technische universität
dortmund

Bachelorarbeit

Planung und Realisierung einer modularen Applikation zur Verwaltung von Steuer- und Messvorschriften für eine QNX-basierte Echtzeitumgebung

Daniel Knobe

27. September 2011

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Dr. Stefan Siepmann

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 27. September 2011

_____

Daniel Knobe

Zusammenfassung

Diese Bachelorarbeit befasst sich mit der Planung und Realisierung einer modularen Applikation zur Verwaltung von Steuer- und Messvorschriften. Bei der Firma Miele & Cie. KG werden neue Produkte vor der Markteinführung in verschiedenen Laboren untersucht. Im Labor des Dauerversuchs werden Geräte auf ihre Langlebigkeit überprüft. Der Prüfungsablauf wird durch die sogenannten Steuer- und Messvorschriften beschrieben, welche gegenwärtig manuell in PEARL90 programmiert werden.
In dieser Bachelorarbeit wird ein modellgetriebenes Konzept entwickelt, das die grafische Erstellung und Bearbeitung von Steuer- und Messvorschriften ermöglicht. Das entworfene Konzept wird dann in einer Applikation, dem SMVEditor, implementiert. Abschließend wird das entworfene Konzept und die Implementierung evaluiert und ein Fazit gezogen.

Inhaltsverzeichnis

1. Einleitung

Bei der Firma Miele & Cie. KG werden innerbetrieblich 20 Labore betrieben, in denen sich in der Entwicklung befindende Geräte oder Gerätekomponenten getestet und überprüft werden. Die verschiedenen Konstruktions- und Entwicklungsabteilungen stellen dabei immer neue Anforderungen an die Labore, die vorhandene Prüfungen anpassen müssen. Die Abläufe der Prüfungen werden durch Steuer- und Messvorschriften (SMV) technisch realisiert.

1.1. Problemstellung

Diese Arbeit befasst sich mit der Planung und Realisierung einer modularen Applikation zur Verwaltung von Steuer- und Messvorschriften.
Diese werden derzeit in der Programmiersprache PEARL90 entwickelt. Dabei besteht der aktuelle Entwicklungsprozess aus der Anpassung bereits vorhandener SMVs. Diese werden anschließend in einer neuen Datei abspeichert. In 12 Jahren sind so insgesamt 265 unterschiedliche Steuer- und Messvorschriften zur Überprüfung von Geräten und Gerätekomponenten entstanden. Als problematisch stellte sich dabei die Verwaltung der SMVs heraus, da die gemeinsame Codebasis nicht in externe Dateien ausgelagert wurde. Somit sind Fehlerkorrekturen und Anpassungen nicht in jeder Steuer- und Messvorschrift integriert worden, weil dies mit einem hohen Zeitaufwand verbunden wäre. Die Folgen des Entwicklungsmodells sind, trotz der prinzipiell ähnlichen Funktionalität der SMVs, die wachsenden Unterschiede zwischen den Vorschriften und die damit nicht mehr gegebene Wartbarkeit. Zudem stellt die Entwicklung neuer Prüfungen bereits bei kleineren Änderungen ein Problem dar, da die Übersicht in den durchschnittlich mehr als 5300 Zeilen Quelltext mangelhaft ist und das Auffinden der zu ändernden Codestellen einen großen Aufwand bedeutet.

1.2. Zielsetzung

Das Ziel der Bachelorarbeit ist die Konzipierung und Implementierung eines alternativen Ansatzes zur Entwicklung von Steuer- und Messvorschriften, welcher die Prüfungsentwicklung in PEARL90 ersetzen soll. Aufgabe ist die Entwicklung der SMVs durch eine

erweiterbare grafische Sprache zu ermöglichen, deren Schwerpunkt auf der einheitlichen Modellierung der Prüfungsabläufe liegt. Dies steigert die Übersicht und ermöglicht somit eine schnelle Anpassung von Prüfungen. Zudem werden die geschilderten Probleme aus Abschnitt 1.1, wie z.B. die schlechte Wartbarkeit und die fehlende Kapselung, vermieden.

1.3. Gliederung

Der folgende Abschnitt gibt einen kurzen Abriss über die Gliederung und die Struktur dieser Arbeit. Zunächst wird in Kapitel 2 der Bereich Dauerversuch der Firma Miele & Cie. KG vorgestellt. Anschließend werden in den Kapiteln 3 und 4 die Anforderungen für die zu entwickelnde Applikation beschrieben und ein darauf zugeschnittenes Konzept entwickelt. Die auf dem Konzept basierende Implementierung und die dafür verwendeten Technologien werden in Kapitel 5 dargestellt. In Kapitel 6 wird die Implementierung evaluiert und abschließend in Kapitel 7 ein Fazit der Arbeit gezogen.

2. Dauerversuch

Im Werk Gütersloh erfolgt die Durchführung des Dauerversuchs. Dabei werden die Geräte und Gerätekomponenten, die sogenannten Prüflinge, auf Langlebigkeit untersucht. Die Inspektion eines Prüflings erfolgt dabei an einem dafür speziell eingerichteten Prüfplatz. Es stehen ungefähr 189 Prüfplätze für den Dauerversuch zur Verfügung. An diesen Prüfplätzen wird je ein Prüfling ununterbrochen 7500 Stunden überprüft. In Abbildung 2.1

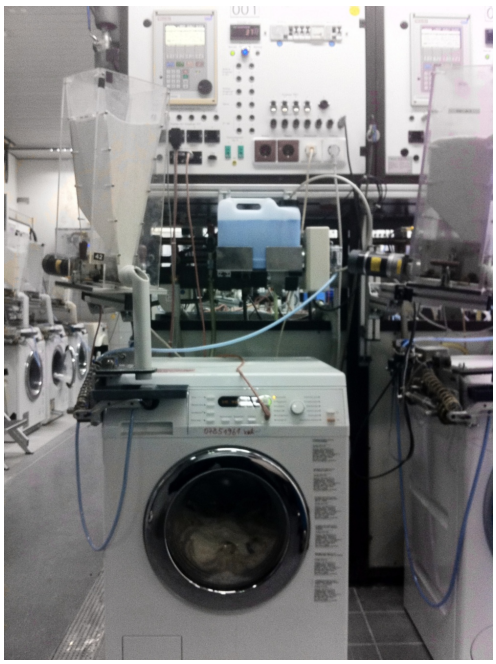


Abbildung 2.1.: Prüfplatz mit Prüfling

ist exemplarisch ein Prüfplatz mit angeschlossenem Prüfling dargestellt. Im Folgenden wird in den Abschnitten 2.1 und 2.2 die Beschreibung der Hard- und Softwarearchitektur der Prüfplätze bei der Firma Miele beschrieben.

2.1. Hardware

Die in Abbildung 2.2 dargestellte Prüfplatzhardware setzt sich aus einem 200MHz PowerPC mit 16MB Arbeitsspeicher und den erforderlichen Schnittstellen zur Kommunikation mit dem Prüfling zusammen. Jeder Prüfplatz besitzt analoge und digitale Schnittstellen zur Steuerung von Aktuatoren. Sie dienen zur physikalischen Interaktion mit den Prüflingen, z.B. zur Zuführung von Waschmittel, exemplarisch dargestellt in Abbildung 2.1. Die optische Schnittstelle des Prüfplatzes dient zur direkten Kommunikation mit dem Prüfling. Zusätzlich besitzt der Prüfplatz eine Mensch-Maschine-Schnittstelle, welche z.B. Informationen über die Betriebsstunden und Drehzahl des Prüflings ausgibt. Während einer Überprüfung werden fortlaufend Daten erhoben. Zu diesem Zweck ist jeder Prüfplatz mit verschiedenen Sensoren ausgestattet, wie z.B mit Thermoelementen (PT-100 und NiCrNi) und Leistungsmessern. Zur Kommunikation mit externen Geräten dient der CAN-Bus.

Abbildung 2.2.: Prüfplatz / Frontansicht

2.2. Software

In der Firma Miele wird an einer Umstellung der Software gearbeitet. Aktuell wird die in Abbildung 2.3 skizzierte Architektur genutzt, welche dreischichtig aufgebaut ist. Die

Grundlage stellt dabei das speicherschonende Echtzeitbetriebssystem RTOS-UH (Kurzform für Realtime Operating System - University Hannover) dar, welches keinen Speicherschutz besitzt. Um Code auszuführen, wird beim RTOS-UH eine Task gestartet, die globalen Speicher nutzt und selber auf diesem ausgeführt wird.
Auf dem Betriebssystem setzt die Prüfplatzsoftware (PPS) auf. Diese liegt als Schicht über dem Betriebssystem und stellt in 33 Modulen verschiedene Schnittstellen zur Steuerung der Prüfplatzhardware und zur Interaktion mit dem Benutzer zur Verfügung.
Im Zentrum dieser Arbeit stehen die Steuer- und Messvorschriften, welche den Ablauf einer jeden Prüfung beschreiben. Diese Vorschriften werden zur Laufzeit auf den Prüfplatz geladen und ausgeführt. Im Rahmen der Umstellung bei der Prüfplatzsoftware wird das RTOS-UH durch QNX Neutrino der Firma QNX Software Systems ausgetauscht. Dabei handelt es sich um ein POSIX-kompatibles Echtzeitbetriebssystem mit einem Mikrokernel auf dem aus Threads bestehende Prozesse in geschützten Speicherbereichen ausgeführt werden und somit vor Zugriffen von anderen Prozessen geschützt sind. Diese Eigenschaft ist der zentrale Grund für die Umstellung des Betriebssystems, weil in der aktuellen Realisierung auf Basis des RTOS-UH die Steuer- und Messvorschriften gelegentlich Fehler verursachen, welche das gesamte System negativ beeinflussen können. Im Rahmen dieser Systemumstrukturierung müssen große Teile des Systems angepasst

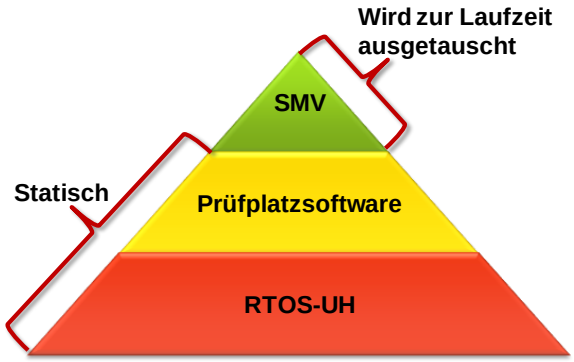


Abbildung 2.3.: Alte Softwarearchitektur

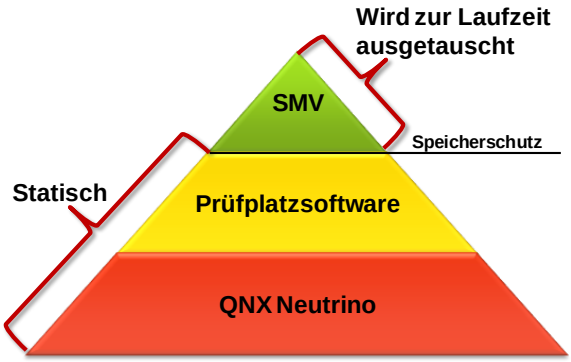


Abbildung 2.4.: Neue Softwarearchitektur

werden. Infolgedessen hat das Unternehmen beschlossen, auch die aktuell genutzte Programmiersprache PEARL90 durch C/C++ zu ersetzen, weil diese einen umfangreicheren Pool an Bibliotheken bietet. Derzeit werden die Steuer- und Messvorschriften noch in PEARL90 entwickelt. Diese sollen in Zukunft grafisch modelliert werden. Durch die grafische Modellierung soll das Erstellen, Bearbeiten und Pflegen der SMV vereinfacht werden. Die grafische Ansicht ermöglicht einen Überblick über die einzelnen in der SMV enthaltenen Funktionsbereiche.

3. Anforderungen

In diesem Kapitel werden alle Anforderungen beschrieben, die dem Entwurf in Kapitel 4 zugrunde gelegt werden. Aufgabe ist es, eine Applikation zu entwickeln, welche die händische Programmierung von Steuer- und Messvorschriften ablöst und grafisch das strukturierte Vorgehen zur Entwicklung von Prüfungen unterstützt.

In Abschnitt 3.1 werden alle Anforderungen an die grafische Modellierung von Steuer- und Messvorschriften formuliert. Im Anschluss daran werden im Abschnitt 3.2 die technischen Anforderungen für die in Kapitel 5 beschriebene Implementierung festgelegt.

3.1. Grafische Modellierung

Die zu entwickelnde Applikation soll das grafische Modellieren von Steuer- und Messvorschriften ermöglichen. Dies ist die zentrale Forderung an das zu entwickelnde Konzept seitens der Firma Miele. Die grafische Sprache soll dabei folgende Anforderungen erfüllen.

- **Modellierungsdesign:** Die zu entwickelnde grafische Modellierung soll keine vollständige Neuentwicklung sein, sondern sich eine bereits vorhandene Modellierung zum Vorbild nehmen. Dies können z.B. Entity-Relationship-Diagramme, UML-Modelle oder Prozessketten sein. Der primäre Grund für diese Entscheidung ist die dadurch erhöhte Lesbarkeit für Benutzer, die bereits den Ursprungsdiagrammtyp kennen.

- **Abstraktion:** Die Modellierung von Steuer- und Messvorschriften soll abstrakter, als bei der händischen Programmierung mit PEARL90 realisiert, gestaltet werden. Die zu modellierenden Abläufe bestehen in den aktuellen Steuer- und Messvorschriften aus Schleifen mit konstanter Wiederholungsanzahl und bedingten Anweisungen ohne Alternativpfad. Die grafische Darstellung soll diese Konstrukte erlauben, jedoch keine darüber hinausgehenden Möglichkeiten für die Strukturierung von Steuer- und Messvorschriften bieten, weil diese Fehlerquellen bergen können.

- **Hierarchie:** Eine weitere Anforderung ist es, die Modellierung von Hierarchie zu ermöglichen, die in den aktuellen Steuer- und Messvorschriften durch Anweisungen

in PEARL90 möglich ist, jedoch an keiner Stelle genutzt wird. Durch das Einfließen dieses Konzepts als zentrale Anforderung sollen Redundanzen vermindert und Modellabschnitte gekapselt werden. Infolgedessen soll der Wartungsaufwand reduziert und die Übersicht erhöht werden, da „der Mensch [...] im Allgemeinen nicht in der Lage [ist], Systeme zu verstehen, die viele Objekte (wie z.B. Zustände, Komponenten) enthalten, die in komplexem Zusammenhang miteinander stehen. Die Einführung von Hierarchie-Ebenen ist der einzige Weg, dieses Problem zu lösen“ [1, S. 13].

- **Parallelität:** In den aktuellen Steuer- und Messvorschriften finden parallele Abläufe statt. Oft beinhalten die SMVs aber noch sequenzielle Programmabschnitte, welche unabhängige, aufeinander wartende Anweisungen enthalten, was zu einer reduzierten Ausführungsgeschwindigkeit der Prüfung führt. Durch die grafische Modellierung von Steuer- und Messvorschriften soll die Darstellung von Parallelität für den Prüfungsentwickler intuitiver gestaltet werden.
- **Modularität:** Ein wichtiger Aspekt ist der geforderte modulare Aufbau der neuen Vorschriften. Dieser modulare Aufbau soll zur flexiblen Funktionserweiterung von SMVs dienen. Zudem unterstützt dieser Aufbau die geplante Aufgabenteilung in dem Entwicklungsprozess von Steuer- und Messvorschriften, weil es zwei Entwicklungsabteilungen mit unterschiedlichen Rollen gibt.
  - **Grundlagenentwickler:** Die Grundlagenentwickler erweitern und ergänzen den SMVEditor, dessen Implementierung in Kapitel 5 beschrieben wird. Zudem sollen die modularen Komponenten der Applikation von dieser Entwicklergruppe implementiert werden.
  - **Prüfungsentwickler:** Die Prüfungsentwickler nutzen den SMVEditor und die modularen Komponenten um Prüfungen zu entwickeln.

3.2. Applikation

In diesem Abschnitt werden die Anforderungen an die zu entwickelnde Applikation festgelegt. Im Folgenden wird eine Liste mit allen wichtigen Punkten angeführt.

- **Modellierung:** Die zu entwickelnde Applikation soll es ermöglichen, Steuer- und Messvorschriften zu erstellen und zu bearbeiten. Dies soll mit einer grafischen Sprache durchgeführt werden, welche die Anforderungen aus Abschnitt 3.1 erfüllt.
- **Benutzbarkeit:** Die Anwendung soll eine einfache Bedienung ermöglichen, welche den Benutzer aufgrund des intuitiven Aufbaus bei der Entwicklung von SMVs unterstützt.

- **Lesbarer Code:** Das Werkzeug soll die modellierten Steuer- und Messvorschriften in Quellcode umwandeln können. Es empfiehlt sich, dass der Quellcode dabei eine einheitliche Formatierung besitzt und somit gut lesbar ist.
- **Lauffähige SMV:** Es soll möglich sein, die modellierten Steuer- und Messvorschriften auf den Prüfplätzen (mit QNX Neutrino) auszuführen, d.h. es soll ausführbarer Code generiert werden, der zudem eine einheitliche Architektur aufweisen soll.
- **Plattformunabhängigkeit:** Aufgrund der Miele IT-Infrastruktur, welche sich aus Windows- und Linuxdesktopsystemen zusammensetzt, soll es möglich sein, die zu entwickelnde Applikation auf beiden Betriebssystemen nutzen zu können.
- **Versionierbarkeit:** Die Applikation soll das Speichern der SMV-Beschreibung in einer versionierbaren Form ermöglichen.

4. Entwurf

In diesem Kapitel wird der Entwurf entwickelt, welcher die Anforderungen aus Kapitel 3 erfüllt. Es wird ein modellgetriebener Ansatz verfolgt, eine Technik, die im Abschnitt 4.1 beschrieben und motiviert wird. Gemäß dem modellgetriebenen Paradigma wird anschließend im Abschnitt 4.2 das Metamodell entworfen. Im Abschnitt 4.3 folgt die Festlegung der domänenspezifischen Sprache, welche die konkrete Syntax zur Modellierung von Steuer- und Messvorschriften definiert.

4.1. Modellgetriebene Softwareentwicklung

„Modellgetriebene Softwareentwicklung (MDSD - Model-Driven Software Development) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen“ [2, S. 11]. Das formale Modell beschränkt sich dabei auf die relevanten Aspekte für die zu erstellende Software, die aber vollständig beschreibbar sein muss. Es dient nicht, wie bei der traditionellen modellbasierten Softwareentwicklung,

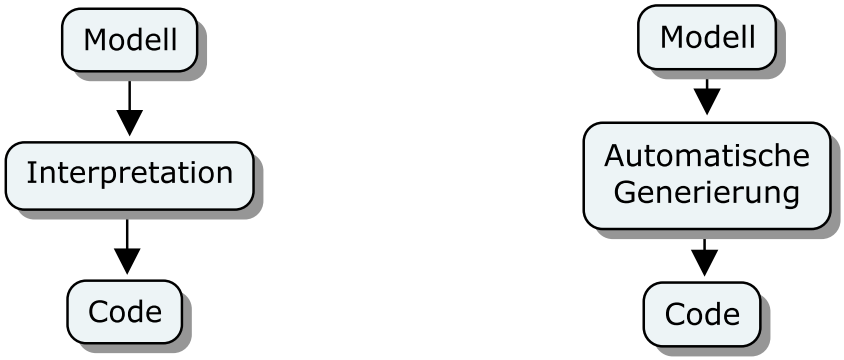


Abbildung 4.1.: Modellbasierter Ansatz (links) und modellgetriebener Ansatz (rechts)

ausschließlich zur Dokumentation und zum Entwurf von Softwaresystemen, welche manuell implementiert werden, sondern als Beschreibung, die zur automatisierten Codegenerierung genutzt wird. Durch den modellgetriebenen Ansatz für Steuer- und Messvorschriften ist somit die Generierung von lauffähigen Prüfungen möglich. Als Grundlage

dient ein Metamodell aus dem Instanzen durch eine speziell für diesen Anwendungsfall entwickelte domänenspezifische Sprache erzeugt werden. Diese Instanzen können dann durch Transformationen, welche „Modelle auf die jeweils nächste Ebene, weitere Modelle oder Sourcecode“ [3, S. 3] abbilden, und mittels Compiler zu ausführbaren Programmen umgewandelt werden.

4.2. Metamodell

Das Metamodell ist die zentrale Idee bei einem modellgetriebenen Ansatz. Aufgabe und Ziel des Metamodells ist es eine Struktur bzw. eine abstrakte Syntax vorzugeben, welche den Prüfungsablauf unter Berücksichtigung der zuvor festgelegten Anforderungen vollständig beschreibt. Außerdem beschreibt das Metamodell die statische Semantik, die festlegt, ob eine Metamodellinstanz valide ist. Das Ziel beim Entwurf des Metamodells ist eine möglichst gute Abbildung für die Domäne zu erzeugen und potenzielle Fehlerquellen und Komplexität zu reduzieren. Eine SMV stellt eine Instanz des Metamodells dar und spezifiziert eine mögliche Prüfung. Sie beschreibt einen Ablauf, welcher aufein-

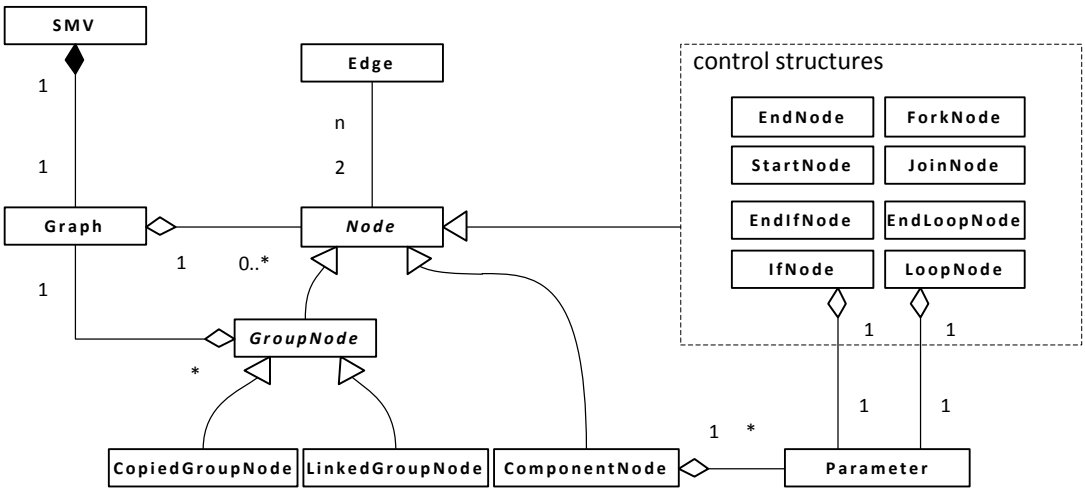


Abbildung 4.2.: Metamodell für Steuer- und Messvorschriften

anderfolgende Aktionen durchführt. Aus diesem Grund bietet es sich an einen gerichteten bzw. orientierten azyklischen Graphen dem Metamodell zugrunde zu legen, der aus Knoten besteht, welche durch gerichtete Kanten in Relation gestellt werden. Durch diese Repräsentation kann die Abfolge der Aktionen eines Testablaufs problemlos modelliert

werden. Das vollständige Metamodell wird in Abbildung 4.2 als UML-Klassendiagramm veranschaulicht, welches im Folgenden detailliert erläutert wird. Wie bereits erwähnt, wird eine SMV durch einen Graphen repräsentiert. Dieser bildet die Grundlage für eine Ablaufbeschreibung und ist deswegen untrennbar mit einer SMV verknüpft. Der Graph besteht aus einer beliebigen Anzahl von Knoten, die durch Kanten verbunden sind. Dabei wird die Anzahl der Kanten durch die jeweilige Knotenausprägung festgelegt. Diese Knoten unterteilen sich in drei Kategorien, den Kontrollknoten, den Gruppenknoten und den Komponentenknoten, die in den Abschnitten 4.2.1, 4.2.2 und 4.2.3 beschrieben werden.

4.2.1. Kontrollknoten

Die aus der Analyse hervorgegangenen benötigten Kontrollstrukturen sind Spezialisierungen des abstrakten Knotens. Sie schaffen einen beschränkten Raum zur Steuerung des Kontrollflusses, erlauben aber keine überflüssigen Funktionalitäten, die mögliche Fehlerquellen bergen. Die Kontrollknoten werden durch das Metamodell festgelegt. Zwei Kontrollknoten bilden jeweils ein Kontrollknotenpaar und realisieren zusammen eine Operation, welche den Kontrollfluss manipuliert. Im Folgenden wird die gesamte Liste der Kontrollknotenpaare beschrieben.

- **Start- und Endknotenpaar:** Der Start- bzw. Endknoten repräsentiert den Beginn bzw. das Ende einer Prüfung. Aufgrund dessen besitzt der Startknoten keine eingehende aber eine ausgehende Kante und der Endknoten keine ausgehende aber eine eingehende Kante. Der Startknoten darf nur einmal im Graphen vorkommen, da sonst der Beginn der Prüfung nicht mehr eindeutig ist. Es sind keine Parameter vorgesehen um die Knoten zu konfigurieren.

- **Schleifenknotenpaar:** Eine Schleife setzt sich aus einem Schleifenstart- und einem Schleifenendeknoten zusammen. Diese beiden Knoten umschließen einen zusammenhängenden Teilgraphen vollständig und erlauben das Festlegen von endlich vielen Wiederholungen. Da eine Festlegung der Wiederholungen zur Laufzeit nicht benötigt wird und außerdem die potenzielle Gefahr von Endlosschleifen besteht, wird nur eine konstante Anzahl von Durchläufen erlaubt. Dieser Parameter wird im Schleifenstartknoten gesetzt. Beide Knoten besitzen jeweils eine eingehende und eine ausgehende Kante.

- **Knotenpaar für bedingte Anweisungen:** In dem hier entworfenen Modell ist es möglich bedingte Teilabläufe durchzuführen. Dies wird benötigt um auf eventuelle unerwünschte Situationen, wie z.B. das nicht Erreichen einer Motordrehzahl,

reagieren zu können. Vor diesem Hintergrund gibt es einen Wenn- und Wennendeknoten. Diese umschließen einen Teilgraphen vollständig, welcher genau dann ausgeführt wird, wenn der gesetzte Parameter im Wennknoten wahr ist. Beide Knoten besitzen jeweils eine eingehende und eine ausgehende Kante.

- **Threadknotenpaar:** In dem konzipierten Metamodell wird die Möglichkeit abgebildet, Abläufe zu parallelisieren und zu synchronisieren. Dazu werden zwei Knoten im Metamodell modelliert, der Gabelknoten mit einer eingehenden Kante und zwei ausgehenden Kanten und ein Synchronisationsknoten mit zwei eingehenden Kanten sowie einer ausgehenden Kante. Dabei wird vorausgesetzt, dass die zwei umschlossenen Teilgraphen vollständig von den beiden Knoten umschlossen werden. Die Threadknoten benötigen keine Parameter.

Das vollständige Umschließen eines bzw. zweier Teilgraphen durch ein Kontrollknotenpaar ist eine notwendige Bedingung für die Validität des Metamodells. Bei einer Verletzung der Bedingung könnte u.a. der Codegenerator keinen validen Code mehr generieren, weil es möglich wäre, dass der Kontrollfluss den Endknoten des Paares nie erreicht oder der Endknoten nicht eindeutig bzw. keinem Startknoten zuzuordnen ist. Abbildung 4.3

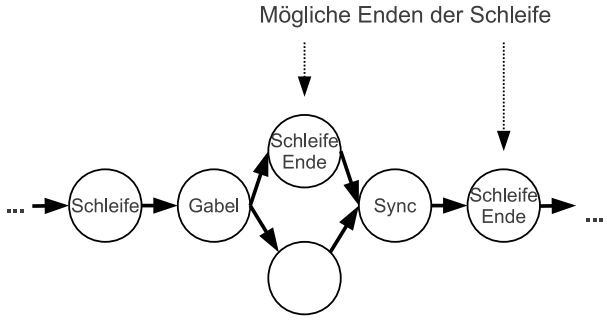


Abbildung 4.3.: Verletzung der Teilgraphenbedingung

zeigt dieses Problem exemplarisch am Schleifenknotenpaar, welches durch die Verletzung der Bedingung kein eindeutig definiertes Schleifenende hat und somit der Wirkungsbereich der Schleife nicht definiert ist.
Eine Ausnahmebehandlung (engl. Exception Handling) für kritische Fehler wird nicht benötigt, weil Ausnahmen nur durch Komponentenknoten verursacht werden können. Diese kümmern sich jedoch intern um Ausnahmen und überlassen das weitere Vorgehen, wie z.B. einen Abbruch der Prüfung oder Fortfahren der Prüfung, der Prüfplatzsoftware. Diese Behandlung ist demnach nicht mehr Teil der Steuer- und Messvorschrift.

4.2.2. Gruppenknoten

Im Metamodell wird das in den Anforderungen festgelegte Konzept der Hierarchie durch Gruppenknoten modelliert. Die Hierarchie erlaubt das Einbetten eines Graphen durch

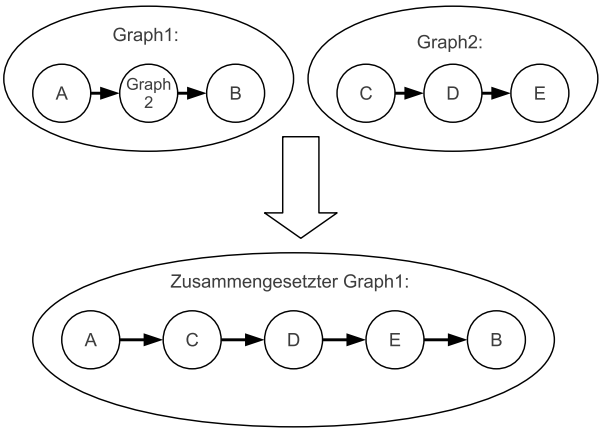


Abbildung 4.4.: Interpretation von Gruppenknoten

einen Gruppenknoten in den zu modellierenden Graphen, exemplarisch dargestellt in Abbildung 4.4. Es gibt zwei Arten von Gruppenknoten, die im Folgenden erläutert werden.

- **Kopiegruppenknoten:** Dieser Knoten ermöglicht das gegenseitige Verschachteln der Graphen von Steuer- und Messvorschriften auf Basis von Kopien. Beim Einfügen eines Gruppenknotens, der auf einen Graphen verweist, wird dieser Graph als Kopie in den Gruppenknoten abgelegt. Bei Veränderungen des Inhalts des inkludierten Graphen wird der Inhalt der Quelle nicht verändert. Umgekehrt würde eine Veränderung der Quelle sich auch nicht auf den Inhalt des Kopiegruppenknotens auswirken.

- **Linkgruppenknoten:** Anders verhält es sich mit dem Linkgruppenknoten. Dieser arbeitet nicht mit Kopien, sondern verknüpft die jeweiligen Graphen miteinander. Somit hat eine Änderung im Gruppenknoten direkte Auswirkungen auf den Graphen der Quelle und umgekehrt.

Gruppenknoten und die von ihnen repräsentierten Graphen dürfen nicht transitiv auf sich selbst verweisen, da dieses zu einer Kreisbildung führen würde und eine endlose Verschachtlung der Graphen zufolge hätte. Dies ist eine notwendige Bedingung für die Validität der Metamodellinstanz. Abbildung 4.5 verdeutlicht die Kreisbildung exemplarisch.

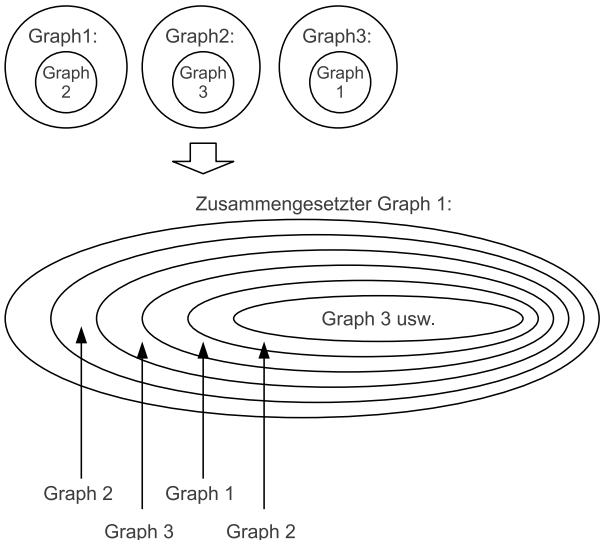


Abbildung 4.5.: Verbotenes Verschachteln von Gruppenknoten

4.2.3. Komponentenknoten

Das Ausführen von Anweisungen bzw. Aktionen, die keine Kontroll- bzw. Gruppierungsanweisungen sind, wird durch Komponentenknoten ermöglicht. Die Implementierung der Komponentenknoten wird durch Module, von denen beliebig viele geladen werden können, bereitgestellt, welche jeweils beliebig viele Komponentenknoten enthalten. Die Module erfüllen somit die in den Anforderungen geforderte Modularität der Modellierung von Steuer- und Messvorschriften. Zudem kann ein Modul beliebig viele Variablen zur Parametrierung von Knoten anbieten.
Komponentenknoten können eine beliebige Anzahl von Parametern besitzen, wodurch sie flexibel anpassbar bleiben, ohne dass eine Änderung des Moduls nötig wird. Das Metamodell legt fest, dass jeder Komponentenknoten eine eingehende und eine ausgehende Kante besitzt, wodurch keine weiteren notwendigen Bedingungen für die Validierung erforderlich sind.

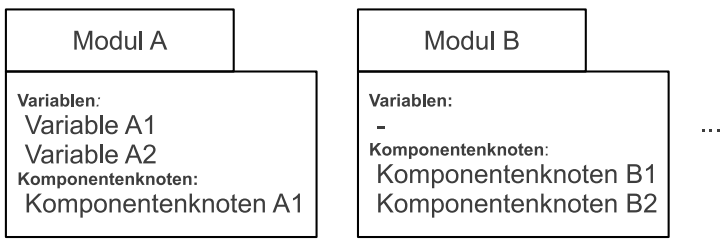


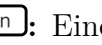
Abbildung 4.6.: Module mit Variablen und Komponentenknoten

4.3. Domänenspezifische Sprache

Ein entscheidender Aspekt für die spätere Erstellung und Bearbeitung von Steuer- und Messvorschriften ist das Festlegen einer passenden domänenspezifischen Sprache (DSL - Domain-Specific Language). Eine domänenspezifische Sprache ist eine textuelle oder eine grafische Sprache, welche für bestimmte Anforderungen entworfen wird, um eine Instanz des Metamodells zu erzeugen. Ein primäres Ziel bei dem Entwurf der zu entwickelnden Anwendung zur Verwaltung von Steuer- und Messvorschriften ist die Steigerung der Verständlichkeit und visuellen Strukturierung der Prüfungsabläufe, welche nur für die Prüfung relevante Aspekte darstellt. Dieses geht einher mit der Festlegung der Firma Miele, dass die DSL grafisch sein muss. Aus dem genannten Grund wird in diesem Abschnitt eine grafische Lösung entworfen. Weil eine SMV einen Prüfungsablauf beschreibt und aus einfachen Kontrollstrukturen besteht, bietet es sich an, bereits vorhandene grafische Sprachen auf Tauglichkeit zu überprüfen und gegebenenfalls anzupassen. Dies ist auch in den Anforderungen als wünschenswert deklariert worden. Die Unified Modeling Language (UML) bietet für diesen Aspekt der Modellierung den Bereich der Verhaltensdiagramme, von denen das Aktivitätsdiagramm am besten geeignet ist, weil der Schwerpunkt des Diagramms auf dem Ablauf nicht reaktiver Systeme liegt [4].

4.3.1. UML-Aktivitätsdiagramm

In diesem Unterabschnitt wird das UML-Aktivitätsdiagramm (Version 2.x) basierend auf [5] eingeführt. Die Beschreibung beschränkt sich dabei auf den Rahmen der für die DSL benötigten Elemente. Die umfassende Semantik und Syntax des UML-Aktivitätsdiagramms kann in der Spezifikation der Object Managment Group (OMG) nachgeschlagen werden [6]. Aktivitätsdiagramme dienen zur Modellierung von Systemverhalten. Es ist z.B. möglich umfangreiche Abläufe eines Systems grafisch zu beschreiben, aber auch Vorschriften von Algorithmen zu visualisieren. Im Folgenden werden die für die DSL relevanten Elemente des Aktivitätsdiagramms anhand der Abbildung 4.7 eingeführt und erläutert.

- **Aktion** 

: Eine Aktion stellt die kleinste ausführbare Einheit dar. Dabei können Aktionen beliebig komplexe Funktionalitäten haben.

- **Kontrollfluss**  **:** Der Kontrollfluss wird durch eine gerichtete Kante zwischen zwei Aktivitätsknoten dargestellt. Durch diese Kante wird die Ausführungsreihenfolge festgelegt.

- **Aktivitäten**  **:** Aktivitäten dienen zur Gruppierung von Aktivitätsknoten, wodurch Funktionalitäten zusammengefasst werden.

- **Entscheidung und Verbindung ◇:** Mit Entscheidungs- und Verbindungsknoten ist es möglich Schleifen und alternative Abläufe zu modellieren.
- **Gabelung und Vereinigung ▮:** Mit den beiden Elementen ist es möglich parallele Abläufe zu definieren.
- **Start- und Endknoten ● ◉:** Der Startknoten stellt den Beginn des Aktivitätsdiagramms dar und der Endknoten dessen Ende.

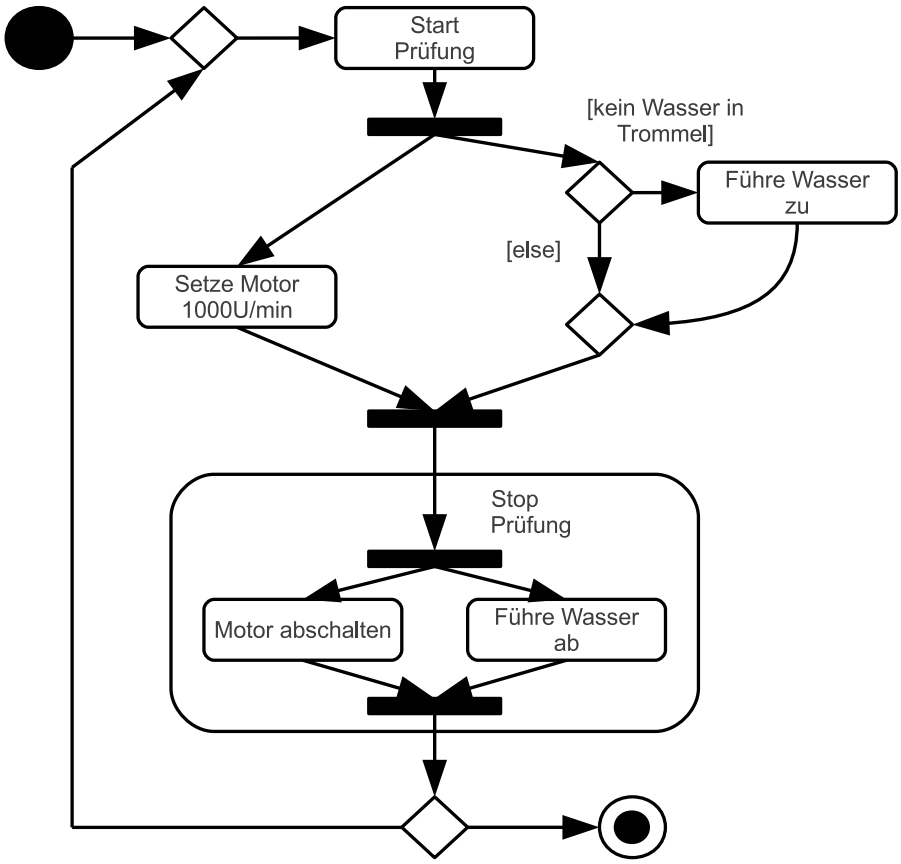


Abbildung 4.7.: Beispiel für ein Aktivitätsdiagramm

4.3.2. Grafische Anpassung und Abbildung

Auf Basis der Vereinfachung des UML-Aktivitätsdiagramms wird im Folgenden die DSL entworfen und mit dem Metamodell verknüpft. Dies ist ein legitimes Vorgehen nach [2, S. 101], da die UML „nicht einfach genug [ist], um effektiv mit einer grafischen Syntax arbeiten zu können. Erst das Weglassen der Repräsentation einer Vielzahl dieser

Konzepte innerhalb der Diagramme macht UML-Diagramme mehr oder weniger leserlich“. Die grafische Anpassung ermöglicht das Ausrichten der Elemente in einem Raster und macht jedes Element eindeutig identifizierbar. Zudem wird die Schwarz-Weiß Darstellung durch Farben ergänzt, wodurch die Übersicht erhöht werden soll.

- **Aktion :** Der Aktionsknoten wird grafisch durch einen gelben Kreis ersetzt und repräsentiert den Komponentenknoten des Metamodells. Für die Identifikation der genutzten Komponente wird der Aktionsknoten mit einem Namen versehen, der sich aus dem Modul- und Komponentenknotennamen zusammensetzt.

- **Entscheidung und Verbindung:** Der Entscheidungs- und der Verbindungsknoten werden im Gegensatz zum Aktivitätsdiagramm in zwei Knotenpaare aufgeteilt und die Darstellung angepasst. Daraus resultiert eine eindeutige Identifikation der Knoteninterpretationen, welche somit auch passend auf das Metamodell abgebildet werden kann. Folgende Knotenpaare werden demzufolge unterschieden.
  - Schleifen L LE: Schleifenknoten werden durch grüne Kreise dargestellt und der Beginn der Schleife mit einem L (für engl. Loop) und das Ende mit einem EL (für engl. EndLoop) bezeichnet. Der Kontrollfluss für den Schleifenrückfluss muss nicht angegeben werden, da dieser sich aus dem Diagramm erschließen lässt.
  - Bedingte Anweisungen ◇ ◆: Für bedingte Anweisungen wird das Ende nun durch ein schwarzes Quadrat dargestellt. Der Kontrollfluss für das Nichterfüllen der Bedingung muss nicht explizit angegeben werden, da dieser sich wie bei Schleifen aus dem Diagramm erschließen lässt und somit leer ist.

- **Aktivitäten:** Der Inhalt der Aktivitäten wird in der hier entworfenen DSL verborgen. Aktivitäten werden auf die Ableitungen des Gruppenknotens des Metamodells abgebildet. Aus diesem Grund werden auch zwei Darstellungen benötigt, damit die Abbildung auf das Metamodell bijektiv ist.
  - Kopieaktivität G: Diese Aktivität wird durch einen blauen Kreis mit einem G dargestellt.
  - Linkaktivität G: Diese Aktivität wird wie die Kopieaktivität dargestellt und durch einen Pfeil an der oberen rechten Ecke ergänzt.

  Ein Gruppenknoten enthält einen vollständigen Graphen und muss somit im Gegensatz zum Aktivitätsdiagramm der UML wieder einen Start- und Endknoten besitzen.

- **Gabelung und Vereinigung F J:** Im Gegensatz zum Aktivitätsdiagramm werden diese Knoten nun auch unterschiedlich und den anderen Knoten ähnelnd

dargestellt. Anstatt eines Rechtecks wird ein schwarzer Kreis als Symbol genutzt. Die Gabelung wird mit einem F (für engl. Fork) und die Vereinigung mit einem J (für engl. Join) beschriftet.

Wird die grafische Anpassung auf das Beispiel in Abbildung 4.7 angewandt, so ergibt sich das Diagramm in Abbildung 4.8. Dabei deuten die grauen Pfeile den Schleifenrücklauf bzw. den Kontrollfluss für das nicht Erfüllen der Bedingung der bedingten Anweisung an. Diese werden in der entworfenen DSL nicht mehr explizit dargestellt.

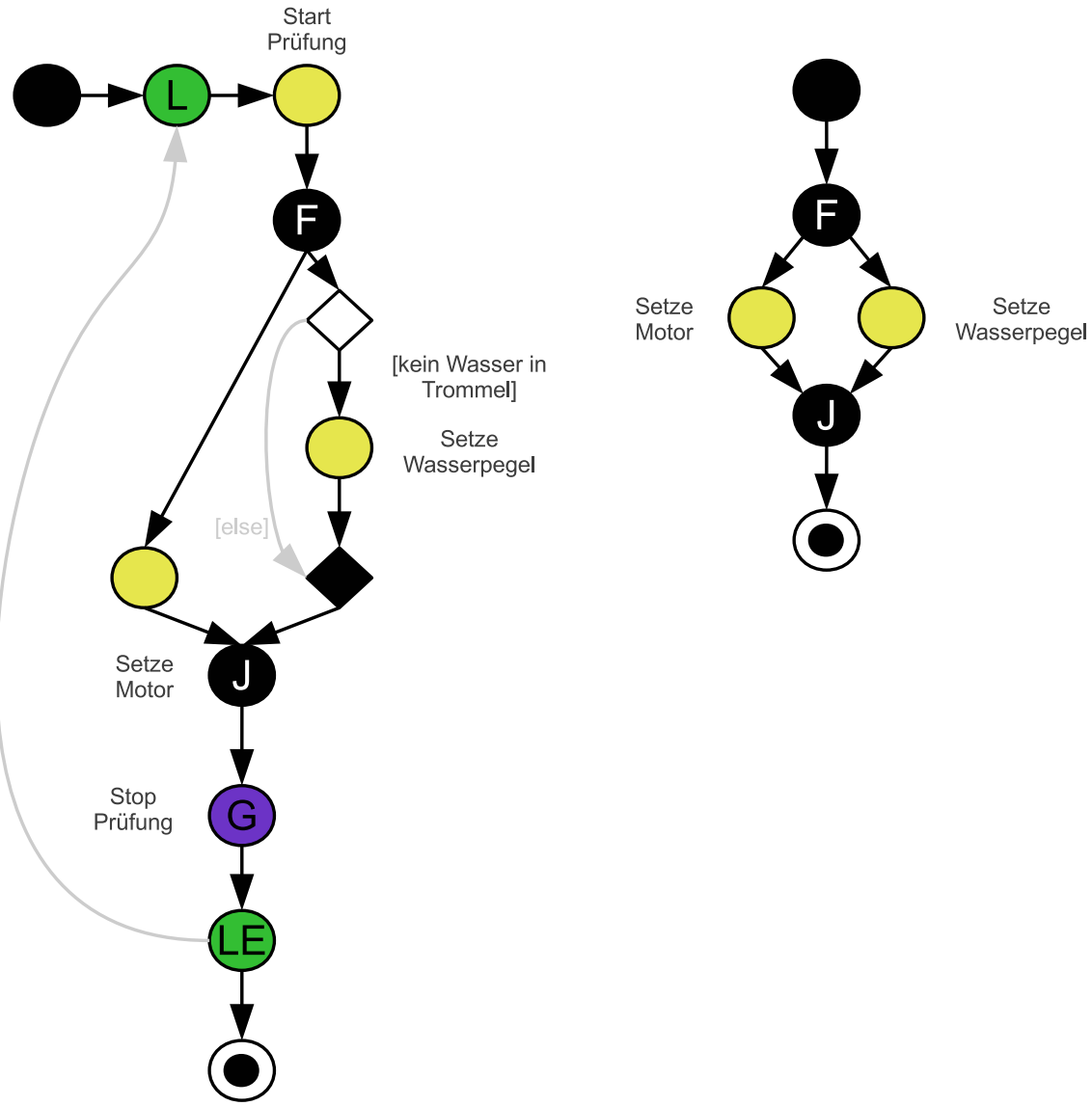


Abbildung 4.8.: Beispiel für die entworfene DSL

5. Implementierung

Neben der Erstellung des Konzepts zur grafischen Verwaltung von Steuer- und Messvorschriften in Kapitel 4 war auch die Implementierung Aufgabe der Bachelorarbeit.

Umgesetzt wurde eine Applikation, der SMVEditor, welcher die grafische Entwicklung von Steuer- und Messvorschriften und die automatisierte Codegenerierung ermöglicht.

Der durch den SMVEditor implementierte Prozess wird in Abschnitt 5.2 erläutert. In den darauf folgenden Abschnitten werden die wichtigsten Aspekte der Implementierung beschrieben und die verwendeten Technologien vorgestellt.

Auf der dieser Bachelorarbeit beiliegenden CD befinden sich der Quellcode, die Dokumentation und eine kompilierte Fassung des SMVEditors, welche unter Windows und Linux getestet worden ist.

5.1. Grundlagen und Architektur

Für die Implementierung des SMVEditors wurde die Programmiersprache Java genutzt, weil diese das in den Anforderungen festgelegte Kriterium der Plattformunabhängigkeit erfüllt, objektorientiert ist und eine umfangreiche Standardbibliothek anbietet. Die Objektorientierung dient zur Strukturierung der Applikation, welche nach dem Model-View-Controller-Muster (MVC) realisiert wird. Die Verwendung des MVC-Musters hat nach [7] den Vorteil, dass ein einfacher Austausch der Benutzerschnittstelle möglich ist, ohne andere Komponenten der Software anpassen zu müssen. Im Bezug auf den SMVEditor erlaubt diese Eigenschaft des MVC-Musters, neben der Austauschbarkeit der Swing-Bibliothek, auch das Wechseln der DSL. Ein weiterer Vorteil des MVC-Musters ist nach [8] die Möglichkeit das Modell unabhängig vom Controller und View zu testen. Von dieser Option wird im Abschnitt 5.10 Gebrauch gemacht.
Bei der Strukturierung des SMVEditors wurden Klassen gleichen Kontextes in Pakete zusammengefasst. Tabelle 5.1 beschreibt den Umfang der Implementierung und die Aufteilung der Pakete des SMVEditors.

Paket	**Beschreibung**
miele.gt.kew.smv.editor.view 2229 Zeilen, 21 Klassen	Paket zum Zeichnen der GUI und der domänenspezifischen Sprache
miele.gt.kew.smv.editor.controller 942 Zeilen, 7 Klassen	Paket zur Steuerung des SMVEditors
miele.gt.kew.smv.editor.model 577 Zeilen, 7 Klassen	Paket enthält Modell des SMVEditors
miele.gt.kew.smv.editor.model.metamodel 740 Zeilen, 16 Klassen	Paket enthält das Metamodell
miele.gt.kew.smv.editor.gen 733 Zeilen, 3 Klassen	Paket zur Validierung der Modellierung, Codegenerierung und Kompilierung
miele.gt.kew.smv.editor.exception 57 Zeilen, 5 Klassen	Paket enthält anwendungsspezifische Ausnahmeklassen
miele.gt.kew.smv.editor.misc 75 Zeilen, 2 Klassen	Paket mit verschiedenen Hilfsklassen
miele.gt.kew.smv.editor.test 284 Zeilen, 3 Klassen	Paket mit Unittests und Benchmark
Gesamt: 5637 Zeilen, 64 Klassen	Gesamtumfang der Implementierung

Tabelle 5.1.: Paketliste des SMVEditors

5.2. Prozessbeschreibung

Im Folgenden wird der durch den SMVEditor implementierte Prozess beschrieben, welcher in Abbildung 5.1 skizziert ist. Der Entwicklungsprozess einer SMV von der Modellierung bis zur ausführbaren Prüfung durchläuft verschiedene Stufen.
Die SMV-Entwicklung beginnt mit dem Editieren oder Erzeugen einer Steuer- und Messvorschrift über die grafische Benutzeroberfläche (GUI - Graphical User Interface) des SMVEditors. Diese wird im Abschnitt 5.3 beschrieben. In Abschnitt 5.4 wird die aus der Modellierung resultierende Metamodellinstanz erläutert. Jede Instanz des Metamodells kann durch den SMVEditor gespeichert und geladen werden (Abschnitt 5.5). Ein wichtiger Aspekt bei der Implementierung ist die Verarbeitung von Modulen, Komponentenknoten und Variablen. Die Beschreibung dieses Aspekts erfolgt in Abschnitt 5.6. Vor der Generierung des Quellcodes (Abschnitt 5.9) wird zunächst die Metamodellinstanz validiert (Abschnitt 5.7) und das Metamodell in die XVCL-Zwischenschicht transformiert (Abschnitt 5.8). Abschließend wird der Quellcode durch den Compiler in eine

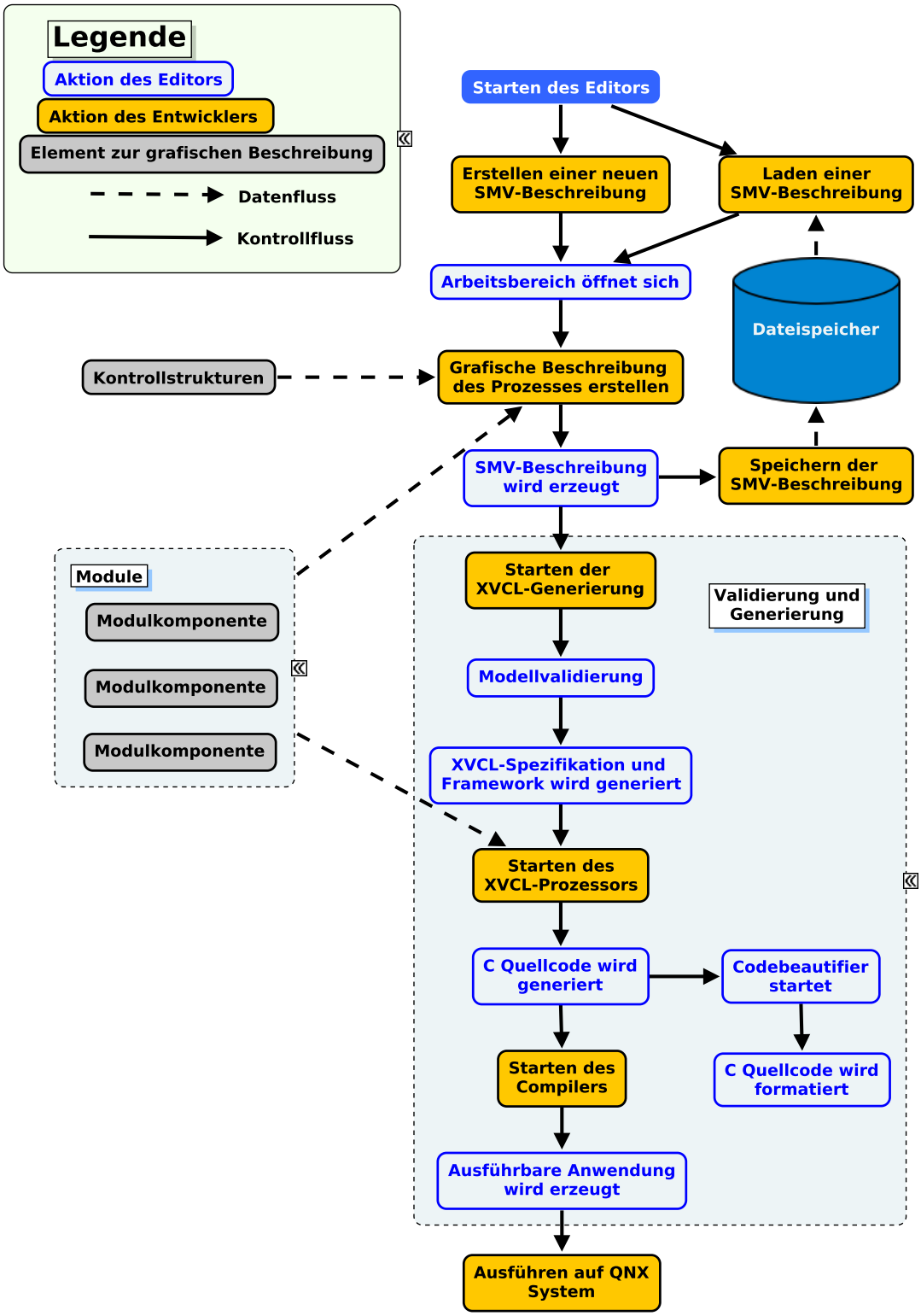
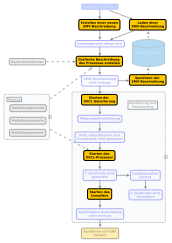


Abbildung 5.1.: Prozessbeschreibung der implementierten Applikation

QNX-kompatible Anwendung kompiliert. Zur besseren Orientierung in den folgenden Abschnitten wird durch eine Miniaturabbildung der Prozessbeschreibung angedeutet, welche Teile der Implementierung im betreffenden Abschnitt beschrieben werden.

5.3. Grafische Benutzeroberfläche



Die GUI wird im Paket *miele.gt.kew.smv.editor.view* auf Basis der plattformunabhängigen Programmierschnittstelle und der Grafikbibliothek Swing implementiert. Die grafische Oberfläche, welche in Abbildung 5.2 dargestellt ist, dient zur komfortablen Verwaltung von Steuer- und Messvorschriften. Sie setzt sich aus einer Menüleiste, einem Arbeitsbereich zur Modellierung mittels der DSL, sowie einer Seitenleiste zur Knotenwahl zusammen, die in den folgenden Unterabschnitten beschrieben werden.

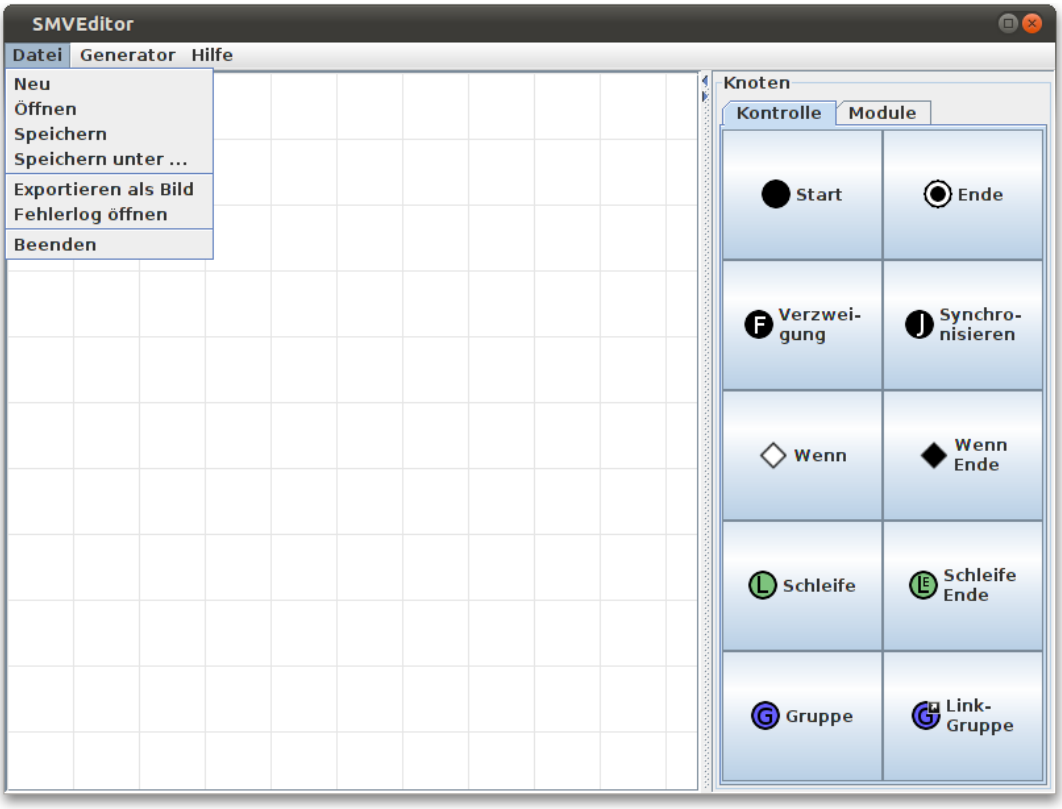


Abbildung 5.2.: Oberfläche des SMVEditors

5.3.1. Arbeitsbereich

Der Arbeitsbereich, welcher in der Klasse *WorkspaceView* implementiert wurde, dient zur Modellierung von Steuer- und Messvorschriften durch die domänenspezifische Sprache. Die Grundlage zur Darstellung des Arbeitsbereiches ist eine Ableitung der Klasse *JPanel* von Swing. Durch das Überschreiben der ursprünglichen Zeichenmethode wird die Rasterdarstellung des Arbeitsbereiches erzeugt, auf der sich Knoten und Kanten ausrichten lassen. Zudem werden Listener für Mauseingaben implementiert, durch die es möglich ist, auf Benutzereingaben zu reagieren. Durch einen Klick auf die linke Maustaste kann ein Knoten, welcher zuvor in der Seitenleiste gewählt wurde, gesetzt werden. Durch gedrückt Halten der linken Maustaste und das Bewegen auf dem Raster können zwei Knoten miteinander verbunden werden. Wird die rechte Maustaste gedrückt gehalten und auf dem Raster bewegt, kann ein Knoten verschoben werden. Durch einen Klick auf einen Knoten mit der rechten Maustaste wird ein Kontextmenü geöffnet, welches die Löschung der mit dem Knoten verbundenen Kanten oder des Knotens selbst erlaubt. Ist der Knoten parametrierbar bzw. ein Gruppenknoten, so enthält das Kontextmenü

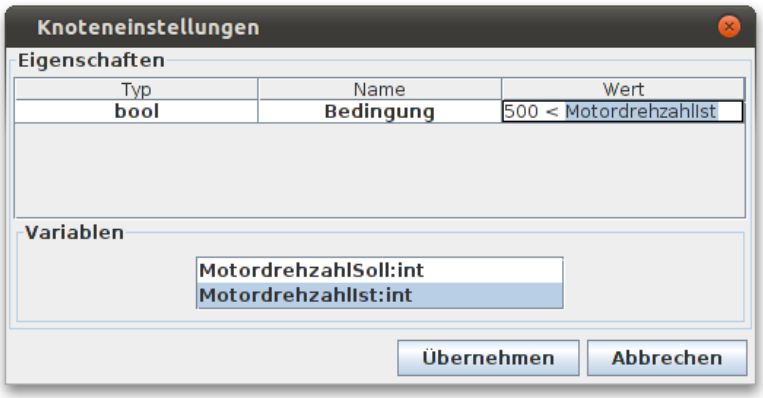


Abbildung 5.3.: Knotenparametrierung im SMVEditor

einen weiteren Menüpunkt, um ein Fenster für die Parametrierung bzw. zur Festlegung des Inhalts des Gruppenknotens zu öffnen. Die Parametrierung eines Wennknotens ist exemplarisch in Abbildung 5.3 skizziert. Dieser Knoten besitzt, wie bereits beschrieben, nur einen Parameter. In der Abbildung wird ein Vergleich einer Modulvariablen dargestellt. Die von den Modulen angebotenen Variablen, welche im gleichen Fenster aufgelistet sind, lassen sich per Drag&Drop in die Parametrierung einbetten.

5.3.2. Menüleiste

Die Menüleiste, welche in der Klasse mit dem Namen *MenuBar* implementiert wurde, bietet eine Auswahl von Menüpunkten zur Steuerung des SMVEditors an. Neben den üblichen Funktionen, wie speichern, laden und neu erstellen gibt es zusätzlich die Möglichkeit eine SMV als Bitmap zu exportieren und ein Logfenster zu öffnen. Das Logfenster, dargestellt in Abbildung 5.4, dient zur Anzeige von aufgetretenen Fehlern und sortiert diese in vier Kategorien. Dabei wird zwischen Fehlern die Interna des SMVEditors betreffen und Fehlern in der SMV, den Modulen und der Generierung unterschieden.

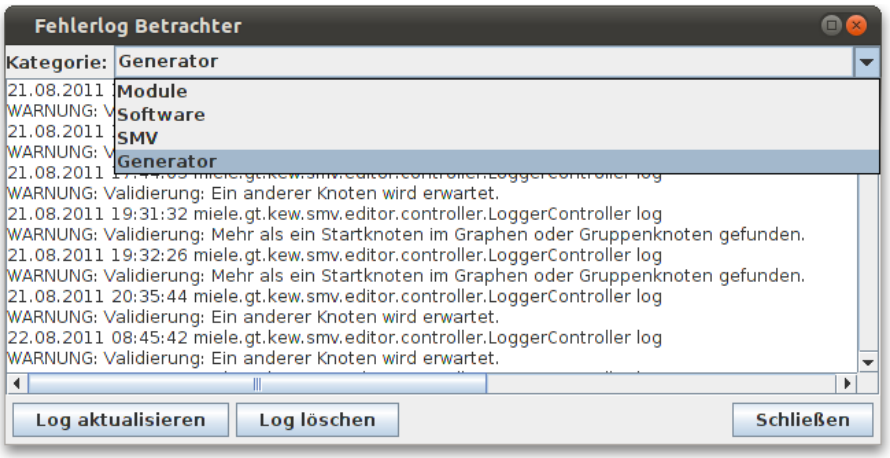


Abbildung 5.4.: Logfenster des SMVEditors

Für die verschiedenen Phasen der Generierung stehen drei Menüpunkte zur Verfügung. Die erste Phase endet mit der Generierung des XVCL-Codes. Die zweite Phase endet mit der Verarbeitung der XVCL-Dateien und erzeugt den C-Quellcode, welcher in Phase drei kompiliert wird.

5.3.3. Seitenleiste

Die Seitenleiste wird durch die Klasse *RSView* implementiert, die in der Swing-Klasse *JSplitter* eingebettet ist. Die ausblendbare Leiste dient zur Auswahl eines zu setzenden Knotens. In der Seitenleiste kommt eine Registerkarte zum Einsatz, um die Komponentenknoten von den Kontroll- und Gruppenknoten zu trennen. Der Grund für diese Unterteilung ist die daraus resultierende Übersicht, welche die dynamisch hinzugeladenen Komponentenknoten von den im Metamodell statisch festgelegten Knoten trennt. Bei der Selektion eines Knotens in der Seitenleiste lässt sich dieser in der Arbeitsfläche durch einen Klick auf die linke Maustaste platzieren.

5.4. Metamodell

Das Paket *miele.gt.kew.smv.editor.model.metamodel* implementiert das Metamodell in insgesamt 16 Javaklassen. Der Aufbau stellt die Implementierung des UML-Klassendiagramms in Abbildung 4.2 aus Kapitel 4 dar. Die Klassen des Metamodells sind serialisiert, damit die Speicherung von Metamodellinstanzen mit der Javastandardbibliothek und Bibliotheken Dritter möglich ist. Die abstrakte Klasse *Node*, und daraus resultierend alle Unterklassen, wird mit Koordinaten zur Positionierung im Raster der Arbeitsfläche erweitert, damit die Darstellung der SMV über die DSL durch das Metamodell möglich ist. Komponentenknoten besitzen als Attribut den Namen des korrespondierenden Moduls und der repräsentierten Modulkomponente, wodurch die Verknüpfung zum Modul und zur Modulkomponente hergestellt wird. Ein Parameter eines Knotens wird durch die Klasse *Variable* repräsentiert und enthält den Namen, den Typ und den String-Wert des Parameters, wodurch auch Ausdrücke als Parameter erlaubt sind. Bei der Modellierung einer SMV über die DSL werden Veränderungen direkt auf der Metamodellinstanz durchgeführt und die DSL anschließend mit den neuen Daten aktualisiert.

5.5. Datenspeicher

Das Laden und Speichern von erstellten SMV-Beschreibungen ist eine wichtige Komponente der Applikation. Aufgrund der Tatsache, dass das Metamodell in Form von Javaklassen im Paket *miele.gt.kew.smv.editor.model.metamodel* vorliegt, bietet es sich an diese zu serialisieren und die Instanzen mittels Serialisierung abzulegen. Dabei ist es von Vorteil, nicht die von Java mitgelieferte Serialisierung in Form von Bytetrömen zu nutzen, sondern die Ablage in ein für den Menschen lesbares Format zu wählen. Zu diesem Zweck wird in der Implementierung die Bibliothek XStream genutzt, welche das Serialisieren und Ablegen von Objekten in der Extensible Markup Language (XML) ermöglicht. Der Vorteil dieses Vorgehens ist, dass eine Erstellung und Bearbeitung der Metamodellinstanz mittels XML möglich ist, was zugleich auch als zusätzliche textuelle DSL verstanden werden kann.

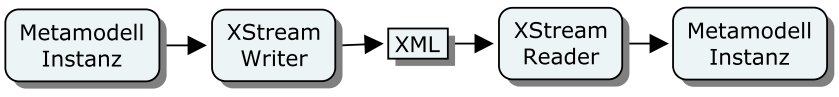


Abbildung 5.5.: Prozesskette zur Speicherung von Metamodellinstanzen

```
1  <miele.gt.kew.smv.editor.model.metamodel.SMV>
2    <graph>
3      <nodes>
4        <entry>
5          <miele.gt.kew.smv.editor.model.metamodel.LoopNode>
6            <x>200</x>
7            <y>150</y>
8            <outgoingEdges>
9              <null/>
10           </outgoingEdges>
11           <incomingEdges>
12             <null/>
13           </incomingEdges>
14           <vars>
15             <miele.gt.kew.smv.editor.model.metamodel.Variable>
16               <name>Wiederholungen</name>
17               <type>const unsigned int</type>
18               <value>5</value>
19             </miele.gt.kew.smv.editor.model.metamodel.Variable>
20           </vars>
21         </miele.gt.kew.smv.editor.model.metamodel.LoopNode>
22       </entry>
23     </nodes>
24   </graph>
25 </miele.gt.kew.smv.editor.model.metamodel.SMV>
```

Listing 5.1: XML-Repräsentation eines Graphen mit Schleifenstartknoten

Die Verkettung der einzelnen Verarbeitungsschritte ist in Abbildung 5.5 dargestellt. In Listing 5.1 wird die Repräsentation eines Schleifenstartknotens im Graphen veranschaulicht. Es ist ersichtlich, dass die Repräsentation genau eine Instanz des Metamodells darstellt, welche zusätzlich mit Koordinateninformationen für die DSL erweitert wurde. Das Bearbeiten der SMV ist via XML möglich und erlaubt z.B. das Ändern der Schleifendurchläufe durch Manipulation des Tag-Körpers in Zeile 18.

5.6. Module und Modulkomponenten

In diesem Abschnitt wird die Implementierung von Modulen und deren Komponenten beschrieben, dessen konzeptioneller Entwurf bereits in Unterabschnitt 4.2.3 durchgeführt wurde. Zwischen Quellcode und Metamodell befindet sich eine Template-Zwischenschicht, welche Anweisungen zur Erzeugung des Quellcodes enthält. Module und die Komponentenknoten werden in solchen Templates definiert und können dann in den SMVEditor geladen werden. Als Templatesprache wurde dazu XVCL verwendet, welche in Unterabschnitt 5.6.1 vorgestellt wird. Anschließend wird in Unterabschnitt

5.6.2 der Aufbau von Modulen mit Hilfe implementierter Templates beschrieben.

5.6.1. XVCL

Im Folgenden wird die XML-based Variant Configuration Language (XVCL) auf Grundlage von [9], [10] und [11] eingeführt. Bei XVCL handelt es sich um eine Mark-Up-Sprache, mittels der es möglich ist textbasierte Dokumente und somit auch Quellcode zu konfigurieren. Das Basiskonzept von XVCL fußt auf Paul G. Bassetts Frametechnologie [12].
Der Grundbaustein jeder XVCL-Datei ist ein x-frame. Dabei stellt ein x-frame eine Mischung aus normalem Text bzw. Quellcode und XVCL-Tags dar. Mit Hilfe dieser Tags lassen sich x-frames konfigurieren und weitere x-frames adaptieren. Die in Relation stehenden x-frames stellen ein x-framework dar. Zusätzlich wird eine XVCL-Spezifikationsdatei (SPC) benötigt, welche das erste x-frame darstellt und zur Konfiguration des x-frameworks dient. Zur Zusammenführung des x-frameworks und der SPC

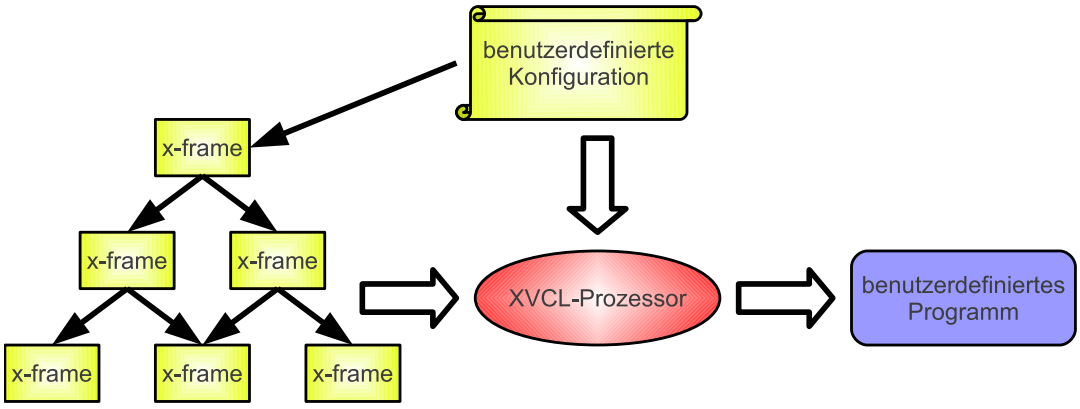


Abbildung 5.6.: Ablauf der XVCL-Verarbeitung [13]

dient der XVCL-Prozessor. Dieser wendet die Konfiguration auf das x-framework an und generiert die gewünschte Lösung. XVCL-Befehle werden durch XML-Tags beschrieben. Sie werden in ein textbasiertes Dokument eingebettet und beschreiben die mögliche Variabilität des Dokuments. In diesem Abschnitt werden die für die Bachelorarbeit relevanten Tags vorgestellt. Eine vollständige Referenz ist im Internet verfügbar [14].

- **x-frame Befehl:** Das <x-frame>-Tag stellt den Rahmen des x-frames dar und ordnet diesem einen Namen zu. Der Tag-Körper muss eine textuelle Form haben und kann zudem weitere XVCL-Befehle beinhalten. Das Ausgabeverzeichnis, der Name der Ausgabedatei, sowie die Sprache der Ausgabedatei können im Tag als Attribute optional festgelegt werden.

```
<x-frame name="Name" [outdir="Pfad"] [outfile="Dateiname"]
      [language="Sprache"]>
   ...
</x-frame>
```

Listing 5.2: Das <x-frame>-Tag

- **adapt Befehl:** Das <adapt>-Tag erlaubt es in einem x-frame weitere x-frames einzubinden. Diese können wiederum weitere x-frames einbinden. Dabei ist festzuhalten, dass ein x-frame sich selbst und transitiv adaptierte auf sich selbst verweisende x-frames nicht einbinden darf, da es möglich ist, dass der Prozessor dadurch in eine Endlosschleife geraten würde. Dem Tag muss als Attribut der Name des zu adaptierenden x-frames zugewiesen werden. Zudem ist es möglich weitere optionale Parameter hinzuzufügen, welche in der XVCL-Spezifikation genauer beschrieben werden.

  ```
  <adapt x-frame="Name" [...]/>
  ```

  Listing 5.3: Das <adapt>-Tag

- **Variablen Befehle:** Es gibt zwei XVCL-Tags um Variablen zu definieren. Um einfache Variablen zu erstellen, wird das <set>-Tag genutzt. Diesem muss ein Name und ein Wert zugewiesen werden.

  ```
  <set var="VarName" value="Wert" [...]/>
  ```

  Listing 5.4: Das <set>-Tag

  Darüber hinaus ist es möglich eine Variable mit mehreren Werten anzulegen. Dazu wird das <set-multi>-Tag genutzt, welches mehrere Werte, durch Kommata getrennt, als Attribut besitzen kann. Diese so definierten Multi-Variablen können später z.B. mit Hilfe von while-Schleifen ausgelesen werden.

  ```
  <set-multi var="MultVarName" value="Wert1[, Wert2, Wert3, ...]"/>
  ```

  Listing 5.5: Das <set-multi>-Tag

  Das <value-of>-Tag dient zum Auslesen von Variablen, welche anschließend in den Quelltext eingefügt werden. Dieses Tag benötigt als Attribut den Variablennamen. Um die in Listing 5.4 definierte Variable auszulesen, muss der Befehl folgende Form haben.

  ```
  <value-of expr="?@VarName?"/>
  ```

  Listing 5.6: Das <value-of>-Tag

- **while Befehl:** Das <while>-Tag erlaubt es über Multi-Variablen zu iterieren. Dazu wird dem Tag als Attribut mindestens eine Multi-Variable zugewiesen. Im Rumpf der so erzeugten Schleife können dann weitere XVCL-Befehle und Code stehen.

```
<while using-items-in="MultVarName[, MultVarName2, ...]">
    ...
</while>
```

Listing 5.7: Das <while>-Tag

In Abbildung 5.7 wird die Verarbeitung durch den XVCL-Prozessor an einem Beispiel veranschaulicht, welcher die SPC als erstes x-frame einließt und das x-framework somit konfiguriert.

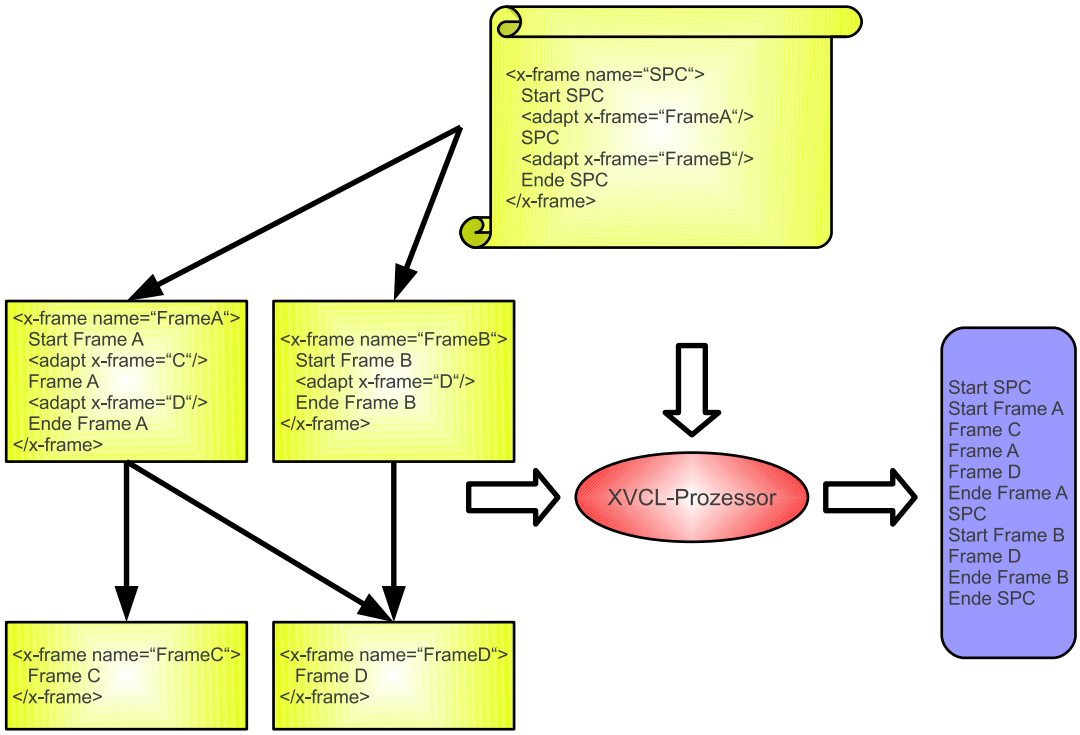


Abbildung 5.7.: Konkreter Ablauf eines XVCL-Beispiels

5.6.2. Repräsentation durch XVCL

Ein Modul besteht aus einer XVCL-Datei, dem Header, und aus weiteren x-frames für die vom Modul angebotenen Komponentenknoten. Der Header wird bei der Codegenerierung in den Kopf des Quellcodes eingefügt und dient zur Deklaration von modulrelevanten Variablen. Weitere Anwendungen des Headers sind die Implementierung von Funktionen und die Deklaration von Prototypen, die von den implementierten Komponentenknoten des Moduls verwendet werden können. Die deklarierten Variablen sind nicht ausschließlich für interne Zwecke der Module vorgesehen, sondern können auch global freigegeben werden, um zur Parametrierung von anderen Knoten zu dienen. Für die Freigabe von

Variablen zur Parametrierung müssen die Variablen im Modulheader mit dem <set>-Tag von XVCL annotiert werden. Diese Tags werden beim Einbinden der Module durch den SMVEditor ausgelesen und als freigegebene Variablen zur Knotenparametrierung, beispielhaft dargestellt in Abbildung 5.3, angeboten. Das <set>-Tag besitzt als Attribute den Namen der Variablen und den Typ. Listing 5.8 implementiert exemplarisch die Freigabe einer Modulvariablen.

```
<set var="MotordrehzahlIst" value="int"/>
int MotordrehzahlIst = 0;
```

Listing 5.8: Freigabe einer Variablen in einem Modul

Neben dem Header besitzt jedes Modul weitere XVCL-Dateien zur Definition von Komponentenknoten. Jede dieser XVCL-Dateien repräsentiert genau einen Komponentenknoten. Sie enthalten Quellcode, der durch Parameter konfiguriert werden kann. Die möglichen Parameter werden durch das <set>-Tag festgelegt. Das <set>-Tag besitzt als Attribute den Namen des Parameters und den Typ. Das Auslesen des über den SMVEditor festgelegten Parameterwertes wird durch das <value-of>-Tag von XVCL ermöglicht. Listing 5.9 implementiert exemplarisch die Festlegung eines Parameters und das Auslesen des gesetzten Wertes.

```
<set var="Drehzahl" value="int"/>
int MotordrehzahlSoll = <value-of expr="?@Drehzahl?"/>;
```

Listing 5.9: Parametrisierung von Komponentenknoten

5.7. Validierung

Die Validierung der Metamodellinstanz erfolgt durch die Methode *validateGraph(Graph graph)* der Klasse *Validator* des Pakets *miele.gt.kew.smv.editor.gen*. Für die Validierung wurde ein Algorithmus entwickelt, der den Graphen in linearer Laufzeit zur Knotenanzahl nach Fehlern untersucht. Bei der Erkennung eines Defekts wird die Ausnahme *NodeException* aus dem Paket *miele.gt.kew.smv.editor.exception* geworfen, welche den Knoten mit zugehörigem Fehlertext enthält. Diese Exception wird abgefangen und eine Dialog-Meldung erzeugt, sowie ein Eintrag für das Logfenster eingefügt. Außerdem wird der in der Ausnahme eingetragene Knoten im Arbeitsbereich des SMVEditors markiert, wodurch der Benutzer die Fehlerquelle eindeutig identifizieren kann. Im Folgenden wird das Vorgehen des Algorithmus beschrieben.

1. **Startknoten überprüfen:** Im ersten Schritt wird zunächst überprüft, ob ein Startknoten vorhanden ist und ob neben diesem keine weiteren Startknoten im Graphen existieren. Ist dies nicht der Fall, erfolgt der Wurf einer Ausnahme, welche eine passende Fehlerbeschreibung enthält, aber keinen Knoten, da ein nicht

vorhandener Knoten auf der Arbeitsfläche nicht markiert werden kann. Ist alles korrekt, wird der Startknoten in der Variablen *selectedNode* abgelegt und mit Punkt 2 fortgefahren.

2. **Variablenvalidierung:** Die Variablenvalidierung dient zur Überprüfung der Parameter des Knotens in der Variablen *selectedNode*. Dazu wird eine C-Datei erzeugt, die alle Modulvariablen deklariert. In der Hauptroutine werden die Parameter des Knotens als Variablen deklariert und mit den zugehörigen Ausdrücken initialisiert. Anschließend erfolgt die Kompilierung der C-Datei durch den QCC-Compiler. Dabei wird auf das Linken durch Aufruf des QCC mit dem Parameter -c verzichtet, da dies weitere Zeit in Anspruch nimmt und keinen Nutzen für die Variablenvalidierung hat. Wenn der QCC einen Fehler erkennt, wird eine Ausnahme geworfen. Findet der Compiler keine Fehler, wird mit Punkt 3 fortgefahren.

3. **Knotenerkennung:** In diesem Schritt wird geprüft, um welchen Typ von Knoten es sich in der Variablen *selectedNode* handelt. Die folgende Auflistung beschreibt die Reaktion bei Erkennung der Knotenausprägung.

   - **Startknoten, Wennknoten, Schleifenstartknoten und Gabelknoten:** Ist der Knoten in der Variablen *selectedNode* einer dieser Knoten, so wird er auf den Stack *nodeStack* abgelegt und mit Punkt 4 fortgefahren. Der Stack dient zur Erkennung der Teilgraphenbedingung, die im Entwurf erläutert wurde. Im Fall des Startknotens ist z.B. erforderlich, dass an passender Stelle im Graphen ein Endknoten folgt.
   - **Schleifenendeknoten, Wennendeknoten und Endknoten:** Bei der Erkennung eines dieser drei Knoten wird das oberste Element des Stacks *nodeStack* ausgelesen und vom Stack gelöscht. Ist der ausgelesene Knoten das Gegenstück des in der Variablen *selectedNode* enthaltenen Knotens, so ist alles korrekt und es wird mit Punkt 4 fortgefahren. Es wird eine Ausnahme geworfen, falls dies nicht der Fall ist.
   - **Komponentenknoten:** Ist der in der Variablen *selectedNode* enthaltene Knoten ein Komponentenknoten, so wird überprüft, ob der Modulname und der Komponentenname dem SMVEditor zur Verfügung stehen. Ist das der Fall, so wird zusätzlich überprüft, ob die Parameter in Anzahl, Typ und Namen übereinstimmen. Tritt an einer Stelle ein Fehler auf, wird eine Ausnahme geworfen. Ansonsten wird mit Punkt 4 fortgefahren.
   - **Synchronisationsknoten:** Es wird überprüft, ob das oberste Element im Stack *nodeStack* ein Gabelknoten ist. Ist dies der Fall, so wird der Knoten vom Stack entfernt. Anschließend wird validiert, ob der Gabelknoten und der

Synchronisationsknoten zwei Teilgraphen vollständig umschließen und somit auch beide ausgehenden Pfade aus dem Gabelknoten auch an passender Stelle wieder in den Synchronisationsknoten vereint werden. Dazu wird überprüft, ob in der Hashmap *threadHashMap* der Gabelknoten als Schlüssel bereits vorkommt. Ist kein Eintrag vorhanden, wird das Paar mit dem Gabelknoten als Schlüssel in die Hashmap eingetragen. Sonst wird überprüft, ob die Threadknotenpaare identisch sind und falls dies der Fall ist, das Paar aus der Hashmap gelöscht. Ist keine der Bedingungen verletzt, wird mit Punkt 4 fortgefahren. Ansonsten erfolgt der Wurf einer Ausnahme.

- **Gruppenknoten:** Bei der Erkennung eines Gruppenknotens wird rekursiv die Methode *validateGraph(Graph graph)* mit dem vom Gruppenknoten enthaltenden Graphen aufgerufen und die statische Variable *groupNodeNesting* der Klasse *Validator* inkrementiert. Nach der Rückkehr von der Gruppenknotenvalidierung wird *groupNodeNesting* dekrementiert und mit Punkt 4 fortgefahren. Steigt der Wert von *groupNodeNesting* über einen festgelegten Wert (in der aktuellen Implementierung 10), wurde eine tiefe Verschachtelung von Gruppenknoten und möglicherweise ein Kreis erkannt. Die Folge ist der Wurf einer Ausnahme.

4. **Kantenvalidierung:** Bei der Kantenvalidierung werden die Kanten des Knotens in der Variablen *selectedNode* überprüft, indem der Algorithmus untersucht ob alle ausgehenden Kanten existieren und auf einen Knoten verweisen. Anschließend wird mit Punkt 5 fortgefahren.

5. **Knoten wählen:** In diesem Schritt wird der nächste zu validierende Knoten in der Variablen *selectedNode* abgelegt oder von der Methode *validateGraph(Graph graph)* zurückgekehrt. Es gibt drei mögliche Fälle.

   - **Analyse abgeschlossen:** Es wird von der Methode *validateGraph(Graph graph)* zurückgekehrt, falls *nodeStack* und *threadHashMap* keine Elemente mehr enthalten. Handelt es sich bei dem validierten Graphen um einen Gruppenknoten, so ist die Überprüfung dieses Teilgraphen abgeschlossen. Anschließend wird die Validierung des aufrufenden Graphen fortgesetzt und der Nachfolger des Gruppenknotens in der Variablen *selectedNode* abgelegt. Die Validierung ist vollständig abgeschlossen, falls es sich nicht um den Graphen eines Gruppenknotens handelt. Ansonsten wird mit Schritt 2 fortgefahren.

   - **Threads vorhanden:** Ist nur *nodeStack* leer, so wird aus der Hashmap ein beliebiger Gabelknoten entnommen und in der Variablen *selectedNode* abgelegt. Die Validierung fährt mit Schritt 3 fort.

- **Nachfolger:** Ist keiner der beiden zuvor genannten Fälle eingetreten, so wird der Knoten in der Variablen *selectedNode* durch seinen Nachfolger überschrieben. Bei einem Gabelknoten wird ein beliebiger Nachfolger ausgewählt, der noch nicht validiert wurde. Die Validierung wird in Schritt 2 fortgeführt.

5.8. Modell zu XVCL Transformation

Das Modell wird nicht direkt in Quellcode übersetzt, sondern zunächst zu XVCL transformiert. Zusammen mit den Modulen und Modulkomponenten, die bereits in XVCL vorliegen, wird dadurch ein x-framework mit zugehöriger SPC erzeugt. Der Grund für diese Zwischenschicht ist die einfache Anpassung von XVCL-Templates, die Änderungen an der Generierung ohne Anpassungen des SMVEditors ermöglichen.

In Abschnitt 5.8.1 wird die Architektur des x-frameworks mit zugehöriger SPC erläutert. Anschließend folgt in Abschnitt 5.8.2 die Beschreibung des Algorithmus zur Erzeugung der XVCL-Schicht.

5.8.1. XVCL-Architektur

Das x-framework mit zugehöriger SPC ist als Schichtenarchitektur realisiert worden. Nach [15] zeichnet sich dieser Aufbau dadurch aus, dass „einzelne Aspekte des [...] Systems konzeptionell einer Schicht (engl. tier oder layer) zugeordnet“ sind und somit ein einheitlicher und übersichtlicher Aufbau möglich ist. Die Architektur wird im Folgenden anhand von Abbildung 5.8 erläutert.

- **SPC-Frame:** Das initiale x-frame ist die SPC. Diese enthält die Anzahl der Threads, die Namen der zu inkludierenden Module und adaptiert den Haupt-Frame.

- **Haupt-Frame:** Dieses x-frame beschreibt den Aufbau und die Anordnung des Quellcodes. Es dient zudem zur Generierung der Thread-Prototypen und Einbindung der Thread-Implementierung. Des Weiteren werden die Modulheader inkludiert und die Hauptroutine deklariert.

- **Modul-Frames:** Diese Schicht setzt sich aus den Modulheadern der Module zusammen.

- **Thread-Frames:** Jeder Thread der Metamodellinstanz wird in einem eigenen x-frame ausgelagert. Diese können sich anschließend gegenseitig starten. Thread_0

stellt dabei den ersten Thread der SMV dar und wird in die Hauptroutine eingebettet. Alle anderen Threads werden fortlaufend mit Thread_1, Thread_2 usw. bezeichnet. Die Thread-Frames dienen zur Anordnung und Parametrierung von Knoten-Frames.

- **Knoten-Frames:** Diese Schicht setzt sich aus den XVCL-Templates der Modulkomponenten und Kontrollknoten zusammen.

Die Templates der vom Metamodell bereitgestellten Kontrollknoten und das Haupt-Frame liegen dem SMVEditor bei und können im Anhang A.2 nachgeschlagen werden.

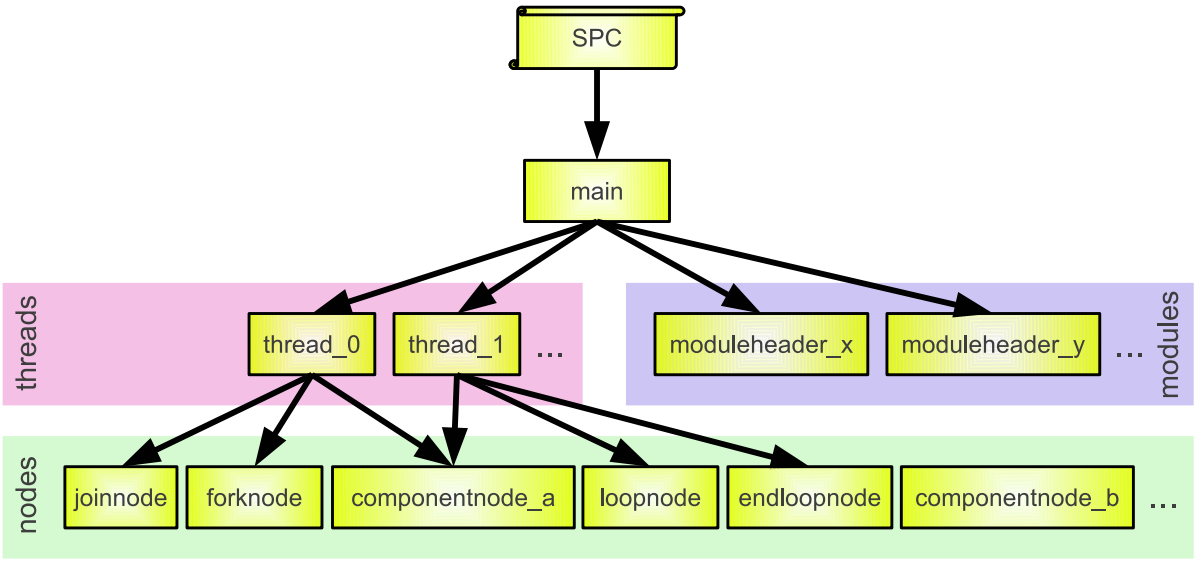


Abbildung 5.8.: XVCL-Schichtarchitektur

5.8.2. XVCL-Generierung

In der Klasse *CodeGenerator* des Pakets *miele.gt.kew.smv.editor.gen* wird die XVCL-Generierung implementiert. Diese erzeugt in linearer Laufzeit aus dem Metamodell das SPC-Frame und die Thread-Frames. Die anderen Frames liegen der Applikation bei oder werden durch Module bereitgestellt.
Im ersten Schritt der Generierung wird die SPC erzeugt. Dazu werden zum einen die Modulnamen und zum anderen die Anzahl der Threads benötigt. Alle dem SMVEditor zur Verfügung stehenden Modulnamen werden mittels des <set-multi>-Tags in die SPC eingetragen. Die Anzahl der Threads einer Metamodellinstanz wird durch die Gabel- bzw. Synchronisationsknotenanzahl des zusammenhängenden Graphen ermittelt. Dazu

wird der Graph ausgehend vom Startknoten traversiert. Anschließend wird ein <set-multi>-Tag in die SPC eingetragen, welcher die Threadnummerierungen, beginnend mit 1 bis einschließlich der Anzahl der erkannten Threads, aufreiht. Das Tag bleibt leer, falls die Threadanzahl 0 ist. In Listing 5.10 wird beispielhaft eine SPC dargestellt, welche zwei Module enthält und einen weiteren Thread deklariert.

```
<x-frame name="spc.xvcl">
  <set-multi var="moduleheaders" value="ModulA,ModulB"/>
  <set-multi var="threadnumbers" value="1"/>
  <adapt x-frame="main.xvcl"/>
</x-frame>
```

Listing 5.10: Beispiel einer SPC

Im zweiten Schritt werden die Thread-Frames erzeugt. Dazu wird der Graph in Abschnitte eingeteilt, die jeweils einen Thread repräsentieren. Dies ist beispielhaft in Abbildung 5.9 dargestellt. Zunächst werden die Gruppenknoten durch ihre beinhaltenden Graphen ohne Start- und Endknoten ersetzt.

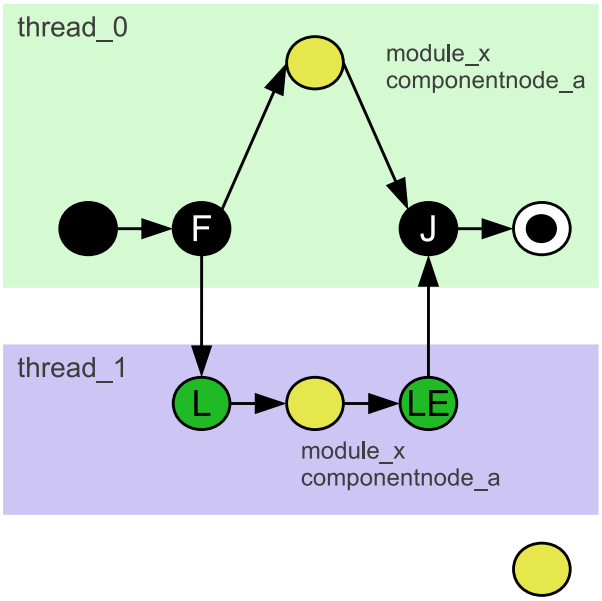


Abbildung 5.9.: Threaderkennung im Graphen

Die Unterteilung des Graphen beginnt mit der Erkennung eines Pfades ausgehend vom Startknoten und endend im Endknoten. Diese Knotenkette repräsentiert den Thread der Hauptroutine, den Thread_0. Alle entlang des Pfades erkannten Gabelknoten werden von 1 beginnend nummeriert und auf einem Stack gepuffert. Die Synchronisationsknoten auf dem Pfad werden als besucht markiert. Anschließend beginnt die Erkennung weiterer Threads. Dazu werden die Gabelknoten aus dem Stack hintereinander entnommen. Die Nummer des Gabelknotens repräsentiert die Nummer des Threads. Ausgehend von der

noch nicht beschrittenen Kante des Gabelknotens wird ein Pfad, der in einem markierten Knoten endet, erkannt und diese Knotenkette dem Thread zugeordnet. Dem Synchronisationsknoten am Ende wird die Nummer des Gabelknotens am Pfadanfang zugewiesen. Der Gabel- und Synchronisationsknoten am Anfang bzw. Ende des Pfades werden aus der Knotenkette entfernt. Jeder entlang dieser Knotenkette erkannte Gabelknoten erhält die nächste freie Nummer und wird auf dem Stack abgelegt. Die Synchronisationsknoten werden als besucht markiert. Dieser Vorgang wird so lange wiederholt, bis kein weiterer Gabelknoten auf dem Stack liegt. Das Resultat dieser Vorgehensweise sind Knotenketten, die einem Thread zugeordnet sind. Diese müssen anschließend in Thread-Frames umgewandelt werden. Dazu wird für jeden Thread eine XVCL-Datei angelegt. Anschließend werden die Knotenketten interpretiert. Jeder Knoten besitzt eine Repräsentation in XVCL, welche mit dem <adapt>-Tag von XVCL adaptiert wird. Besitzt der Knoten Parameter, werden diese dem <adapt>-Tag mit einem <set>-Tag, welcher die Parameterbelegung enthält, vorangestellt. Gabel- und Synchronisationsknoten bekommen die Nummer des Knotens als Parameter übergeben. In Listing 5.11 wird exemplarisch das Ergebnis des Algorithmus am Thread_0 aus der Abbildung 5.9 dargestellt.

```
<x-frame name="thread_0.xvcl">
  <set var="thread" value="1"/>
  <adapt x-frame="fork.xvcl"/>
  <adapt x-frame="module_x/componentnode_a"/>
  <set var="thread" value="1"/>
  <adapt x-frame="join.xvcl"/>
</x-frame>
```

Listing 5.11: Beispiel eines Thread-Frames

5.9. Codegenerierung, Formatierung und Kompilierung

Zur Erzeugung des Quellcodes wird der XVCL-Prozessor gestartet. Dieser interpretiert die XVCL-Befehle und generiert die Quellcodedatei. Eine Funktion die XVCL nicht beherrscht, ist das korrekte Einrücken von C-Codezeilen zur besseren Lesbarkeit. Aus diesem Grund wurde eine Codeformatierung im Paket *miele.gt.kew.smv.editor.gen* implementiert, die diese Aufgabe erfüllt. Die Codeformatierung erfolgt, indem einmal über jede Zeile des vom XVCL-Prozessor generierten Quellcodes iteriert wird. Durch einen Zähler wird festgehalten, wie tief eine Anweisung in Blöcken verschachtelt ist. Jede Codezeile wird zunächst getrimmt und anschließend wieder mit Whitespace aufgefüllt, dessen Größe durch den Zähler festgelegt wird. Abschließend wird das Zeilenende auf geöffnete oder geschlossene geschweifte Klammern überprüft und gegebenenfalls der Zähler erhöht bzw. reduziert.
Die Laufzeit ist linear zur Codezeilenanzahl der generierten Quelltextdatei, da der Code-

formatierter einmal über jede Zeile des Codes iteriert. Der Quellcode kann anschließend mit dem QCC kompiliert werden.

5.10. Tests

Neben dem SMVEditor wurde im Paket *miele.gt.kew.smv.editor.test* in der Klasse *Unittest* eine Test-Suite implementiert. Grundlage stellt dabei das Open-Source Framework JUnit dar, welches zum Testen von in Java entwickelten Klassen dient.
Nach [2] ist die Anpassung und Erweiterung des Metamodells eine häufige Tätigkeit in MDSD-Projekten. Wird das Metamodell beim SMVEditor verändert, muss unter Umständen die Validierung und Codegenerierung angepasst werden. Aus diesem Grund wurde im Rahmen dieser Arbeit der Validator getestet. Dabei dienen die Tests nicht nur zur Erkennung von Fehlern, sondern auch als Regressionstests. Damit soll sichergestellt werden, dass bei Anpassungen des Metamodells oder des Validators valide und nicht valide Steuer- und Messvorschriften vom Validator weiterhin korrekt erkannt werden. Der Testumfang umfasst 16 unterschiedliche Steuer- und Messvorschriften, von denen 4 valide sind. Alle 16 SMVs werden nacheinander durch den Validator überprüft. Der Test ist erfolgreich, wenn alle validen und nicht validen SMVs korrekt erkannt wurden.

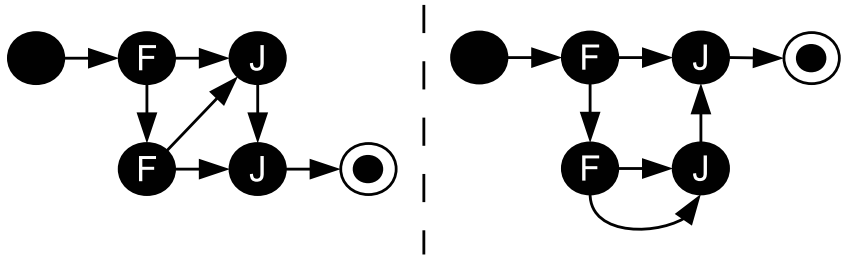


Abbildung 5.10.: Beispiel für nicht validen (links) und validen (rechts) Graphen

Bei der Durchführung stellte sich heraus, dass alle Tests erfolgreich sind. Dies lässt den Schluss zu, dass der Validierungsalgorithmus korrekt konzipiert und implementiert wurde, garantiert dies aber nicht. Edsger Wybe Dijkstra hatte diesen Sachverhalt schon im Jahre 1972 wie folgt zusammengefasst: „Program testing can be used to show the presence of bugs, but never to show their absence“[1][16, S. 6].

[1] Übersetzung: Programmtests können dafür genutzt werden die Präsenz von Bugs zu zeigen, aber niemals deren Abwesenheit.

6. Evaluation

Dieses Kapitel beschreibt die Evaluation der Implementierung. Im Abschnitt 6.1 wird ein Benchmark zur Messung der Validierungs- und Generierungsgeschwindigkeit durchgeführt und bewertet. Im Anschluss daran wird in Abschnitt 6.2 beurteilt, ob die in den Anforderungen festgelegten Kriterien erfüllt worden sind.

6.1. Benchmark

Damit der SMVEditor praktikabel eingesetzt werden kann, muss dieser die Metamodellinstanz performant validieren und die ausführbaren Steuer- und Messvorschriften in hinreichend kurzer Laufzeit generieren können. Die Laufzeiten der eingesetzten Algorithmen besitzen eine lineare Abhängigkeit zur Knotenzahl des Graphen. Es wird überprüft, ob die Theorie mit der Praxis übereinstimmt.

6.1.1. Szenario

Der Benchmark wurde auf dem PC durchgeführt, der üblicherweise im Dauerversuch verwendet wird. Das System ist mit einem Intel Pentium Dual-Core E2160 Prozessor mit 1,8GHz ausgestattet. Als Arbeitsspeicher kommen 2GB DDR2-Ram zum Einsatz. Derzeit wird als Betriebssystem Windows XP SP3 in der 32-Bit Version eingesetzt.
Damit die verschiedenen Daten erfasst werden können, wird ein Satz unterschiedlich großer Graphen benötigt. Aus diesem Grund wurde im Paket *miele.gt.kew.smv.editor.test* in der Klasse *Benchmark* ein Graphengenerator implementiert, welcher beliebig große Graphen erzeugen kann. Die erzeugten Graphen haben eine gerade Anzahl an Knoten. Bei der Graphengenerierung werden zwischen dem Start- und Endknoten so lange Schleifenknotenpaare aufgereiht, bis die gewünschte Knotenanzahl erreicht ist. Somit konvergiert die relative Anzahl der parametrierbaren Knoten gegen 50 Prozent.

6.1.2. Durchführung

Für den Benchmark werden Graphen mit 2 bis 300 Knoten generiert. Diese werden anschließend validiert und zu ausführbaren SMVs transformiert. Die Zeiterfassung erfolgt

mittels *System.currentTimeMillis()*, die je nach Betriebssystem auf wenige Millisekunden genau ist.
Bei der Messung der Zeit für die einzelnen Abschnitte der Generierung wurden die in der Tabelle 6.1 notierten Werte ermittelt. Diese zeigen deutlich, dass die im SMVEditor eigens entwickelten Generierungsalgorithmen eine lineare Abhängigkeit der Laufzeit zur Knotenanzahl aufweisen. Die Generierung des Codes durch den XVCL-Prozessor erweist sich als entscheidend für die Gesamtdauer der Generierung. Das Diagramm in Abbildung 6.1 verdeutlicht diesen Sachverhalt visuell.

Knoten	**XVCL-Generierung**	**Code-Generierung**	**Kompilierung**	**G. Gesamt**
2	2ms	206ms	53ms	261ms
50	5ms	189ms	62ms	256ms
100	11ms	232ms	70ms	313ms
150	14ms	233ms	78ms	325ms
200	19ms	239ms	77ms	335ms
250	26ms	259ms	87ms	372ms
300	33ms	287ms	97ms	417ms

Tabelle 6.1.: Generierungsdauer

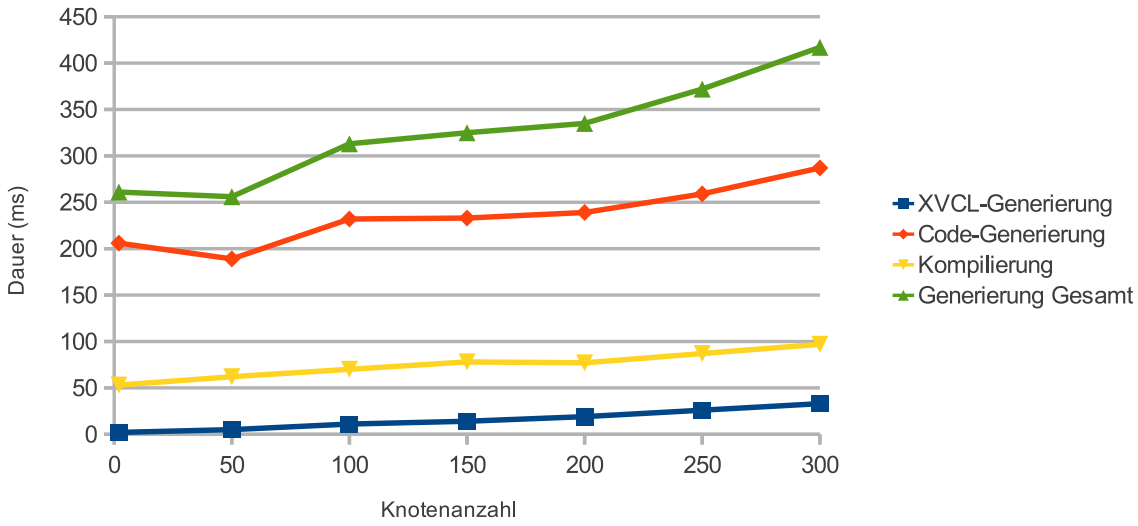


Abbildung 6.1.: Generierungsdauer

In Tabelle 6.2 sind die Messdaten der Validierungs-, Generierungs- und der Gesamtdauer eingetragen, welche in Abbildung 6.2 grafisch dargestellt werden. Es wird deutlich,

dass die Dauer der Validierung erheblich stärker wächst als die der Generierung. Mit steigender Knotenanzahl wird die Generierungsdauer zunehmend irrelevant.
Es liegt die Vermutung nahe, dass die hohe Frequenz der Compilerstarts die Validierung verlangsamt. Aus diesem Grund wurde in Tabelle 6.3 die Validierungsdauer ohne Variablenüberprüfung gemessen und mit der vollständigen Validierung verglichen. Diese zeigt, dass bei 24 parametrierbaren Knoten von insgesamt 50 Knoten die Parametervalidierung 93 Prozent der Validierungsdauer ausmacht und somit eindeutig einen Flaschenhals darstellt, welcher jedoch bei 300 Knoten noch akzeptabel ist. Die Parametervalidierung bietet Spielraum für mögliche Optimierungen.

Knoten	**Validierung**	**Generierung**	**Gesamt**
2	13ms	261ms	274ms
50	785ms	256ms	1041ms
100	1503ms	313ms	1816ms
150	2459ms	325ms	2784ms
200	3311ms	335ms	3646ms
250	4465ms	372ms	4837ms
300	5615ms	417ms	6032ms

Tabelle 6.2.: Validierungs-, Generierungs- und Gesamtdauer

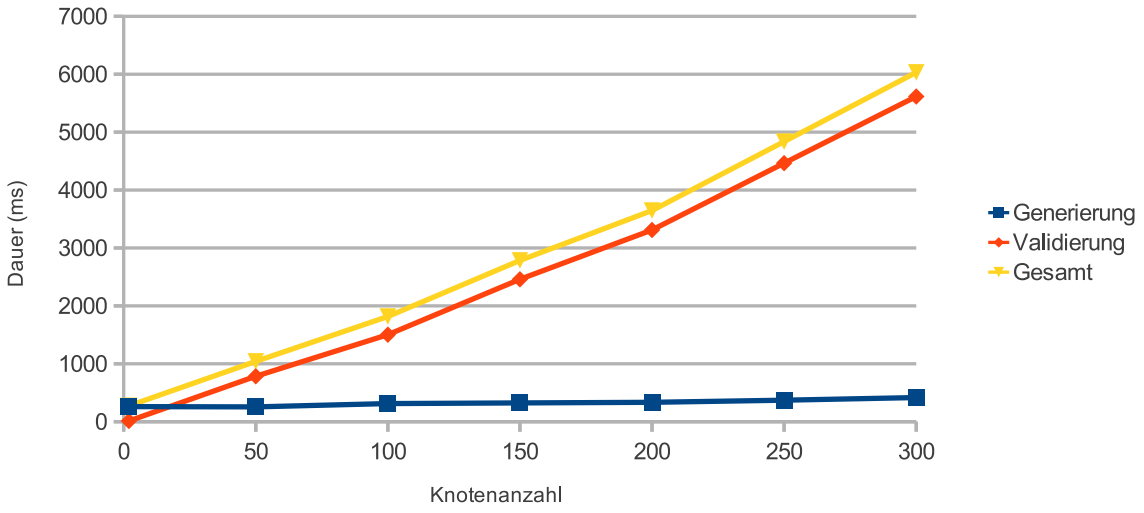


Abbildung 6.2.: Validierungs-, Generierungs- und Gesamtdauer

Knoten	**Validierung**		**Anteil**
	ohne Parameter	**mit Parameter**	**Parameterprüfung**
2	13ms	13ms	-
50	58ms	785ms	93%
100	68ms	1503ms	95%
150	85ms	2459ms	97%
200	91ms	3311ms	97%
250	99ms	4465ms	98%
300	104ms	5615ms	98%

Tabelle 6.3.: Validierung

6.1.3. Auswertung

Die Auswertung der Daten führt zu folgenden vier Erkenntnissen.

- Die Laufzeit der Validierung und Generierung ist linear zur Knotenanzahl.
- Die benötigte Zeit für die Validierung und Generierung hängt maßgebend von der Validierungsdauer ab.
- Die Validierungsdauer hängt maßgeblich von der Anzahl der parametrierbaren Knoten ab.
- Die Geschwindigkeit der Validierung und Generierung ist hoch genug für einen produktiven Einsatz der Software.

6.2. Anforderungsevaluation

In diesem Abschnitt wird überprüft und beurteilt, ob die Implementierung des SMV-Editors die im Kapitel 3 festgelegten Anforderungen erfüllt.

6.2.1. Grafische Modellierung

Im SMVEditor ist die im Entwurf konzipierte grafische Sprache implementiert worden. Im Folgenden wird überprüft, ob die DSL den gestellten Anforderungen genügt.

- **Modellierungsdesign**  **:** Ziel war es eine Modellierung zu entwerfen, welche sich an eine bereits existierende Modellierungssprache anlehnt. Die im SMVEditor

implementierte DSL wurde vom UML-Aktivitätsdiagramm abgeleitet und modifiziert. Entstanden ist eine schlanke Sprache, die dem Aktivitätsdiagramm weiterhin ähnelt. Somit ist das Kriterium erfüllt.

- **Abstraktion**  **:** Es war gefordert, dass die zu entwickelnde grafische Sprache eine abstraktere Programmierung als in PEARL90 erlaubt. Benötigte Sprachkonstrukte, wie das Modellieren von Schleifen mit konstanter Wiederholungsanzahl und bedingten Anweisungen ohne Alternativpfad, sollten dabei weiterhin zur Verfügung stehen. Diese Eigenschaften sind durch die DSL gegeben.
- **Hierarchie**  **:** Die Modellierung von Hierarchie wird durch zwei Typen von Gruppenknoten ermöglicht. Diese dienen zur Verknüpfung von Steuer- und Messvorschriften oder erlauben das Einfügen dieser ineinander. Damit ist diese Anforderung erfüllt.
- **Parallelität**  **:** Diese gewünschte Funktion wird durch das Threadknotenpaar realisiert, welche das Parallelisieren und Synchronisieren von Threads ermöglicht.
- **Modularität**  **:** Im SMVEditor ist das Laden von Modulen möglich. Diese realisieren die geforderte Modularität, indem sie Komponentenknoten bereitstellen mit denen die Modellierung erweitert werden kann.

6.2.2. Applikation

Neben den benötigen Funktionen zur Modellierung von Steuer- und Messvorschriften sollte die Applikation Eigenschaften haben, die das effiziente Erstellen von Prüfungen erlaubt. Im Folgenden wird überprüft, ob diese Anforderungen erfüllt worden sind.

- **Benutzbarkeit**  **:** Der SMVEditor soll das Erstellen von Prüfungen erleichtern. Ein wichtiger Aspekt dabei ist die Benutzbarkeit (engl. Usability) der implementierten Software, die überprüft worden ist. Die Vorgehensweise und die Ergebnisse der Überprüfung werden im Folgenden vorgestellt. Die gängige Vorgehensweise nach [17] ist die Bewertung der Anwendung durch Versuchspersonen. Der SMVEditor wurde von drei Mitarbeitern des Unternehmens überprüft. Dazu haben diese jeweils einen Graphen erstellt und zu einer ausführbaren SMV kompiliert. Anschließend wurde von jeder Versuchsperson ein Bewertungsbogen ausgefüllt. Als Bewertungsschema wurde die Schulnotenskala von 1 (sehr gut) bis 6 (ungenügend) gewählt. Die ISO/IEC 9126-1 [18] schlägt als Bewertungskriterien die Verständlichkeit, Erlernbarkeit, Bedienbarkeit, Attraktivität und Konformität vor, welche für den Bewertungsbogen übernommen worden sind. Tabelle 6.4 enthält die Ergebnisse der Befragung. Alle Versuchspersonen empfanden den SMVEditor als

leicht verständlich und konform zu den üblichen Designkonventionen von grafischen Oberflächen. Im Bereich Erlernbarkeit wurde die fehlende Hilfe und im Bereich Bedienbarkeit die fehlenden Hotkeys als großes Manko empfunden. Die Abwertung in der Kategorie Attraktivität kam durch die leichtgewichtigen Komponenten der Swingbibliothek, die sich in der Darstellung dem Betriebssystem nicht optimal anpassen, zustande.

Verständlichkeit	**Erlernbarkeit**	**Bedienbarkeit**	**Attraktivität**	**Konformität**
1	2	2	1	1
1	1	2	3	1
1	2	2	2	1

Tabelle 6.4.: Usability-Test

- **Lesbarer Code**  **:** Die Schwäche von XVCL, Quellcode nicht einrücken zu können, wurde zum Teil durch den Codeformatierer ausgemerzt. Jedoch erkennt dieser lediglich Codeblöcke, welche in geschweiften Klammern eingebettet sind. Nicht geklammerte bedingte Anweisungen werden beispielsweise nicht korrekt eingerückt.
- **Lauffähige SMV**  **:** Der SMVEditor kann den generierten C-Quellcode mit Hilfe des QCC-Compilers direkt in eine ausführbare Steuer- und Messvorschrift übersetzen.
- **Plattformunabhängigkeit**  **:** Der SMVEditor ist für Linux und Windows entwickelt worden. Damit eine Kompatibilität gewährleistet ist, besitzt die Anwendung für diese Systeme angepasste Codepassagen. Auf anderen Betriebssystemen wurde die Software nicht getestet, weshalb die Verträglichkeit nicht garantiert werden kann.
- **Versionierbarkeit**  **:** Die Applikation erlaubt durch das Speichern der SMVs in XML das versionierte Ablegen in einem Versionskontrollsystem. Die direkte Integration einer Versionskontrolle fehlt im SMVEditor.

7. Zusammenfassung und Ausblick

Dieses Kapitel stellt den Abschluss dieser Bachelorarbeit dar. In Abschnitt 7.1 werden die Zusammenfassung und das Ergebnis der Arbeit dargelegt. Abschließend folgt in Abschnitt 7.2 ein Ausblick auf zukünftige Aufgaben und Optimierungsmöglichkeiten.

7.1. Zusammenfassung

In dieser Arbeit wurde eine modellgetriebene Lösung zur Entwicklung von Steuer- und Messvorschriften realisiert. Diese Lösung stellte sich als praktikabel, jedoch im Rahmen der initialen Implementierung als sehr zeitaufwendig heraus.
In Kapitel 4 wurde ein Metamodell entwickelt, welches den Ablauf von Prüfungen durch einen gerichteten Graphen beschreibt und dabei die Anforderungen aus Kapitel 3 weitestgehend erfüllt. Das Metamodell bietet neben einem festgelegten Satz von Elementen zur Steuerung und Strukturierung einer Prüfung auch eine Schnittstelle zur Einbindung von Modulen, die als Erweiterung dienen. Eine Instanz des Metamodells wird durch eine dem UML-Aktivitätsdiagramm ähnliche domänenspezifische Sprache modelliert, welche die grafische Entwicklung von Prüfungen erlaubt. Aufgrund der einheitlichen Struktur und der Möglichkeit zur Zerlegung der Vorschriften durch Gruppenknoten ist die Wartbarkeit stark verbessert worden.
Der Entwurf wurde in Kapitel 5 in Form des SMVEditors implementiert. Dieser basiert auf Java und erlaubt es Steuer- und Messvorschriften zu verwalten und mit Hilfe von XVCL ausführbare Prüfungen zu erzeugen. XVCL dient als Schicht zwischen dem Metamodell und dem C-Quellcode, wodurch die direkte Transformation des Metamodells in Quellcode vermieden wird. Dies hat den Vorteil, dass bei Bedarf Templates ausgewechselt werden können. Außerdem ist die Transformation des Metamodells in XVCL weniger komplex, da die vom XVCL-Prozessor bereitgestellten Funktionen nicht durch eine eigene Implementierung realisiert werden müssen.
Zum Abschluss wurde in Kapitel 6 die Implementierung evaluiert. Dort stellten sich kleine Schwächen der Implementierung heraus, für die in Abschnitt 7.2 Lösungen vorgeschlagen werden. Als größte Schwäche der Implementierung sticht der Validator heraus, da dieser das Erzeugen von ausführbaren SMVs massiv verlangsamt.

7.2. Ausblick

Der SMVEditor soll nach der Umstellung der PPS zum Einsatz kommen und die Programmierung von Steuer- und Messvorschriften in PEARL90 ablösen. Die in dieser Arbeit vorgestellte Implementierung stellt dabei nur das Werkzeug für die Modellierung und Generierung dar, liefert jedoch keine Modulsammlung mit. Die Module sollten im Laufe der Umstellung der PPS mitentwickelt werden und können, dank der flexiblen Einbindung durch den SMVEditor, iterativ verbessert und erweitert werden.
Ein weiterer Punkt ist die Bewährung des Metamodells und der DSL in der Praxis. Bei auftretenden Schwächen ist die Einarbeitung von Anpassungen möglich. Dazu genügt die Erweiterung kleiner Passagen im Quellcode.
Neben diesen Aufgaben gibt es die Möglichkeit den SMVEditor bezüglich seiner Geschwindigkeit zu optimieren. In der Evaluation wurde festgestellt, dass die Validierung von Metamodellinstanzen sehr langsam ist, weil das Anstoßen des QCC zur Parametervalidierung sehr viel Zeit benötigt. Eine schnelle und einfache Lösung wäre ein verbesserter Einsatz des Compilers, indem alle Parameterzuweisungen der Knoten in einer einzigen Quelltextdatei gesammelt und kompiliert würden. Dies würde zu einem einzigen Start des Compilers führen und somit die Geschwindigkeit der Parametervalidierung drastisch erhöhen. Beim Auftreten eines Compilerfehlers wird die Fehlerzeile interpretiert, um den zugehörigen Knoten zu lokalisieren.
Eine im Verhältnis zum Nutzen sehr umfangreiche Lösung wäre die Entwicklung eines Parsers, welcher die Parameter überprüft.

A. Anhang

A.1. Die Firma Miele & Cie. KG

Die Firma Miele & Cie. KG ist der führende Hersteller von Haushalts- und gewerblichen Küchengeräten in Deutschland. Der Leitspruch des Familienunternehmens lautet „Immer besser“ und ist seit der Gründung Bestandteil der Unternehmensphilosophie.
Das Unternehmen wurde am 01.07.1899 von Carl Miele und Reinhard Zinkmann in Herzebrock gegründet. Zu Beginn produzierte der Betrieb in einer Säge- und Kornmühle mit elf Mitarbeitern Milchzentrifugen und Butterschleudern. Noch im Jahr der Gründung wurde die erste Waschmaschine produziert und verkauft. Im Jahr 1907 wechselte das Unternehmen den Standort von Herzebrock nach Gütersloh, dem heutigen Hauptsitz. Dort wurde das Produktsortiment stetig erweitert. 1916 vergrößerte sich das Unternehmen, indem ein weiteres Werk in Bielefeld zur Produktion von Elektromotor- und Milchzentrifugenteilen erworben wurde. 22 Jahre nach der Gründung produzierte Miele den ersten Geschirrspüler Europas und wurde drei Jahre später der größte Zentrifugenproduzent in Deutschland. 1950 gehörte der Betrieb zu den führenden Motorradherstellern in Deutschland, jedoch wurde später die Produktion zugunsten von Waschmaschinen und Geschirrspülern eingestellt. Im Jahr 1965 vergrößerte sich der Betrieb nochmals um ein Werk für Waschmaschinen und Melkanlagen und 1973 um eine weitere Fabrik für Küchenmöbel in Warendorf [19].
Geführt wird das Familienunternehmen seit dem Jahre 2004 von Dr. Markus Miele und Dr. Reinhard Zinkann. Heute produziert die Firma Miele & Cie. KG an elf Standorten weltweit. Acht Werke befinden sich dabei in Deutschland. Im Ausland befindet sich jeweils ein Werk in China, Österreich und Tschechien. Aktuell sind insgesamt über 16500 Mitarbeiter im Unternehmen beschäftigt. Von den insgesamt 10000 Mitarbeitern in Deutschland sind allein 5000 Mitarbeiter in Gütersloh (Stand April 2010) angestellt. Im Geschäftsjahr 2009/2010 wurde ein Gesamtumsatz von 2,83 Milliarden Euro erwirtschaftet [20].

A.2. XVCL-Frames

```
<x-frame name="main">
  #include <pthread.h>
  #include <stdio.h>
  #include <stdlib.h>
  #include <stdbool.h>

  <while using-items-in="moduleheaders">
    <select option="moduleheaders">
      <option value="" comp-operator="!=">
      <adapt x-frame="?@moduleheaders?Header.xvcl"/>
      </option>
    </select>
  </while>

  <while using-items-in="threadnumbers">
    <select option="threadnumbers">
      <option value="0" comp-operator="!=">
      void * thread_<value-of expr="?@threadnumbers?"/>(void *c);
      </option>
    </select>
  </while>

  int main(int argc, char *argv[]) {
    <adapt x-frame="thread_0.xvcl"/>
    return 0;
  }

  <while using-items-in="threadnumbers">
    <select option="threadnumbers">
      <option value="0" comp-operator="!=">
      void * thread_<value-of expr="?@threadnumbers?"/>(void *c) {
        <adapt x-frame="thread_?@threadnumbers?.xvcl"/>
        return NULL;
      }
    </option>
  </select>
  </while>
</x-frame>
```

Listing A.1: Haupt-Frame

```
<x-frame name="loop">
  for(int <value-of expr="?@loopvarname?"/> = 0;
    <value-of expr="?@loopvarname?"/> < <value-of expr="?@loopcount?"/>;
    <value-of expr="?@loopvarname?"/>++) {
</x-frame>
```

Listing A.2: Schleifen-Frame

```
<x-frame name="endloop">
  }
</x-frame>
```

Listing A.3: Schleifenende-Frame

```
<x-frame name="if">
  if(<value-of expr="?@ifexpression?"/>) {
</x-frame>
```

Listing A.4: Wenn-Frame

```
<x-frame name="endif">
  }
</x-frame>
```

Listing A.5: Wennende-Frame

```
<x-frame name="fork">
  pthread_t thread_<value-of expr="?@thread?"/>_id;
  pthread_create(&thread_<value-of expr="?@thread?"/>_id,
    NULL, thread_<value-of expr="?@thread?"/>,
    NULL);
</x-frame>
```

Listing A.6: Gabel-Frame

```
<x-frame name="join">
  pthread_join(thread_<value-of expr="?@thread?"/>_id, NULL);
</x-frame>
```

Listing A.7: Synchronisation-Frame

A.3. Inhalt der CD

Die beiliegende CD enthält die digitale Fassung der Bachelorarbeit im PDF-Format. Der SMVEditor liegt als Quellcode mit zugehöriger JavaDoc Dokumentation und in einer kompilierten Fassung für Windows und Linux vor.

Literaturverzeichnis

[1] Peter Marwedel: *Eingebettete Systeme*. 1. Auflage. Springer-Verlag, 2008. – ISBN 978–3540340485

[2] Thomas Stahl, Markus Völter, Sven Efftinge, Arno Haase: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2. Auflage. dpunkt.verlag, 2007. – ISBN 978–3898644488

[3] Thomas Stahl, Peter Rossbach, Wolfgang Neuhaus: *Model Driven Architecture*. Artikel in: Javamagazin 9, 2003

[4] Heide Balzert: *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2*. 2. Auflage. Spektrum Akademischer Verlag, 2004. – ISBN 978–3827411624

[5] Christoph Kecher: *UML 2: Das umfassende Handbuch*. 4. Auflage. Galileo Computing, 2011. – ISBN 978–3836217521

[6] Object Management Group: *Unified Modeling Language 2.3 Superstructure*. http://www.omg.org/spec/UML/2.3/Superstructure/PDF/, 2009

[7] Karl Eilebrecht, Gernot Starke: *Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung*. 3. Auflage. Spektrum Akademischer Verlag, 2010. – ISBN 978–3827425256

[8] Martin Fowler: *Patterns of Enterprise Application Architecture*. 1. Edition. Addison-Weslay, 2002. – ISBN 978–0321127426

[9] XVCL Website: *XVCL: A Tutorial*. http://xvcl.comp.nus.edu.sg/cms/index.php/learn-xvcl/xvcl-tutorial-1.html, 2011

[10] Matthias Meier: *Merkmalbasierte statische Konfigurierung von MPSoCs*. Technische Universität, Dortmund, Deutschland, 2009

[11] Wikipedia: *Frame Technology*. http://en.wikipedia.org/wiki/Frame_Technology_(software_engineering), 2011

[12] Bassett, Paul G.: *Framing Software Reuse: Lessons From The Real World*. 1. Edition. Prentice Hall, 1996. – ISBN 978–0133278590

[13] XVCL Website: *How Does XVCL Work?* http://xvcl.comp.nus.edu.sg/overview_technical_a.php, 2011

[14] National University of Singapore, Netron Inc.: *XML-based Variant Configuration Language (XVCL) Specification Version 2.10*. http://xvcl.comp.nus.edu.sg/cms/index.php, 1996

[15] Wikipedia: *Schichtenarchitektur*. http://de.wikipedia.org/wiki/Schichtenarchitektur, 2011

[16] Edsger Wybe Dijkstra: *Notes On Structured Programming*. Technological University, Eindhoven, Netherlands, 1972

[17] Wikipedia: *Usability-Test*. http://de.wikipedia.org/wiki/Usability-Test, 2011

[18] International Organization for Standardizazion: *ISO/IEC 9126-1: Software Engineering - Product Quality*. Genf, Schweiz, 2001

[19] Wikipedia: *Miele*. http://de.wikipedia.org/wiki/Miele, 2011

[20] Miele Website: *Starke Marke/Erreichtes*. http://www.miele-nachhaltigkeit.de/international/de/nachhaltigkeit/2902.htm, 2011

Abkürzungsverzeichnis

CAN	Controller Area Network
DSL	Domain-Specific Language
GUI	Graphical User Interface
MDSD	Model-Driven Software Development
MVC	Model-View-Controller
OMG	Object Managment Group
POSIX	Portable Operating System Interface
PPS	Prüfplatzsoftware
QCC	QNX Compiler Collection
RTOS-UH	Realtime Operating System - University Hannover
SMV	Steuer- und Messvorschrift
SPC	XVCL-Spezifikationsdatei
UML	Unified Modelling Language
XML	Extensible Markup Language
XVCL	XML-based Variant Configuration Language

Abbildungsverzeichnis

Listings

Tabellenverzeichnis