HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Bachelorarbeit
Abteilung Informatik
Hochschule für Technik Rapperswil

Modellgetriebene Visualisierung von Echtzeitsystemen im Browser

---

Autoren:	Lukas Wegmann, Dominique Wirz
Betreuer:	Prof. Dr. Olaf Zimmermann
Projektpartner:	actifsource GmbH, Baden-Dättwil
	B&R Automation GmbH, Frauenfeld
Experte:	Dr. Michael Wahler, ABB
Gegenleser:	Prof. Dr. Josef M. Joller

14. Juni 2013

Abstract

Modellgetriebene Entwicklungsumgebungen ermöglichen es, Software für Echtzeitsteuerungen mithilfe von domänenspezifischen Modellen und Diagrammen zu spezifizieren. Die Modelle können anschliessend für die Generierung des Quellcodes für unterschiedliche Programmiersprachen und Zielplattformen verwendet werden. Nach dem Ausliefern stehen jedoch oft keine Möglichkeiten zur Verfügung, dass Laufzeitverhalten anhand der entworfenen Domänendiagramme zu analysieren. Deshalb wird eine Lösung benötigt, die diese Diagramme anhand der Zustandsdaten der Steuerung im Webbrowser animieren kann.

Die entwickelte Lösung enthält unterschiedliche Komponenten, um ein derartiges Diagnosetool in Steuerungen zu integrieren, die mit der actifsource Entwicklungsumgebung modelliert wurden. Die Grundlage stellt ein zustandsloses Kommunikationsmodell basierend auf HTTP dar. Dies ermöglicht den kontinuierlichen Datenaustausch zwischen der Browserapplikation und einem, auf der Steuerung integrierten, Webserver. Request Multiplexing ersetzt dabei nebenläufige Anfragen, auch wenn mehrere Diagramme gleichzeitig zu animieren sind. Die Browserapplikation ist komplett eigenständig ausführbar und benötigt kein dynamisches Seitenrendering auf dem Server. Der auf der Steuerung ausgeführte Webservice kann durch eine Abstraktionsschicht auf unterschiedlichen Plattformen integriert werden. Da die benötigte Datenstruktur aus dem zu animierenden Modell als C-Quellcode generiert wird, sind keine Zugriffe auf das Dateisystem notwendig und es kann auf fehleranfällige dynamische Allokationen von Speicher auf dem Heap verzichtet werden. Dadurch sind die Anforderungen an die Zielplattform sehr gering. Performancetests zeigten zudem, dass die Lösung Systeme mit mehreren tausend Diagrammen und über 10'000 Zustandswechsel pro Sekunde animieren kann.

Eigenständigkeitserklärung

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben,
- dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben.

Ort, Datum:

Rapperswil, 13.06.13

Name, Unterschrift:

Lukas Wegmann

Dominique Wirz

Inhaltsverzeichnis

Typografische Konventionen

Begriffe, die *kursiv* geschrieben sind, werden im Glossar aufgeführt und kurz erläutert. Dazu gehören auch *Abkürzungen (Abk.)*, die beim ersten Gebrauch ausgeschrieben werden. Danach wird stattdessen jeweils die entsprechende *Abk.* verwendet. Des Weiteren wird `Code` und im weiteren Sinn verwandte Fragmente, wie zum Beispiel Dateinamen, in der `Monospace`-Schriftart dargestellt.

1. Management Summary

1.1. Ausgangslage

Modellgetriebene Entwicklungsumgebungen ermöglichen es Anwendungen mithilfe von domänenspezifischen Modellen und Diagrammen zu entwickeln. Eine derartige Entwicklungsumgebung ist actifsource, welche unter anderem zur Implementation der Steuerungslogik auf Echtzeitsystemen eingesetzt wird (Abbildung 1.1). Dabei werden oft Modelle eingesetzt, die auf dem Prinzip der deterministischen Zustandsautomaten basieren.

Nachdem die Steuerungslogik auf ein Echtzeitsystem ausgeliefert ist, stehen momentan nur sehr einfache Methoden zur Analyse des Laufzeitverhaltens zur Verfügung. Weshalb Fehler im logischen Modell anhand des generierten Code nachvollzogen werden müssen und dadurch schwer zu entdecken sind.

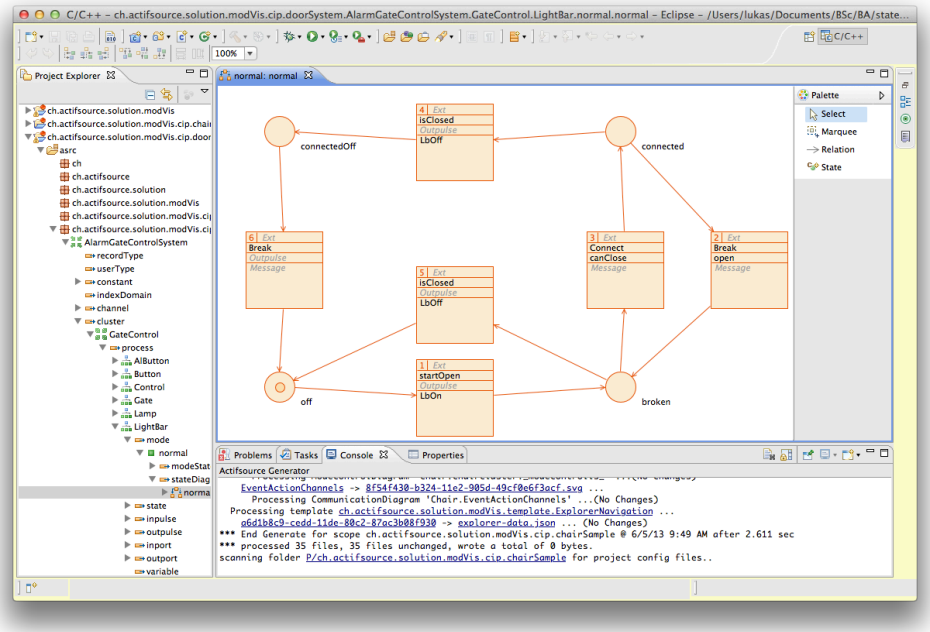


Abbildung 1.1.: Entwicklung einer Torsteuerung in actifsource

Die Diagnose des laufenden Systems kann vereinfacht werden, indem der Benutzer den Systemzustand anhand der entworfenen Domänendiagramme beobachten kann. Dazu sollen diese Diagramme mit den aktuellen Zustandsdaten angereichert und fortlaufend animiert werden. Zum Beispiel kann bei einem Diagramm eines Zustandsautomaten jeweils der letzte Zustandsübergang sowie der neu eingetretene Zustand farblich hervorgehoben werden.

1.2. Zielsetzung

Im Rahmen dieser Bachelorarbeit soll eine Lösung zur Animation von Zustandsmaschinen auf Echtzeitsystemen im Browser gefunden werden. Die fraglichen Zustandsmaschinen werden nach der *Communicating Interacting Processes (CIP)* Methode [Fie99] mithilfe der Entwicklungsumgebung actifsource entworfen. Die *CIP* Methode dient jedoch nur als Referenzmodell; die Lösung soll möglichst universell für unterschiedliche Darstellungen von Prozessabläufen verwendet werden können.

Grundlage für die Lösung soll ein geeignetes Kommunikationsmodell für den Austausch der Zustandsdaten zwischen Webapplikation und Echtzeitsystem bilden. Darauf aufbauend werden die im Browser ausgeführte Webapplikation und der auf der Hardwaresteuerung eingebettete Webservice konzeptioniert und umgesetzt. Zudem müssen Schnittstellen zum Echtzeitsystem sowie zur actifsource Entwicklungsumgebung definiert werden.

Da Echtzeitsysteme oft zur Steuerung von Industriemaschinen eingesetzt werden, bei denen Unterbrüche zu schwerwiegenden Konsequenzen führen können, werden an die gesamte Software hohe Anforderungen bezüglich Stabilität und Zuverlässigkeit gestellt. Zudem sollen die erarbeiteten Komponenten leicht auf weitere Hardwaretypen portierbar sein.

1.3. Resultat

Das erarbeitete Kommunikationsmodell basiert ausschliesslich auf weitverbreiteten Standards und geht sparsam mit den ohnehin schon limitierten Ressourcen von Echtzeitsystemen um. Die Bündelung der Anfragen an den Webservice reduziert deren Anzahl auf eine pro Benutzer, auch wenn gleichzeitig mehrere Diagramme geöffnet sind.

Des Weiteren beinhaltet die erarbeitete Lösung eine Sammlung von Komponenten, welche es ermöglichen die Visualisierung in beliebige Modelle einzubinden und sie auf unterschiedlichen Hardwaretypen zu betreiben.

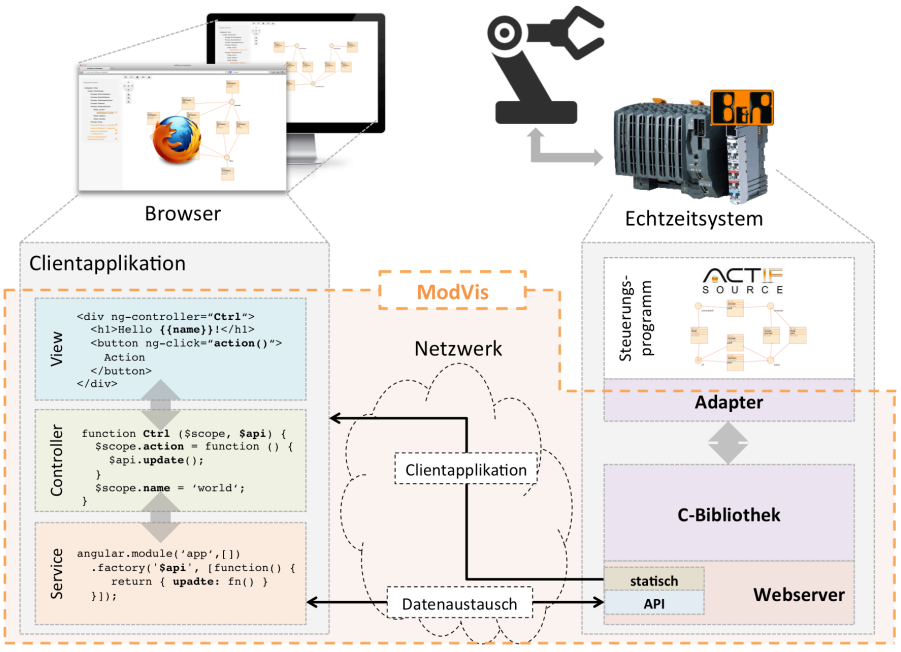


Abbildung 1.2.: Architekturüberblick der umgesetzten Lösung ModVis

Eine Erweiterung der actifsource Umgebung generiert für jedes Modell C-Quellcode, welcher die gesamte benötigte Datenstruktur zur Verwaltung des Systemzustands auf der Steuerung definiert. Dadurch kann vollständig auf fehleranfällige dynamische Speicherallokationen verzichtet werden.

Die entwickelte C-Bibliothek für die Verwaltung der Zustandsdaten kann über eine einfache Schnittstelle an bestehende Webserver angebunden werden. Die Interprozesskommunikation zu den Steuerungsprozessen wird zudem in einem plattformspezifischen Adapter gekapselt und ist entsprechend der jeweiligen Zielplattform austauschbar.

Die Clientapplikation wurde als eigenständig im Browser ausführbare Webanwendung entwickelt, die kein dynamisches Seitenrendering auf dem Webserver voraussetzt. Dadurch werden die Anforderungen an den Webserver stark reduziert und die gesamte Bedien- und Darstellungslogik wird auf die Clientseite ausgelagert.

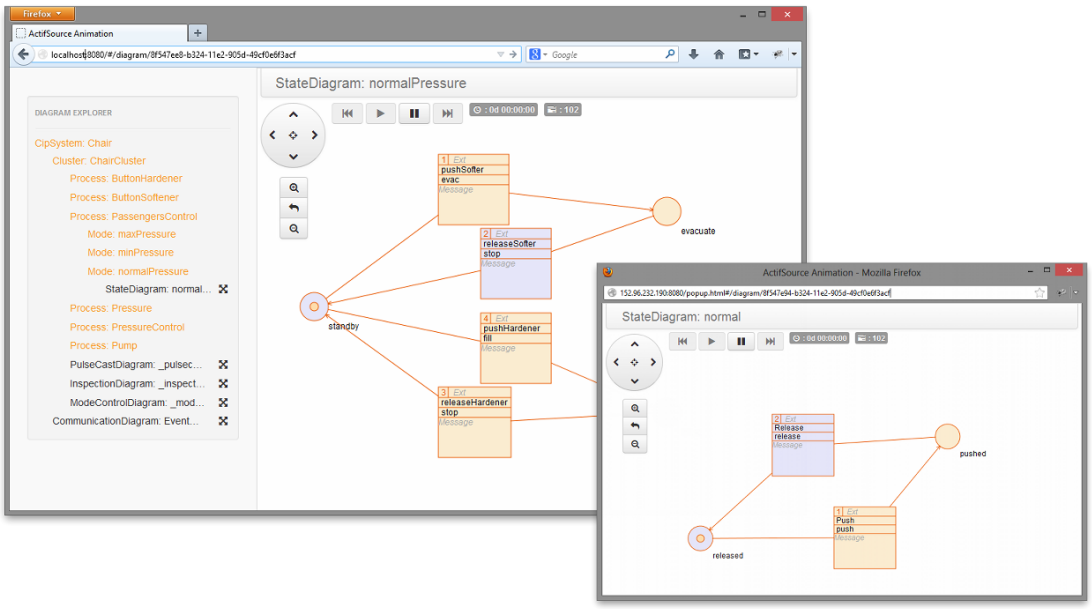


Abbildung 1.3.: Benutzeroberfläche im Browser mit zwei geöffneten Diagrammen

Zur Demonstration der Funktionsweise der Lösung wurde schliesslich mit dem *CIP* Tool ein Modell zur Steuerung eines automatischen Tors entworfen und auf einer von der Firma B&R zur Verfügung gestellten *Speicherprogrammierbare Steuerung (SPS)* ausgeliefert. Des Weiteren wurde diese *SPS* verwendet, um mithilfe von Performancetests die Kapazitätsgrenzen der erarbeiteten Komponenten auf einer möglichen Zielplattform auszuloten. Die Resultate zeigen, dass selbst Modelle mit mehreren tausend Diagrammen und mit über 10'000 Zustandswechsel pro Sekunde noch animiert werden können.

1.4. Ausblick

Obwohl die praktische Anwendbarkeit des Produktes aufgezeigt wird, fehlen für den produktiven Einsatz noch einige Aspekte. In erster Linie sollte die Integration in die actifsource Umgebung noch erweitert werden, damit die Visualisierung einfacher in Projekte eingebunden werden kann.

Auch verschiedene zusätzliche Funktionalitäten wären noch denkbar. Dazu gehören beispielsweise das clientseitige Aufzeichnen der Zustandswechsel über eine längere Zeitdauer

oder die Unterstützung zur Darstellung von weiteren Daten wie etwa Werte von Variablen.

In einem nächsten Schritt könnte die Clientapplikation auch direkt als Plugin für die actifsource Entwicklungsumgebung umgesetzt werden, um so dem Entwickler das Ausführen und Überprüfen eines Modells schon während dessen Entwicklung zu ermöglichen.

2. Analyse der Anforderungen

2.1. Allgemeine Beschreibung

Das in dieser Arbeit zu erarbeitende Produkt "ModVis" ist nicht als eigenständig ausführbare Software zu verstehen, sondern als eine Sammlung von Werkzeugen zur Erweiterung von mit actifsource entwickelten Steuerungen. Die nachfolgenden Anforderungen richten sich dementsprechend an die Gesamtmenge dieser Komponenten, ohne genauer auf die Einzelteile einzugehen.

2.1.1. Produktperspektive

Mit dem *CIP* Tool vertreibt actifsource eine Umgebung, mit der Echtzeitsysteme modelgetrieben entwickelt werden können. Die Ausführungslogik wird dabei in Zustandsmaschinen abgebildet [Fie99]. ModVis soll es dem Benutzer ermöglichen, diese Zustandsdiagramme am laufenden System analysieren und verstehen zu können. Dazu sollte lediglich ein Browser und Netzwerkzugriff auf das entsprechende System (z.B. eine Industriemaschine) erforderlich sein.

Da CIP nur eine mögliche Methode zur Modellierung von Echtzeitsystemen darstellt und die Entwicklungsumgebung stetig weiterentwickelt wird, muss ModVis universell für beliebige Modelle einsetzbar sein. CIP dient dabei lediglich als Referenz für die Implementation und wird für die Aufnahme der Anforderungen als Grundlage genommen.

Zum einen muss die Navigation durch die Modelle des Systems möglich sein. Die Benutzeroberfläche soll sich dabei an der dem Benutzer bekannten Interface des *CIP* Tools orientieren. Insbesondere sollen die Modelldiagramme möglichst identisch aussehen.

Elemente in den Diagrammen müssen zudem animierbar sein. Die Animationen sind innerhalb der Diagramme spezifiziert und werden durch Ereignisse auf dem Zielsystem gestartet. So sollen zum Beispiel Zustandsübergänge sichtbar werden und aktive Zustände gekennzeichnet werden.

In groben Teilen besteht ModVis aus einem Webserver, der auf dem Echtzeitsystem ausgeführt wird und die Zustandsänderungen filtert und aufbereitet, und einer Webapplikation im Browser, die die *CIP* Modelle darstellt und animiert.

2.1.2. Produktfunktionen

Folgende Ziele soll die Softwarelösung erfüllen:

- Die Applikation wird in Form einer Webapplikation zur Verfügung gestellt, der Benutzer kann somit ohne Installation zusätzlicher Software damit arbeiten.
- Unterschiedliche Hierarchiestufen aus dem *CIP* Tool können dargestellt werden:

  System Mehrere nebenläufige *Cluster*; wird als Kommunikationsdiagramm dargestellt (Abbildung 2.1)

  Cluster Über *Pulses* kommunizierende Prozesse; wird als Pulse Cast Diagram dargestellt

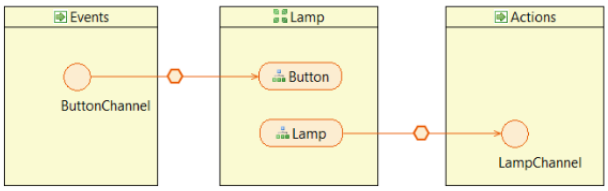


Abbildung 2.1.: Ein System mit einem “Lamp” Cluster und zwei *Channels*

Mode Unterschiedliche Versionen von Prozessen, die je nach Systemzustand aktiv sind; wird als Mode Control Diagram dargestellt

Process Zustände und Zustandsübergänge; wird als Extended Finite State Machine [Fie99] dargestellt (Abbildung 2.2)

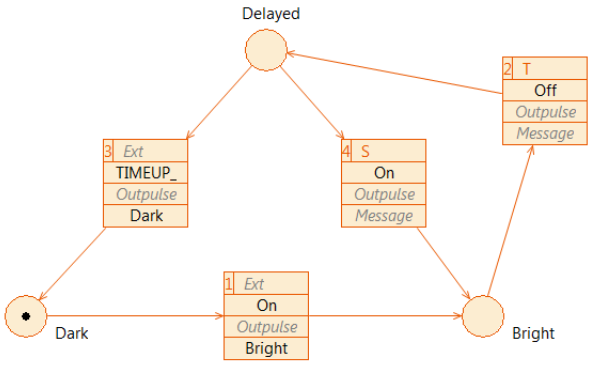


Abbildung 2.2.: Darstellung eines Prozesses im CIP Tool

Process Array Mehrere Instanzen eines Prozesses; wird als Liste von Prozessen dargestellt

- Die Hierarchie und Darstellung der Diagramme sollen ohne Anpassungen der Software änderbar sein.
- Die Navigation durch die einzelnen Diagramme in der Systemhierarchie soll sich an dem *CIP* Tool orientieren.
- Die Zustandsübergänge einzelner Zustandsmaschinen (Modes) sind animiert.
- Der Server kann auf unterschiedliche Plattformen portiert werden:
  - Embedded Systems
  - Microsoft Windows

2.1.3. Einschränkungen

Durch die Vorgabe eine Webapplikation zu entwickeln, kann es vorkommen, dass gewisse Funktionen von älteren Browser nicht oder nicht vollständig unterstützt werden. Zwingend erforderlich ist die Unterstützung der aktuellen Version von Firefox (19.0). Bei der Technologieevaluation sollen jedoch alle gängigen Browser (Internet Explorer 10.0+, Firefox 19.0+, Chrome 25.0+, Safari 6.0) berücksichtigt werden, damit eine zukünftige Erweiterung der zu unterstützenden Browser möglich bleibt.

Auf die Unterstützung von Mobile Browsern wird vorerst verzichtet.

Falls der Server auf einem Embedded System läuft, steht kein Dateisystem zur persistenten Speicherung zur Verfügung. Es können somit keine Zustände oder zur Laufzeit getätigte Konfigurationen persistent gespeichert werden. Des Weiteren ist durch den begrenzten Speicherplatz auf den Embedded Systemen die Verwendung von überflüssigen Third Party Libraries zu vermeiden.

2.1.4. Annahmen und Abhängigkeiten

Damit die Modelldiagramme in ModVis identisch wie im *CIP* Tool dargestellt werden, werden diese automatisch aus dem *CIP* Tool als *Scalable Vector Graphics (SVG)* exportiert. Diese sollen dann von der Webanwendung dargestellt und animiert werden.

Listing 2.1 zeigt Teile eines aus dem CIP Tool exportierten SVGs eines Prozesses. Ersichtlich ist die Beschreibung eines Zustandes und Teile eines Zustandüberganges. Besonders hervorzuheben ist die eindeutige Identifizierung der Elemente durch die in den `id` Attributen enthaltenen *Universally Unique Identifiers (UUIDs)*. Eine systemweit eindeutige Adressierung der verschiedenen Elemente ist für die Navigation und Animation zwingend notwendig.

```
1  <?xml version="1.0" standalone="no"?>
2  <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
3  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
4
5  <svg:svg xmlns:svg="http://www.w3.org/2000/svg" version="1.1" xmlns:xlink="http://www.w3.
       org/1999/xlink">
6    <svg:g transform="translate(100,100)"><!-- 4 Node -->
7      <!-- ModeState 'blocked' [4957a2ac-5240-11e1-a84d-5303f23f60ab] -->
8      <svg:g id="guid:4957a2ac524011e1a84d5303f23f60ab" transform="translate(750 105)">
9        <svg:ellipse cx="20" cy="20" rx="20" ry="20" stroke="rgb(234,88,00)" fill="rgb
             (252,237,209)"/>
10       <svg:text font-family="Arial" font-size="12" x="40" y="40" fill="black">blocked</svg
             :text>
11     </svg:g>
12     <!-- ... -->
13   </svg:g>
14   <svg:g transform="translate(100,100)"><!-- 6 Edge -->
15     <!-- ModeState 'blocked' [4957a2ac-5240-11e1-a84d-5303f23f60ab] -&gt; ModeState '
           delayed' [4a101d47-5240-11e1-a84d-5303f23f60ab] -->
16     <svg:polyline fill="none" points="751,121 505,66 330,137" stroke="rgb(234,88,00)"/>
17     <svg:g><!-- 1 Node -->
18       <!-- Transition '4 release' [c7f8c75f-5240-11e1-a84d-5303f23f60ab] -->
19       <svg:g id="guid:c7f8c75f524011e1a84d5303f23f60ab" transform="translate(462 25)">
20         <svg:rect x="0" y="0" width="86" height="83" stroke="rgb(234,88,00)" fill="rgb
               (252,237,209)"/>
21         <!-- ... -->
22         <svg:text font-family="Arial" font-size="12" x="0" y="45" dy="12" fill="rgb
               (169,169,169)" font-style="italic">Message</svg:text>
23         <!-- ErrorIcon -->
24       </svg:g>
25     </svg:g>
26     <!-- ... -->
27   </svg:g>
28 </svg:svg>
29 <!-- Actifsource ID=[d472a564-405f-11e2-b54c-bf6e415d1beb,65d8eda9-895b-11e1-a5da-5188
       f28b763a,vaNW5csJM4ZSmHbOKkFvpm5gF2E=] -->
```

Listing 2.1: Aus CIP Tool exportiertes SVG einer State Machine

Des Weitern soll geprüft werden, ob der ModVis Server in den von einigen Embedded Systems enthaltenen Webservern integriert werden kann. Dazu wird von der Firma B&R Automation eine *SPS* zur Verfügung gestellt.

2.2. Funktionale Anforderungen

2.2.1. Aktoren und Stakeholder

Techniker

- Möchte sich ein Bild über die Funktionsweise der Maschinensteuerung machen können
- Will auf eine Dokumentation der effektiv installierten Softwareversion zugreifen können
- Beabsichtigt im Fehlerfall die Ursache des Fehlverhaltens finden zu können

2.2.2. Spezifikation der Use Cases

Die in der Abbildung 2.3 aufgelisteten Use Cases beschreiben neben der Kernfunktionalität (UC01 und UC02) auch weitere Funktionalitäten, die nicht zwingend im Rahmen dieser Bachelorarbeit umgesetzt werden müssen. Vielmehr sollen sie ein Bild über die weiteren Möglichkeiten von ModVis geben.

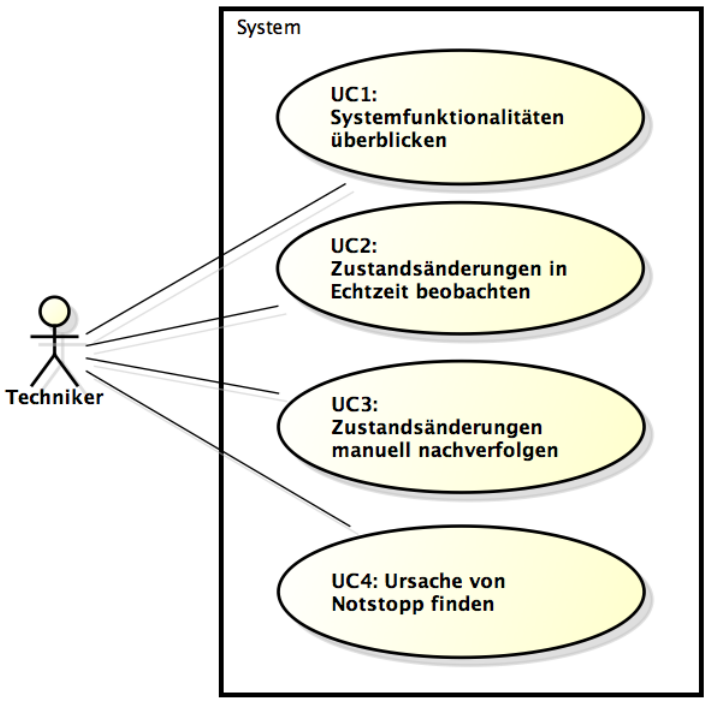


Abbildung 2.3.: Use Case Diagram

Zur Veranschaulichung der Use Cases beziehen sich die einzelnen Beschreibungen auf eine einfache, von einem Echtzeitsystem gesteuerten, Lichtanlage. Teile der Lichtsteuerung sind in den Abbildungen 2.1 und 2.2 beschrieben. Dieses Beispiel ist selbstverständlich stark vereinfacht im Gegensatz zu einer industriell eingesetzten Maschinensteuerung.

UC01: Systemfunktionalitäten überblicken

Aktor

Techniker

Beschreibung

Der mit der Wartung eines Lichtsystems beauftragte Techniker möchte sich über dieses einen Überblick verschaffen. Die beiliegenden Dokumentationen sind leider oft nicht aktualisiert und beschreiben nicht mehr die aktuell installierte Softwareversion. Daher würde der Techniker den Zugriff auf die Diagramme bevorzugen, mit deren Hilfe die Steuerung entwickelt wurde.

Dazu öffnet er den Browser auf seinem Laptop und verbindet sich mit der an der Steuerung angeschriebenen Host Adresse (Domainname oder IP). Er gelangt nun auf das Systemdiagramm. Von diesem aus kann er anschliessend durch Anklicken der verschiedenen Elemente (Cluster, Process etc.) durch die Diagramme navigieren.

UC02: Zustandsänderungen in Echtzeit beobachten

Aktor

Techniker

Beschreibung

Im Laufe der Wartungsarbeiten stösst der Techniker auf ein Verhalten, das er so nicht erwartet hat: Nach dem Betätigen des Aus-Schalters wird ein Licht nicht ummittelbar dunkel. Er sucht in den Diagrammen nach dem Prozess, der auf den Channel der entsprechenden Lampe zugreift und öffnet diesen. Nun sieht er die in Echtzeit animierten Zustandsübergänge des Prozesses. Er sieht, dass sich der Prozess zur Zeit im Zustand “Bright” befindet und betätigt den Aus-Schalter. Statt wie erwartet den Zustand “Dark” einzunehmen, wechselt der Prozess vorerst auf “Delayed”. Dadurch konnte eine Erklärung für das unerwartete Verhalten gefunden werden.

UC03: Zustandsänderungen manuell nachverfolgen

Aktor

Techniker

Beschreibung

Bei einem anderen Teil der Lichtanlage, in dem sehr schnelle Farbänderungen auftreten, ist der Techniker ebenfalls auf Unklarheiten gestossen. Da im entsprechenden Prozess jedoch sehr viele Zustandsübergänge pro Sekunde auftreten, kann er aus der Animation keine Erkenntnisse gewinnen. Deshalb setzt er einen Marker auf einen Zustand, der ihm besonders interessant erscheint. Sobald der Prozess den fraglichen Zustand erreicht hat, stoppt die Animation (ohne die Anlage anzuhalten) und alle nachfolgenden Zustandsänderungen werden im Hintergrund aufgezeichnet. Durch das Betätigen der Vorwärts und Rückwärts Tasten, kann der Techniker nun in aller Ruhe die Zustandsänderungen Schritt für Schritt durchgehen.

UC04: Ursache von Notstopp finden

Aktor

Techniker

Beschreibung

Durch eine Input Message, die für den aktuellen Zustand des Systems nicht definiert ist, wird die Lichtanlage automatisch angehalten. Da auf der Steuerung ein kleiner Buffer die letzten Zustandsänderungen aufzeichnet, kann der Techniker die Vorkommnisse kurz vor dem Notstopp manuell durchgehen und daraus Rückschlüsse auf die Ursache des Notstopps herleiten.

2.3. Weitere Anforderungen

2.3.1. Allgemeine Qualitätsanforderungen

Um eine hohe Qualität der zu entwickelnden Software zu gewährleisten, werden in diesem Abschnitt die Anforderungen in Teilen gemäss [ISO01] erhoben und beschrieben.

Interoperabilität

Die verschiedenen Teile von ModVis soll möglichst transparent miteinander kommunizieren. So soll sich die Webapplikation immer gleich verhalten, egal ob sie mit einem lokal installierten Server, der an eine Simulation angebunden ist, oder einem entfernten Echtzeitsystem verbunden ist.

Interoperabilität ist insofern erforderlich, dass ModVis alle im *CIP* Tool entwickelten Systeme darstellen kann.

Zuverlässigkeit

Das System soll soweit fehlertolerant sein, so dass es auf dem Server zu keinen Ausfällen kommt und allfällige Fehleingaben abgefangen und behandelt oder ignoriert werden. Hierzu soll eine Strategie ausgearbeitet werden, um einem Ausfall des Servers entgegen zu wirken.

Die Wiederherstellbarkeit auf Clientseite kann nicht vollumfänglich gewährleistet werden, da bei einem Absturz des Browsers die dynamisch geladenen Daten unumgänglich verloren sind. Eine Funktion um diesen Umständen entgegen zu wirken wird nicht weiterverfolgt, da sich das Diagramm im Verlauf des Browser Neustarts bereits in einem neun Zustand befindet. Der Server soll sich nur mit den aktuellen Zuständen bzw. deren Übergängen befassen, wodurch die Möglichkeit der Speicherung von Benutzerdaten, und somit auch der Wiederherstellbarkeit, entfällt.

Benutzerfreundlichkeit

Die Bedienung im Allgemeinen zielt darauf ab benutzerfreundlich und reaktionsfähig zu sein, also kurze Antwortzeiten zwischen Aktion und Reaktion aufweisen. Die Beschriftungen der Schaltflächen sollen klar verständlich und ihre Funktion offensichtlich sein.

Die Navigation durch die Modelldiagramme soll sich möglichst an der entsprechenden Darstellung in actifsource orientieren.

Effizienz

Die Tabelle 2.1 listet die Randbedingungen auf, die ModVis im Minimum erfüllen muss. Darunter wird verstanden, dass bei diesen Bedingungen noch alle abonnierten Zustandsänderungen über die Netzwerkverbindung übertragen und dargestellt werden können. Zudem

sollen Diagramme in der Browseranwendung in weniger als einer Sekunde geladen und angezeigt werden. Die Verzögerung zwischen dem geschehen der Zustandsänderung auf der Steuerung und der entsprechenden Animation soll 0.5s nicht überschreiten.

Kriterium	Zielwert
Zustandsübergänge im gesamten System [1/s]	1000
Zustandsübergänge in einem Zustandsautomat [1/s]	100
Anzahl Zustände im gesamten System	100
Anzahl Zustände in einem Prozess	10
Anzahl Clients	1
Anzahl geöffnete Diagramme pro Client	5

Tabelle 2.1.: Randbedingungen für die Performancemessungen

Wartungsfreundlichkeit

Durch die Aufteilung in eine Clientseitige Applikation, ein definiertes Protokoll und einen eigenständigen Webserver soll die Wartung und Erweiterbarkeit der Applikation wesentlich vereinfacht. Die Änderungen an der Oberfläche auf der Clientseite soll so zu keinen Änderungen am Server führen und durch die Definition eines Kommunikationsprotokolls, erkennt der Client keine Unterschiede bei Modifikation an der Server Implementation.

Übertragbakeit

Das System soll in drei unterschiedlichen Szenarien ausgeführt werden können:

1. Simulation der Steuerung auf einem PC
2. Steuerung auf einem TCP fähigem Embedded System
3. Steuerung auf einem nicht TCP fähigem Embedded System, über Bussystem mit Webserver verbunden (optional)

Durch die Umsetzung als Webapplikation wird auf Clientseite kein bestimmtes Betriebssystem vorausgesetzt. Zudem soll die Webapplikation unabhängig vom unterliegenden Szenario funktionsfähig sein.

2.3.2. Skalierung

Da von Benutzerzahlen im einstelligen Bereich ausgegangen wird, werden keine besonderen Anforderungen an die Skalierbarkeit des Systems gestellt.

2.3.3. Lokalisierung

Im Rahmen der Bachelorarbeit soll die Lokalisierung für den deutschsprachigen Raum der Schweiz vorgenommen werden.

2.3.4. Security

Da es sich bei den verwendeten Daten um keine sensitiven Informationen handelt, wird nicht weiter auf Verschlüsselung bzw. Sicherung der Kommunikation geachtet. Des Weiteren befinden sich die Systeme bereits in dezentralen Netzwerken, wodurch der Zugriff auf die Geräte nur autorisiertem Personal ermöglicht wird.

2.3.5. Error Handling

Ungültige Requests dürfen auf keinen Fall die Funktionsweise des Hostsystems beeinträchtigen. Deshalb kann die Serveranwendung diese ignorieren oder falls möglich mit einem Aussagekräftigen Statuscode beantworten. Fehlerhafte Eingaben sollten also soweit als möglich Clientseitig bearbeitet werden.

2.4. Evaluation Kommunikationstechnologie

Als Grundlage für die Ausarbeitung einer geeigneten Architektur soll zuerst ein geeignetes Kommunikationsmodell evaluiert werden. Dies ist notwendig, da je nach eingesetzter Kommunikationstechnologie der Kontrollfluss zwischen Client und Server unterschiedlich ausfallen kann.

2.4.1. Kriterienkatalog Kommunikationstechnologie

Damit eine ideale Wahl der Kommunikationstechnologie zwischen Client und Server getroffen werden kann, werden geeignete Kriterien zur Evaluation der Technologie definiert. Im ersten Abschnitt werden organisatorische Kriterien festgelegt. Worauf im zweiten Teil die technischen Kriterien definiert werden. Dabei werden die gefunden Kriterien nach ihrer Relevanz zur Erfüllung der Anforderungen eingestuft, wodurch versucht wird die Entscheidung für eine definitive Technologiewahl zu vereinfachen.

Die **organisatorischen Kriterien** sind vorwiegend für die Umsetzung der Lösung in Bezug auf Aufwand zur Wissenserarbeitung und mögliche Wissensquellen wichtig.

K1: *Hype Cycle* und Zukunftsträchtigkeit

- In welcher Phase des *Hype Cycle*s befindet sich die Technologie?
- Hat die Technologie Zukunft?

Bewertung: kein Standard, nicht oft genutzter Standard, Standard mit Zukunft, verbreiteter Standard
Relevanz: mittel

K2: Hersteller- oder Community-Support

- Gibt es eine Community mit fundiertem Fachwissen zu der Technologie?
- Bestehen Herstellerlösungen mit Supportmöglichkeiten?

Bewertung: nicht vorhanden, mässig, vorhanden, ausgeprägt
Relevanz: mittel

K3: Dokumentation

- Gibt es informative Einführung bzw. ausführliche Getting Started Tutorials? Wie ist die Qualität dieser?
- Bestehen Best-Practices Beispiele?

- Gibt es weiterführende Literatur?

Bewertung: keine, ungenügend, gut, ausführlich
Relevanz: hoch

K4: Implementationsaufwand und -kosten

- Muss alles neu implementiert werden oder gibt es Libraries?
- Falls Libraries vorhanden, nur Client, nur Server oder beides?
- Sind die bestehenden Libraries kostenpflichtig?

Bewertung:

- Eigene Lösung nötig, nur Client, nur Server, Framework
- zusätzlich jeweils: kostenpflichtig oder frei verfügbar

Relevanz: hoch

K5: Ausbildung und Aufwand

- Wie gut ist das Wissen des Projektteams bezüglich der Technologie?
- Wie hoch ist der Einarbeitungsaufwand in eine bestehende Library?

Bewertung:

- kein Vorwissen, mässig, vorhanden, fundiert
- zusätzlich jeweils Aufwand: keiner, mittel, hoch

Relevanz: mittel

Die **technischen Kriterien** beziehen sich auf die Möglichkeiten der verschiedenen Technologien und deren Ausprägung. Hierbei werden wiederum funktionale und nichtfunktionale Kriterien unterschieden.

K6: *Message Exchange Pattern (MEP)*

- Welche *MEP* werden unterstützt?

Bewertung:

- Server-Push möglich, request/response
- synchron, asynchron
- Verbindungsart: Unicast, Multicast, Channel based

Relevanz: mittel

K7: Datenformate

- Plaintext (keine Typsicherheit), XML, *JavaScript Object Notation (JSON)* (keine Typsicherheit)
- Ist das Übertragen von Binärdaten möglich? (z.B. Bilder)
- Gibt es Generatoren für das Un-/Marshalling?

Bewertung:

- unterstützt, nicht unterstützt
- binär oder nicht
- Generatoren vorhanden: ja/nein

Relevanz: hoch

K8: Einschränkungen

- Gibt es serverseitig Einschränkungen durch Anforderungen an die Hardware? (Bsp. nur x86 Prozessorfamilie)
- Bestehen clientseitig Einschränkungen durch Policies durch Browser? (Bsp. *Same Origin Policy*)
- Müssen Richtlinien von Spezifikationen eingehalten werden?
- Gibt es Grössenbegrenzung der zu übertragenden Daten?

Bewertung: vorhanden (Falls ja, welche?), keine
Relevanz: hoch

Die folgenden Kriterien beziehen sich auf die in Abschnitt 2.3 beschriebenen nichtfunktionalen Anforderungen:

K9: Testbarkeit

- Bestehen Werkzeuge, mit welchen unabhängig von Server oder Client, die Kommunikationstechnologie getestet werden kann?
- Können automatisierte Tests eingesetzt werden?

Bewertung: keine, manuell, automatisiert
Relevanz: mittel

K10: Performance

- Wird ein Overhead generiert? (Traffic)
- Wie hoch ist die Antwortzeit pro Nachricht? (Latency)

Bewertung:

- Overhead: gross, mässig
- Latency: langsam (20ms+), genügend (10-20ms), schnell (<10ms)

Relevanz: hoch

K11: Skalierbarkeit

- Mit welcher Anzahl Verbindungen von unterschiedlichen Clients kann das Kommunikationsmodell umgehen?
- Wieviele Request bzw. Messages kann das Modell vom gleichen Client beantworten?

Bewertung:

- Clients: wenige (0-1K), einige (1K-20K), viele (20K+)
- Requests pro Client: wenige (<1K/s), einige (1K-20K/s), viele (20K+/s)

Relevanz: niedrig, da actifsource die Anzahl Clients auf einen begrenzt hat

K12: Kompatibilität clientseitig

- Ist die Technologie bezüglich Hardware und Betriebssystem plattformunabhängig?

- Ist die Kompatibilität gewährleistet, werden alle benötigten Browser unterstützt?

Bewertung:

- Client OS: Microsoft Windows, Linux, Mac OS X
- Browser: *Microsoft Internet Explorer (IE)* 9.0+, *Mozilla Firefox (FF)* 19.0+, Chrome 25.0+, Safari 6.0

Relevanz: niedrig, da actifsource für dieses Projekt *FF* als Referenz festgelegt hat

K13: Kompatibilität Embedded System

- Wird die Technologie bereits vom Embedded System unterstützt?
- Sind zusätzliche Libraries zur Umsetzung notwendig? Und sind diese kompatibel mit dem System?
- Kann die Technologie mit angemessenen Zeitaufwand komplett neu implementiert werden?

Bewertung: native, kompatibel, Eigenimplementation
Relevanz: hoch

Die hier zusammengetragenen Kriterien sind die für dieses Projekt nötigen und können für andere oder ähnliche Projekte variieren. Weitere Punkte könnten Caching, Security oder Clock (Taktrate zwischen Client und Server) sein, werden hier aber wegen fehlender Anforderungen weggelassen.

2.4.2. Überblick der Technologiealternativen

Alternative 1: Polling

Das Polling-Verfahren beschreibt das zyklische Abfragen von Informationen vom Client beim Server (Request und Response). Dabei wird auf dem Client eine Endlosschleife implementiert die in vordefinierten Zeitintervallen beim Server nachfragt ob neue Daten vorhanden sind. Dadurch wird die im HTTP-Protokoll fehlende Funktion, Daten ohne vorhergehenden Request des Clients vom Server an den Client zu schicken, umgangen. Dies wird vorzugsweise mit Hilfe von *Asynchronous JavaScript and XML (AJAX)* und dem *XMLHttpRequest (XHR)* gelöst, wobei bei jedem durchlauf der Schlaufe ein neuer *XHR*-Request an den Server gesendet wird. (siehe Abbildung 2.4)

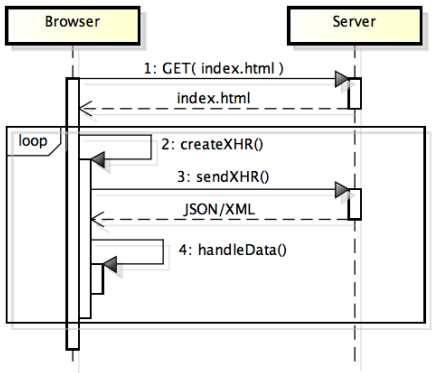


Abbildung 2.4.: Ablauf-Diagramm: Polling Verfahren

Die Definition der abrufbaren Daten und Operationen geschieht dabei über bekannte Techniken wie z.B. *Web Services Description Language (WSDL)* oder *Simple Object Access Protocol (SOAP)* Web Services.

Alternative 2: HTTP-Streaming

Wie allgemein in der Informatik wird bei Kommunikationstechnologien im Bereich Webservices unter Streaming das übertragen von Informationen in Form eines Datenflusses ohne absehbarem Ende verstanden. Bei HTTP-Streaming [Lor+11] werden zwei Möglichkeiten unterschieden, zum einen das Hidden-Frame und zum anderen der Einsatz von *XHR*.

Bei der Hidden-Frame Technik wird ein *iframe*-Element in einer HTML Seite eingebaut, in welches vom Server blockweise `<script>`-Tags geschriebenen werden. Diese `<script>`-Tags werden vom Browser inkrementel, interpretiert und fortlaufend geladen bzw. ausgeführt. Somit können Daten vom Server an den Client übertragen werden, ohne das für jede Übertragung ein HTTP-Request vom Client geöffnet werden muss. In der Abbildung 2.5 wird der Ablauf in einer abstrahierten Form dargestellt. Zu beachten ist, dass der "loop"-Block nur für das Senden und Interpretieren der `<script>`-Tags gilt, die index.html-Datei wird asynchron zu diesem Prozess fertig geladen bzw. interpretiert. Deshalb empfiehlt es sich das *iframe*-Element am Ende der Datei einzufügen, so dass alle anderen Tags bereits vom Browser erkannt wurden.

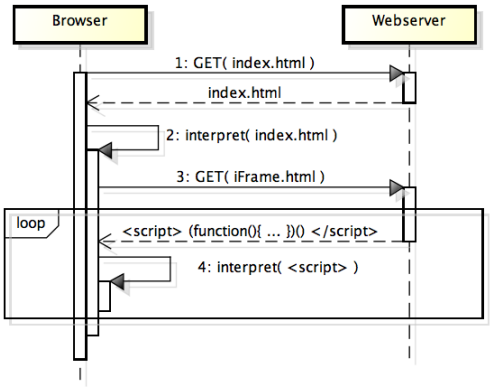


Abbildung 2.5.: Ablauf-Diagramm: Hidden-Frame Streaming

Der zweite Ansatz basiert auf der *XHR* Schnittstelle des Browsers und der Option den Content-Type des Response-Headers auf `multipart/x-mixed-replace` zu setzen. Dadurch wird der Browser angewiesen, die Verbindung vom Server offen zu halten, wodurch der Server die angeforderte Ressource in mehreren Stücken ausliefern kann. Da dieser Content-Type nicht von allen Browser unterstützt wird, kann alternativ der Response wie bei der Hidden-Frame Methode auf dem Server ohne Angabe des expliziten Content-Types offen gehalten werden um die zu übertragenden Daten wiederum fortlaufend in den Response zu schreiben. In beiden Varianten wird clientseitig eine Javascript-Callback Funktion auf den "onReadyStateChange"-Event des Request registriert. Die registrierte Funktion wird nun bei jeder Mitteilung vom Server aufgerufen, womit die empfangenen Daten verarbeitet werden können. Der Ablauf kann in Abbildung 2.6 nachverfolgt werden. Hierbei ist wichtig, dass das Senden des Response-Headers einmal vor der eigentlichen Datenübertragung stattfindet und der "loop"-Block für alle Durchläufe den selben offenen Response-Kanal verwenden.

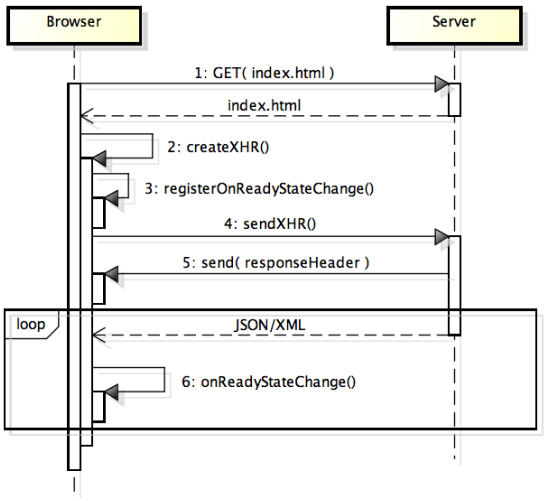


Abbildung 2.6.: Ablauf-Diagramm: *XHR* Streaming

Alternative 3: Long-Polling

Die Long-Polling [Lor+11] Variante ist eine Erweiterung der Alternative 1: Polling. Hierbei werden dem Server ebenfalls Anfragen gesendet, welche jedoch nicht sofort beantwortet werden, sondern solange offen gehalten werden, bis neue Daten zur Übertragung bereitstehen. Dadurch können die Anzahl Requests an den Server massiv gesenkt werden. Hierzu gibt es wiederum zwei unterschiedliche Implementationsansätze, zum einen *XHR*-Long-Polling und zum anderen das `<script>`-Tag Long Polling. Mit der `<script>`-Tag Long Polling Variante besteht die Möglichkeit den abzurufenden Code auf einem anderen Server zu platzieren als die eigentliche Seite. Dies ist aufgrund der vom Browser durchgesetzt *Same Origin Policy* bei den anderen Technologien nicht möglich. Die Abbildung 2.7 zeigt die Long Polling Variante mit dem *XHR*-Ansatz. Hierbei gibt es einen äusseren “loop”, in welchem jeweils neue *XHR*-Request generiert werden, nachdem ein Response vom Server empfangen wurde. Auf dem Server wird wiederum nach Empfang des *XHR*-Requests (3) in einem “loop” solange gewartet, bis neue Daten vorhanden sind.

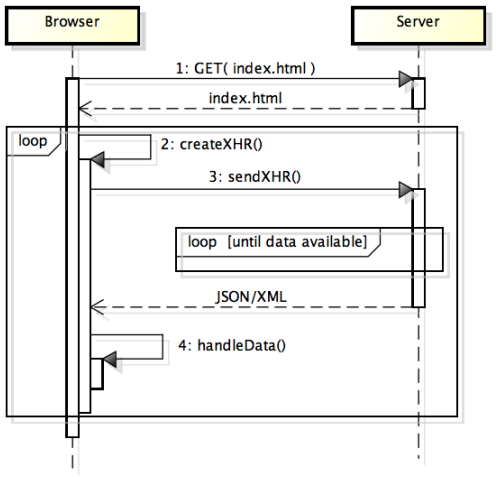


Abbildung 2.7.: Ablauf-Diagramm: *XHR* Long-Polling

Alternative 4: WebSockets

Die WebSocket-Technologie beschreibt neben HTTP ein eigenes Protokoll [FM11], welches auf TCP aufbaut und dem Browser bzw. Webserver ermöglicht einen bidirektional Kanal aufzubauen. Um eine WebSocket-Verbindung aufbauen zu können, führen die beiden Parteien (Server und Client) zu Beginn einen Handshake durch. Dabei erhält die Verbindung einen sogenanntes "Connection: upgrade" auf das WebSocket-Protokoll. Danach kann der Browser über den Server Port 80 mit dem Webserver kommunizieren und dieser seinerseits mit dem Client. Obwohl diese Technologie ursprünglich für Webapplikationen entwickelt wurde, kann dieses Protokoll auch unabhängig davon für andere Applikationen genutzt werden.

2.4.3. Vergleich der Technologien

Aus Tabelle 2.2 geht hervor, dass die Polling Varianten gegenüber WebSocket und HTTP-Streaming weiter verbreitet sind und von den Projektmitgliedern auch schon eingesetzt wurden. Die ungenügende Dokumentation und der mässige Support von Industrie und Community sprechen klar gegen die Verwendung einer HTTP-Streaming Lösung. Aus einer zukunftsorientierten Sicht wäre die Technologie WebSocket zu empfehlen, wobei bei dieser ein erhöhter Lernaufwand für die Projektmitglieder gegenüber den Polling-Strategien bestehen würde.

Kriterium	Alternative 1: Polling	Alternative 2: HTTP-Streaming	Alternative 3: Long-Polling	Alternative 4: WebSockets
K1: *Hype Cycle* und Zukunftsträchtigkeit	verbreiteter Standard	nicht oft genutzter Standard [FM11]	verbreiteter Standard [FM11]	Standard mit Zukunft
K2: Hersteller- oder Community-Support	ausgeprägt	mässig	ausgeprägt	vorhanden
K3: Dokumentation	gut	ungenügend	gut	ausführlich
K4: Implementationsaufwand und -kosten	gering (Framework)	gering (Framework)	gering (Framework)	gering (Framework)
K5: Ausbildung und Aufwand	vorhanden / keiner	kein Vorwissen / hoch	vorhanden / keiner	vorhanden / mittel

Tabelle 2.2.: Technologievergleich mit organisatorischen Kriterien

Die Tabelle 2.3 zeigt, dass sich die unterschiedlichen Kommunikationstechnologien lediglich in deren *MEP* unterscheiden. Dabei hebt sich WebSocket mit der Zweiweg (push/pull) Kommunikationsmöglichkeit gegenüber den anderen Technologien (request/response) ab. Die *Same Origin Policy* schränkt ausser *iframe*-Streaming, alle Technologien ein, kann jedoch mit Änderungen am Response Header umgangen werden. Da in diesem Projekt jedoch alle Daten vom gleichen Server ausgeliefert werden und somit von der gleichen Origin stammen, müssen keine Änderungen vorgenommen werden.

Kriterium	Alternative 1: Polling	Alternative 2: HTTP-Streaming	Alternative 3: Long-Polling	Alternative 4: WebSockets
K6: *MEP*	request / response, synchron, Unicast	request / response, "asynchron", Unicast	request / response, synchron, Unicast	push/pull, asynchron, alle Verbindungsarten
K7: Datenformate	alle, binär möglich	alle, binär möglich	alle, binär möglich	alle, binär möglich
K8: Einschränkungen	*Same Origin Policy*	XHR-Streaming *Same Origin Policy*, iFrame keine	*Same Origin Policy*	*Same Origin Policy*

Tabelle 2.3.: Technologievergleich mit funktionalen Kriterien

Die Kriterien "K10: Performance" und "K11: Skalierbarkeit" aus der Tabelle 2.4 wurden auf Grund der Quellen [Sim11] und [Nir] bestimmt. Hierbei fällt auf, dass es für das vorliegende Projekt keinen wesentlichen Unterschied macht, welche Technologie eingesetzt wird, da sich die unterschiedlichen Varianten bei kleinen Client und Request Zahlen gleich Verhalten. Zum Kriterium "K12: Kompatibilität clientseitig" ist zu erwähnen, dass *WebSocket*s auch unter *IE* ab Version 10.0 eingesetzt werden könnten.

Kriterium	Alternative 1: Polling	Alternative 2: HTTP-Streaming	Alternative 3: Long-Polling	Alternative 4: WebSockets
K9: Testbarkeit	automatisiert	automatisiert	automatisiert	automatisiert
K10: Performance (Overhead, Latency)	goss, genügend	mässig, schnell	gross, genügend	mässig, schnell
K11: Skalierbarkeit	einige Clients und Requests	einige Clients und Requests	einige Clients und Requests	viele Clients und Requests
K12: Kompatibilität clientseitig	alle	alle	alle	alle, ausser IE
K13: Kompatibilität Embedded System	native	Eigenimplementation	native	kompatibel

Tabelle 2.4.: Technologievergleich mit nicht-funktionalen Kriterien

2.4.4. Entscheidungsempfehlung

Der Hauptunterschied bei den Technologien liegt hauptsächlich beim eingesetzten *MEP*. Dabei hebt sich die Alternative 4: WebSockets Variante gegenüber den konventionellen Varianten ab, da diese Technologie als einzige eine echte bidirektionale Verbindung zur Verfügung stellt. Bei den organisatorischen Kriterien heben sich die beiden Polling-Varianten hingegen deutlich von den anderen Technologien ab. Die Kriterien K2: Hersteller- oder

Community-Support, K3: Dokumentation und K5: Ausbildung und Aufwand sind hierbei ausschlaggebend für ein bevorzugen der Polling-Technologien. Über alle Kriterien gesehen empfiehlt es sich deshalb eine der Polling-Varianten in der Implementation zu verwenden, da diese bereits auf den Embedded System native unterstützt werden und ohne grösseren Aufwand eingesetzt werden können. Jedoch sollte Long-Polling in Anbetracht des geringeren Datenverkehrs gegenüber der einfachen Polling Variante bevorzugt werden.

3. Universelle Domäne zur Animation von Prozessabläufen

Aus den funktionalen Anforderungen (Abschnitt 2.2) geht hervor, dass ein Datenmodell benötigt wird, welches beliebige graphisch beschriebene Prozessabläufe mithilfe von Zustandsdaten des Systems animieren kann. Als Referenzmodell für die Konzeptionierung der Domäne wird die *CIP*-Methode herangezogen, für die actifsource bereits eine umfängliche Werkzeugunterstützung bereitstellt.

3.1. Spezifische Domäne eines Referenzmodelles

Abbildung 3.1 zeigt ein vereinfachtes konzeptionelles Modell der *CIP*-Methode. Das *System* beschreibt die auf einer Steuerung ausgeführte Software und stellt die oberste Hierarchiestufe dar. Es enthält mehrere *Cluster*, die über asynchrone *Channels* kommunizieren. Channels ermöglichen zudem den Austausch von *Messages* zwischen Clustern und der Umgebung des Systems (Sensoren und Aktoren). Das Modell beschränkt sich jedoch lediglich auf die Entitäten, die die Struktur des Systems beschreiben und nicht dessen Verhalten (wie z.B. *Channel*, *Inpulse*, *Outpulse*, *Message*). [Fie99]

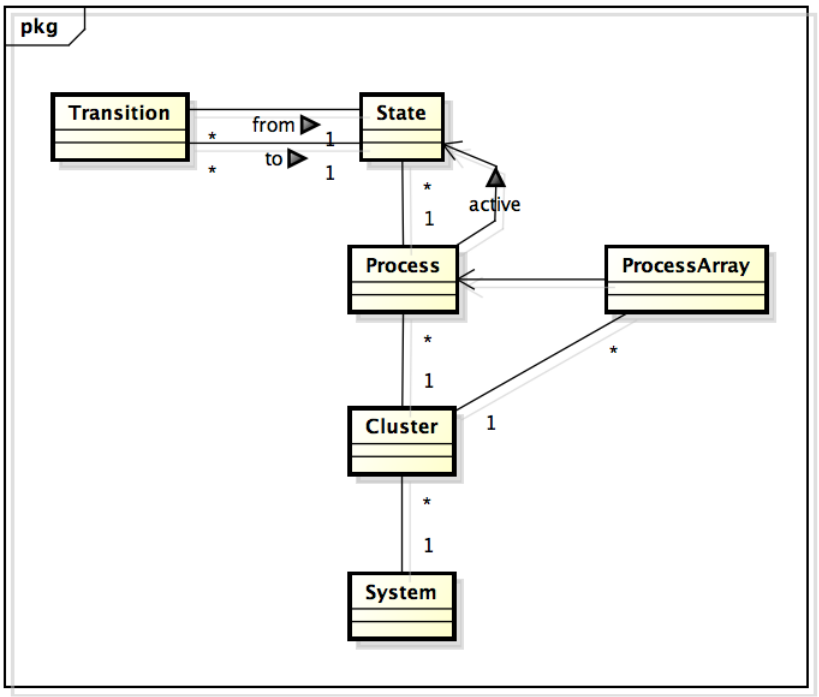


Abbildung 3.1.: Domain Model CIP

Ein *Cluster* stellt eine Sammlung von nebenläufigen Prozessen dar. Diese Prozesse kommunizieren synchron über den Austausch von *Inpulses* respektive *Outpulses*.

Ein Prozess repräsentiert einen um Timer und Operationen erweiterten *Deterministischen Endlichen Automaten* (*DEA*). Da es sich um einen deterministischen Automaten handelt, kann zu jedem Zeitpunkt nur genau ein Zustand aktiv sein.

3.2. Generalisierung der Domäne

Da die Organisationsstufen des CIP-Modells jeweils durch "part of" Relationen verbunden sind, können die Entitäten (System, Cluster, Process und ProcessArray) für die Visualisierung auch in einer Container-Entität zusammengefasst werden. Dies ist möglich, da bei der Visualisierung keine Kenntnisse über das Verhalten dieser Komponenten benötigt werden. Abbildung 3.2 zeigt dieses verallgemeinerte Modell.

Das generalisierte Modell bietet den Vorteil, dass ModVis von der *CIP*-Domäne weitgehend entkoppelt werden kann. Die Lösung wird somit universeller einsetzbar und ist robuster gegenüber Änderungen in der actifsource Entwicklungsumgebung.

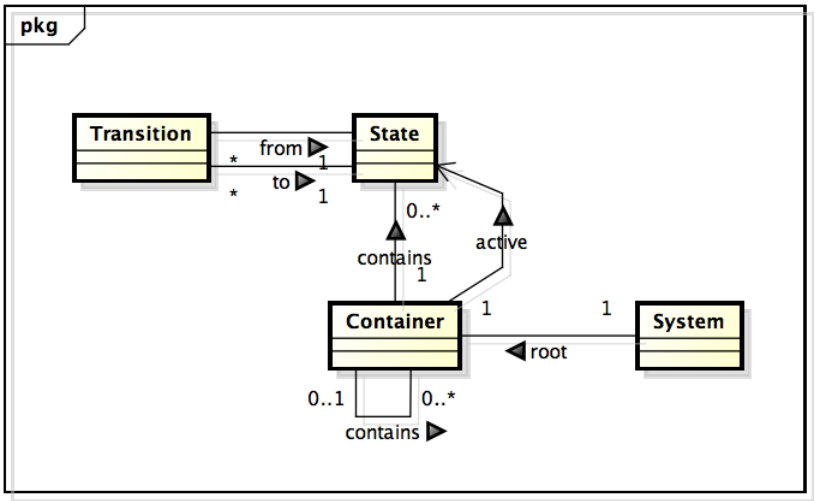


Abbildung 3.2.: Generalisiertes Domain Model

Die Systementität stellt im generalisierten Modell nicht mehr eine Ansammlung von Komponenten dar, sondern beschreibt nur noch die Attribute der Steuerung wie z.B. den Systemnamen und verfügt über eine Referenz auf den obersten Container der Hierarchie. Alle Container können weitere Container enthalten sowie mehrere Zustände mit den entsprechenden Übergängen.

Die wichtigste Voraussetzung für die Vereinheitlichung der Komponenten ist jedoch, dass alle Elemente innerhalb eines Systems eindeutig identifiziert werden können. Dies kann entweder in der Form einer *UUID* oder als hierarchische Adressierung im Stil einer URI [BFM05] implementiert sein.

3.3. Erweitertes Modell gemäss den Anforderungen

Das generalisierte Modell der CIP-Domäne impliziert, dass das System Kenntnisse darüber hat, wie sich die Komponenten des animierten Diagrammes verhalten. Dies ermöglicht einerseits eine sehr effiziente Kommunikation und Datenhaltung, da nur die Information eines Zustandsüberganges benötigt wird, um den Zustand eines Diagramms zu einem

bestimmten Zeitpunkt zu rekonstruieren. Andererseits wird der Einsatz von zusätzlichen Diagrammtypen dadurch erschwert.

Damit die Erweiterbarkeit der Software sichergestellt ist, wird ModVis auf Basis des in Abbildung 3.3 dargestellten Modells umgesetzt. Das Modell wurde zudem um die Aspekte der Darstellung (`ElementState`) der Elemente und Aufnahme der Systemzustände erweitert. Des Weiteren wird das System vom Container abgeleitet und wird nicht wie in Abbildung 3.2 als separate Systementität geführt.

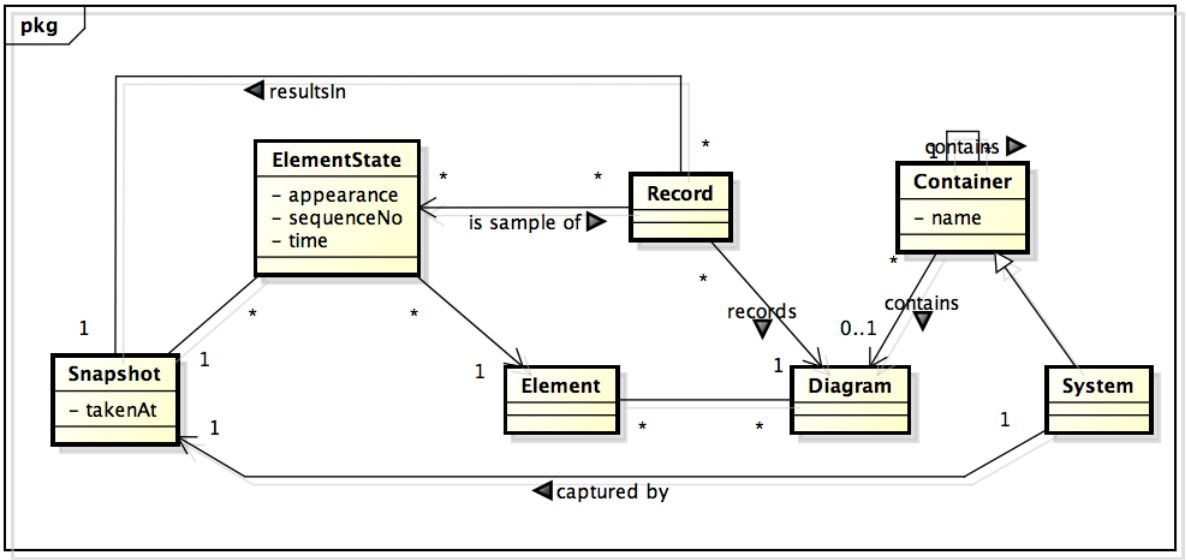


Abbildung 3.3.: Finales Domain Model von ModVis

3.3.1. Systemhierarchie

Wie im generalisierten CIP-Modell, repräsentieren `Container` die Struktur des Echtzeitsystems (Cluster, Prozesse etc.). `System` stellt dabei das Wurzelelement der Systemhierarchie dar. Dadurch sind alle Container direkte oder indirekte Nachfahren des Systemcontainers.

Die Systemhierarchie eignet sich, um eine Navigationsstruktur auf der Benutzeroberfläche aufzubauen, die mit der dem Benutzer bekannten Struktur aus der actifsource Umgebung identisch ist.

3.3.2. Diagramme und Diagrammelemente

Falls das Modell, in dem die Steuerung des Systems entworfen wurde (z.B. *CIP*), für einen Container eine grafische Darstellung vorsieht, verweist der Container auf das entsprechende `Diagram`. Das Diagramm wird durch ein generiertes *SVG* dargestellt, welches verschiedene Elemente enthält, die animiert werden können.

Ein `Element` ist eine Instanz einer Entität, die unterschiedliche Zustände einnehmen kann. In einem CIP-Zustandsdiagramm sind dies alle Zustände und Zustandsübergänge. Die Darstellung eines Elementes zu einem bestimmten Zeitpunkt wird in der `ElementState` Entität festgehalten. (Abbildung 3.3)

Das `appearance` Attribut verweist auf eine der möglichen Darstellungsarten von Elementen. Die Attribute `time` und `sequenceNo` werden benötigt, um den Zustand zeitlich einzuordnen, z.B. zum Zeitpunkt `time=100` und `sequenceNo=42` hatte das Element

`appearance=1`. Dabei wird `time` verwendet, um den absoluten Zeitpunkt zu ermitteln, zu dem der Zustand eingetreten ist. Da der Zeitpunkt eines Ereignisses aber unter Umständen nicht genau genug bestimmt werden kann, um zwei kurz aufeinanderfolgende Ereignisse zu trennen, wird die `sequenceNo` benötigt. Diese ermöglicht relative Aussagen über die Abfolge von Ereignissen im Sinne von "der Zustand mit der Sequenznummer 11 ist vor dem Zustand mit der Sequenznummer 12 eingetreten". `ElmentStates` können auch über identische Sequenznummern verfügen, falls die Ereignisse in der abgebildeten Problemdomäne miteinander auftreten. In einem Zustandsdiagramm der CIP-Domäne tritt zum Beispiel die Transaktion von Zustand A zu Zustand B zeitgleich mit der Deaktivierung des Zustands A und der Aktivierung des Zustands B auf. Eine zeitliche Trennung der drei Ereignisse wird nicht benötigt.

3.3.3. Momentaufnahme des Systemzustands mit Snapshots

Die Darstellung aller Diagramme in einem System zu einem Zeitpunkt wird in einem `Snapshot` abgebildet. Ein `Snapshot` enthält somit zu jedem Element im System den `ElementState`, der zum Zeitpunkt `takenAt` gültig ist. Somit hat jeder `ElementState` eines Snapshots eine `sequenceNo` die kleiner oder gleich `takenAt` ist und es existiert jeweils nur ein ElementState pro `sequenceNo` für das selbe Element.

Diese Definition erlaubt es, mithilfe des Snapshots A und der Differenz zwischen Snapshot A und B den Inhalt von Snapshot B zu rekonstruieren, falls der Snapshot A einen Zeitpunkt vor Snapshot B abbildet. Die Differenz der beiden Snapshots sind alle ElementStates von B die eine höhere `sequenceNo` als `takenAt` von Snapshot A aufweisen.

3.3.4. Aufnahme der Zustandsänderungen mit Records

Für die Aufzeichnung aller Änderungen eines Diagramms über einen bestimmten Zeitraum werden in einem `Record` alle betroffenen ElementStates zusammengefasst. Im Gegensatz zum Snapshot enthält ein Record mehrere ElementStates eines Elementes. Da ein Record nur alle Änderungen eines Diagramms während eines bestimmten Zeitraumes enthält, kann der Gesamtzustand des Diagramms zu einem Zeitpunkt nicht aus dem Record allein hergeleitet werden. Aus diesem Grund wird ein Snapshot benötigt, von dem aus mithilfe der einzelnen Samples (Abbildung 3.3 Abhängigkeit `record` und `ElementState`) der Zustand aller Elemente rekonstruiert werden kann.

3.4. Einschränkungen des Modells

Durch die weitgehende Abstraktion des Domänenmodelles wird ModVis weitgehend von der CIP-Methode entkoppelt und es wird ein sehr breites Einsatzgebiet ermöglicht. Allerdings entstehen dadurch auch einige Einschränkungen bei der Abbildung spezifischer Modelle.

Da die ModVis Domäne keinerlei Kenntnis über das Verhalten eines animierten Diagramms besitzt, müssen explizit alle Änderungen des Zustandes eines Elements bekannt sein. Im Falle eines Diagramms eines *DEA* bedeutet dies, dass bei jedem Zustandsübergang der alte Zustand deaktiviert und der neue Zustand aktiviert werden muss. Mit dem Wissen, dass es sich beim animierten Diagram um einen deterministischen Zustandsautomaten handelt, würde das Aktivieren des neuen Zustandes ausreichen, um auf die korrekte Darstellung des Diagramms schliessen zu können, indem implizit alle anderen Zustände deaktiviert werden. Durch die Generalisierung des Modells kann somit unter Umständen eine eigentlich überflüssige Datenredundanz notwendig werden.

Eine weitere Einschränkung des vorgeschlagenen Modells betrifft die Umsetzung eines Process Arrays aus der *CIP*-Domäne. Da CIP die Instanzen eines Prozesses im Process Array nicht als eigenständig identifizierbare Ressourcen umsetzt, können diese Instanzen nicht in separaten Diagrammen dargestellt werden. Da dieses Problem jedoch sehr spezifisch für die CIP-Domäne ist, wurde zusammen mit dem Auftraggeber beschlossen, das Konzept von mehreren Instanzen eines Elementes nicht in die ModVis Domäne aufzunehmen.

3.5. Konzessionen an domänenspezifisches Verhalten

Da, wie im vorhergehenden Abschnitt erwähnt, durch den Verzicht auf jegliches Wissen über das Verhalten eines Diagramms unter Umständen deutlich mehr Daten für dessen Animation benötigt werden als eigentlich notwendig, lohnt es sich, das Modell wieder an die jeweilige Zieldomäne anzunähern. Dazu werden für gewisse Diagramme alternative Strategien für deren Animation definiert.

Eine solche `AnimationStrategy` wird vor dem Darstellen eines Snapshots auf diesen angewendet. Dadurch können implizite Änderungen am Diagrammzustand vorgenommen werden, die nicht explizit übermittelt werden müssen.

Für die Abbildung der CIP-Domäne werden die folgenden Animationsstrategien benötigt:

default Standardmässig verwendete Strategie; alle Zustandswechsel müssen explizit übertragen werden.

implicitDiagramReset Vor dem Anwenden eines Snapshots auf ein Diagramm werden implizit alle Elemente auf ihre Standarddarstellung zurückgesetzt.

Durch die `implicitDiagramReset` Strategie müssen zum Beispiel bei Diagrammen von deterministischen Zustandsautomaten die nicht mehr aktiven Zustandselemente nach einer Transition nicht deaktiviert werden.

Die vorgestellte Domäne für die Animation von Prozessabläufen verfügt somit durch die starke Generalisierung über die notwendige Flexibilität, um Diagramme aus verschiedenen Zieldomänen animieren zu können. Durch die Möglichkeit, alternative Animationsstrategien auf Diagrammtypen anzuwenden, kann zudem die durch die Generalisierung entstandene Informationsredundanz wieder verringert werden.

4. Design einer flexiblen Komponentenarchitekur

ModVis ist eine Komponentenarchitektur mit unterstützenden Artefakten, die das Einbinden zusätzlicher Funktionalität in ein bestehendes System ermöglichen. Aus diesem Grund ist neben den einzelnen Komponenten auch der Arbeitsfluss fur die Integration von ModVis von zentraler Bedeutung fur die Architektur.

Da einige gängige Begriffe aus dem Software Engineering in der *CIP*-Domäne zum Teil über eine zusätzliche Bedeutung verfügen, wird in den folgenden Abschnitte explizit die CIP-Domäne erwähnt, falls ein Begriff in diesem Kontext zu verstehen ist. Zum Beispiel umschreibt der Begriff "Zustand" in der CIP-Domäne ein Element eines Zustanddiagramms. Allgemeiner kann darunter aber auch der Zustand einer Software respektive eines Systems im Sinne der Werte aller definierten Variablen verstanden werden. "Prozess", "Cluster" und "Message" sind weitere Begriffe, die auch als Entitäten der CIP-Domäne verstanden werden können.

4.1. Architekturübersicht

Grundlage der Architektur von ModVis sind die definierten Schnittstellen zwischen den beteiligten Komponenten. Diese Schnittstellen erlauben eine lose Koppelung der einzelnen Programmteile und ermöglichen durch deren Austauschbarkeit eine hohe Portabilität der gesamten Lösung. Abbildung 4.1 zeigt eine vollständige Übersicht aller beteiligten Komponenten und wie diese über die Schnittstellen kommunizieren.

Wie aus der Abbildung 4.1 an dem annotierten Stereotyp «common» ersichtlich ist, verfügt ModVis über drei Komponenten, die Plattform- und Projektunabhängig eingesetzt werden können:

ModVis Webservice
: Der Webservice empfängt Updates zu den Zuständen der Diagrammelemente von der Steuerungssoftware des animierten Systems. Über eine an dem *REST*-Architekturstil angelehnte HTTP-Schnittstelle können beliebige Clients die aufgezeichneten Daten in Form von Snapshots und Records anfordern.

ModVis Client
: Der Client ist eine eigenständig ausführbare Browserapplikation und benötigt kein dynamisches, serverseitiges Rendering. Angelehnt an die *Serviceorientierte Frontend Architektur (SOFEA)* nach [Nel13] wird die gesamte Bedienungs- und Darstellungslogik auf den Browser ausgelagert.

ModVis Code Templates
: Da Client und Service auf Konfigurationsdateien angewiesen sind, die die Struktur des zu animierenden Systems beschreiben, wird eine Erweiterung der actifsource Umgebung benötigt. Die Erweiterung kann für jedes actifsource Projekt, welches ein definiertes Konfigurationsobjekt zur Verfügung stellt, die benötigten Dateien automatisch generieren.

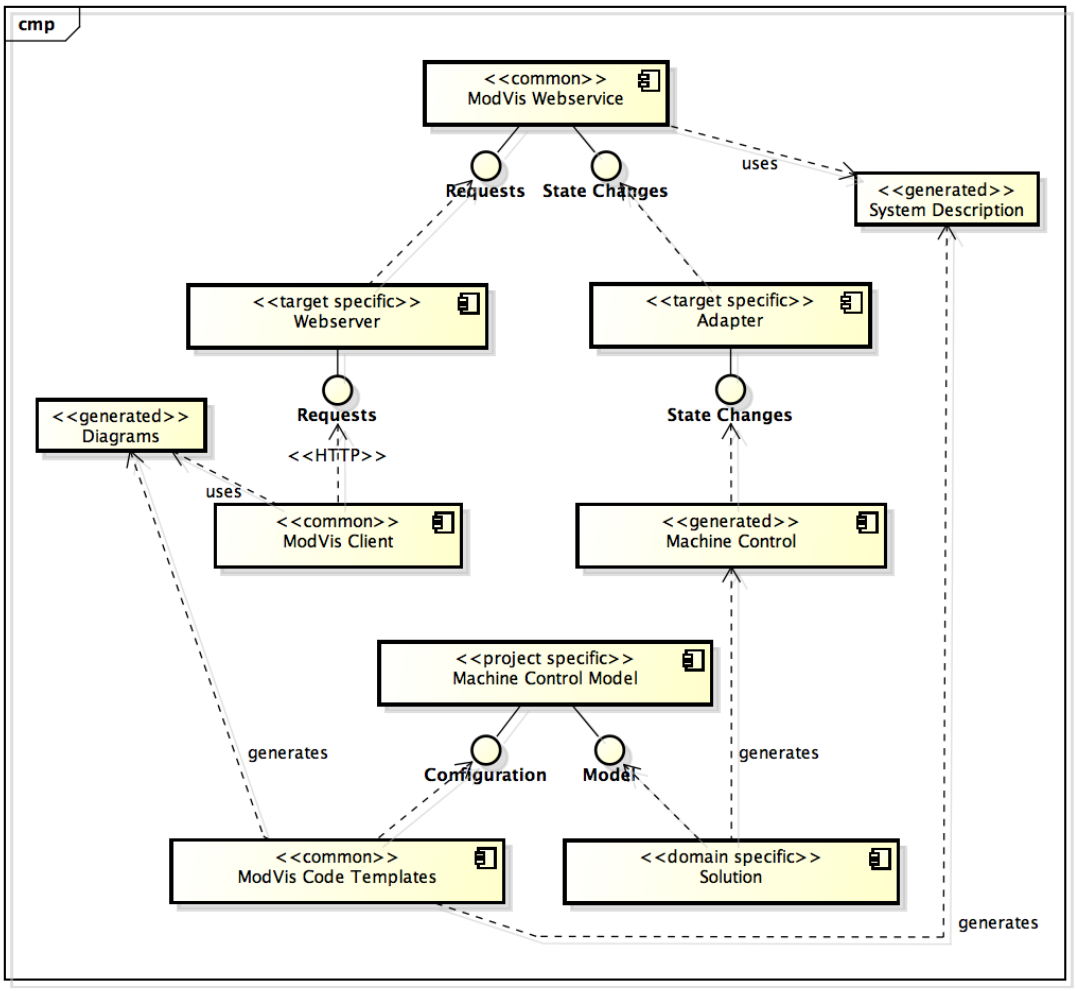


Abbildung 4.1.: UML-Komponentendiagramm über alle Komponenten und Schnittstellen

Für jede zu unterstützende Zielplattform werden zudem weitere Komponenten benötigt, die jedoch projektunabhängig wiederverwendet werden können. Diese sind durch den Stereotyp «target specific» ausgezeichnet.

Webserver
: Der ModVis Webservice wird auf der Zielplattform von einem Webserver eingebunden. Dieser leitet anschliessend alle HTTP Requests über eine *Common Gateway Interface (CGI)* ähnliche Schnittstelle an den Service weiter.

Adapter
: Da die Probleme der Interprozesskommunikation und Nebenläufigkeit je nach Zielplattform unterschiedlich gelöst werden müssen, werden die Benachrichtigungen über Zustandsänderungen über eine zusätzliche Abstraktionsschicht dem ModVis Webservice übergeben. Diese Abstraktionsschicht kann als Adapter verstanden werden, der der Zielplattform entsprechend ausgetauscht wird.

Eine weitere Kategorie von Komponenten stellen die domänenspezifischen Teile dar («domain specific»). Diese können ebenfalls für unterschiedliche Projekte in der selben Domäne wiederverwendet werden. Ein Beispiel einer solchen Domäne ist die CIP Methode. Diese definiert ein Metamodell mit den unterschiedlichen Typen, die in einem Modell

eingesetzt werden können, sowie Aussehen und Inhalt der möglichen Diagramme. Das Metamodell enthält somit auch das Wissen über die Art der Information, die in einem Diagramm dargestellt wird, und aufgrund welchen Ereignisse die Darstellung eines Diagrammelementes geändert werden soll.

Solution
: Eine *Solution* umfasst in actifsource das Metamodell einer bestimmten Problemdomäne, die Definition der verwendbaren Diagrammtypen sowie Code Templates zur Generierung von Quellcode (bei der CIP Solution unter anderem C-Quellcode). Damit die Animation durch ModVis unterstützt werden kann, müssen diese Code Templates um die Funktionsaufrufe zum ModVis Adapter Interface erweitert werden.

Die letzte Kategorie umfasst die projektspezifische Komponenten («project specific»). Für das Einbinden der Visualisierung in ein actifsource Projekt müssen gewisse Bedingungen erfüllt werden. Die Architektur ist jedoch so konzeptioniert, dass die notwendigen manuellen Anpassungen möglichst minimiert werden können und setzt zu einem grossen Teil auf die automatische Codegenerierung («generated»).

Machine Control Model
: Das in actifsource entworfene Modell der Steuerung enthält unter anderem die vom Entwickler erstellten Diagramme der verwendeten Domäne. Damit die von ModVis benötigten Dateien generiert werden, muss der Entwickler zusätzlich die ModVis Code Templates referenzieren und ein gültiges Konfigurationsobjekt erstellen.

Machine Control
: Der Code der Steuerungssoftware wird anhand der Code Templates der verwendeten Solution generiert. Die Software ruft, nach für die Animation relevanten Ereignissen, die entsprechende Funktion auf dem ModVis Adapter Interface auf.

Diagramme
: Damit der ModVis Client die Diagramme darstellen kann, müssen diese als Vektorgrafiken im *SVG* Format verfügbar sein. Zudem werden für das Aufbauen eines Navigationsmenüs Informationen zur Projektstruktur in Form einer *JSON* Datei benötigt. Diese Dateien werden automatisch mithilfe der ModVis Code Templates generiert.

System Description
: Der ModVis Service benötigt einige Informationen über die Beziehungen zwischen den verschiedenen Diagrammen und deren Elemente, damit die Zustandsinformationen effizient verwaltet und aufbereitet werden können. Diese Informationen werden direkt in eine C-Datenstruktur generiert, auf die der ModVis Webservice Zugriff bekommt. Dadurch können teure I/O-Operationen sowie die fehleranfällige dynamische Speicherallokation umgangen werden.

Durch das vorgestellte Komponentenmodell wird eine hohe Flexibilität beim Deployment von ModVis ermöglicht, was eine wichtige Voraussetzung für die erforderte Portabilität ist.

4.2. Verschiedene mögliche Deploymentmodelle

Aus den im Kapitel 2 beschriebenen Anforderungen geht hervor, dass ModVis auf verschiedenen Hardwaretypen in unterschiedlichen Konfigurationen verteilt werden soll. Zudem sollen auch auf PCs simulierte Modelle visualisiert werden können.

Dadurch, dass die plattformspezifischen Aspekte von der ModVis Service Library durch eine Abstraktionsschicht entkoppelt sind, besteht eine sehr hohe Flexibilität beim Deployment der Servicekomponente. Nachfolgend werden einige der möglichen Deploymentmodelle beschrieben.

4.2.1. Steuerungslogik wird von Webserver ausgeführt

Dieses Szenario ist insbesondere für die Simulation eines Modells auf der Maschine des Entwicklers interessant. Der Webserver wird dabei von einem Kommandozeilenprogramm ausgeführt, welches simulierte Ereignisse über Benutzereingaben verarbeiten kann.

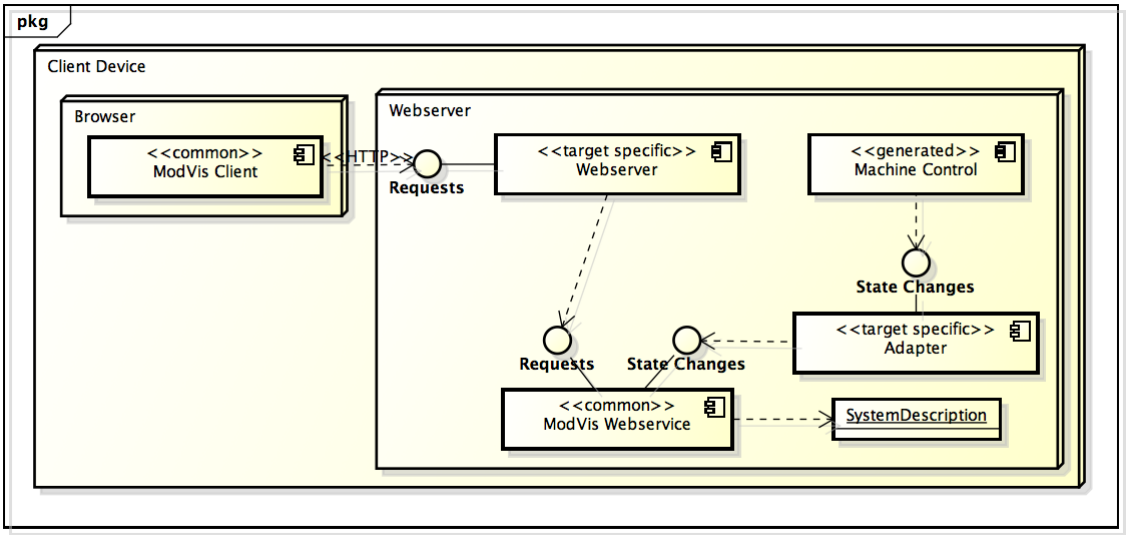


Abbildung 4.2.: UML-Deploymentdiagramm von ModVis auf einem Prozess

Abbildung 4.2 zeigt das Deployment von ModVis in einer solchen Einprozessumgebung. Da keine Interprozesskommunikation benötigt wird, müssen im Adapter lediglich alle Funktionsaufrufe an den ModVis Service weitergeleitet werden.

4.2.2. Steuerungslogik und Webserver auf getrennten Prozessen

Da der Betrieb von Webserver und Steuerungssoftware durch einen gemeinsamen Prozess aufgrund der fehlenden Nebenläufigkeit nicht geeignet ist für den Einsatz auf Echtzeitsystemen, empfiehlt sich für den produktiven Einsatz ein Deployment auf zwei unterschiedlichen Prozessen gemäss Abbildung 4.3.

Bei diesem Deploymentmodell wird die *Interprozesskommunikation (IPC)* im Adapter gekapselt. Auf Seite des Steuerungsprozesses nimmt der Adapter die Zustandsänderungen entgegen und übermittelt diese (z.B. über Sockets) an den Webserverprozess. Die andere Hälfte des Adapters auf dem Webserverprozess empfängt die Informationen und wandelt diese wieder in die entsprechenden Funktionsaufrufe auf dem ModVis Service um.

4.2.3. Steuerungslogik und Webserver auf getrennten Systemen

Analog zum Deploymentmodell auf zwei getrennten Prozessen können Webservice und Steuerung auch auf getrennten Systemen betrieben werden. Dies kann notwendig werden,

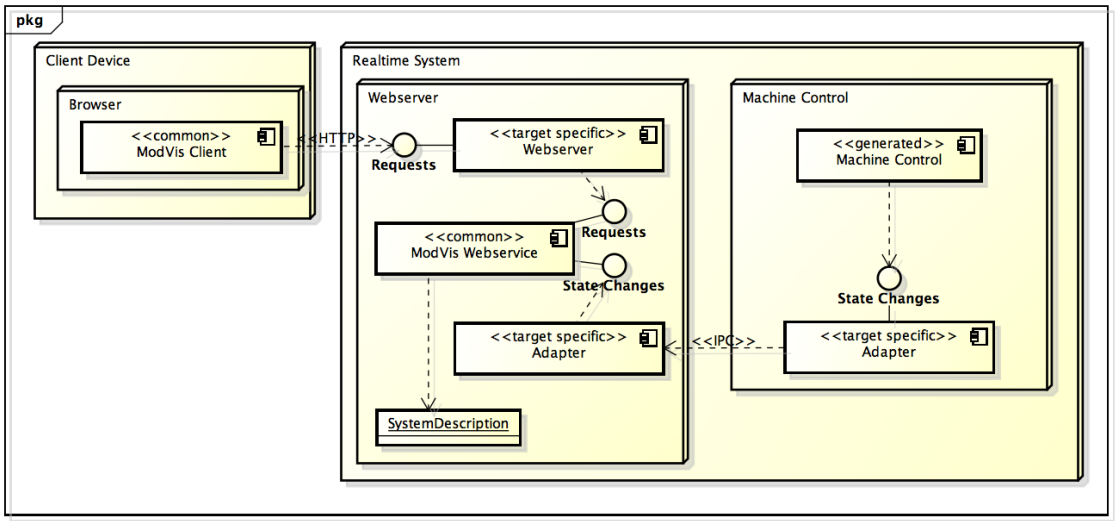


Abbildung 4.3.: Deployment von ModVis auf separaten Prozessen

falls die Steuerungshardware nicht über eine Ethernet Schnittstelle verfügt. Der Webservice kann in diesem Fall zum Beispiel auf einem über ein Bussystem verbundenen *Single Board Computer* betrieben werden.

Die Konfiguration ist bei diesem Szenario die selbe wie beim Deployment auf zwei getrennten Prozessen gemäss Abbildung 4.3, ausser dass die *IPC* zwei physikalisch getrennte Systeme verbindet.

4.2.4. Getrennte Prozesse mit geteiltem Speicherbereich

Da die *IPC* über Sockets, Pipes oder ähnliche Mechanismen oft aufwendigere Implementationen benötigt und deren Koordination durch das Betriebssystem relativ teuer ist, ist die Lösung mit zwei komplett getrennten Prozessen nicht immer praktikabel. Falls die Zielplattform geteilte Speicherbereiche zulässt kann auch das in Abbildung 4.4 beschriebene Deploymentmodell verwendet werden.

In dieser Konfiguration wird sowohl auf dem Webserverprozess wie auch auf dem Steuerungsprozess je eine Instanz des ModVis Services ausgeführt. Beide Instanzen greifen auf die gemeinsam genutzte Datenstruktur mit der Systembeschreibung zu. Die Instanz auf dem Steuerungsprozess ist dabei zuständig für das Aktualisieren der Daten, die direkt vom Adapter weitergeleitet werden. Die Instanz auf dem Serverprozess greift ausschliesslich lesend auf die Datenstruktur zu.

Durch diese Konfiguration kann eine einfache *IPC* realisiert werden, die ohne teure Synchronisation auskommt. Dies ist möglich, da der ModVis Webservice ausschliesslich mit einer statischen Datenstruktur arbeitet und keine Referenzen manipuliert werden. Allerdings müssen einige Einschränkungen beachtet werden:

Nur ein Schreibprozess erlaubt

Falls mehr als ein Steuerungsprozess nebenläufig ausgeführt wird und alle schreibend auf die Datenstruktur zugreifen, können zwischen diesen write-write Konflikte auftreten. In gewissen Fällen können write-write Konflikte die Datenstruktur korrumpieren und es kann zu unerwarteten Fehlern kommen. *Bei zwei oder mehr nebenläufigen Steuerungsprozessen ist somit eine Synchronisation zwingend notwendig!*

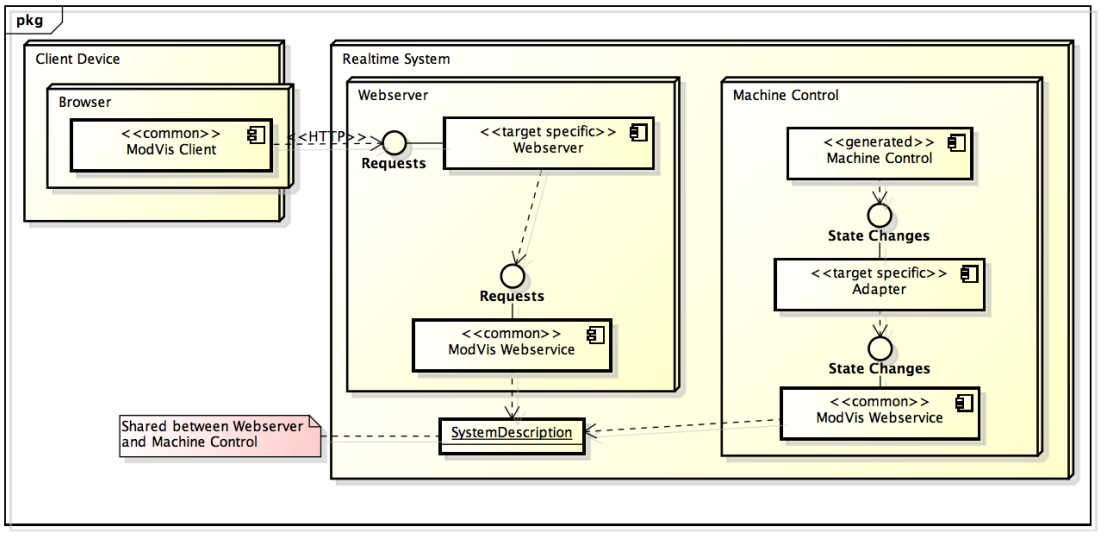


Abbildung 4.4.: Deployment von ModVis auf separaten Prozessen mit Zugriff auf eine geteilten Speicherbereich

Inkonsistente Daten möglich

Beim Auslesen der Daten durch den Webserverprozess können bei gleichzeitigem schreiben durch den Steuerungsprozess inkonsistente Daten an den Client gesendet werden. Abgesehen von einer inkorrekten Animation im Browser sind jedoch keine weiteren Konsequenzen zu erwarten.

Vorwärtskompatibilität möglicherweise eingeschränkt

Da die ModVis Web-API zur Zeit nur Leseoperationen zulässt, ist garantiert, dass durch den Webserverprozess nur lesend auf die Systembeschreibung zugegriffen wird. Falls in einer zukünftigen Version die Schnittstelle um zusätzliche Funktionalitäten erweitert wird, die auch Manipulationen der Daten auf dem Server erfordern, kann diese Garantie jedoch nicht mehr sichergestellt werden. Dadurch könnten durch Updates auf eine neuere Version von ModVis unerwartete Nebenläufigkeitsprobleme auftreten.

Trotz den erwähnten Einschränkungen ist das Realisieren der *IPC* mithilfe von geteilten Speicherbereichen ein pragmatischer Ansatz für die meisten Anwendungsfälle. Sofern die erwähnten Punkte beachtet werden, sind auch keine Stabilitätseinbussen auf der Zielplattform zu erwarten.

Insgesamt kann durch die beschriebenen Deploymentmodelle eine sehr breite Abdeckung von möglichen Einsatzszenarien erreicht werden. Zudem kann die Implementation des Adapters gut auf die auf dem Echtzeitsystem vorhandenen Ressourcen und Features ausgerichtet werden.

4.3. Grundlagen zur Client-Server-Kommunikation

Die Evaluation des Kommunikationsmodells in Abschnitt 2.4 ergab, dass für einen flexiblen Einsatz auf unterschiedlichen Plattformen HTTP-Polling, respektive für den konkreten Anwendungsfall von ModVis HTTP-Long-Polling, zu bevorzugen ist.

Während der Umsetzung des ersten Prototypen auf der *Automation Runtime (AR)*

Laufzeitumgebung, stellte sich jedoch heraus, dass HTTP-Long-Polling für den Einsatz auf derartigen Plattformen nicht ideal ist.

Der in der *AR* Laufzeitumgebung integrierte Webserver verfügt über die Einschränkung, dass ein registrierter Webservice die Requests der Reihe nach abarbeiten muss. Falls ein Request noch nicht beantwortet ist und ein weiterer empfangen wird, wird der neue Request in eine Warteschlange eingereiht. Diese Warteschlange darf auf maximal drei Einträge anwachsen. Alle weiteren Requests werden vom Server umgehend mit dem HTTP-Statuscode `500 Internal Server Error` beantwortet. Long-Polling kann jedoch nur effektiv eingesetzt werden, falls mehrere, nebenläufig abgearbeitete Requests möglich sind. Diese Einschränkung ist zwar sehr spezifisch für die *AR* Plattform, allerdings ist damit zu rechnen, dass andere Laufzeitumgebungen für Echtzeitsysteme ebenfalls auf eine sequentielle Requestverarbeitung setzen, da diese nur sehr wenig Ressourcen beansprucht und eine sehr hohe Fehlertoleranz aufweist [Lea00].

Aus diesem Grund wird für die Client-Server-Kommunikation ausschliesslich auf klassisches HTTP-Polling gesetzt. Um eine zeitnahe Animation der Diagramme zu ermöglichen, müssen jedoch kontinuierlich im Abstand von 100 bis 200 ms neue Daten angefordert werden. Damit die Serverbelastung nicht zu schnell wächst, werden durch das Multiplexing von Anfragen gewisse Requests in einem einzigen zusammengefasst.

4.4. Spezifikation der HTTP-Schnittstelle

Die HTTP-Schnittstelle zwischen dem ModVis Webservice und der Webapplikation definiert ein auf HTTP aufbauendes Kommunikationsmodell. Die Schnittstelle wird so weit als möglich im Sinne des *REST* Architekturstils nach [Fie00] entworfen. Folgende Kernpunkte von *REST* sind ebenfalls in der ModVis Web-API umgesetzt:

Aufgabentrennung zwischen Client und Server
: Die Darstellung der Benutzeroberfläche und Verwaltung deren Zustands wird ausschliesslich durch den Client ausgeführt. Der Server verfügt über keinerlei Kenntnisse über die Darstellungs- und Bedienungslogik auf der Benutzeroberfläche.

Zustandslose Kommunikation
: Jeder Request enthält alle benötigten Informationen damit dieser vom Server abgearbeitet werden kann.

Einheitliche Schnittstelle
: Der Server stellt Ressourcen zur Verfügung, die über die HTTP-Methoden manipuliert werden können.

Neben einer erhöhten Skalierbarkeit ermöglichen diese Prinzipien eine starke Entkopplung zwischen Client und Server und erhöhen die Austauschbarkeit einzelner Komponenten.

4.4.1. Zeichenkodierung

Die ModVis Web-API verwendet ausschliesslich die *ASCII* Zeichenkodierung ohne länderspezifische Erweiterungen [Cer69]. Dies erleichtert die Portierung auf verschiedene Plattformen, da von einer Unterstützung des *ASCII* Zeichensatzes immer ausgegangen werden kann.

Zudem ergab die Anforderungsanalyse keine Notwendigkeit um Texteingaben des Benutzers verarbeiten zu können. Die Menge der verwendeten Zeichen kann deshalb durch die Schnittstelle gut eingeschränkt werden.

4.4.2. Serialisierung

Für die Serialisierung von Objekten zur Datenübertragung sowie zur Persistierung wird das *JSON* Format verwendet. Wobei als *Multipurpose Internet Mail Extensions (MIME)*-Typ nach [Cro06] `application/json` zur Übertragung an den Browser verwendet wird. *JSON* wird der *Extensible Markup Language (XML)* vorgezogen, da dessen Syntax schlanker und etwas einfacher ist. Durch den geringeren Overhead von *JSON* wird während der Serialisierung weniger Speicher benötigt, was auf einem Echtzeitsystem mit beschränkten Ressourcen von grosser Bedeutung ist.

```
1 <Snapshot takenAt="8400">
2   <ElementStates>
3     <ElementState element="101" appearance="active" time="7480" />
4     <ElementState element="102" appearance="inactive" time="7480" />
5   </ElementStates>
6 </Snapshot>
```

Listing 4.1: Snapshot als XML serialisiert

```
1 {
2   "takenAt":8400,
3   "elementStates":[
4     {"element":101,"appearance":"active","time":7480},
5     {"element":102,"appearance":"inactive","time":7480}
6   ]
7 }
```

Listing 4.2: Snapshot als JSON serialisiert

Die Listings 4.1 und 4.4 zeigen die unterschiedliche Verbosität von *XML* und *JSON*. Das erste Listing umfasst 194 Zeichen, das zweite 136 (jeweils ohne optionale Leerzeichen). Dieses Beispiel ermöglicht zwar keine fundierte Aussage über einen allgemeinen Overhead von *XML* gegenüber *JSON*, zeigt aber eine gewisse Tendenz.

Da jedes JSON Dokument ein gültiges JavaScript Objekt Literal darstellt, eignet sich JSON zudem besser für den Einsatz in Rich Client Webapplikationen.

4.4.3. Caching

Da alle zurzeit definierten Methoden der Schnittstelle den aktuellen Zustand des Systems zurückgeben ist ein Caching der Antworten grundsätzlich nicht möglich.

Damit das Caching in Browsern, Proxies und weiteren Netzwerkaktoren deaktiviert wird, muss der Response Header `Cache-Control: no-cache` gesetzt sein.

4.4.4. Ressourcenbasierte URLs

Im Sinne des HTTP Standards [Fie+99] repräsentieren die *Uniform Resource Identifier (URI)*-Pfade der Schnittstelle Ressourcen auf denen Methoden ausgeführt werden können. Der Pfad `.../bilder/2` könnte beispielsweise die Bildressource mit der ID 2 in einer Sammlung von Bildern beschreiben.

Da die ModVis Web-API für den Zugriff auf einige Ressourcen ein Request Multiplexing vorsieht und das oben erwähnte Schema nur die Adressierung einer Ressource erlaubt, wird ein leicht angepasstes Schema verwendet. Der Pfad in der *URI* adressiert

weiterhin die Sammlung von Ressourcen, auf die zugegriffen wird. Die Ressourcen selber werden jedoch durch einen oder mehrere Queryparameter identifiziert. Das vorhergehende Beispiel mit dem Zugriff auf Bilder würde in der ModVis API durch die URI `.../bilder?bild=2` ersetzt. Der Zugriff auf mehrere Bilder wäre dementsprechend durch `.../bilder?bild=2&bild=3&...` ermöglicht.

Mögliche vom Standard [Fie+99] definierte HTTP Methoden, die auf eine Ressource angewendet werden können sind:

GET Gibt eine Ressource zurück, ohne weitere Seiteneffekte auf dem Server

POST Erstellt eine neue Ressource

PUT Ersetzt eine bestehende Ressource

DELETE Löscht eine Ressource

Die aktuelle Version der ModVis Web-API definiert zur Zeit ausschliesslich GET Abfragen auf Ressourcen. Datenmanipulationen sind (noch) keine vorgesehen.

4.4.5. Methoden der HTTP Schnittstelle

Snapshot anfordern

`GET /snapshot` gibt den aktuellen Zustand aller Elemente von einem oder mehreren Diagrammen zurück.

Erforderliche Parameter

`diagram` UUID des betroffenen Diagrammes; kann mehrmals vorkommen

Optionale Parameter

`fromSequenceNo` Die Sequenznummer ab der die Änderungen berücksichtigt werden sollen; falls nicht angegeben wird der Wert `0` verwendet

Beispielrequest

```
GET /snapshot?diagram=9fb088cd-c1ee-11e2-9f5d-13cdb5f85860&fromSequenceNo=2
```

Beispielresponse

```
1  {
2      "takenAt": 1370698735,
3      "lastSequenceNo": 5,
4      "elementStates": [
5          {
6              "element": "9f8f4501-c1ee-11e2-9f5d-13cdb5f85860",
7              "appearance": 1,
8              "time": 1370698733,
9              "sequenceNo": 3
10         },
11         {
12             "element": "9f8f4507-c1ee-11e2-9f5d-13cdb5f85860",
13             "appearance": 1,
14             "time": 1370698733,
15             "sequenceNo": 3
16         }
17     ]
```

```
18 }
```

Listing 4.3: JSON-serialisiertes Snapshot Objekt

Die Response enthält den JSON-serialisierten Systemsnapshot wobei dessen Attribut `elementStates` nur die Menge der ElementStates enthält, die auf die Request Parameter zutreffen. `takenAt` enthält die Serverzeit zum Zeitpunkt des letzten Updates des Systemsnapshots in Sekunden. `lastSequenceNo` ist die Sequenznummer des zuletzt eingefügten ElementStates.

Das `lastSequenceNo` Attribut kann als `fromSequenceNo` Parameter für den nächsten Request verwendet werden. Dadurch erhält der Client lediglich die Differenz der Zustandsänderungen seit dem letzten Request.

Record anfordern

`GET /record` gibt die auf dem Server vorhandene Aufzeichnung der letzten Zustandsänderungen eines Diagramms zurück.

Erforderliche Parameter

`diagram` UUID des betroffenen Diagrammes

Beispielrequest

```
GET /record?diagram=9fb088cd-c1ee-11e2-9f5d-13cdb5f85860
```

Beispielresponse

```
1  {
2      "elementStates": [
3          {
4              "element": "9f8f4507-c1ee-11e2-9f5d-13cdb5f85860",
5              "appearance": 1,
6              "time": 1370698733,
7              "sequenceNo": 3
8          },
9          // ...
10         {
11             "element": "9f8f4501-c1ee-11e2-9f5d-13cdb5f85860",
12             "appearance": 1,
13             "time": 1370700541,
14             "sequenceNo": 19
15         }
16     ]
17 }
```

Listing 4.4: JSON-serialisiertes Record Objekt

Die Response enthält einen JSON-serialisierten Record der alle aufgezeichneten Zustandsänderungen des betroffenen Diagramms im `elementStates` Attribut zusammenfasst. Die Grösse des Records ist von der Systemkonfiguration abhängig.

4.4.6. Minimierung der Anzahl und Grösse der Requests

Die beschrieben Schnittstelle ist darauf ausgelegt, dass der Server möglichst wenig Requests bearbeiten muss und trotzdem noch eine zeitnahe Darstellung des Systemzustands

möglich ist. Insbesondere sollen keine nebenläufige Requests notwendig sein um mehrere Diagramme animieren zu können.

Die Anzahl und Grösse der Requests wird durch zwei Konzepte minimiert:

Diagramsnapshots gemeinsam anfordern
Mit der `GET /snapshot` Methode können die aktuellen Zustandsdaten aller geöffneten Diagramme in einem Request angefordert werden.

Nur Differenz zum bekannten Zustand anfordern
Der Client fordert durch das Setzen des `fromSequenceNo` Parameters nur die Differenz zum bereits bekannten Diagrammzustand an. Dadurch müssen nur noch die tatsächlich benötigten Zustandsdaten übermittelt werden.

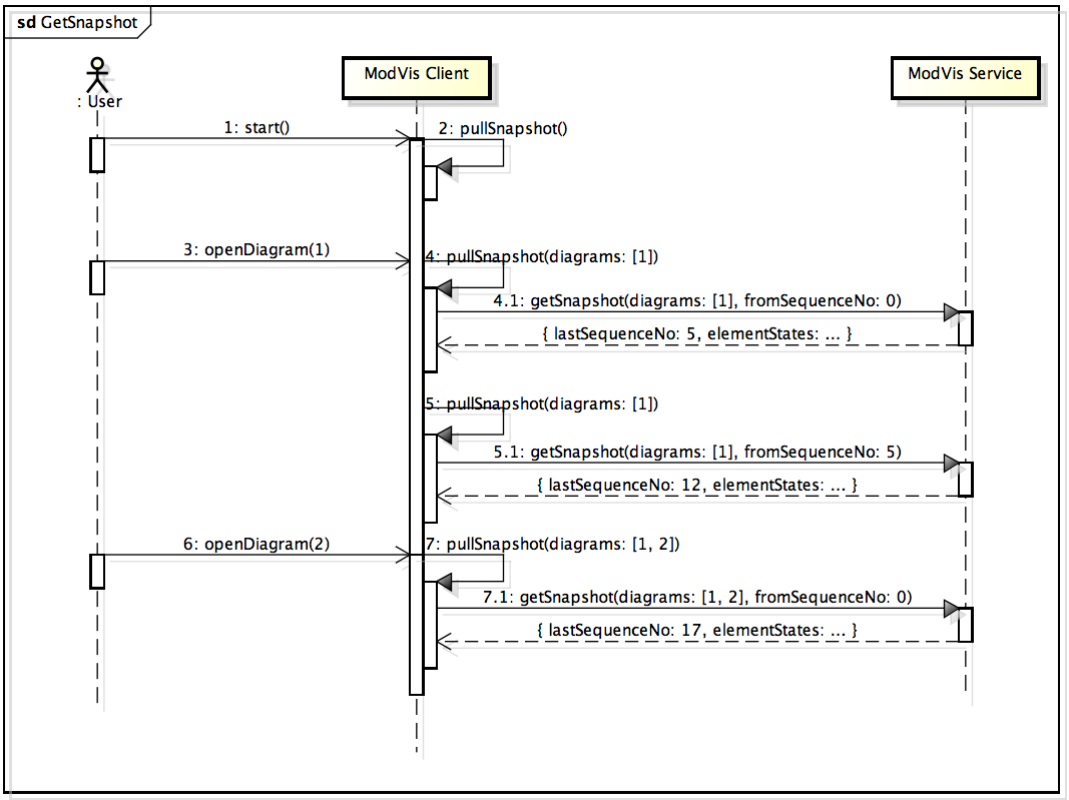


Abbildung 4.5.: Kontinuierliches anfordern des Snapshots über die ModVis Web-API

Das in Abbildung 4.5 dargestellte Sequenzdiagramm erläutert die Prinzipien hinter der Client-Server-Kommunikation anhand eines Beispiels:

1. Der Benutzer startet den ModVis Client.
2. Der Client überprüft kontinuierlich ob Zustandsdaten angefordert werden müssen, da keine Diagramme geöffnet sind, werden keine Requests gesendet.
3. Der Benutzer öffnet das Diagramm mit der ID 1.
4. Der Client fordert den Snapshot mit den Zustandsdaten für Diagramm 1 an. Die `GET /snapshot` Methode wird mit dem Attribut `fromSequenceNo = 0` aufgerufen, da noch keine Zustandsdaten bekannt sind.

   a) Die Response enthält alle Zustandsdaten des Diagramms 1. Die letzte dem Server bekannte Sequenznumber ist 5.

5. Der Client benötigt nur noch die Daten ab der Sequenznummer 5. `GET /snapshot` wird mit `fromSequenceNo = 5` aufgerufen.
   a) Die Response enthält die Differenz des Diagrammzustandes seit der Sequenznummer 5.

6. Der Benutzer öffnet das Diagramm mit der ID 2.

7. Da noch keine Informationen zum Zustand von Diagramm 2 vorhanden sind, wird `GET /snapshot` wieder mit `fromSequenceNo = 0` aufgerufen.
   a) Die Response enthält alle Zustandsdaten der Diagramme 1 und 2.

Nach dem Öffnen des zweiten Diagramms müssten die Daten für Diagramm 1 eigentlich nicht mehr ab der Sequenznummer 0 angefordert werden, da diese dem Client schon bekannt sind. Damit die Schnittstelle einfach ausfällt, wurde jedoch beschlossen, dass der `fromSequenceNo` Parameter immer für alle angeforderten Diagramme gültig ist. Zudem wird im Verhältnis zu allen Requests relativ selten ein neues Diagramm geöffnet. Öffnet ein Benutzer zum Beispiel alle 10 Sekunden ein neues Diagramm und der Client fordert alle 100 ms die neuen Zustandsdaten an, so wird nur in jedem hundertsten Request der gesamte Diagrammzustand angefordert.

4.5. Plattformunabhängiger Webservice

Der ModVis Webservice wird im Hinblick auf eine möglichst breite Plattformunterstützung konzeptioniert. Deshalb wird er als C-Bibliothek umgesetzt, die einfach in bestehende Webserver eingebunden werden kann.

Abbildung 4.6 zeigt die logischen Abhängigkeiten zwischen den Komponenten auf dem Server. Die untere Schicht dient zur Abstraktion der plattformspezifischen Aspekte der Zielplattform und steuert den Kontrollfluss des Services. Die obere Schicht enthält die ModVis Domainlogik, einige Hilfskomponenten zur Verarbeitung der Requests sowie die Request Handler für die Implementation der Web-API.

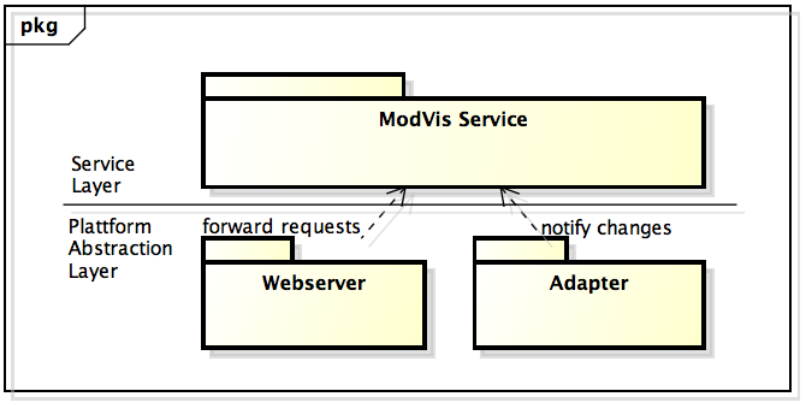


Abbildung 4.6.: Logische Architektur des Webservice

4.5.1. Einschränkungen zugunsten des plattformübergreifenden Einsatzes

Da der ModVis Webservice in unterschiedlichste Webserver sowohl auf Desktop-Betriebssystemen wie auch auf Echtzeitumgebungen integrierbar sein muss, unterliegt dessen Architektur einigen grundlegenden Einschränkungen. Die folgenden Punkte liegen unter anderem den während der Umsetzung eines Prototyps für die *AR* Laufzeitumgebung gewonnen Erfahrungen zugrunde.

Dynamische Speicherverwaltung

Die dynamische Speicherverwaltung führt auf Echtzeitsystemen oft zu schwerwiegenden Problemen. Insbesondere das allozieren von Speicher auf dem Heap über die C-Funktion `malloc()` ist problematisch, da die entsprechenden Operationen oft keine deterministisch begrenzte Laufzeit aufweisen [Wal10]. Häufig ist `malloc()` auch nicht oder nur durch andere Funktionen umgesetzt. In der *AR* Laufzeitumgebung müssen Speicherbereiche z.B. über `TMP_alloc()` oder `MEM_alloc()` alloziert werden.

Aus diesem Grund soll bei der Implementation des ModVis Webservice vollständig auf die dynamische Allokation von Speicher auf dem Heap verzichtet werden.

Auslagern von plattformspezifischen Funktionalitäten

Folgende Funktionalitäten sind zwar durch den C-Standard zum Teil berücksichtigt [ISO99], werden jedoch je nach Zielplattform unterschiedlich umgesetzt.

- Interprozesskommunikation
- Nebenläufigkeit und Prozesssynchronisation
- Zugriff auf Datum und Uhrzeit

Diese Aspekte müssen deshalb beim Einbinden des Services durch den Benutzer umgesetzt werden.

Allgemeine Schnittstelle zum Webserver

Viele Webserver bieten eine *CGI* ähnliche Schnittstelle an, damit Requests an zusätzliche ausführbare Dateien delegiert werden können. Andere Webserver müssen zusammen mit den eingebundenen Services kompiliert werden oder werden über Systemfunktionen angesteuert. Damit der ModVis Service einfach integriert werden kann, soll er über eine möglichst einfache Schnittstelle zum Server verfügen.

4.5.2. Die C-Schnittstelle des Webservice

Der Webservice ist eine C-Bibliothek, die Funktionen zur Verfügung stellt, um Manipulationen auf der ModVis Datenstruktur durchzuführen oder um Teile der Datenstruktur anhand eines HTTP Requests in einen String zu realisieren. Die Bibliothek selber verwaltet keinen eigenen Zustand und greift auf keine nicht deterministischen Input- oder Outputfunktionen zu. Dies bedeutet, dass jeder Aufruf einer Funktion des Services mit den gleichen Parametern (und im Falle von Referenzparametern den gleichen referenzierten Datenstrukturen) garantiert immer den identischen Rückgabewert liefert, beziehungsweise die gleichen Operationen auf den durch Parametern referenzierten Datenstrukturen durchführt. Diese Garantie besteht immer unter der Voraussetzung, dass genügend Arbeitsspeicher zur Verfügung steht.

Die Verwaltung des Systemzustands sowie der Zugriff auf I/O-Operationen muss somit durch den Benutzer des Services implementiert werden. Dies bedeutet zwar einen leicht erhöhten Aufwand bei der Integration auf einer neuen Plattform, anderseits werden dadurch die vielseitigen, in Abschnitt 4.2 beschriebenen, Deploymentvarianten ermöglicht.

Die `ModVis.h` Headerdatei stellt die öffentliche Schnittstelle zum ModVis Webservice bereit und deklariert alle benötigten Datenstrukturen und Funktionen.

Die folgenden beiden Funktionen werden durch die Adapterkomponente (siehe Abbildung 4.1) aufgerufen und fügen Zustandsänderungen in die ModVis Datenstruktur ein:

`mvSetElementStates`

Aktualisiert in der referenzierten Systembeschreibung die Darstellung von einem oder mehreren Elementen

Parameter:

`systemDescription` Referenz auf die zu aktualisierende Systembeschreibung vom Typ `MvSystem`

`elementCount` Anzahl der zu aktualisierenden Elemente

`elementIds` Array der Grösse `elementCount` mit den UUIDs aller zu aktualisierenden Element

`appearances` Array der Grösse `elementCount` mit Integern, welche die Darstellung der entsprechenden Elemente repräsentieren

`sequenceNo` Fortlaufende Nummer, die eine zeitliche Einordnung der Zustandsänderungen ermöglicht; muss beim nachfolgenden Aufruf mindestens um eins erhöht werden

`time` Aktuelle Serverzeit in Sekunden; kann auch 0 sein; die Zeit wird ausschliesslich zur Information des Benutzers benötigt

`mvSetAllElementStatesOfDiagram`

Aktualisiert in der referenzierten Systembeschreibung die Darstellung von allen Elementen eines Diagramms

Parameter:

`systemDescription` Referenz auf die zu aktualisierende Systembeschreibung vom Typ `MvSystem`

`diagramId` UUID des betroffenen Diagramms

`appearance` Integer, der die neue Darstellung aller Elemente des Diagramms repräsentiert

`sequenceNo` Fortlaufende Nummer, die eine zeitliche Einordnung der Zustandsänderungen ermöglicht; muss beim nachfolgenden Aufruf mindestens um eins erhöht werden

`time` Aktuelle Serverzeit in Sekunden; kann auch 0 sein; die Zeit wird ausschliesslich zur Information des Benutzers benötigt

Der andere Teil der Schnittstelle bilden die Funktionen, die vom Webserver die Requests entgegennehmen und diese anhand der ModVis Datenstruktur beantworten:

`mvHandleRequest`

Bearbeitet einen HTTP Request entsprechend der Web-API Definition mithilfe der referenzierten Systembeschreibung und schreibt die Response in den String `responseDestination`.

Gibt den entsprechenden HTTP Statuscode zurück (200 im Erfolgsfall) [Fie+99].

Parameter:

systemDescription Referenz auf die zu aktualisierende Systembeschreibung vom Typ `MvSystem`

responseDestination Referenz auf ein Char Array in das die Response geschrieben werden soll

responseLength Maximale länge der Response; Grösse der `responseDestination`

requestMethod Die HTTP Methode des Requests z.B. `GET`

requestUrl Die Request URL; kann auch nur den Pfad der URL enthalten z.B. `http://www.example.com/modVis/snapshot?diagram=12...` oder `/modVis/snapshot?diagram=12...`

requestBody Der Request Body; wird in der momentanen Version der ModVis Web-API nicht benötigt, kann aber für zukünftige Erweiterungen notwendig werden falls z.B. `POST` Requests bearbeitet werden müssen

mvTranslateStatusCode

Hilfsfunktion für Server, die die Integerrepräsentation von HTTP Status Codes nicht interpretieren können; schreibt die entsprechende Textrepräsentation in einen String z.B. `200 OK`

Parameter:

dst Char Array in das der Status Code geschrieben werden soll

dstLength Maximal zu schreibende Länge

code Der umzuwandelnde Status Code

4.6. Adapter zur Entkopplung von Steuerungslogik und Plattform

Damit in der generierten Steuerungslogik die Aufrufe zum ModVis Service keine Aspekte der Nebenläufigkeit und *IPC* behandelt werden müssen, ruft der generierte Code lediglich die entsprechenden Funktionen auf der Adapterschnittstelle auf. Entsprechend der umgesetzten Deploymentvariante (siehe Abschnitt 4.2) leitet diese die Funktionsaufrufe anschliessend an den ModVis Webservice weiter. Eine Implementation des Adapters kann zudem für jedes Projekt auf derselben Plattform wiederverwendet werden.

```
1  #include <time.h>
2  #include "modVis.h" // ModVis service
3  #include "modVisAdapter.h" // The adapter interface
4  #include "modVisStandalone.h" // Integration to the mongoose webserver, contains
       declaration of refSystemDescription
5
6  static MvTime sequenceNo = 0;
7
8  void mvaSetElementStates(unsigned elementCount, MvUuid elementIds[],
9                unsigned appearances[]){
10        // forward function call including pointer to system description, next sequenceNo
11        // and current time
12        mvSetElementStates(refSystemDescription, elementCount, elementIds, appearances,
13               ++sequenceNo, (MvTime) time(0));
14 }
15
16 void mvaSetAllElementStatesOfDiagram(MvUuid diagramId, unsigned appearance){
17        // forward function call including pointer to system description, next sequenceNo
```

```
18          // and current time
19          mvSetAllElementStatesOfDiagram(refSystemDescription, diagramId, appearance,
20                  ++sequenceNo, (MvTime) time(0));
21  }
```

Listing 4.5: Beispielimplementation des Adapters

Eine sehr einfache Implementation des Adapters für Unix und Windows Plattformen, gemäss dem in Abschnitt 4.2.4 ausgeführten Deploymentmodell, wird in Listing 4.5 vorgeschlagen. Der Adapter kapselt den Systemzustand in den Variablen `sequenceNo` und `refSystemDescription` und übernimmt den Zugriff auf die Systemfunktion `time()`. Jeder Funktionsaufruf wird angereichert mit diesen Informationen an den ModVis Service weitergeleitet. Da in diesem Fall keine Synchronisation zwischen mehreren Prozessen benötigt wird, fällt die Umsetzung des Adapters entsprechend einfach aus.

4.7. Clientapplikation im Browser

Wie im Abschnitt 4.1 Architekturübersicht erläutert wird, wird die eigentliche Darstellungs- und Bedienungslogik vollständig in den Browser verschoben. Wodurch der Server entlastet wird, da dieser kein Rendering von Vorlagen oder Ähnliches vornehmen muss. Der damit verbundene Aufbau des Clients wird nun in diesem Abschnitt näher behandelt, wobei die Schwerpunkte auf den möglichen Client Frameworks, einer kurzen Einführung in AngularJS, der Aufteilung des Clients auf Basis von AngularJS und der Kommunikation zwischen Server und Client gelegt werden.

4.7.1. Unterschiedliche Frameworks zur Schichtenabstrahierung des Clients im Browser

Die Abstrahierung der unterschiedlichen Komponenten nach dem *Model View Controller (MVC)*-Modell auf dem Client ist ein wichtiger Punkt zur Optimierung der *Document Object Model (DOM)*-Manipulationen, sowie der Vereinfachung der Codewartungs- und Erweiterungsarbeiten. Durch den Einsatz eines geeigneten JavaScript Frameworks kann die Unterteilung der unterschiedlichen Schichten und die damit verbundenen Arbeiten effektiver gelöst werden. Dabei gibt es wichtige Eigenschaften, die ein JavaScript *MVC*-Framework für den Einsatz im ModVis Client mit sich bringen sollte:

- **Observables** — Objekte des Models (M in MVC) können auf Änderungen überwacht werden.
- **Routing** — Seitenänderungen können dem Browser per *URI* Fragmentbezeichner (siehe [BFM05] Abschnitt 3.5) bekannt gegeben werden, bzw. auf Änderungen des Bezeichners reagieren.
- **View Bindings** — Änderungen an Objekten lösen in angebundenen View-Elementen automatisch Aktualisierungen aus.
- **Two Way Bindings** — Das View Binding ist auch in die entgegengesetzte Richtung anwendbar. Änderungen in Eingabefelder lösen automatisch die Aktualisierung der angebundenen Objekte aus.
- **Partial Views** — Views können ineinander verschachtelt werden.

Mögliche Framework Alternativen

Die Auswahl der untersuchten Alternativen wurde aufgrund der grossen Auswahl an JavaScript Frameworks und der beschränkten Zeit dieser Arbeit, anhand einer grundlegenden Internet Recherche und mit der Hilfe von [OS13] erstellt. Dabei haben sich die folgenden drei Frameworks als passende Alternativen ergeben.

Backbone.js

Backbone.js ist eine JavaScript-Bibliothek mit einer ausgeprägten Funktionalität für *REST*-Schnittstellen und basiert auf dem *MVC*-Prinzip. Des Weiteren ist Backbone.js für seine geringe Grösse bekannt und nur von der JavaScript-Bibliothek Underscore.js abhängig. Backbone.js wird vorwiegend zur Programmierung von *Single Page Application (SPA)* und zur Synchronität von Webanwendungen (z. B. verschieden Clients und Server) verwendet.

Pro

- OpenSource Lizenz (nach [MIT88])
- weit verbreitetes Framework und verfügt dadurch über eine grosse Community
- kleine Dateigrösse
- hohe Flexibilität bei Erweiterungen des Frameworks, mit wenigen Konventionen und Vorgaben

Kontra

- Gefahr von Memory Leaks (siehe [Kat13])
- erhöhter Aufwand um Applikation zu testen
- hoher Programmieraufwand
- Es werden nur die Eigenschaften Observables und Routing unterstützt

Ember

Ember ist ein JavaScript Framework mit welchem ambitionierte Web Applikationen erstellt werden können. Es stellt eine ausgereifte HTML-Vorlagen Rendering zur Verfügung und beinhaltet weitere Funktionen, welche die Handhabung des Frameworks vereinfachen.

Pro

- OpenSource Lizenz (nach [MIT88])
- Sehr gutes HTML-Vorlagen System
- einfache Konventionen, wodurch die Programmier Produktivität gesteigert werden kann
- Das Framework bringt die zu Beginn genannten Eigenschaften mit

Kontra

- sehr grosse Dateigrösse, da zusätzlich *jQuery* und *Handlebars* benötigt werden
- erhöhter Aufwand um Applikation zu testen

AngularJS

AngularJS ist ein Open-Source-Framework von Google, welches die Erstellung von browserbasierten *SPA* mit einem *MVC*-Modell unterstützt. Weiter erleichtert AngularJS die Entwicklung von grösseren dynamischen Web Applikationen, sowie das Komponententesten entsprechender Anwendungen.

Pro

- OpenSource Lizenz (nach [MIT88])
- Gute Testbarkeit, da das Framework nach dem *Test-Driven Development (TDD)* Ansatz entwickelt wurde
- Fundiertes Vorwissen im Projektteam vorhanden
- Das Framework bringt die zuvor genannten Eigenschaften mit

Kontra

- langsamerer Aufbau von Webseiten (siehe Vergleich [Por13])
- erhöhter Lernaufwand für Anfänger, da Dokumentation sehr spezifisch ist und viel AngularJS "Fachjargon" enthält

Grund für die Verwendung von AngularJS

Da sich AngularJS bei den Vorteilen vor allem durch die gute Testbarkeit und dem Vorwissen des Projektteams gegenüber den anderen Frameworks abhebt, wurde entschieden AngularJS als *MVC*-Framework zu verwenden.

4.7.2. Das JavaScript Framework AngularJS

Das JavasScript Framework AngularJS wird von Google unter der OpenSource-Lizenz nach [MIT88] entwickelt und verbessert. Mittels AngularJS lassen sich nach dem *SPA*-Ansatz dynamische Webapplikationen erstellen. Weiter verfolgt AngularJS das Prinzip der *MVC* Aufteilung, wobei die Views deklarativ beschrieben werden. Dazu wird das normale HTML-Vokabular um einige AngularJS-Attribute erweitert. Kombiniert mit Two-Way-Data-Binding gibt dies eine leicht lesbare und übersichtliche Trennung zwischen Views und Programmcode.

Die einzelnen Bestandteile einer AngularJS-Applikation werden in der Abbildung 4.7 gezeigt. Im AngularJS Umfeld wird eine Applikation auch als Modul identifiziert und kann die in der Abbildung ersichtlichen Komponenten enthalten.

- **Config** — Dieser Block wird beim Laden von Modulen als erstes ausgeführt und kann zur Initialisierung von Einstellungen oder Ähnlichem genutzt werden.
- **Routes** — In diesem Abschnitt werden die erlaubten *URI*s festgelegt. Weiter können die Templates und Controller pro Route definiert werden.
- **Scope** — Der Scope wird zwischen der View und dem Controller als Verbindungsglied verwendet und dient als sogenanntes ViewModel (siehe [Hil09]).
- **Views** — Mittels der Views werden die eigentlichen Benutzeroberflächen beschrieben. Sie werden in normalem HTML geschrieben und können AngularJS spezifische Elemente und Attribute enthalten.

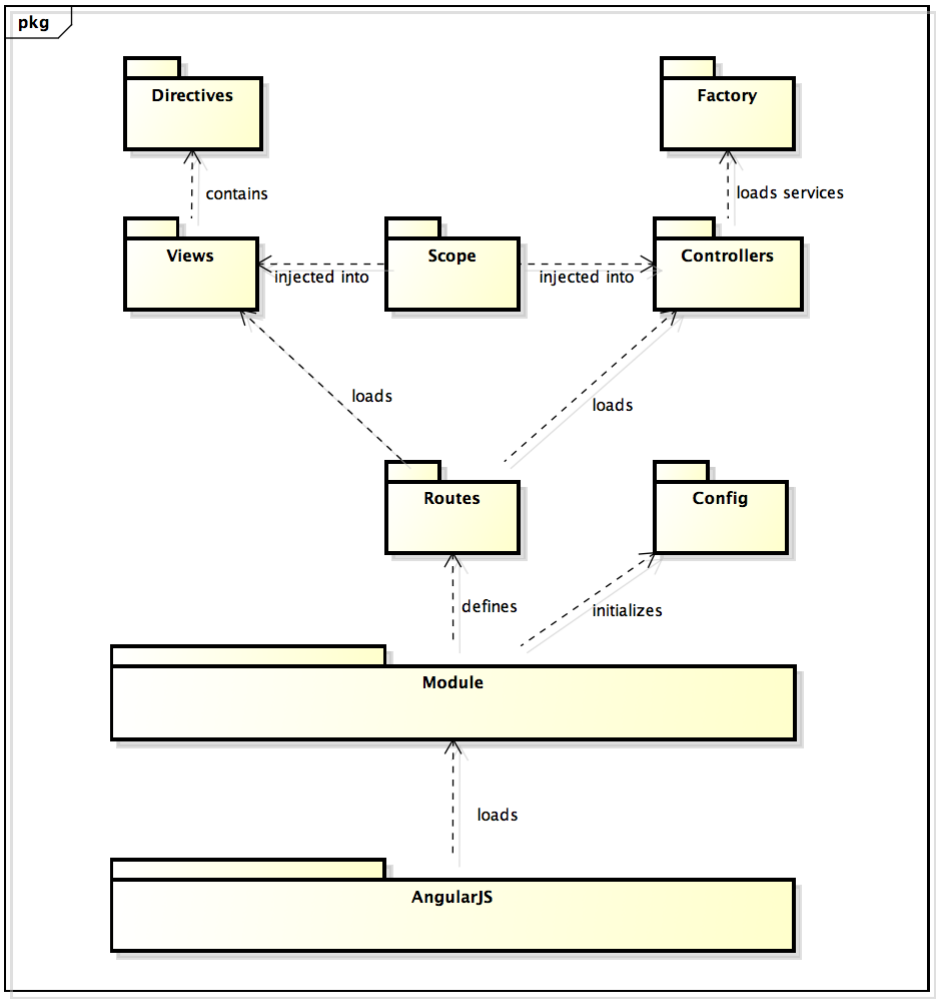


Abbildung 4.7.: Übersicht der AngularJS Komponenten

- **Directives** — Mithilfe von Direktiven können eigene HTML-Elemente als "Komponente" zusammengefasst werden. Hierbei kann den HTML-Elementen neue Funktionalität hinzugefügt oder der damit verbundene *DOM* manipuliert werden.
- **Controllers** — Hier werden die für die Applikation benötigten Controller registriert.
- **Factory** — Die Factory wird verwendet um Service Logik vom Controller zu trennen.

Für eine ausführlichere Dokumentation wird auf die offizielle Website von AngularJS [Tea10] verwiesen.

4.7.3. Aufteilung des Clients auf Basis von AngularJS

Auf Basis des zuvor vorgestellten Aufbaus des AngularJS Frameworks, ist der Client wie folgt in die unterschiedlichen Komponenten unterteilt.

Routes

Der ModVis CLient stellt folgende Routes zur Verfügung:

- `/` — Mit dieser Route wird die Applikation geladen, hierbei werden die Views `diagram-explorer` und `empty` angezeigt, als Controller werden `ErrorCtrl` und `ExplorerCtrl` benötigt, welche im Folgeabschnitt erläutert werden.
- **`/404`** — Die `/404`-Route wird für alle unbekannten Routes verwendet. Sie zeigt die View `error-page`, mit einem Hinweis auf den aus dem HTTP Standard [Fie+99] bekannten Error Code 404, welcher für *Seite nicht gefunden* steht.
- **`/diagram/:diagramId`** — Mittels der Route `/diagram/:diagramId` kann ein Diagram geöffnet werden. Der Parameter `:diagramId` steht als Platzhalter für eine *UUID* eines spezifischen Diagrams und wird innerhalb des Controllers `DiagramCtrl` benötigt.

Views und Directives

Die Benutzeroberfläche ist in die, aus Abbildung 4.8 ersichtlichen, Views aufgeteilt.

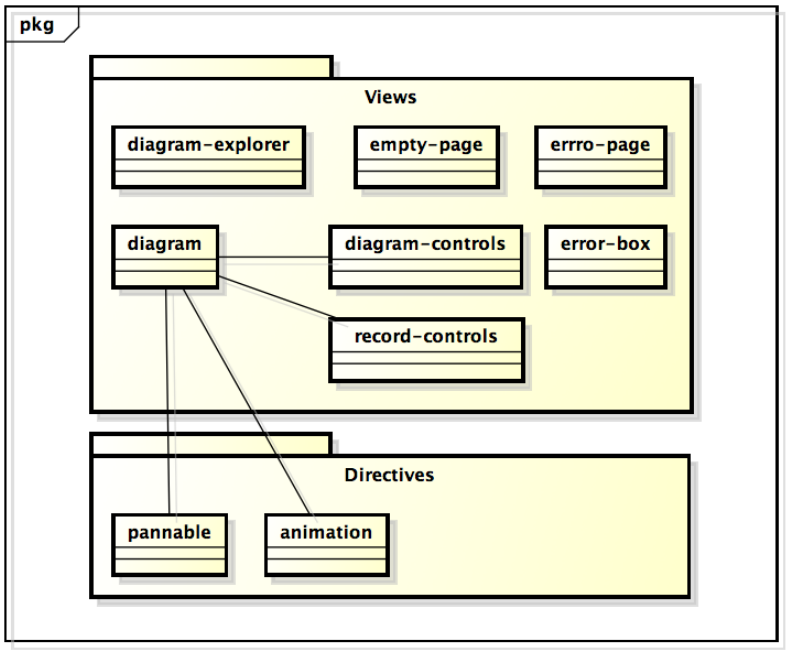


Abbildung 4.8.: Übersicht der ModVis-Client Komponenten: Views und Directives

- **diagram-explorer** — Diese View stellt eine baumartige Struktur der unterschiedlichen Diagramme im System dar.
- **empty-page** — Falls kein Inhalt geladen wurde wird diese View aktiv, sie stellt eine leere Seite dar.
- **error-page** — Die `error-page` wird, wie bereits erwähnt, für nicht gefundene Seiten verwendet.
- **diagram** — Mittels dieser View können Diagramme geladen werden. Sie beinhaltet weiter, die nachfolgenden Teil-Views `diagram-controls` und `record-controls`. Ebenfalls werden die Directiven `pannable` und `animation` darin verwendet.

- **diagram-controls** — Durch die Verwendung der `diagram-controls` werden Bedienungselemente zur Diagramverschiebung bzw. Grössenänderung der Diagramme eingeblendet.
- **record-controls** — Die `record-controls`-View beinhaltet Kontrollelemente die benötigt werden, um eine Aufnahme von Diagrammänderungen zu durchlaufen.
- **error-box** — Über die `error-box` können Fehlermeldungen ausgegeben werden.

Des weiteren werden in der View `diagram` folgende Directives eingesetzt:

- **pannable** — Diese Directive ergänzt die bereits beschriebene `diagram`-View um die zuvor erwähnten `diagram-controls` und deren Funktionalität ein Diagramm zu verschieben und zu vergrössern bzw. verkleinern.
- **animation** — Per `animation`-Directive wird ein Diagramm um die Eigenschaft erweitert, eine Aufnahme von Zustandsänderungen zu übernehmen und auf die Diagrammelemente anzuwenden.

Die genannten Directives können jedoch auch separat und somit unabhängig der `diagram`-View verwendet werden.

Controller und Factory

ModVis verwendet die in der Abbildung 4.9 definierten Controller und die durch die Factory verfügbaren Services.

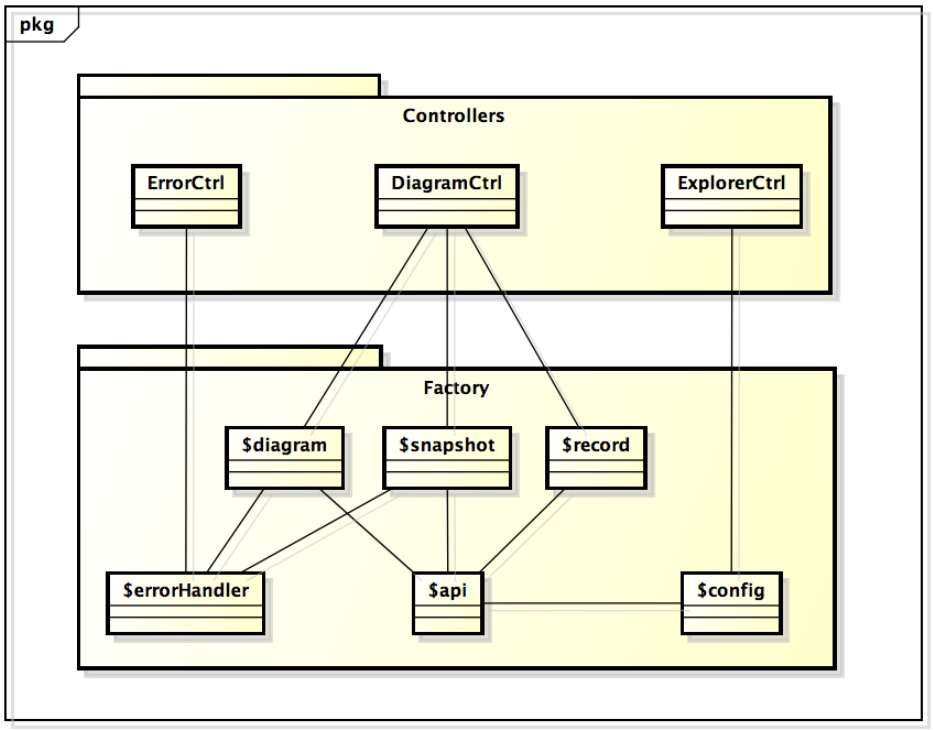


Abbildung 4.9.: Übersicht der ModVis-Client Komponenten: Controllers und Factory

- **ErrorCtrl** — Dieser Controller steuert das Verhalten der `error-box`-View, beispielsweise bestimmt er wann die View eingeblendet wird und welchen Inhalt sie hat.
- **ExplorerCtrl** — Der `ExplorerCtrl` stellt den Diagrammbaum des Systems zur Verfügung und erlaubt es Diagramme in der `diagram`-View zu öffnen.
- **DiagramCtrl** — Der `DiagramCtrl` kontrolliert das abfragen von Diagrammänderungen und stellt die für die `record-controls` benötigten Funktionen bereit.

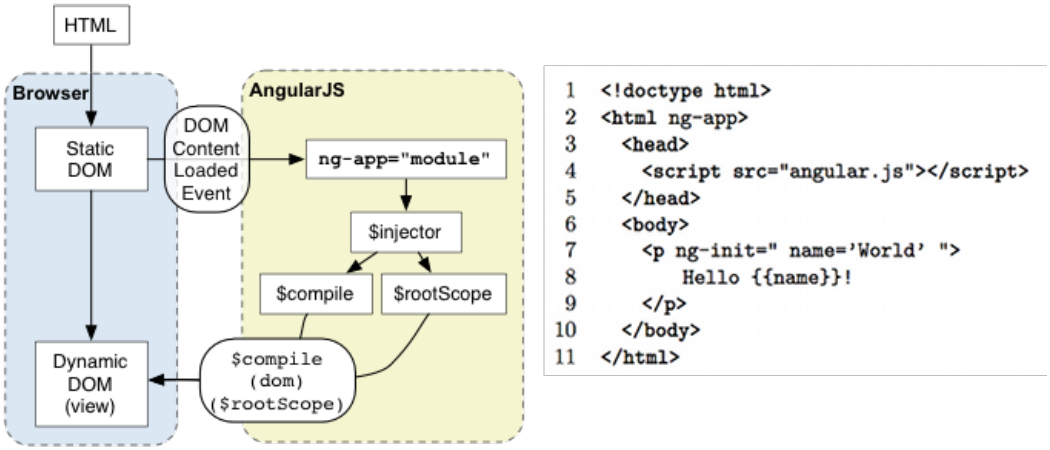
Die beschriebenen Controller bauen auf den folgenden, von der Factory bereitgestellten Services auf:

- **$api** — Der `$api`-Service ist zuständig für grundlegende Funktionen, wie Daten von einer *URI* abfragen.
- **$errorHandler** — Dieser Service stellt einen Mechanismus zur Registrierung von Fehlerbehandlung zur Verfügung.
- **$config** — Mithilfe des `$config`-Service kann unteranderem der Diagrammbaum des Systems beim Server abgeholt werden, wozu er den `$api`-Service verwendet.
- **$diagram** — Der `$diagram`-Service ermöglicht es Diagramme vom Server herunterzuladen, dazu verwendet er ebenfalls den `$api`-Service. Des Weiteren können über diesen Service, Informationen wie beinhaltete Elemente, Titel oder Animationsstratgie eines Diagramms abgerufen werden.
- **$snapshot** — Mittels dieses Services können einzelne Aufnahmen von Zustandsänderungen beim Server abgefragt werden, hierzu wird wiederum der `$api`-Service benötigt. Darüber hinaus beinhaltet dieser Service einen automatisierten Abfragezyklus für die fortlaufende Abfrage von `snapshots`, wobei dieser Zyklus mithilfe einer eigenen Funktion gestartet wird.
- **$record** — Dank des `$record`-Serivce kann eine Abfolge von Zustandsänderungen vom Server über den `$api`-Service heruntergeladen werden. Zugleich kann mit diesem Service die Abfolge von Änderungen in einzelne zusammengehörende Pakete aufgeteilt werden.

4.7.4. Startroutine und Datenaustausch zwischen Client und Server

Aus der Abbildung 4.10 ist ersichtlich, das der Browser zu Beginn das HTML und angular.js herunterlädt. Nachdem der Browser das HTML geparst und den statischen *DOM* erstellt hat, löst er einen DOMContentLoaded Event aus. Aufgrund dieses Events durchsucht Angular den *DOM* nach der `ng-app` Directive, welche eine Angular-Applikation auszeichnet. Falls ein Module in der `ng-app` definiert wurde, wird dieses zur Konfiguration des `$injector` verwendet. Der `$injector` wiederum instanziert den `$compile`-Service und `$rootScope`. Mit dem nun vorhanden `$compile`-Service wird der *DOM* durch Angular kompiliert und mit dem `$rootScope` verknüpft. Weiter wird mittels Directive `ng-init` (siehe Code in Abbildung 4.10) dem Property `name` der Wert `World` zu gewiesen, wodurch {{name}} durch `World` ersetzt wird. Nachdem die Startroutine durchgeführt wurde, kann der Benutzer mit der Applikation interagieren.

In Abbildung 4.11 wird grob gezeigt, wie sich die Applikation beim öffnen eines Diagrams verhält. Hierbei ist zu beachten, dass sich der Ablauf in zwei Schritte unterteilen




```
1  <!doctype html>
2  <html ng-app>
3    <head>
4      <script src="angular.js"></script>
5    </head>
6    <body>
7      <p ng-init=" name='World' ">
8         Hello {{name}}!
9      </p>
10   </body>
11 </html>
```

Abbildung 4.10.: Startroutine AngularJS von [Tea10]

lässt. In einem ersten Schritt wird vom `DiagramCtrl` die *SVG*-Grafik über den Service `$diagram` beim Server abgeholt. Die Grafik wird dann wiederum über den `$scope` der View bekannt gemacht, welche das Diagramm in der View einbettet. Danach registriert sich der `DiagramCtrl` mit einer Callback-Funktion beim `$snapshot`-Service für das zuvor geladene Diagramm. Danach führt der Service die `startPolling()`-Funktion aus, welche einen Abfragezyklus startet. Dieser Zyklus holt in einem vordefinierten Zeitabstand den jeweils nächsten `snapshot` des Diagramms beim Server ab.

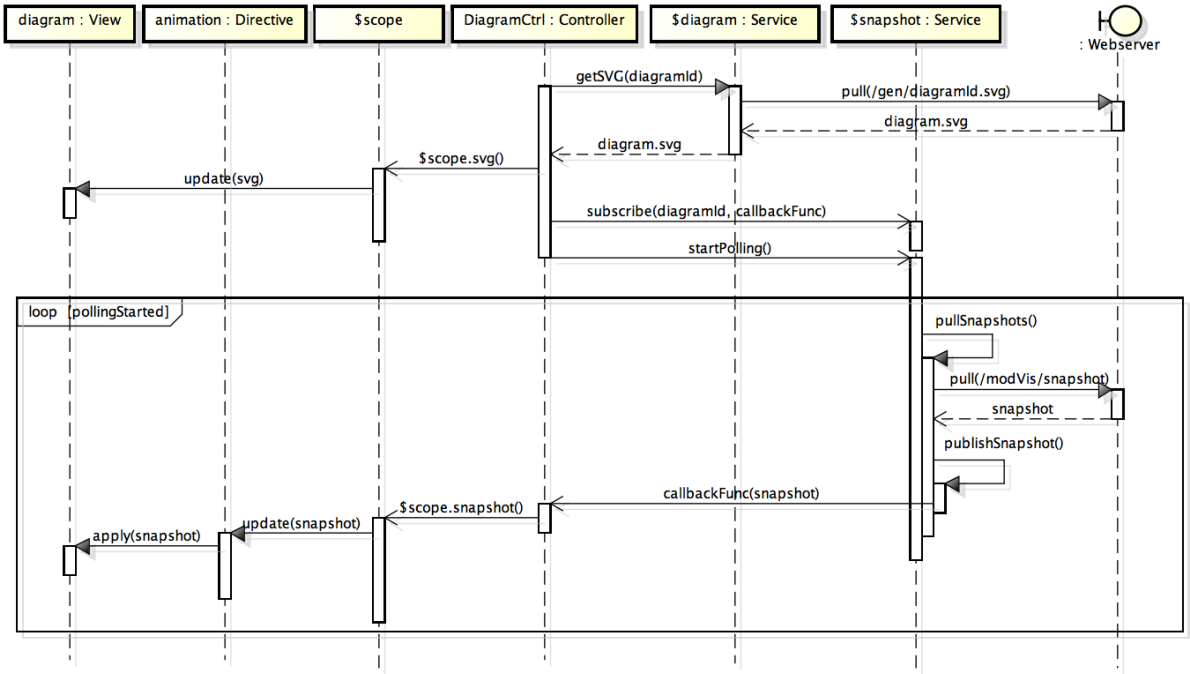


Abbildung 4.11.: Ablaufdiagramm: Öffnen eines Diagramms und Diagrammänderungen abfragen

4.8. Externes Design der Benutzeroberfläche

Bei der Benutzeroberfläche liegt der Fokus auf einer einfachen Struktur der Komponenten. Dementsprechend besteht sie aus zwei Hauptbereichen. Auf der linken Seite befindet sich die Baumstruktur der im System vorhandenen Diagramme, womit durch die Systemhierarchie navigiert werden kann. Im Hauptbereich rechts, werden die Diagramme dargestellt. Dieser Bereich ist wiederum in einen Steuerungsbereich, ähnlich einem Musikspieler, und dem Visualisierungsbereich des Diagramms unterteilt. Weiter besteht eine zweite vereinfachte Ansicht (siehe Abbildung 4.13 rechts), in welcher nur der Diagrammbereich angezeigt wird. Diese zweite Ansicht wird benötigt, damit parallel mehrere Diagramme geöffnet werden können. Die Abbildung 4.12 zeigt die Anordnung der, im Abschnitt 4.7.3 beschriebenen, Views. Abbildung 4.13 stellt dazu das endgültige Layout der Benutzeroberfläche im Detail dar.

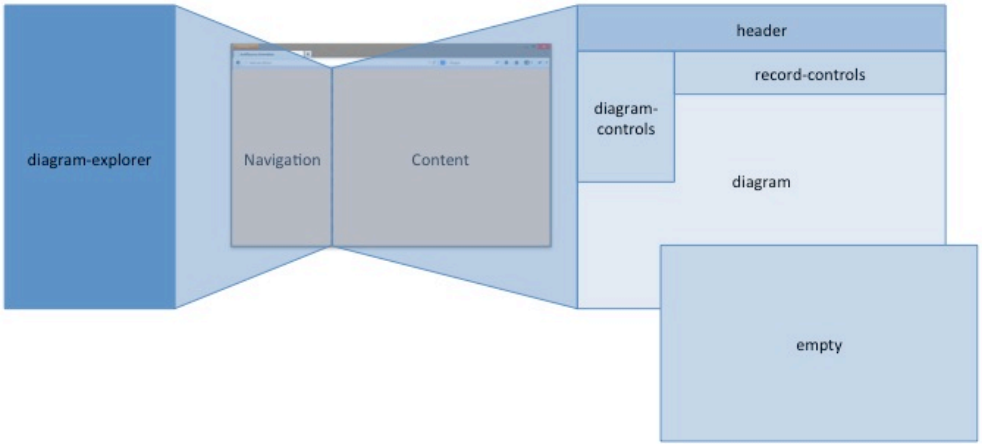


Abbildung 4.12.: Bestandteile der Benutzeroberfläche

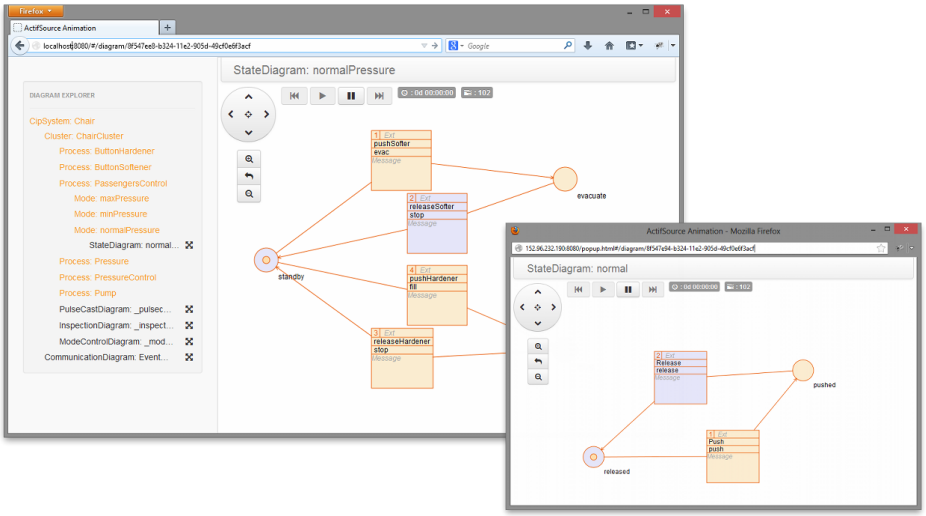


Abbildung 4.13.: Endgültiges Layout der Benutzerinterface

5. Implementation der Komponenten

5.1. Übersicht der Projektstruktur

ModVis besteht aus mehreren Unterprojekten die die verschiedenen Komponenten implementieren oder Beispiele für deren Integration bereitstellen:

ch.actifsource.solution.modVis
: Die ModVis Code Templates, die Definition der Konfigurationsklasse sowie die von actifsource benötigten Build Konfigurationen

ch.actifsource.solution.modVis.cip.doorSystem
: Beispiel für die Integration in ein CIP Projekt; das Projekt implementiert die Steuerung eines Tors

  asrc
  : Das Modell der Torsteuerung

  brProject
  : Beispiel für das Deployment der Steuerung auf einer SPS in Form eines zum Teil generierten Projektes

  standalone
  : Beispiel für das Deployment der Steuerung auf einem lokal ausgeführten Webserver

ch.actifsource.solution.modVis.performance
: actifsource Projekt für die Generierung der Performancetests

modVisLib
: C-Bibliotheke; Implementation des ModVis Webservice

modVisLibTest
: Unit Tests des ModVis Webservice

modVisStandaloneLib
: C-Bibliotheke; Anbindung an den eigenständig ausführbaren *Mongoose* Webserver

web
: Implementation des ModVis Clients

Die folgenden Abschnitte beschreiben die Umsetzung der zentralen Komponenten ModVis Code Templates, ModVis Service und ModVis Client sowie die Anbindung an den lokal ausführbaren Webserver. Das Integrationsprojekt wird im Abschnitt 6.2 Integration der Lösung in ein Steuerungsprojekt weiter ausgeführt.

5.2. Implementation des Webservice

Die Bibliothek `/modVisLib` stellt eine Sammlung von Funktionen und Strukturen zur Verfügung um den ModVis Webservice auf unterschiedlichen Webserveren implementieren zu können.

Alle Komponente von `/modVisLib` sind in C implementiert und basieren auf dem *C99* Standard.

5.2.1. Designgrundsätze

Da der ModVis Webservice hohe Anforderungen an Stabilität, Robustheit und Portabilität erfüllen muss, sollen einige Grundsätze bei dessen Implementation eingehalten werden. Dadurch soll auch die geforderte Qualität sichergestellt werden.

Zustandslose Schnittstelle

Die Verwaltung des Zustands des Webservices wird an die Plattformintegration ausgelagert (siehe Kapitel 4). Dies bedeutet, das die Webservice Bibliothek über keinen eigenen internen Zustand verfügen darf und jeder identische Funktionsaufruf auch das selbe Resultat liefern muss. Zugriffe auf I/O-Operationen des Betriebssystems sind dadurch ebenfalls nicht erlaubt. Diese Einschränkung führt zum einen zu einer höheren Flexibilität bei der Deploymentkonfiguration und zum anderen auch zu einer erhöhten Testbarkeit des Webservices.

Speicherverwaltung

Der Service benötigt keine dynamische Allokation von Speicher auf dem Heap. Dadurch werden einige möglichen Fehlerquellen reduziert und die Plattformunabhängigkeit erhöht.

Abhängigkeiten zu externen Bibliotheken

Die Abhängigkeiten zu anderen Bibliotheken sollen möglichst gering gehalten werden. Dies betrifft auch die Standard C-Bibliothek. Folgende Funktionen aus der Standardbibliothek sollen aus Kompatibilitätsgründen nicht verwendet werden:

- Alle I/O Operationen, unter anderem auch Zugriffe auf die Systemzeit
- Funktionen zur dynamischen Speicherallokation
- Die Funktionen der `printf()` und `scanf()` Familie, da diese unter anderem auf der *AR* Laufzeitumgebung nicht verfügbar sind

Zudem ist bei den Funktionen für die Manipulation von Strings immer die gegen Buffer Overflows abgesicherte Variante (z.B. `strncpy()` statt `strcpy()`) zu verwenden.

Macros

C-Preprocessor-Macros sollen zugunsten der Verständlichkeit des Codes sehr vorsichtig eingesetzt werden. Ausnahmen sind unter anderem:

- Include Guards
- Kapselung der Traversierungslogik von Datenstrukturen
- Definition von magischen Konstanten

5.2.2. Logische Struktur

Abbildung 5.1 zeigt die logische Übersicht der Komponenten der ModVis Service Implementation.

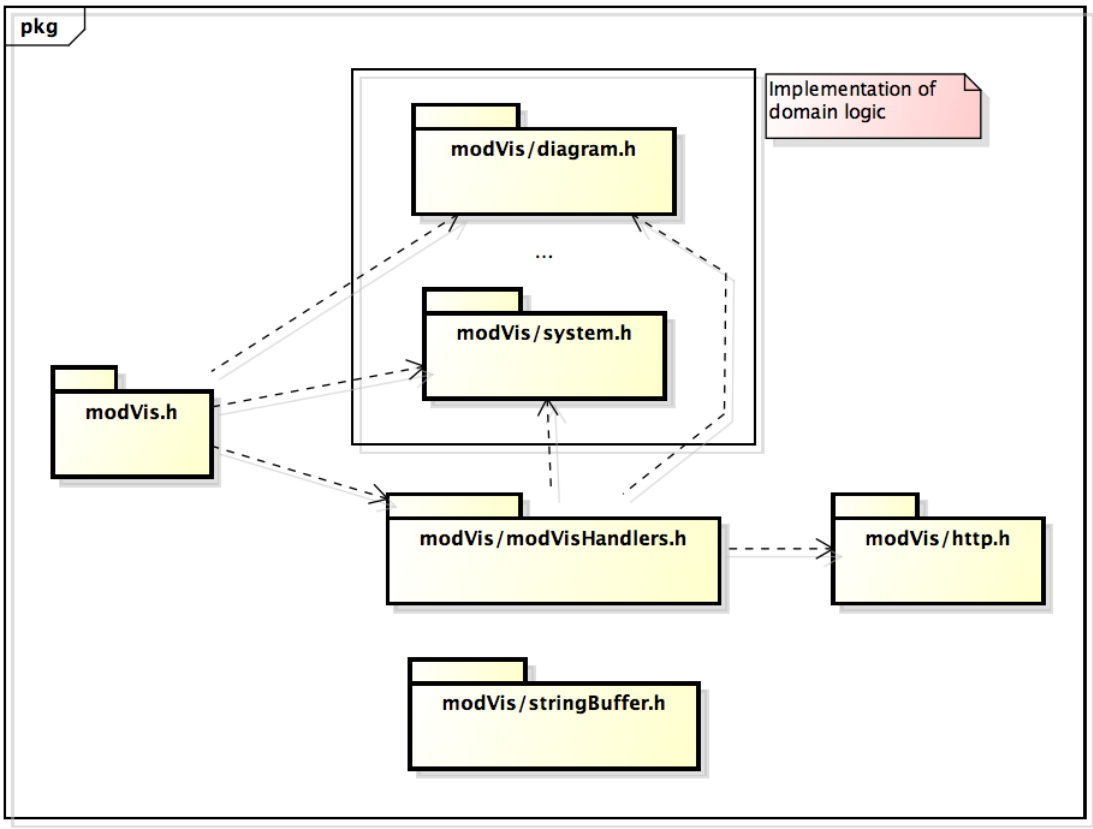


Abbildung 5.1.: Logische Übersicht über die Komponenten der Serviceimplementation und die wichtigsten Abhängigkeiten

Der C-Header `modVis.h` deklariert die öffentliche Schnittstelle des Services sowie die benötigten Datenstrukturen. Die öffentliche C-Schnittstelle wird im Abschnitt 4.5.2 behandelt. Auf die übrigen Teile wird in den folgenden Abschnitten genauer eingegangen.

5.2.3. Identifizierung von Elementen mit UUIDs

Da actifsource alle Ressourcen über *UUIDs* identifiziert, verfügt jedes Diagramm und Diagrammelement über einen global eindeutigen Bezeichner.

Eine UUID ist eine 16 Byte lange Zahl, die in fünf Gruppen unterteilt ist. Im Textformat werden die Gruppen durch Bindestriche getrennt hexadezimal formatiert aneinandergereiht. Somit ist die Textrepräsentation einer UUID 36 Zeichen lang.

UUIDs werden oft als eine Struktur von 5 Ganzzahlen implementiert, wozu jedoch Datentypen mit einer zugesicherten Breite erforderlich sind, wobei die grösste Gruppe 48 Bit umfasst. Da aber nicht alle Plattformen die spezifizierten Datentypen `uint8_t`, `uint16_t` und `uint32_t` bereitstellen, wird für ModVis eine Datenstruktur mit einem `unsigned char` Array der Grösse 16 verwendet.

`modVis/uuid.h` deklariert Hilfsfunktionen zum `MvUuid` Datentyp, für die Konvertierung zwischen der internen und der formatierten Repräsentation sowie für den lexikalischen

Vergleich zwischen zwei UUIDs.

5.2.4. Serverzeit und Sequenznummer

Serverzeit und Sequenznummer (logische Zeit) werden in ModVis durch den Datentyp `MvTime` repräsentiert. `MvTime` ist als `unsigned long` definiert und hat somit gemäss [ISO99] eine minimale Reichweite von 0 bis 4'294'967'295. Da die Serverzeit in Sekunden angegeben wird, kann somit eine Zeitspanne von über 136 Jahren abgebildet werden. Die Serverzeit ist zudem nicht an eine Epoche gebunden und wird lediglich als Hilfe für den Benutzer verwendet, um den zeitlichen Abstand zwischen zwei Ereignissen darzustellen.

Die Implemtation des ModVis Service ist zudem robust gegenüber arithmetische Überläufe. Falls bei der Sequenznummer also ein arithmetischer Überlauf auftritt, treten keine kritischen Fehler auf, aber bei der Animation ist keine konsistente Darstellung mehr garantiert.

5.2.5. Umsetzung der Domänenlogik

Die Umsetzung des ModVis Domänenmodells gemäss Kapitel 3 als C-Strukturtypen wird in Abbildung 5.2 dargestellt. Dabei ist anzumerken, dass die gesamte Datenstruktur statisch umgesetzt ist. Alle 1:n Relationen sind dementsprechend durch fixe Arrays umgesetzt.

Im unterschied zum Domänenmodell enthält die Datenstruktur keine Informationen zur Systemhierarchie, da diese Ausschliesslich für die Darstellung der Benutzernavigation auf dem Client verwendet wird.

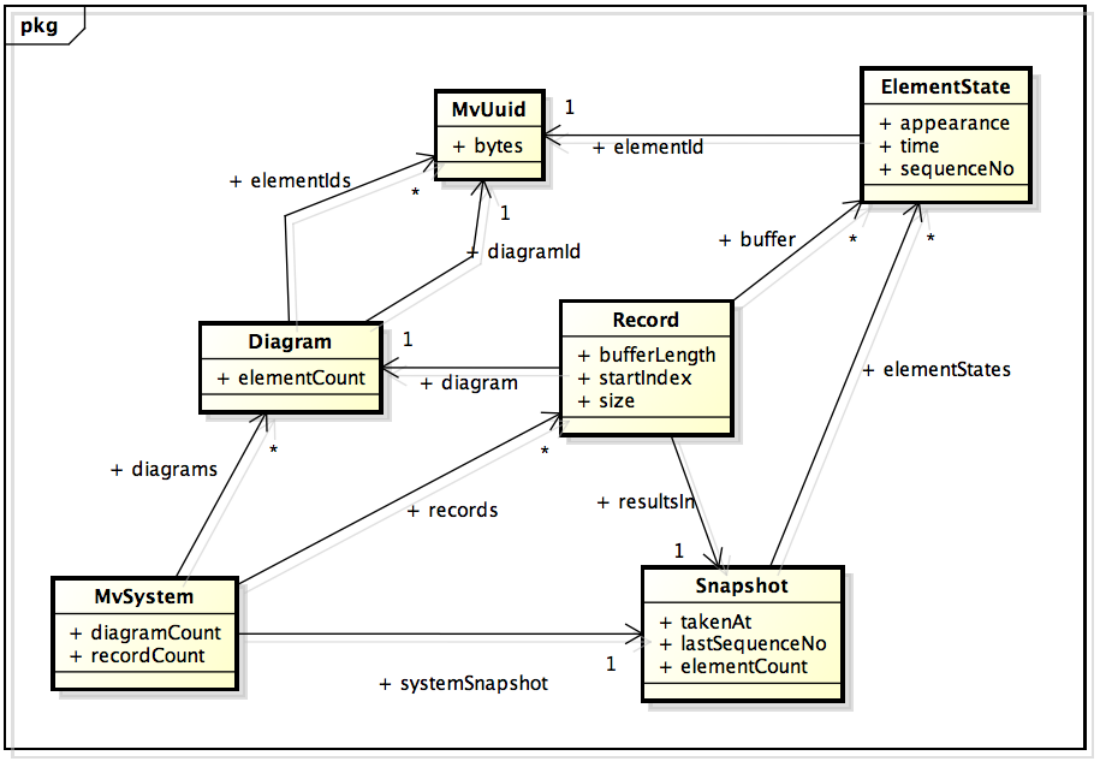


Abbildung 5.2.: Umsetzung der ModVis Domäne als C-Datenstrukturen

MvSystem

`MvSystem` ist das Wurzelelement der gesamten Systembeschreibung und enthält Referenzen auf alle Diagramme, Records und Snapshots des animierten Systems. Listing 5.1 enthält ein Beispiel einer Systembeschreibung eines sehr kleinen Systems. Der Übersicht halber wurden die verwendeten UUIDs vereinfacht, das Literal `{{0x01}}` definiert zwar einen gültigen `MvUuid` Wert, wird so aber in einer generierten Codebeschreibung nicht vorkommen.

```
1  Snapshot systemSnapshot = { 0, 0, 5, (ElementState[5]) {
2          {{{0x01}}}, // byte representation of the UUID 01000000-0000-0000-0000-000000000000
3          {{{0x02}}},
4          {{{0x03}}},
5          {{{0x04}}},
6          {{{0x05}}}
7  }};
8
9  Diagram diagrams[] = { //define 2 diagrams
10         {{{ 0x01, 0xff }}, 3, (MvUuid[3]){{{0x01}}, {{0x02}}, {{0x03}}}},
11         {{{ 0x02, 0xff }}, 2, (MvUuid[2]){{{0x04}}, {{0x05}}}}
12 };
13
14 Record records[] = { // define 2 records with 10 samples capacity
15         {&exampleDiagrams[0], 10, (ElementState[10]){}, 0, 0, &systemSnapshot},
16         {&exampleDiagrams[1], 10, (ElementState[10]){}, 0, 0, &systemSnapshot}
17 };
18
19 // systemDescription holds references to every defined diagram, snapshot and record
20 MvSystem systemDescription = {&systemSnapshot,
21         sizeof diagrams / sizeof(Diagram), diagrams,
22         sizeof records / sizeof(Record), records};
```

Listing 5.1: Beispiel einer Systembeschreibung mit zwei Diagrammen und 5 Elementen

Snapshot

Ein `Snapshot` enthält den gesamten Zustand des Systems zu einem gewissen Zeitpunkt. Da aktuell nur ein Snapshot im System definiert ist, bildet der Snapshot immer den aktuellen Zustand des Systems ab. Das `elementStates` Array enthält zu jedem Element im System genau ein `ElementState` aufsteigend sortiert nach dessen UUID. Bei Aktualisierungen des Systemzustands werden die ElementStates nicht ausgewechselt, sondern es werden bloss deren Attribute aktualisiert.

Diagram

Die `Diagram` Strukturen werden benötigt, um zu überprüfen, welches Element zu welchen Diagrammen gehört. Diese Information wird für das beantworten von Requests und für die Aktualisierung der Records benötigt.

Record

Ein Record enthält die letzten (abhängig von `bufferLength`) ElementStates, die für das Diagramm `diagram` aufgezeichnet wurden. Der Record ist als Ring-Buffer umgesetzt wodurch das Aktualisieren der Inhalte immer in konstanter zeitlicher Komplexität durchgeführt werden kann. Das Attribut `startIndex` markiert den Index des ältesten Eintrags des Records im Array `buffer` und `size` enthält die Anzahl gültiger Einträge. Wobei `size` nicht grösser als `bufferLength` werden kann.

5.2.6. Manipulation von Zeichenketten mit StringBuffer

Da im HTTP-Protokoll die gesamten Protokolldaten als Textfragmente spezifiziert sind und die ModVis Web-API eine Datenserialisierung im *JSON*-Format vorsieht, besteht das Behandeln der HTTP-Requests zu einem grossen Teil aus dem Interpretieren von Textdaten und dem Zusammensetzen der Response. Deshalb wird eine Infrastruktur zur Stringmanipulation benötigt, die die Möglichkeiten der in der C-Standardbibliothek enthaltenen Stringfunktionen erweitern.

Da die bestehenden Open Source Bibliotheken, die derartige Funktionen bereitstellen, auf nicht verfügbare Funktionen wie `sscanf()` oder `malloc()` zurückgreifen, wird für den ModVis Service eine eigene Implementation benötigt.

`StringBuffer` ist eine Datenstruktur, die ein Char Array kapselt und über einen Lesebeziehungsweise Schreibzeiger verfügt. `modVis/stringBuffer.h` stellt Funktionen zu der Datenstruktur bereit.

Der `StringBuffer` kann entweder zum Zusammenfügen oder zum Parsen von Zeichenketten verwendet werden. Alle Schreibfunktionen implementieren zudem eine interne Fehlerbehandlung, wodurch die Fehlerbehandlung im Client Code einfacher ausfallen kann.

5.2.7. Bearbeitung von Requests

`modVis/modVisHandlers.h` deklariert für jede Methode der ModVis Web-API eine Funktion des Typs `RequestHandler`. Ein `RequestHandler` wird aufgerufen, wenn ein empfangener Request über die korrekte HTTP-Methode und den passenden URI Pfad verfügt. Der Request `GET /modVis/snapshot?diagram=123...789` wird somit an die Funktion `getSnapshot()` delegiert.

Jeder `RequestHandler` ist selber für das Parsen des Query Teils der URI zuständig. Dadurch kann die URI vom Request Handler direkt in die benötigte Datenstruktur gelesen werden.

5.2.8. JSON-Serialisierung

Die Serialisierung der Datenstrukturen ins *JSON* Format wird durch einfache Konkatenation von Zeichenketten vorgenommen, wie in Listing 5.2 am Beispiel des `ElementState` Datentyps gezeigt wird.

```
1  MvError elementStateAsJson(const ElementState * es, StringBuffer * sb){
2          stringBufferAppend(sb, "{\"element\":\"");
3          uuidAppendToStringBuffer(sb, es->elementId);
4          stringBufferAppend(sb, "\",\"appearance\":");
5          stringBufferAppendLong(sb, es->appearance, 10);
6          stringBufferAppend(sb, ",\"time\":");
7          stringBufferAppendLong(sb, es->time, 10);
8          stringBufferAppend(sb, ",\"sequenceNo\":");
9          stringBufferAppendLong(sb, es->sequenceNo, 10);
10         return stringBufferAppend(sb, "}");
11 }
```

Listing 5.2: Serialisierung eines ElementStates ins JSON Format

Durch den Verzicht einer Zwischenrepräsentation von JSON-Objekten in einer separaten Datenstruktur und deren anschliessenden Serialisierung in einer einzigen Implementation wird bewusst eine Verletzung des *Don't Repeat Yourself (DRY)*-Prinzips in Kauf genommen. Zum einen ist dies notwendig, da eine solche Zwischenrepräsentation aufgrund der verschachtelten und in der Grösse stark variierenden Struktur nur sehr aufwendig ohne

dynamische Speicherallokation umgesetzt werden kann. Zudem kann die Serialisierung so stark vereinfacht werden und fällt effizienter aus.

5.2.9. Unit Testing mit Google Test

Unit Tests werden mithilfe des C++ Testing Frameworks von Google durchgeführt. Google Test verfügt gegenüber anderen C und C++ Unit Testing Frameworks über den Vorteil, dass Test Suites und Test Cases nicht explizit registriert werden müssen und die Tests sehr deskriptiv geschrieben werden können. Durch den geringeren Mehraufwand um einen neuen Test zu verfassen, sinkt die Hürde für den Entwickler den Code intensiv zu testen.

```
1  // define test for test suite "testStringBuffer"
2  TEST_F(testStringBuffer, appendCharsToBuffer){
3          MvError error = stringBufferAppend(&sb, "123456789");
4          EXPECT_EQ(MV_OK, error);
5          EXPECT_EQ(strlen("123456789"), stringBufferLength(&sb));
6
7          error = stringBufferAppend(&sb, "0");
8          EXPECT_EQ(MV_STRING_BUFFER_EXCEEDED, error);
9          EXPECT_STREQ("123456789", stringBufferFinalize(&sb));
10 }
```

Listing 5.3: Beispiel eines C Unit Tests mit dem Google Test Framework

Die Unit Tests befinden sich im separaten Eclipse Projekt `modVisLibTest` und müssen mit einem C++ Compiler kompiliert werden.

5.2.10. Dokumentation mit Doxygen

Damit der Code der ModVis Service C-Bibliothek einheitlich und durchgängig kommentiert werden kann, wird das Doxygen Format für die Dokumentation der Datentypen und Funktionsdefinitionen verwendet. Die Code Dokumentation kann zudem mit dem Kommandozeilenaufruf `doxygen` im Verzeichnis `modVisLib` automatisch generiert werden. Die Datei `modVisLib/Doxyfile` enthält die benötigten Konfigurationen. Die verwendete Version von Doxygen ist 1.8.3.1.

5.3. Eigenständig ausführbarer Webserver für die lokale Modellsimulation

Das Projekt `modVisStandaloneLib` enthält die Integration des ModVis Webservices in den *Mongoose* HTTP-Server. Diese Integration kann verwendet werden, um den lokal, auf dem PC des Entwicklers ausgeführten C-Code eines Modells mit ModVis zu animieren. Zum Beispiel kann der Webserver mit der CIP Test Suite gestartet werden. Der Betrieb mit der CIP Test Suite wird in den Abschnitten 5.6.4 und 6.2.6 noch näher beschrieben.

5.3.1. Der Mongoose HTTP-Server

Mongoose ist ein leichtgewichtiger HTTP Server der für den Einsatz auf möglichst vielen Plattformen entwickelt wurde. Unter anderem werden Windows, Mac OS X, Unix, Android und iOS unterstützt. Die Mongoose Bibliothek ist unter der MIT Lizenz [MIT88] erhältlich und besteht lediglich aus einer Header- und einer Quellcodedatei. Zudem verfügt Mongoose über eine einfache API um den Server in ein bestehendes Projekt einzubinden.

Mongoose setzt auf das Pre-Threading Pattern für die Requestverarbeitung [Lea00] und startet in einem eigenen Master-Thread [Lyu13]. Somit kann für die ModVis Integration

die Deployment Variante mit geteiltem Speicherbereich gemäss Abschnitt 4.2.4 umgesetzt werden.

5.3.2. Weiterleitung der HTTP-Requests

Die Datei `modVisStandalone.h` deklariert die Funktionen `mvsStart()` und `mvsStop()` für das Starten und Stoppen des Mongoose Servers mit dem integrierten ModVis Webservice. Die Funktion `mvsStart()` benötigt zudem eine Referenz auf die generierte Systembeschreibung des animierten Systems.

`mvsStart()` konfiguriert einen Request Handler Callback für Mongoose und startet den Mongoose Master Thread. Der Aufruf von `mvsStart()` und `mvsStop()` ist somit nicht blockierend. Der bei Mongoose registrierte Callback wird bei jedem empfangenen Request aufgerufen und leitet diesen an den ModVis Webservice weiter. Da die Mongoose API nicht vollständig mit der ModVis C-Schnittstelle übereinstimmt sind zudem noch einige Transformationen durchzuführen:

Wiederzusammenführen der URL Fragmente
Mongoose parst die Request URL und stellt die Bestandteile in der `mg_request_info` Datenstruktur zur Verfügung. Die ModVis C-Schnittstelle benötigt jedoch die Request URL in einem String.

Delegieren der Requestbearbeitung an Mongoose falls Ressource nicht gefunden
Da Mongoose jeden Request dem Request Handler Callback übergibt, auch wenn der URL Pfad eine statische Ressource adressiert, muss der Callback 0 zurückgeben, falls er den Request nicht selber behandeln kann (HTTP-Statuscode 404). Mongoose sucht anschliessend die Ressource im Document Root Verzeichnis und gibt diese zurück oder antwortet ebenfalls mit dem Statuscode 404.

Zusammensetzen der Response
Mongoose überlässt das Konstruieren der gesamten Response, inklusive Response Header, dem Request Handler Callback. Listing 5.4 zeigt wie die Response zusammengefügt wird. Dabei wird der HTTP-Header gemäss der Web-API Spezifikation in Abschnitt 4.4 gesetzt.

```
1  mg_printf(conn, "HTTP/1.1 %s\r\n"
2          "Content-Type: application/json\r\n" // JSON MIME-type
3          "Content-Length: %lu\r\n" // Always set Content-Length
4          "Cache-Control: no-cache\r\n" // Disable caching
5          "\r\n"
6          "%s", status, strlen(content), content);
```

Listing 5.4: Zusammensetzen der HTTP-Response durch den ModVis Standalone Server

5.3.3. Umsetzung des Adapters

`modVisStandaloneLib` stellt neben der Integration in den Mongoose Webserver auch eine Implementation des ModVis Adapters zur Verfügung. Da für den Standalone Betrieb ein Deploymentmodell mit mehreren Threads verwendet wird, fällt die Implementation des Adapters sehr einfach aus (Listing 5.5).

```
1  #include "modVisStandalone.h" // for refSystemDescription
2  #include "modVisAdapter.h"
3  #include <time.h>
4
```

```
5  static MvTime sequenceNo = 0;
6
7  static MvTime nextSequenceNo(){
8          return ++sequenceNo;
9  }
10
11 static MvTime now(){
12         return (MvTime) time(0);
13 }
14
15 void mvaSetElementStates(unsigned elementCount,
16                 MvUuid elementIds[], unsigned appearances[]){
17         mvSetElementStates(refSystemDescription, elementCount, elementIds, appearances,
               nextSequenceNo(), now());
18 }
19
20 void mvaSetAllElementStatesOfDiagram(MvUuid diagramId, unsigned appearance){
21         mvSetAllElementStatesOfDiagram(refSystemDescription, diagramId, appearance,
               nextSequenceNo(), now());
22 }
```

Listing 5.5: Implementation des Adapters für den Standalone Server

Wie in Abschnitt 4.2.4 erläutert wurde, ist trotz des Zugriffs von mehreren Threads (Adapter und Request Handler) keine Zugriffssynchronisation notwendig.

5.4. Implementation der Clientapplikation

Die unterschiedlichen Komponenten des ModVis-Clients wurden bereits im Kapitel 4.7.3 erklärt. Weshalb hier nur die wichtigsten Erkenntnisse der Implementation näher beleuchtet werden. Dazu wird das Augenmerk auf das Multiplexen von Serveranfragen, die Animation der Diagrammelemente und die Grössen- und Positionsanderungen der Diagramme durch den Benutzer gelegt. Abschliessend wird beschrieben mit welchen Testverfahren die Client-Applikation geprüft wurde.

5.4.1. Multiplexen der Serveranfragen

Damit die neuen Zustandsdaten aller geöffneten Diagramme in einem `GET /snapshot` Request angefordert werden können, müssen Diagramme in neuen Fenstern, über die in Listing 5.6 ersichtliche Funktion `window.open()` geöffnet werden.

```
1 $scope.popup = function (diagramId) {
2         // ...
3
4         window.open(url, windowId, windowConfig).focus();
5 }
```

Listing 5.6: Öffnen eines Diagramms in einem neuen Fenster

Dadurch kann auf Basis eines Proxyartigen Mechanismus (siehe Proxy [Gam+95]), auf die vom Hauptfenster instanzierte globale `modVis`-Variable zugegriffen werden, welche somit als Singleton [Gam+95] implementiert ist. Falls es sich um das Hauptfenster selbst handelt, wird ein leeres Objekt erstellt, in welches alle vom Eltern-Fenster instanzierten Serivces und dessen Funktionen gespeichert werden (hierzu Listing 5.7). Dies wird jeweils zu Beginn, nachdem die `service.js`-Datei geladen und geparst wurde, ausgeführt.

```
1 // check if current window has opener and is therefore a child window
2 var modVis = (window.opener) ? window.opener.modVis : {};
```

```
3
4  // each call checks if the service is already instantiated in parent window
5  if (modVis.service) {
6          return modVis.service;
7  }
8
9  // else instantiate new service object
10 modVis.service = {};
11
12 // add new function to the service
13 modVis.service.func = function () { /* ... */ };
```

Listing 5.7: Instanzieren und abrufen der globalen modVis-Variable

Mithilfe der eingeführten globalen Variable, registrieren sich die Kind-Fenster nach dem Observer-Pattern aus [Gam+95] für die gewünschten Diagramme beim Hauptfenster bzw. dessen `$diagram`-Service (dazu Listing 5.8). Damit geprüft werden kann ob die registrierten Fenster noch geöffnet sind, wird der Registrierungsfunktion eine Instanz des aktuellen Fenster (`window`) übergeben.

```
1  $diagram
2          .getSVG(diagramId)
3          .then(function (data) {
4
5                  // ...
6
7                  // register onSnapshot as callback function and save returned handle for
                       unsubscribe
8                  callbackHandle = $snapshot.subscribe(diagramId, onSnapshot, window);
9                  if ( !disableAutoPolling ) $snapshot.startPolling();
10
11                 // ...
12
13         });
```

Listing 5.8: Registrierung für gewünschte Diagramme

Aufgrund der registrierten Fenster können die abzurufenden Diagrammsnapshots ermittelt werden und jeweils durch aufrufen der Funktion `pullSnapshots()` beim Server abgeholt werden. Die genannte Funktion wird, falls dieser Zyklus gestartet wurde, jeweils nach Ablauf eines gesetzten Timers erneut aufgerufen, wodurch die `snapshots` kontinuierlich aktualisiert werden. Die erhaltenen Snapshots werden jeweils über `publishSnapshot()` den registrierten Parteien bekannt gemacht. Der Code dazu kann in Listing 5.9 überblickt werden.

```
1  modVis.snapshot.pullSnapshots = function (sequenceNo) {
2
3          // get all diagrams which have subscribers
4          var diagrams = getDiagrams();
5
6          $api
7                  .pull($settings.snapshotUrl,
8                         {diagram: diagrams, fromSequenceNo: sequenceNo})
9                  .then( function (snapshot) {
10
11                         // send the received snapshot to all subscribers
12                         publishSnapshot(snapshot);
13
14                         // ...
15
```

```
16                      // check if polling is still required (registered callbacks
                            available)
17                      if(polling && diagrams.length){
18                              // set new timeout for next call to pullSnapshots()
19                              $timeout(function () {
20                                      modVis.snapshot.pullSnapshots(sequenceNo);
21                              }, $settings.snapshotPullInterval);
22                      }
23              });
24  }
```

Listing 5.9: Polling Mechanismus für snapshots

Durch den gezeigten Aufbau der Applikation wird ein multiplexen der Anfragen über das Hauptfenster erreicht. Dadurch wird die Anzahl Verbindungen pro Benutzer auf die gewünschte Anzahl reduziert. Des Weiteren werden die Service Funktion nur noch im Hauptfenseter ausgeführt.

5.4.2. Animation von Diagrammelementen

Für die Animation der *SVG* Grafiken wurden unterschiedliche Möglichkeiten wie die Animation durch *Synchronized Multimedia Integration Language (SMIL)* und durch *Cascading Style Sheets (CSS)* und JavaScript analysiert. Daraus hat sich ergeben, dass es sich aufgrund des eingesetzten JavaScript Framework anbietet die Elemente mit Hilfe von *CSS* und JavaScript zu animieren. Da es mit *SMIL* nicht möglich ist auf allen modernen Browsern die Elemente zu animieren, wurde diese Entscheidung weiter bestätigt. Auf Basis dieser Entscheidung wurden *CSS*-Klassen eingeführt, welche das Format der Elemente entsprechend anpassen (siehe Listing 5.10). Durch separate *CSS*-Definitionen wird die Struktur des *SVG* besser von der Formatierung getrennt. Die definierten Klassen werden dabei, nach eintreffen einer Zustandsänderungen, auf die von der Änderung betroffenen Elemente appliziert.

```
1  // The defined formats are applied to the first svg rectangle/ellipse
2  // inside the element which the class is append to
3
4  // format inactive element
5  .appearance0 rect:first-child, .appearance0 ellipse:first-child {
6          fill: rgb(252,237,209); // orange
7          stroke: rgb(234,88,00);
8  }
```

Listing 5.10: Beispiel einer zur Animation verwendete CSS Klassen

Inwiefern die CSS-Klassen auf einen Diagrammtypen angewendet werden, wird mithilfe des Strategy Patterns [Gam+95] gelöst. Dabei wird bei der Generierung der Systemhierarchie jeweils zu jedem Diagramm angegeben welchem Typus dieses angehört. Die Strategien sind über die Funktion `getAnimationStrategy()` im Service `$config` abrufbar.

Somit kann auf dem Client für jedes Diagramm die passende Strategie bestimmt und auf die anfallenden Zustandsänderungen angewendet werden. Dies erleichtert es, neue Strategien einzuführen und erhöht somit die Wartbarkeit des Codes.

```
1  // ... initial work
2
3  // request svg for diagram from server
4  $diagram
5          .getSVG( diagramId )
6          .then( function ( data ) {
7                  // ... apply received data to $scope
```

```
8
9               strategy = $diagram.getStrategy( diagramId );
10
11              // ... start snapshot polling
12      });
13
14  // callback function which gets called after receiving a new snapshot
15  function onSnapshot ( newSnapshot ) {
16
17      $scope.snapshot = strategy.prepareSnapshot( $scope.snapshot, newSnapshot );
18
19  }
```

Listing 5.11: Setzen und aufrufen der Animationsstrategie

Um *CSS*-Klassen einem Element anzuhängen bzw. zu entfernen, stellt AngularJS die Funktionen `addClass()` und `removeClass()` zur Verfügung. Diese sind jedoch nicht auf *SVG*-Elemente anwendbar. Deshalb wird auf die Alternative zurückgegriffen, über die AngularJS Funktion `attr()` die CSS-Klasse des Elements zu setzen. (Listing 5.12)

Das Problem der genannten Funktionen ist, dass sie auf das Property `className` des *DOM*-Elements zugreifen möchten. Dieses wird jedoch im HTML-*DOM* Standard [HHS03] als `DOMString` und im *SVG-DOM* als `SVGAnimatedString` (siehe "Appendix C: IDL Definitions" in [FJ03]) implementiert. Da AngularJS unter anderem zur Manipulation von HTML *DOM*-Elementen entwickelt wurde, wird das Property `className` des *SVG* nicht erkannt.

```
1   // function gets called if $scope.snapshot changes
2   attrs.$observe( 'snapshot', function ( strSnapshot ) {
3
4       // create elementStates array from strSnapshot
5       var elementStates = angular.fromJson(strSnapshot || {});
6
7       angular.forEach( elementStates, function ( es ) {
8           // ... init variables
9           var newClass = 'appearance' + es.appearance;
10
11          // remove current appearance class from element
12          if(currentClass){
13              currentClass = currentClass.replace(/ appearance\d+/g, '');
14          }
15
16          element.attr( 'class', currentClass + ' ' + newClass);
17      });
18  });
```

Listing 5.12: Hinzufügen der CSS-Klassen mithilfe der ModVis Directive

5.4.3. Grössen- und Positionsänderungen der Diagramm Grafiken

Um die Analyse von Diagrammen zu erleichtern, bestehen Funktionen, die es erlauben eine geöffnet Diagramm Grafik zu verschieben, zu vergrössern oder zu verkleinern. Um dies zu ermöglichen wird auf der obersten Ebene des *SVG-DOM*'s des Diagramms ein neues Element mit der ID `viewport` eingefügt. Wobei zusätzlich der momentane *DOM*-Inhalt des *SVG*s in das neue Element verschoben wird. (siehe Listing 5.13)

```
1   // to create a sag element, it is necessary to use svg namespace
2   viewport = document.createElementNS( 'http://www.w3.org/2000/svg', 'g' );
3   viewport.setAttribute( 'id', 'viewport' );
4   angular.element( viewport ).append( svgRootElement.childNodes );
```

```
5  svgRootElement.appendChild( viewport );
```

Listing 5.13: Erstellen des Viewports in der SVG-Grafik

Auf Basis des zuvor eingeführten `viewport`-Elements kann nun die Position und Grösse des Diagramms verändert werden. Hierzu kann die Funktion `startPanning(evt)` (Listing 5.14) aufgerufen werden, welche das Verschieben des Diagramms initiiert. Weshalb diese Funktion auf den Mausklick-Event des Viewport registriert wird, so wird sie sofort nach anklicken des Diagramms ausgeführt. Damit eine *SVG*-Grafik verändert werden kann, wird dessen Transormationsmatrix [FJ03] benötigt, womit die momentane Mauszeigerposition auf eine Position innerhalb der Grafik umgerechnet werden kann. [Mic13]

```
1  function startPanning(evt) {
2          evt.preventDefault();
3
4          panStarted = true;
5
6          // get the inverse of the current transform matrix of the viewport
7          panOriginTransformMatrix = viewport.getCTM().inverse();
8
9          // apply a matrix transformation to the current event point (x-, y-coordinates of
               mouse)
10         panOrigin = getEventPoint(evt).matrixTransform(panOriginTransformMatrix);
11 }
```

Listing 5.14: Diagramm verschieben starten

Wird der Mauszeiger mit geklickter Maustaste verschoben, kann mit der umgerechneten Position jeweils die neue Position der Grafik im `viewport` berechnet und abgebildet werden. Die dazu verwendete Funktion wird in Listing 5.15 abgebildet und wird auf den `onmousemove`-Event registriert.

```
1  function pan(evt) {
2          evt.preventDefault();
3
4          if(panStarted) {
5                  var currentMousePos = getEventPoint(evt).matrixTransform(
                       panOriginTransformMatrix);
6                  var newPosX = currentMousePos.x - panOrigin.x;
7                  var newPosY = currentMousePos.y - panOrigin.y;
8                  var newMatrix = panOriginTransformMatrix.inverse().translate(newPosX,
                       newPosY);
9                  setTransformMatrix(viewport, newMatrix);
10         }
11 }
```

Listing 5.15: Diagramm verschieben

Ähnlich der Transformation der Grafik, kann eine Grafik auch vergrössert und verkleinert werden. Hierzu wird wiederum die momentane Mauszeiger Position als Ausgangspunkt verwendet. An dieser Position skaliert die Grafik je nachdem angegebenen `scale` mehr oder weniger. Mittels `translate(-x, -y)` wird die Grafik jeweils so verschoben, dass sie sich genau in der Mausposition vergrössert bzw. verkleinert. (Listing 5.16) Die Abbildung 5.3 zeigt hierzu ein Beispiel.

```
1  // ...
2
3  var currentMousePos = getEventPoint(evt).matrixTransform(currentTransformMatrix);
4
5  // ...
6
```

```
7  var k = svgRootElement
8          .createSVGMatrix()
9          .translate(currentMousePos.x, currentMousePos.y)
10         .scale(scale)
11         .translate(-currentMousePos.x, -currentMousePos.y);
12
13 setTransformMatrix(viewport, viewport.getCTM().multiply(k));
```

Listing 5.16: Diagramm vergrössern bzw. verkleinern

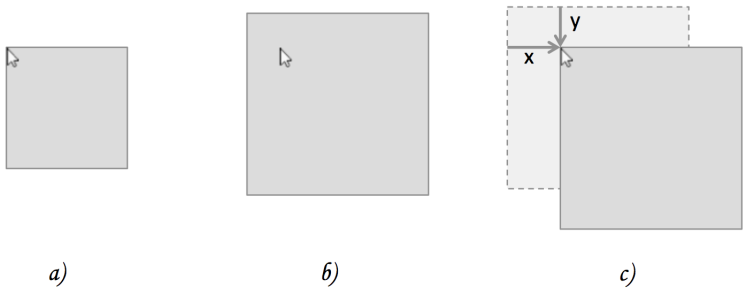


Abbildung 5.3.: a) Grafik in Originalgrösse b) Vergrössern der Grafik ohne Verschiebung c) Vergrössern der Grafik mit Verschiebung

5.4.4. Testen der Clientapplikation mit Jasmine und Karma

Die Clientapplikation wird mit Hilfe des Jasmine Testing Framework [Lab12] und dem Test Runner Karma [Tea13] getestet. Jasmine ist ein *Behaviour-driven Development (BDD)*-Framework, welches speziell für JavaScript entwickelt wurde. Das Framework ist Nachfolger von JsUnit, einer direkten Adaption von JUnit. Es ermöglicht Tests auf eine beschreibende Art zu erstellen, wodurch die Tests aussagekräftig und einfach zu lesen sind, hierzu Listing 5.17. Jasmine verfügt darüber hinaus über unterschiedlichste Testvarianten wie benutzerspezifizierte Matcher, Spione (engl. spies) oder asynchrone Spezifikationen. Die verschiedenen Varianten werden von [Lab12] ausführlich und mit Beispielen beschrieben.

```
1 describe( 'Hello world', function () {
2         it( 'says hello', function () {
3                 expect( helloWorld() ).toEqual( "Hello world!" );
4         });
5 });
```

Listing 5.17: Einfaches Jasmine Test Beispiel

Zum Ausführen der Tests wird wiederum Karma, ehemals Testacular, genutzt. Karma erlaubt es die definierten Tests parallel in mehreren Browsern auszuführen und sammelt dabei die erreichten Ergebnisse ein. Die besagten Tests in echten Browsern auszuführen, bietet einen wesentlichen Vorteil: Es ermöglicht die Nutzung bereits vorhandener Tools, wie z.B. der Debugging-Werkzeuge. Karma startet automatisch die verschiedenen Browser, welche sich über eine einfache Konfigurationsdatei im *JSON*-Format definieren lassen. Des Weiteren kann Karma als eine Art lokales Continuous Integration [Fow06] System eingesetzt werden, wobei es das zu testende Projekt auf Änderungen überwacht und bei allfälligen Veränderungen die konfigurierten Tests ausführt.

Wichtige Konfiguration der Unittests

Wichtig für effektive Tests ist es, dass für jeden Test die Vorraussetzungen unabhängig von anderen Tests sind. Durch die in Abschnitt 5.4.1 eingeführte globale Variable `modVis`, können jedoch ungewollte Seiteneffekte entstehen. Da die Variable jeweils nur einmal erstellt wird und danach für alle Tests verwendet wird, würde sie allfällig geänderte Properties beibehalten. Dies hätte wiederum zur Folge, dass der Zustand des Objekts nicht zurückgesetzt wird, wodurch die einzelnen Tests nicht mehr unabhängig voneinander sind.

Deshalb wird diese globale Variable jeweils zu Beginn einer Jasmine Testsuite neu initialisiert, wodurch mögliche vorhandene Zustände gelöscht werden und für jeden Test eigenständig ist.

```
1 beforeEach( inject( function ( ... ) {
2       // ...
3
4       if(modVis) modVis = {};
5
6       // ...
7 }
```

Listing 5.18: Ausschalten von Seiteneffekten der globalen Variable

Testen mittels End-to-End Verfahren

Da eine Webapplikation oftmals mit dem Server kommuniziert, ist es erforderlich eine Applikation End-to-End zu überprüfen. Dies bedeutet die Applikation von einem dafür vorgesehenen Webserver herunterzuladen, um sie danach mit simulierten Mausklicks zu testen. Wodurch das Verhalten der Tests in unterschiedlichen Browser getestet werden kann und eine Art durchgängiger Integrationstest vom Server bis zum Client stattfindet. Im Listing 5.19 wird ein solcher Test gezeigt.

```
1  beforeEach(function() {
2        browser().navigateTo('/index.html?disableAutoPolling');
3  });
4
5  it('should activate expand checkbox', function() {
6
7        element('label[for^=expand]').click();
8
9        expect( element('input[id^=expand]').attr('checked') ).toBeTruthy();
10 });
```

Listing 5.19: Beispiel wines End-to-End Tests

Zu beachten ist das bei End-to-End Tests jeweils alle Anfragen an einen Server beendet werden müssen, bevor ein neuer Test gestartet wird. (siehe [Sno13]) Aufgrund dessen wird bei jedem Aufruf der Parameter `disableAutoPolling` mitgegeben, welcher den in 5.4.1 beschriebenen Abfrage Zyklus unterbindet.

5.5. Automatische Codegenerierung

Damit ein mit Actifsource entwickeltes System durch ModVis animiert werden kann, müssen die Datenstrukturen für Snapshots und Records aus dem Modell generiert und dem ModVis Webservice zur Verfügung gestellt werden (siehe Abschnitt 5.2.5). Zusätzlich benötigt die Benutzeroberfläche Informationen zur Struktur der definierten Ressourcen um das Navigationsmenü darzustellen sowie die SVG Repräsentationen aller Diagramme.

Alle für die Codegenerierung benötigten Ressourcen befinden sich im actifsource-Projekt `ch.actifsource.solution.modVis` und werden in den folgenden Abschnitten näher erläutert.

5.5.1. Konfigurationsmodell

Damit die Codegenerierung durchgeführt werden kann, wird eine actifsource-Ressource vom Typ `ch.actifsource.solution.modVis.configuration.VisualizedSystemConfiguration` benötigt. Das Modell in Abbildung 5.4 beschreibt alle für die Konfiguration benötigten Klassen.

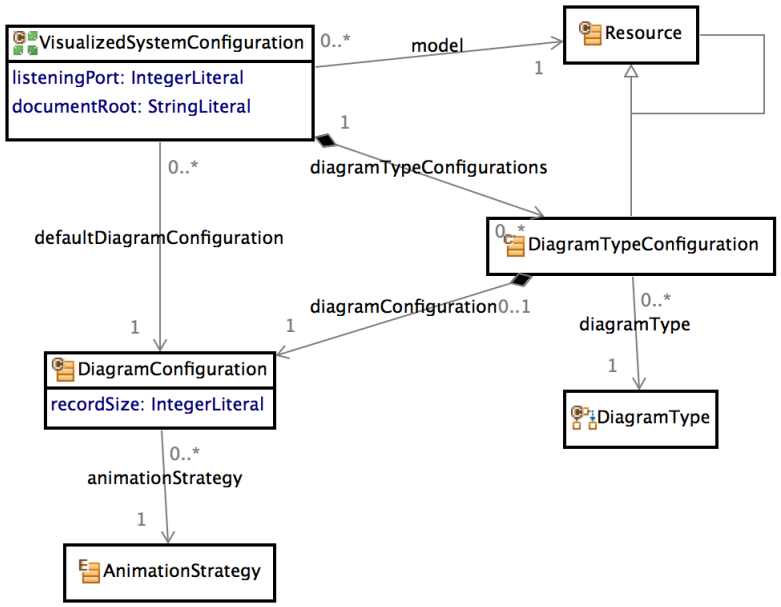


Abbildung 5.4.: Übersicht über das Konfigurationsmodell für die Codegenerierung in actifsource

`VisualizedSystemConfiguration` stellt die Wurzel der Konfiguration dar und enthält im Attribut `model` eine Referenz auf das zu animierende Modell. `model` muss vom Typ `ch.actifsource.core.Ressource` sein. Falls zum Beispiel ein *CIP* Modell animiert werden soll, muss `model` auf eine Instanz von `ch.actifsource.solution.cip.generic.core.meta.system.CipSystem` referenzieren.

Die weiteren Attribute `listeningPort` und `documentRoot` können für die Konfiguration des eingesetzten Webservers verwendet werden. Die Konfiguration erlaubt es zudem, für unterschiedliche Diagrammtypen unterschiedliche Konfigurationen zu verwenden. Die Standardeinstellung für alle Diagramme in `default DiagramConfiguration` kann durch zusätzliche diagrammtypspezifische Konfigurationen in `diagramTypeConfigurations` überschrieben werden.

`DiagramConfiguration` ermöglicht durch das `recordSize` Attribut die Konfiguration der Recordgrösse. Eine `recordSize` kleiner 1 deaktiviert die Aufnahme der Records für alle Diagramme des entsprechenden Typs. Zudem können die Animationsstrategien gemäss Abschnitt 3.5 festgelegt werden.

`AnimationStrategy` ist als Enum Typ umgesetzt und spezifiziert die folgenden Werte:

None Das Diagramm wird durch ModVis nicht animiert. Alle betroffenen Diagramme und

deren Element werden somit nicht in die Systembeschreibung für den ModVis Service generiert. Die SVGs der Diagramme stehen aber trotzdem dem ModVis Client zur Verfügung und werden von diesem als statische Grafiken dargestellt.

Default Die Steuerungslogik steuert alle Darstellungsänderungen des Diagramms.

ImplicitDiagramReset Die Steuerungslogik kommuniziert nur, welche Elemente miteinander eine bestimmte Darstellung einnehmen. Alle nicht erwähnten Elemente des selben Diagramms werden implizit auf die Darstellungsklasse 0 zurückgesetzt.

5.5.2. Code Templates

Die Code Templates beschreiben die Übersetzung von den actifsource Ressourcen in die benötigten Quellcodedateien. ModVis definiert die folgenden Templates:

DomainDiagramSVG Generiert die vom ModVis Client benötigten Diagramme im SVG-Format. Das Template enthält lediglich die erforderlichen XML Deklarationen und ruft die actifsource Templatefunktion `graphAsSvg` für das Generieren der SVG-Elemente auf.

ExplorerNavigation Generiert die vom ModVis Client benötigte Beschreibung der Systemhierarchie im JSON-Format in die Datei `explorer-data.json`. Das Template enthält rekursive Hilfsfunktionen, damit der gesamte Objektbaum durchsucht werden kann.

SystemConfiguration Generiert die vom ModVis Service benötigte Systembeschreibung gemäss Abschnitt 5.2.5 als C-Datenstruktur in die Dateien `SystemDescription.c` und `SystemDescription.h`.

Alle Templates benötigen im zu animierenden Projekt genau eine Instanz von `Visualized SystemConfiguration`.

Zusätzliche Template Funktionen, die von einem oder mehreren Templates benötigt werden, sind in der Ressource `CommonFunctions` definiert. Einige dieser Funktionen sind als Java Methoden der Klasse `ch.actifsource.solution.modVis.template.CommonFunctions` umgesetzt, deren Implementation sich im Projektordner `src-gen` befinden.

Zudem wird die Klasse `ch.actifsource.solution.modVis.helper.ElementIdentifier` im Projektordner `src` benötigt, um UUIDs in C-Literale zu transformieren.

5.5.3. Build Konfigurationen

Build Konfigurationen beschreiben Aktionen für die Erstellung von Code Files, die in ein gemeinsames Zielverzeichnis generiert werden sollen und verfügen über einige konfigurierbare Optionen. ModVis stellt folgende Build Konfigurationen zur Verfügung:

ModVis_C_Model_Visualisation Für die Generierung der vom ModVis Servcie benötigten Systembeschreibung

ModVis_HTML_Model_Visualisation Für die Generierung der vom ModVis Client benötigten Systembeschreibungen

5.6. Erweiterungen der Communicating Interacting Processes (CIP) Tools

Damit ModVis über Änderungen des Zustandes einer Steuerung informiert wird, muss aus dem generierten Programm bei jeder Zustandsänderung `mvSetElementStates()` aufgerufen werden (siehe Abschnitt 4.5.2). Für den Einsatz von ModVis in einer mit der *CIP*-Methode entwickelten Steuerung, müssen deshalb die Code Templates des CIP Tools leicht erweitert werden.

Die nachfolgende Beschreibung der notwendigen Anpassungen kann auch als Leitfaden für die Integration in weitere actifsource *Solutions* verwendet werden.

5.6.1. CIP Code Options

Die Generierung des C-Codes eines CIP-Modelles kann durch eine Instanz von `C_CodeOptions` im Package `ch.actifsource.solution.cip.generic.generator.tpl_c.c_codeoptions` konfiguriert werden. Bestehende Konfigurationsmöglichkeiten erlauben zum Beispiel das im generierten Code nach jedem Zustandsübergang ein `printf()` Statement ausgeführt wird, damit die Transaktionen auf der Konsole nachverfolgt werden können.

Ein zusätzliches Attribut `animation` vom Typ `Animation` in `C_CodeOptions` soll die Konfiguration der ModVis Aufrufe ermöglichen. Abbildung 5.5 zeigt einen Teil der erweiterten `C_CodeOptions` Klasse.

`Animation` enthält zur Zeit nur das Attribut `modVis_Visualisation` welches das Generieren der ModVis Aufrufe aktiviert.

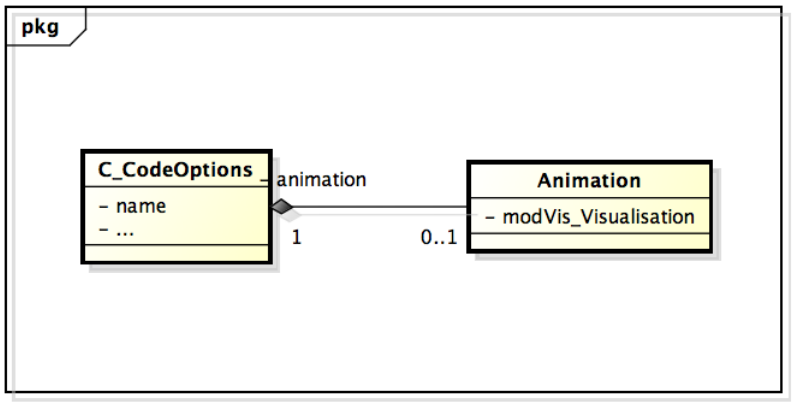


Abbildung 5.5.: Zusätzliches Attribut der CIP Code Options

5.6.2. ModVis Adapter

Damit die Kommunikation zwischen den Prozessen, die die Steuerungslogik ausführen, und dem ModVis Webservice auf das jeweilige Zielsystem angepasst werden kann, wird das Aktualisieren der ModVis Datenstruktur durch ein zusätzliches Interface abstrahiert.

Dies bedeutet, dass sobald die Animation über die Code Options aktiviert wurde, zusätzlich das Header File `modVisAdapter.h` generiert werden muss. `modVisAdapter.h` deklariert die Methoden für das Aktualisieren der ElementStates sowie den `MvUuid` Datentypen.

```
1 #ifndef MODVISADAPTER_H_
2 #define MODVISADAPTER_H_
3
4 #ifndef MODVIS_UUID_
```

```
5  #define MODVIS_UUID_
6  /**
7   * Represents an UUID as an array of 16 bytes.
8   */
9  typedef struct {
10        unsigned char bytes[16];
11 } MvUuid;
12 #endif
13
14 /**
15  * Informs the modVis webservice about appearance changes of one or more elements.
16  */
17 void mvaSetElementStates(unsigned elementCount,
18                MvUuid elementIds[], unsigned appearances[]);
19
20 /**
21  * Informs the modVis webservice about appearance changes of all elements of the diagram.
22  */
23 void mvaSetAllElementStatesOfDiagram(MvUuid diagramId, unsigned appearance);
24
25 #endif /* MODVISADAPTER_H_ */
```

Listing 5.20: Deklaration des ModVis Adapters

Die Definitionen der Funktionen in `modVisAdapter.h` müssen je nach Zielplattform implementiert werden und können deshalb nicht von der CIP Umgebung generiert werden. Allerdings kann das CIP Tool die Funktionsrümpfe mit Protected Regions generieren, damit der generierte Code kompiliert werden kann.

```
1  #include "modVisAdapter.h"
2
3  void mvaSetElementStates(unsigned elementCount,
4                MvUuid elementIds[], unsigned appearances[]){
5        // protected region for specific implementation
6  }
7
8  void mvaSetAllElementStatesOfDiagram(MvUuid diagramId, unsigned appearance){
9        // protected region for specific implementation
10 }
```

Listing 5.21: Definition des ModVis Adapters mit leeren Funktionsrümpfen für die plattformspezifische Implementation

5.6.3. CIP Code Templates

Neben den Code Options müssen auch die CIP Templates für die Generierung des C Codes für die Prozesse erweitert werden.

Listing 5.22 zeigt Auszüge aus dem generierten Quellcodes eines Prozesses erweitert um die für die Integration von ModVis benötigten Codezeilen. Zu beachten sind insbesondere die Zeile 1, welche die ModVis Bibliothek einbindet, sowie die Zeilen 16 bis 21 die den Aufruf von `mvaSetElementStates()` enthält.

```
1  #include "modVisAdapter.h"
2
3  // ...
4
5  void fICHAN_Evt_ButtonHardener(enum eMSG_Evt_ButtonHardener name_)
6  {
7    switch(name_)
8    {
```

```
9     /* MESSAGE Push of INPORT ButtonHardener */
10   case C1_Push:
11     switch(status_ButtonHardener.read_access_.STATE)
12     {
13     case released:
14       // tell modVis some elements should change
15       mvaSetElementStates(
16          // number of elements to be updated
17          2,
18          // ids of elements to update
19          (MvUuid[2]) { {{0x01 /*...*/}}, {{0x02 /*...*/}} },
20          // new appearances for every element
21          (unsigned[2]){1, 1});
22       status_ButtonHardener.write_access_.STATE = pushed;
23       fPULSE_PassengersControl (O4_push);
24       break;
25     default:
26       return;
27     }
28     break;
29
30   // ...
31
32   default:
33     return;
34   }
35   return;
36 }
```

Listing 5.22: Um die modVis Aufrufe erweiterter Code eines CIP Prozesse

Mithilfe von `mvaSetAllElementStatesOfDiagram()` kann, neben dem setzten einzelner ElementStates, auch die Darstellung aller Elemente eines Diagrammes gesetzt werden. Dies ist zum Beispiel beim wechseln eines Modes notwendig. Dabei müssen alle Elemente des Diagrammes des vorherigen Modes zurückgesetzt oder deaktiviert werden.

5.6.4. CIP Test Suite Templates

Damit die Visualisierung einer lokal simulierten CIP Steuerung ermöglicht wird, muss beim ausführen der CIP Test Suite der ModVis Standalone HTTP Server gestartet werden. Um dies zu ermöglichen, müssen ebenfalls die Code Templates der CIP Test Suite erweitert werden. Zeilen 1 und 2 in Listing 5.23 enthalten die benötigten Include Anweisungen. `SystemDescription.h` enthält jeweils die generierte Beschreibung des zu visualisierenden Systems sowie die Konfigurationsoptionen des HTTP Servers.

In Zeile 15 wird der HTTP Server gestartet. Mit dem Start des HTTP Server wird auch die ModVis Umgebung initialisiert. Deshalb sollte `mvsStart()` vor der Initialisierung der Steuerung (`fINIT_()`) ausgeführt werden, damit die Änderungen des Systemzustandes von Beginn an aufgenommen werden.

Damit alle Ressourcen sauber freigegeben werden, muss der HTTP Server vor dem Beenden der Test Suite durch den Aufruf von `mvsStop()` angehalten werden.

```
1  #include "modVisStandalone.h" // for mvsStart() and mvsStop()
2  #include "SystemDescription.h" // for systemDescription and mvsServerOptions
3
4  // ...
5
6  int main(int argc, char *argv[])
7  {
```

```
8   int ret;
9   if (argc == 1)
10  {
11    displayInfoText(argv[0]);
12    return 1;
13  }
14
15  mvsStart(&systemDescription, mvsServerOptions);
16
17  if (strcmp(argv[1],"manual")==0)
18  {
19    ret = manualAnimation()?1:0;
20    mvsStop();
21    return ret;
22  }
23  if (strcmp(argv[1],"regressive")==0)
24  {
25    ret = regressiveAnimation()?1:0;
26    mvsStop();
27    return ret;
28  }
29  displayInfoText(argv[0]);
30
31  mvsStop();
32
33  return 1;
34 }
```

Listing 5.23: Starten und Stoppen des ModVis Standalone HTTP Server in der CIP Test Suite

6. Validierung der umgesetzten Lösung

6.1. Performance und Kapazitätsgrenzen

Der Einsatz von ModVis in einer produktiven Umgebung setzt voraus, dass die zusätzlich dafür benötigten Systemressourcen nicht die Stabilität der Steuerung beeinträchtigen. Besonders im Hinblick zum Einsatz auf Echtzeitsystemen ist es wichtig, den Einfluss auf die Prozessorbelastung zu verstehen, da die konfigurierten Zykluszeiten auch mit aktivierter Animation zwingend eingehalten werden müssen.

Aus diesen Gründen werden Performance- respektive Stresstests durchgeführt, die die effektiven Kapazitätsgrenzen des Systems aufdecken und einen Vergleich mit den Anforderungen ermöglichen. Zudem können aufgrund der Resultate Vorschläge für die Konfiguration von ModVis auf unterschiedlichen Systemen abgeleitet werden.

Die Tests sollen insbesondere die Beantwortung folgender Fragen ermöglichen:

- Können die in den Anforderungen festgelegten Kriterien erfüllt werden?
- Was sind geeignete Zykluszeiten für den Betrieb des ModVis Webservices?
- Wie grosse Modelle können noch animiert werden?
- Wieviele Zustandsänderungen von animierten Elementen können pro Sekunde verarbeitet werden?
- Wie wirkt sich die serverseitige Aufzeichnung von Zustandsänderungen auf die Kapazitätsgrenze aus?

Die für die Testumgebung spezifischen Begriffe Zykluszeiten, Taskklasse und Priorität können im Handbuch der B&R Entwicklungsumgebung nachgelesen werden und werden daher nicht näher erklärt.

6.1.1. Testumgebung

Als Referenzsystem für die Performancetests wird die von der Firma B&R Automation zur Verfügung gestellte *SPS* X20CP1586 verwendet. Aus Tabelle 6.1 ist ersichtlich, dass sich die X20CP1586 für SPS Steuerungen eher im oberen Leistungssegment bewegt und dadurch gut geeignet ist, um zusätzliche Diagnoseservices wie ModVis zu betreiben.

Prozessor	Intel Atom E680T 1.60 GHz
RAM	512 MB DDR2-SDRAM
Minimale Zykluszeit für Taskklassen	100 μs
Durchschnittliche Instruktionsdauer	0.0027 μs
Ethernet	10/100/1000 Mbps

Tabelle 6.1.: Wichtigste Kennzahlen der X20CP1586

Die SPS Steuerung wird über ein 100 cm langes Ethernet Kabel direkt mit dem PC verbunden, von dem aus die Messungen vorgenommen werden. Da sich die Antwortszeiten des in der Automation Runtime integrierten Webservers erfahrungsgemäss im Bereich ab 90 ms bewegen, wird davon ausgegangen, dass die Dauer für das Empfangen und Verarbeiten der Antworten verhältnismässig gering (< 1ms) ist und daher vernachlässigt werden kann. Die Messungen sollten daher mit jedem handelsüblichen Laptop oder Desktop PC wiederholbar sein und vergleichbare Resultate liefern.

Konfiguration der Hardwaresteuerung

Für alle beschriebenen Tests werden die in Tabelle 6.2 aufgelisteten Konfigurationen verwendet. Einzige Ausnahme ist die Taskklasse 8, deren Zykluszeit für das Ermitteln der optimalen Einstellung noch variiert wird.

Entwicklungstools	
Eclipse	Indigo Service Release 2
Actifsource Enterprise	5.8.0.201305150054
Automation Studio Desktop	AS4.0.14.170
GNU C Compiler	V 4.1.2
Messtools	
ApacheBench	2.3
Steuerung	
Hardware	X20CP1586
Laufzeitumgebung	Automation Runtime I4.04
Netzwerkkonfiguration	
Interface	X20CP1586.IF3
Modus	Ethernet
MTU Grösse	1500
Baudrate	100 MBit Half Duplex
Taskklasse 1 (Steuerung)	
Zykluszeit	10 ms
Toleranz	10 ms
Taskklasse 8 (Webservice)	
Zykluszeit	10 ms
Toleranz	30 s

Tabelle 6.2.: Konfiguration der Testumgebung

6.1.2. Akzeptanzkriterien

Die in den Anforderungen definierten Randbedingungen stellen die Grundlage für die Performancemessungen dar und sollen im Minimum erfüllt werden. Aufgrund der Erkenntnisse die während der Konzeptionierung und Umsetzung gewonnen wurden, sind die Kriterien aus den Anforderungen noch genauer spezifiziert.

Kriterium	Zielwert
Zykluszeit des Steuerungsprozesses	10 ms
Toleranz des Steuerungsprozesses	10 ms
Animierte Elemente	1000
Animierte Diagramme	100
Zustandswechsel eines Elementes pro Sekunde	1000
Zustandswechsel eines Elementes pro Zyklus	10
Recordgrösse pro Diagramm	10
Angeforderte Diagramme pro Request	5
Maximale Reaktionszeit des Clients auf Zustandsänderungen	500 ms
Maximale Antwortszeit des Services	200 ms

Tabelle 6.3.: Akzeptanzkriterien für Performance- und Stresstests

Die einzuhaltenden Zykluszeiten entsprechen den in der durch Automation Studio festgelegten Standardeinstellungen für die X20CP1586 SPS in der höchstpriorisierten Taskklasse. Die übrigen Kriterien leiten sich direkt von den in den Anforderungen festgelegten Kriterien ab.

Unter der maximalen Reaktionszeit auf Zustandsänderungen ist die Zeitspanne zwischen dem effektiven Ereignis, z.B. das Bedienen eines Schalters, und dem entsprechenden Zustandswechsel in der Animation im Browser zu verstehen. Daraus lässt sich ableiten, dass die maximale Antwortszeit auf einen Request inklusive der Zeit um die empfangenen Daten darzustellen 250 ms nicht überschreiten darf. Die in den Akzeptanzkriterien festgelegten 200 ms gehen von der Annahme aus, dass das Darstellen der Daten sowie das versenden des nächsten Requests nicht länger als 50 ms dauern, was auch auf älteren Laptops noch ein grosszügiger Wert darstellt.

6.1.3. Testszenarien

Um die oben erwähnten Fragestellungen zu beantworten wurden unterschiedliche Testszenarien entworfen und durchgeführt.

Szenario 1: Variierende Zykluszeiten für den Webservice

Damit der ModVis Webservice in den folgenden Szenarien mit einer hinreichenden Konfiguration betrieben werden kann, muss zuerst der Einfluss der Zykluszeiten, mit denen der Service betrieben wird, auf die Antwortszeiten untersucht werden.

Die Standardkonfiguration der X20CP1586 SPS in Automation Studio bietet für Services, die nicht direkt für die Steuerung der Hardware zuständig sind, die Taskklasse 8 mit der niedrigsten Priorität. Per Default ist diese Taskklasse mit einer Zykluszeit von 10 ms und einer Toleranz von 30 Sekunden konfiguriert. Durch die geringe Priorität dieser Taskklasse, wird sichergestellt, dass zusätzliche Services nicht den Betrieb der Steuerung, durch das Aushungern anderer Prozesse, beeinträchtigen können.

Im Testszenario 1 werden die Zykluszeiten, mit denen der Webservice ausgeführt wird, zwischen den verschiedenen Testläufen variiert. Dabei wird die Steuerung unter der in den Akzeptanzkriterien festgelegten Auslastung betrieben.

Die verwendete Implementation des Webservervices benötigt jeweils mindestens einen Zyklus für das Empfangen eines Requests, die Verarbeitung des Requests und das Versenden der Response. Deshalb ist anzunehmen, dass die Antwortszeit im Minimum drei mal

grösser als die Zykluszeit ist.

Szenario 2: Grösse des zu animierenden Modelles

Die Prozessorbelastung für das Aktualisieren des Systemsnapshots und der Records bei Zustandsänderungen ist stark von der Anzahl Diagramme und Elemente pro Diagramm abhängig. Insbesondere der Aufwand für das Aktualisieren der Records steigt quadratisch zur Grösse des Modells.

Im Szenario 2 sollen ausgehend von den, in den Akzeptanzkriterien beschriebenen, Parametern schrittweise die Anzahl Diagramme im System erhöht werden. Die Anzahl animierter Elemente pro Diagramm bleibt dabei unverändert bei 10. Das Ziel ist, die maximale Anzahl Diagramme zu finden, die noch ohne verletzen der Zykluszeiten aktualisiert werden können. Um den Einfluss der Recordsverwaltung auf die Leistungsfähigkeit zu überprüfen, werden die Messungen zweimal durchgeführt. Zuerst mit deaktivierten Records und anschliessend mit 10 Recordsamples pro Diagramm.

Szenario 3: Frequenz der Zustandswechsel

Im Szenario 3 sollen die Kapazitätsgrenzen hingehend auf die maximale Anzahl Aktualisierungen pro Sekunde untersucht werden. Wiederum wird von den Parametern in den Akzeptanzkriterien ausgegangen. Diesmal werden jedoch die Anzahl Zustandswechsel pro Zyklus schrittweise erhöht, bis die Aktualisierungen nicht mehr verarbeitet werden können. Die Messungen werden ebenfalls je einmal mit und einmal ohne Records durchgeführt.

6.1.4. Implementation der Testanordnung

Damit möglichst einfach Systeme mit den oben genannten Parametern erstellt werden können, wurde ein zusätzliches actifsource Projekt `ch.actifsource.solution.modVis.performance` implementiert, dass anhand unterschiedlicher Konfigurationen die entsprechenden Projekte für die B&R Entwicklungsumgebung Automation Studio generieren kann.

Für die Messung der Antwortszeiten wird Apache Bench 2.3 eingesetzt.

Generieren der Testsysteme

Das Projekt `ch.actifsource.solution.modVis.performance` enthält verschiedene Templates, die die Variablen Bestandteile eines Automation Studio Projektes . Das generierte Projekt befindet sich im Ordner `ch.actifsource.solution.modVis.performance/brPerformance`.

Folgende Templates werden verwendet um Quellcode und Automation Studio Metadaten zu generieren:

`BRCpuConf`
: Generiert die Zuordnung der erstellten Prozesse respektive Programme (Automation Studio Terminologie) zu den Taskklassen. Ziel der Codegenerierung sind die `Cpu.sw` Dateien in der Hardwarekonfiguration.

`BRPackageConf`
: Registriert für jeden Prozess ein Program in der `Package.pkg` Datei.

`BRPrgConf`
: Erstellt für jeden Prozess eine Programmkonfiguration in `processN/ANSIC.prg`.

`MockedProcess`
: Erstellt für jeden Prozess die entsprechenden C Codefiles in `processN/processN.c`

`MockedSystemDescription`
: Erstellt den Quellcode der Systembeschreibung in `SystemDescription.c`

`RunBenchmark`
: Erstellt eine Windows Batch Datei mit der korrekten Kommandozeilenanweisung um den Performancetest durchzuführen.

Die Templates verwenden unterschiedliche Template-Funktionen in `TestGeneratorFunctions`. Diese Template-Funktionen verwenden die Java Hilfsklasse `SystemMocker` um die UUIDs der fiktiven Diagramme und Elemente zu erzeugen.

Konfiguration des Testsystemes

Die Templates zur Codegenerierung des Testsystemes verwenden eine Instanz der Klasse `PerfromanceTest` als Quelle der Testparameter. Folgende Attribute können für die Konfiguration des Testes angepasst werden:

`numberOfDiagrams`
: Anzahl Diagramme im fiktiven Testsystem

`numberOfElementsPerDiagram`
: Anzahl animierter Elemente pro Diagramm

`recordSize`
: Grösse der Aufzeichnung in den Records; ein Wert kleiner oder gleich eins deaktiviert die Aufnahme

`numberOfActiveProcesses`
: Anzahl der Prozesse die den Systemzustand aktualisieren; in allen umgesetzten Szenarien wird jeweils nur ein Prozess verwendet

`diagramsPerProcess`
: Anzahl Diagramm die von einem Prozess bearbeitet werden; sollte einem Teiler von `numberOfDiagrams` entsprechen

`updatesPerCycle`
: Anzahl der Elemente die pro Zyklus aktualisiert werden

`diagramsPerRequest`
: Anzahl der Diagramme die während der Performancemessung per HTTP angefordert werden; sollte einem Teiler von `numberOfDiagrams` entsprechen

Listing 6.1 zeigt Auszüge eines generierten Prozesses für einen Performancetest mit den Parametern `numberOfDiagrams = 10`, `elementsPerDiagram = 10`, `diagramsPerProcess = 3` und `updatesPerCycle = 10`. Der Prozess simuliert das Verhalten einer Hardwaresteuerung indem in jedem Zyklus 10 Elemente aktiviert, respektive deren `appearance` Attribute auf 1 gesetzt werden.

Damit eine gleichmässige Verteilung der aktualisierten Elemente innerhalb des Systemsnapshots erreicht wird und nicht nur Elemente zu Beginn oder am Ende manipuliert werden, kennt der Prozess die UUIDs von Elementen aus unterschiedlichen Bereichen. In diesem Beispiel greift der Prozess auf Elemente aus dem ersten, vierten und siebten Diagramm von insgesamt 10 zu.

```
1  // ...
2  #include "modVisAdapter.h"
3
4  static int counter = 0;
5
6  // each uuid {0x00 ... 0x00} describes an element within a diagram
7  static MvUuid diagramElements[] = (MvUuid[30]){
8          {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0
               x00, 0x00, 0x00}},
9          // ...
10         {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0
               x00, 0x00, 0x09}}
11 };
12
13 // ...
14
15 // method 'process0Cyclic' gets called on every cylic of the realtime system
16 void _CYCLIC process0Cyclic(void){
17         int i;
18         for(i = 0; i < 10; i++){
19                 mvaSetElementStates(1, (MvUuid[1]) {
20                                 diagramElements[(counter + (i * 3)) % 30]
21                         }, (unsigned[1]){1});
22         }
23
24         counter++;
25 }
26
27 // ...
```

Listing 6.1: Auszüge eines zur Performancemessung verwendeten Prozesses

Performancemessungen mit Apache Bench

Für die Messung der Antwortzeiten des Webservers wird *Apache Bench (ab)* verwendet. *ab* wurde entwickelt um die Performance von Installationen des Apache Webservers zu untersuchen, kann aber auch mit jedem anderen HTTP Server eingesetzt werden.

ab sendet eine gewünschte Anzahl HTTP Requests an den Server und sammelt Daten über die Antwortzeiten sowie über allfällige Verbindungs- und Serverfehler.

```
1  ab -n 1000 -c 1 "http://169.254.54.100/modVis/snapshot?diagram=00000000-0000-0000-7fff-
       ffffffffffff&diagram=00000000-0000-0002-7fff-ffffffffffff&diagram=00000000-0000-0004-7
       fff-ffffffffffff&diagram=00000000-0000-0006-7fff-ffffffffffff&diagram
       =00000000-0000-0008-7fff-ffffffffffff"
```

Listing 6.2: Beispielaufruf von *ab*

Der Kommandozeilenaufruf in Listing 6.2 startet eine Messung mit *ab*. Während der Messung werden 1000 mal die Snapshots von fünf Diagrammen angefordert. Durch die Option `-c 1` wird jeweils nur ein nebenläufiger Request abgesendet, was dem Verhalten des ModVis Clients entspricht. Durch ein erhöhen des Wertes könnten auch mehrere Benutzer zur selben Zeit simuliert werden.

Alle Messungen werden mit 1000 aufeinanderfolgenden Requests durchgeführt.

6.1.5. Messergebnisse

Szenario 1: Variierende Zykluszeiten für den Webservice

Testszenario 1 wurde mit den in Tabelle 6.4 beschriebenen Konfigurationen durchgeführt. Die Konfigurationen entsprechen den in den Akzeptanzkriterien festgelegten Werten und simulieren ein eher grosses Modell mit insgesamt 1000 animierbaren Diagrammelementen, wie Zustände oder Zustandsübergänge.

Parameter	Wert
numberOfDiagrams	100
numberOfElementsPerDiagram	10
recordSize	10
numberOfActiveProcesses	1
diagramsPerRequest	5
updatesPerCycle	10
Zykluszeit Taskklasse 1	10 ms
Zykluszeit Taskklasse 8	Variabel (1 - 100 ms)

Tabelle 6.4.: Testkonfiguration Szenario 1

Tabelle 6.5 und Abbildung 6.1 zeigen den Zusammenhang zwischen der Zykluszeit mit der der Webservice betrieben wird und dessen Antwortzeiten. Wie erwartet wurde, unterschritten die Antwortzeiten nie das Dreifache der Zykluszeit. Zudem zeigte sich, dass durch Änderungen an Zykluszeiten von weniger als 20 ms nur noch geringe Performancesteigerungen erzielt werden können.

Zykluszeit [ms]	Min [ms]	Durchschnitt [ms]	Median [ms]	Max [ms]	Erfolgreich [Request]	Fehlgeschlagen [Request]
1	79	108	109	130	1000	0
10	93	114	114	136	1000	0
20	103	120	120	175	1000	0
30	119	156	155	184	1000	0
100	340	400	400	425	1000	0

Tabelle 6.5.: Resultate Testszenario 1

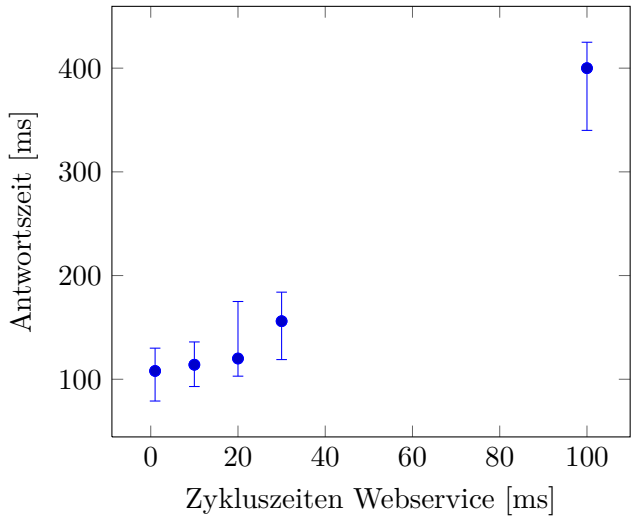


Abbildung 6.1.: Resultate Testszenario 1

Die Ergebnisse der Messungen zeigen, dass für den Betrieb des Webservices Zykluszeiten zwischen 10 und 30 ms gut geeignet sind, um auch bei grösseren Steuerungsmodellen angemessene Reaktionszeiten zu erreichen. Tiefere Werte erhöhen die Reaktionszeit nur noch unwesentlich. Aus diesem Grund wird für die weiteren Tests mit der Standardeinstellung von 10 ms für die Taskklasse 8 gearbeitet. Dieser Wert scheint ein guter Kompromiss zwischen Performance und Prozessorbelastung darzustellen.

Szenario 2: Grösse des zu animierenden Modelles

Tabelle 6.6 enthält die für das Testszenario 2 verwendeten Konfigurationsparameter. Die Anzahl der Diagramme im gesamten Modell wurde dabei kontinuierlich erhöht, bis das Testsystem die Zustandsänderungen nicht mehr verarbeiten konnte.

Parameter	Wert
numberOfDiagrams	Variabel (10 - ...)
numberOfElementsPerDiagram	10
recordSize	0 / 10
numberOfActiveProcesses	1
diagramsPerRequest	5
updatesPerCycle	10
Zykluszeit Taskklasse 1	10 ms
Zykluszeit Taskklasse 8	10 ms

Tabelle 6.6.: Testkonfiguration Szenario 2

Bei den Messungen mit deaktivierten Records konnten die Kapazitätsgrenzen des getesteten Systems nicht ausgemacht werden, da das Generieren der fiktiven Steuerungslogik in actifsource bei mehr als 10000 Diagrammen mehr als eine Stunde beanspruchte.

Mit aktivierten Records trat ab 3000 Records nach dem Übertragen des Steuerungsprozesses augenblicklich eine *Cycletime violation* auf und die Hardware wechselte in den Service Modus. Requests konnten keine mehr abgesendet werden.

Wie aus Tabelle 6.7 und Abbildung 6.2 ersichtlich ist, ist die Reaktionszeit von ModVis bei Systemen mit bis zu 1000 Diagrammen noch in einem für den Benutzer angemessenen Bereich. Grössere Systeme, insbesondere mit aktivierten Records, resultieren in mehreren Sekunden Differenz zwischen dem effektiven Ereignis und der Animation im Browser.

Diagramme [Stk]	Min [ms]	Durchschnitt [ms]	Median [ms]	Max [ms]	Erfolgreich [Request]	Fehlgeschlagen [Request]
Records deaktiviert						
10	86	101	101	110	1000	0
100	92	120	119	136	1000	0
1000	317	336	339	352	1000	0
10000	2955	2979	2980	3994	1000	0
10 Records per Diagram						
10	89	100	100	103	1000	0
100	100	117	119	131	1000	0
1000	568	574	570	601	1000	0
2000	1159	1160	1160	1163	1000	0
2500	4720	4783	4760	4859	1000	0
3000		*Cycletime violation*			0	0

Tabelle 6.7.: Resultate Testszenario 2

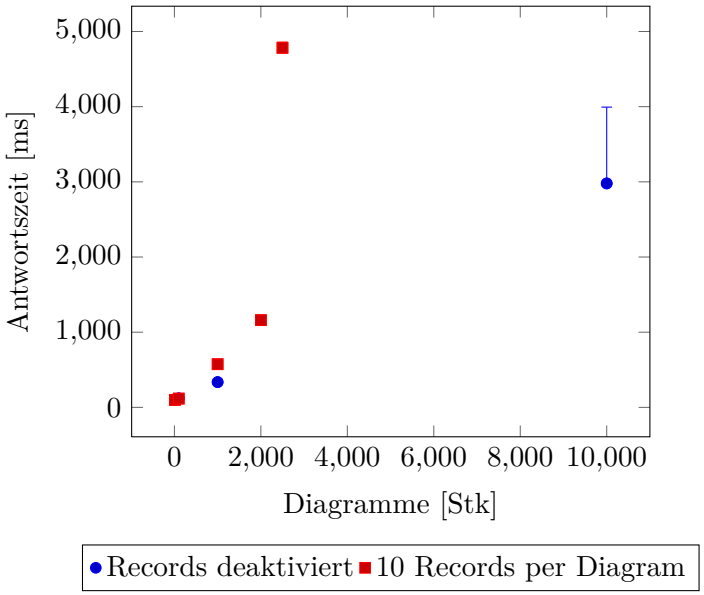


Abbildung 6.2.: Resultate Testszenario 2

Szenario 3: Frequenz der Zustandswechsel

Für das dritte Testszenario wurden Steuerungen mit variierender Anzahl Zustandswechsel pro Zyklus generiert. Tabelle 6.8 enthält die genauen Konfigurationsparameter.

Parameter	Wert
numberOfDiagrams	100
numberOfElementsPerDiagram	10
recordSize	0 / 10
numberOfActiveProcesses	1
diagramsPerRequest	5
updatesPerCycle	Variabel (1 - ...)
Zykluszeit Taskklasse 1	10 ms
Zykluszeit Taskklasse 8	10 ms

Tabelle 6.8.: Testkonfiguration Szenario 3

Die Resultate des dritten Testszenarios zeigen eine noch grössere Diskrepanz zwischen der Ausführung mit und der ohne Records. Mit aktivierten Records ist bereits bei 250 Updates pro Zyklus (25'000 / s) das Programm nicht mehr ausführbar.

Updates [Stk/Zyklus]	Min [ms]	Durchschnitt [ms]	Median [ms]	Max [ms]	Erfolgreich [Request]	Fehlgeschlagen [Request]
Records deaktiviert						
1	95	114	114	136	1000	0
1000	111	125	121	140	1000	0
5000	233	240	240	260	1000	0
8000	371	382	380	559	1000	0
9000	*Cycletime violation*				0	0
10 Records per Diagram						
1	106	128	130	335	1000	0
100	179	192	190	209	1000	0
200	265	280	280	301	1000	0
250	*Cycletime violation*				0	0

Tabelle 6.9.: Resultate Testszenario 3

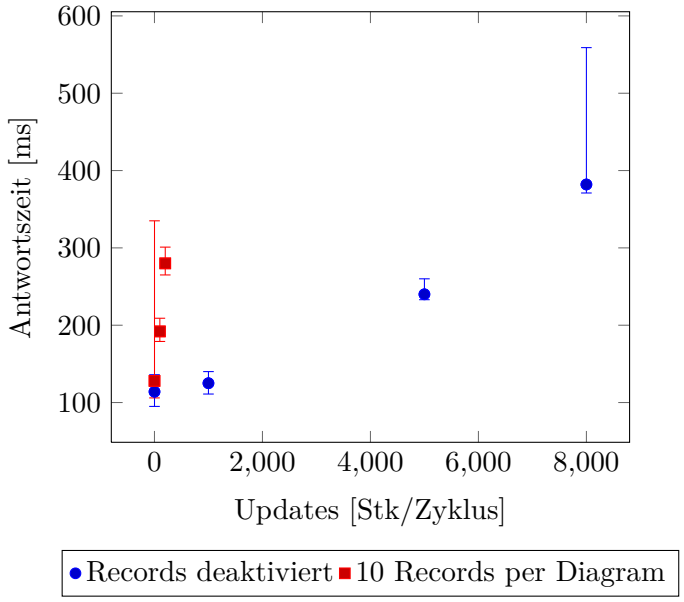


Abbildung 6.3.: Resultate Testszenario 3

6.1.6. Schlussfolgerungen

Mit den Performancemessungen konnte gezeigt werden, dass ModVis die festgelegten Anforderungen vollumfänglich erfüllen kann. Es können Steuerungen animiert werden, die mehrere tausend Diagramme umfassen und pro Sekunde mehrere zehntausend Zustandsänderungen vollziehen.

Als grösster Flaschenhals stellte sich das Aufzeichnen der Zustandsänderungen in den Records heraus. Dies lässt sich mit der aktuellen Implementation der ModVis Datenstruktur erklären. Da für jeden Zustandswechsel jeder Record überprüft werden muss, ob dieser für die Aufzeichnung des betroffenen Elements zuständig ist, ergibt sich eine hohe zeitliche Komplexität. Dieser Umstand könnte durch eine zusätzliche Indexstruktur behoben werden, die für jedes Element im System auf die betroffenen Records verweist. Ein solcher Index könnte mit actifsource generiert und auf der Hardware im statischen Speicher abgelegt werden. Dies wäre ein Beispiel für eine häufig vorgenommene Optimierung, bei der die Prozessorbelastung gegen erhöhten Speicherbedarf eingetauscht werden kann.

Für die Messungen wurde zudem eine Implementation des ModVisAdapters verwendet, bei der der selbe Prozess, der eine Zustandsänderung auslöst, auch für das Aktualisieren der Datenstruktur zuständig ist. Dadurch wurde durch ModVis die Auslastung des Steuerungsprozesses massiv erhöht. Da die Steuerungsprozesse mit sehr strikten Toleranzwerten bezüglich dem Einhalten der Zykluszeiten konfiguriert werden, führte dies ab einer gewissen Systemgrösse zu *Cycletime Violations*. Durch den Einsatz eines zusätzlichen Prozesses, der die ModVis Datenstruktur aktualisiert und mit einer höheren Toleranzzeit ausgeführt wird, könnte somit die Stabilität des Systems zusätzlich erhöht werden.

6.1.7. Konfigurationsempfehlungen

Aufgrund der Resultate konnten einige Erfahrungswerte betreffend der Konfiguration von ModVis für den produktiven Einsatz gewonnen werden. Zwar beziehen sich die Erkenntnisse in erster Linie auf die verwendete X20CP1586, können aber auch, unter Berücksichtigung der Kennzahlen, auf weitere Hardwaresteuerungen übertragen werden.

Ausführung Webservice

Der ModVis Webservice kann auf einem Prozess mit sehr geringer Priorität und hohen Toleranzzeiten ausgeführt werden. Beim Einsatz auf Systemen mit der AR Laufzeitumgebung sind Zykluszeiten zwischen 10 und 30 ms ein guter Kompromiss zwischen Prozessorbelastung und Antwortszeiten.

Records

Das Aufzeichnen der Zustandsänderungen sollte insbesondere bei grossen Systemen mit bedacht aktiviert werden. Es ist auch möglich in der ModVis Konfiguration die Records nur für gewisse Diagrammtypen zu aktivieren. Die Anzahl Samples in einem Record hat auf die Performance jedoch keinen wesentlichen Einfluss.

6.2. Integration der Lösung in ein Steuerungsprojekt

Um die Funktionalität der ModVis Komponenten und deren Zusammenspiel zu überprüfen, wird zusätzlich die Integration in ein bestehendes actifsource Projekt durchgeführt. Durch die Integration sollen konzeptionelle Probleme aufgedeckt werden, die durch das Testen der einzelnen Komponenten nicht direkt ersichtlich sind. Zudem soll die Zweckmässigkeit der Lösung im Bezug auf die *CIP* Referenzdomäne gezeigt werden.

Nachfolgend wird der Finale stand des Integrationsprojektes beschrieben. Die Integration wurde jedoch als fortlaufender Prozess durchgeführt und mit jedem Zwischenrelease von ModVis aktualisiert.

6.2.1. Umgebung und Konfiguration

Wie schon bei den Performancemessungen wird die von der Firma B&R Automation zur Verfügung gestellte *SPS* X20CP1586 eingesetzt. Die Kennzahlen der Hardware sind im Abschnitt 6.1.1 bereits beschrieben.

Die Integration wird mit der in Tabelle 6.10 aufgeführten Konfiguration durchgeführt. Anzumerken ist, dass für das Compilieren der Standalone Umgebung der GCC V 4.7.2 der MinGW Entwicklungstools verwendet wird. Für das Deployment auf der SPS wird hingegen der mit Automation Studio mitgelieferte GCC V 4.1.2 benutzt. Zudem wurden die in Abschnitt 5.6 beschriebenen Erweiterungen an der *CIP*-Solution vorgängig durch die Firma actifsource durchgeführt.

6.2.2. Modellierung der Torsteuerung

Für die Integration wird die Steuerung des in Abbildung 6.4 dargestellten Tormodells mit den *CIP* Tools modelliert. Das CIP Modell wurde von actifsource zur Verfügung gestellt und bildet die Steuerungslogik mit folgenden Funktionen ab:

- Durch drücken der Taste 1 öffnet sich das Tor.
- Das vollständig geöffnete Tor schliesst sich nach Ablauf eines Timers automatisch.
- Wird die Lichtschranke im Durchgang unterbrochen oder die Taste 1 betätigt, während sich das Tor schliesst, öffnet es sich wieder.
- Durch drücken der Taste 2 wird der Alarmmodus gestartet.
- Im Alarmmodus wird das Tor sofort geschlossen, die Lichtschranke und Taste 1 werden ignoriert.

Entwicklungsumgebung (Modellierung)	
Microsoft Windows	8
Eclipse	Juno Service Release 2
Actifsource Enterprise	5.8.0.201306041844
MinGW	20120426
GNU C Compiler	V 4.7.2
Mongoose Webserver	3.7
Entwicklungsumgebung (Deployment)	
Microsoft Windows	8
Automation Studio Desktop	AS4.0.14.170
GNU C Compiler	V 4.1.2
Browser	
Mozilla Firefox	21
Steuerung	
Hardware	X20CP1586
Laufzeitumgebung	Automation Runtime I4.04
Netzwerkkonfiguration	
Interface	X20CP1586.IF3
Modus	Ethernet
MTU Grösse	1500
Baudrate	100 MBit Half Duplex
Taskklasse 1 (Steuerung)	
Zykluszeit	10 ms
Toleranz	10 ms
Taskklasse 8 (Webservice)	
Zykluszeit	10 ms
Toleranz	30 s

Tabelle 6.10.: Konfiguration der Testumgebung

- Durch erneutes drücken der Taste 2 wird der Alarmmodus wieder deaktiviert.

Das von der Firma actifsource vorbereitete CIP-Modell befindet sich im Projekt `ch.actifsource.solution.modVis.cip.doorSystem` und besteht aus mehreren unterschiedlichen Diagrammen. Unter anderem sind mehrere animierbare Zustandsdiagramme enthalten.

6.2.3. Setup des Projektes

Das Projekt mit dem Tormodell wurde zusammen mit den folgenden Projekten in Eclipse Importiert:

- ch.actifsource.solution.modVis
- modVisLib
- modVisStandaloneLib

Diese Projekte mussten zudem in Eclipse als Projektabhängigkeiten definiert werden. Weiter wurde innerhalb des Projekts `ch.actifsource.solution.modVis.cip.doorSystem`

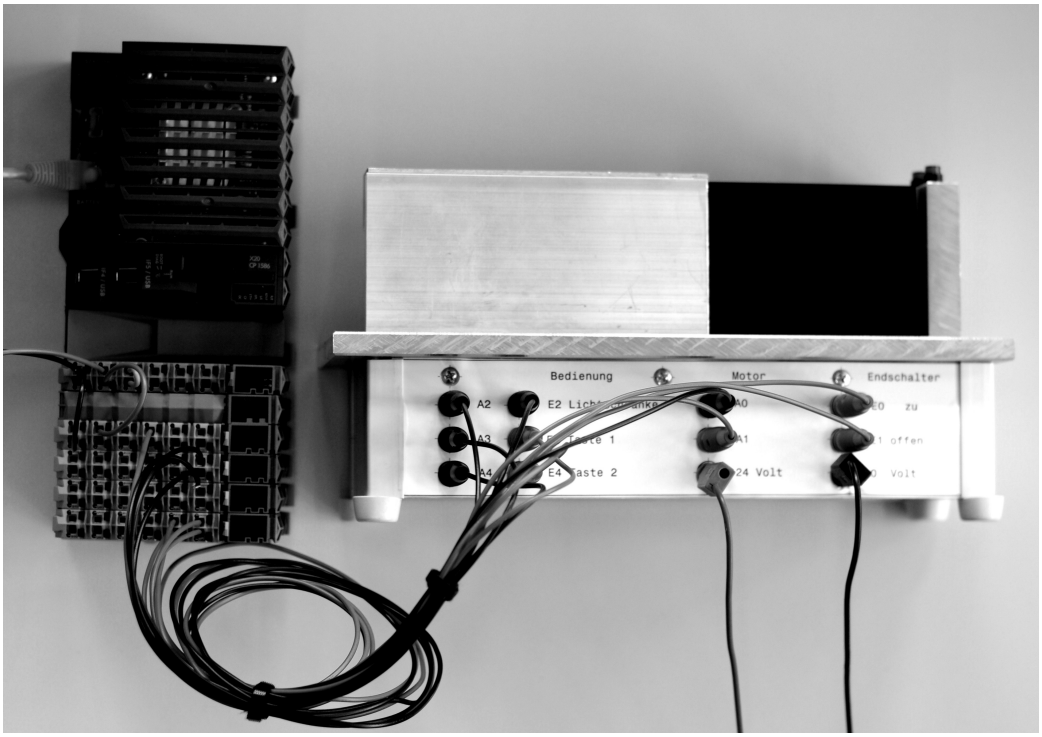


Abbildung 6.4.: Tormodell mit der X20CP1586 SPS von B&R

im Ordner `/brProject` ein neues *Automation Studio (AS)* Projekt erstellt. Das *AS* Projekt wurde innerhalb der actifsource Projektstruktur hinzugefügt, damit Pfade aus dem Projekt als Build Targets definiert werden können.

6.2.4. Einbinden der Animationskonfiguration

Für die Konfiguration von ModVis wurde die in Abbildung 6.5 dargestellte Instanz von `VisualizedSystemConfiguration` verwendet.

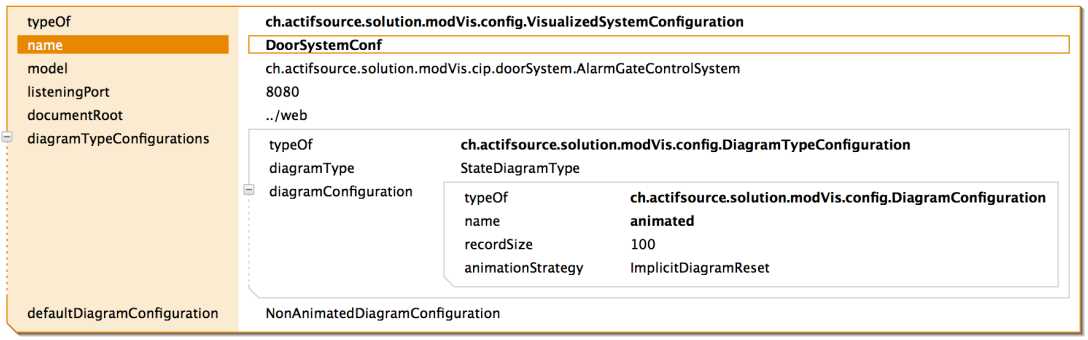


Abbildung 6.5.: Für das Integrationsprojekt verwendete Animationskonfiguration

Das `model` Attribut referenziert die `CIPSystem` Instanz des Torprojekts. Die beiden Attribute `listeningPort` und `documentRoot` werden für die Konfiguration des Standalone Webservers benötigt. Der Server der *AR* Laufzeitumgebung wird in Automation Studio konfiguriert.

Durch verwenden von `NonAnimatedDiagramConfiguration` als `defaultDiagramConfiguration`

wird die Animation von allen Diagrammen deaktiviert. Die einzige Ausnahme sind die Diagramme vom Typ `StateDiagramType`, die mit der Animationsstrategie `implicitDiagramReset` animiert werden und über einen 100 Samples umfassenden Aufnahmebuffer (`recordSize`) verfügen.

6.2.5. Automatische Codegenerierung

Abbildung 6.6 zeigt die konfigurierten Zielverzeichnisse für die Codegenerierung.

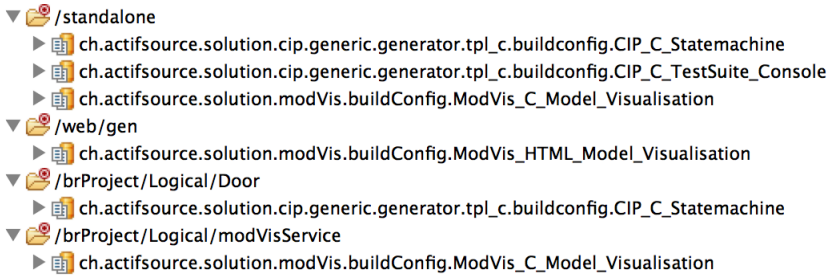


Abbildung 6.6.: Für das Integrationsprojekt verwendete Build Konfigurationen

Das Verzeichnis `/standalone` enthält alle benötigten Codedateien, um den Standalone Server zu kompilieren. Dazu gehören die CIP Steuerungslogik, die Dateien der CIP TestSuite mit der `main()` Funktion sowie die vom ModVis Service benötigte Systembeschreibung.

Die vom ModVis Client benötigten projektspezifischen Daten (SVGs und JSON Systemhierarchie) werden im Zielverzeichnis `/web/gen` abgelegt. Damit der ModVis Client aus dem selben Document Root wie die generierten Daten geladen werden kann, mussten zudem noch alle Dateien des Clients (HTML, JavaScript und CSS) manuell in das `/web` Verzeichnis kopiert werden.

Für das Deployment auf der *SPS* wurden zudem noch zwei Zielverzeichnisse im bereits erstellten *AS* Projekt konfiguriert. `/brProject/Logical/Door` enthält das Programm, welches die CIP Steuerungslogik ausführt und `/brProject/Logical/modVisService` den Webservice, der wiederum die Systembeschreibung benötigt.

6.2.6. Deployment der CIP TestSuite mit dem Standalone Server

Die bis hierher beschriebenen Schritte waren ausreichend um das Projekt in Eclipse zu kompilieren und im Standalone Betrieb auszuführen. Das Projekt musste in ein C-Projekt konvertiert werden und das Verzeichnis `/standalone` als einziges C-Source Folder definiert werden. Zudem mussten die beiden C-Bibliotheken `modVisLib` und `modVisStandaloneLib` als Abhängigkeiten hinzugefügt werden.

Nach dem Kompilieren konnte die CIP Test Suite mit der Kommandozeilenanweisung `ch.actifsource.solution.modVis.cip.doorSystem.exe manual` gestartet werden und der Client war über `http://localhost:8080/` erreichbar.

6.2.7. Deployment der Steuerung auf der SPS

Das komplette Einrichten des Projektes in der *AS* Umgebung wird nicht genauer ausgeführt, da viele umgesetzten Arbeitsschritte sehr spezifisch für die B&R Umgebung sind. Statt dessen wird die Integration der ModVis Bibliothek und die Kommunikation zwischen Steuerung- und Webservice Prozess beschrieben.

Da die Torsteuerung von einem Prozess (Programm in der *AS* Terminologie) ausgeführt wird und *AS* das gemeinsame Benutzen von Variablen zwischen Programmen erlaubt, konnte ein Deployment eingesetzt werden, dass dem im Abschnitt 4.2.4 beschriebenen Ansatz mit zwei separaten Prozessen und einem geteilten Speicherbereich gleicht.

Steuerungsprozess und Webservice teilen sich die global definierte Variable `pSystem`, die durch den Webservice in der Initialisierungsphase auf die Systembeschreibung referenziert wird. Listing 6.3 zeigt die für die Integration verwendete Adapterimplementation.

```
1  #ifdef _DEFAULT_INCLUDES // Automation Studio include mechanism, includes i.a. pSystem,
       clock_ms()
2  #include <AsDefault.h>
3  #endif
4
5  #include "modVisAdapter.h"
6  #include "modVis.h"
7
8  static unsigned long sequenceNo = 0;
9
10 MvTime nextSequenceNo(){
11         return ++sequenceNo;
12 }
13
14 MvTime now(){
15         return clock_ms()/1000; // AR sepcific time function
16 }
17
18 void mvaSetElementStates(unsigned elementCount,
19                 MvUuid elementIds[], unsigned appearances[]){
20         // update modVis data structrue
21         mvSetElementStates(pSystem, elementCount, elementIds, appearances, nextSequenceNo()
               , now());
22 }
23
24 void mvaSetAllElementStatesOfDiagram(MvUuid diagramId, unsigned appearance){
25         // update modVis data structrue
26         mvSetAllElementStatesOfDiagram(pSystem, diagramId, appearance, nextSequenceNo(),
               now());
27 }
```

Listing 6.3: Umsetzung des Adapters für die *AR* Laufzeitumgebung

Die Integration des Webservices in `/brProject/modVisService/modVisService.c` fällt etwas umfangreicher aus, da die ModVis C-Schnittstelle an die `AsHTTP` Funktionsblöcke (*AS* Konzept) angebunden werden muss. Der Webservice wird in einem Programm ausgeführt, dass vom System Scheduler zyklisch aufgerufen wird. Zudem wird jeweils mindestens ein Zyklus benötigt, um ein Request zu Empfangen, um den Request zu bearbeiten und um die Response zurückzusenden. Deshalb wurde das `modVisService` Programm im Sinne des State Patterns [Gam+95] durch die drei Funktionen `idle()`, `received()` und `sent()` umgesetzt, die je nach Systemzustand aufgerufen werden.

6.2.8. Erkenntnisse aus der Projektintegration

Die Integration von ModVis in ein bestehendes CIP-Projekt ist in wenigen Schritten durchführbar und das simulierte Modell kann mithilfe der CIP Test Suite und dem ModVis Standalone Server mit wenig Aufwand animiert werden. Aufwändiger ist hingegen das Ausliefern der Steuerung auf der Zielplattform, da der Adapter und die Anbindung an den Webserver umgesetzt werden müssen. Diese Komponenten können bei weiteren Projekten auf der selben Plattform jedoch ohne Anpassungen wiederverwendet werden.

Der Integrationsprozess kann jedoch an einigen Stellen noch verbessert werden. So könnte zum Beispiel zusammen mit den SVG Diagrammen gleich die ganze ModVis Clientapplikation an einem Stück in das `/web` Verzeichnis generiert werden.

7. Schlussfolgerung

7.1. Zentrale Konzepte für die Erfüllung der Anforderungen

In dieser Arbeit konnte eine Lösung für die Animation von Prozessabläufen auf Echtzeitsystemen gefunden werden, die die geforderte Flexibilität mit mehreren Konzepten untertstützt. So kann die Lösung durch das stark generalisierte Domänenmodell auf unterschiedliche Prozessmodelle, wie zum Beispiel die *CIP*-Methode, angewendet werden. Die entworfene Komponentenarchitektur mit einer zentralen Plattformabstraktionsschicht erlaubt es, mit wenig Aufwand die Lösung auf zusätzlichen Plattformen einzusetzen. Dabei werden mehrere unterschiedliche Deploymentszenarien unterstützt, die je nach verfügbaren Ressourcen und Funktionalitäten der Zielplattform eingesetzt werden können.

Die Anforderungen an die Zielplattform konnten stark reduziert werden. Grundlage dafür ist die ausgearbeitete Web-API, die eine kontinuierliche Animation von mehreren gleichzeitig geöffneten Diagrammen zulässt, ohne dass nebenläufige HTTP-Requests notwendig sind. Statt dessen können mehrere Ressourcen in einem gemeinsamen Request angefordert werden. Zudem wird die Grösse der Response reduziert, indem nur die dem Client noch nicht bekannten Zustandsdaten der Diagramme übermittelt werden.

Auch die eigenständig im Browser ausführbare Clientapplikation stellt keine besonderen Anforderungen an den Webserver, der sie zur Verfügung stellt. Dieser muss lediglich das Ausliefern von statischen Dateien unterstützen. Zudem führt die Clientapplikation die gesamte Darstellungs- und Bedienungslogik aus.

Der Webservice, der die Web-API zur Verfügung stellt, ist als C-Bibliothek umgesetzt, die ohne I/O-Operationen auskommt und absolut zustandslos arbeitet. Dadurch können die Aspekte der Nebenläufigkeit und Prozesssynchronisation an die plattformspezifische Integration ausgelagert werden. Zudem bestehen so keine Abhängigkeiten zu Systemfunktionen, die je nach Plattform unterschiedlich oder gar nicht zur Verfügung stehen. Auch wird die Testbarkeit des Webservices stark erhöht.

Neben der erwähnten Flexibilität und der Portabilität waren auch Stabilität und Robustheit wichtige Anforderungen an die Lösung. Damit Laufzeitfehler minimiert werden können, wird auf ein statisches Datenmodell gesetzt. Die gesamte vom Webservice benötigte Datenstruktur wird aus dem zu animierenden actifsource Modell in eine C-Quellcodedatei generiert und zusammen mit der Ausführungslogik kompiliert. Dadurch müssen zur Laufzeit keine Referenzen der Datenstruktur mehr verändert werden und es wird keine dynamische Allokation von Speicher auf dem Heap benötigt. Somit wird eine ganze Klasse von Fehlerquellen, die für hardwarenahe Programmierung typisch und erfahrungsgemäss schwer zu entdecken sind, ausgeschlossen.

7.2. Voraussetzungen für den produktiven Einsatz

Durch die Performancetests und die durchgeführte Integration in ein bestehendes actifsource Projekt konnte gezeigt werden, dass die entwickelte Lösung bereits zwecksmässig eingesetzt werden kann. Zudem ermöglichen die bereits umgesetzten Funktionalitäten die Diagnose von einfachen Steuerungen. Damit die Lösung jedoch in die actifsource Entwick-

lungsumgebung aufgenommen werden kann, müssen noch einige Aspekte verfeinert und weitere Funktionen umgesetzt werden.

Insbesondere könnte durch eine verbesserte Integration in die actifsource Entwicklungsumgebung, der Aufwand für die Konfiguration und das Deployment der Animationssoftware durch den Entwickler stark verringert werden. Zur Zeit sind noch einige manuelle Schritte notwendig, wie zum Beispiel das Kopieren der Webapplikation in das Webserver-Verzeichnis, die das Einbinden der Lösung erschweren. Durch eine vollständige Integration könnte beispielsweise die Clientanwendung an einem Stück direkt in das gewünschte Zielverzeichniss generiert werden.

Des Weiteren kann die für die Animation benötigte Konfiguration in actifsource noch zusätzlich unterteilt werden, sodass domänen- und plattformspezifische Optionen nicht in jedem Projekt definiert werden müssen. So könnten zum Beispiel die korrekten Animationsstrategien der verschiedenen Diagrammtypen einer Domäne durch das entsprechende Metamodell festgelegt werden.

Auch die möglichen Darstellungsklassen der Diagrammelemente könnten durch das Metamodell des zu generierenden Systems definiert werden und anschliessend zusammen mit den vom Client benötigten SVG-Dateien als CSS Klassen generiert werden.

7.3. Mögliche Folgearbeiten

Die umgesetzte Lösung könnte auch als Grundlage für eine Testumgebung von Modellen dienen. Diese könnte ähnlich der CIP-Test-Suite eine automatisierte oder manuelle Simulation des umgesetzten Modells ermöglichen.

Durch das Ausführen des Modells im Browser muss kein C-Code kompiliert werden, was bei grossen Modellen viel Zeit sparen kann. Zudem können die Tests während der Ausführung animiert werden und Fehler sind direkt im entsprechenden Diagramm darstellbar.

A. Tabellenverzeichnis

B. Abbildungsverzeichnis

C. Listings

D. Literatur

[BFM05] T. Berners-Lee, R. Fielding und L. Masinter. *RFC 3986, Uniform Resource Identifier (URI): Generic Syntax*. Hrsg. von Internet Engineering Task Force (IETF). Request For Comments (RFC). 2005. URL: `http://www.ietf.org/rfc/rfc3986.txt`.

[Cer69] V.G. Cerf. *ASCII format for network interchange*. RFC 20. Internet Engineering Task Force, Okt. 1969. URL: `http://www.ietf.org/rfc/rfc20.txt`.

[Cro06] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627 (Informational). Internet Engineering Task Force, Juli 2006. URL: `http://www.ietf.org/rfc/rfc4627.txt`.

[Fie00] Roy Thomas Fielding. „Architectural styles and the design of network-based software architectures“. Diss. University of California, 2000.

[Fie+99] R. Fielding u. a. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Proposed Standard). Internet Engineering Task Force, Juni 1999. URL: `http://www.ietf.org/rfc/rfc2616.txt` (besucht am 07. 04. 2013).

[Fie99] Hugo Fierz. *The CIP Method: Component- and Model-Based Construction of Embedded Systems*. 1999. URL: `http://www.actifsource.com/_downloads/thecipmethod.pdf` (besucht am 26. 02. 2013).

[FJ03] Jon Ferraiolo, Dean Jackson und ?? ? *Scalable Vector Graphics (SVG) 1.1 Specification*. Jan. 2003. URL: `http://www.w3.org/TR/2003/REC-SVG11-20030114/`.

[FM11] I. Fette und A. Melnikov. *The WebSocket Protocol*. RFC 6455 (Proposed Standard). Internet Engineering Task Force, Dez. 2011. URL: `http://www.ietf.org/rfc/rfc6455.txt` (besucht am 07. 04. 2013).

[Fow06] Martin Fowler. *Continuous Integration*. Mai 2006. URL: `http://martinfowler.com/articles/continuousIntegration.html`.

[Gam+95] Erich Gamma u. a. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[HHS03] Philippe Le Hegaret, Arnaud Le Hors und Johnny Stenback. *Document Object Model (DOM) Level 2 HTML Specification*. Jan. 2003. URL: `http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109`.

[Hil09] David Hill. *The ViewModel Pattern*. Jan. 2009. URL: `http://blogs.msdn.com/b/dphill/archive/2009/01/31/the-viewmodel-pattern.aspx`.

[ISO01] ISO/IEC. *ISO/IEC 9126. Software Entwicklung - Produkt Qualität*. ISO/IEC, 2001.

[ISO99] ISO. *ISO C Standard 1999*. Techn. Ber. ISO/IEC 9899:1999 draft. 1999. URL: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

[Kat13] Oz Katz. *Avoiding Common Backbone.js Pitfalls*. März 2013. URL: `http://ozkatz.github.io/avoiding-common-backbonejs-pitfalls.html`.

[Lab12] Pivotal Labs. *Jasmine Testing Framework*. Dezember 2012. URL: http://pivotal.github.io/jasmine/.

[Lea00] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. 2. Reading, MA: Addison-Wesley, 2000.

[Lor+11] S. Loreto u. a. *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*. RFC 6202 (Informational). Internet Engineering Task Force, Apr. 2011. URL: http://www.ietf.org/rfc/rfc6202.txt.

[Lyu13] Sergey Lyubka. *Mongoose User Manual*. Juni 2013. URL: https://github.com/valenok/mongoose/blob/master/UserManual.md.

[Mic13] Microsoft. *SVG Coordinate Transformations*. Mai 2013. URL: http://msdn.microsoft.com/en-us/library/ie/hh535760(v=vs.85).aspx.

[MIT88] MIT. *The MIT License (MIT)*. 1988. URL: http://opensource.org/licenses/MIT.

[Nel13] Darryl Nelson. *Next-Gen Web Architecture for the Cloud Era*. 2013. URL: http://www.sei.cmu.edu/library/assets/presentations/nelson-saturn2013.pdf (besucht am 29. 05. 2013).

[Nir] Nirvana. *Nirvana Realm Benchmarks*. URL: http://www.my-channels.com/developers/nirvana/concepts/scalability/benchmarks.html (besucht am 19. 03. 2013).

[OS13] Addy Osmani und Sindre Sorhus. *todoMVC - Helping you select an MV* framework*. Feb. 2013. URL: http://www.todomvc.com/.

[Por13] Sebastian Porto. *Angular VS Knockout VS Ember vs CanJS*. Apr. 2013. URL: http://jsperf.com/angular-vs-knockout-vs-ember/118 (besucht am 01. 05. 2013).

[Sim11] Simon. *CometD 2.4.0 WebSocket Benchmarks*. Sep. 2011. URL: http://webtide.intalio.com/2011/09/cometd-2-4-0-websocket-benchmarks/ (besucht am 19. 03. 2013).

[Sno13] Zach Snow. *AngularJS Headaches: Hanging End-to-End Scenarios*. 2013. URL: http://zachsnow.com/#!/blog/2013/angularjs-headache-hanging-end-end-scenarios/.

[Tea10] AngularJS Team. *Official AngularJS Website*. 2010. URL: http://angularjs.org/.

[Tea13] AngularJS Team. *Karma - Spectacular Test Runner for JavaScript*. Apr. 2013. URL: http://karma-runner.github.io/.

[Wal10] Colin Walls. *Dynamic Memory Allocation and Fragmentation in C and C++*. Techn. Ber. Mentor Graphics, 2010.

E. Glossary

ab Apache Bench. 82, IV

AJAX Asynchronous JavaScript and XML (AJAX) bezeichnet ein Konzept der asynchronen Datenübertragung zwischen einem Browser und dem Server.. 21

AR Von B&R Automation entwickelte Laufzeitumgebung für Echtzeitsysteme. 37, 38, 44, 57, 90, 92, IV

AS Automation Studio. 90–92

ASCII American Standard Code for Information Interchange; 7-Bit Zeichenkodierung bestehend aus 95 druckbaren Zeichen. 38

BDD Behaviour-driven Development. 69

C99 Standardisierter C Dialekt nach ISO/IEC 9899:1999. 57

CGI Common Gateway Interface. 33, 44

Channel Kommunikationskanal in einem CIP Modell. 12, 27, II

CIP Communicating Interacting Processes. 8, 9, 11–13, 16, 27–29, 31, 32, 71, 73, 88, 94

Cluster Sammlung von Prozessen in einem CIP Model. 11, 27

CSS Cascading Style Sheets. 66, 67

DOM Das Document Object Model (DOM) ist eine Schnittstelle, mit der Softwareprogramme auf den Inhalt von Dokumenten (einschlieSSlich Webseiten) zugreifen und deren Inhalt, Struktur und Stil aktualisieren können. Das DOM einer Webseite enthält alle Elemente, die auf der Webseite angezeigt werden, einschlieSSlich Inhalt, Links, Stildefinitionen und Skripts (kleine Programme). 47, 50, 53, 67

DRY Don't Repeat Yourself. 61

FF Mozilla Firefox. 21

FUB Funktionsblock. *Glossary:* Funktionsblock

Handlebars Handlebars ist eine JavaScript-Bibliothek, mit welcher HTML Vorlagen mit spezifizierten Platzhaltern zur Laufzeit evaluiert werden können. 48

Hype Cycle Der Hype Cycle stellt dar, welche Phasen der öffentlichen Aufmerksamkeit eine neue Technologie bei deren Einführung durchläuft. Phasen: Technologischer Auslöser, Gipfel der überzogenen Erwartungen, Tal der Enttäuschungen, Pfad der Erleuchtung und Plateau der Produktivität. 18, 24

IE Microsoft Internet Explorer. 21, 25

iframe Mit einem iframe oder auch inline Frame kann innerhalb einer HTML-Seite eine weitere integriert werden. 22, 24

Inpulse Internes Event in einem CIP Cluster, welches einen Zustandsübergang auslöst. 27

IPC Interprozesskommunikation. 35–37, 46

jQuery jQuery ist eine freie, umfangreiche JavaScript-Bibliothek, welche komfortable Funktionen zur DOM-Manipulation und -Navigation zur Verfügung stellt. 48

JSON JavaScript Object Notation. 19, 34, 39, 61, 69

MEP Message Exchange Pattern. 19, 24, 25

Message Externes Event in einem CIP Modell; Zustandsübergänge können zum einen Messages auslösen wie auch von selber von Messages ausgelöst werden. 27

MIME Multipurpose Internet Mail Extensions. 39

Mongoose Leichtgewichtiger HTTP-Webserver; in C geschrieben und auf allen gängigen Desktop Betriebssystemen ausführbar (https://github.com/valenok/mongoose). 56, 62

MVC Model View Controller. 47–49

Outpulse Internes Event in einem CIP Cluster, welches von einem Zustandsübergang ausgelöst wird. 27

Pulse Internes Event in einem CIP Cluster. 11

REST Representational State Transfer (REST bzw. RESTful) bezeichnet ein Programmierparadigma für Webanwendungen. REST beschreibt die Idee, dass eine URL genau einen Seiteninhalt als Ergebnis einer serverseitigen Aktion (etwa das Anzeigen einer Trefferliste nach einer Suche) darstellt, wie es der Standard HTTP für statische Inhalte bereits vorsieht. 32, 38, 48

Same Origin Policy Die Same-Origin-Policy ist ein Sicherheitskonzept, welches clientseitigen Skriptsprachen wie JavaScript und ActionScript, aber auch Cascading Style Sheets untersagt, auf Objekte (zum Beispiel Grafiken) zuzugreifen, wenn sie von einer anderen Webseite stammen oder dessen Speicherort nicht der Origin entsprechen. 20, 23–25

Single Board Computer Vollwertiger Computer bei dem alle notwendigen Komponenten auf einer Platine angebracht sind; oft Kreditkartengross; bekannteste Vertreter sind Raspberry Pi und Arduino. 36

SMIL Die Synchronized Multimedia Integration Language (SMIL) ist ein auf XML basierender, von dem World Wide Web Consortium (W3C) entwickelter Standard für eine Auszeichnungssprache für zeitsynchronisierte, multimediale Inhalte. 66

SOAP SOAP ist ein Netzwerkprotokoll, mit dessen Hilfe Daten zwischen Systemen ausgetauscht und Remote Procedure Calls durchgeführt werden können. 22

SOFEA Serviceorientierte Frontend Architektur. 32

Solution Wiederverwendbare Problemdomäne in actisource, umfasst das Metamodell, Definitionen von Diagrammtypen sowie Codetemplates; z.B. die CIP Solution von actifsource. 34, 73

SPA Eine Single Page Application (SPA) ist eine Webanwendung, die keinen Seitenwechsel durchführt, sondern die Anzeige nur durch Austausch von Seitenelementen via JavaScript/DOM verändert. 48, 49

SPS Speicherprogrammierbare Steuerung. 9, 14, 77, 88, 91

SVG Scalable Vector Graphics. 13, 29, 34, 54, 66–68

TDD Test-Driven Development. 49

URI Uniform Resource Identifier. 39, 47, 49, 53

UUID Universally Unique Identifier. 13, 28, 51, 58

WebSocket Auf TCP basierendes, bidirektionales Protokoll für die Kommunikation zwischen Webanwendung und Server. 25

WSDL Die Web Services Description Language (WSDL) ist eine plattform-, programmiersprachen- und protokollunabhängige Beschreibungssprache für Netzwerkdienste (Webservices) zum Austausch von Nachrichten auf Basis von XML.. 22

XHR XMLHttpRequest (XHR) ist eine API zum Transfer von beliebigen Daten über das Protokoll HTTP. 21–23, II

XML Extensible Markup Language. 39