv nach dem Laden eines Darstellungs-Elements. Eine `onload`-Aktion kann für gruppierende, als für auch grafische Elemente (siehe voriges Kapitel 3.4.1) nützlich sein.

Benutzer-Ereignisse können von vielen verschiedenen Eingabegeräten initiiert werden. Dies kann eine Maus, eine Tastatur (auch Sondertastaturen) oder ein Touchscreen (siehe auch VDI/VDE 3699 Blatt 6 [VDI13]) sein.

Für eine Mausbedienung sind die Ereignisse *Klick*, *Doppel-Klick* und *Klick mit rechter Maustaste* die wichtigsten. Aber auch das Bewegen in (`mouseover` genannt) oder Verlassen (`mouseout`) eines Bereichs eines Elements ist für eine Mausbedienung ein typisches Ereignis.

Die Kombinationsgeste *Ziehen und Ablegen* („Drag and Drop“) ist eine weitere häufig genutzte Aktion einer Darstellung. Dabei wird die Maus über ein Element bewegt, eine Maustaste gedrückt und damit das Element „festgehalten“. Das Element folgt daraufhin der Mausbewegung, bis die Maustaste los gelassen wird. Hauptsächliches Ziel dieser Interaktionstechnik ist ein Verschieben eines Elements in der gleichen Hierarchieebene sowie eine Interaktion mit dem „Ziel“. Hierbei wird ein Element von einem Kontext in einen Anderen überführt. Letzteres wird zum Beispiel häufig in einem Engineeringwerkzeug (siehe das Beispiel in Kapitel 5.4.1 auf Seite 46) bei einem Löschvorgang benutzt. Dabei wird ein Element aus dem Anlagenkontext über ein Mülleimer (in dessen Kontext) geschoben.

Eine „Drag und Drop“ Sequenz ist sehr komplex. Der Standard HTML5 ([BFL+14]) definiert beispielsweise acht Ereignisse im Umfeld dieser Interaktionsform (`dragstart`, `drag`, `dragenter`, `dragleave`, `dragover`, `dragexit`, `drop`, `dragend`). Das in dieser Arbeit vorgeschlagene Konzept soll jedoch möglichst einfach gehalten werden. Daher ist als „Drag and Drop“ Ereignis nur `aftermove` definiert. Die zugehörigen Aktionen werden nach dem Ende der Verschiebegeste ausgeführt. Weiterhin kennzeichnet das Vorhandensein des Ereignisses ein Objekt als verschiebbar. In Kombination mit dem Ereignis *mouseover* sind jedoch auch unterschiedliche Aktionen für verschiedene Ziele möglich.

Wird in einer verschachtelten Gruppe das gleiche Ereignis (zum Beispiel ein Klick) mehreren Gruppenelementen zugeordnet, so wirkt das Ereignis immer auf das lokal „oberste“Element. In dem Beispiel aus Abbildung 3.7 soll beispielsweise bei einem Klick auf `Gruppe-Pumpe1` die Pumpe angeschaltet werden, beim Klick auf den `TextSollwert` jedoch der Sollwert der Pumpe geändert werden. Es wurde festgelegt, das beim Klick auf den Text das Ereignis2 ausgeführt wird, da dort das Ereignis „näher“ definiert wurde. Diese Festlegung wird *bubbling* (siehe [Koc06, Pix00]) genannt. Die nicht genutzte Alternative ist *capturing*. Hier findet eine Ausführung des Ereignis1 statt, da dies näher an der Basis definiert ist.

Auch auf einem Touchscreen gibt es äquivalente Interaktionen die man den Ereignissen *Klick*, *Doppel-Klick* und *Drag und Drop* zuordnen kann, daher spricht auch diese zukunftsträchtige Technologie nicht gegen die Festlegung dieser Interaktionsereignisse.

Es fällt auf, dass durch eine Tastatur keine Anwender-Ereignisse ausgelöst werden können. Die Tastatureingabe wird in diesem Konzept nur als Datenquelle (wie zum Beispiel auch eine Mausposition) innerhalb der Aktionen (siehe nächstes Kapitel 3.4.4) modelliert.

Das letzte wichtige Ereignis für eine Anwendung sind zeitgesteuerte Aufgaben. Eine Annahme dieser Dissertation ist, dass für übliche Anwendungen in der Prozessleittechnik eine zyklische Bear-

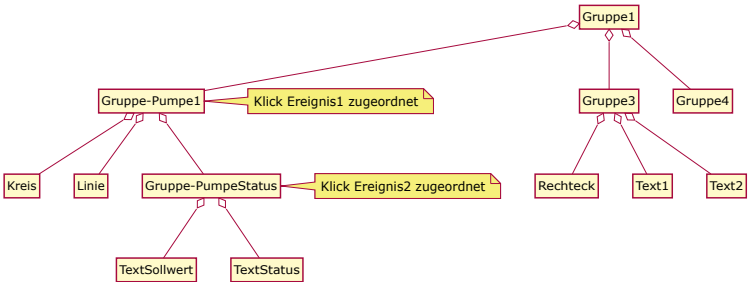

Abbildung 3.7: Capture und Bubbling

beitung mit konstanter Zykluszeit ausreichend ist. Im Gegensatz zu den bisher vorgestellten Ereignissen wird hier ein Attribut benötigt: die Zykluszeit. Die zugeordneten Aktionen werden regelmäßig nach Ablauf der eingestellten Zykluszeit ausgeführt.

3.4.4 Aktionen

Neben den Ereignis-Zeitpunkten selbst sind die Aktionen, die jeweils ausgeführt werden sollen, festzulegen. Genau wie jedem Element beliebig viele Ereignisse zugeordnet werden können, so können diesem wiederum beliebig viele Aktionen zugeordnet werden (siehe Abbildung 3.8).

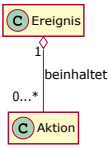

Abbildung 3.8: Erlaubte Assoziation von Ereignis und Aktion

Da keine Beschreibungssprache für Darstellungssysteme gefunden wurde, welche die Interaktion mit Fremddaten und dem Benutzer über ein ähnliches Konzept realisiert (siehe Kapitel 2.3), werden hier die gewählten Konzepte anhand von Anforderungen für Anwendungen definiert. Diese Aktionen gehören daher zur **abstrakten Ebene des Gesamtkonzepts** (siehe Kapitel 3.3), da sie in den vielen Bedienoberflächentechnologien unterschiedlich definiert werden.

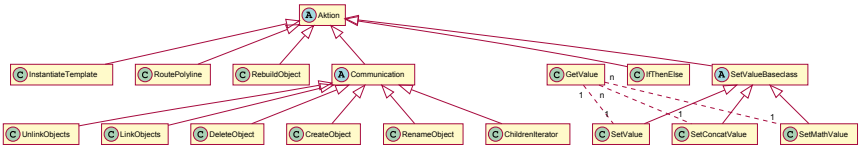

Abbildung 3.9: UML-Grundstruktur aller Aktionen

Abbildung 3.9 listet diese Aktionen auf. Allen Aktionen gemeinsam ist die Abarbeitungsreihenfolge. Diese ist ähnlich zur Überlappung der Elemente festgelegt, sodass die Reihenfolge in der Hierarchie gespeichert werden muss und in dieser die Aktionen abgearbeitet werden.

Die Grundidee der Modellierung ist es, keine Trennung zwischen der Darstellung und dem Automatisierungssystem vorzunehmen. In diesem Konzept wird in der Anwendung die gleiche Aktion genutzt, um beispielsweise einen Sollwert einer Pumpe zu setzen oder eine Farbe eines Bedienknopfes zu setzen. Auch das Auslesen eines Temperatursensors sollte sich den gleichen Mitteln bedienen wie die X-Koordinate einer Interaktion auf einer Schaltfläche zu ermitteln.

So sind generische Aktionen definiert, welche Werte lesen (**Datenquelle**, `getValue`) und andere Aktionen die diese Werte anschließend (eventuell verändert) schreiben (**Datensenke**, `setValue`). Eine grundlegende Philosophie ist, dass ein lesendes Element (Datenquelle) nicht alleine existieren kann, sondern immer an seinen Daten-Nutzer (Datensenke) gekoppelt ist. Ein `setValue` verfügt daher über genau ein `getValue`. Diese beiden Aktionen müssen für die Aufgabe natürlich passend konfiguriert werden.

Es folgt ein Beispiel, um dieses Konzept zu verdeutlichen: Soll beispielsweise der Messwert eines Temperatursensors zyklisch auf dem Bildschirm gebracht werden, so sind hierfür einige Komponenten nötig.

Zur Anzeige wird zwingend eine Gruppe benötigt, in dieser liegt hier als Beispiel nur ein Text-Element für den Wert. Diesem ist ein TimeEvent zugeordnet (konfigurierte Zykluszeit beispielsweise 1 pro Sekunde), welche wiederum ein setValue ausführt. Dieses *SetValue* ist eingestellt, dass es den Inhalt (`content`) des Textes überschreibt. Das *GetValue* ist dagegen so konfiguriert, dass der Messwert aus dem Automatisierungssystem gelesen wird. Die Modellierungskette lautet daher:

```
1  INSTANCE   /TechUnits/cshmi/group1 :
2      CLASS /acpIt/cshmi/Group;
3      VARIABLE_VALUES
4        x : INPUT SINGLE = 0.000000;
5        y : INPUT SINGLE = 0.000000;
6        width : INPUT SINGLE = 1000.000000;
7        height : INPUT SINGLE = 900.000000;
8      END_VARIABLE_VALUES;
9  END_INSTANCE;
10 INSTANCE   /TechUnits/cshmi/group1/CaptionText :
11     CLASS /acpIt/cshmi/Text;
12     VARIABLE_VALUES
13       x : INPUT SINGLE = 75.000000;
14       y : INPUT SINGLE = 15.000000;
15       content : INPUT STRING = "loading...";
16     END_VARIABLE_VALUES;
17 END_INSTANCE;
18 INSTANCE   /TechUnits/cshmi/group1/CaptionText/timer :
19     CLASS /acpIt/cshmi/TimeEvent;
20     VARIABLE_VALUES
21       cyctime : INPUT SINGLE = 1.000000;
22     END_VARIABLE_VALUES;
23 END_INSTANCE;
24 INSTANCE   /TechUnits/cshmi/group1/CaptionText/timer/setContent :
25     CLASS /acpIt/cshmi/SetValue;
26     VARIABLE_VALUES
27       elemVar : INPUT STRING = "content";
28     END_VARIABLE_VALUES;
29 END_INSTANCE;
30 INSTANCE   /TechUnits/cshmi/group1/CaptionText/timer/setContent.value :
31     CLASS /acpIt/cshmi/GetValue;
32     VARIABLE_VALUES
33       ksVar : INPUT STRING = "/TechUnits/TU10/add.value";
34     END_VARIABLE_VALUES;
35 END_INSTANCE;
```

Wie oben erwähnt kann ein *SetValue* nicht nur Text, sondern auch alle Eigenschaften der Anzeige verändern. Dies kann zum Beispiel die Position, Schriftfarbe eines Textes, aber auch die Größe, Strichfarbe und Füllfarbe eines beliebigen Form-Elements sein. Der Einfachheit halber ist in diesem Konzept jedoch nur eine Änderung des direkt zugeordneten grafischen Elements (beispielsweise der Text) möglich. Dies vereinfacht die Modellierung einfacher Anwendung. Für weitergehende Änderungen existiert ein weiteres Ereignis, welches auf anderen Elementen Aktionen und damit Veränderung auslösen kann (siehe Kapitel 3.5.2).

Werte aus dem Automatisierungssystem können entweder direkt adressiert werden oder über die Objekt- beziehungsweise Variablenreferenzen des Vorlagensystems (siehe vorigen Abschnitt 3.4.2) gelesen und geschrieben werden. Die Konfigurationswerte (`ConfigValues`) des Vorlagensystems können auch als lokale Variablen zur Zwischenspeicherung im Anzeigesystem genutzt werden.

Eine Datenquelle `GetValue` kann neben den Eigenschaften der eigenen Darstellung, Werten aus einem Automatisierungssystem und den lokalen Variablen, auch noch Information des Bedieners liefern. Dies kann (während einer Interaktion) eine Mausposition sein oder auch eine Texteingabe, welche in diesem Fall angefordert wird. Zusätzlich ist noch möglich einen konstanten Wert als Datenquelle festzulegen.

Für **komplexere Manipulationen** der Daten wurde `SetConcatValue` und `SetMathValue` definiert. Beide heben die 1:1 Verknüpfung zu den Datenquellen auf und nutzen eine beliebige Anzahl Quellen. `SetConcatValue` hängt alle Werte direkt hintereinander und nutzt daraufhin diesen Wert. Eine mögliche Anwendung wäre beispielsweise den Messwert um eine physikalische Einheit zu ergänzen:

```
INSTANCE   /TechUnits/cshmi/group1 :
   CLASS /acplt/cshmi/Group;
   VARIABLE_VALUES
     x : INPUT SINGLE = 0.000000;
     y : INPUT SINGLE = 0.000000;
     width : INPUT SINGLE = 1000.000000;
     height : INPUT SINGLE = 900.000000;
   END_VARIABLE_VALUES;
END_INSTANCE;
INSTANCE   /TechUnits/cshmi/group1/CaptionText :
   CLASS /acplt/cshmi/Text;
   VARIABLE_VALUES
     x : INPUT SINGLE = 75.000000;
     y : INPUT SINGLE = 15.000000;
     content : INPUT STRING = "loading...";
   END_VARIABLE_VALUES;
END_INSTANCE;
INSTANCE   /TechUnits/cshmi/group1/CaptionText/timer :
   CLASS /acplt/cshmi/TimeEvent;
   VARIABLE_VALUES
     cyctime : INPUT SINGLE = 1.000000;
   END_VARIABLE_VALUES;
END_INSTANCE;
INSTANCE   /TechUnits/cshmi/group1/CaptionText/timer/setContent :
   CLASS /acplt/cshmi/SetConcatValue;
   VARIABLE_VALUES
     elemVar : INPUT STRING = "content";
   END_VARIABLE_VALUES;
END_INSTANCE;
INSTANCE   /TechUnits/cshmi/group1/CaptionText/timer/setContent/Value :
   CLASS /acplt/cshmi/GetValue;
   VARIABLE_VALUES
     ksVar : INPUT STRING = "/TechUnits/TU10/add.value";
   END_VARIABLE_VALUES;
END_INSTANCE;
INSTANCE   /TechUnits/cshmi/group1/CaptionText/timer/setContent/Unit :
   CLASS /acplt/cshmi/GetValue;
   VARIABLE_VALUES
     value : INPUT STRING = " ms";
   END_VARIABLE_VALUES;
```

```
41 END_INSTANCE;
```

Einige Darstellungen benötigen (wenigstens rudimentäre) Berechnungen. Als Beispiel sei hier die Anzeige von Messwerten mithilfe eines Balkendiagramms genannt. Der Messwert kann in einem beliebigen Wertebereich liegen. Der korrespondierende Balken muss nun prozentual die gleiche Höhe verglichen mit dem Maximalausschlag haben, wie der Messwert von seinem Maximalwert. Ist beispielsweise der Messwert 45 und der Maximalwert 90, so soll der Balken die Hälfte der maximal erlaubten Höhe erhalten.

Für diese mathematischen Operationen wurde eine einfache Bearbeitungsvorschrift geschaffen. Diese wird mit dem Baustein `SetMathValue` abgearbeitet. Die Rechnung beginnt mit dem Zahlenwert 0. Jeder Datenquelle wird zusätzlich noch eine mathematische Operation zugewiesen. Der Zahlenwert wird nacheinander mit der Operationen und den Werten der Datenquellen verändert. Als Beispiel sei ein Rotationszeiger genannt. Der Winkel in Grad kann über folgende Formel berechnet werden:

$$Rotation = \frac{PV * 180}{PV_{max} - PV_{min}} = \frac{PV * 180}{ValueRange} \tag{3.4.4.1}$$

`SetMathValue` Logik:

$$ValueRange = (0 + PV_{max}) - PV_{min} \tag{3.4.4.2}$$

$$Rotation = ((0 + PV)/ValueRange) * 180 \tag{3.4.4.3}$$

In eine lokale Variable wird der Wert $PV_{max} - PV_{min}$ vorberechnet. Dazu wird `SetMathValue` auf eine neue lokale Variable (beispielsweise $ValueRange$) konfiguriert und zwei `GetValue`-Aktionen zugeordnet (siehe Gleichung 3.4.4.2 und Listing 3.1). Die erste Aktion mit der Operation „Addition“ und dem Wert PV_{max} (dies kann fest konfiguriert sein oder beispielsweise aus dem Sensor ausgelesen werden) addiert den Wert von PV_{max} auf den Zahlenwert 0. Die zweite Aktion hat als mathematische Operation „Subtraktion“ und den Wert PV_{min}. Zusammen wird hiermit die Differenz gespeichert.

Zyklisch wird mit dieser lokalen Variable der Winkel des Zeigerinstruments berechnet. Eine weitere `SetMathValue`-Aktion wird konfiguriert auf die Rotation des Zeigerelements (siehe Gleichung 3.4.4.3 und Listing 3.2). Die erste zugehörige `GetValue`-Aktion ist konfiguriert als „Addition“ und dem Messwert, PV der aus dem Leitsystem geholt wird. Dies addiert den Wert auf den Start-Zahlenwert 0. Die zweite `GetValue`-Aktion ist eine „Division“ mit der lokalen Variable $ValueRange$ und teilt daher den aktuellen Wert durch die Differenz der Maximal- und Minimal-Werte. Die letzte nötige `GetValue`-Aktion ist eine „Multiplikation“ mit dem festen Wert 180.

Die Syntax ist vom Konzept ähnlich zur umgekehrten polnischen Notation. Zu beachten ist, dass es in diesem einfachen System keine Klammerung gibt, die Stackgröße daher genau 1 ist. Die aktuelle Operation manipuliert immer den gemeinsamen Zahlenwert. Als Konvention liefert der Präfix für die getValue Bausteine dessen mathematische Operation. Folgende Präfixe nutzen wie erwähnt den

mathematischen Operator mit dem bisherigen Wert: add*, sub*, mul*, div*. Die folgenden Präfixe addieren das Ergebnis: abs*, acos*, asin*, atan*, cos*, exp*, log*, sin*, sqrt*, tan*. Mit pow* wird der alte Wert potenziert mit dem neuen Wert. Ein Zufallswert von 0 bis zum neuen Wert liefert ran*.

```
INSTANCE   /TechUnits/cshmi/Templates/IOdriverlib/AIRotationPointerDisplay/onload/Save_ValueRange :
    CLASS /acplt/cshmi/SetMathValue;
    VARIABLE_VALUES
      TemplateConfigValues : INPUT STRING = "ValueRange";
    END_VARIABLE_VALUES;
END_INSTANCE;
INSTANCE   /TechUnits/cshmi/Templates/IOdriverlib/AIRotationPointerDisplay/onload/Save_ValueRange/addMax :
    CLASS /acplt/cshmi/GetValue;
    VARIABLE_VALUES
      TemplateFBReferenceVariable : INPUT STRING = "Max";
    END_VARIABLE_VALUES;
END_INSTANCE;
INSTANCE   /TechUnits/cshmi/Templates/IOdriverlib/AIRotationPointerDisplay/onload/Save_ValueRange/subMin :
    CLASS /acplt/cshmi/GetValue;
    VARIABLE_VALUES
      TemplateFBReferenceVariable : INPUT STRING = "Min";
    END_VARIABLE_VALUES;
END_INSTANCE;
```

Listing 3.1: SetMathValue Umsetzung von Gleichung 3.4.4.2

```
INSTANCE   /TechUnits/cshmi/Templates/IOdriverlib/AIRotationPointerDisplay/VerlaufGruppe/Zeiger/CyclicTime/Set_Rotation :
    CLASS /acplt/cshmi/SetMathValue;
    VARIABLE_VALUES
      elemVar : INPUT STRING = "rotate";
    END_VARIABLE_VALUES;
END_INSTANCE;
INSTANCE   /TechUnits/cshmi/Templates/IOdriverlib/AIRotationPointerDisplay/VerlaufGruppe/Zeiger/CyclicTime/Set_Rotation/
      addActualValue :
    CLASS /acplt/cshmi/GetValue;
    VARIABLE_VALUES
      TemplateFBReferenceVariable : INPUT STRING = "Pv";
    END_VARIABLE_VALUES;
END_INSTANCE;
INSTANCE   /TechUnits/cshmi/Templates/IOdriverlib/AIRotationPointerDisplay/VerlaufGruppe/Zeiger/CyclicTime/Set_Rotation/
      divValueRange :
    CLASS /acplt/cshmi/GetValue;
    VARIABLE_VALUES
      TemplateConfigValues : INPUT STRING = "ValueRange";
    END_VARIABLE_VALUES;
END_INSTANCE;
INSTANCE   /TechUnits/cshmi/Templates/IOdriverlib/AIRotationPointerDisplay/VerlaufGruppe/Zeiger/CyclicTime/Set_Rotation/mul180
      :
    CLASS /acplt/cshmi/GetValue;
    VARIABLE_VALUES
      value : INPUT SINGLE = 180.000000;
    END_VARIABLE_VALUES;
END_INSTANCE;
```

Listing 3.2: SetMathValue Umsetzung von Gleichung 3.4.4.3

Applikationen benötigen auch zwingend die Möglichkeit zu einer **bedingten Ausführung**. So wurde eine `IfThenElse`-Aktion inklusive Bedingungen definiert (siehe auch Abbildung 3.10). Die Verarbeitung mehrerer dieser Bedingungen kann mit einem logischen ODER beziehungsweise UND verknüpft werden. Ist eine (oder alle im Falle einer UND Konfiguration) Bedingung erfüllt, so werden die „then“-Aktionen ausgeführt, alternativ die „else“-Aktionen.

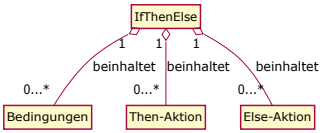

Abbildung 3.10: Erlaubte Assoziation zur IfThenElse-Aktion

Eine Auflistung der definierten Bedingungen liefert Abbildung/Listing 3.11. Ein *Vergleich* nutzt die gleichen Datenquellen die auch schon die SetValue-Aktionen genutzt haben. Der Zahlenwert zweier solcher Quellen kann jeweils verglichen werden (<,<=,==,!=,>= und >) und liefert damit für die Bedingung eine Aussage zu WAHR oder FALSCH. Für den zweiten zu vergleichenden Wert ist es sinnvoll mehrere Werte angeben zu können. Somit kann ein Wert bequem gleichzeitig auf mehrere Werte verglichen werden (beispielsweise Klassenname ist „add“, „sub“ oder „mul“).

```
1
2  INSTANCE   /TechUnits/cshmi/group/checkClassIterator :
3      CLASS /acplt/cshmi/ChildrenIterator;
4      VARIABLE_VALUES
5        ChildrenType : INPUT STRING = "OT_DOMAIN";
6      END_VARIABLE_VALUES;
7  END_INSTANCE;
8  INSTANCE   /TechUnits/cshmi/group/checkClassIterator.forEachChild/If :
9      CLASS /acplt/cshmi/IfThenElse;
10     VARIABLE_VALUES
11     END_VARIABLE_VALUES;
12 END_INSTANCE;
13 INSTANCE   /TechUnits/cshmi/group/checkClassIterator.forEachChild/If.if/Permitlist :
14     CLASS /acplt/cshmi/CompareIteratedChild;
15     VARIABLE_VALUES
16       childValue : INPUT STRING = "OP_NAME";
17       comptype : INPUT STRING = "==";
18     END_VARIABLE_VALUES;
19 END_INSTANCE;
20 INSTANCE   /TechUnits/cshmi/group/checkClassIterator.forEachChild/If.if/Permitlist.withValue :
21     CLASS /acplt/cshmi/GetValue;
22     VARIABLE_VALUES
23       value[3] : INPUT STRING = {"add" , "sub" , "mul"};
24     END_VARIABLE_VALUES;
25 END_INSTANCE;
```

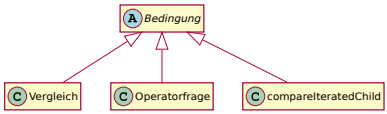

Abbildung 3.11: UML-Grundstruktur der Bedingungen

Die Bedingung *Operatorfrage* (`confirm`) stellt dem Bediener eine beliebige Frage (wiederum über eine Datenquelle festzulegen), welche dieser mit Ja oder Nein beantworten kann.

```
1  INSTANCE   /TechUnits/cshmi/Templates/Pandix/Pipe/click/If_DeleteMode.then/Confirm.if/Confirm :
2      CLASS /acplt/cshmi/Confirm;
3      VARIABLE_VALUES
4      END_VARIABLE_VALUES;
5  END_INSTANCE;
6  INSTANCE   /TechUnits/cshmi/Templates/Pandix/Pipe/click/If_DeleteMode.then/Confirm.if/Confirm.question :
7      CLASS /acplt/cshmi/GetValue;
8      VARIABLE_VALUES
9        value : INPUT STRING = "Do you really want to delete this object?";
10     END_VARIABLE_VALUES;
11 END_INSTANCE;
```

Neben den soeben vorgestellten Kontrollstrukturen sind weiterhin auch **strukturverändernde Interaktion** mit dem Automatisierungsystem definiert. Das Kommunikationssystem ACPLT/KS[Alb03] liefert eine Basisliste von sinnvollen Kommandos. Dazu gehört *Erstellen*, *Umbenennen*, *Löschen*, *Verknüpfung erstellen* und *Verknüpfung aufheben*. Die Konfiguration dieser Aktionen wird wiederum über die allgemeinen Datenquellen realisiert. Im folgenden Beispiel wird ein Objekt der Klasse `myClass` der Bibliothek `/acplt/myLib` an der Stelle `/TechUnits/TU10` erstellt, wobei der Name beim Benutzer angefragt wird.

```
1
2  INSTANCE    /TechUnits/cshmi/Templates/Pandix/internal/PandixClassButton/click/actionCreate :
3     CLASS /acplt/cshmi/CreateObject;
4     VARIABLE_VALUES
5     END_VARIABLE_VALUES;
6  END_INSTANCE;
7  INSTANCE    /TechUnits/cshmi/Templates/Pandix/internal/PandixClassButton/click/actionCreate.Name :
8     CLASS /acplt/cshmi/GetValue;
9     VARIABLE_VALUES
10       OperatorInput : INPUT STRING = "textinput:Please enter the name for the new object";
11    END_VARIABLE_VALUES;
12 END_INSTANCE;
13 INSTANCE    /TechUnits/cshmi/Templates/Pandix/internal/PandixClassButton/click/actionCreate.Place :
14    CLASS /acplt/cshmi/GetValue;
15    VARIABLE_VALUES
16       value : INPUT STRING = "/TechUnits/TU10";
17    END_VARIABLE_VALUES;
18 END_INSTANCE;
19 INSTANCE    /TechUnits/cshmi/Templates/Pandix/internal/PandixClassButton/click/actionCreate.Library :
20    CLASS /acplt/cshmi/GetValue;
21    VARIABLE_VALUES
22       value : INPUT STRING = "/acplt/myLib";
23    END_VARIABLE_VALUES;
24 END_INSTANCE;
25 INSTANCE    /TechUnits/cshmi/Templates/Pandix/internal/PandixClassButton/click/actionCreate.Class :
26    CLASS /acplt/cshmi/GetValue;
27    VARIABLE_VALUES
28       value : INPUT STRING = "myClass";
29    END_VARIABLE_VALUES;
30 END_INSTANCE;
```

Ein weiterer, jedoch nur lesender, Zugriff auf ein Automatisierungssystem bietet die Aktion des **Iterators**. Dieser kann über Vektorvariablen oder über eine Struktur iterieren. Eine Struktur kann zum Beispiel eine Liste aller Variablen oder alle Bausteine eines CFC sein. Für jedes gefundene Element können daraufhin beliebige weitere Aktionen ausgeführt werden. (siehe [Roc12])

Bei Iteratoren ist es teilweise möglich mehr Metadaten zu erhalten, so dass hier eine **spezielle Bedingung** `compareIteratedChild` definiert wurde. Solche Metainformation können zum Beispiel Zugriffsrechte oder Vererbungsinformationen beinhalten.

Gerade im Hinblick auf solche Iteratoren ist es sinnvoll eine **Kopiervorlage** (siehe Kapitel 3.4.2) auch **als Aktion** instanziieren zu können. Dafür wurde die Aktion `InstantiateTemplate` definiert, welches eine Kopiervorlage in Abhängigkeit von Iterator-Werten erstellt.

Das folgende Beispiel zeigt die Erstellung einer Kopiervorlage. Dabei wird der neuen Instanz mit dem speziellen Parameter `OP_NAME` die FBReferenz des aktuellen Iteratorschritts mitgegeben.

```
1  INSTANCE    /TechUnits/cshmi/Templates/Pandix/internal/activeView/onload/PandixIterator.forEachChild/If_ModulFound.then/
          Inst_Modul :
2     CLASS /acplt/cshmi/InstantiateTemplate;
3     VARIABLE_VALUES
4        TemplateDefinition : INPUT STRING = "Pandix/internal/ModulButton";
5        x : INPUT SINGLE = 400.000000;
6        y : INPUT SINGLE = 300.000000;
7        FBReference : INPUT STRING = "OP_NAME";
8     END_VARIABLE_VALUES;
9  END_INSTANCE;
```

Wäre dieser Iterator über eine Vektorvariable statt einer Struktur iteriert, so kann der Wert per `OP_VALUE` übergeben werden.

```
1  INSTANCE    /TechUnits/cshmi/Templates/Processcontrol/FaceplatePCUSSC/BtnCommands/TemplCommands/onload/readCommands.forEachChild
          /instCommandOperatorinputValue :
2     CLASS /acplt/cshmi/InstantiateTemplate;
3     VARIABLE_VALUES
4        TemplateDefinition : INPUT STRING = "Processcontrol/PCUCommandButton";
5        x : INPUT SINGLE = 0.000000;
```

```
6         y : INPUT SINGLE = 0.000000;
7         ConfigValues[2] : INPUT STRING = {"buttontext:OP_VALUE", "PFCommand:OP_VALUE"};
8       END_VARIABLE_VALUES;
9   END_INSTANCE;
```

In objektorientierten Automatisierungssystemen gibt es häufig eine Zugehörigkeit zwischen zwei Objekten. Dies kann beispielsweise in einem CFC der IEC 61131-3 eine Verbindung sein. Um diese anzeigen zu können wird eine grafische Darstellung dieser Zugehörigkeit benötigt. Die wäre mit einem Polygonzug statisch realisierbar. Spätestens, wenn der Benutzer jedoch ein Objekt verschieben kann, wird eine echte, **logische Verknüpfung zwischen zwei Objekten** nötig. Eine Überführung dieser logischen Zugehörigkeit in die grafische Darstellung liefert die Aktion `Routepolyline` (siehe [Roc12]). Diese kann einen Polygonzug passend berechnen, so dass zwei Punkte verbunden dargestellt werden.

In Bedienoberflächen ist es nötig eine **Teildarstellung vollständig neu aufzubauen**. Dies kann beispielsweise nötig sein, wenn das Automatisierungssystem strukturell verändert wurde oder ein anderer Teilbereich nun dargestellt werden soll. Daher wurde die Aktion `RebuildObject` definiert, welches ein beliebiges (grafisches oder gruppierendes) Element neu aufbaut.

3.4.5 Baustein zur Freitext-Programmierung

Das vorgestellte Modell wurde bewusst einfach gehalten um die meisten, jedoch nicht alle Anwendungen modellieren zu können. Komplexe, dynamische Darstellungen wie beispielsweise ein x,t-Diagramm oder x,y-Diagramm ist mit den vorgestellten Mitteln schwer bis gar nicht effizient zu realisieren.

Daher wurde eine Ergänzung namens `Blackbox` erstellt. Der Name wurde gewählt da die Verarbeitungs-Logik nicht direkt einsehbar ist [Fin13]. Sie bietet zwei unabhängige Funktionen. So ist sowohl eine flexible Anzeige komplexer Semantiken, als auch eine textbasierte Programmierung zur Manipulation der Anzeige möglich. Diese beiden Funktionen sind kombinierbar und erlauben eine sehr hohe Flexibilität.

Als Anzeige-Technologie wurde hier bewusst HTML [BFL+14] gewählt. Diese Auszeichnungssprache ist die Basis des World Wide Web. Daher ist sie vielen Entwicklern gut bekannt und es ist sehr viel Literatur verfügbar. Weiterhin ist die Auszeichnungssprache mit performanten Bibliotheken in viele Projekte einzubetten. Als Beispiel sei hier *QtWebEngine* oder *Chromium Embedded Framework (CEF)* genannt. Die Nutzung dieser Technologie bietet eine einfache Möglichkeit beispielsweise Tabellen, Auflistungen oder Fließtext darzustellen.

Für die textbasierte Programmierung wurde ECMAscript (ECMA-262 [ecm99]) gewählt, welche auch im World Wide Web gemeinsam mit HTML weite Verbreitung findet. Es gibt mehrere ECMAScript-Implementierungen für diese Skriptsprache. Am bekanntesten dürfte wohl Mozilla-JavaScript (eingesetzt in *Mozilla Firefox*), JScript (eingesetzt in *Microsoft Internet Explorer* und *Microsoft Edge*), Google V8 (*Google Chrome*) sowie JavaScriptCore (eingesetzt in *Apples Safari*) sein.

Durch diese breite Unterstützung der Technologie scheint die Wahl der Erweiterung keine zu starke Einschränkung in der zukünftigen Nutzbarkeit zu sein. Trotzdem wird hier das Konzept der Plattformunabhängigkeit und damit Zukunftssicherheit bewusst verlassen um mehr Freiheiten in der Darstellung bieten zu können.

Zur Interaktion der Programmierung mit dem Modell wurde eine spezielle JavaScript-API namens `cshmimodel` definiert.

Um das Konzept der „Parametrierung statt Programmierung“ nicht vollständig aufzugeben, wurden spezielle Variablen für den Baustein ermöglicht. So kann eine `Blackbox` ohne Verständnis des JavaScript-Codes an die eigenen Bedürftnisse angepasst werden. Die Erweiterung kann so ähnlich einer parametrierbaren Vorlage arbeiten. Dazu wurde das JavaScript-Objekt `cshmimodel.variables` in der API definiert. So kann beispielsweise die Aktualisierungsgeschwindigkeit eines x,t-Diagramms direkt parametrierbar gestaltet werden.

Die API bietet Funktionen, um auf alle Komponenten des Anzeigesystems zuzugreifen, sodass die textbasierte Programmierung nicht auf Interna der Anzeigetechnologie angewiesen ist. So liefert beispielsweise `cshmimodel.SvgElement` ein `SVGElement`-DOM-Interface (siehe [FJF03]) zur Manipulation der SVG-Seite der Blackbox selber. Der HTML-Teil (als `HTMLElement`-DOM-Interface, siehe [BFL+14]) der Blackbox ist über `cshmimodel.HtmlFirstElement` erreichbar.

Zur Kommunikation mit dem Automatisierungssystem bietet die API weitere Funktionen. So sind alle Funktionen des Kommunikationssystems KS erreichbar. Über die Funktion `cshmimodel.getVar` ist beispielsweise ein Abruf von einer oder auch mehrerer Variablen möglich.

Ein Beispiel liefert das Kapitel 5.6. Eine vollständige Beschreibung der Programmier-API ist im Anhang 3 im Listing 2 auf Seite 65 abgedruckt. Diese API-Beschreibung kann auch von JavaScript-Editoren für umfangreiche Unterstützung dienen.

3.5 Erweiterung der Grundkomponenten

Für komplexere Benutzeroberflächen ist das in Kapitel 3.4 vorgestellte Grundmodell nicht ausreichend. Daher wurde es um einige Details ergänzt, ohne jedoch die Grundphilosophie zu verletzen.

3.5.1 Erweiterung der Darstellung

Zur Vereinfachung der Erstellung von Applikationen wurde auf die Definition von Datentypen innerhalb der Applikationen verzichtet. Alle Werte werden einheitlich als Zeichenkette (String) und damit reiner Text interpretiert. Nur die Vergleichsoperatoren (siehe Kapitel 3.4.4) wandeln beispielsweise für einen größer/kleiner-Vergleich den Text kurzzeitig in eine Zahlen-Variable um.

Daraus ergeben sich für die einfache Definition der **Textbausteine** (siehe Kapitel 3.4.1) in der Praxis zwei Probleme.

- Oft ist möglich, dass eine Zahl in unterschiedlichen Genauigkeiten auf dem Bildschirm gebracht werden soll. Da alle Werte, wie erwähnt, als Text behandelt werden, liegt beispielsweise die Fließkommazahl `2` (wenn sie vom Automatisierungssystem geholt wird) in Gleichkommadarstellung als `2,0000000` vor. Dies ist in einer Anzeige meist unerwünscht.
- Weiterhin ist ein Text, der aus dem Automatisierungssystem kommt, zu lang um ihn direkt anzeigen zu können. Dies kann beispielsweise ein langer Name einer Klasse sein, der nicht in den generisch vorgesehenen Platz passt.

Für beide Probleme wurde eine gemeinsame Lösung gefunden. Das Text-Element erhält zusätzlich das Attribut `trimToLength`. Ist dieser auf den Initialwert `0` so wird der Text ohne Änderung genutzt. Ist der Wert des Attributes jedoch positiv (beispielsweise `10`) so wird ein Text auf diese Länge gekürzt. Aus „IdentifierType“ wird beispielsweise „Identifier...“. Mit dem Wert `-10` wird aus „IdentifierType“ wird „...tifierType“. Der ungekürzte Text wird als Tooltip bereitgestellt, sodass dieser beispielsweise als Pop-up-Fenster erscheint, wenn die Maus über dem Text ruht.

Wird eine Fließkommazahl erkannt, so bestimmt `trimToLength` die Anzahl der anzuzeigenden Nachkommastellen.

In vielen Anwendungsfällen ist es sinnvoll einen Teil der **Anzeige einzublenden**. So soll beispielsweise das Faceplate einer Pumpendarstellung nicht durchgehend angezeigt werden. Es ist möglich dies mit den vorhandenen Ereignissen und Aktionen zu realisieren, dies ist jedoch sehr umständlich. Daher wurde für alle Gruppen ein boolsches Attribut namens `hideable` definiert. Ist dieses `wahr` so kann die Sichtbarkeit über einen Klick auf das Vater-Element einfach umgeschaltet werden. Die Sichtbarkeit beim Laden ist davon unabhängig und je nach Anwendung passend festzulegen.

Eine wichtige Optimierung ist der **bedarfgestützte dynamische Aufbau** der Applikation. Einige Objekte sind schon beim Laden der Anzeige unsichtbar. Dies kann dauerhaft sein (bei Objekten, welche nur zum Entwickeln der Applikation benötigt wurden) oder sich während der Laufzeit der Anzeige ändern. Als Beispiel sei hier eine Tab-Navigation genannt, welche viele Tabs definiert jedoch die meisten beim Laden versteckt. Erst eine Interaktion mit dem Benutzer wechselt die aktive Anzeige.

Solche Seiten können auch sehr umfangreiche Darstellungen beinhalten. Daher ist ein vollständiges Laden aller unsichtbaren Kind-Elemente nicht sinnvoll. Hier wurde daher der Ansatz genutzt, dass alle grafischen Kind-Elemente beim Laden nicht aufgebaut werden. Die Ereignisse des unsichtbaren Element werden jedoch schon interpretiert. Mit dieser Maßnahme wird gewährleistet, dass beispielsweise ein `onload`-Ereignis die Sichtbarkeit direkt aktivieren kann. Wird ein bisher verstecktes Element später sichtbar geschaltet, so werden alle Kind-Elemente erstellt.

3.5.2 Erweiterung der Ereignisse

Interagiert ein Bediener mit einer Anzeige, so erwartet er eine **schnelle Reaktion**. So wurde schon 1968 von Miller [Mil68] erkannt, dass diese Reaktion schneller als 200 ms erfolgen sollte. Ansonsten

ist der Bediener nicht sicher, dass die Interaktion erfolgreich angenommen wurde. Da diese schnelle Reaktion nicht immer garantiert werden kann, wurde eine direkte visuelle Rückmeldungen bei Klick, Doppelklick und Rechtsklick implementiert. Das aktivierte Element wird für 800 ms farbig hervorgehoben. Die wirkliche Reaktion des Systems kann daher wesentlich langsamer erfolgen, der Bediener ist trotzdem sicher, dass seine Interaktion vom System registriert wurde.

Wie in Kapitel 3.4.4 erwähnt erlaubt die SetValue-Aktion nur eine Manipulation des direkt zugeordneten Elements. Dies reicht für viele Anwendungen nicht aus, so dass eine Art **„publisher subscriber"-System** innerhalb des Anzeigesystems definiert wurde. So wurde neben dem `onload` - und `TimeEvent`-Ereignis ein `globalvarchanged`-Ereignis definiert. Die zugehörigen Aktionen werden ausgeführt, wenn eine (beliebige) globale Variable geändert wurde. Somit ist es möglich, durch ein bestimmtes Ereignis eine Änderung an einer anderen Stelle in der Darstellung zu erzeugen.

3.5.3 Erweiterung der Aktionen

Die **Kopiervorlage als Aktion** `instantiateTemplate` erstellt eine Instanz abhängig von Informationen eines Iterators. Wird die Aktion beispielsweise für eine Engineering-Umgebung genutzt, so hat sich ein zusätzlicher Parameter namens `preventClone` bewährt. Mit diesem Parameter kann eine vollständig identische Kopie einer Vorlage verhindert werden. So würde beispielsweise eine mehrfache Referenzierung ein grafisches Objekt mehrfach auf dem Bildschirm dargestellt. Dies kann mit dem erwähnten Parameter verhindert werden.

Mit der Aktion `instantiateTemplate` kann beispielsweise ein Bedienknopf für mehrere unterstützte Kommandos einer Prozessführung (welches in einem Vektor bereitgestellt wurde) erstellt werden. Alternativ kann in einer Engineering-Oberfläche eine Liste aller vorhandenen Klassen erstellt werden.

Damit die neu erstellten Darstellungselemente nicht übereinander liegen, wurden die Parameter `xOffset`, `yOffset` sowie `maxTemplatesPerDirection` ergänzt. Mit den ersten Beiden kann jede neue Instanz verschoben zur Vorigen erstellt werden.

`maxTemplatesPerDirection` wird zum Beispiel benötigt, wenn sehr viele Instanzen erstellt werden sollen und damit eine Art „Zeilenumbruch" simuliert werden soll. Steht in diesem Parameter beispielsweise „x:3" so werden maximal drei Instanzen horizontal erstellt und anschließend (um den yOffset verschoben) die nächste Reihe.

Der **Baustein für Bedingungen** (siehe Kapitel 3.4.4) vergleicht zwei beliebige Werte (geliefert durch zwei getValue Datenquellen). Im Grundmodell wurde nicht definiert, wie das Ergebnis des Vergleichs im Fehlerfall eines dieser getValue-Quellen auszusehen hat. Damit der Applikationsentwickler hier alle Möglichkeiten hat, wurde das Verhalten parametrierbar gestaltet. Ist `ignoreError` bei einer Bedingung auf wahr eingestellt so wird der Wert als leere Zeichenkette angenommen. Darauf wird anschließend wie gewohnt der Vergleich dieser Bedingung angewendet.

Wurde `ignoreError` auf falsch parametriert, so wird im Fehlerfall die Verarbeitung der Bedingung und damit der `IfThenElse` Aktion abgebrochen.

Eine Anwendung in der Prozessführung (siehe Kapitel 5.5) erforderte eine **dynamische Übersetzung** von (englischen) Kommandos in deutsche Beschreibungstexte. So wurde der Baustein `TranslationSource` definiert. Dieser Baustein ist von `SetValue` referenzierbar und liefert eine zentrale Zuordnung der Ursprungstexte in die Zieltexte über die Variable `translationMapping`. Dies kann beispielsweise den Wert `OPEN:OFFEN, CLOSED:ZU` haben.

Als Test der einfachen Erweiterung des Modells wurde die experimentelle Aktion `Vibrate` erstellt, welche bei unterstützten Geräten (hauptsächlich Mobilgeräten) den **Vibrationsmotor** kurzzeitig aktiviert. Die Syntax wurde von der Vibration API [Kos14] des World Wide Web Consortiums übernommen. So ist neben einer einfachen Zeitdauer der Vibration (beispielsweise 500 ms) auch ein Muster durch einen Vektor möglich. So vibriert das Gerät mit der Angabe von $[50, 100, 150]$ für 50 ms, pausiert für 100 ms und vibriert abschließend noch einmal für 150 ms.

4 Realisierung

Das Modell des vorangegangenen Kapitels 3 ist unabhängig von einer Technologie definiert. In diesem Kapitel wird eine prototypische Implementierung des Konzepts vorgestellt, um die Praxistauglichkeit unter Beweis zu stellen. Diese Implementierung bildet die **technologische Ebene des Gesamtkonzepts** (siehe Kapitel 3.3).

Zur Evaluation des Gesamtsystems muss die Technologie für die Datenbasis (wo die Applikation gespeichert ist) und das Anzeigesystem (welche die Applikation auf dem Bildschirm bringt und mit dem Bediener interagiert) festgelegt werden. Diese Wahl beider Systeme ist unabhängig möglich.

Zur Gestaltung einer Anwendung bietet sich eine **Datenbank** an, welches schon beim Erstellen eine Syntaxprüfung erlaubt. So darf eine Aktion nur zu einem Ereignis assoziiert werden. Eine Auflistung der grundlegenden Restriktionen finden sich in den Abbildungen 3.4, 3.6, 3.8 und 3.10 des vorigen Kapitels. Auch sind nicht alle Werte für die Parameter der Elemente, Ereignisse und Aktionen sinnvoll. Auch hier ist eine Überprüfung und gegebenenfalls Korrektur schon bei der Erstellung hilfreich.

Das **Anzeigesystem** muss als wichtigste Anforderung auf gewünschten Plattformen (das können neben Windows, Linux, macOS auch Mobilgeräte verschiedenster Hersteller sein) zur Verfügung stehen. Sind mehrere Plattformen gewünscht, so kann das Anzeigesystem mehrfach erstellt werden oder das System selbst mehrere Plattformen unterstützen.

Weiterhin muss ein Anzeigesystem Kommunikation mit dem Automatisierungssystem erlauben. Das Anzeigesystem agiert als Client und greift auf einen oder mehrere Automatisierungssysteme als Server zu. Dazu müssen beide Systeme entweder direkt das gleiche Kommunikationsprotokoll beherrschen oder es wird ein Gateway zur Umsetzung benötigt.

4.1 Prototypische Implementierung

Zur Validierung wurde am Lehrstuhl für Prozessleittechnik in Aachen das Konzept prototypisch implementiert. [JE12, JE13] Als **Datenbasis** zur Speicherung der Anwendung wurde die objektorientierte ACPLT/OV-Umgebung (Objekt-Verwaltung) des Lehrstuhls für Prozessleittechnik in Aachen gewählt. Hiermit lassen sich Modelle der Leittechnik einfach realisieren und ausprobieren. Alle Logik wird hier in Form von Objekten modelliert. Vor der Nutzung muss eine Modellierung der Vererbungs-Klassen mit den jeweiligen Variablen erfolgen. Sowohl ein Instanziieren als auch ein schreibender Zugriff auf die Variablen erlaubt eine Ausführung von Programmcode, so dass in dieser Softwareumgebung eine einfache Validierung der Anwendung schon beim Erstellen möglich ist. Auch die

Benutzerfreundlichkeit des Engineeringsprozesses wird damit erhöht, indem häufige Fehler automatisch korrigiert werden.

Die verschiedenen Abstraktionsschichten der einzelnen Elemente, Ereignisse und Aktionen kann über Vererbung einfach umgesetzt werden.

Da ACPLT/OV als Konsolenanwendung konzipiert ist, besitzt es selbst keine Bedienoberfläche. Es existieren jedoch Software-Werkzeuge, um ein Engineering der Umgebung zu ermöglichen. Dies wird realisiert durch eine standardisierte Schnittstelle nach dem Client/Server-Prinzip. Jeder OV-Server bietet daher ein offenes Kommunikationsprotokoll ACPLT/KS [Alb03] als Serverdienst welchen diese Werkzeuge als Klienten nutzen.

Die Umgebung erlaubt eine Gliederung beliebiger Objekte über eine spezielle `containment`-Assoziation. Diese Assoziation wird in den Engineering-Werkzeugen als Hierarchie-Ebene dargestellt. Dies wird in der Implementierung der Bedienoberfläche genutzt um die Zuordnungen zu gruppierenden Elementen zu modellieren. Die Elemente, welche logisch innerhalb einer Gruppe liegen sollen, werden in OV unterhalb des Gruppenobjekts platziert. Aber auch die Zuordnung von Ereignissen zu Elementen und von Aktionen zu Ereignissen wird über die Hierarchie-Ebene realisiert. Somit wird ein Objektbaum mit der gesamten Haupt-Anwendung aufgebaut. Die Kopiervorlagen (siehe Kapitel 3.4.2) sind in einem separaten Objektbaum hinterlegt, um so Updates der genutzten Vorlagen einfacher handhabbar zu machen. Dahinter steckt die Idee, dass Vorlagen für viele Anwendungen erstellt werden und zusätzlich von anderen Entwicklern stammen (können).

Wie erwähnt muss das Anzeigesystem das Modell der Anwendung von der Datenbasis erhalten. Dies passiert in der Implementierung über ACPLT/KS. Das Kommunikationsprotokoll ist in einer binären Variante ([Alb03]; ksXDR genannt) sowie mehreren textbasierte Varianten über HTTP ([FR14a, FR14b]; ksHTTP genannt) spezifiziert. Gerade die Variante XML (ACPLT/KSX, [ME07]) über HTTP ist für viele potenzielle Anzeigesysteme eine einfach zu nutzende Kommunikationsform.

Die Wahl der Technologie des **Anzeigesystems** ist nicht so kritisch wie bei konventionellen Grafiksystemen, da diese (wie die Datenbasis) einfach ausgetauscht werden kann. Es muss nur das Metamodell (siehe Kapitel 3.4) implementiert werden und schon sind alle vorhandenen Anwendungen direkt nutzbar. Dieses Metamodell ist mit neun grafischen Elementen, dem Gruppen und Vorlagensystem, acht Ereignissen und dreizehn Aktionen ziemlich schlank. Die sieben Form-Elemente, viele Benutzerereignisse (click, rightclick, doubleclick) und die Kommunikations-Aktionen (create, rename, delete, link, unlink) sind sich jeweils sehr ähnlich, was den Erstellungsaufwand weiter senkt.

Da die Bedienoberfläche am Lehrstuhl häufig für zusätzliche Diagnosen oder Monitoringanwendungen genutzt wird, bietet sich hier ein installationsfreies Anzeigesystem an. Die Wahl fiel daher auf eine Webanwendung, also eine Kombination von HTML [BFL+14] und JavaScript [ecm99]. Für die Darstellung der Form-Elemente wird der Standard Scalable Vector Graphics (SVG) [FJF03] genutzt, der in allen modernen Browser integriert ist. Somit muss nicht jeder Anwender die Software installieren und insbesondere aktuell halten, da die Webanwendung von einer zentralen Stelle gepflegt werden kann (siehe auch [Sch10]).

Zusammenfassend greift die Abbildung 4.1 nochmal das Konzeptbild 3.1 des vorigen Kapitels auf. Es zeigt eine ähnliche Darstellung mit Hinweisen auf die jeweilige genutzte Technologie der Implementierung.

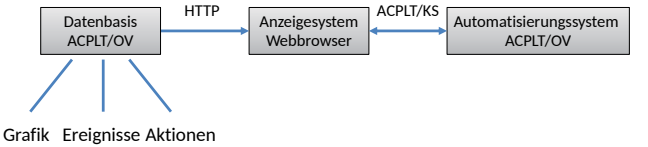

Abbildung 4.1: Grundstruktur von ACPLT/cshmi

- Als Datenbasis zur Speicherung der Applikation dient ACPLT/OV.
- Als Anzeigesystem dient eine Webanwendung: Mit diesem interagiert der Bediener direkt. Das Applikationsmodell wird zum Ladezeitpunkt der Applikation per HTTP übertragen.
- Da Webbrowser nur das HTTP-Protokoll unterstützen benötigt die Implementierung ein Automatisierungssystem, welches einen HTTP-Server integriert hat (wie bei ACPLT/ksHTTP) oder ein passendes Gateway was die Übersetzung der Kommunikation übernimmt.

5 Evaluation im Lebenszyklus (durch Anwendungen)

Um die Praxistauglichkeit des vorgestellten Modells für Bedienoberflächen zu beweisen, wurde im Rahmen dieser Dissertation zentrale Anwendungsbereiche untersucht und exemplarisch in Form von kleinen Applikationen evaluiert.

So sollte gezeigt werden, dass das Modell im Rahmen der Planung, des Engineering und im Betrieb erfolgreich eingesetzt werden kann, um Betriebsabläufe zu vereinfachen und den Programmieraufwand zu senken. Entsprechend beschäftigt sich Kapitel 5.1 mit der Eignung des Modells zur automatischen Erstellung von Bedienoberflächen, während Kapitel 5.2-5.5 auf modellbasierte Programme zur Anlagenplanung, der Simulation, zum Engineering und zum Betrieb eingegangen wird. In Kapitel 5.6 wird schließlich untersucht, inwieweit das Modell flexibel durch Freitextprogrammierbausteine erweiterbar ist bevor in Kapitel 5.7 ein Fazit gezogen wird.

5.1 Eignung zur automatischen Erstellung von Bedienoberflächen

Eine Bedienoberfläche wird im Regelfall per Hand von einem Anwendungsentwickler erstellt. So nutzt er zum Beispiel den Qt Creator oder den XAML-Designer von Visual Studio, um eine Bedienoberfläche zu erstellen. Dabei muss jede Komponente einzeln programmiert werden.

Um den Prozess zu vereinfachen und den Programmieraufwand zu senken, nutzen die meisten Hersteller eigene Bibliotheken mit vordefinierten Bausteinen. Auch gibt es Ansätze, wie zum Beispiel *autoHMI* der TU Dresden[DDFU11], aus R&I-Fließbildern automatisch Bedienoberflächen zu erzeugen. Die entwickelten Softwaretools sind jedoch bislang herstellerspezifisch und erlauben keinen universellen Import von Planungsdaten. So ergänzt die Software der TU Dresden hauptsächlich die Positionierung der herstellereigenen Generierung der Bedienoberfläche. Daher sollte eine neue Visualisierungsinfrastruktur die vollständige Generierung und Platzierung von Bedienelementen sowie die Verknüpfung dieser mit der Anlagensteuerung erlauben. Sie sollte in der Lage sein softwareneutrale Datenaustauschformate wie PandIX auszuwerten, um somit auf Planungsdaten unterschiedlicher Hersteller zuzugreifen und diese automatisch in eine Bedienoberfläche zu überführen.

Ein Ziel dieser Dissertation war es daher die Eignung des Modells zur automatischen Erstellung von Bedienoberflächen zu zeigen. Dabei kann die automatische Generierung eine Basis für händische Optimierung oder auch die endgültig benutzte Variante sein. Siehe hierzu auch Kapitel 2.3.

Evaluation

Zuerst wurde für alle wichtigen Anlagen-Elemente jeweils ein Grafikbaustein erstellt. Diese bestehen ausschließlich aus den Elementen des HMI-Metamodells. Diese wurden generisch aufgebaut, um die Wiederverwendbarkeit zu gewährleisten. Die Bausteine werden noch detailliert in Kapitel 5.5 vorgestellt.

Für die eigentliche Evaluation wurden aus dem Planungswerkzeug COMOS von Siemens Planungsdaten einer Versuchsanlage im PandIX-Format exportiert und in die ACPLT Laufzeitumgebung geladen (siehe hierzu auch Kapitel 5.2 und [ERD11, SE12, SE13]).

Anschließend wurde für jedes Anlagen-Element geprüft, ob ein entsprechendes Bedienelement als Grafikbaustein im Modell vorhanden ist. Konnte dieses nicht gefunden werden, so wurde stattdessen ein Platzhalter erstellt, welche mit dem generischen Faceplate immerhin eine Grundfunktion bietet. Die so generierten Anlagen-Elemente wurden in einem weiteren Schritt an die gleiche Position gesetzt, die sie auch in den Planungsdaten enthaltenen R&I-Fließbildes eingenommen haben. So konnte aus dem R&I-Fließbild automatisch eine Bedien- und Beobachtungsfläche für die Versuchsanlage erstellt werden, welche anschließend einfach optimiert werden konnte. Abbildung 5.1 zeigt die Grundstruktur der Bedienoberfläche. Diese besteht aus den generischen Bausteinen zur Visualisierung der Prozessführung sowie der generierten (und später händisch optimierten) Bedienoberfläche der Anlage. Bei der Nutzung wird aus beiden Teilen die Bedienoberfläche der Anlage als Instanz erstellt. Dieses beinhaltet ein HMI-Anlagenmodell, welches mit der Anzeige (und damit dem Benutzer) sowie der Anlage kommuniziert. Eine ausführliche Beschreibung einer ähnlichen Aufgabenstellung auf Basis von regelbasierten Modelltransformationen ist in [Mer18] nachzulesen.

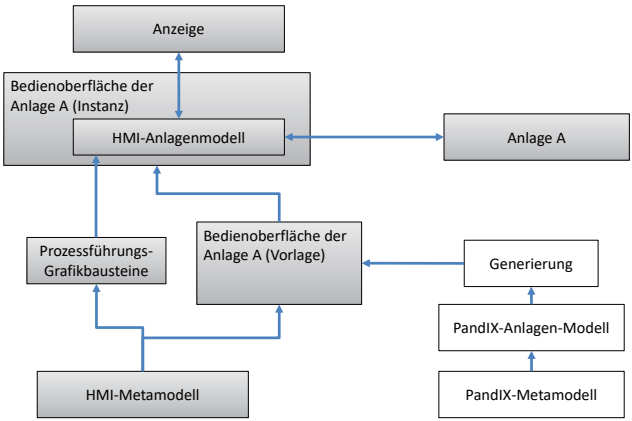

Abbildung 5.1: Struktur der generierten Bedienoberfläche

5.2 Engineering von Anlagenplanungsdaten (R&I-Fließbilder)

Nachdem in Kapitel 5.1 gezeigt werden konnte, dass der vorgestellte Ansatz durch sein Vorlagensystem die automatische Erstellung von Bedienoberflächen ermöglicht, soll in diesem Abschnitt auf die Eignung des Modells zur Anlagenplanung eingegangen werden. Dabei wäre es wieder von Vorteil, wenn die Anlagenplanung in einem herstellerunabhängigen Datenformat realisiert würde und mit möglichst geringem Programmieraufwand machbar wäre.

Evaluation

Als Datenformat wurde dabei wieder das herstellerunabhängige PandIX-Format gewählt. Es wurde eine vollständige Engineeringoberfläche für PandIX-Elemente mit ACPLT/csHMI erstellt. Ein Screenshot dieser Applikation ist in Abbildung 5.2 zu sehen. Die Darstellung ist zweigeteilt. So wird der aktuelle Stand der PandIX-Daten auf der rechten Seite der Engineeringoberfläche dargestellt und die PandIX-Klassen als Liste im linken Bereich. Im dargestellten Fall enthalten die PandIX-Daten bereits eine kleine Anlagenstruktur bestehend aus zwei Pumpen, einem Sensor und einem Behälter.

Der Aufruf der Applikation kann durch folgendes Listing 5.1 erfolgen. So werden über den Parameter `TemplateDefinition` (Zeile 6) das PandIX-Engineering und über den Befehl `FBReference` (Zeile 7) die anzuzeigenden PandIX-Daten referenziert.

```
1 INSTANCE /TechUnits/cshmi/engineeringPandIXSheet :
2   CLASS /acplt/cshmi/Group;
3   VARIABLE_VALUES
4     width : INPUT SINGLE = 1675.000000;
5     height : INPUT SINGLE = 1020.000000;
6     TemplateDefinition : INPUT STRING = "Pandix/PandixEngineering";
7     FBReference : INPUT STRING = "/TechUnits/pandix";
8   END_VARIABLE_VALUES;
9 END_INSTANCE;
```

Listing 5.1: Nutzung des PandIX Engineerings

Mithilfe einer Iterator-Aktion wird anschließend automatisch der Inhalt der PandIX-Daten ausgelesen und mit den hinterlegten Modellbausteinen des PandIX-Engineerings verglichen. So werden die aktuellen Anlagenteile (Pumpen, Ventile ...) erkannt und auf dem Bildschirm automatisch dargestellt. Aktoren wie Pumpen, Ventile sowie Behälter werden mit allen ihren Anschlusspunkten visualisiert. Die Rohrleitungen werden grafisch mit Linien dargestellt und verbinden die jeweils korrekten Anschlusspunkte. Die Messstellen (beispielsweise ein Temperatursensor) werden in der Anzeige platziert und durch Wirklinien mit der richtigen Stelle verbunden.

Soll das PandIX-Modell nun modifiziert werden, lassen sich aus der Bibliothek der PandIX-Bausteine neue Instanzen der Grafikbausteine erzeugen und in das HMI-Anlagenmodell integrieren. Der so genannte „Create Mode“ erlaubt weiterhin neue Anschlüsse (PandIX External Interfaces) an ein Objekt, z. B. einen Behälter, zu erstellen. Diese können auch per Drag und Drop zum Beispiel an den oberen oder unteren Rand des Objektes verschoben werden. Neben diesem Modus wurde

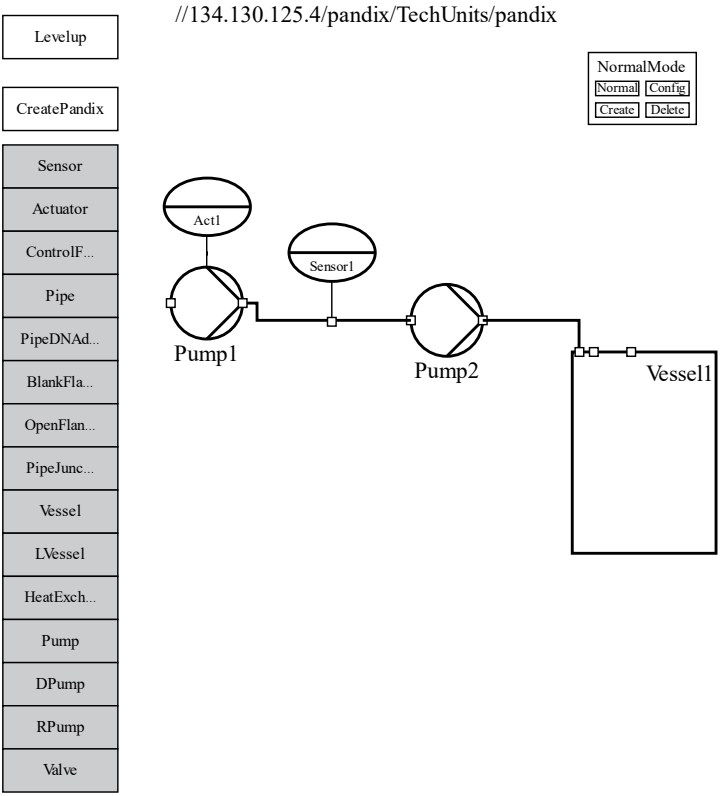

Abbildung 5.2: Engineeringoberfläche für PandIX (Klassenliste für die Abbildung gekürzt)

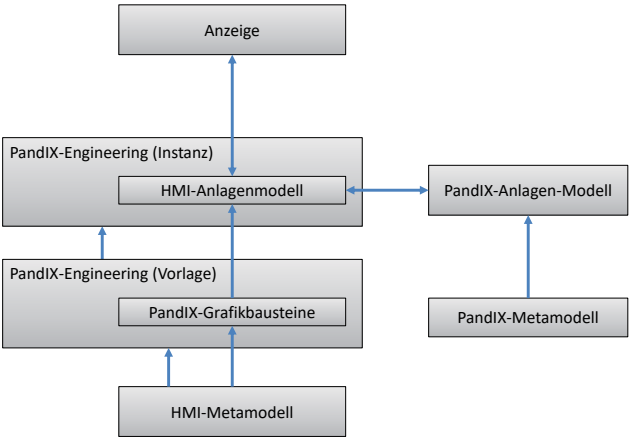

Abbildung 5.3: Struktur der PandIX-Engineeringoberfläche

ein weiterer Modus implementiert. So kann im „Config Mode“ ein Konfigurations-Faceplate eingeblendet werden, in dem das PandIX-Element über eine Tabelle konfiguriert werden kann.

Da das HMI-Anlagenmodell ein Abbild der Anlagendaten ist, werden synchron zur Darstellung die PandIX-Daten verändert. Abbildung 5.3 verdeutlicht noch einmal die beschriebenen Zusammenhänge zwischen dem in der Bedienoberfläche dargestellten Anlagenmodell, dem in den PandIX-Daten gespeicherten PandIX Anlagen-Modell und der Anzeige am Bildschirm.

Zusammenfassend kann festgehalten werden, dass die Anlagenplanung hohe Anforderungen an das vorgestellte Konzept stellt. Es konnten 33 verschiedene PandIX Grafikbausteine als Vorlage implementiert werden, sodass auch sehr komplexe Anlagen planbar sind und die Eignung des Modells für die Anlagenplanung damit gezeigt ist.

Eine genaue Betrachtung der Interna der Applikation befindet sich im Anhang 1 auf Seite 59.

5.3 Eignungen des Modells zur Simulationssteuerung

Eine wichtige Anwendung, welche im Lebenszyklus sowohl in der Planung und dem Betrieb genutzt wird, ist die Simulation einer Anlage. So existiert für die HART-Praktikumsanlage M3P.AC des Lehrstuhls[1] ein Simulator, der die gesamte Anlage nachbildet. Der Simulator verfügt über einen Profibus Anschluss. Am Leitsystem wird er als Siemens ET200M Remote-IO konfiguriert. So kann die gesamte Konfiguration mit dieser Installation geprüft werden. Die Simulation umfasst die ET200M, die 32 Sensoren und Aktoren sowie die Anlage inklusive der Produkte selbst. Um physikalisch

[1] `http://m3p.ac/`

realistisches Verhalten der Produkte zu erreichen wurde eine Simulation der Massen- und Energie-Gleichungen erstellt.

Verriegelungen in einer prozesstechnischen Anlage verhindern auf niedriger Ebene eine Gefahr für die Anlage sowie Geräte. So kann mit einer Verriegelung beispielsweise verhindert werden, dass eine Pumpe angeschaltet wird oder bleibt wenn ein Zielbehälter voll ist. Auch eine Aktivierung einer Pumpe, wenn ein Ventil im Flussweg vollständig geschlossen ist, kann so verhindert werden. Die Programmierung solcher Verriegelungen muss gründlich getestet werden, ist jedoch potenziell gefährlich, da die Anlage dabei naturgemäß im Grenzbereich arbeitet. Auch ist ein solcher Test zeitintensiv, da beispielsweise alle Behälter vollständig leer beziehungsweise voll sein müssen.

Bei der Entwicklung dieser Verriegelungen ist ein oben genannter Simulator sehr hilfreich, da diese Zustände simuliert wesentlich schneller erreicht werden können und außerdem keine Gefährdung der Anlage zu befürchten ist. Auch kann so die Anlagensteuerung früher fertiggestellt werden, da die echte Anlage noch nicht fertiggestellt sein muss. Entsprechend häufig werden Simulationen in der Anlagenplanung eingesetzt.

Ziel dieses Kapitels ist es daher zu zeigen, dass auch für die Steuerung von Simulationen eine Bedienoberfläche mit dem vorgestelltem Konzept erstellt und erfolgreich angewendet werden kann.

Evaluation

Für die vorhandene Simulation wurde im Rahmen dieser Dissertation eine Bedienoberfläche erstellt, welche beispielsweise die einzelnen Füllstände der simulierten Behälter verändern kann.

Wie auch im vorhergegangenen Kapitel wird in der Anzeige ein HMI-Simulationsmodell angezeigt, welches aus einzelnen Instanzen von Grafikbausteinen besteht und mit der eigentlichen Simulation (vergleichbar mit den PandIX Daten) im Austausch steht. Abbildung 5.4 verdeutlicht auch hier die Zusammenhänge zwischen der Anzeige, der Datenquelle und den Modellbausteinen.

Abbildung 5.5 zeigt die erstellte, statische Bedienoberfläche. Bei dieser wurden, im Gegensatz zum Beispiel des PandIX-Engineerings aus Kapitel 5.2, alle dargestellten Applikationselemente direkt programmiert. Über das `TimeEvent` (siehe Kapitel 3.4.3) werden regelmäßig zyklisch alle Simulationswerte geladen und an den entsprechenden Stellen angezeigt. Blaue Balkenanzeigen symbolisieren intuitiv den Füllstand der Behälter und wurden über die SetMathValue (siehe Kapitel 3.4.4) berechnet. Dabei wird über die Proportionalität aus den Grenzwerten der Behälterfüllstände die Höhe des Rechtecks dynamisch berechnet. Da jedes Form-Element auch rotiert werden kann, konnte auch ein Rotationszeiger einfach erstellt werden.

Die Ansicht stellt immer den Zustand der aktuell simulierten Anlage da. Für alle simulierten Anlageteile werden die Simulationsinformationen über ein Faceplate angezeigt und können verändert werden. So lassen sich die Temperaturen der Anlagekomponenten per Touchscreen modifizieren. Die Simulationssteuerung wird seit Jahren am Lehrstuhl im Rahmen von Praktika durch Studenten genutzt, sodass die Eignung des Modells zur Simulationssteuerung damit gezeigt ist.

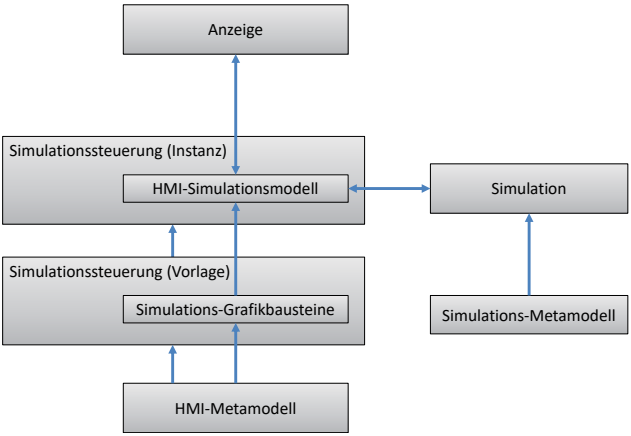

Abbildung 5.4: Zusammenarbeit der verschiedenen Modell-Bausteine für die Simulationssteuerung

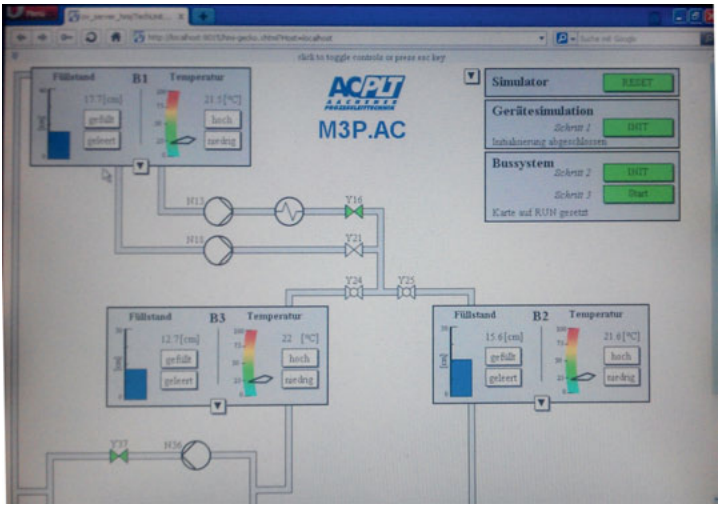

Abbildung 5.5: Pumpwerk Simulation

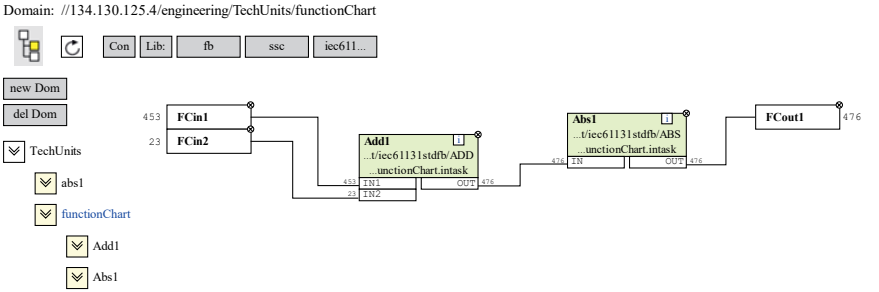

Abbildung 5.6: Engineeringoberfläche mit Continuous Function Chart

5.4 Engineering von Anlagensteuerungen

Während bislang gezeigt werden konnte, dass das Modell erfolgreich für Bedienoberflächen zur Anlagenplanung und Simulation eingesetzt werden kann, wird im nachfolgenden Kapitel das Erstellen einer Anlagensteuerung untersucht. So werden in der Prozessleittechnik Anlagen nur in den seltensten Fällen in Programmiersprachen wie C oder C# programmiert, sondern meist unter Zuhilfenahme einer der IEC61131 Sprachen sowie anwendungsspezifischer Sensor/Aktor Bausteinen. Siehe auch Kapitel 2.1. Entsprechend wurde im Rahmen dieser Dissertation geprüft, ob sich das Modell auch zur Erstellung von Engineeringwerkzeugen für eine Anlagensteuerung eignet. Im nachfolgenden Kapitel 5.4.1 wird daher auf die Eignung des Modells für ein Engineering mittels der Funktionsbausteinsprache, auch bekannt als Continuous Function Chart (CFC, [IEC03]), eingegangen, während sich Kapitel 5.4.2 dem Engineering mit der Ablaufsprache (Sequential Function Chart, SFC, [IEC03]) widmet.

Die Hauptphilosophie aller vorgestellten Engineering-Modelle ist übernommen von der gesamten ACPLT-Modelllandschaft: Das Automatisierungssystem ist die zentrale Datenbank des Prozesses („Wahrheit liegt im Zielsystem“ [Mey00]). Das Anzeigesystem speichert keine Daten, sondern liest und schreibt direkt ins Automatisierungssystem.

5.4.1 Engineering einer Funktionsbausteinsprache nach IEC 61131-3

Beim Engineering von Funktionsbausteinen wird das Zusammenspiel verschiedener Aktoren und Sensoren betrachtet, die über Operatoren miteinander verknüpft sind. Dabei sollten alle Funktionsbausteine mit ihren Variablen (inklusive der aktuellen Werte) angezeigt werden können, damit eine einfache und für den Benutzer leicht verständliche Verknüpfung der Bausteine untereinander möglich ist. Bei einem Additionsoperator würden also entsprechend Abbildung 5.6 zwei Eingangsgrößen (Volumenstrom 1 und Volumenstrom 2) sein und eine Ausgangsgröße (gesamter Volumenstrom) dem Benutzer mit Variablenname und Wert angezeigt werden. Bei einem Motorbaustein wären es zum Beispiel die Zielgeschwindigkeit, die Beschleunigungszeit sowie die Beschleunigungskurve als Eingangsvariable und die aktuelle Drehgeschwindigkeit als Ausgabevariable. Weiterhin sollte

die Anzeige ein Gruppieren der Bausteine ermöglichen, um auch bei komplexen Anlagen eine gute Übersichtlichkeit zu gewährleisten. Schließlich sollten diese Hierarchieebenen für den Benutzer einfach zugänglich sein.

Evaluation

Im Rahmen dieser Arbeit wurde das in Abbildung 5.6 dargestellte Engineering-Werkzeug für die Funktionsbausteinsprache, auch bekannt als Continuous Function Chart (CFC, [IEC03]), programmiert. Im Zentrum der Enineeringoberfläche sind zwei Elementarfunktionsbausteine zu sehen, ein Additionsbaustein (Add1) sowie ein Absolutbaustein (Abs), die miteinander über eine Variable verknüpft sind. Gemeinsam ergeben sie den Funktionsbaustein „functionChart“ der zwei Eingangsgrößen FCin1 und FCin2 hat sowie eine Ausgangsgröße FCout1. Neben den Eingangs-/ Ausgangsgrößen steht jeweils der aktuelle Wert der Variable. So wird im Additionsbaustein zu 453 der Wert 23 addiert und damit eine Variable mit dem Wer 476 an den Absolutbaustein übergeben. Dieser bildet den Betrag des Wertes und gibt diesen als Ausgangsvariable des FunctionChart-Bausteines wieder aus.

Weiterhin enthält die dargestellte Engineeringoberfläche links eine Anzeige der bislang verfügbaren Hierarchieebenen, wobei die einzelnen Ebenen durch Anklicken der Pfeil-Buttons aus- und einklappbar sind, sowie zahlreiche Buttons zur Modifizierung der Anzeige und damit der Anlagensteuerung. So erscheinen beim Klicken auf die Buttons *fb*, *SSC* sowie *iec61131stdb* eine Liste der in den entsprechenden Bibliotheken enthaltenen Funktionsbausteine. Wählt ein Benutzer einen neuen Baustein aus einer der Bibliotheken aus, so erscheint ein Konfigurationsfenster zur Definition des Bausteinnamens. Variablenanzahl, Variablenname sowie Typ sind bei jedem Baustein vordefiniert und werden dem Benutzer mit dem eingegeben Variablennamen im Anschluss auf dem Display an einer vordefinierten Position eingeblendet. Der Initiator der Abarbeitung (Task Parent) wird automatisch konfiguriert und auch im Kopfbereich angezeigt.

Die Position des Bausteins lässt sich anschließend per Drag und Drop frei anpassen und jederzeit verändern. Ebenso lässt sich nachträglich der Name durch Doppelklick auf die entsprechende Zeile editieren. Der Button *Con* ermöglicht wiederum dem Benutzer neue Verbindungen zwischen bestehenden Funktionsbausteinen zu erzeugen. Weiterhin können über den Button *Lib:* weitere Bibliotheken geladen werden, die im Anschluss dem Benutzer wie auch die bisher aktiven Bibliotheken im oberen Teil der Bedienoberfläche angeboten werden. Schließlich enthält die Oberfläche noch je einen Button zum Erzeugen und Löschen einer Hierarchieebene, einen Button zum Aktualisieren der Anzeige sowie einen Button zum Ausblenden der kompletten linken Baumstruktur. Diese letzte Funktion wurde implementiert, damit der Benutzer zu Dokumentationszwecken ein übersichtliches Abbild der Engineeringoberfläche drucken kann.

Wie auch in den vorhergegangenen Anwendungen wurde die Anzeige von allen vorhandenen Funktionsbausteinen und deren Variablen als Iterationsschleife implementiert. Diese findet beispielsweise einen Funktionsbaustein und erstellt die Visualisierung für diesen Baustein über eine parametrierte Kopiervorlage. Identisch wurde auch die Anzeige der nutzbaren Bibliotheken und deren Klassen abgefragt. Da die Baumansicht immer nur so weit aufgeklappt wird wie benötigt

und unsichtbare Teile der Anzeige vom System nicht weiter aufgebaut werden müssen (siehe Kapitel 3.5.1), ist die Bedienung auch bei sehr komplexen Hierarchiestrukturen noch performant. Weiterhin konnte durch die Realisierung der Bausteinverbindungen über Polygonlinien eine sehr hohe Gebrauchstauglichkeit erreicht werden. So kann ein Bediener zu jeder Zeit Bausteine per Drag und Drop (Event: `aftermove`) verschieben während die Polygonzüge dabei über die Aktion `RoutePolyline` zyklisch an die aktuelle Position der Verbindungspartner angepasst werden. Da für das menschliche Auge eine Neuberechnung alle 0,3 Sekunden ausreichend war, konnte auch diese Funktion sehr performant realisiert werden. Eine Schwachstelle wurde allerdings im Zuge der Implementierung erkannt. So war die Darstellung und Manipulation der Abarbeitungsreihenfolge nicht mit dem Konzept realisierbar. Dies wäre jedoch über eine Erweiterung der Aktion `linkObjects` möglich und solle in nachfolgenden Forschungsarbeiten daher Betrachtung finden.

5.4.2 Engineering einer Ablaufsprache nach IEC 61131-3

Neben der Funktionsbausteinsprache ist die Ablaufsprache Sequential Function Chart (SFC, [IEC03]) die weit verbreitete Programmiersprache für die Prozesstechnik. Im Gegensatz zur Funktionsbausteinsprache laufen dabei die Bausteine nicht parallel ab, sondern werden sequenziell abgearbeitet. Ziel dieses Kapitels ist die Eignung des Modells auch zur Erstellung eines Engineeringtools für die Anlagensteuerung mittels Ablaufsprache zu zeigen.

Evaluation

Der am Lehrstuhl entwickelte `SequencialControlChart` Funktionsbaustein ist ein dynamischer Funktionsbaustein, welcher „innen“ per Ablaufsprache programmiert wird [YGE13a, Yu16, YGE13b]. Für diesen Baustein wurde die in Abbildung 5.7 dargestellte Engineeringoberfläche erstellt. Um der starken Koppelung von CFC und SFC dabei Rechnung zu tragen wurde diese Bedienoberfläche direkt in das CFC-Engineering-Werkzeug des vorigen Kapitels 5.4.1 integriert.

Bei Auswahl eines `SequencialControlChart` Bausteines wird entsprechend ein spezieller Header in der Mitte der Bedienoberfläche erzeugt. Darunter werden die Bestandteile des Funktionsbausteines untereinander dargestellt: Schritte, Transitionen und Aktionen. In diesem Beispiel folgt auf den Startbaustein `INIT` eine Transition (trans1), die aktuell geschlossen ist (schwarze Kennzeichnung). Beim Öffnen würde das Programm zu Schritt 1 (*step1*) übergehen und die Eingangsvariablen von Funktionsbaustein *add1* (`add1.IN1 = 42` und `add1.IN2 = 23`) sowie die Eingangsvariable von Funktionsbaustein *abs1* (`abs1.IN1 = -5`) setzen. Nach Erfüllung der Bedingung in trans2 (aktuell gegeben), würde die Addition in *step2* ausgeführt werden. Der Baustein *abs1* würde durch diese Programmierung jedoch nicht angestoßen werden, da in *step2* nur die Ausführung des *add1*-Bausteins angestoßen wird. Da anschließend *transEnd* nicht geöffnet ist, würde die Routine in step2 stehen bleiben und auf eine Änderung der Transition warten.

Wie auch im CFC-Engineering-Werkzeug des vorigen Kapitels 5.4.1 wurden auch bei dieser Bedienoberfläche wieder Buttons zur Modifikation vorgesehen. So kann mit *AddStep* ein neuer Schritt generiert, mit *AddTrans* eine neue Transition (mit zwei Verbindungen) erzeugt werden und

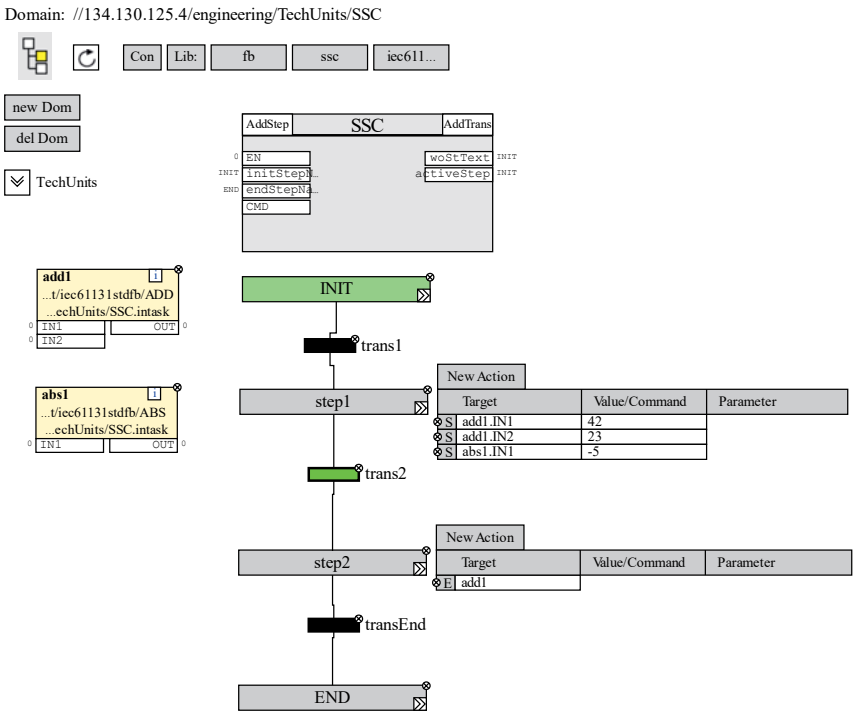

Abbildung 5.7: Engineeringoberfläche mit Sequential Control Chart

mit *new Action* eine Variable gesetzt (set: `S`) oder einen Baustein ausgeführt werden (execute: `E`). Des Weiteren kann über *EN* der Baustein aktiviert (enable) werden, über initStepName bzw endStepName der Startname/Endename der Routine gesetzt und über *CMD* ein Reset ausgelöst werden. Die Anzeigen *WoStText* und *activeStep* geben schließlich Auskunft über den aktuell aktiven Schritt. Im Hintergrund laufen auch bei dieser Applikation wieder zahlreiche Suchschleifen und Instanziierungsoperationen ab, die auf die SFC/SSC-Bausteine zugreifen. So wird beim SequencialControlChart erst (zum Ereignis `onload`) geprüft, wie der initiale Schritt heißt (dieser hat im Beispiel den Standardnamen INIT und wird in der Applikation im Header als initStepName angezeigt). Anschließend wird dieser per Iterator gesucht und die Kopiervorlage für einen Schritt erstellt. Daraufhin werden alle verbundenen Transitionen über die Assoziation *nextTransitions* gesucht und diese mithilfe einer Kopiervorlage dargestellt. Diese Kopiervorlage der Transition sucht nun über die Assoziation *previousTransitions* die nächsten Schritte und lädt auch hierfür jeweils eine Kopiervorlage. Werden Zustandsmaschinen beschrieben sind in Ablaufsprachen Schleifen sehr häufig. Um zu verhindern, dass die rekursive Analyse der Struktur in eine Endlosschleife läuft, wurde die Option `preventClone` (siehe Kapitel 3.5.3) implementiert. Müsste ein `InstantiateTemplate` ein exakt gleiches Darstellungs-Objekt (geprüft über den Namen der Referenz) erstellen, so wird die Erstellung des Duplikats abgebrochen und die Endlosschleife damit unterbrochen. Durch intensiven Einsatz von Iteratoren und `globalvarchanged`-Ereignissen konnte der Großteil der Anforderung an das SFC-Engineering erfüllt werden. Allerdings wird durch den Aufbau aus HMI-Modellbausteinen die Struktur der Software sehr komplex, sodass Sie nur schwierig zu warten ist.

5.5 Eignung für Bedienoberflächen im Betrieb

Auch während des Betriebs einer Anlage kommen Bedienoberflächen zum Einsatz. So muss der Bediener zu jeder Zeit einen schnellen Einblick in den aktuellen Zustand der Anlage erlangen. Hierfür gibt es von jedem Hersteller eines Prozessleitsystems eigene Tools um anwendungsspezifische Bedienoberflächen (das sogenannte *Bedienen und Beobachten*) zu erstellen.

Häufig ist die Interaktion dabei jedoch über eine Freitextprogrammierung gelöst. So stellt zum Beispiel *Honeywell* einen grafischen Editor zur Verfügung, mit dem man ein grafisches Abbild der Anlage erstellen kann. Die Kommunikation mit der Anlage, die Darstellung von Messwerten und die Interaktion muss jedoch in JavaScript frei programmiert werden. Entsprechend erfordert die Wartung und Anpassung solcher Bedienoberflächen in der Regel spezielle Programmierkenntnisse und ist nicht von Anwendern durchführbar. In diesem letzten Abschnitt des Kapitels wird daher untersucht, ob sich die Wartung und Anpassung von für den Betrieb vorgesehenen Bedienoberflächen durch einen modellbasierten Aufbau vereinfachen lässt, sodass auch Personen mit nur begrenzten Programmierkenntnissen Modifikationen vornehmen können.

Als Beispiel soll dabei die Prozessführung eines Elektro-Lichtbogenofens betrachtet werden, welcher aus Kühlsystemen, Hydraulik, Ofen und Gleichrichter besteht. Alle Sensoren und Aktoren benötigen dabei eine angepasste Darstellung mit farbigem Hinweis auf den aktuellen Zustand und die Möglichkeit der Interaktion. So sollte über die Bedienoberfläche nicht nur der aktuelle Zustand erkannt, sondern zum Beispiel die Pumpleistung einer Pumpe auch verändert werden können.

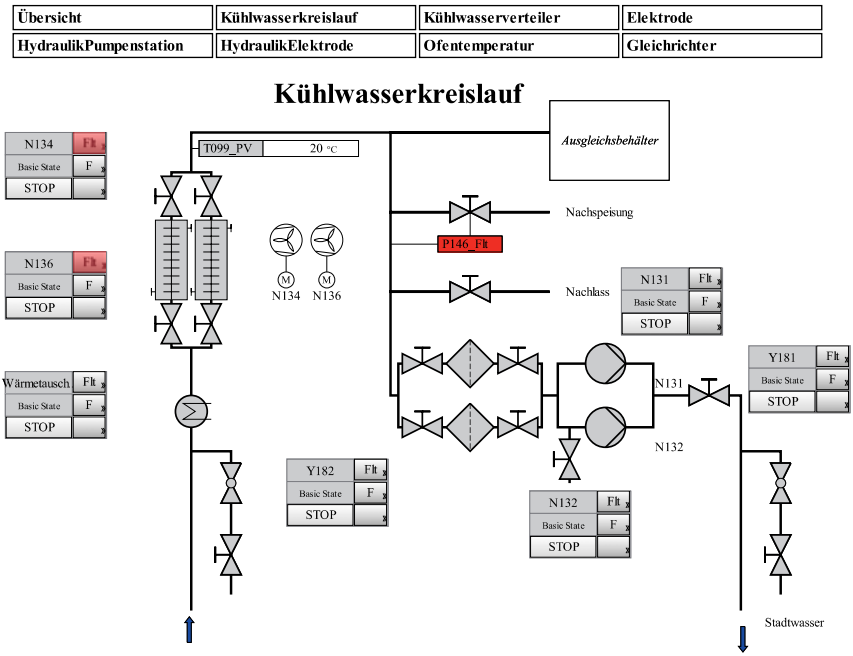

Abbildung 5.8: Bedienoberfläche einer komplexen Anlage

Evaluation

Für die Prozessführung des Elektro-Lichtbogenofens wurde die in Abbildung 5.8 dargestellte Bediensoftware erstellt. Im oberen Bereich hat der Benutzer dabei die Möglichkeit sich verschiedene Bereiche der Anlage auf den Bildschirm zu holen. Im Screenshot ist der Kühlwasserkreislauf ausgewählt, dessen Komponenten im rechten Bereich der Bedienoberfläche in Form eines R&I-Fließbildes dargestellt werden. Entsprechend sind hier Darstellungen für alle verfahrenstechnischen Anlagenteile (wie Pumpe oder Ventil) vorhanden. Klickt man auf eine dieser Darstellungskomponenten, so öffnet sich ein passendes Faceplate.

Neben diesen Anlagenkomponenten werden dem Benutzer Faceplates zu besonders wichtigen Anlagenkomponente dauerhaft dargestellt. So besteht zum Beispiel ein Prozessführungsbaustein für die Motoren (N134 und N136) des Kühlkreislaufes. Hier werden dem Benutzer zu jeder Zeit Fehlermeldungen über das Submenü *Flt* angezeigt und direkte Möglichkeiten der Interaktion geboten, wie zum Beispiel das Anhalten über einen STOP-Knopf. Weiterhin beinhaltet die Bedienoberfläche Anzeigefelder mit aktuellen Sensorwerten. So wird von einem Temperatursensor *T099_PV* aktuell eine Kühlmitteltemperatur von 20 °C gemessen und dem Bediener darstellt.

Die Prozessführungsbausteine der Steuerung wurden über ein Vererbungssystem erstellt, sodass ihre Grundstruktur immer gleich aufgebaut ist und sich nur in spezialisierten Diensten unterscheidet (siehe [YQE10]). Dies wurde auch in der Bedienoberfläche genutzt. So haben alle Prozessführungsbausteine ein gemeinsames Faceplate und damit eine ähnliche Anzeige. Hier wurde das gleiche Anzeigemodell der Prozessführungsbausteine genutzt wie auch schon in Kapitel 5.1.

Das Faceplate muss kaum parametriert werden, sondern analysiert die jeweiligen Fähigkeiten selbst und stellt entsprechend angepasste Bedienelemente dar. Diesem generischen Ansatz kommt zugute, dass alle Prozessführungsbausteine über eine Ausgangs-Variable die jeweils unterstützten Kommandos zur Verfügung stellen. Diese Kommandos werden über die Aktion *ChildrenIterator* ausgelesen und den Bediener als Schaltflächen präsentiert. Nachdem diese Vorarbeit einmalig erstellt wurde, ist das Erstellen der Bedienoberfläche sehr einfach, da pro Prozessführungselement ein identisches Template angesprochen wird. Listing 5.5 gibt einen kleinen Einblick in den Programmiercode. So wird das generische Faceplate (Zeile 6) an einer Position erstellt und ein Prozessführungsbaustein referenziert (Zeile 7).

```
1 INSTANCE    /TechUnits/cshmi/ElboMainSheet/hydraulicPumpUnit/bubFrame/Pumpstation/GCU010 :
2     CLASS  /acplt/cshmi/Group;
3     VARIABLE_VALUES
4       x : INPUT SINGLE = 200.000000;
5       y : INPUT SINGLE = 200.000000;
6       TemplateDefinition : INPUT STRING = "Processcontrol/FaceplatePCUGeneric";
7       FBReference : INPUT STRING = "TechUnits/P30/IC10/PU10/PS20/TU10/GCU010";
8     END_VARIABLE_VALUES;
9 END_INSTANCE;
```

Schließlich wurde eine Alarmtabelle programmiert, in der der Status von 81 möglichen Alarmen dargestellt wird. Diese lässt sich ebenfalls über die Bedienoberfläche aufrufen und stellt dem Benutzer eine Liste der aller aktuellen Fehlermeldungen zur Verfügung.

Die Alarmtabelle fragt zyklisch den aktuellen Alarmzustand der benötigten Prozessführungsbausteine ab. Im Fehlerfall sollte die Applikation die jeweiligen Einträge in der Tabelle blinken lassen. Diese Anforderung erforderte eine Synchronisation aller TimeEvents. Andernfalls war es sehr störend, dass jeder Alarm zwar in der gleichen Frequenz blinkte, jedoch zu den Anderen phasenverschoben war. Die Synchronisation erreichte, dass alle `TimeEvents` (beziehungsweise dessen Aktionen) basierend auf der jeweiligen Zykluszeit gleichzeitig abgearbeitet wurden. So wurde erreicht, dass der Farbwechsel des Blinkens gleichzeitig auf dem Bildschirm sichtbar wurde. Zusätzlich hatte diese Änderung den Vorteil, dass die jeweilige Kommunikation mit dem Automatisierungssystem in eine Anfrage gebündelt werden konnten und damit weniger Ressourcen verbraucht.

In Abbildung 5.9 ist die Struktur der Modell-Bausteine dargestellt. Die hier gezeigten Prozessführungs-Grafikbausteine sind dieselben, welche auf in Kapitel 5.1 genutzt wurden. Hier ist nur die Bedienoberfläche selbst vollständig händisch erzeugt worden.

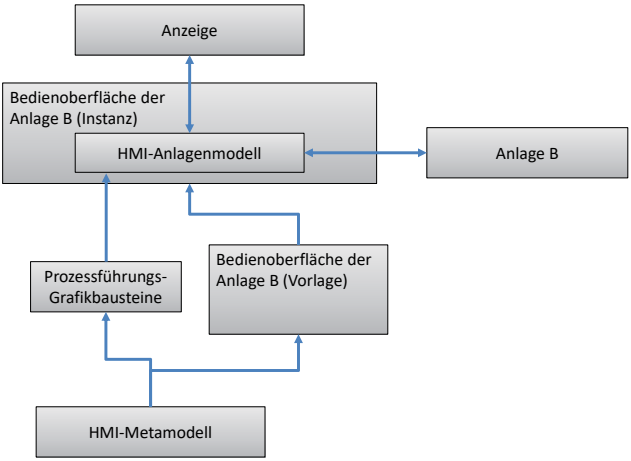

Abbildung 5.9: Zusammenarbeit der verschiedenen Modell-Bausteine für das Bedienen und Beobachten

5.6 Integration von fremden Bibliotheken in die Modellstruktur

Während in den bisherigen Kapiteln der weite Einsatzbereich des entwickelten Modells gezeigt werden konnte, soll in diesem Kapitel die Offenheit des Modells für die Integration fremder Bibliotheken gezeigt werden.

So wurde für die Blackbox (siehe Kapitel 3.4.5) ein einfach zu verwendendes x(t)-Diagramm erstellt und damit die einfache Nutzbarkeit von vorhandenen JavaScript-Bibliotheken gezeigt. Hierfür wurde die Bibliothek „Smoothie Charts“[2] eingebunden. Diese erlaubt es, wie in Abbildung 5.10 dargestellt, Live-Daten direkt anzuzeigen. Das Template wurde so angelegt, dass die Interna von Smoothie komplett gekapselt wurden. Der Nutzer kann in der Objektwelt der gewohnten Applikation bis zu zehn Werte des Automatisierungssystem referenzieren. Weiterhin ist es möglich, den minimalen und maximalen Wert, die Farben der Beschriftung und die Laufgeschwindigkeit der Anzeige zu manipulieren.

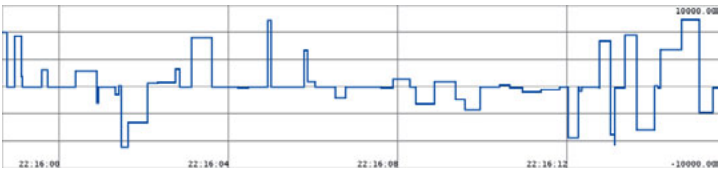

Abbildung 5.10: Beispieldarstellung der JavaScript-Bibliothek „Smoothie Charts“

Nachfolgende Listings geben einen Einblick in die Programmierung.

[2] http://smoothiecharts.org/

```
1  INSTANCE  /TechUnits/cshmi/xtshowcase/xtdiagram :
2      CLASS /Libraries/cshmi/Group;
3      VARIABLE_VALUES
4        x : INPUT SINGLE = 0.000000;
5        y : INPUT SINGLE = 0.000000;
6        width : INPUT SINGLE = 1000.000000;
7        height : INPUT SINGLE = 900.000000;
8        TemplateDefinition : INPUT STRING = "xtdiagram";
9        FBReference : INPUT STRING = "";
10       FBVariableReference[1] : INPUT STRING = {"data1:/TechUnits/TU/random.OutS"};
11       ConfigValues[4] : INPUT STRING = {
12         "minValue:-1000" , "maxValue:1000" , "velocity:10" , "cycTime:0.05"
13       };
14     END_VARIABLE_VALUES;
15 END_INSTANCE;
```

Listing 5.2: Parametrierung des x(t)-Diagrams

So lassen sich zum Beispiel Sensordaten einer Anlage in Echtzeit mit der Konfiguration aus Listing 5.2 auf dem Bildschirm anzeigen. Die Positionierung und Größendefinition erfolgt per `x`, `y`, `width` und `height`. Der Parameter `TemplateDefinition` referenziert die Kopiervorlage mit dem entsprechenden Namen. Der sonst viel genutzte Parameter `FBReference` wird hier nicht genutzt, da dieser eine Objekt-Referenz entgegennimmt. Stattdessen wird auf den Parameter `FBVariableReference` gesetzt, da dieser mehrere benannte Variablen adressieren kann. Im Beispiel ist das ein Baustein, welcher Zufallswerte liefert. Der Name `data1` muss äquivalent zu den Interna der Kopiervorlage übergeben werden. Hier ist wie oben angedeutet `data1` bis `data10` möglich. In dem Parameter `ConfigValues` erfolgt die erwähnte optionale Konfiguration.

```
1 INSTANCE  /TechUnits/cshmi/Templates/xtdiagram/diagrammagic/jsMinValue.value :
2     CLASS /acplt/cshmi/GetValue;
3     VARIABLE_VALUES
4       TemplateConfigValues : INPUT STRING = "minValue";
5     END_VARIABLE_VALUES;
6 END_INSTANCE;
```

Listing 5.3: Koppelung der Variablen der JavaScriptwelt mit der Modellwelt

Das Listing 5.3 zeigt einen Ausschnitt der Koppelung der Modell-Welt mit der JavaScript-Welt. Auf die erwähnte `ConfigValue` `minValue` der Kopiervorlage (Zeile 4: `TemplateConfigValue`) kann per JavaScript lesend (Zeile 2: `GetValue`) über den Namen `jsMinValue` (Zeile 1) zugegriffen werden.

```
var minValue = 0;
if(cshmimodel.variables.jsMinValue){
  var temp = parseFloat(cshmimodel.variables.jsMinValue.getValue());
  if(!isNaN(temp)){
    minvalue = temp;
  }
}
```

Listing 5.4: Parametrierung des x(t)-Diagrams

Schließlich gibt Listing 5.4 einen Einblick in den Code innerhalb der Blackbox. Zuerst werden Standardwerte in eine Variable geschrieben und daraufhin mithilfe der Blackbox-API (siehe Anhang 3) geprüft, ob der Parameter `jsMinValue` vorhanden ist und eine gültige Zahl repräsentiert. In diesem Fall wird dieser Wert übernommen und im Anschluss an SmoothieCharts übergeben.

5.7 Fazit

Zusammenfassend hat die Evaluation des Modells ergeben, dass sich das Konzept auf klassische Applikationen der Prozessleittechnik wie die Simulationssteuerung oder das „Bedienung und Beobachten“ gut anwenden lässt.

Der in dieser Arbeit vorgestellte Ansatz der Modellierung vereinfachte weiterhin durch sein Vorlagensystem die automatische Erstellung der Bedienoberfläche. Hier konnten durch die gemeinsame Nutzung der Visualisierung aller benötigten Prozessführungsbausteine auch bei der Evaluation „Bedienung und Beobachten“ direkt Synergieeffekte genutzt und Engineeringaufwand eingespart werden.

Selbst die Erweiterung um den Freitextprogrammier-Baustein fügt sich schlüssig ins Gesamtkonzept ein. Somit kann das Modell für mehr grafisch komplexere Applikationen genutzt werden. Der Anwender benötigt dabei keine JavaScript-Kenntnisse und kann schnell zu einem zufriedenstellenden Ergebnis gelangen. Dass er bei der Nutzung dieses Bausteins die Plattformunabhängigkeit verliert, ist er Nachteil, den er explizit in Kauf nehmen muss.

Muss eine Applikation jedoch in einer sehr stark dynamischen Umgebung oder mit sehr komplexen Benutzerinteraktion arbeiten, so kommt das Modell an seine Grenzen. Dies war beispielsweise bei den Engineering-Applikationen der Fall gewesen. Hier wird auch die Wartbarkeit der Applikation eine Herausforderung und erfordert gute Dokumentation der Applikationsinterna.

6 Diskussion und Ausblick

Ziel der Arbeit war die Evaluation eines neuartigen Konzepts der Modellierung einer Benutzungsschnittstelle für die Prozessleittechnik. Dieses sieht die Trennung in drei Komponenten vor: Die Datenbasis (1) speichert das vollständige Modell der fertigen Benutzungsschnittstelle. Das Anzeigesystem (2) lädt und interpretiert dieses Modell und kommuniziert bei Bedarf mit dem Automatisierungssystem (3).

Im Gegensatz zu vorhandenen Modellierungen werden in dem vorgestelltem Ansatz nicht nur alle grafischen Elemente, sondern auch alle Ereignisse und dazugehörigen Aktionen als atomare Komponenten einzeln modelliert. So wurden neben den grundlegenden Grafikelementen (wie Text, Rechteck, Kreis) auch die wichtigsten Benutzerinteraktionen und ausgewählte übergeordnete Ereignisse (wie einmalige oder zyklische Abarbeitung) in das Metamodell mit aufgenommen. Weiterhin wurden Aktionen zur Manipulation der Anzeige sowie des Automatisierungssystems festgelegt. Schließlich wurde im Metamodell für wiederkehrende Fragmente der Applikation auch ein parametrierbarer Kopiervorlagen-Mechanismus entwickelt. So wurden wiederverwendbare Elemente wie ein Button aus einem Rechteck und einem Text (inklusive Logik zur Veränderung der Textinhaltes) erstellt und für alle späteren Applikationsentwickler in einer Kopiervorlage hinterlegt. Somit wird eine Applikation vollständig technologieneutral modelliert.

Da die ganze Bedienoberfläche modellbasiert vorliegt, ist sie prädestiniert, um selbst auf der Grundlage von beliebigen Regelwerken erstellt oder verändert zu werden. Das Modell bietet jedoch auch umfangreiche Möglichkeiten der Strukturanalyse und -manipulation des Automatisierungssystems. Dies ermöglicht es auf der einen Seite Applikationen zu entwickeln, welche je nach analysiertem Kontext eine angepasste Darstellung zeigt. Auf der anderen Seite können jedoch auch komplexe Engineering-Werkzeuge erstellt werden.

Die reine Beschränkung des Modells auf atomare Elemente erleichtert die Implementierung auf unterschiedliche Plattformen. So ist es einfach möglich, die Anzeigekomponente zu realisieren, da nur 11 Elemente, 8 Ereignisse sowie 14 Aktionen implementiert werden müssen. Dies gewährleistet, dass eine Applikation über die gesamte Lebensdauer einer Anlage nutzbar ist, selbst wenn zwischendurch die eingesetzte Technologie gewechselt werden muss.

Um das Modell für ein größeres Anwendungsspektrum nutzbar zu machen, wurde mit dem Baustein `Blackbox` eine Möglichkeit geschaffen komplexere Darstellungen mit der Beschreibungssprache HTML und/oder komplexe Logiken mit der Programmiersprache JavaScript zu erstellen. Dabei wird jedoch die Plattformunabhängigkeit potenziell verlassen.

Das Metamodell wurde prototypisch in der Laufzeitumgebung des Lehrstuhls für Prozessleittechnik ACPLT/RTE implementiert. So konnte gezeigt werden, dass die Speicherung eines Applikationsmodells möglich ist und dem Anzeigesystem bereitgestellt werden kann. Für die Darstellung selbst wurde auf Webtechnologie gesetzt, um eine geforderte Plattformunabhängigkeit zu erreichen. Dazu sind performante Browser für Desktopbetriebssysteme wie Windows, Linux, macOS und auch für Mobilbetriebssysteme wie Android oder iOS verfügbar. Die Kommunikation mit dem Automatisierungssystem wurde per ACPLT/KS auf HTTP-Basis realisiert.

Die Tauglichkeit des Modells für unterschiedlichste Aufgaben wurde mithilfe von Beispielanwendungen über den gesamten Lebenszyklus einer Anlage geprüft. Es konnten alle Anwendungen erstellt werden. Weiterhin ließ sich feststellen, dass der Ansatz der vollständigen Modellierung mittels atomarer Bausteine für einfache Bedienoberflächen sehr gut nutzbar ist. Insbesondere, wenn für die zu visualisierenden leittechnischen Funktionen Kopiervorlagen existieren ist die Nutzung sehr vorteilhaft. Da die Kopiervorlagen in der gleichen Technologie implementiert sind, ist es weiterhin sehr einfach diese an eigene Wünsche anzupassen. Für die Editor-Unterstützung bei der Programmierung des Bausteins zur Freitext-Programmierung (`Blackbox`) konnte mit einer TypeScript-Definitionsdatei eine sehr einfache Nutzung erreicht werden. Bei komplexeren Applikationen wurden die Grenzen des Konzepts jedoch deutlich. Hier wäre eine bessere Software-Unterstützung der Anwendungsentwickler wünschenswert.

Zudem zeigte die Evaluation Optimierungspotential in der Modellierung selbst. Die Beschränkung des Metamodells auf wenige atomare Elementarbausteine für Grafikelemente, Ereignisse und Aktionen ist ein Vorteil in der Entwicklung der Grundlagensoftware führte jedoch zum Nachteil in der aufwändigeren Programmierung bei der Erstellung der Applikationen selbst.

Hier ist zu entscheiden, ob in späteren Arbeiten dieser Ansatz etwas aufgeweicht wird und ausgewählte abstraktere Elemente, Ereignisse und Aktionen hinzugefügt werden. Während der Entwicklung wurden beispielsweise die folgenden grafischen Elemente als Kandidaten einer Erweiterung identifiziert:

- Auswahllisten
- Radio-Buttons / Checkboxen

Genau für diese grafischen Elemente zugeschnitten wäre ein Ereignis, welche nach einer erfolgten Bedienerauswahl ausgelöst wird. Die Einbindung dieser Auswahl ist über die bisherige Philosophie problemlos möglich. So würde zum Beispiel ein GetValue-Baustein diese Auswahl als weitere Datenquelle, ähnlich wie aktuell bei einer Mausposition, erhalten.

Eine weitere potenzielle Erleichterung zur Erstellung von Bedienoberflächen wäre die Entwicklung eines Import-Werkzeugs. Dieses könnte Vektorgrafiken einlesen und unterstützte Grafikelemente in HMI-Bausteine überführen. Diese würde anschließend als Basis für eine Applikation dienen, indem die Geschäftslogik in Form von Ereignissen und Aktionen ergänzt würde.

Zudem wird in der Prozessleittechnik aktuell das Konzept der Diensteorientierung vorangetrieben ([WE17]). Hierfür wäre eine Aktion zum „Dienstbefehl absetzen“ eine interessante Erweiterung des Metamodells.

Schließlich wäre zur weiteren Evaluation des Ansatzes eine Entwicklung eines alternativen Anzeigesystems, beispielsweise in der Programmiersprache *C#* mit *Windows Presentation Foundation (WPF)* oder *C++* mit *Qt*, wünschenswert. Ob diese Implementierung den Modell-Baustein für die Freitextprogrammierung (`Blackbox`) unterstützt wäre zu entscheiden.

Anhang

1 Anwendung R&I-Fließschema-Editor im Detail

In diesem Anhang wird eine umfangreiche Anwendung im Detail vorgestellt, um dem Leser einen Eindruck der prototypischen Implementierung des in dieser Arbeit vorgestellten Modells zu geben.

Die Applikation bietet eine Kopiervorlage als Haupteinsprungpunkt namens `Pandix/PandixEngineering` . Dieses muss nur in einer Gruppe referenziert werden, siehe Kapitel 3.4.2 und Listing 1 in Zeile 6. Zeile 7 definiert, dass das PandIX Modell unter „/TechUnits/pandix“ visualisiert werden soll.

```
1  INSTANCE   /TechUnits/cshmi/engineeringPandIXSheet :
2      CLASS  /acplt/cshmi/Group;
3      VARIABLE_VALUES
4        width  : INPUT SINGLE = 1675.000000;
5        height : INPUT SINGLE = 1020.000000;
6        TemplateDefinition : INPUT STRING = "Pandix/PandixEngineering";
7        FBReference : INPUT STRING = "/TechUnits/pandix";
8      END_VARIABLE_VALUES;
9  END_INSTANCE;
```

Listing 1: Nutzung des PandIX Engineerings

Die Applikation durchsucht über einen Iterator zuerst einmal die Assoziation `/acplt/ov/library.instance` um die `pandix`-Bibliothek zu finden. Diese iteriert wiederum über alle Klassen um über eine Positivliste die gewünschten Klassen zu erhalten:

```
1  INSTANCE   /TechUnits/cshmi/Templates/Pandix/PandixEngineering/CreateObjectButton/onload/pandixPathIterator.forEachChild/
       If_Found.then/ClassIterate.forEachChild/If.if/Permitlist.withValue :
2      CLASS  /acplt/cshmi/GetValue;
3      VARIABLE_VALUES
4        value[25] : INPUT STRING = {"Actuator" , "BlankFlange" , "CheckValve" , "ControlFunction" , "Connector" , "DPump" , "
             GeneralItem" , "HeatExchanger" , "HeatSource" , "LVessel" , "OpenFlange" , "Pipe" , "PipeDNAdapter" , "PipeJunction"
              , "Pump" , "RPump" , "RuptureDisk" , "SafetyValve" , "Sensor" , "ThreeWayValve" , "Valve" , "Vessel" , "
             PlantSection" , "ProcessPlant" , "IndustrialComplex"};
5      END_VARIABLE_VALUES;
6  END_INSTANCE;
```

Für alle diese Klassen wird anschließend ein neuer Button (Zeile 4) per Aktion InstantiateTemplate erzeugt. Der erste Button wird an die Position x:0, y: 40 Pixel (Zeile 5 und 6) erstellt. Alle Weiteren werden um 30 Pixel nach unten (Zeile 8) versetzt. Jeder Button erhält als FBReferenz (siehe Kapitel 3.4.2) das aktuelle Objekt des Iterators:

```
1  INSTANCE   /TechUnits/cshmi/Templates/Pandix/PandixEngineering/CreateObjectButton/onload/pandixPathIterator.forEachChild/
       If_Found.then/ClassIterate.forEachChild/If.then/Inst_Button :
2      CLASS  /acplt/cshmi/InstantiateTemplate;
3      VARIABLE_VALUES
4        TemplateDefinition : INPUT STRING = "Pandix/internal/PandixClassButton";
5        x : INPUT SINGLE = 0.000000;
6        y : INPUT SINGLE = 40.000000;
7        xOffset : INPUT SINGLE = 0.000000;
8        yOffset : INPUT SINGLE = 30.000000;
9        FBReference : INPUT STRING = "OP_NAME";
```

```
10      END_VARIABLE_VALUES;
11  END_INSTANCE;
```

Mit dieser Referenz kann ein neues PandIX-Objekt beim Klick auf diesen Button erstellt werden:

```
1   INSTANCE   /TechUnits/cshmi/Templates/Pandix/internal/PandixClassButton/click/actionCreate :
2       CLASS /acplt/cshmi/CreateObject;
3   END_INSTANCE;
4   INSTANCE   /TechUnits/cshmi/Templates/Pandix/internal/PandixClassButton/click/actionCreate.Name :
5       CLASS /acplt/cshmi/GetValue;
6       VARIABLE_VALUES
7         OperatorInput : INPUT STRING = "textinput:Please enter the name for the new object";
8       END_VARIABLE_VALUES;
9   END_INSTANCE;
10  INSTANCE   /TechUnits/cshmi/Templates/Pandix/internal/PandixClassButton/click/actionCreate.Place :
11      CLASS /acplt/cshmi/GetValue;
12      VARIABLE_VALUES
13        globalVar : INPUT STRING = "RefDomain";  # Hier ist der aktuell angezeigte Pfad
14      END_VARIABLE_VALUES;
15  END_INSTANCE;
16  INSTANCE   /TechUnits/cshmi/Templates/Pandix/internal/PandixClassButton/click/actionCreate.Library :
17      CLASS /acplt/cshmi/GetValue;
18      VARIABLE_VALUES
19        TemplateFBReferenceVariable : INPUT STRING = "CSHMIfullqualifiedparentname"; # Pfad zur PandIX-Bibliothek
20      END_VARIABLE_VALUES;
21  END_INSTANCE;
22  INSTANCE   /TechUnits/cshmi/Templates/Pandix/internal/PandixClassButton/click/actionCreate.Class :
23      CLASS /acplt/cshmi/GetValue;
24      VARIABLE_VALUES
25        TemplateFBReferenceVariable : INPUT STRING = "identifier"; # Name der Klasse des Buttons
26      END_VARIABLE_VALUES;
27  END_INSTANCE;
```

Ein weiterer Iterator analysiert das aktuell anzuzeigende „Verzeichnis“. Hier ist der Ausschnitt abgebildet, welcher für die Pumpenklasse `Pump` eine Anzeige für eine Pumpe erstellt.

```
1   INSTANCE   /TechUnits/cshmi/Templates/Pandix/internal/activeView/onload/PandixIterator.forEachChild/If_Pump.if/If_Pump :
2       CLASS /acplt/cshmi/CompareIteratedChild;
3       VARIABLE_VALUES
4         childValue : INPUT STRING = "OP_CLASS";
5         comptype : INPUT STRING = "==";
6       END_VARIABLE_VALUES;
7   END_INSTANCE;
8   INSTANCE   /TechUnits/cshmi/Templates/Pandix/internal/activeView/onload/PandixIterator.forEachChild/If_Pump.if/If_Pump.
        withValue :
9       CLASS /acplt/cshmi/GetValue;
10      VARIABLE_VALUES
11        value[1] : INPUT STRING = { "/acplt/pandix/Pump"};
12      END_VARIABLE_VALUES;
13  END_INSTANCE;
14  INSTANCE   /TechUnits/cshmi/Templates/Pandix/internal/activeView/onload/PandixIterator.forEachChild/If_Pump.then/Inst_Pump :
15      CLASS /acplt/cshmi/InstantiateTemplate;
16      VARIABLE_VALUES
17        TemplateDefinition : INPUT STRING = "Pandix/Pump";
18        x : INPUT SINGLE = 100.000000;
19        y : INPUT SINGLE = 300.000000;
20        xOffset : INPUT SINGLE = 100.000000;
21        yOffset : INPUT SINGLE = 100.000000;
22        maxTemplatesPerDirection : INPUT STRING = "x:10";
23        FBReference : INPUT STRING = "OP_NAME";
24      END_VARIABLE_VALUES;
25  END_INSTANCE;
```

In Zeile 20 bis 22 ist definiert, dass zehn gefundene Pumpen jeweils um 100 Pixel (Zeile 22) seitlich verschoben werden. Die 11. bis 20. Pumpe würden eine Zeile drunter bilden (siehe Kapitel 3.5.3). Mit Zeile 23 wird die Anzeige der Pumpe mit dem PandIX-Datenobjekt verknüpft. Eine ähnliche Logik existiert für alle weiteren anzuzeigenden Elemente.

Neben den PandIX-Elementen (in PandIX/CAEX übergreifend PPE_Request genannt) werden auch für PandIX-Verbindungen (CAEX InternalLink) zwischen den einzelnen PandIX-Elementen jeweils grafische Elemente erstellt. Da diese auch im HMI mit den PandIX-Verbindungen verknüpft

sind, können die logischen Verbindungen als Polylinie automatisch per `routePolyline` positioniert werden.

Wird ein Hierarchieelement entdeckt, so wird eine Option angeboten die Ansicht in diese tiefere Hierarchie zu wechseln.

```
1  INSTANCE   /TechUnits/cshmi/Templates/Pandix/internal/activeView/globalvarchanged/If_activePandix.then/Set_reference :
2     CLASS /acpIt/cshmi/SetValue;
3     VARIABLE_VALUES
4        TemplateFBReferenceVariable : INPUT STRING = "fullqualifiedname";
5     END_VARIABLE_VALUES;
6  END_INSTANCE;
7  INSTANCE   /TechUnits/cshmi/Templates/Pandix/internal/activeView/globalvarchanged/If_activePandix.then/Set_reference.value :
8     CLASS /acpIt/cshmi/GetValue;
9     VARIABLE_VALUES
10       globalVar : INPUT STRING = "activePandix";
11    END_VARIABLE_VALUES;
12 END_INSTANCE;
13 INSTANCE   /TechUnits/cshmi/Templates/Pandix/internal/activeView/globalvarchanged/If_activePandix.then/reload :
14    CLASS /acpIt/cshmi/RebuildObject;
15 END_INSTANCE;
```

Der Hierarchiewechsel wird über ein Überschreiben der FBReferenz (Zeile 4) der Hauptanzeige `activeView` mit dem neuen Pfad (Zeile 10) erreicht. Anschließend wird in Zeile 14 die Hauptanzeige neu geladen.

In dieser Applikation werden auch die verschiedenen Rohrleitungen und Wirklinien automatisch geroutet:

```
1  INSTANCE   /TechUnits/cshmi/Templates/Pandix/Pipe/Polyline :
2     CLASS /acpIt/cshmi/Polyline;
3     VARIABLE_VALUES
4        points : INPUT STRING = "0,0 0,0";
5        strokeWidth : INPUT SINGLE = 2.000000;
6        stroke : INPUT STRING = "black";
7     END_VARIABLE_VALUES;
8  END_INSTANCE;
9  INSTANCE   /TechUnits/cshmi/Templates/Pandix/Pipe/Polyline/Time :
10    CLASS /acpIt/cshmi/TimeEvent;
11    VARIABLE_VALUES
12       cyctime : INPUT SINGLE = 0.500000;
13    END_VARIABLE_VALUES;
14 END_INSTANCE;
15 INSTANCE   /TechUnits/cshmi/Templates/Pandix/Pipe/Polyline/Time/RouteLine :
16    CLASS /acpIt/cshmi/RoutePolyline;
17    VARIABLE_VALUES
18       offset : INPUT SINGLE = 10.000000;
19       gridWidth : INPUT SINGLE = 5.000000;
20    END_VARIABLE_VALUES;
21 END_INSTANCE;
22 INSTANCE   /TechUnits/cshmi/Templates/Pandix/Pipe/Polyline/Time/RouteLine.SourceBasename :
23    CLASS /acpIt/cshmi/GetValue;
24    VARIABLE_VALUES
25       TemplateFBReferenceVariable : INPUT STRING = "/PIn.SideA";
26    END_VARIABLE_VALUES;
27 END_INSTANCE;
28 INSTANCE   /TechUnits/cshmi/Templates/Pandix/Pipe/Polyline/Time/RouteLine.SourceVariablename :
29    CLASS /acpIt/cshmi/GetValue;
30    VARIABLE_VALUES
31       value : INPUT VOID = ;
32    END_VARIABLE_VALUES;
33 END_INSTANCE;
34 INSTANCE   /TechUnits/cshmi/Templates/Pandix/Pipe/Polyline/Time/RouteLine.TargetBasename :
35    CLASS /acpIt/cshmi/GetValue;
36    VARIABLE_VALUES
37       TemplateFBReferenceVariable : INPUT STRING = "/POut.SideA";
38    END_VARIABLE_VALUES;
39 END_INSTANCE;
40 INSTANCE   /TechUnits/cshmi/Templates/Pandix/Pipe/Polyline/Time/RouteLine.TargetVariablename :
41    CLASS /acpIt/cshmi/GetValue;
42    VARIABLE_VALUES
43       value : INPUT VOID = ;
44    END_VARIABLE_VALUES;
45 END_INSTANCE;
```

Die Polylinie der Rohrleitung wird zyklisch zweimal die Sekunde (Zeile 12) neu berechnet. Dabei werden die jeweiligen Verbindungspartner beim PandIX-Pipe-Objekt unter den Namen `PIn.SideA` und `POut.SideA` gesucht.

2 Interner Aufbau der Anzeigekomponente

Der aktuelle Quelltext der Anzeigekomponente kann auf der Github-Webseite des Lehrstuhls `https://github.com/acplt/rte` unter dem Pfad `/addonlibs/hmi/hmiJavaScript` eingesehen werden. Dieser Anhang liefert eine Beschreibung des Aufbaus zum Zeitpunkt der Erstellung dieser Dissertationsschrift.

Zuerst wird eine HTML Seite vom Browser geladen, mit allen grafischen Elementen, welche immer nötig ist. Darin wird eine JavaScript-Ressource namens **hmi-hub-loader.js** nachgeladen. Diese läd nun alle weiteren Ressource nach, welche für das HMI nötig ist. **hmi-generics.js** ist eine Sammlung von Hilfsfunktionen, hauptsächlich zur Unterstützung nicht ganz aktueller Browser. **hmi-class-HMI.js** sammelt alle generischen HMI Funktionen, zum Beispiel die Interaktion mit dem Bediener beim Laden der Webseite. **hmi-class-HMIKSClient.js** liefert eine Abstrahierung des KS Protokolls. Dies kann direkt genutzt werden, wenn der Zielserver einen entsprechenden Webserver mit KS-Erweiterung (zum Beispiel ein ACPLT/OV-Server mit der Bibliothek *kshttp*) bietet. Alternativ bietet die Firma LTSoft ein Gateway um das etablierte Binärprotokoll ACPLT/KS [Alb03] nutzen zu können. Weiterhin ist die Anwendung in der Lage das Darstellungsmodell ACPLT/HMI von Stefan Schmitz [Sch10] auf den Bildschirm zu bringen, wofür weitere JS-Ressourcen nötig sind.

Die Hauptdatei für die vorgestellte Anzeigekomponente dieser Dissertation ist **hmi-class-cshmi.js**. Wenn die gewünschte Anzeige ausgewählt wurde, wird eine Funktion (`HMI.cshmi.instanciateCshmi`) zur Initialisierung der Darstellung aufgerufen. Da für komplexe Anwendungen sehr viele (viele hundert) Darstellungsprimitive benötigt werden, ist es sinnvoll die Konfiguration aller dieser Primitive in einem einzigen Netzwerkzugriff vom Server zu laden. Diese Funktion versucht daher diese Konfiguration, als `JSON`[1] kodiert, von einem speziellen Baustein unter der Adresse `/TechUnits/cshmi/turbo.asJSON` abzurufen. Ist dies erfolgreich, wird die Konfiguration im Javascript-Objekt `HMI.cshmi.ResourceList` zentral zur späteren Nutzung gespeichert.

Anschließend wird die Funktion `HMI.cshmi._interpreteElementOrEventRecursive` aufgerufen. Diese Funktion ist ein zentraler Punkt, welcher grafische Elemente und Ereignisse auf verschiedene Subfunktionen aufteilt. Nach der Erstellung der grafischen Repräsentanz zum Beispiel einer Gruppe, werden weitere grafische Kindelemente und Ereignisse geladen und interpretiert.

Elemente

Ist ein grafisches Element, wie ein Kreis, anzuzeigen so wird zum Beispiel die Funktion `HMI.cshmi._buildSvgCircle` aufgerufen. Äquivalente Funktionen existieren zu allen Grafikprimitiven wie `Path`,

[1] `http://json.org/`

`Line`, `Polyline`, `Polygon`, `Text`, `Ellipse`, `Rectangle` und `Image`. Alle diese Funktionen erhalten als Parameter neben einer eindeutigen Bezeichnung (als `String` namens `ObjectPath`) noch das Gruppenobjekt (als `DOM Element` namens `VisualParentObject`), wo das neue Element eingebettet werden wird.

Gemeinsame Parameter vieler Grafikelemente wie Strich- oder Füllfarbe werden mithilfe der Hilfsfunktion `HMI.cshmi._processBasicVariables` gesetzt. Liegt die Konfiguration im Javascript-Objekt `HMI.cshmi.ResourceList` vor, so kann die Darstellung direkt aufgebaut werden, andernfalls muss die Konfiguration über ein Netzwerkzugriff abgefragt werden. Die Funktionen erhält noch einen boolschen Parameter namens `preventNetworkRequest`. Ist dieser auf wahr gesetzt und die Konfiguration des Bausteins nicht bekannt, so wird dieser Netzwerkzugriff unterlassen. Damit kann ein aufwendiger Netzwerkzugriff bei nicht sichtbaren Elementen verhindert werden.

Die Erstellung einer Gruppe in `HMI.cshmi._buildSvgGroup` ist etwas aufwendiger, da sie mehrere Funktionen in sich vereint. Als Erstes kann die Gruppe weitere Objekte über ein Vorlagensystem einbinden (über `TemplateDefinition`) und außerdem gewissen Konfigurationsparameter setzen, welche für den ganzen Darstellungs-Zweig Gültigkeit hat.

Wird eine Instanziierung einer Vorlage benötigt, so wird dessen Konfiguration (Breite und Höhe) über einen identischen Mechanismus wie die Grafikprimitive geholt, wenn sie zu diesem Zeitpunkt nicht schon bekannt sind. Wird eine `FBReference` benötigt, so wird diese im grafischen Objekt an der Stelle `VisualObject.ResourceList.FBReference` gespeichert. Es ist jedoch möglich, dass diese Referenz über einen URL-Parameter namens `FBReference` des Browsers überschrieben wird. `FBVariableReference`n sind mehrere möglich, so dass diese in einem JavaScript-Objekt `VisualObject.ResourceList.FBVariableReference` gespeichert werden. Ähnlich werden alle `ConfigValues` im JavaScript-Objekt `VisualObject.ResourceList.ConfigValues` gespeichert. Weiterhin wird im grafischen Objekt das SVG-Attribut `overflow="visible"` gesetzt. Damit werden alle Kindelemente dargestellt, unabhängig von der Größe der Gruppe selbst.

Ist die Gruppe nicht über das Element, sondern über die Aktion `InstantiateTemplate` erstellt worden und der Parameter `preventClone` gesetzt, so wird anschließend geprüft, ob dieses Objekt schon identisch vorhanden ist. Ist dies der Fall, wird die Darstellung dieser Instanz verhindert. Anschließend wird die genaue X-Y-Position des neuen Objektes berechnet. Die Parameter `xOffset` , `yOffset` und `maxTemplatesPerDirection` erlaubt dazu eine genaue Parametrierung. Details zur Logik siehe Kapitel 3.5.3.

Soll eine Gruppe selbst „versteckbar“ sein, so wird dies im Vater-Objekt der Darstellung vermerkt, sowie dort ein Event-Handler zum Ausblenden und Anzeigen aller Kinder hinterlegt. Innerhalb dieser Logik ist es manchmal nötig die Reihenfolge der Objekte innerhalb des DOM des Browsers zu ändern, um eine vollständige Anzeige zu erlauben. Für die Funktion `previousTemplateCount` ist die Original-Reihenfolge jedoch nötig, sodass diese zusätzlich in einem Array `cshmiOriginalOrderList` gespeichert wird.

Am Ende der Funktion `HMI.cshmi._buildSvgGroup` werden noch alle Kindelemente interpretiert, welche eventuell über ein Template angefordert wurden.

Um eine Blackbox zu erstellen wird die Funktion `HMI.cshmi._buildBlackbox` genutzt. Dieser Code holt (wenn bisher noch nicht bekannt) die Inhalte der Variablen `HTMLcontent`, `sourceOfLibrary`, `jsOnload` und `jsOnglobalvarchanged` vom Modell.

Wird `HTMLcontent` genutzt, so wird der entsprechende HTML-Code in den Darstellungsbaum des Browsers eingehangen. Soll JavaScript-Code ausgeführt werden, so wird ein API-Object namens `cshmimodel` (siehe Anhang 3) erstellt, welche Kommunikation mit dem Automatisierungssystem und der Darstellung ermöglicht. `jsOnload` wird ausgeführt, wenn alle Bibliotheken über `sourceOfLibrary` vollständig geladen sind.

Ereignisse

Hat ein Element ein TimeEvent assoziiert, so wird die Haupt-Funktion `HMI.cshmi._interpreteElementOrEventRecursive` die Helper-Funktion `HMI.cshmi._interpreteTimeEvent` aufrufen. Diese sorgt dafür, dass Ereignisse für die die gleiche Zykluszeit angefordert wurde, gemeinsam ausgeführt werden. Ist diese Zeit größer als eine Sekunde so wird dafür gesorgt, dass der erste Aufruf kurz nach dem Laden der Gesamtanzeige vorgezogen wird, um schneller einen definierten Zustand zu erhalten. Die Funktion `HMI.cshmi._handleTimeEvent` erhält eine Liste von Variablen-Namen, welche in diesem Zyklus benötigt werden und holt alle diese Werte gemeinsam ab und speichert sie zentral. Anschließend werden alle Aktionen über die Hilfsfunktion `HMI.cshmi._interpreteAction` ausgeführt.

Für die ClientEvents „onload“ und „globalvarchanged“ wird die Funktion `HMI.cshmi._interpreteClientEvent()` genutzt. Für beide Ereignisarten wird das aktive Element in jeweils eine Liste geschrieben. Die erste Liste wird nach dem vollständigen Aufbau der Anzeige abgearbeitet und führt damit die gewünschten Aktionen aus (und löscht dabei die jeweilige Aktion aus der Liste). Die zweite Liste wird später beim Setzen einer globalen Variable abgearbeitet. Diese Liste bleibt dabei natürlich bestehen, da deren Aktionen im Gegensatz zur „onload“ Liste mehrfach ausgeführt werden soll.

Alle Benutzer-Ereignisse werden einheitlich über `HMI.cshmi._interpreteOperatorEvent()` verarbeitet. Die Ereignisse „Klick“, „Doppel-Klick“, „Rechtsklick“, „mouseover“ und „mouseout“ sind sehr ähnlich. Das Element, für das eines der drei Klick-Ereignisse definiert wurde, wird speziell markiert, sodass der Mauszeiger eine spezielle Form erhält, wenn er über einem solchen Element gehalten wird. Tritt später dieses Ereignis ein, so wird vom Browser Programmcode ausgeführt. Zuerst wird mithilfe der Funktion `Event.stopPropagation()` die Propagierung des Ereignisses gestoppt und damit verhindert, dass ein überlagertes Element beispielsweise zusätzlich eine Aktion zu diesem Klick ausführt. Eine Besonderheit ist das Ereignis zum Rechtsklick. Dieses liefert normalerweise ein Kontextmenü des Browsers. Daher wird dies über `Event.preventDefault()` verhindert. Bei allen wird das entsprechende `MouseEvent`-Objekt (siehe [Pix00]) gespeichert, um in einer Aktion beispielsweise Mauskoordinaten abfragen zu können. Weiterhin wird über die Funktion `HMI.displaygestureReactionMarker()` für 0,8 Sekunden ein kleines Rechteck als schnelles optisches Feedback eingeblendet. Anschließend wird über die Funktion `HMI.cshmi._interpreteAction()` die zugeordnete Aktion ausgeführt.

3 JavaScript API cshmimodel

Die Aktion Blackbox (siehe Kapitel 3.4.5) erlaubt die Ausführung von beliebigem JavaScript-Code. Zur Unterstützung des Applikationsentwicklers wurde eine API entwickelt mit der beispielsweise mit den grafischen Elementen interagiert werden kann. Weiterhin hat der Entwickler die Möglichkeit auf die Modellwelt zuzugreifen um zum Beispiel eine neue Instanz der Kopiervorlagen erstellen zu können. Als letzte Möglichkeit bietet die API eine direkte Kommunikation mit dem Automatisierungssystem um direkt Werte lesen und schreiben zu können, aber auch strukturelle Änderungen (Erstellen, Löschen...) anzustoßen.

Zur einfachen Programmierung wurde für die API eine TypeScript-Definitions-Datei[2] erstellt. Mit dieser ist sehr gute Editor-Unterstützung beim Programmieren möglich. Visual Studio Code und Visual Studio sind hier zum Zeitpunkt der Erstellung der Dissertationsschrift zu empfehlen.

Weiterhin dient diese Datei hier als Referenz des Umfangs der API.

Listing 2: cshmimodel API als TypeScript Definition

```
declare namespace cshmimodel {
    /** Common callback definition for KS Communication */
    interface IKsCallback {
        (
            /** HMIJavaScriptKSClient object */
            client: object,
            /** the plain request object */
            req: XMLHttpRequest
        ): void
    }
    interface Dictionary<T> {
        [index: string]: T | undefined;
    }

    /** html body node of the HTML content */
    let HtmlBody: HTMLBodyElement | null;
    /** SVG Element of the blackbox */
    let SvgElement: SVGElement;
    /** first HTML element of blackbox html content */
    let HtmlFirstElement: HTMLElement | null;
    let Modelpath: string;
    /** document object of the blackbox */
    let document: Document;
    /** Window object of the blackbox */
    let window: Window;
    /** API to variables below the blackbox object */
```

[2] https://github.com/acplt/rte/blob/master/addonlibs/hmi/cshmi_blackbox.d.ts

```
27      let variables: Dictionary<{
28          varName: string;
29          getValue: () => string;
30          setValue: (newValue: string) => void;
31      }>;
32      /**
33       * Creates a new template below the current blackbox
34       * @param x
35       * @param y
36       * @param rotate
37       * @param hideable
38       * @param PathOfTemplateDefinition
39       * @param FBReference
40       * @param FBVariableReference
41       * @param ConfigValues
42       */
43      function instantiateTemplate (
44          x: string,
45          y: string,
46          rotate: string,
47          hideable: string,
48          PathOfTemplateDefinition: string,
49          FBReference: string,
50          FBVariableReference: string,
51          ConfigValues: string
52      ): void;
53      /**
54       * Requests an Engineering Property
55       * @param path of object to query
56       * @param requestType = OT_DOMAIN type of KS Object to query
            ("OT_DOMAIN", "OT_VARIABLE", "OT_LINK" or "OT_ANY"). "
            OT_DOMAIN" if not supplied
57       * @param requestOutput Array of interesting objects
            properties ("OP_NAME", "OP_TYPE", "OP_COMMENT", "OP_ACCESS
            ", "OP_SEMANTIC", "OP_CREATIONTIME", "OP_CLASS" or "OT_ANY
            "). "OP_NAME" if not supplied
58       * @param cbfnc callback function for a async request
59       * @param responseFormat Mime-Type of requested response (
            probably "text/tcl", "text/ksx", "text/plain" used). "text
            /tcl" if not supplied
60       * @return "{fb_hmi1} {fb_hmi2} {fb_hmi3} {MANAGER} {fb_hmi4}
            {fb_hmi5}" or null or true (if callback used)
61       */
62      function getEP (
```

```
63          path: string,
64          requestType?: "OT_DOMAIN" | "OT_VARIABLE" | "OT_LINK" | "
                OT_ANY",
65          requestOutput?: "OP_NAME" | "OP_TYPE" | "OP_COMMENT" | "
                OP_ACCESS" | "OP_SEMANTIC" | "OP_CREATIONTIME" | "
                OP_CLASS" | ("OP_NAME" | "OP_TYPE" | "OP_COMMENT" | "
                OP_ACCESS" | "OP_SEMANTIC" | "OP_CREATIONTIME" | "
                OP_CLASS")[] | "OT_ANY",
66          cbfnc?: IKsCallback,
67          responseFormat?: "text/tcl" | "text/ksx" | "text/plain"
68      ): string | null | true;
69      /**
70       * Requests a KS Variable
71       * @param path of the variable to fetch, multiple path
             possible via an Array
72       * @param requestOutput Array of interesting objects
             properties ("OP_NAME", "OP_TYPE", "OP_VALUE", "
             OP_TIMESTAMP" or "OP_STATE"). "OP_VALUE" if not supplied
73       * @param cbfnc callback function for a async request
74       * @param responseFormat Mime-Type of requested response (
             probably "text/tcl", "text/ksx", "text/plain" used). "text
             /tcl" if not supplied
75       * @return "{{/TechUnits/HMIManager}}", response: "{/TechUnits
             /Sheet1}" or "TksS-0042::KS_ERR_BADPATH {{/Libraries/hmi/
             Manager.instance KS_ERR_BADPATH}}"
76       */
77      function getVar (
78          path: string | string[],
79          requestOutput: "OP_NAME" | "OP_TYPE" | "OP_VALUE" | "
                OP_TIMESTAMP" | "OP_STATE",
80          cbfnc?: IKsCallback,
81          responseFormat?: "text/tcl" | "text/ksx" | "text/plain"
82      ): string;
83      /**
84       * Sets a KS Variable
85       * @param path of the variable to set
86       * @param {String} value to set (StringVec are Arrays)
87       * @param {String} type variable type (for example "
             KS_VT_STRING") to set, null if no change
88       * @param cbfnc callback function for a async request
89       * @param responseFormat Mime-Type of requested response (
             probably "text/tcl", "text/ksx", "text/plain" used). "text
             /tcl" if not supplied
90       * @return true, "" or null
```

```
91      */
92     function setVar (
93         path: string,
94         value: string | string[],
95         type: "KS_VT_BOOL" | "KS_VT_INT" | "KS_VT_UINT" | "
               KS_VT_SINGLE" | "KS_VT_DOUBLE" | "KS_VT_STRING" | "
               KS_VT_TIME" | "KS_VT_TIME_SPAN" | "KS_VT_STATE" | "
               KS_VT_STRUCT" | "KS_VT_BYTE_VEC" | "KS_VT_BOOL_VEC" | "
               KS_VT_INT_VEC" | "KS_VT_UINT_VEC" | "KS_VT_SINGLE_VEC"
               | "KS_VT_DOUBLE_VEC" | "KS_VT_STRING_VEC" | "
               KS_VT_TIME_VEC" | "KS_VT_TIME_SPAN_VEC" | "
               KS_VT_TIME_SERIES" | "KS_VT_STATE_VEC" | null,
96         cbfnc?: IKsCallback,
97         responseFormat?: "text/tcl" | "text/ksx" | "text/plain"
98     ): "" | true | null
99     /**
100     * Rename a KS object
101     * @param path of the object to rename
102     * @param newname (optional with full path) of the object
103     * @param cbfnc callback function for a async request
104     * @param responseFormat Mime-Type of requested response (
           probably "text/tcl", "text/ksx", "text/plain" used). "text
           /tcl" if not supplied
105     * @return true, "" or null
106     */
107    function renameObjects (
108        oldName: string,
109        newName: string,
110        cbfnc?: IKsCallback,
111        responseFormat?: "text/tcl" | "text/ksx" | "text/plain"
112    ): "" | true | null
113    /**
114     * Create a KS object
115     * @param path of the object to create
116     * @param classname full class name of the new object
117     * @param cbfnc callback function for a async request
118     * @param responseFormat Mime-Type of requested response (
           probably "text/tcl", "text/ksx", "text/plain" used). "text
           /tcl" if not supplied
119     * @return true, "" or null
120     */
121    function createObject (
122        path: string,
123        classname: string,
```

```
124         cbfnc?: IKsCallback,
125         responseFormat?: "text/tcl" | "text/ksx" | "text/plain"
126     ): "" | true | null
127     /**
128      * Delete a KS object
129      * @param path ob the object to delete
130      * @param cbfnc callback function for a async request
131      * @param responseFormat Mime-Type of requested response (
            probably "text/tcl", "text/ksx", "text/plain" used). "text
            /tcl" if not supplied
132      * @return true, "" or null
133      */
134     function deleteObject (
135         path,
136         cbfnc?: IKsCallback,
137         responseFormat?: "text/tcl" | "text/ksx" | "text/plain"
138     ): "" | true | null
139     /**
140      * Link two KS objects
141      * @param pathA of the first object
142      * @param pathB of the second object
143      * @param portnameA  name of the port
144      * @param cbfnc callback function for a async request
145      * @param responseFormat Mime-Type of requested response (
            probably "text/tcl", "text/ksx", "text/plain" used). "text
            /tcl" if not supplied
146      * @return true, "" or null
147      */
148     function linkObjects (
149         pathA: string,
150         pathB: string,
151         portnameA: string,
152         cbfnc?: IKsCallback,
153         responseFormat?: "text/tcl" | "text/ksx" | "text/plain"
154     ): "" | true | null
155     /**
156      * Unlinks two KS objects
157      * @param pathA of the first object
158      * @param pathB of the second object
159      * @param portnameA  name of the port
160      * @param cbfnc callback function for a async request
161      * @param responseFormat Mime-Type of requested response (
            probably "text/tcl", "text/ksx", "text/plain" used). "text
            /tcl" if not supplied
```

```
162      * @return true, "" or null
163      */
164     function unlinkObjects (
165         pathA: string,
166         pathB: string,
167         portnameA: string,
168         cbfnc?: IKsCallback,
169         responseFormat?: "text/tcl" | "text/ksx" | "text/plain"
170     ): "" | true | null
171
172     /**
173      * Prints an info message on the website
174      * @param text
175      */
176     function log_info_onwebsite (
177         text: string
178     ): void;
179     /**
180      * Prints an error message on the website
181      * @param text
182      */
183     function log_error_onwebsite (
184         text: string
185     ): void;
186     /**
187      * returns the KS Response as an Array, or an empty Array
188      * if the optional argument recursionDepth is > 0,
189      */
190     function splitKsResponse (
191         response: string,
192         recursionDepth: number
193     ): any[];
194 }
```

Literaturverzeichnis

[Alb03] ALBRECHT, Harald: *On Meta-Modeling for Communication in Operational Process Control Engineering*. Düsseldorf, Lehrstuhl für Prozessleittechnik der RWTH Aachen University, Diss., 2003

[BD98] BOWLER, John ; DISTER, Brian: *Vector Markup Language (VML) Specification*. Mai 1998 http://www.w3.org/TR/1998/NOTE-VML-19980513

[BFL+14] BERJON, Robin ; FAULKNER, Steve ; LEITHEAD, Travis ; PFEIFFER, Silvia ; O'CONNOR, Edward ; NAVARA, Erika D.: HTML 5 / W3C. 2014. – W3C Recommendation. – http://www.w3.org/TR/2014/REC-html5-20141028/

[BHH+16] BERNSHAUSEN, Jens ; HALLER, Axel ; HOLM, Thomas ; HOERNICKE, Mario ; OBST, Michael ; LADIGES, Jan: Namur Modul Type Package – Definition. In: *atp edition - Automatisierungstechnische Praxis* 1 (2016), S. 72–81

[Dam96] DAMMERT, Jürgen: *Plattformübergreifende Konstruktion graphischer Benutzeroberflächen .* Verlag Dr. Kovac, 1996. – ISBN 3860644777

[DDFU11] DOHERR, F. ; DRUMM, O. ; FRANZE, V. ; URBAS, L.: Bedienbilder auf Knopfdruck. In: *Automatisierungstechnische Praxis atp* 53 (2011), Nr. 11, S. 30–39. – ISSN 0178–2320

[DIN14] ; DIN Deutsches Institut für Normung e. V. (Veranst.): *DIN SPEC 40912: Kernmodelle - Beschreibung und Beispiele*. 2014

[DU11] DOHERR, F. ; URBAS, L.: autoHMI: a model driven software engineering approach for HMIs in process industries. In: *2011 IEEE International Conference on Computer Science and Automation Engineering*. Piscataway, NJ : IEEE, 07 2011. – ISBN 978–1–4244–8727–1, S. 627 – 631

[ecm99] ; ECMA (European Association for Standardizing Information and Communication Systems) (Veranst.): *ECMA-262: ECMAScript Language Specification*. `http://www.ecma-international.org/publications/standards/Ecma-327.htm`. Version: Third, Dezember 1999

[ERD11] EPPLE, Ulrich ; REMMEL, Markus ; DRUMM, Oliver: Modellbasiertes Format für RI-Informationen. In: *Automatisierungstechnische Praxis (atp EDITION), 53. Jahrgang, 1-2/2011* (2011), S. 62–71

[FD98] FERRAIOLO, Jon ; DISTER, Brian: *Precision Graphics Markup Language (PGML) Specification*. April 1998 http://www.w3.org/TR/1998/NOTE-PGML-19980410.html

[Fer01] FERRAIOLO, Jon: *Scalable Vector Graphics (SVG) 1.0 Specification*. September 2001 http://www.w3.org/TR/2001/REC-SVG-20010904

[Fin13] FINK, Eugen: *Erweiterung der Visualisierungsinfrastruktur ACPLT/csHMI zur Nutzung generischer JavaScript-Fremdbibliotheken*, Bachelorarbeit, 2013

[FJF03] FUJISAWA, Jun ; JACKSON, Dean ; FERRAIOLO, Jon: *Scalable Vector Graphics (SVG) 1.1 Specification*. Januar 2003 http://www.w3.org/TR/2003/REC-SVG11-20030114/

[FR14a] FIELDING, R. ; RESCHKE, J.: *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230 (Proposed Standard). `http://www.ietf.org/rfc/rfc7230.txt`. Version: Juni 2014 (Request for Comments)

[FR14b] FIELDING, R. ; RESCHKE, J.: *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231 (Proposed Standard). `http://www.ietf.org/rfc/rfc7231.txt`. Version: Juni 2014 (Request for Comments)

[HB11] HENNIG, Stefan ; BRAUNE, Annerose: Sustainable visualization solutions in industrial automation with Movisa — A case study. In: *Proceedings of INDIN 2011*. Caparica, Lisbon, Portugal, 08 2011. – ISBN 978–1–4577–0433–8, S. 634 – 639

[Hen12] HENNIG, Stefan: *Design of sustainable solutions for process visualization in industrial automation with model-driven software development*. Dresden, Diss., 2012

[HLW+16] HOLM, Thomas ; LADIGES, Jan ; WASSILEW, Sachari ; ALTMANN, Paul ; FAY, Alexander ; URBAS, Leon ; HEMPEN, Ulrich: DIMA im realen Einsatz - Von der Idee zum Prototypen. In: *Automation 2016: der 17. Branchentreff der Mess- und Automatisierungstechnik / VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik*. Düsseldorf : VDI-Verlag, 2016 (VDI-Berichte ; 2284). – ISBN 978–3–18–092284–0, S. 71–83. – CD-ROM

[Hon14] HONEYWELL INTERNATIONAL SÀRL: *Experion LX, HMIWeb Display Building Guide*. Release 110. Honeywell International Sàrl, Z.A. La Pièce 16, 1180 Rolle (VD), Schweiz, Feb 2014. `https://www.honeywellprocess.com/library/support/Public/Documents/HMIWeb_Display_Building_Guide_EXDOC-XX54-en-110.pdf`

[IEC03] Norm März 2003. *IEC 61131-3, 2nd edition. Programmable controllers – Part 3: Programming languages*

[IEC10a] Norm 2010. *IEC 62541: OPC Unified Architecture*

[IEC10b] Norm 2010. *IEC 62714: Engineering data exchange format for use in industrial automation systems engineering - Automation Markup Language*

[JE12] JEROMIN, Holger ; EPPLE, Ulrich: Anwendungs- und herstellerneutrales Modell zur Darstellung und Interaktion mit leittechnischen Funktionen. In: *Automation 2012 : der 13. Branchentreff der Mess- und Automatisierungstechnik / VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik*. Düsseldorf : VDI-Verlag, 2012 (VDI-Berichte ; 2171). – ISBN 978–3–18–092171–6, S. 219–222. – CD-ROM

[JE13] JEROMIN, Holger ; EPPLE, Ulrich: Modellbasiertes und technologieneutrales HMI für eingebettete Komponenten. In: GIESE, Holger (Hrsg.) ; HUHN, Michaela (Hrsg.) ; PHILLIPS, Jan (Hrsg.) ; SCHÄTZ, Bernhard (Hrsg.): *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IX, Schloss Dagstuhl, Germany, April 24-26, 2013, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, fortiss GmbH, München, 2013, 80–89

[Jer08] JEROMIN, Holger: *Browserbasierte Visualisierung aktiver Flusswege in komplexen Abfüllstationen*, RWTH Aachen, Diplomarbeit, 12 2008

[Kir07] KIRMAS, M: Anwenderbericht zur Nutzung von typischen Funktionsbausteinen (Typicals) bei der Erstellung von leittechnischer Anwendersoftware. In: *Automation 2007 VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik*. Düsseldorf : VDI-Verlag, 2007 (VDI-Berichte ; 2284), S. 783–790. – CD-ROM

[Koc06] KOCH, Peter-Paul: *ppk on JavaScript, 1/e*. New Riders, 2006. – ISBN 0321423305

[Kos14] KOSTIAINEN, Anssi: *Vibration API*. Juni 2014 http://www.w3.org/TR/2014/WD-vibration-20140619/

[LVM+05] In: LIMBOURG, Quentin ; VANDERDONCKT, Jean ; MICHOTTE, Benjamin ; BOUILLON, Laurent ; LÓPEZ-JAQUERO, Víctor: *USIXML: A Language Supporting Multi-path Development of User Interfaces*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005. – ISBN 978–3–540–31961–0, 200–220

[ME07] MÜLLER, Jochen ; ENSTE, Udo: *Datenkommunikation in der Prozessindustrie*. Oldenbourg Industrieverla, 2007. – ISBN 3835631160

[Mer18] MERSCH, Tina: *Regelbasierte Modelltransformation in prozessleittechnischen Laufzeitumgebungen*. Düsseldorf, Lehrstuhl für Prozessleittechnik der RWTH Aachen University, Diss., 2018

[Mey00] MEYER, Dirk: Dezentrale Intelligenz durch Metamodell-basierte Objektverwaltung. In: MEHLHORN, Kurt (Hrsg.) ; SNELTING, Gregor (Hrsg.): *Informatik 2000*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2000. – ISBN 978–3–642–58322–3, S. 304–317

[Mil68] MILLER, Robert B.: Response Time in Man-computer Conversational Transactions. In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. New York, NY, USA : ACM, 1968 (AFIPS '68 (Fall, part I)), 267–277

[MPV11] MEIXNER, Gerrit ; PATERNÒ, Fabio ; VANDERDONCKT, Jean: Past, Present, and Future of Model-Based User Interface Development. In: *i-com* 10 (2011), Nr. 3, 2–11. http://dx.doi.org/10.1524/icom.2011.0026. – DOI 10.1524/icom.2011.0026

[OHU+15] OBST, Michael ; HOLM, Thomas ; URBAS, Leon ; FAY, Alexander ; KREFT, Sven ; HEMPEN, Ulrich ; ALBERS, Thomas: Beschreibung von Prozessmodulen - Ein weiterer Schritt zur Umsetzung der NE 148. In: *atp edition - Automatisierungstechnische Praxis* 1 (2015), S. 48–59

[Pix00] PIXLEY, Tom: Document Object Model (DOM) Level 2 Events Specification / W3C. 2000. – W3C Recommendation. – http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113

[Pri06] PRIBEANU, Costin: Task Modeling for User Interface Design–A Layered Approach. In: *International Journal of Information Technology* 3 (2006), Nr. 2, S. 86–90

[Roc12] ROCKS, Yannick: *Erstellung einer modellbasierten Engineering Software für ACPLT/csHMI*, Bachelorarbeit, 2012

[Sch10] SCHMITZ, Stefan: *Grafik- und Interaktionsmodell für die Vereinheitlichung grafischer Benutzungsschnittstellen der Prozessleittechnik*. Düsseldorf, Lehrstuhl für Prozessleittechnik der RWTH Aachen University, Diss., 2010

[Sch12] SCHNELLER, Anne: Parametrieren statt programmieren. In: *VDI nachrichten 10.02.2012*, 2012

[SE07] SCHMITZ, Stefan ; EPPLE, Ulrich: Automatisierte Projektierung von HMI-Oberflächen. In: *GMA Kongress 2007 – Automation im gesamten Lebenszyklus*. Düsseldorf : VDI-Verlag, Juni 2007 (VDI-Berichte, No. 1980, ISBN: 978-9-18-091980-5), S. 127–138

[SE12] SCHÜLLER, Andreas ; EPPLE, Ulrich: PandIX – Exchanging P&I diagram model data. In: *ETFA 2012: IEEE 17th International Conference on Emerging Technologies and Factory Automation ; September 17-21, 2012, Krakow, Poland*. Piscataway, NJ : IEEE, 2012. – ISBN 978–1–4673–4737–2. – 1 CD-ROM

[SE13] SCHÜLLER, Andreas ; EPPLE, Ulrich: Ein Modellserver zur Nutzung von R&I-Fließbild-Informationen. In: *AUTOMATION 2013: 14. Branchentreff der Mess- und Automatisierungstechnik*. Düsseldorf : VDI-Verlag GmbH, Juni 2013. – ISBN 978–3–18–092209–6, S. 223–226

[Sie13] SIEMENS AG: *SIMATIC HMI, WinCC: Scripting (VBS, ANSI-C, VBA) - Systemhandbuch*. WinCC V7.2. Siemens AG, Industry Sector, Postfach 48 48, 90026 Nürnberg, Deutschland, 2013. https://cache.industry.siemens.com/dl/files/640/73453640/att_67199/v1/WinCCInformationSystemScripting_de-DE.pdf

[UHH+11] URBAS, L. ; HENNIG, S ; HAGER, H ; DOHERR, F ; BRAUNE, A: Towards context adaptive HMIs in process industries. In: *2011 9th IEEE International Conference on Industrial Informatics*. Caparica, Lisbon, Portugal : IEEE, 07 2011. – ISBN 978–1–4577–0435–2

[VDI02] ; Verband Deutscher Ingenieure (VDI) (Veranst.): *VDIVDE Richtlinie 3850, Nutzergerechte Gestaltung von Bediensystemen für Maschinen*. 2002

[VDI13] ; Verband Deutscher Ingenieure (VDI) (Veranst.): *VDIVDE Richtlinie 3699, Prozessführung mit Bildschirmen*. 2013

[WE17] WAGNER, Constantin A. ; EPPLE, Ulrich: Integration von Serviceschnittstellen in Funktionsbausteinarchitekturen. In: *Automation 2017: der 18. Branchentreff der Mess- und Automatisierungstechnik / VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik*. Düsseldorf : VDI-Verlag, 2017 (VDI-Berichte ; 2284). – ISBN 978–3–18–092284–0. – CD-ROM

[YGE13a] YU, L. ; GRÜNER, S. ; EPPLE, U.: An engineerable procedure description method for industrial automation. In: *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, 2013. – ISSN 1946–0740, S. 1–8

[YGE13b] YU, Liyong ; GRÜNER, Sten ; EPPLE, Ulrich: An Engineerable Procedure Description Method for Industrial Automation. In: *ETFA 2013: IEEE 18th International Conference on Emerging Technologies and Factory Automation*. Piscataway, NJ : IEEE, 2013. – ISBN 978–1–4799–0864–6

[YQE10] YU, Liyong ; QUIRÓS, Gustavo ; EPPLE, Ulrich: Service-Oriented Process Control for Complex Multifunctional Plants: Concept and Case Study. In: *ETFA 2010: 15th IEEE International Conference on Emerging Technologies and Factory Automation*. Bilbao : IEEE, September 2010. – ISBN 978–1–4244–6849–2

[Yu16] YU, Liyong: *A reference model for the integration of agent orientation in the operative environment of automation systems*. Düsseldorf, Lehrstuhl für Prozessleittechnik der RWTH Aachen University, Diss., 2016

Die Reihen der Fortschritt-Berichte VDI:

1 Konstruktionstechnik/Maschinenelemente
2 Fertigungstechnik
3 Verfahrenstechnik
4 Bauingenieurwesen
5 Grund- und Werkstoffe/Kunststoffe
6 Energietechnik
7 Strömungstechnik
8 Mess-, Steuerungs- und Regelungstechnik
9 Elektronik/Mikro- und Nanotechnik
10 Informatik/Kommunikation
11 Schwingungstechnik
12 Verkehrstechnik/Fahrzeugtechnik
13 Fördertechnik/Logistik
14 Landtechnik/Lebensmitteltechnik
15 Umwelttechnik
16 Technik und Wirtschaft
17 Biotechnik/Medizintechnik
18 Mechanik/Bruchmechanik
19 Wärmetechnik/Kältetechnik
20 Rechnerunterstützte Verfahren (CAD, CAM, CAE CAQ, CIM ...)
21 Elektrotechnik
22 Mensch-Maschine-Systeme
23 Technische Gebäudeausrüstung

ISBN 978-3-18-526808-3