Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Continuous Integration von AUTOSAR Software

Andreas Razmyslov

Studiengang:	Master of Science Informatik
Prüfer/in:	Prof. Dr. rer. nat. Stefan Wagner
Betreuer/in:	Sebastian Vöst, M.Sc. Dennis Müller, M.Sc. Verena Käfer, M.Sc.
Beginn am:	29. November 2017
Beendet am:	29. Mai 2017
CR-Nummer:	D.2.5, I.7.2

Kurzfassung

Die vorliegende Masterarbeit evaluiert und optimiert ein Konzept zu Continuous Integration (CI) von AUTOSAR Software der Firma Vector Informatik GmbH. Aus der Evaluierung des Konzepts folgte, dass die manuelle Konfiguration der Runtime Environment einer AUTOSAR Software, bestehend aus den Schritten Port-Mapping und Runnable-Task-Mapping, zu Problemen bei der Umsetzung der CI-Praktiken führte. Aus diesem Grund wurde als Ziel der Arbeit eine Optimierung des Konzepts durch die Automatisierung der Konfiguration festgelegt. Dazu wurde ein Ansatz zur Automatisierung des Port-Mappings und des Runnable-Task-Mappings entwickelt, welches sowohl auf Ideen aus wissenschaftlichen Publikation als auch Industriepraktiken basiert. Darauf folgte die Implementierung des Ansatzes als eine Groovy-Applikation, um anschließend eine Evaluierung basierend auf einem aktuellen AUTOSAR Projekt durchzuführen. Die Evaluierung lieferte ein positives Ergebnis in Bezug auf das Runnable-Task-Mapping, wohingegen die Verwendung des Port-Mappings für CI nur mit Einschränkungen möglich ist.

Abstract

This master thesis evaluates and optimizes a Continuous Integration (CI) concept for AUTOSAR software from Vector Informatik GmbH. The evaluation of the concept has shown, that the manual configuration of the runtime environment of a AUTOSAR software, consisting of the steps of port mapping and runnable task mapping, led to problems in the implementation of CI practices. For this reason, the goal of the work was to optimize the concept by automating the configuration. For this purpose, an approach for the automation of port mapping and runnable task mapping has been developed, which is based both on ideas from scientific publications as well as on industrial practices. This was followed by the implementation of the approach as a Groovy application in order to carry out an evaluation based on a current AUTOSAR project. The evaluation yielded a positive result with respect to the runnable task mapping, whereas the use of port mapping for CI is only possible with restrictions.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Um den Wunsch der Kunden nach mehr innovativen Funktionalitäten in Fahrzeugen zu erfüllen und sich dadurch im starken Wettbewerber auf dem Automobilmarkt durchzusetzen, sind Original Equipment Manufacturers (OEMs) (übersetzt Originalausrüstungshersteller) gefordert, in ihren Fahrzeugen mehr Steuergeräte einzusetzen und dafür komplexere Software zu entwickeln.[SAA10] Die branchenübliche Stückkostenkalkulation, verbunden mit engen Budgets, resultiert in einem großen Kosten-, Zeit- und Qualitätsdruck für die Entwickler. Folglich sind sowohl OEMs als auch Zulieferer gezwungen ihre Entwicklungsprozesse zu optimieren.[PBKS07]

Ein wichtiger Schritt in dieser Richtung war die Einführung von AUTOSAR (AUTomotive Open System ARchitecture). AUTOSAR ist eine offene und standardisierte Softwarearchitektur für Steuergeräte in Kraftfahrzeugen, welche in der seit 2003 bestehenden Partnerschaft von führenden Herstellern der Automobilindustrie konzipiert und umgesetzt wurde.[AUT03] Neben der Verwendung einer gemeinsamen Softwarearchitektur, lag der Fokus auch auf dem Einsatz von alternativen Entwicklungsmethoden und -praktiken in einer vom V-Modell dominierten Branche. Die Anwendung von agilen Methoden hat bereits in anderen Branchen gezeigt, dass es positive Effekte sowohl auf die Entwicklungszeit als auch auf die Kosten haben kann. Aus diesem Grund arbeiten Unternehmen der Automobilindustrie aktuell an der Veränderung bestehender Prozesse durch die Integration von agilen Methoden.[TLD+14]

Das Problem bei dem Einsatz des klassischen V-Modells liegt vor allem daran, dass die Test- und Integrationsphasen am Ende des Projektverlaufs liegen. Somit führt es häufig dazu, dass Fehler erst sehr spät erkannt werden und der erforderliche Aufwand diese zu beheben in den Schlussphasen am höchsten ist [SDD+04]. Dieses Problem ist besonders kritisch in der Entwicklung von eingebetteten Systemen, da hierbei die Entwicklung von Software und Hardware oftmals parallel verläuft. Die Software muss anschließend auf der Hardware integriert werden. Geschieht dieser Schritt am Ende der Projektlaufzeit, können zwischen Soft- und Hardware Kompatibilitätsprobleme auftreten, die erst sehr spät erkannt werden.[SDL09]

Im Jahr 2005 stellte Martin Fowler in einem Blogpost [FF06] das Konzept der Continuous Integration (CI) vor. Die Idee dahinter ist, dass neue Komponenten einer Software sofort integriert und getestet werden. Folglich sollen Fehler, so früh wie möglich, erkannt und behoben werden. Dieses Ziel soll vor allem erreicht werden, indem der Integrationsschritt

nicht mehr manuell, sondern automatisiert durchgeführt und durch Tools unterstützt wird, um mögliche Fehlerquellen, welche durch manuelles Arbeiten resultieren können, zu eliminieren und eine Durchführung der Integration mehrmals täglich zu ermöglichen.

Aufgrund der nun mittlerweile weiten Verbreitung von AUTOSAR Software herrscht auch hierbei der Wunsch nach dem Einsatz von CI während des Entwicklungsprozesses. Vector Informatik GmbH (Vector), ein Unternehmen mit dem Fokus auf eingebettete Systeme in der Automobilindustrie, hat zu diesem Zweck ein Konzept entwickelt, dessen Evaluierung und Umsetzung aber noch ausstehen.

1.2. Problembeschreibung

Der AUTOSAR Standard sieht für die Entwicklung von Steuergeräte-Software eine Architektur mit einer klaren Trennung in Abstraktionsschichten vor (siehe Abschnitt 2.1.2). Ganz oben befindet sich die Applikationsschicht, in der die Anwendung als Softwarekomponenten (SWCs) implementiert ist, welche auf dem Steuergerät ausgeführt werden soll. Die Basissoftware (BSW) stellt für die Software Dienste bereit, wie z.B. Buskommunikation, Speicherverwaltung und IO-Zugriff. Die zwischen den SWCs und der BSW liegende Schicht ist die Runtime-Environment (RTE), welche die Schnittstellen der beiden miteinander verbindet und somit die Kommunikation ermöglicht. Angenommen, dass sowohl die Applikation als auch die BSW fertig implementiert und konfiguriert sind, ist es erforderlich, die Kommunikation zwischen den beiden Schichten herzustellen. Dazu erfordert es die Konfiguration und die anschließende Generierung der RTE (siehe Abschnitt 2.1.3). Erst danach kann die Applikation auf dem Steuergerät ausgeführt bzw. getestet werden kann. Die Konfiguration und das Testen einer neuen Version der Applikation soll durch CI realisiert werden.

1.2.1. Vector-CI-Pipeline

Dazu wurde bei Vector ein Konzept zu der Vector-CI-Pipeline entwickelt, welche in der Abbildung 1.1 dargestellt wird. Die Entwickler der Applikation (in diesem Fall der OEM) und Vector teilen sich ein SVN (Apache Subversion)[1] Repository. Durch das Einchecken von einer neuen ARXML Datei in das Repository wird die Pipeline ausgelöst. Das CI-Testsystem läuft auf einem Jenkins[2]-Server. Jenkins führt nach einem Update des Repositorys nacheinander verschiedene Aufgaben und wird dabei von Vector Softwarewerkzeugen DaVinci Developer (siehe 2.2.2) und DaVinci Configurator Pro (Cfg Pro) (siehe 2.2.1) unterstützt.

Der erste Schritt nach dem Update ist der Import der neuen SWC Beschreibung in den Developer, wobei neue Instanzen der SWCs angelegt werden und bereits instanziierte SWCs aktualisiert

[1] https://subversion.apache.org/
[2] https://jenkins.io/

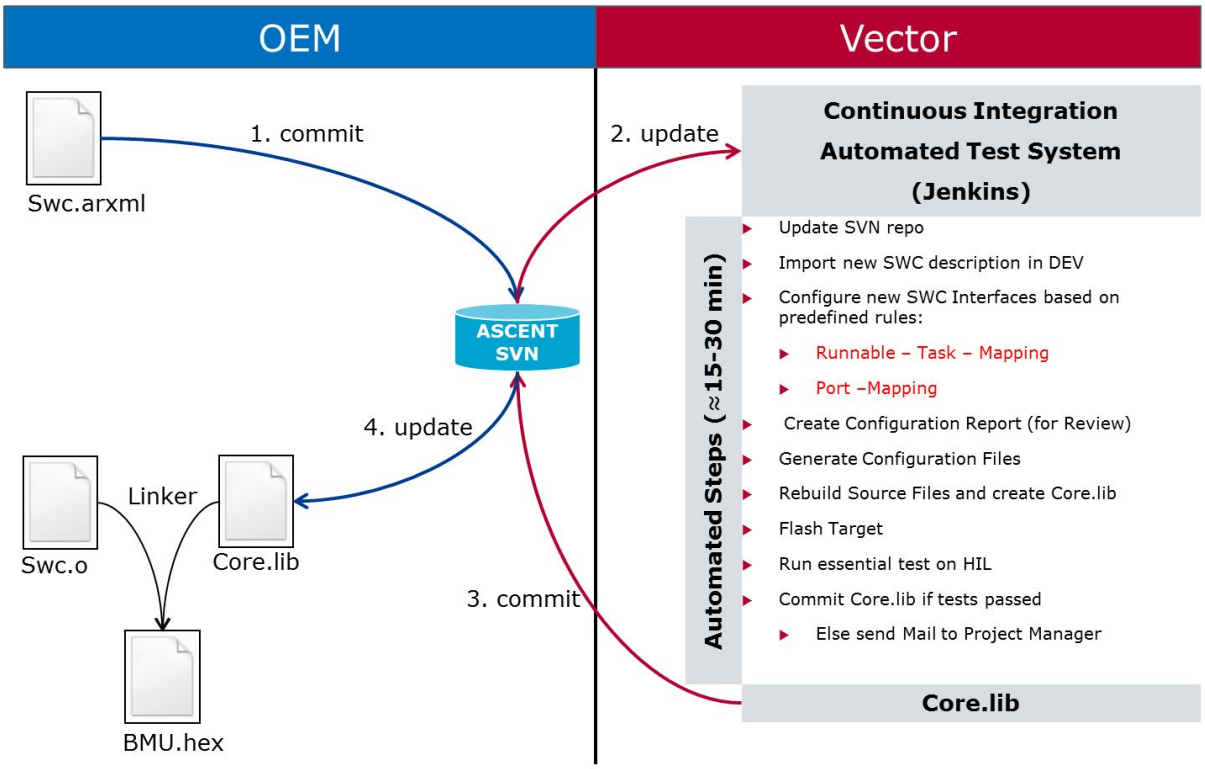


Abbildung 1.1.: Konzept der Vector-CI-Pipeline. Manuelle Schritte sind mit roter Schrift markiert. Übernommen aus [Gmb16a].

werden. Wenn neue Schnittstellen in Form von Ports und Runnables vorhanden sind, dann muss die Konfiguration von diesen durchgeführt werden. Zu der Konfiguration der Schnittstellen gehören die Teilschritte Port-Mapping (PM) und Runnable-Task-Mapping (RTM). Diese sind in der Abbildung rot markiert, da sie manuell durch einen Integrator durchgeführt werden müssen, unterstützt durch das Tool Cfg Pro. Danach kann die RTE generiert und der Quellcode der Applikation angepasst werden, was in der Datei Core.lib resultiert. Die konfigurierte Software wird dann auf dem Zielgerät installiert und am Schluss automatisch getestet. Sind die Tests erfolgreich abgeschlossen, wird die Core.lib in das Repository eingecheckt und kann vom OEM für das Linken mit den Objektdateien verwendet werden.

1.2.2. Validierung der Pipeline

In seinem Artikel „*Continuous Integration*“ (2006) [FF06] benennt Martin Fowler die Kriterien, die ein CI System erfüllen muss. Dieser Abschnitt dient dem Zweck, zu untersuchen, ob diese Kriterien auch von der Vector-CI-Pipeline erfüllt werden und in welchen Bereichen noch Verbesserungsbedarf besteht.

1. Pflege eine Versionsverwaltung

Da ein Softwareprodukt meist aus mehreren Dateien besteht, muss man sicherstellen, dass diese nicht verloren gehen und auch an einem Ort gelagert werden, damit das Produkt gebaut werden kann. Für die Verwaltung von Dateien wurde Software zur Versionsverwaltung entwickelt, welche genau diese Aufgaben erfüllt. In der Vector-CI-Pipeline kommt für diesen Zweck ein Apache Subversion Repository zum Einsatz, welches von allen Projektbeteiligten genutzt werden kann. Somit ist diese Bedingung erfüllt.

2. Automatisches Bauen

Automatisches Bauen besagt, dass alle Aufgaben, die für das Erstellen des Produkts aus den Quelldateien notwendig sind, voll automatisiert werden sollten. Dieser Punkt wird von der Pipeline nicht erfüllt, denn die Konfiguration der RTE ist ein manueller Schritt, welcher im Cfg Pro von einem Integrator durchgeführt wird. Zusätzlich besteht diese Aufgabe hauptsächlich aus zeitaufwendigen, monotonen Klickarbeit, welche zu Fehleranfälligkeit neigt.

3. Mache dein Produkt selbst testend

Um sicherzustellen, dass das resultierende Produkt sich so verhält, wie man es erwartet, kann man bei jeder neuen Version automatische Tests durchführen lassen, um Fehlverhalten im Programmablauf festzustellen. Wie man an der Abbildung 1.1 erkennen kann, sind automatische Test in der Pipeline bereits enthalten.

4. Jeder sollte jeden Tag seinen Stand einchecken

Arbeiten mehrere Beteiligte an einem Projekt, kann es dazu kommen, dass diese bei der Implementierung von Softwarefunktionalitäten in einen Konflikt geraten können. Um diese Problematik so schnell wie möglich feststellen zu können, erfordert es regelmäßige Kommunikation zwischen den Entwicklungspartnern. Durch das tägliche Einchecken des aktuellen Standes von jedem Mitarbeiter, erfahren die Beteiligten sehr schnell, welche Fortschritte in anderen Teilen gemacht worden sind und wo es möglicherweise zu Konflikten kommen kann. Bevor man aber seinen Stand einchecken kann, ist es notwendig das Produkt zu bauen und zu testen. Dies wird dadurch erschwert, dass die Pipeline manuelle Schritte enthält. Erfahrungen aus vergangenen Projekten zur Entwicklung von AUTOSAR Software haben gezeigt, dass die Konfiguration der RTE häufig erst in einem umfassenden Integrationsschritt durchgeführt wird, welcher sehr zeitaufwändig ist und den Entwicklungsprozess verzögert, da die Entwicklungspartner auf das Ergebnis warten müssen [Zee12]. Somit werden Konflikte oder Fehler in der Applikation erst sehr spät erkannt und es resultiert oftmals in einem höheren Aufwand, als wenn man die Probleme gleich nach dem Commit in das Repository festgestellt hätte.

5. Jeder Commit sollte das Projekt auf einem Integrationsserver bauen

Die Verwendung eines Integrationsservers soll dem Zweck dienen, dass die Applikation sich immer in einem stabilen Zustand befindet. Wichtig dabei ist, dass Entwickler auf dem neuesten Stand in der gleichen Umgebung arbeiten. Möchte einer der Entwickler einen neuen Stand einchecken, wird dieser gezwungen, davor einen Update zu machen und seine Aufgabe gilt erst als erledigt, wenn der Stand auf dem Integrationsserver lauffähig ist. Vector verwendet in seiner Lösung einen zentralen Jenkins-Testrechner, welcher mit dem Zielsteuergerät verbunden ist. Die notwendigen Aufgaben werden von Jenkins ausgeführt und der Entwickler wird gleich benachrichtigt, wenn etwas schief gelaufen ist.

6. Fehlerhafte Applikation sollte sofort repariert werden

Eines der Kernpunkte der CI ist, dass Entwickler auf einem stabilen Stand arbeiten. Folglich sollte sichergestellt werden, dass wenn es beim Bauen der Software zu fehlerhaften Verhalten kommt, diese so schnell wie möglich beheben werden sollte. Die Suche nach den Fehlern sollte dabei aber nicht im Hauptstand auf dem Integrationsserver passieren, sondern in der lokalen Entwicklungsumgebung des Entwicklers. Damit die restlichen Beteiligten nicht von den Fehlern betroffen werden, braucht man eine Möglichkeit, wie man zu einen früheren stabilen Hauptstand zurückkommen kann. Dank der Verwendung von Apache Subversion wird diese Option in der Pipeline zur Verfügung gestellt.

7. Der Erstellungsprozess sollte schnell sein

Ein Entwickler ist erst von der Integrationsarbeit seines Updates für die Applikation befreit, wenn alle Schritte der Pipeline durchlaufen sind. Erst danach kann die Entwicklung der Applikation fortgesetzt werden. Ein langer Erstellungsprozess führt dazu, dass weniger Zeit für die Entwicklung bereit steht, was sich folglich auch auf die Entwicklungszeit der Software negativ auswirkt. Eines der wichtigsten Ziele der CI ist, diesen Prozess zu beschleunigen. Die obere Abbildung der Vector-CI-Pipeline zeigt, dass für den gesamten Erstellungsprozess eine Gesamtlaufzeit von 15 - 30 Minuten gefordert ist. Praktisch wird dieses Ziel aber nur selten erreicht, da die manuelle Konfiguration der RTE, wie unter 1.2.2 bereits beschrieben, ein zeitaufwendiger Prozess ist. Dadurch kommt es zu einem Engpass, welcher drastische Auswirkungen auf die Entwicklungsdauer hat, denn erst nach der Durchführung kann man durch Test sicherstellen, dass die funktionalen Anforderungen erfüllt worden sind.

8. Verwende zum Testen einen Klon der Produktivumgebung

Um unerwartete Probleme zu vermeiden, sollte die Applikation unter den gleichen Bedingungen getestet werden, wie es später im Produktivsystem zum Einsatz kommt. Das bedeutet, dass

das Betriebssystem, die Hardware und andere unterstützende Software gleich sein müssen. Für das Testen der Applikation in der Vector-CI-Pipeline steht das Steuergerät zur Verfügung, welches auch später in den Fahrzeugen zum Einsatz kommen wird. Auch die BSW wird auf jedem dieser Geräte identisch sein. Diese Anforderung ist somit erfüllt.

9. Jeder sollte, so einfach wie möglich, zu der neuesten Version kommen

Bei der Entwicklung von Software kann es vorkommen, dass Entwickler und Stakeholder unterschiedliche Vorstellungen haben können, wie das Endprodukt aussehen sollte. Die große Motivation hinter agilen Entwicklungsmethoden ist, diese Unklarheiten und Konflikte so schnell wie möglich zu erkennen und zu beseitigen. Um diesen Teil zu vereinfachen, solle jeder Projektbeteiligte die Möglichkeit haben, auf die neueste lauffähige Version der Applikation zugreifen zu können. Dies wird in diesem Fall durch das Apache Subversion Repository, welches sich Vector und der OEM teilen, erreicht, denn nach dem Erzeugen wird die Applikation im Repository abgelegt.

10. Jeder kann sehen, was gerade passiert

Bei diesem Punkt geht es um die Visualisierung der Schritte des Erzeugungsprozesses, die gerade durchlaufen werden und an welchen dieser Schritte der Prozess möglicherweise scheitert. Die Visualisierung soll den Beteiligten helfen, eine schnellere Übersicht über den aktuellen Stand des Projekts zu erlangen. Da die automatischen Aufgaben der Vector-CI-Pipeline durch Jenkins ausgeführt werden, wird die Visualisierung auch von diesem Tool übernommen. Im Benutzerinterface des Jenkins werden die Status aller ausgeführten Aufgaben dargestellt.

11. Automatische Verteilung

In Fällen, bei denen mehrere Testsysteme verwendet werden, kommt es oftmals dazu, dass Dateien zwischen den Systemen kopiert werden müssen. Manuelles Kopieren und Verteilen von Dateien kann z.B. durch Unkonzentriertheiten zu Fehlern führen. Eine Automatisierung der Verteilung wäre aus diesem Grund vorteilhaft. Da bei der Vector-CI-Pipeline nur ein Testsystem zum Einsatz kommt und alle notwendigen Daten im Apache Subversion Repository gespeichert werden, ist dieser Punkt in dem Fall zu vernachlässigen.

1.2.3. Schlüsse aus der Validierung

Die Validierung hat gezeigt, dass das vorliegende Konzept zur Vector-CI-Pipeline nicht alle Praktiken anwendet und folglich noch Raum für Optimierung bietet. Die Praktiken 2, 4 und 7 finden aktuell keine Anwendung und der gemeinsame Grund dafür liegt in der manuellen Konfiguration der RTE. Die manuelle Konfiguration ist der Flaschenhals in der Pipeline, denn

die aktuelle Durchführung findet nur wöchentlich bzw. monatlich statt. Dies führt dazu, dass die Applikation erst nach der Durchführung getestet werden kann. Folglich können Fehler in der Applikation und Konflikte zwischen den Entwicklungspartnern mit hoher Wahrscheinlichkeit erst sehr spät erkannt werden. Die späte Erkennung von Problemen hat zum einen Einfluss auf die Entwicklungszyklen, denn ein Zyklus wird erst als abgeschlossen angesehen, wenn die darin festgelegten Funktionalitäten korrekt implementiert und getestet sind, zum anderen kann es starke Auswirkungen auf die Kosten des Projekts haben, denn eine Fehlererkennung in einer späteren Phase kann preislich sehr viel höher sein [SDD+04]. Aus diesen Gründen erscheint eine Überlegung für alternative Konzepte zur Durchführung der Konfiguration sinnvoll.

Eine mögliche Antwort auf die Problematik wäre die Automatisierung des PM und RTM. Durch eine Automatisierung könnte die gewünschte Erzeugungszeit von 15 - 30 Minuten zu jeder Zeit erreicht werden, denn Eingriffe und Verzögerungen durch einen Integrator wären eliminiert. Dadurch können neue Funktionalitäten der Applikation sofort getestet und in das Produkt integriert werden. Zeitnahe Erkennung der Fehler und Konflikte würde Probleme bei späteren Integrationstest vermeiden.

1.3. Ziel

Wie in den vorangegangenen Unterkapiteln erläutert, führt der nicht-optimale Erzeugungsprozess der Vector-CI-Pipeline zu negativen Auswirkungen auf die Entwicklungszyklen von AUTOSAR Software. Als Grund für die Probleme wurde in erster Linie die manuelle Konfiguration der RTE identifiziert. Folglich liegt der Fokus dieser Arbeit auf der Entwicklung von automatisierten Konzepten zum PM und RTM.

Dazu sollen im Laufe dieser Arbeit folgende Forschungsfragen beantwortet werden:

1. Wie sieht der aktuelle Stand der Forschung in relevanten Forschungsbereichen aus?
2. Wie sieht ein mögliches Konzept zur Automatisierung der Konfiguration von der RTE aus?
3. Wo liegen die Vorteile und Grenzen dieses Konzepts?

Erkenntnisse zum Stand der Forschung in relevanten Themengebieten sollen durch eine Literaturrecherche gewonnen werden. Im Laufe einer Beurteilung der daraus gewonnen Informationen soll geprüft werden, ob eine Integration dieser in einem möglichen Konzept für die automatische Konfiguration der RTE umsetzbar ist. Darauffolgend soll ein konkretes Konzept präsentiert werden und dessen Machbarkeit in Form einer Implementierung gezeigt wird. Zum Schluss soll eine Evaluierung des Konzepts basierend auf echten Industriedaten die Vorteile und Grenzen der automatisierten Integration von AUTOSAR Software aufzeigen.

1.4. Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: Hier werden werden die Grundlagen dieser Arbeit beschrieben, wobei der große Fokus hierbei auf den Grundlagen von AUTOSAR liegt. Empfohlen wird dieses Kapitel vor allem für Leser, welche noch nicht mit AUTOSAR vertraut sind.

Kapitel 3 – Verwandte Arbeiten: Hier werden themenrelevante, wissenschaftliche Publikationen vorgestellt. Zum einen sollen die Gemeinsamkeiten zum Thema der Masterarbeit erläutert werden, zum anderen auch die Unterschiede, um eine Abgrenzung dieser Arbeit zu verdeutlichen.

Kapitel 4 – Konzeptentwicklung: In diesem Kapitel werden die Konzepte für das automatische PM und RTM entwickelt und vorgestellt.

Kapitel 5 – Implementierung: Dieses Kapitel befasst sich mit der Implementierung der Applikation zur automatisierten Konfiguration der RTE.

Kapitel 6 – Evaluierung: In diesem Kapitel wird das Konzept gegen echte Industriedaten evaluiert, um Vorteile und Grenzen zu identifizieren.

Kapitel 7 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2. Grundlagen

2.1. AUTOSAR

2.1.1. Einführung

Im Jahr 2003 haben sich Automobilhersteller, Automobilzulieferer und Werkzeughersteller zusammengeschlossen, um einen offenen Standard für die Softwarearchitektur von Steuergeräten in Fahrzeugen zu entwickeln. Durch die Erwartungen von steigender Komplexität in zukünftigen Projekten, war die Motivation der Zusammenarbeit, die notwendigen Rahmenbedingungen zu kreieren, um die Komplexität handhaben zu können. Zusätzlich erhoffte man sich, für alle Projektbeteiligten die Kosten zu minimieren, aber dennoch die Qualität der Software zu verbessern. Das Ergebnis dieser Kooperation war AUTOSAR (AUTomotive Open System ARchitecture). [KF09]

Zur Umsetzung wurden von den Partnern folgende Ziele festgelegt [AUT03]:

- Erfüllung von zukünftigen Anforderungen zu Verfügbarkeit, Sicherheit und Wartbarkeit
- Bessere Skalierbarkeit und Flexibilität der Funktionen
- Steigerung des Angebots für seriengefertigte Produkte
- Minimierung der Komplexität und des Risikos
- Optimierung der Kosten von skalierbaren Systemen

2.1.2. Architektur

Abbildung 2.1 zeigt die Softwarearchitektur für ein Steuergerät. Wie man erkennen kann, ist die Software in mehrere Schichten unterteilt. Die Motivation hinter der Schichtenarchitektur war die Trennung der Hardwareaspekte des Steuergeräts, wie Prozessoreigenschaften, Eigenschaften des Steuergerätes und der Sensoren und Aktoren, in der Software. Diese Abstraktion hat zum einen den Vorteil, dass die Umsetzung der Schichten konfliktfrei von mehreren Parteien durchgeführt werden kann. Beispielsweise können die Applikation, die BSW und die Hardware allesamt von verschiedenen Parteien entwickelt werden.

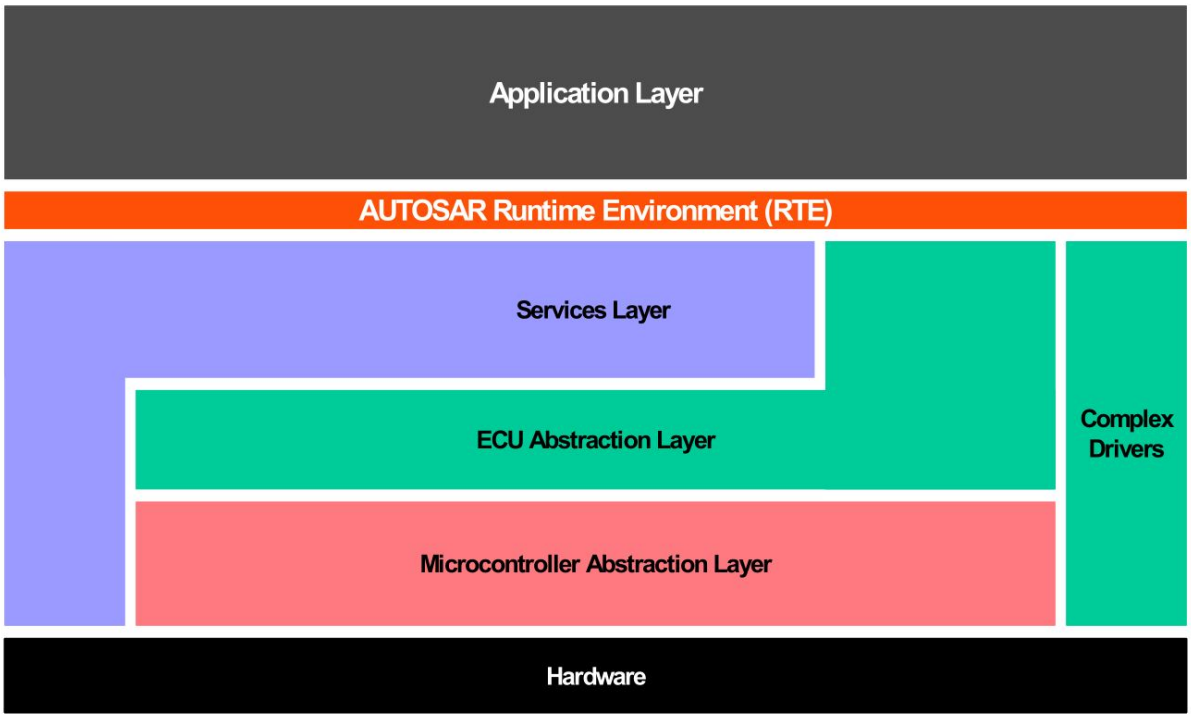


Abbildung 2.1.: Darstellung des AUTOSAR Schichtenmodells

Ganz oben befindet sich die Anwendungsschicht, welche aus einzelnen SWCs zusammengesetzt ist. In den SWCs sind die Funktionalitäten der Anwendung umgesetzt. Man unterscheidet dabei zwischen Anwendungs-SWCs und Sensor/Aktor-SWCs. Durch diese Trennung ist es möglich Anwendungs-SWCs problemlos auf andere Steuergeräte zu portieren, denn diese sind im Gegensatz zu Sensor/Aktor-SWCs hardwareunabhängig. Durch die Teilung der Applikation in mehrere SWCs ist zwischen ihnen oftmals Kommunikation notwendig.

Darunter liegt die RTE, welche für die Kommunikation zwischen den SWCs untereinander und den SWCs und der BSW verantwortlich ist. Die Anforderungen an die Kommunikation sind stark von der Funktionalität der konkreten Anwendung abhängig. Folglich muss zur Herstellung der Kommunikation die RTE für jede neue Applikation angepasst und neu generiert werden. Damit die RTE konfiguriert werden kann, benötigt man die Beschreibung der Schnittstellen der SWCs und der BSW.

Unter der RTE-Schicht liegt die BSW, welche in weitere Einzelteile geteilt werden kann. Unter anderem besitzt es eine Serviceschicht, welche der Anwendung Dienste wie Speicherverwaltung, Kommunikationsfunktionen und Diagnose zu Verfügung stellt. Des weiteren gibt es die Steuergeräteabstraktionsschicht. Es hat die Aufgabe steuergerätspezifische Eigenschaften zu abstrahieren. Für die Konfiguration und die Initialisierung des Steuergerätes ist die Mikrocontrollerabstraktionsschicht verantwortlich. Durch die Hardwarenähe ist es notwendig, diese Schicht der BSW neu zu konfigurieren, wenn das Steuergerät ausgetauscht wird. Die Schicht Complex Device Drivers wird verwendet, wenn die Notwendigkeit besteht aus der Anwendung

direkt auf die Hardware zuzugreifen. Dadurch wird die Abstraktion umgangen, weshalb es nur speziellen Fällen, wie z.B. bei ressourcen-kritischen Anwendungen, zum Einsatz kommen sollte. [KF09]

2.1.3. Konfiguration der RTE

Wie zuvor erwähnt, ist die RTE für die Kommunikation zwischen der Applikation und der BSW verantwortlich. Dadurch, dass die BSW für jedes Steuergerät angepasst wird, muss eine Implementierung der RTE für jedes Steuergerät neu generiert werden. Bevor das gemacht werden kann, muss die RTE konfiguriert werden. Wie dieser Prozess aussieht, wird im Folgenden erklärt. Zuvor werden aber noch wichtige Begriffe aus AUTOSAR erläutert, damit der Schritt der Konfiguration einfacher nachvollzogen werden kann.

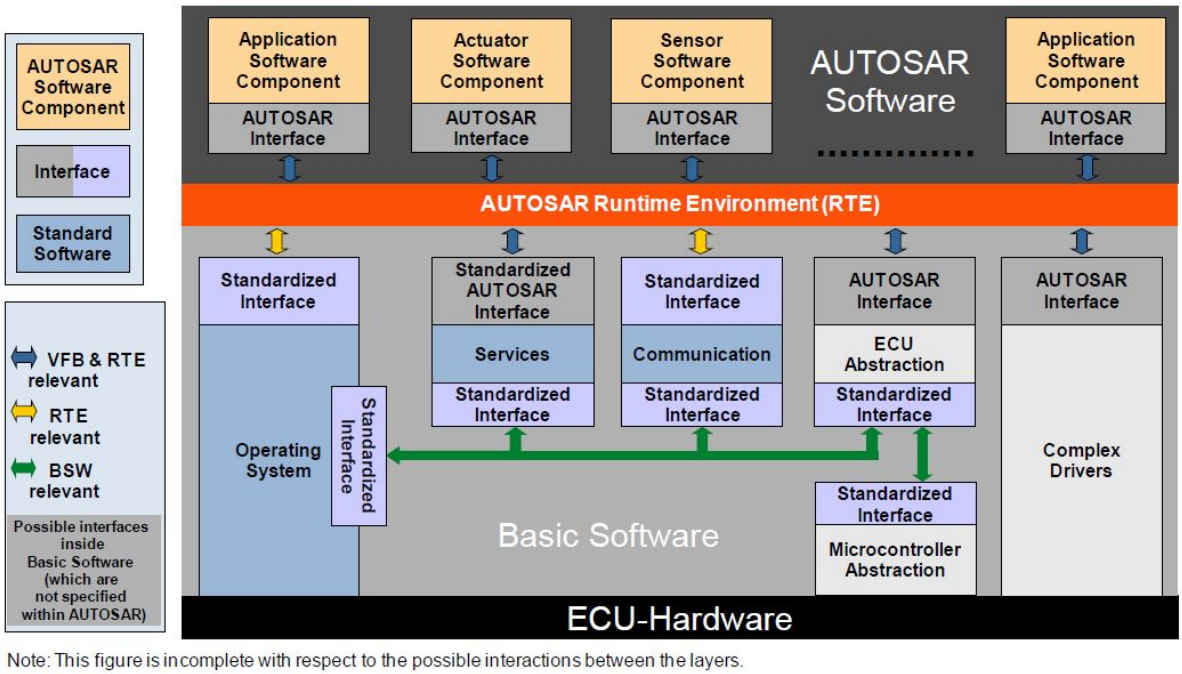
Übersicht

Abbildung 2.2 zeigt, wie die Kommunikation zwischen der Applikation und der BSW über die RTE implementiert ist. Die Kommunikation verläuft über Schnittstellen, von welchen es drei Arten gibt:

1. **AUTOSAR Schnittstelle:** Diese Art von Schnittstellen wird sowohl von SWCs als auch von der Steuergeräteabstraktionsschicht und den Complex Device Drivers verwendet. Diese Schnittstelle stellt Ports zur Verfügung, die zur Kommunikation verwendet werden.
2. **Standardisierte AUTOSAR Schnittstelle:** Die Definition dieser Schnittstellen ist vom selben Typen, wie die der AUTOSAR Schnittstellen. Das besondere daran ist aber, dass diese in jeder BSW für bestimmte Dienste standardisiert sind und deshalb den Entwicklern der Applikation von vornherein bekannt sind.
3. **Standardisierte Schnittstellen:** Diese Schnittstellen können nicht von SWCs verwendet werden, sondern nur von der RTE. Der Grund dafür ist, dass man den Zugriff der Applikation auf einige Dienste der BSW, wie z.B. das Scheduling von Tasks, einschränken wollte und diese Funktionen somit nur durch die RTE genutzt werden können. [Nau09]

Softwarekomponente

Eine AUTOSAR Applikation besteht aus SWCs, in denen die Funktionalitäten implementiert sind. Eine SWC ist ein Container, der mindestens eine Runnable (siehe 2.1.3) beinhaltet, worin der Code zur Umsetzung der Funktionalitäten enthalten ist. Zur Kommunikation mit anderen SWCs und der BSW besitzt eine SWC Ports (siehe 2.1.3). Die Informationen zu den Runnables und Ports einer SWC sind für die Konfiguration eines RTE essenziell und sind in einer SWC Beschreibung enthalten. [AUT03; KF09]



Note: This figure is incomplete with respect to the possible interactions between the layers.

Abbildung 2.2.: Detailierte Ansicht der SWCs in der AUTOSAR Architektur aus [AUT13a]

SWC Beschreibung

Eine SWC Beschreibung enthält die Informationen zu den SWCs einer Applikation und liegt in einem ARXMLs (AUTOSAR XMLs)-Format vor. Diese beinhaltet unter anderem die verwendeten Ports und PortInterfaces, welche zur Konfiguration der Kommunikation notwendig sind. Zusätzlich findet man auch die Informationen zu den implementierten Runnables jeder SWC und zu den RTEEvents (siehe 2.1.3), welche die Runnables auslösen. Die Abbildungen A.1 und A.2 zeigen jeweils Beispiele für die Beschreibung der Ports und Runnables im ARXML-Format. Der beschriebene Port verfügt unter dem Element *DATA-ELEMENT-REF* über eine Verlinkung zu dem PortInterface.

Runnable

Die AUTOSAR Spezifikation definiert ein Runnable als eine Sequenz von Instruktionen. Der darin enthaltene Code kann sowohl einfache Algorithmen als auch komplexte Programme implementieren. Runnables werden durch Events aufgerufen, welche durch die RTE ausgelöst werden. Die Ausführung eines Runnables geschieht im Kontext eines Tasks (siehe 2.1.3). Dadurch werden dem Runnable notwendige Ressourcen (Kontext, Stapelspeicher) zugeteilt. Wird ein Runnable keinem Task zugeordnet, wird der darin implementierte Code nicht ausgeführt. [AUT13a]

In der Abbildung 2.3 ist eine SWC dargestellt, welche zwei Runnables besitzt. Runnable 1 bekommt über ein Receiver-Port Daten übermittelt, welche es dann verarbeitet. Anschließend wird das Ergebnis der Verarbeitung in der Variable x gespeichert. Dadurch kann Runnable 2 nun auf diese Daten zugreifen, sie verarbeiten und über den Sender-Port weiter versenden. [KF09]

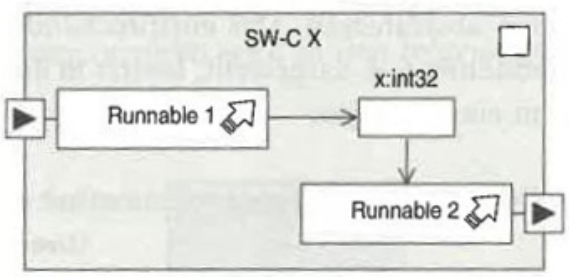


Abbildung 2.3.: Beispiel einer SWC mit zwei Runnables. Übernommen aus [KF09]

Zusätzlich werden Runnables in zwei Kategorien eingeteilt. Bei Runnables der Kategorie 1 geht man davon aus, dass diese in einer endlichen Zeit terminieren werden. Diese werden üblicherweise einfachen Tasks (siehe 2.1.3) zugeordnet. Runnables der Kategorie 2 können *WaitPoints* enthalten, welche die Ausführung einer Runnable blockieren können (z.B. durch Warten auf Daten), bis der *WaitPoint* von einem RTEEvent aufgelöst wurde. Runnables der zweiten Kategorie sollen in erweiterten Tasks ausgeführt werden, da sie in den Zustand *wartend* übergehen können.

Es kann vorkommen, dass mehrere Runnables gemeinsam auf einen Speicher zugreifen müssen. Dabei ist es möglich, dass Tasks während der Ausführung von Tasks mit einer höheren Priorität unterbrochen werden können. Um *Race Conditions* zu vermeiden und Datenkonsistenz zu gewährleisten, spezifiziert AUTOSAR eine Reihe von Schutzmechanismen. Zum Beispiel ist es möglich, das Scheduling der RTE so zu konfigurieren, dass Tasks, welche auf kritische Abschnitte zugreifen können, von anderen Tasks nicht unterbrochen werden dürfen. Unter anderem steht auch die Funktion zur Verfügung, Kopien von Datenelementen für alle Runnables zu erstellen, welche gemeinsamen Zugriff auf die Datenelemente haben. [AUT13b]

RTEEvent

RTEEvents (u.a. auch Trigger genannt) werden von der RTE verwendet, um Runnables aufzurufen. Ein Runnable muss mindestens ein RTEEvent implementieren, damit es ausgeführt werden kann. Die AUTOSAR Spezifikation definiert 15 Arten von Events, siehe [AUT13a]. Zur Vereinfachung ordnet man in der Praxis die verschiedenen Arten in 4 Gruppen ein:

- **Init**: Das InitEvent kommt bei Runnables zum Einsatz, die zu Start der Applikation Initialwerte setzen sollen.

- **Timing**: Das TimingEvent wird für Runnables verwendet, welche zyklisch z.B. alle 5ms, aufgerufen werden sollen.
- **ModeSwitch**: Darunter fallen alle Events (z.B. SwcModeSwitchEvent), die Runnables auslösen, welche für den Wechsel der Steuergerätebetriebsmodi verantwortlich sind.
- **Event**: Diese Gruppe enthält alle Events, welche keiner der oberen drei Gruppen zugeordnet werden können, z.B. das DataReceivedEvent, welches ein Runnable auslöst, wenn dieses über einen Port Daten erhält.

Task

Tasks sind Container, deren Inhalt aus Runnables besteht. Man unterscheidet zwischen einfachen und erweiterten Task. Der Unterschied zwischen diesen beiden ist, dass neben den gemeinsamen Zuständen *suspendiert*, *bereit* und *laufend*, der erweiterte Task zusätzlich den Zustand *wartend* besitzt, welcher für Runnables mit *WaitPoints* wichtig ist.

Einfache Tasks haben am Anfang und am Ende jeweils einen Synchronisationspunkt. Erweiterte Tasks können weitere Synchronisationspunkte besitzen, abhängig davon, ob Runnables darin *WaitPoints* besitzen.

Das Scheduling der Tasks wird von der RTE in einer FIFO (first-in-first-out)-Warteschlange durchgeführt. Dafür erhält jedes der Tasks während der Konfiguration der RTE eine statische Priorität, welche nicht während der Laufzeit geändert werden kann. [AUT13b; KYM+09]

Port

Ein Port wird von einer SWC zur Kommunikation verwendet. Ein Port basiert auf einem Port Interface und wird durch diesen in seiner Funktionsweise festgelegt. Zwei wichtige Typen von Port Interfaces sind Client/Server und Sender/Receiver. Server- und Sender-Ports bezeichnet man unter anderem auch als Provider-Ports, weil diese Funktionen bzw. Daten zur Verfügung stellen. Im Gegenzug nennt man Client- und Receiver-Ports als Require-Ports. [KF09]

Durch die Client/Server-Kommunikation wird es den Clients ermöglicht auf Dienste zuzugreifen, die ein Server zur Verfügung stellt. Zur Nutzung eines der Dienste, bietet ein Server Operationen an, die ein Client aufrufen kann. Eine Operation kann Parameter besitzen, die ein Client beim Aufruf an den Server übergeben muss. Zusätzlich kann eine Operation einen Wert zurückgeben. Somit ist die Kommunikation bidirektional. Abbildung 2.4 zeigt ein Beispiel für eine Client/Server-Kommunikation. Links im Bild findet man die SWC *Blinkleuchte-HintenRechts* und auf der rechten Seite befindet sich die *I/O Hardware Abstraction (IoHwA)* der BSW. Die Funktionen der *IoHwA* sind für SWCs als Server-Ports zur Verfügung gestellt. Der abgebildete Server-Port *Lamp6Control* ist für die Steuerung der Lampe sechs zuständig. Konkret wird dafür die Funktion *setLamp(IN ARGUMENT boolean Lamp)* bereit gestellt. *BlinkLeuchte-HintenRechts* besitzt den Client-Port *LampControl*, welcher mit dem Server-Port verbunden ist. Möchte

nun die SWC die Lampe sechs steuern, dann muss es beim Aufruf der Operation *setLamp* das Argument *Lamp* übergeben. [KF09]

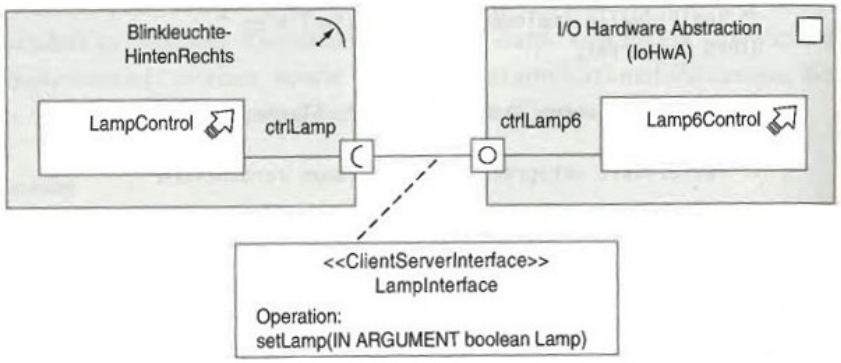


Abbildung 2.4.: Beispiel einer Client/Server-Kommunikation zwischen zwei SWCs. Übernommen aus [KF09]

Die Sender/Receiver-Kommunikation wird verwendet, um Datenelemente von einem Kommunikationspartner zum anderen zu übermitteln. Die Kommunikation findet hierbei nur in eine Richtung statt und ist somit unidirektional. Wenn man eine Verbindung in die andere Richtung herstellen möchte, damit der Sender eine Antwort empfangen kann, dann braucht man eine zweite Sender/Receiver-Verbindung. Abbildung 2.5 zeigt ein Beispiel für eine Sender/Receiver-Kommunikation zwischen den SWCs *WarnblinkTaster* und *BlinkerSteuerung*. In diesem Fall werden Daten vom *WarnblinkTaster* zur *BlinkerSteuerung* übertragen. Dafür besitzt der *WarnblinkTaster* einen Sender-Port mit dem Namen *giveStatus* und ein Runnable namens *SensorRunnable*. Um nun das Datenelement, in diesem Fall *UInt8 TasterVal*, an die *BlinkerSteuerung* zu senden, wird das Runnable SensorRunnable ausgeführt. Das Runnable verwendet nun die Port-Verbindung, um das Datenelement darüber an den entsprechenden Receiver-Port *takeStatus* der *BlinkerSteuerung* zu übertragen. Nach dem die Übertragung abgeschlossen ist, kann das Datenelement vom Runnable *TasterRunnable* der *BlinkerSteuerung* verarbeitet werden. [KF09]

Port-Mapping

Wie zuvor beschrieben, verläuft die Kommunikation zwischen SWCs und zwischen SWCs und der BSW über Ports. Damit diese Kommunikation hergestellt werden kann, müssen kompatible Ports miteinander verbunden werden. Den konkreten Prozess des Verbindens von Ports nennt man auch Port-Mapping, welcher aktuell manuell von einem Integrator durchgeführt wird. Der Prozess besteht, im Grunde genommen, aus der Zuordnung von Server/Client- und Sender/Receiver-Ports, welche der Inhalt der SWC Beschreibung sind. Das resultierende PM ist statisch, d.h. es kann nicht zur Laufzeit geändert werden.

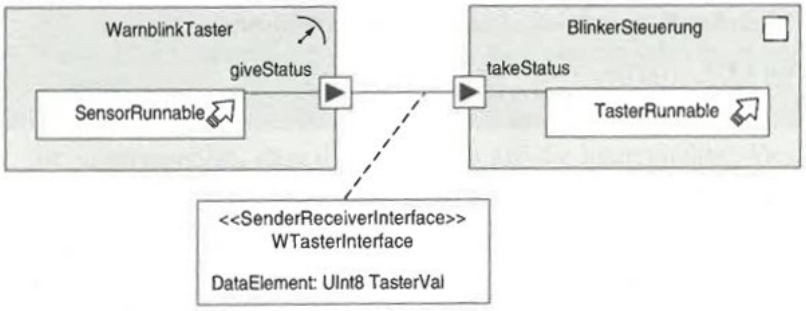


Abbildung 2.5.: Beispiel einer Sender/Receiver-Kommunikation zwischen zwei SWCs. Übernommen aus [KF09]

Runnable-Task-Mapping

So wie das Port-Mapping, ist das Runnable-Task-Mapping eine manuelle Aufgabe des Integrators. Hierbei besteht die Aufgabe aus der Zuordnung von Runnables aus der SWC Beschreibung zu Tasks. Der Integrator ist somit dafür verantwortlich, die Applikation so aufzuteilen, dass die RTE ein ordnungsgemäß funktionierendes Scheduling ausführen kann, ohne auf Probleme zur Laufzeit zu stoßen.

Erst nach der Durchführung der beiden Konfigurationsschritte PM und RTM kann die RTE generiert werden. Beim Generieren wird Code für die AUTOSAR Software erzeugt, welcher die Kommunikation und das Scheduling der RTE implementiert.

2.2. Software

2.2.1. DaVinci Configurator Pro

Der DaVinci Configurator Pro ist ein kommerzielles Softwareprodukt der Vector Informatik GmbH. Neben anderen Funktionen, hilft dieses Werkzeug einem Integrator bei der Konfiguration und Generierung der RTE. [Gmb17a]

2.2.2. DaVinci Developer

Der DaVinci Developer ist ein kommerzielles Softwareprodukt der Vector Informatik GmbH. Der Verwendungszweck dieses Werkzeugs ist der Entwurf von der Architektur einer AUTOSAR Applikation. Darunter fällt auch die Konzeption einzelner SWCs mit den entsprechenden Ports und Runnables. [Gmb17b]

2.2.3. Groovy

Apache Groovy ist eine dynamische Open-source Programmier- und Skriptsprache für die Java-Plattform, welche ursprünglich das Ziel hatte, durch die Vereinfachung der Java-Syntax, die Produktivität der Entwickler zu steigern. Es kann problemlos in jedes Java-Programm integriert werden und erweitert diese um eine Reihe von nützlichen Funktionalitäten. [Pro17]

2.2.4. H2 Database Engine

H2 Database Engine ist eine Open-source Java SQL Datenbankmanagementsystem (DBMS), welche seit 2004 entwickelt wird. Der Fokus bei der Entwicklung lag vor allem auf der Verbesserung der Geschwindigkeit bei der Ausführung von Datenbankabfragen. Neben der Geschwindigkeit, liegen die Vorteile von H2 darin, dass es sehr einfach in andere Java-Anwendungen integriert werden kann und auch problemlos plattformübergreifend arbeiten kann. [Gro17]

2.2.5. Drools

Drools ist ein Open-source Geschäftsregel-Managementsystem (engl. Business-Rule-Management-System (BRMS)), welches eine Weiterentwicklung des Rete-Algorithmus namens PHREAK als Rule Engine verwendet. Drools kann einfach als Abhängigkeit in eine bestehende Java-Applikation eingebunden werden. Für die Formulierung der Regeln wird eine eigene Rule-Sprache zur Verfügung gestellt. [Com17]

3. Verwandte Arbeiten

3.1. Port-Mapping

Das Zuordnen von Datenelementen ist ein bekanntes Problem, welches man auch unter dem Begriff *Matching*. Relevant für das Port-Mapping in dieser Arbeit sind vor allem Veröffentlichungen, die sich mit Port-Matching und Matching auf Basis von Namensähnlichkeiten beschäftigten.

McKeown et al. (1999) [MMAW99] stellen zwei Algorithmen zur Erhöhung des Durchsatzes in einem Input-Queued Switch vor. Ein Input-Queued Switch besitzt Eingangs- und Ausgangsports, welche einander zugeordnet werden müssen, damit die Pakete versendet werden können. Interessant hierbei ist, dass das Problem als ein bipartiter Graph formuliert wurde. Auf der linken Seite befinden sich Knoten, welche die Eingangsports repräsentieren. Diese haben gewichtete Kanten, die mit den gegenüberliegenden Knoten der Ausgangsports verbunden sind. Um die optimale Lösung zu finden, entwickelten die Autoren zwei Algorithmen, welche den Knotenpaaren Kanten mit den höchsten Gewichten zuordnet. Das AUTOSAR Port-Mapping-Problem hat starke Ähnlichkeiten zu dem Problem, mit dem sich die Autoren des Papers auseinandersetzen. AUTOSAR definiert Provider- und Require-Ports, welche zur Kommunikation miteinander verbunden werden müssen. Diese können analog auch als bipartiter Graph modelliert werden. Der entscheidende Unterschied liegt jedoch bei der Gewichtung der Kanten. McKeown et al. verwenden dazu domänenspezifische Informationen des Switchs, die keine Relevanz zum AUTOSAR Port-Mapping besitzen.

Melnik et al. (2002) [MGR02] entwickelten einen Algorithmus, um zu einander passende Datenelemente in einem Graphen zu finden. Ähnlich wie bei [MMAW99] wurden die einzelnen Datenelemente als Knoten, die untereinander durch Kanten verbunden sind, dargestellt. Der wichtige Unterschied hierbei ist, dass für die Gewichtung der Kanten Namensähnlichkeiten verwendet werden. Dieser Aspekt spielt auch für das Mapping von AUTOSAR Ports ein Rolle, da auch in dieser Domäne Portnamen verwendet werden können, um den entsprechenden Gegenport zu finden. Zusätzlich wurden in dieser Arbeit auch Filter definiert, die dazu genutzt werden können, die Komplexität des Problems zu reduzieren. Dabei werden weitere Eigenschaften der Datenelemente wie z.B. Kardinalität in Betracht gezogen, um inkompatible Elemente schon vor dem Matching ausschließen zu können.

Avigdor Gal (2008) [Gal08] hat sich als Ziel gesetzt, eine vollautomatische Lösung für das Schema Matching zu entwickeln, da vorherige automatischen Methoden eine manuelle Nachbear-

beitung erforderten, welche aber ab einer bestimmten Größe des Problems auf Schwierigkeiten bei der Skalierung stießen. Analog zu den vorher erwähnten Veröffentlichungen, wurde das Schema Matching auch als bipartiter Graph formuliert. Auch hier lief die Berechnung der Kantengewichte über die Namensähnlichkeiten der Schema-Elemente.

Cohen et al. (2003) [CRF03] führten eine Evaluation der Leistung von 13 verschiedenen String-Metriken zur Berechnung der Namensähnlichkeiten durch. Die darin evaluierten Metriken können drei Gruppen zugeordnet werden. Die erste Gruppe enthält Methoden, welche zur Berechnung der Ähnlichkeit die Anzahl von Operationen (Einfügen, Löschen, Ersetzen, Vertauschen) zählen, um einen Namen in den Zielnamen zu überführen. Die zweite Gruppe enthält tokenbasierte Verfahren, welche zwei String jeweils als Zusammensetzung aus mehreren Wörtern betrachten. Die Ähnlichkeit berechnet sich aus der Anzahl von gleichen Wörtern, welche beide String enthalten. Die dritte Gruppe beinhaltet Hybrid-Verfahren, welche eine Mischung aus den Ansätzen der beiden zuvor genannten Gruppen darstellen.

Achananuparp et al. (2008) [AHS08] bietet eine Übersicht über eine Reihe von Algorithmen, welche sich für die Berechnung von Satzähnlichkeiten eignen. Die darin genannten Verfahren sind in die drei Kategorien *Maße zur Wortüberschneidung*, *TF-IDF Maße* und *Linguistische Maße* unterteilt. Der grundlegende Unterschied zwischen diesen Gruppen ist, ist die Basis die man verwendet, um letztendlich die Ähnlichkeit zwischen zwei Sätzen zu berechnen. Algorithmen aus der Gruppe der *Maße zur Wortüberschneidung* sind den tokenbasierten Verfahren aus [CRF03] ähnlich, denn diese bestimmen auch die Ähnlichkeit anhand der Anzahl von gleichen Wörtern, die in den beiden Sätzen enthalten sind. Die Gruppe *TF-IDF Maße* beinhaltet Algorithmen, die für sich längere Texte eignen. Hier werden die Frequenzen der Wörter in den Texten berechnet und anhand dieser die Ähnlichkeit bestimmt. Die dritte Gruppe *Linguistische Maße* verwendet semantische Beziehungen zwischen den Wörtern und den syntaktischen Aufbau der Sätze, um den Vergleich zwischen den Sätzen zu ziehen.

3.2. Runnable-Task-Mapping

Ferrari et al. (2009) [FNG+09] formulierten das Runnable-Task-Mapping als ein Optimierungsproblem. Das Ziel der Optimierung ist die Minimierung des Speicherverbrauchs unter Beachtung der Zeiteinschränkungen. Zusätzlich stellten die Autoren Schutzmechanismen zur Sicherstellung von Datenkonsistenz vor, die bei dem Runnable-Task-Mapping zum Einsatz kommen können. Eine Implementierung und Evaluation des Optimierungsproblem wurde von den Autoren nicht durchgeführt.

Zhang und Gu (2011) [ZG11] nahmen das Konzept aus [FNG+09] und implementierten und lösten das Optimierungsproblem mit einem Evolutionären Algorithmus. Für das initiale Mapping von Runnables verwendeten die Autoren Industriepraktiken. Darauf wurde dann die Optimierung angewendet, die unter anderem auf dem Zugriff auf gemeinsame Variablen der Runnables basiert.

Eine weitere Implementierung und Lösung für das in [FNG+09] formulierte Optimierungsproblem lieferten Zeng und Natale (2012) [ZN12]. Anders als bei [ZG11] wurden hierbei zur Optimierung sowohl die Ausführungsreihenfolge der Runnables als auch die Wahl der Mechanismen für die Datenkonsistenz in Betracht gezogen. Zur Lösung des Problems wurde es als ein Integer Linear Problem formuliert. Die Evaluierung mit Daten aus einem echten Industrieprojekt ergab eine Reduktion des Speicherverbrauchs um 69%. Das Problem bei dieser Implementierung ist die Skalierung. Für 90 Runnables brauchte der Algorithmus über vier Stunden, um das optimale Ergebnis zu finden, was im Falle der Integration in die Vector-CI-Pipeline an der zeitlichen Begrenzung von 30 Minuten scheitern würde.

Ein anderer Ansatz für das Runnable-Task-Mapping wurde in Long et al. (2009) [LLP+09] vorgestellt. Dieser basiert auf der Optimierung von der intra-Electronic Control Unit (ECU)-Kommunikation. Das Ziel hierbei ist, Datenabhängigkeiten und Aufrufe zwischen Runnables zu erkennen. Auf Basis dieser Kommunikation soll ein optimiertes Mapping entstehen, welches Kontextwechsel reduziert und Dateninkonsistenz vermeidet. Die Autoren haben auf dieser Grundlage Regeln festgelegt, die auf ein Runnable-Task-Mapping zum Optimieren angewendet werden können. Ein wichtiger Aspekt hierbei ist, dass die zur Optimierung verwendeten Daten in Form von einer SWC Beschreibung (siehe 2.1.3) auch für die Vector-CI-Pipeline zur Verfügung stehen.

4. Konzeptentwicklung

4.1. Verfügbare Daten

Bevor das Konzept für die kontinuierliche Integration von AUTOSAR Software entwickelt werden kann, ist es wichtig zu wissen, welche Informationen zur Verfügung stehen und in welchem Format diese vorliegen. Bei einem Commit über das gemeinsame SVN-Repository liefert der OEM eine Beschreibung der SWCs als eine ARXML Datei. Diese beinhaltet Informationen zu den verwendeten Ports und Runnables der einzelnen SWCs. Zu den Ports erfährt man daraus die Namen, den Typen (Server/Receiver oder Client/Server) und die Variablenzugriffe. Die Runnables sind mit ihren Namen, Events und Portzugriffen beschrieben. Hier kann man entnehmen, wie sie von der BSW aufgerufen werden und wie sie eventuell mit anderen Runnables kommunizieren. Was man nicht daraus entnehmen kann, ist das konkrete Verhalten der Runnables zur Laufzeit, d.h. es gibt keine Informationen darüber, welche Funktion ein Runnable erfüllt, da der darin implementierte Code nicht bekannt ist. Folglich stehen auch keine Informationen zur Laufzeit, Speicherverbrauch, Deadlines usw. zur Verfügung. Weitere Information und Beispiele zu der SWC Beschreibung sind im Abschnitt 2.1.3 zu finden.

4.2. Port-Mapping

In Kapitel 3 wurden bereits verwandte Arbeiten vorgestellt, die sich mit PM und Matching von Datenelementen in Graphen beschäftigen. Durch die Ähnlichkeit des vorliegenden Problems zum Mapping von AUTOSAR Ports können einige Aspekte und Ideen, der darin entwickelter Konzepte, übernommen werden. In diesem Abschnitt wird das Konzept für das automatische Mapping von AUTOSAR Ports vorgestellt.

4.2.1. Modellierung des Problems

Das Ziel vom PM ist die korrekte Zuordnung von Client- zu Server-Ports und Sender- zu Receiver-Ports (siehe Abschnitt 2.1.3). Das optimale Zuordnen von Elementen aus einer Gruppe zu den Elementen einer anderen Gruppe ist in der Graphentheorie als Zuordnungsproblem bekannt. Der erste Schritt der Modellierung ist die Überführung des Problems in einen Graphen. Analog zu [MMAW99], wo Eingangsports und Ausgangsports einander zugeordnet

wurden, können Provider- und Receiver-Ports als Knoten in einem bipartiten Graph dargestellt werden. Die Knoten auf der einen Seite sind über Kanten mit den Knoten auf der anderen Seite verbunden. Zur Veranschaulichung ist ein bipartiter Graph in der Abbildung 4.1 für Server- und Client-Ports dargestellt.

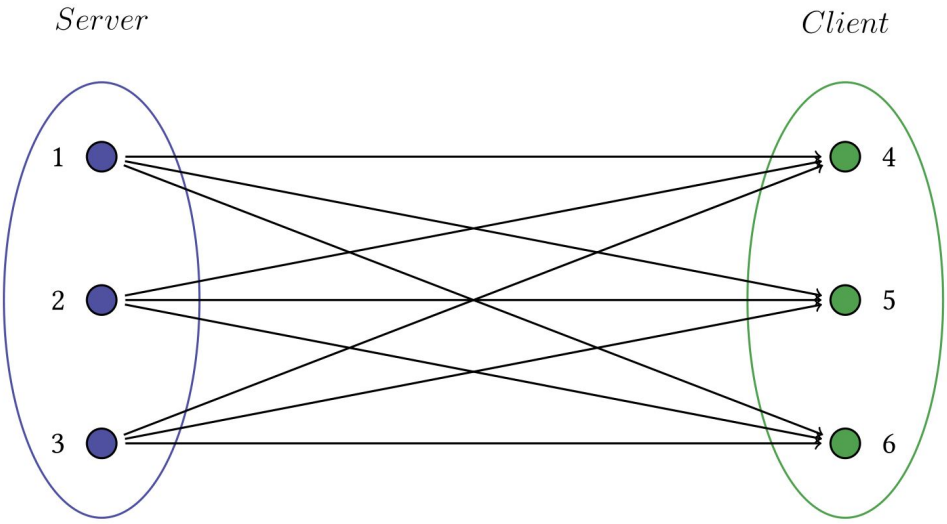


Abbildung 4.1.: Darstellung eines bipartiten Graphen für CS-Ports.

4.2.2. Gewichtung der Kanten

Damit eine optimale Zuordnung der Ports gefunden werden kann, bedarf es vorher einer Gewichtung der Kanten in dem bipartiten Graphen. Aus den zur Verfügung stehenden Informationen kann man entnehmen, dass die Namen der Ports die einzige Eigenschaft ist, die dafür in Frage kommt. Die Verwendung der Portnamen ist deswegen sinnvoll, da die Vergabe dieser in der Praxis oftmals nach der konkreten auszuführenden Funktion bzw. nach dem Namen des zu verarbeiteten Signals erfolgt. Aus diesem Grund weisen die meisten Ports, die verbunden werden sollen, Namensähnlichkeiten auf. Zu beobachten ist außerdem, dass Portbezeichnungen oftmals auch die Namen der SWCs der Gegenports enthalten. Folglich ist eine Einbeziehung der Namen der SWCs in die Gewichtung als sinnvoll zu erachten.

In der Abbildung 4.2 ist diese Tatsache nochmal veranschaulicht. Auf der linken Seite ist eine SWC mit dem Namen *CpApMySwc* abgebildete. Diese Komponente hat zwei Receiver-Ports *PpDoorStateFrontLeft* und *PpDoorStateFrontRight*, jeweils für die linke und für die rechte Tür. Diese zwei Ports sind mit den Sender-Ports der SWCs *CpSaDoorFrontLeft* und *CpSaDoorFrontRight* verbunden. Würde man bei der Zuordnung der Ports nur von den Namen der Ports ausgehen, wäre es schwierig, da die Namen der Sender-Ports identisch sind und dadurch nicht klar ist, welcher der beiden Türen diese angehören. Nimmt man aber die Namen der SWCs hinzu, wird

sofort klar, wie die Zuordnung aussehen muss, da diese Informationen darüber enthalten, auf welcher Fahrzeugseite sich die Tür befindet.

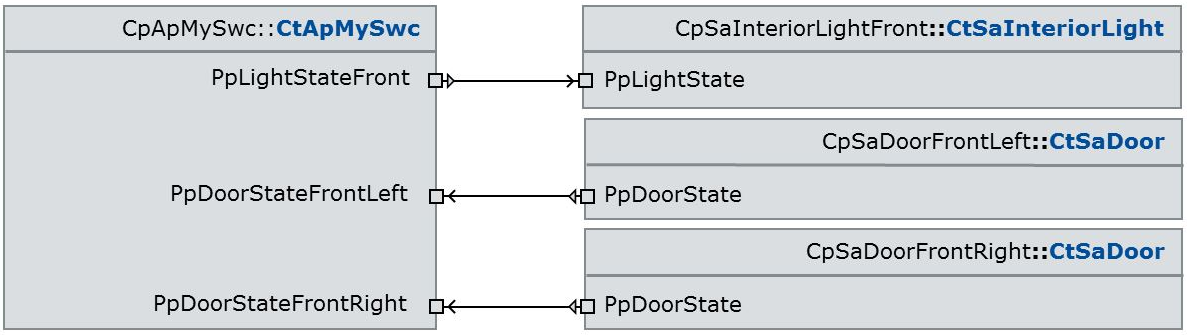


Abbildung 4.2.: Darstellung von SWCs mit den dazugehörigen Portverbindungen. Abbildung übernommen aus [Gmb16b].

Die Aufgabe liegt nun darin, ein passendes Verfahren zu finden, um die Namensähnlichkeiten zwischen den Ports zu bestimmen. Zu diesem Zweck existiert bereits eine lange Reihe von Algorithmen, die man in der Literatur unter String-Metriken kennt. In [CRF03] wurde ein Vergleich zwischen der Leistung einiger Methoden durchgeführt (siehe Abschnitt 3.1). Die Methoden, welche die Ähnlichkeiten auf Basis des Vergleichs einzelner Zeichen in einem String berechnen, sind für diesen Anwendungsfall ungünstig. Beispielsweise zählt die Levenshtein-Distanz die Anzahl der Einfüge-, Lösch- und Ersetz-Operationen, die minimal notwendig sind, um einen String in den Vergleichsstring zu überführen [Lev66]. Unter genauer Betrachtung der Portnamen von zwei passenden Ports fällt auf, dass diese zwar oftmals aus gleichen Worten bestehen, die Anordnung von diesen aber unterschiedlich ist. Das führt dazu, dass String-Metriken aus dieser Gruppe, trotz starker Ähnlichkeit, keine gute Ergebnisse erzielen, da nur einzelne Zeichen verglichen werden, wie in der Tabelle 4.1 zu sehen ist. In den beiden Beispielen wird der selbe Quell-String mit unterschiedlichen Ziel-Strings auf Basis der Levenshtein-Distanz verglichen. Und obwohl der Ziel-String aus dem zweiten Beispiel semantische keine Ähnlichkeit zum Quell-String besitzt, kommt nach der Berechnung die selbe Distanz raus.

Tabelle 4.1.: Beispiele für die Anwendung der Levenshtein-Distanz

	Beispiel 1	**Beispiel 2**
Quelle	CpApMySwcPpDoorStateFrontLeft	CpApMySwcPpDoorStateFrontLeft
Ziel	CpSaDoorFrontLeftPpDoorState	IrgendeinBeispielStringTest
Levenshtein-Distanz	24	24

Ein alternativer Ansatz zum Bestimmen der Ähnlichkeit der Namen wäre die Verwendung von Algorithmen zur Berechnung der Satzähnlichkeit, wie sie in [AHS08] beschrieben sind. Dadurch, dass die Portnamen aus mehreren Worten zusammengesetzt sind, ist es möglich diese in einzelne Worte zu zerlegen und als Sätze zu betrachten.

Bezieht man die Kenntnisse über die Algorithmen aus [AHS08] auf die Problemstellung des Port-Mappings, stellt man fest, dass nur eine der drei darin aufgeführten Gruppen zur Anwendung in Frage kommt. Das es sich bei Portnamen nicht um längere Texte handelt, ist die Anwendung der Algorithmen aus der Gruppe *TF-IDF Maße* wenig sinnvoll. Auch eine semantische und syntaktische Analyse erscheint ungünstig, da die einzelnen Wörter oftmals kryptisch sind und keine Semantik vorhanden ist. Folglich bleiben nur die vier Algorithmen aus der Gruppe der *Maße zur Wortüberschneidung*. Zwei der darin aufgeführten Algorithmen *IDF Overlap* und *Phrasal Overlap* eignen sich nur für längere Texte. *Jaccard Similarity Coefficient* und *Simple Word Overlap* zählen die Wörter, die in zwei Sätzen vorkommen und setzen sie in Verhältnis zu der Wortmenge bzw. Satzlänge. Sinngemäß ergibt sich, dass die Verwendung eines dieser zwei Verfahren für den Anwendungsfall praktisch ist, denn sie operieren unabhängig von der Linguistik und Textlänge. Die Abbildung 4.3 zeigt ein Beispiel für einen gewichteten bipartiten Graphen.

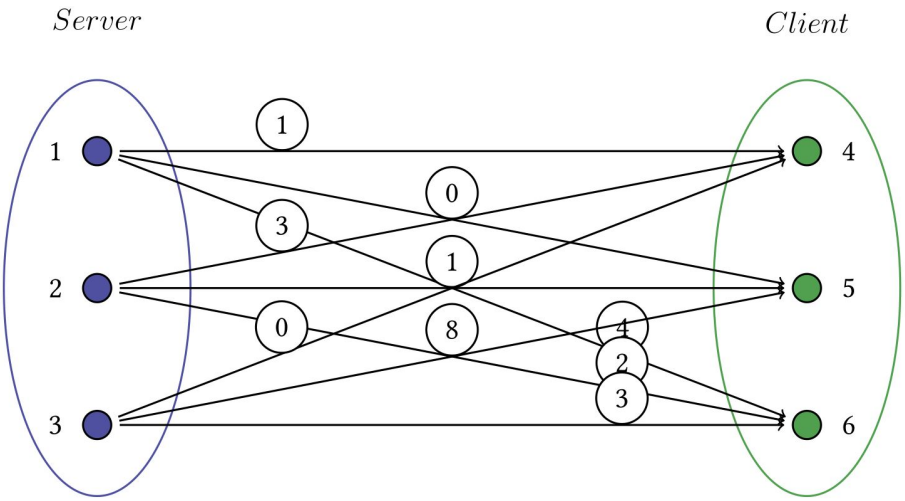


Abbildung 4.3.: Hier ist ein Beispiel für einen bipartiten Graphen dargestellt, dessen Kanten gewichtet sind. Das Gewicht einer Kanten symbolisiert die Namensähnlichkeit zwischen zwei darüber verbundenen Ports.

4.2.3. Filter

Nach dem Vorbild von [MGR02] werden für das Port-Mapping Filter definiert. Die Verwendung von Filtern hat die Absicht, die Komplexität zu reduzieren und das Endresultat qualitativ zu verbessern, in dem man Port-Paare ausschließt, die nicht miteinander kompatibel sind. Die erste Einschränkung ist ziemlich offensichtlich und basiert darauf, dass nur bestimmte Typen von Ports kompatibel sind. Es können nur Server- mit Client-Ports und Sender- mit Receiver-Ports verbunden werden. Aus diesem Grund wird für die beiden Paare jeweils ein eigener bipartiter Graph verwendet. Ein weiterer Aspekt der Kompatibilität sind die Parameter

bei einer Sender-Receiver-Verbindung und die Funktionsargumente bei einer Client-Server-Verbindung. Möchte ein Sender-Port Daten an einen Receiver-Port senden, müssen diese Daten einen Datentypen haben, den beide Ports unterstützen. Ähnlich ist es auch bei der Client-Server-Verbindung. Ein Server stellt Funktionen zur Verfügung, welche Argumente entgegennehmen können. Damit ein Client eine dieser Funktionen aufrufen kann, muss er an die Funktion die passenden Argumente übergeben können. Aus diesem Grund werden aus dem bipartiten Graphen die Kanten entfernt, die zwei inkompatible Ports verbinden. Die Abbildung 4.4 zeigt einen gefilterten bipartiten Graphen.

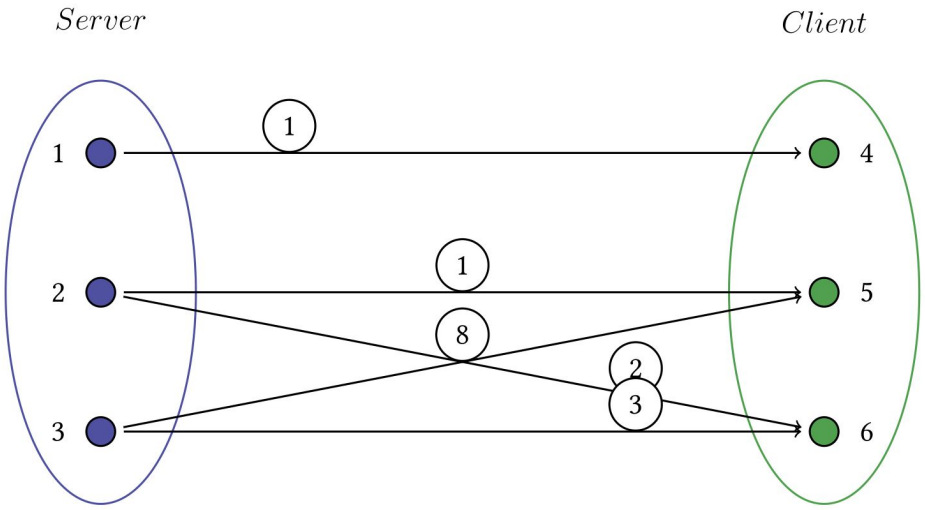


Abbildung 4.4.: Die Darstellung zeigt einen gefilterten bipartiten Graph. Zwischen inkompatiblen Server- und Client-Ports gibt es keine Kanten.

4.2.4. Lösung des Zuordnungsproblems

Nach dem Gewichten der Kanten und dem Filtern von inkompatiblen Verbindungen liegt die Aufgabe nun darin, aus dem vorliegenden bipartiten Graphen das Port-Mapping zu erzeugen. Betrachtet man die Kardinalität [0,1 - 0,1] der Ports, sieht man, dass jeder Port entweder mit genau einem oder keinem anderen Port verbunden werden kann. Ports können frei bleiben, wenn diese von der Applikation nicht genutzt werden. Der letzte Schritt liegt also darin, unter Beachtung der Kardinalität, die Kanten mit den höchsten Gewichten auszuwählen, sodass die Summe der Kantengewichte maximal ist. Dieses Problem kennt man auch unter dem Namen *Maximum-Matching*. Um ein Maximum-Matching aus einem bipartiten Graphen zu gewinnen, steht eine lange Reihe von Algorithmen zur Verfügung, von denen einige in [BDM12] beschrieben sind. Prinzipiell ist die Wahl des konkreten Verfahrens nicht kritisch, solange es garantiert ist, dass dieses zu einem korrekten Ergebnis führt. [MMAW99] verwendeten bei der Lösung ihres Problem die Ungarische Methode, welche eine Laufzeit von $\mathcal{O}(n^3)$ hat, um

das Maximum-Matching zu finden. Dieses Verfahren ist auch für das Lösen des vorliegenden Problems des Port-Mappings vollkommen ausreichen und wird deshalb auch verwendet.

4.3. Runnable-Task-Mapping

Das Ziel in diesem Abschnitt ist, ein Konzept zu erarbeiten, welches ermöglicht automatisch ein gültiges, funktional korrektes RTM (siehe Abschnitt 2.1.3) zu konfigurieren. Betrachtet man die Veröffentlichungen aus dem Abschnitt 3.2, stellt man fest, dass der Fokus in diesem Forschungsbereich bisher hauptsächlich auf der Optimierung lag. Optimierung des Mappings kann zwar in einigen Anwendungsfällen und Projekten kritisch sein, spielt im Kontext dieser Arbeit aber eine untergeordnete Rolle. Der Schwerpunkt liegt viel mehr darin, eine automatische Lösung für das Problem des RTM zu erhalten, welche in der Vector-CI-Pipeline verwendet werden kann und dabei die zeitlichen Anforderungen erfüllt. Das soll nicht heißen, dass die Optimierung keine Beachtung erhalten soll, sondern dass diese nur so weit gehen kann, dass die zeitlichen Einschränkungen von 30 Minuten für den Gesamtdurchlauf der Vector-CI-Pipeline nicht verletzt werden. Um Ideen für ein mögliches Konzept zu sammeln, müssen deshalb, neben wissenschaftlichen Veröffentlichungen, auch alternative Quellen hinzugezogen werden. Zum einen wurden Experten in der Konfiguration von AUTOSAR Software befragt, nach welchen Regeln und Praktiken ein RTM durchgeführt wird, zum anderen bietet die AUTOSAR Spezifikation zu diesem Thema Empfehlungen an.

Im Folgenden werden die dabei gewonnenen Informationen beschrieben und analysiert. Basierend darauf, wird dann ein Konzept erstellt.

4.3.1. Industriepraktiken

Industriepraktiken beschreiben das Vorgehen eines Integrator bei dem Erstellen eines RTM unter der Verfügbarkeit der selben Daten, wie sie in dieser Arbeit vorliegen. Die große Besonderheit dabei ist, dass die Runnables nach ihrem Trigger (siehe Abschnitt 2.1.3) gruppiert werden und die jeweiligen Gruppen einen eigenen Task bekommen:

Z1 Alle Runnables mit einem Init-Trigger sollen einem eigenen Task zugeordnet werden.

Z2 Alle Runnables mit einem Periodic-Trigger und der gleichen Periode sollen einem eigenen Task zugeordnet werden. Zu beachten ist hierbei, dass Runnables, die unterschiedliche Perioden haben (z.B. 5 ms und 10 ms), jeweils unterschiedlichen Tasks zugeordnet werden.

Z3 Alle Runnables mit einem Event-Trigger sollen einem eigenen Task zugeordnet werden.

Z4 Alle Runnables mit einem Mode-Switch-Trigger sollen einem eigenen Task zugeordnet werden.

Hierbei ist außerdem noch wichtig zu erwähnen, dass einem Task zwar unbegrenzt viele Runnables zugeordnet werden können, dieser aber nur begrenzt viele verschiedene Events aufnehmen kann. Das bedeutet, dass ein Task unbegrenzt viele Runnables haben kann, die z.B. durch einen Init-Trigger gestartet werden, aber nur begrenzt bei der Aufnahme von Event-Runnables ist, da die Events üblicherweise unterschiedlich sind. Die Begrenzung ist davon abhängig, ob es sich bei der verwendeter Hardware um eine 32- oder 64-Bit-Architektur handelt. Folglich können die Task jeweils 32 oder 64 Events aufnehmen. Diese betrifft vor allem die Gruppen 3 und 4.

Ein großer Vorteil dieser Aufteilung in Gruppen ist die Vermeidung von Kontextwechseln. Die Funktionsaufrufe passieren durch die BSW, in dem ein Trigger ausgelöst wird. Wenn z.B. n Runnables mit einem Init-Trigger auf n Task verteilt werden, müssen zum Aufruf jeder dieser Runnables jedes Mal der Task gewechselt werden. Wenn diese Runnables aber auf einem Task liegen, dann können sie auch alle in dessen Kontext ausgeführt werden und der Overhead durch die vielen Kontextwechsel wird vermieden. Ein weiterer Vorteil ist der deterministische Ablauf der Applikation innerhalb der Init- und Periodic-Tasks. Durch die Gruppierung ist das Verhalten zur Laufzeit in diesen Tasks eindeutig. Würden diese Tasks zusätzlich Runnables mit Event- und ModeSwitch-Triggern enthalten, wäre der Ablauf nicht mehr deterministisch, da das Verhalten zur Laufzeit von Eintritten bestimmter Ereignisse abhängig wäre.

Zusätzlich zu der Gruppierung gibt es noch weitere Praktiken, die zum Einsatz kommen:

E1 Ist der Server-Port eines Server-Runnable nicht mit einem Client-Port verbunden, dann wird das Server-Runnable keinem Task zugeordnet. Dadurch, dass die Ports nicht verbunden werden, wird das Server-Runnable nicht aufgerufen werden können.

E2 Gibt es bei einer Client-Server-Verbindung nur ein Client-Runnable, dann können die Server-Runnables im Kontext des Client-Runnable ausgeführt werden. In diesem Fall müssen die Server-Runnables nicht auf einen Task gelegt werden. Eine Zuordnung dieser Runnables auf einen anderen Task, als den des Client-Runnable, hätte sogar einen Kontextwechsel als Nachteil.

E3 Besitz ein Server-Runnable das Attribut *canBeInvokedConcurrently*, muss diese keinem Task zugeordnet werden. Dieses Attribut wird verwendet, wenn ein Server-Runnable von mehreren Client-Runnables aufgerufen wird. Dadurch können mehrere Instanzen des Server-Runnable im Kontext des jeweiligen Client-Runnable ausgeführt werden.

4.3.2. AUTOSAR Spezifikation

Die AUTOSAR Spezifikation der RTE geht unter anderem auch auf die Thematik des RTM ein. Dabei werden die verschiedenen Eigenschaften der Runnables betrachtet und daraus eine Empfehlung für die Zuordnung zu einem Task gegeben. Hauptsächlich geht es darum, ob ein Runnable der Kategorie 1 oder 2 in einem einfachen oder einem erweiterten Task (siehe Abschnitt 2.1.3) ausgeführt werden sollte. Dazu wurden in der Spezifikation Szenarien

beschrieben, aus denen sich Empfehlungen für ein RTM ableitet lassen. Diese wurden bereits in [LLP+09] als Regeln zusammengefasst und lauten folgendermaßen:

1. Runnables der Kategorie 1 können entweder auf einen einfachen oder einen erweiterten Task gelegt werden.
2. Runnables, die zur Kategorie 2 gehören und einen *WaitPoint* besitzen, werden auf einen erweiterten Task gelegt.
3. Runnables mit einem *SynchronousServerCallPoint* können unter bestimmten Bedingungen beiden Tasktypen zugeordnet werden. Wenn keine Timeout-Überwachung notwendig ist oder das Server-Runnable direkt aufgerufen werden kann und zur Kategorie 1 gehört, dann wird diese auf einen einfachen Task gelegt. Liegen die Eigenschaften nicht vor, dann sollte das Server-Runnable einem erweiterten Task zugeordnet werden.

4.3.3. Optimierung

Wie anfangs bereits erwähnt, existieren zu dem Thema Optimierung von RTM bereits einige Veröffentlichungen, die im Abschnitt 3.2 beschrieben worden sind. Wichtig ist nun zu prüfen, inwiefern diese in die Lösung des Problems dieser Arbeit integrierbar sind. Im Vordergrund steht vor allem die Zeiteinschränkung der Vector-CI-Pipeline, die bei 30 Minuten liegt. Diese Grenze darf nicht überschritten werden. Zum anderen muss auch untersucht werden, ob die Umsetzung, basierend an den zur Verfügung stehenden Daten, realisiert werden kann.

Vergleich der Ansätze

In der Tabelle 4.2 sind die wichtigsten Informationen der drei vorgestellten Ansätze aufgeführt.

Eines der Optimierungsziele aus [ZG11] fokussiert sich darauf, den durch Runnables gemeinsam verwendeten Datenelementen die passenden Mechanismen zur Datenkonsistenz zuzuweisen. Dadurch soll der Bedarf an die Speichergröße reduziert werden, aber dennoch die Deadlines nicht verletzt werden. Die dabei notwendigen Daten stehen bei dieser Arbeit nicht zur Verfügung, denn sowohl die Mechanismen zur Datenkonsistenz als auch die Ausführungsinformationen wie z.B. WCETs, Deadlines und kritische Regionen liegen in der Applikationsschicht. Das zweite Optimierungsziel ist auf die Kommunikation zwischen mehreren ECUs ausgerichtet und ist deshalb nicht für die Arbeit relevant.

Das Optimierungsziel in [ZN12] ist die Suche nach einem Mapping, welches den geringsten Speicherverbrauch hat, aber trotzdem die Zeiteinschränkungen in Form von Deadlines der Runnables einhält. Verwendet werden dabei Informationen zur Ausführung der Runnables. Da diese Informationen in den vorliegenden Daten nicht vorhanden sind, wird die Implementierung nicht möglich sein. Zudem gaben die Autoren an, dass die Laufzeit dieses Verfahrens für

90 Runnables bei über vier Stunden lag. Somit ist auch die zeitliche Einschränkung durch die Vector-CI-Pipeline nicht erfüllt, da die Anzahl der Runnables erfahrungsgemäß höher liegt.

Anders als die beiden Publikationen zuvor, geht es in [LLP+09] nicht um die Minimierung von Speicherbedarf, sondern um die Reduzierung von Latenzen und die Sicherstellung von Datenkonsistenz durch die optimale Verteilung von Runnables auf Tasks. Wie gut ein Mapping ist, wird anhand einer Kostenfunktion ermittelt, die auf Regeln basiert. Die Regeln leiten sich aus den Attributen der Runnables und deren Kommunikation untereinander ab. Die zur Anwendung der Regeln notwendige Informationen finden sich in der Beschreibung der SWCs. Folglich wäre dieser Ansatz zur Optimierung auch für den Anwendungsfall dieser Arbeit umsetzbar. Das einzige Problem hierbei ist allerdings die Dauer zur Findung der optimalen Lösung. Die Autoren generieren hierfür jede mögliche Kombination des Mappings und wenden darauf die Kostenfunktion, um die optimale Lösung zu ermitteln. Die erwartete Laufzeit hierfür liegt bei $\mathcal{O}(n! \cdot c)$, wobei $n!$ die Anzahl der möglichen Kombinationen der Runnables und c die Kostenfunktion darstellt. Bei mehreren Hundert Runnables würde die Suche nach der optimalen Lösung als schwierig erweisen, wenn man die Gesamtlaufzeit von 30 Minuten nicht überschreiten möchte.

Tabelle 4.2.: Inhalt dieser Tabelle sind Publikationen, die sich mit der Optimierung des Runnable-Task-Mapping beschäftigt haben. Aufgeführt sind die Daten und Verfahren, welche zur Optimierung verwenden worden sind.

Veröffentlichung	Daten	Implementierung
Zhang und Gu [ZG11]	Kommunikation über Ports Mechanismen zur Datenkonsistenz Ausführungsinformationen der Runnables	Evolutionärer Algorithmus
Zeng und Di Natale [ZN12]	Ausführungsinformationen der Runnables	Integer Linear Programming
Long et al. [LLP+09]	Kommunikation über Ports Attribute der Runnables	Zufällige Generierung Kostenfunktion

Mögliche Umsetzung der Optimierung

Aus dem vorangegangenem Abschnitt kann man entnehmen, dass keines der vorgestellten Ansätze die ideale Lösung für das in dieser Arbeit gegebene Problem darstellt. Die nur limitiert vorhandenen Daten zu den SWCs und die zeitliche Beschränkung der Vector-CI-Pipeline behindern eine mögliche Umsetzung dieser Verfahren. Dennoch kann man das Problem auch anders auffassen: Optimierung einer vorhandenen, funktional korrekten Lösung des RTM. Folglich steht nun vordergründig die Suche nach Verbesserungsmöglichkeiten und nicht mehr die Suche nach der optimalen Lösung. Dadurch kann der in [LLP+09] vorgestellte Ansatz auf die darin beschriebenen Regeln reduziert und umgesetzt werden, da die langwierige Suche nach der optimalen Kombination entfällt.

Long

Dieser Abschnitt dient dem Zweck den in [LLP+09] präsentierten Optimierungsansatz im Detail vorzustellen. Die Idee für diesen Ansatz entstand aus der Betrachtung der intra-ECU-Kommunikation zwischen Runnables und der Probleme, die dabei entstehen können:

Vergleich zwischen einfachen und erweiterten Tasks

Das erste Problem bezieht sich auf die Wahl zwischen einfachen und erweiterten Tasks. Ein einfacher Task kann nicht in der Ausführung blockiert werden, wenn es für eine Runnable notwendig erscheint. Dies kann z.B. vorkommen, wenn ein Runnable auf ein Event warten muss, bis es die Ausführung fortsetzen kann. Für diese Zwecke sind erweiterte Tasks bestimmt, denn diese enthalten, im Gegensatz zu einfachen Tasks, mehr Synchronisationspunkte, an denen diese Wartepunkte aufgelöst werden können. Diese Synchronisationspunkte erzeugen zusätzlichen Rechenaufwand während der Ausführung. Aus diesem Grund ist es ratsam, erweiterte Tasks nur zu verwendet, wenn es wirklich notwendig ist.

Häufige Kontextwechsel

Ein Kontextwechsel kommt vor, wenn ein Task unterbrochen oder beendet wird und ein anderer Task gestartet wird. Dieses Vorgehen verursacht normalerweise zusätzlichen Aufwand und Latenzen, da der eine Task gesichert und der andere wiederhergestellt werden muss. Während der Kommunikation zwischen Runnables, kann es vorkommen, dass man einen Kontextwechsel durchführen muss. Liegen Sender und Empfänger der Nachricht auf unterschiedlichen Tasks, muss nach dem Versand der Task des Empfängers gestartet werden, damit die Nachricht empfangen werden kann. Dies kann man verhindern, in dem man sowohl den Sender als auch den Empfänger dem selben Task zuordnet.

Datenkonsistenz

Sender und Empfänger kommunizieren oftmals über Variablen. Um Nachrichten zu versenden, wird das Sender-Runnable einen Wert in eine Variable schreiben. Dementsprechend wird ein Receiver-Runnable den Wert aus der Variable auslesen. In Fällen, in denen Sender und Empfänger nicht der Teil des gleichen SWCs sind, sind die Variablen global. Oftmals ist es sogar so, dass es mehrere Sender und Empfänger existieren, die auf die gleiche globale Variable zugreifen können. In diesem Fall muss die Datenkonsistenz sichergestellt werden, da es zu *Racing Conditions* kommen kann. Die Sicherung kann unter anderem durch verschiedene Mechanismen wie Semaphore passieren. Eine andere Möglichkeit es durch besseres Mapping zu machen, wird im weiteren Verlauf vorgestellt.

Zeitverzögerung

Zeitverzögerungen in der Kommunikation können auftreten, wenn ein Task durch mehrere *WaitPoints* blockiert wird. Müssen Runnable A und Runnable B auf Events warten und der *WaitPoint* von A wird daraufhin aufgelöst, kann es nicht mit der Kommunikation von Runnable A fortgeschritten werden. Erst wenn auch der *WaitPoint* von Runnable B aufgelöst wird, kann

der Task weiter ausgeführt werden, was auch das Ausführen und die Kommunikation von Runnable A wieder möglich macht.

Um die beschriebenen Probleme der intra-ECU-Kommunikation zu lösen, wurden von den Autoren Regeln festgelegt, deren Anwendung zur Lösung der Probleme führen soll. Im folgenden werden die einzelnen Regeln vorgestellt.

Regel 1: *Wenn ein Sender-Runnable nur einen DataSendPoint und keinen WaitPoint hat und das Receiver-Runnable nur einen DataReceivePoint und höchstens einen WaitPoint hat, dann sollten diese beiden Runnables dem selben einfachen Task zugeordnet werden.*

Die Abbildung 4.5 zeigt den Vorteil von der Zuordnung des Senders und des Empfängers zum selben Task gegenüber der Variante der Zuordnung zu verschiedenen Tasks. Um die gesendete Nachricht zu empfangen, muss in der letzteren Variante ein Kontextwechsel vollzogen werden. Dieser kann vermieden werden, wenn beide Runnables im selben Task ausgeführt werden.

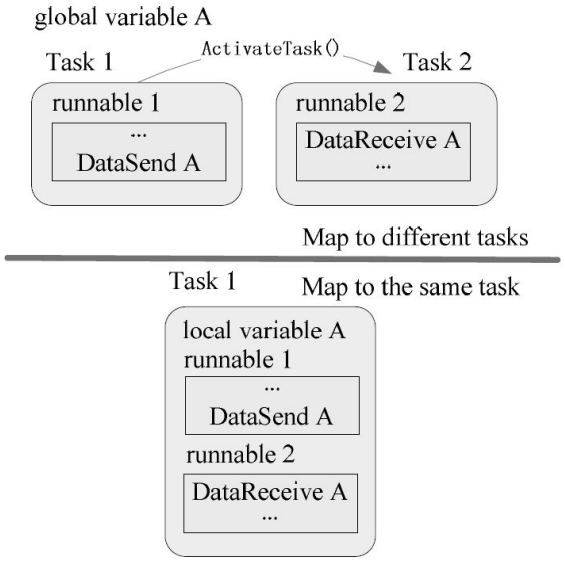


Abbildung 4.5.: Sender- und Receiver-Runnable werden dem gleichen Task zugeordnet, um einen Kontextwechsel zu vermeiden. Übernommen aus [LLP+09].

Regel 2: *Gibt es mehr als zwei Receiver-Runnables und jede von ihnen hat nur einen DataReceivePoint, welcher Zugriff auf die selbe Variable hat, dann sollten diese Runnables auf dem selben Task liegen.*

Abbildung 4.5 verdeutlicht das Problem, wenn mehrere Empfänger der selben Nachricht auf unterschiedlichen Tasks liegen. Zum Empfang muss jedes Mal ein Kontextwechsel durchgeführt werden. Zusätzlich greifen die Empfänger-Runnables auf die globale Variable A zu, was zu *Racing Conditions* führen kann. Diese beiden Probleme können umgangen werden, indem alle Empfänger-Runnables auf den selben Task gelegt werden. Neben der Vermeidung von

Kontextwechseln kann so für jedes Runnable eine Kopie der globalen Variable A erstellt werden, um damit *Racing Conditions* zu verhindern.

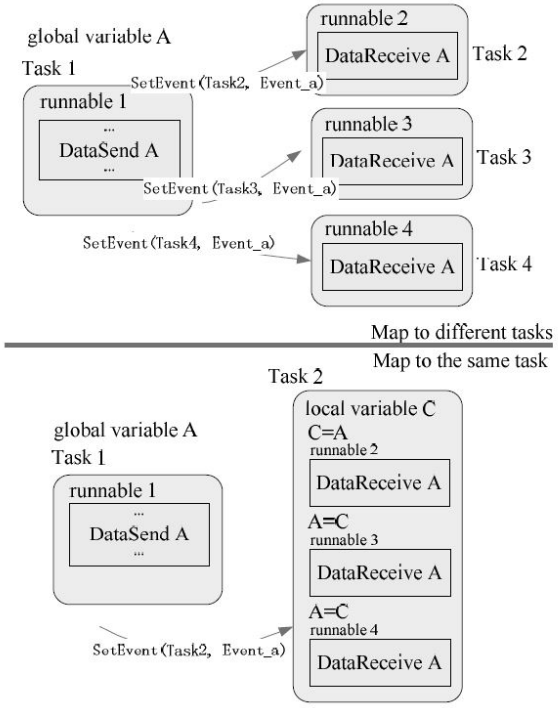


Abbildung 4.6.: Mehrere Receiver-Runnables werden dem selben Task zugeordnet, wenn sie das selbe Sender-Runnable haben. Übernommen aus [LLP+09].

Regel 3: *Wenn zwei Runnables jeweils nur einen DataReadAccess besitzen, über den sie das gleiche Element empfangen, dann sollten diese Runnables auf dem selben einfachen Task liegen.*

Diese Regel ist sehr ähnlich zu der Regel 2; betrifft aber Runnables der Kategorie 1.

Regel 4: *Wenn mehrere zyklische Runnables jeweils nur einen DataSendPoint oder einen DataWriteAccess haben und Daten zum selben Empfänger senden, dann sollten die Runnables auf dem selben einfachen Task liegen.*

Diese Regel ist ähnlich zu der Regel 2, behandelt aber die Problematik von mehreren Sender-Runnables. Wie es die Abbildung 4.7 zeigt, kann man mehrere zyklische Sender-Runnables einem einfachen Task zuordnen, um damit wieder sowohl Kontextwechsel als auch Racing-Conditions zu vermeiden.

Regel 5: *Wenn eine Server-Runnable das Attribut canBeInvokedConcurrently gesetzt hat und zu Kategorie 1 gehört, dann sollten Server und Client auf dem selben Task liegen.*

Durch das Attribut *canBeInvokedConcurrently* können mehrere Instanzen von dem selben Server-Runnable kreiert werden, damit eine parallele Ausführung beim Aufruf durch mehrere

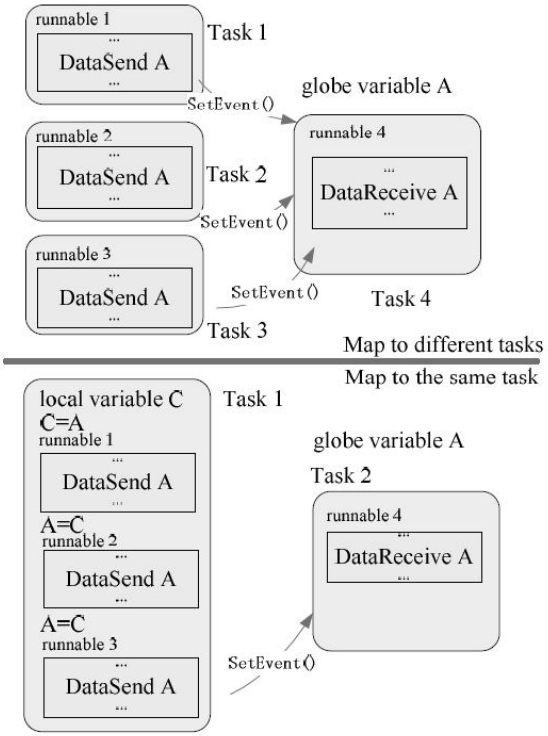


Abbildung 4.7.: Mehrere Sender-Runnables werden dem selben Task zugeordnet, wenn sie den selben Empfänger haben. Übernommen aus [LLP+09].

Clients möglich ist. Diese Regel besagt, dass Server und Client in diesem Fall auf einem Task liegen sollten. Ruft man sich nochmal die Industrieregeln in Erinnerung, dann erkennt man, dass diese Regel im Widerspruch zu der Praktik E3 steht, die besagt, dass es keine Notwendigkeit gibt, Server-Runnables mit diesen Attribut Tasks zuzuordnen. Das Ausführen der Server-Runnables wird bereits im Kontext der Clients durchgeführt.

Regel 6: *Wenn zwei Runnables unterschiedliche WaitPoints haben, dann sollten diese nicht auf dem gleichen Task liegen.*

Mit der Anwendung dieser Regel wird die Problematik der Zeitverzögerung angegangen. Dadurch, dass es keine zwei unterschiedliche *WaitPoints* auf dem selben Task gibt, können Runnables sich nicht mehr gegenseitig blockieren.

4.3.4. Flexibilität des Verfahrens

In der Praxis zeigt sich oft, dass die Verwendung von einfachen Industriepraktiken nicht ausreichend ist. Häufig ist es der Fall, das Integratoren projektspezifische Anpassungen des RTM durchführen müssen, um die korrekte Ausführung der Applikation sicherstellen zu können. Beispielsweise kann es erforderlich sein, bestimmte Runnables einem speziellen Task

zuzuordnen. Logischerweise kann ein generisches Verfahren, wie es in diesem Kapitel vorgestellt wurde, solche Anforderungen nicht erfüllen. Deshalb ist es erforderlich, Überlegungen anzustellen, wie man dem Konzept für das automatische RTM die Flexibilität geben kann, damit die projektspezifischen Anforderungen erfüllt werden können.

Vorangegangene Abschnitte haben gezeigt, dass der Ablauf von einem RTM in Form von Regeln beschrieben werden kann. Diese Idee kann man aufgreifen und durch die Erweiterung der Regeln die gewünschten projektspezifischen Anpassungen erzwingen. Dem Integrator soll also eine Möglichkeit gegeben werden, Regeln zu formulieren und es an die Programmlogik zu übergeben.

Eine Anpassung des Programmcodes soll vermieden werden, um einerseits Fehler im Ablauf zu vermeiden und andererseits Trennung zwischen generischen und projektspezifischen Komponenten zu erreichen. Deshalb ist es ratsam, dem Integrator eine benutzerfreundliche Schnittstelle zur Verfügung zu stellen, in der er sein Projektwissen in Form von Regeln an das Programm übergeben kann.

Eine gute Option dafür stellt ein Business-Rule-Management-System (BRMS) dar. Dieses ermöglicht, durch die Formulierung von Regeln, Einfluss auf den Programmablauf zu nehmen, ohne dabei den Programmcode zu ändern. Zum Erstellen und Editieren von Regeln wird durch das BRMS ein Editor zur Verfügung gestellt. Eingetragene Regeln werden in einer Datenbank gespeichert und bei der Ausführung des Programms durch die Softwarekomponente Business-Rule-Engine angewendet. [Ros03]

4.3.5. Gesamtkonzept

In den vorangegangenen Kapiteln wurden Ideen vorgestellt, die zu einer Lösung für das automatische RTM führen sollen. In diesem Abschnitt soll nun beschrieben werden, wie diese Ideen zu einem Gesamtkonzept integriert werden.

Der erste Schritt sollte sein, eine funktional korrekte Lösung zu erhalten. Die Industriepraktiken reichen dafür meist nicht aus; diese bieten aber ein Mapping-Verfahren für Runnables, welche nicht von projektspezifischen Anforderungen betroffen sind. Durch den Einsatz des BRMS kann die entsprechende Erweiterung durchgeführt werden, um die gewünschten Funktionalitäten zu realisieren. Dafür muss durch den Integrator ein Konfigurationsschritt vorgenommen werden, wobei die notwendigen Regeln erstellt werden. Anschließend erfolgt das Zuordnen der Runnables zu den Tasks, was in einem voll funktionsfähigen RTM resultiert.

Im nächsten Schritt können angelegte Tasks eine Zuweisung in einfache und erweiterte Tasks erhalten. Dafür werden die im Abschnitt 4.3.2 vorgestellten AUTOSAR Regeln verwendet.

Die Optimierung des Resultats erfolgt anschließend im letzten Schritt. Abschnitt 4.3.3 beschreibt die Regeln, die zur Optimierung verwendet werden. Zur Ausführung dieser Regeln kann die BRMS aus dem ersten Schritt genutzt werden. Zu beachten ist hierbei, dass durch die Anwendung der Optimierung und der folgenden Änderung der Belegung der Tasks, nicht

die vom Integrator festgelegten Regeln verletzt werden, da es die korrekte Ausführung der Applikation behindern könnte.

5. Implementierung

5.1. Datenexport

Wie bereits in Abschnitt 2.1.3 erwähnt wurde, liegen die Beschreibungen der SWCs im ARXML Format vor. Dank der großen Softwareunterstützung für AUTOSAR gibt es eine Reihe von praktischen Werkzeugen, welche die Verarbeitung des ARXML Formats stark erleichtern. So können die Dateien in den Cfg Pro (siehe Abschnitt 2.2.1) importiert werden. Der große Vorteil von der Verwendung des Cfg Pro ist, dass diesem das AutomationInterface (AI) zur Verfügung gestellt wird. Das AI ermöglicht es, auf den Inhalt der ARXML Dateien in Form von Java-Objekten zuzugreifen und diese ggf. zu ändern. Damit entfällt die Notwendigkeit die ARXML Dateien zu parsen. Das AI unterstützt Groovy (siehe Abschnitt 2.2.3) und Java als Programmiersprache. Über Groovy-Skripte können die Informationen zu SWCs, Runnables und Ports ausgelesen und verändert werden.

5.2. Port-Mapping

Der erste Schritt der Implementierung des PM besteht in der Zuordnung von Ports in die Gruppen Sender, Receiver, Server und Client. Der bipartite Graph zwischen Provider- und Require-Ports kann als eine Matrix dargestellt werden, wie es die Abbildung 5.1 zeigt. Der Zeilenindex p repräsentiert dabei ein Provider-Port und der Spaltenindex r ein Require-Port. Der Wert s_{pr} an der Stelle (p, r) stellt das Gewicht der Kante zwischen den Ports p und r dar.

$$\begin{array}{c} \\ p_1 \\ p_2 \\ \vdots \\ p_n \end{array} \begin{array}{c} \begin{array}{cccc} r_1 & r_2 & \dots & r_n \end{array} \\ \begin{pmatrix} s_{11} & 0 & \dots & a_{1n} \\ 0 & s_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & s_{nn} \end{pmatrix} \end{array}$$

Abbildung 5.1.: Darstellung der Ähnlichkeitsmatrix für Provider- und Require-Ports

Im Abschnitt 4.2.2 wurde festgestellt, dass die Metriken *Jaccard Similarity* und *Simple Word Overlap* gut für die Berechnung der Ähnlichkeit zwischen zwei Portnamen geeignet sind. Der Vergleich in [CRF03] zwischen diesen beiden Verfahren hat gezeigt, dass die Leistungsunterschiede sehr gering sind. Für die Implementierung fiel die Entscheidung auf *Jaccard Similarity*, da hierfür bereits eine Java-Bibliothek existiert. Die *java-string-similarity*[1] Bibliothek stellt eine Reihe von verschiedenen Algorithmen zur Verfügung, darunter auch *Jaccard Similarity*.

Der nächste Schritt wäre nun, die Matrix und die Portnamen an den Algorithmus für *Jaccard Similarity* zu übergeben, damit die Kantengewichte berechnet werden können. Es besteht allerdings die Notwendigkeit zur Vorverarbeitung von den Portnamen. Diese liegen in einzelnen Strings vor, welche aus mehreren Wörtern bestehen. Folglich müssen diese Strings in mehrere Teilstrings aufgeteilt werden, damit die Wörter verglichen werden können. Für die konkrete Trennung können die Großbuchstaben als Orientierungspunkt verwendet werden, denn ein neues Wort fängt immer mit einem Großbuchstaben an. Beachtet werden sollte allerdings, dass einige Wörter nur aus Großbuchstaben bestehen und in diesem Fall besonders behandelt werden müssen. Für die Implementierung der Trennung wird ein regulärer Ausdruck (Regex) verwendet.

Nachdem die Namen in mehrere Wörter aufgeteilt sind, sollte man sicherstellen, dass die selben Wörter, welche sich nur in der Groß- und Kleinschreibung unterscheiden, als gleich erkannt werden. Dazu werden Java-interne Methoden zur Stringverarbeitung verwendet, um alle Buchstaben eines Wortes in Kleinbuchstaben umzuwandeln. Nach der Vorverarbeitung kann die Berechnung der Ähnlichkeit durchgeführt und die Kantengewichte in die Matrix eingetragen werden.

Im Abschnitt 4.2.3 wurde erläutert, dass nicht alle Ports miteinander kompatibel sind und diese nicht miteinander verbunden werden dürfen. Aus diesem Grund wird die Berechnung der Ähnlichkeit nicht durchgeführt, wenn zwei Ports nicht kompatibel sind. Der Matrixeintrag für zwei inkompatible Ports bekommt stattdessen einen negativen Ähnlichkeitswert.

Nach der Gewichtung der Kanten und der Anwendung der Filter folgt nun der letzte Schritt des PM. Das Ziel ist, die Kanten mit den höchsten Gewichten auszuwählen und somit ein Maximum-Matching zu erhalten. Im Abschnitt 4.2.4 wurde festgelegt, dass zur Lösung dieses Problem der Ungarische Algorithmus verwendet werden soll. Die Java-Bibliothek *software-and-algorithms*[2] stellt eine Implementierung des Verfahrens zur Verfügung. Als Resultat liefert es einen Java-Array zurück. Der Index steht dabei für einen Provider-Port und der Wert des Feldes unter dem Index, gibt den Require-Port an. Da der Ungarische Algorithmus auch Ports verbindet, die nicht miteinander kompatibel sind, werden diese Verbindungen in einem anschließenden Schritt gelöst. Hierfür wird nach Portverbindungen mit negativen Kantengewichten gesucht und diese entbunden.

[1] https://github.com/tdebatty/java-string-similarity
[2] https://github.com/KevinStern/software-and-algorithms

5.3. Runnable-Task-Mapping

5.3.1. Datenmodel

Sowohl für die Industriepraktiken als auch für die Optimierung ist es notwendig zu wissen, wie Runnables miteinander kommunizieren. Die Beschreibung der SWCs in den ARXML Dateien gibt zwar an, welche Ports ein Runnable zur Kommunikation verwendet, aber es enthält keine expliziten Informationen darüber, welche Runnables mit welchen anderen Runnables kommunizieren. Um die Kommunikation zwischen Runnables erfassen zu können, muss man das PM hinzuziehen. Sind zwei Ports miteinander verbunden und verwenden zwei Runnables jeweils einen dieser Ports, dann bedeutet es im Folgeschluss, dass zwischen den Runnables Kommunikation besteht.

Um Beziehungen zwischen Entitäten, in diesem Fall Runnables, abzubilden, eignet sich besonders gut eine relationale Datenbank. Für die Umsetzung der Industriepraktiken und der Optimierung, muss man aus dem Datenmodel folgende Informationen entnehmen können:

1. Zugriff von Runnables auf Ports
2. Mapping von Ports
3. Zugriffspunkte von Runnables
4. WaitPoints von Runnables

Abbildung A.3 stellt das Datenmodel dar, welches diese Informationen zur Verfügung stellt. Die Tabellen enthalten die benötigten Informationen, um die Regeln aus dem Konzept anwenden zu können. Die Kommunikation zwischen Runnables kann man über eine Abfrage an die Tabellen *RequirePort*, *ProviderPort* und *PortConnection* herausfinden. Die Tabellen *RequirePort* und *ProviderPort* enthalten Informationen darüber, welche Runnables auf welche Ports zugreifen. *PortConnection* zeigt, wie die Ports verbunden sind. Die Informationen für die Zugriffspunkte und WaitPoints der Runnables sind in den Tabellen *AccessPoint* und *WaitPoint* zu finden. Tabelle *Runnable* beinhaltet, neben sonstigen Attributen, auch welchem Task die Runnable zugeordnet ist.

Für die Implementierung des Datenmodels wurde die Java SQL Datenbank *H2 Database Engine* (siehe Abschnitt 2.2.4) verwendet.

5.3.2. Rule Engine

Für die Umsetzung der BRMS wurde die Open Source Lösung *Drools* (siehe Abschnitt 2.2.5) verwendet. Für die Nutzungs der Rule Engine von Drools wird dem Benutzer eine DRL-Datei zur Verfügung gestellt. Drools verfügt über eine eigene Regelsprache, in der die Regeln formuliert sein müssen. Die definierten Regeln müssen in dieser Regelsprache in die DRL-Datei

geschrieben werden. Abbildung 5.2 zeigt die generelle Struktur für eine Regel in der Sprache von Drools. Eine Regel hat immer einen Namen und kann optional noch Attribute bekommen. Hinzukommt, dass eine Regel zwei Seiten besitzt. Auf der linken Seite werden die Bedingungen formuliert und auf der rechten Seite die folgenden Aktionen, die durchgeführt werden müssen, wenn die Bedingungen erfüllt worden sind. In der vorliegenden Format kommt der Inhalt der linken Seite unter dem Token *when* und der Inhalt der rechten Seite unter dem Token *then*.

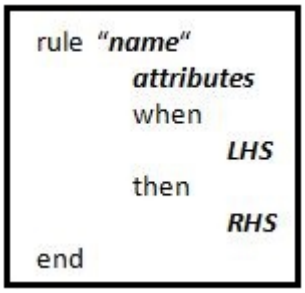


Abbildung 5.2.: Generelle Struktur einer Regel in Drools-Sprache. Nach [Com17].

6. Evaluierung

6.1. Testdaten

Für die Evaluierung des Konzepts wurde von Vector ein aktuell laufendes Projekt zur Verfügung gestellt. Dabei handelt es sich um die Entwicklung eines Telematiksteuergeräts für die kommende Fahrzeuggeneration eines großen deutschen OEM. Der aktuelle Entwicklungsstand befindet sich in der Vorserienentwicklung; der erste Serienlauf ist für die Mitte des Jahres 2018 geplant.

Abbildung 6.1 zeigt die Architektur des AUTOSAR Softwareprojekts. Der Vector-Anteil an der Entwicklung liegt dabei auf der Umsetzung der Funktionalitäten für den Mikrocontroller (engl. Microcontroller Unit (MCU)). Die implementierte Software soll dabei Funktionalitäten wie z.B. Software Update, Bindung der Fahrzeugnetzwerke an den Applikationsprozess und Peripherie-Diagnose enthalten. Zusätzlich findet man darunter auch zeitkritische Prozesse wie z.B. Koppelnavigation.

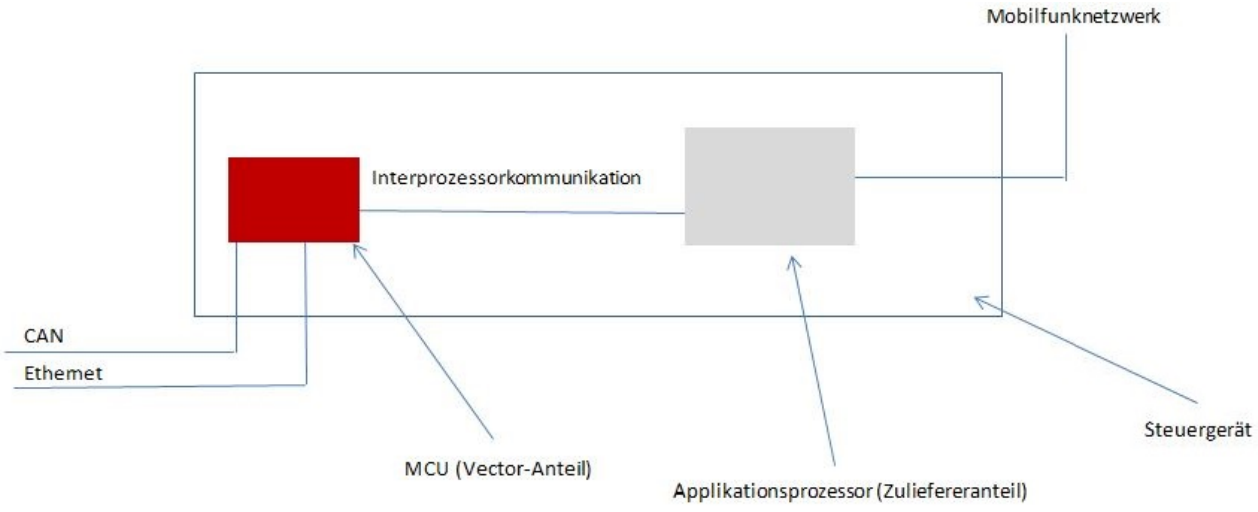


Abbildung 6.1.: Architektur des Telematik-Projekts

Insgesamt besitzt die vorliegende AUTOSAR Software 825 Sender/Receiver-Ports und 103 Client/Server-Ports. Die Gesamtanzahl der Runnables liegt bei 604.

6.2. Port-Mapping

6.2.1. Methoden

Funktionale Korrektheit

Bei einem PM ist die Zuordnung von Provider-Ports zu Require-Ports immer eindeutig. Im vorliegenden Fall liegt mit der durch den Integrator manuell erstellten Lösung des Problems ein korrekt ausgeführtes PM vor. Folglich wird diese als Referenz verwendet, um eine Leistungsbeurteilung von dem Resultat aus dem automatischen PM durchzuführen.

Aus der Betrachtung der Integratorlösung lässt sich ableiten, dass nicht alle Ports einander zugeordnet sind. Aus der Tatsache, dass ein Teil der Ports verbunden ist und ein Teil nicht verbunden ist, lässt sich sagen, dass Ports zwei Klassen zugeordnet werden können: *gemappt* und *nicht gemappt*. Analog dazu, kann der konzipierte Algorithmus zum PM als ein binärer Klassifikator betrachtet werden, der die einzelnen Ports diesen Klassen zuordnet. Eine häufige Methode zur Leistungsbewertung eines binären Klassifikators ist die Ableitung von quantitativen Merkmalen aus den Fehlern, die dieser während der Klassifikation macht [Pow11]. In diesem Fall kann der Klassifikator zwei Fehler machen. Der erste Fehler passiert, wenn ein Port, welcher gemappt sein sollte, einem falschen oder keinem Port zugeordnet wird. Der zweite Fehler tritt dann auf, wenn ein Port, welcher frei bleiben sollte, auf einen anderen Port gemappt wird. Insgesamt können bei der Klassifizierung eines Ports vier mögliche Fälle auftreten:

1. **Richtig positiv:** Ein Port wurde korrekt gemappt.
2. **Falsch negativ:** Ein Port wurde nicht oder falsch gemappt, obwohl dieser gemappt sein sollte.
3. **Falsch positiv:** Ein Port wurde gemappt, obwohl dieser nicht gemappt sein sollte.
4. **Richtig negativ:** Ein Port wurde korrekt nicht gemappt.

Zählt man nun, wie oft jedes der vier Fälle in dem Ergebnis des PM vorkommt, kann man die resultierenden relativen Häufigkeiten in einer Wahrheitsmatrix eintragen, wie es Tabelle 6.1 zeigt.

Tabelle 6.1.: Wahrheitsmatrix zum Port-Mapping

	Gemappt	**Nicht gemappt**
Positiv	richtig positiv	falsch positiv
Negativ	falsch negativ	richtig negativ

Aus den relativen Häufigkeiten lassen sich nun Kenngrößen errechnen, die letztendlich zur Bewertung des automatischen Verfahren verwendet werden. Folgende Kenngrößen werden verwendet:

1. **Sensitivität:** Anteil der Ports, die korrekt als gemappt klassifiziert worden sind, an der Gesamtheit der tatsächlich gemappten Ports
2. **Falsch-negativ-Rate:** Anteil der Ports, die als falsch negativ klassifiziert worden sind, obwohl sie eigentlich gemappt sind
3. **Spezifität:** Anteil der Ports, die korrekt als nicht gemappt klassifiziert worden sind, an der Gesamtheit der tatsächlich nicht gemappter Ports
4. **Falsch-positiv-Rate:** Anteil der Ports, die als falsch positiv klassifiziert worden sind, obwohl sie eigentlich nicht gemappt sind
5. **Korrektklassifikationsrate:** Anteil der falsch positiven und falsch negativen Ports
6. **Falschklassifikationsrate:** Anteil an richtig positiven und richtig negativen Ports an der Gesamtzahl

Ausführungszeit

Zum Vergleich der Ausführungszeiten zwischen dem manuellen und automatischen PM wurde der zeitliche Aufwand gemessen, um die jeweilige Methode durchzuführen. Im Fall des manuellen Mappings wurde das gegebene PM des Integrators im Cfg Pro nachgebildet. Für die automatische Lösung wurde die Ausführungszeit der Applikation gemessen.

6.2.2. Ergebnisse

Sender-Receiver-Interface

In der Tabelle 6.2 sind die Ergebnisse des automatischen PM für die Sender- und Receiver-Ports aufgeführt. Die Tabelle 6.3 zeigt die dazugehörigen Kennzahlen. Mit 98% wurde bei der Sensitivität ein sehr hoher Wert erreicht, welcher bedeutet, dass die meisten gemappten Ports korrekt einander zugeordnet worden sind. Nur zwei Ports konnten nicht korrekt einander zugeordnet werden, was sich in einer Falsch-nagetiv-Rate von nur 2% zeigt. Ein deutlich schlechteres Resultat lässt sich bei der Klassifikation von nicht-gemappten Ports finden. Die Spezifizität liegt gerade mal bei 56%, da mit 5 Ports nur etwas mehr als die Hälfte der freien Ports korrekt keinem anderen zugeordnet worden sind. Dazu muss allerdings erwähnt werden, dass die Anzahl der nicht-gemappter Ports bei 9 Ports liegt. Insgesamt betrachtet, zeigt sich eine hohe Korrektklassifikationsrate von 94%, da 97 der 103 Ports vom Algorithmus die richtige Konfiguration erhalten haben.

Tabelle 6.2.: Wahrheitsmatrix aus dem automatischen Port-Mapping für die Sender-Receiver-Ports

	Gemappt	**Nicht gemappt**
Positiv	92	4
Negativ	2	5

Tabelle 6.3.: Kennzahlen aus dem automatischen Port-Mapping für die Sender-Receiver-Ports

Sensitivität	**Falsch-negativ-Rate**	**Spezifizität**
0,98	0,02	0,56
Falsch-positiv-Rate	**Korrektklassifikationsrate**	**Falschklassifikationsrate**
0,44	0,94	0,06

Client-Server-Interface

In der Tabelle 6.4 sind die Ergebnisse des automatischen Port-Mappings für die Client- und Server-Ports aufgeführt. Die Tabelle 6.5 zeigt die dazugehörigen Kennzahlen. Im Gegensatz zum Sender-Reicever-Mapping ist in diesem Fall zu sehen, dass sowohl die Sensitivität mit 93% als auch die Spezifizität mit 92% sehr hoch sind. 430 von 464 Ports wurden korrekt gemappt und nur 34 konnten nicht richtig zugeordnet werden. 327 von 361 Ports wurden korrekterweise keinem anderen Port zugeordnet. Insgesamt wurden 757 Ports korrekt klassifiziert und damit liegt die Korrektklassifikation bei 92%. 8% der Ports wurden falsch zugeordnet.

Tabelle 6.4.: Wahrheitsmatrix aus dem automatischen Port-Mapping für die Sender-Receiver-Ports

	Gemappt	**Nicht gemappt**
Positiv	430	34
Negativ	34	327

Ausführungszeit

Das manuelle PM wurde mit dem Tool Cfg Pro durchgeführt. Die Dauer für die Nachbildung der Integratorlösung lag bei rund 67 Minuten. Die Ausführung des automatisierten Port-Mappings erfordert keine manuelle Konfiguration. Die Ausführungszeit besteht somit nur aus der Laufzeit von dem Start der Applikation bis zur Ausgabe des resultierenden Mappings. Die Ausführungszeit für das vorliegende Projekt lag bei genau 2,8 Sekunden.

Tabelle 6.5.: Kennzahlen aus dem automatischen Port-Mapping für die Client-Server-Ports

Sensitivität	**Falsch-negativ-Rate**	**Spezifizität**
0,93	0,07	0,91
Falsch-positiv-Rate	**Korrektklassifikationsrate**	**Falschklassifikationsrate**
0,09	0,92	0,08

6.2.3. Diskussion

Dieser Abschnitt dient dem Zweck zu prüfen, ob eine Integration des in dieser Arbeit konzipierten Verfahrens für das automatische Port-Mapping in der Vector-CI-Pipeline als sinnvoll erscheint. Um das zu ermöglichen, muss das Verfahren einige Bedingungen erfüllen. Zum einen muss es ein funktional korrektes Ergebnis liefern und zum anderen soll es in einen akzeptablen Zeitrahmen geschehen. Ob diese Bedingungen erfüllt worden sind, soll nun anhand der Ergebnisse diskutiert werden.

Von vornherein kann man sagen, dass die Ausführungszeit mit 2,8 Sekunden ein sehr gutes Ergebnis ist. Bei einer maximal geforderten Durchlaufzeit von 30 Minuten von der Vector-CI-Pipeline wird dieses Verfahren zum Port-Mapping mit sehr hoher Wahrscheinlichkeit nicht die Zeitgrenze überschreiten lassen. Der zeitliche Aspekt ist in diesem Fall erfüllt.

Um ein funktional korrektes Mapping zu erhalten, muss die Korrektklassifizierungsrate bei 100% liegen. Für das automatisierte PM liegt diese allerdings nur bei 94% für das Sender-Receiver-Mapping und bei 92% für das Client-Server-Mapping. Folglich wurde das Ziel nicht erreicht und eine nachträgliche manuelle Konfiguration durch den Integrator ist notwendig. Ob die Applikation auf dem Steuergerät korrekt ausgeführt wird, kann durch Test festgestellt werden. Diese Test sind bereits automatisiert und in der Vector-CI-Pipeline integriert. Fehlgeschlagene Tests, welche z.B. ein inkorrektes PM als Basis haben könnten, werden dem Integrator sofort mitgeteilt. Das Problem dabei ist, dass es nicht klar ist, welche konkreten Ports ein falsches Mapping haben. Folglich muss der Integrator Zeit investieren, um die Ports zu finden, welche anders konfiguriert sein sollen.

Um eine Korrektklassifizierungsrate von 100% zu erreichen, müssten alle Ports korrekt einander zugeordnet werden. Die Wahrscheinlichkeit dafür ist eher gering, denn nicht alle Ports der BSW sind standardisiert (siehe Abschnitt 2.1.3). Dadurch, dass die Applikation und die BSW meist unabhängig voneinander entwickelt werden, steht das Wissen über die Portnamen der BSW für die Entwickler der Applikation selten zur Verfügung. Deshalb erschwert es die Zuordnung der Ports über die Namensähnlichkeit, was die Notwendigkeit der manuellen Integration zur Folge hat.

Eine Idee, wie man die manuelle Nachkonfiguration beschleunigen könnte, wäre die Bereitstellung der Namensähnlichkeiten. Es ist davon auszugehen, dass bei einer hohen Namensähnlichkeit, die Wahrscheinlichkeit größer ist, dass die Ports zusammenpassen. Daran könnte sich der Integrator orientieren, um die Fehler in der Konfiguration zu finden. Ein weiterer Punkt,

der hier beachtet werden sollte, ist, dass anders als in dieser Arbeit, die SWCs eines Projekts sehr selten alle auf einmal integriert werden. Üblicherweise passiert es über die gesamte Laufzeit eines Projekts stückweise. Daraus folgt, dass nur eine kleine Teilmenge der Ports in einem Integrationsschritt verbunden werden soll. Das reduziert den Aufwand der manuellen Nachkonfiguration, falls diese notwendig sein sollte.

Zusammenfassend kann man sagen, dass der vorgestellte Ansatz zum Port-Mapping nur bedingt geeignet ist, da immer noch regelmäßige manuelle Eingriffe durch den Integrator zu erwarten sind. Der Vorteil bei der Verwendung ist, dass diese Eingriffe nur auf die Fälle reduziert werden können, bei denen die Tests fehlschlagen. Liefert der Algorithmus ein korrektes Mapping, wird manuelle Arbeit vermieden.

6.3. Runnable-Task-Mapping

6.3.1. Methoden

Um die Evaluation des RTM durchzuführen, wird ein Vergleich zwischen der Lösung des Integrators und der Lösung aus dem automatisierten Ansatz durchgeführt.

Sicherstellen der funktionalen Korrektheit

Wenn man bei der Integratorlösung davon ausgehen konnte, dass diese funktional korrekt ist, muss man die Korrektheit bei der automatischen Lösung überprüfen. Die Vorgabe dazu, liefern die projektspezifischen Eigenschaften des Mappings aus der Integratorlösung. Folgende Eigenschaften müssen durch die automatische Lösung erfüllt werden:

1. Alle Runnables der SWC *LSC* müssen auf eigenen Tasks liegen.
2. Es gibt drei Runnables, die für die Signaturprüfung verantwortlich sind. Diese müssen auf einem Task liegen. Die Integratorlösung sieht dafür den Task *WorkerH* vor.
3. Drei spezielle Server-Runnables der SWC *PiaClient* müssen einem Task zugeordnet werden. Diese würden ansonsten durch die Anwendung der Industriepraktiken kein Mapping erhalten.
4. Alle Main-Funktionen der Service-Komponenten müssen auf einem eigenen Task ausgeführt werden. In der Integratorlösung sind diese im Task *SchM* enthalten.

Vergleich von nicht-funktionalen Anforderungen

Anders als bei funktionalen Anforderungen, kann in diesem Fall nicht davon ausgegangen werden, dass die Verteilung der Runnables auf die Tasks optimal ist. Um einen Vergleich auf Basis von nicht-funktionalen Eigenschaften zwischen der manuellen und der automatischen Lösung durchführen zu können, wurden zwei Fehlerarten definiert. Definition 6.3.1 beschreibt Zuordnungsfehler, welche entstehen, wenn die im Abschnitt 4.3.1 definierten Industriepraktiken nicht eingehalten werden. Verletzungen dieser Praktiken können einen negativen Effekt auf den Determinismus und die Latenz zur Laufzeit haben, wie im Abschnitt 4.3.1 beschrieben worden ist. Definition 6.3.2 zeigt die zweite Fehlerart. Optimierungsfehler werden durch das Verletzen der Optimierungsregeln verursacht. Wie im Abschnitt 4.3.3 beschrieben, kann die Nichtbeachtung dieser Regeln zu Latenzproblemen und Dateninkonsistenz führen.

Definition 6.3.1 (Zuordnungsfehler)
Sei τ ein Task, welcher für das Mapping von Runnables mit dem RTEEvent α festgelegt wurde. Ein Zuordnungsfehler tritt genau dann auf, wenn ein Runnable mit einem RTEEvent β dem Task τ zugeordnet wird.

Definition 6.3.2 (Optimierungsfehler)
Ein Optimierungsfehler tritt dann auf, wenn das Mapping eines Runnable gegen die in [LLP+09] festgelegten Optimierungsregeln verstößt.

Vergleich der Ausführungszeiten

Des Weiteren werden die Ausführungszeiten zwischen der manuellen und der automatischen Methode verglichen. Unter Ausführungszeiten versteht man in diesem Kontext den zeitlichen Aufwand, den man betreiben musste, um die entsprechende Lösung des Problems zu erhalten. Um die Ausführungszeit des manuellen Mapping mit dem automatischen vergleichbar zu machen, wurde die Annahme getroffen, dass die projektspezifischen Eigenschaften vollständig bekannt sind. Das beutet, dass der Integrator bereits das notwendige Wissen besitzt, um ein funktional korrektes Mapping durchzuführen.

6.3.2. Ergebnisse

Integratorlösung

Analyse von nicht-funktionalen Eigenschaften

Die Analyse der Integratorlösung ergab 78 Zuordnungsfehler. In der Abbildung A.4 ist die Integratorlösung graphisch dargestellt und alle Zuordnungsfehler sind rot markiert. Tabelle 6.6 zeigt die Verteilung in absoluten Zahlen von korrekt zugeordneten Runnables und falsch zugeordneten Runnables in den entsprechenden Tasks. Besonders auffällig sind dabei die ersten

vier Tasks der oberen Reihe, die zusammen 63 Fehler enthalten. Aus der näheren Betrachtung der ersten drei Tasks, die alle jeweils periodisch sind, lässt sich erkennen, dass auch die Runnables mit einem Init-Trigger diesen Tasks zugeordnet worden sind. Die Absicht dahinter ist, dass man erst die Initialwerte setzen möchte, bevor man die Main-Funktion ausführt. Neben den bereits genannten Folgen eines Zuordnungsfehlers, führt die Verteilung der Init-Runnables auf drei Tasks dazu, dass das Auslösen des Init-Triggers durch die BSW, es zu zwei überflüssigen Kontextwechseln kommt. Das kann man vermeiden, wenn man alle Init-Runnables auf einen Tasks legt.

Tabelle 6.6.: Zuordnungsfehler der Integratorlösung aufgegliedert nach Tasks

Task	**5 ms**	**10 ms**	**100 ms**	**Event 1**	**Event 2**	**LSC 1**
Richtig	6	17	4	3	14	26
Falsch	27	19	5	12	0	0
Task	**LSC 2**	**ModeSwitch 1**	**ModeSwitch 2**	**ModeSwitch 3**	**SchM**	**WorkerH**
Richtig	23	15	6	4	44	3
Falsch	0	6	2	2	5	0

Aus der Betrachtung der Verteilung der periodischen Runnables lässt sich weiteres Optimierungspotential erkennen. Liegen alle Runnables mit der gleichen Periode auf einem Task, können diese bei dem entsprechenden Aufruf durch die BSW im gleichen Kontext ausgeführt werden. Sind sie allerdings über mehrere Tasks verteilt, kommt es zwangsläufig zum Kontextwechsel. Dieses Problem findet man bei Runnables mit den Perioden von 10 ms (2 Tasks), 100 ms (2 Tasks) und 1000 ms (3 Tasks). Durch das korrekte Mapping dieser Runnables kann man insgesamt 4 Kontextwechsel vermeiden.

Starke Auffälligkeiten bemerkt man bei dem Mapping von Event- und ModeSwitch-Runnables. Diese sind nicht vollständig den dafür entsprechenden Tasks zugeordnet. Jedes der drei ModeSwitch-Tasks enthält auch Event-Runnables und das Mapping des Tasks Event 1 besteht zum Großteil aus ModeSwitch-Runnables. Der Task *5 ms* beinhaltet sowohl Event- als auch ModeSwitch-Runnables. Ein Event-Runnable findet man auch in dem Task *SchM*, welches für Runnables der Service-Komponenten bestimmt ist. Keines dieser Zuordnungsfehler hat einen funktionalen Grund und bringt somit nur negative Effekte aus der nicht-funktionalen Sicht.

Die Auswertung der Integratorlösung in Bezug auf die Optimierungsfehler ist in der Tabelle 6.7 aufgeführt. Es gab jeweils eine Optimierungsmöglichkeit durch die Anwendung der Regeln 1 bis 3, wobei die ersten zwei Möglichkeiten vom Integrator korrekt erkannt worden sind. Die dritte Möglichkeit wurde nicht erkannt und führt somit zu einem Optimierungsfehler durch den Verstoß gegen Regel 3. Dieser resultiert in zwei unnötigen Kontextwechseln. Die einzelnen Optimierungsmöglichkeiten sind in der Abbildung A.5 farblich dargestellt. Der Sender und Receiver für die Regel 1 (blau) sind korrekterweise zusammen in dem Task *5 ms* zu finden. Die beiden Receiver, die zur zweiten Regel (orange) gehören, findet man im Task *LSC 1* und der

Tabelle 6.7.: Optimierungsmöglichkeiten und Optimierungsfehler der automatisierten und der manuellen Lösung

Regel	Anzahl	Integrator	Automatisierung
1	1	1	1
2	1	1	1
3	1	0	1
4	0	0	0
5	0	0	0
6	0	0	0
Kontextwechsel		2	0

Sender befindet sich im Task *5 ms*. Die drei Runnables, die ein violette Markierung besitzen, befinden sich jeweils in drei unterschiedlichen Tasks. Deshalb werden zum Empfangen zwei Kontextwechsel notwendig sein.

Analyse der Ausführungszeit

Um das manuelle Mapping durchzuführen, steht dem Integrator das Tool Cfg Pro zur Verfügung. Dieses Tool bietet ein Benutzerinterface, welches das Zuordnen von Runnables zu Tasks durch einfaches *Drag and Drop* ermöglicht. Um die Ausführungszeit zu ermitteln, wurde die Zeit gemessen, die man gebraucht hat, um das vorliegende Mapping der Integratorlösung im Cfg Pro nachzubilden. Die resultierende Ausführungszeit lag bei 50 Minuten und 24 Sekunden.

Automatische Lösung

Sicherstellen der funktionalen Korrektheit

Per Definition kann der generische Runnable-Task-Algorithmus keine projektspezifischen Eigenschaften erfassen und die geforderten Eigenschaften in das Ergebnis einfließen lassen. Praktischerweise eröffnet die Verwendung einer Rule Engine für den Integrator die Möglichkeit, durch das Einfügen von einfachen Regeln, auf das Laufzeitverhalten des Algorithmus einzuwirken (Namen der Runnables wurden anonymisiert):

1. Wenn ein Runnable zur SWC *LSC* gehört,
   dann ordne es dem Task *LSC* zu.
2. Wenn ein Runnable den Namen *X* oder *Y* oder *Z* hat
   dann ordne es dem Task *WorkerH* zu.
3. Wenn ein Runnable den Namen *A* oder *B* oder *C* hat und zur SWC *PiaClient* gehört, dann ordne es einem Event-Task zu.

4. Wenn ein Runnable eine Main-Funktion ist und zu einer Service-Komponente gehört, dann ordne es dem Task *SchM* zu.

Aus der Abbildung A.6 kann man erkennen, dass alle geforderten projektspezifischen Eigenschaften korrekt umgesetzt worden sind. In den Tasks *Task_LSC_1* und *Task_LSC_2* findet man die Runnables der SWC *LSC*. Analog zur Integratorlösung gibt es einen Task *WorkerH*, welcher die Runnables für die Signaturprüfung beinhaltet. Die Server-Runnables der SWC *PiaClient* findet man im Task *Event_2*. Mit dem Task *SchM*, welcher die Main-Funktionen der Service-Komponenten enthält, ist auch die letzte Anforderung erfüllt.

Analyse von nicht-funktionalen Eigenschaften

Lässt man projektspezifische Eigenschaften der Lösung außer Acht, dann sieht man, dass die Runnables strikt nach ihrem jeweiligen RTEEvents verteilt sind. Dadurch lassen sich die meisten der Zuordnungsfehler vermeiden. Die Folge davon ist, dass unnötige Kontextwechsel vermieden werden. Insgesamt hat die automatische Lösung zwei Zuordnungsfehler. Diese sind jedoch das Resultat der Optimierung.

Aus der Abbildung A.7 kann man entnehmen, dass alle drei Optimierungsmöglichkeiten vom Algorithmus korrekt erkannt und umgesetzt worden sind. Der Sender und der Receiver aus der ersten Regel, in Blau markiert, liegen auf dem selben Task. Die Optimierungsmöglichkeit aus der zweiten Regel, in Orange markiert, ist ebenfalls erfüllt, da beide Receiver auf dem selben Task liegen. Es gäbe die Option, diese Receiver auf den Task des Senders (*Task_5ms*) zu legen. Das würde allerdings die erste funktionale Anforderung verletzen. Das Mapping der zwei in Violette markierten Event-Runnables, die in dem Task *Task_100ms* zu finden sind, ist das Resultat aus der Anwendung der dritten Optimierungsregel. Beide Event-Runnables sind in diesem Fall Receiver und das in Violette markierte periodische Runnable hat die Rolle des Senders. Durch dieses Mapping wird mindestens ein Kontextwechsel vermieden, da das Empfangen der Nachricht im selben Kontext durchgeführt werden kann. Insgesamt liegen somit keine Optimierungsfehler vor und es folglich keine vermeidbaren Kontextwechsel gibt.

Analyse der Ausführungszeit

Als ersten Schritt sollte man betrachten, wie sich die Ausführungszeit im Falle eines automatisierten Mappings zusammenstellt. Wie bereits erwähnt, kann ein generischer Algorithmus keine projektspezifischen Eigenschaften in das Ergebnis einfließen lassen. Aus diesem Grund bedarf es einer zusätzlichen Konfiguration durch den Integrator in Form von Regeldefinitionen für die Rule Engine. Je nach Komplexität kann die Definition einer Regel unterschiedlich lang dauern. Die Regeln in dem vorliegenden Fall kann man als simpel einstufen und auf Basis von Erfahrungswerten wird die Ausführungszeit auf zwei Minuten pro Regel gesetzt. Insgesamt ergibt sich daraus eine Konfigurationszeit von acht Minuten. Die Ausführungszeit des eigentlichen Algorithmus kann man in zwei Phasen unterteilen. Die erste Phase ist die Erstellung des initialen Mappings und die zweite Phase ist die Durchführung der Optimierung. Die Ausführungszeit für die erste Phase lag bei 2,64 Sekunden und für die zweite bei rund 1,85 Sekunden. Zusammengenommen ergibt sich für den Algorithmus eine Ausführungszeit

von 4,49 Sekunden. Insgesamt ergeben Konfigurations- und Ausführungszeit einen zeitlichen Aufwand von acht Minuten und 4,49 Sekunden.

Manuelle und automatische Ausführung im direkten Vergleich

In vorangegangenen Abschnitten wurden die manuelle und die automatische Methode auf ihre funktionalen und nicht-funktionalen Eigenschaften untersucht. In diesem Teil sollen die Resultate der beiden Analysen zum Vergleich gegenüber gestellt werden. Zur Veranschaulichung sind die Resultate in der Tabelle 6.8 zusammengefasst. Das wichtigste Kriterium für die Beurteilung der automatischen Lösung war die funktionale Korrektheit, welche erfüllt worden ist. Die hierbei zu erfüllenden Punkte wurden von der manuellen Lösung vorgegeben, da diese als funktional korrekt angenommen werden kann.

Bei der Untersuchung von Zuordnungsfehlern erzielte die automatische Lösung mit nur zwei Fehlern ein deutlich besseres Ergebnis als die manuelle Lösung mit 78 Fehlern. Auch bei der Erkennung von Optimierungsmöglichkeiten schnitt die automatische Lösung besser ab. Damit konnten bei der automatischen Methode insgesamt fünf unnötige Kontextwechsel gegenüber der manuellen Methode vermieden werden.

Einen deutlichen Unterschied kann man auch bei dem Vergleich von den Ausführungszeiten sehen. Während das Erstellen des Mappings manuell rund 50 Minuten und 24 Sekunden gedauert hat, erreichte man dank der Automatisierung schon nach rund 8 Minuten und 4 Sekunden das passende Ergebnis.

Tabelle 6.8.: Gegenüberstellung der Resultate zwischen der manuellen und automatischen Methode

	Manuell	**Automatisch**
Funktionale Korrektheit	Erfüllt	Erfüllt
Zuordnungsfehler	78	2
Optimierungsfehler	1	0
Kontextwechsel	5	0
Ausführungszeit (abgerundet auf volle Sekunden)	50 min 24 sec	8 min 4 sec

6.3.3. Diskussion

Dieser Abschnitt dient dem Zweck, zu diskutieren, ob das entwickelte Konzept zum RTM die definierten Vorgaben erfüllt und somit eine kontinuierliche Integration realisiert werden kann. Damit es ermöglicht werden kann, erwartet man von dem Algorithmus nach akzeptabler

Ausführungszeit als Resultat ein funktional korrektes Mapping. Die Ergebnisse aus dem vorangegangenen Unterkapitel sollen die Grundlage liefern, ob diese Erwartungen erfüllt worden sind.

Die Ergebnisse haben gezeigt, dass die Erwartungen an die Automatisierung des RTM erfüllt werden konnten. Die Entscheidung für die Verwendung einer Rule-Engine hat sich bezahlt gemacht, indem diese dem Algorithmus die notwendige Flexibilität zur Verfügung stellte, um mit wenig Aufwand projektspezifische Vorgaben zu realisieren. Als Folge davon, verändert sich die Tätigkeit des Integrators, welcher nicht mehr die einzelnen Runnables den Tasks zuordnen muss. Seine Aufgabe besteht nun darin, Regeln für die Rule-Engine zu definieren. Anders als das einfache manuelle Zuordnen, muss dieser Konfigurationsschritt nicht bei jeder Änderung der Applikation durchgeführt werden, sondern nur dann, wenn an das Mapping besondere Anforderungen gegeben werden. Nach der Durchführung der Konfiguration kann das Runnable-Task-Mapping wieder vollautomatisch durchlaufen.

Bewertet man die Leistung der automatischen Lösung gegenüber der manuelle aus der nichtfunktionalen Sicht, erkennt man klar dessen Vorteile. Die strikte Einhaltung der Regeln ist bei einer manuellen Durchführung des Mappings nicht garantiert. Wenn man sich die Ergebnisse ansieht, erkennt man, dass die Integratorlösung mit 78 Zuordnungsfehlern deutlich über den 2 Zuordnungsfehlern der automatischen Lösung liegt. Der Vergleich der Ergebnisse von dem Optimierungsschritt zeigt zwar nur einen Regelverstoß, deutet aber das Potenzial des Algorithmus an. Trotz der großen Toolunterstützung zur Integration von SWCs, sind die Beziehungen zwischen den Runnables für den Integrator nur schwer zu erkennen, weshalb die darauf basierende Optimierungsmöglichkeiten oftmals keine Beachtung finden. Die automatische Durchführung der Optimierung ist somit ein großer Vorteil, den der Algorithmus bietet. Insgesamt zeigt sich also, dass durch die Verwendung des konzipierten Verfahrens eine Verbesserung in den Bereichen von Determinismus und Latenzminimierung erreicht werden kann.

Ein wichtiger Punkt bei der Bewertung des Verfahrens ist auch die Ausführungszeit, denn die Gesamtdurchlaufzeit der Vector-CI-Pipeline wurde auf maximal 30 Minuten festgelegt. Mit nur 4,49 Sekunden Laufzeit fällt die Ausführung des Mappings kaum ins Gewicht, was die Einhaltung dieser Vorgabe problemlos ermöglicht.

Zusammengefasst kann man sagen, dass die gestellten Anforderungen erfüllt worden sind. Folglich ist die Integration und der Einsatz des in dieser Arbeit entwickelten automatischen Runnable-Task-Mapping empfehlenswert.

7. Zusammenfassung und Ausblick

Aufgrund von steigenden Kosten- und Qualitätsdruck bei der Entwicklung von Steuergeräten verspricht sich die Automobilindustrie eine Effizienzsteigerung durch den Einsatz von agilen Methoden. Eine der Methoden, welche in den Entwicklungsprozess eingebettet werden soll, ist Continuous Integration (CI). Dafür hat die Vector Informatik GmbH ein Konzept zur Umsetzung von CI in AUTOSAR Software Projekten für Steuergeräte entwickelt.

Wie es am Anfang dieser Masterarbeit erläutert wurde, bietet die Vector-CI-Pipeline Optimierungspotenzial. Eine Evaluierung der Pipeline hat gezeigt, dass im Ablauf durch die manuelle Konfiguration der RTE ein Flaschenhals entsteht, welcher negative Auswirkungen auf die Entwicklungszyklen haben kann. Deshalb wurde als Ziel dieser Arbeit eine Optimierung der Pipeline gesetzt, welche ein Konzept zur Automatisierung des Schrittes zur Konfiguration, bestehend aus den Teilschritten Port-Mapping und Runnable-Task-Mapping, beinhalten soll.

Um den Stand der Forschung zu ermitteln, wurde eine Literaturrecherche zum Port-Mapping und zum Runnable-Task-Mapping in gleichen oder verwandten Forschungsgebieten durchgeführt. Für den Entwurf des Konzepts für die beiden Teilschritte wurden Ideen aus den betrachteten Publikationen übernommen und auf den AUTOSAR Kontext angepasst.

Das Port-Mapping wurde als ein Zuordnungsproblem klassifiziert und als ein gewichteter bipartiter Graph modelliert. Die Berechnung der Kantengewichte fand auf Basis der Namen der Ports und der Softwarekomponenten statt. Dazu wurde die Namensähnlichkeit zwischen kompatiblen Ports mit Hilfe des Jaccard-Koeffizienten berechnet. Zur Lösung des Zuordnungsproblems wurde anschließend der Ungarische Algorithmus verwendet.

Zur Konzeptentwicklung des automatisierten Runnable-Task-Mappings wurden neben dem Wissen aus der Literaturrecherche, zusätzlich Informationen von Experten und der AUTOSAR Spezifikation hinzugezogen. So besteht der Ablauf des Mappings zu Beginn aus einer Zuordnung von Runnables zu Tasks nach den RTEEvents der Runnables. Anschließend wird eine regelbasierte Optimierung nach [LLP+09] durchgeführt. Zusätzlich besteht die Möglichkeit durch ein BRMS und die entsprechende Formulierung von Regeln Einfluss auf den Ablauf der Zuordnung zu nehmen.

Nach der Implementierung des Konzepts als ein Groovy-Skript für den Cfg Pro wurde eine Evaluierung mit einem aktuell laufenden AUTOSAR-Projekt durchgeführt. Das Port-Mapping erreichte eine Korrektklassifizierungsrate von 94% für das Sender-Receiver-Mapping und 92% für das Client-Server-Mapping. Beim Runnable-Task-Mapping konnte das Mapping des Integrators mit einigen Optimierungen in einer kürzeren Zeit nachgebildet werden.

Insgesamt empfiehlt sich der Einsatz des automatischen Runnable-Task-Mappings, da der Aufwand deutlich minimiert wird und sich ein korrektes Ergebnis in der Zeitbeschränkung der Vector-CI-Pipeline problemlos erreichen lässt. Trotz der hohen Rate bei der korrekten Zuordnung ist der Einsatz des automatisierten Port-Mapping in diesem Fall fraglich, da aufgrund von fehlerhaften Zuordnungen häufige manuelle Eingriffe des Integrators zu erwarten sind.

Ausblick

Ein großes Problem bei der Erstellung des Konzepts waren die nur begrenzt verfügbaren Daten zur Applikation. Da nur die Beschreibung der SWCs zur Verfügung stand, musste das Konzept darauf ausgerichtet werden, was gleichzeitig die möglichen Optionen zur Umsetzung eingeschränkt hat. Deshalb eröffnet sich die Frage, ob ein besseres Konzept zur Automatisierung realistisch ist, wenn man zusätzliche Informationen aus der Applikationsschicht, wie z.B. WCET, Speicherverbrauch und Deadlines, zur Verfügung hat. Der Grund für die Beschränkung der Informationen liegt aber oftmals in der Geheimhaltung des Quellcodes durch den Entwickler der Applikation. Ohne die Bereitschaft zur Kooperation von allen beteiligten Seiten des Projekts ist die notwendige Analyse leider nicht möglich.

A. Anhang

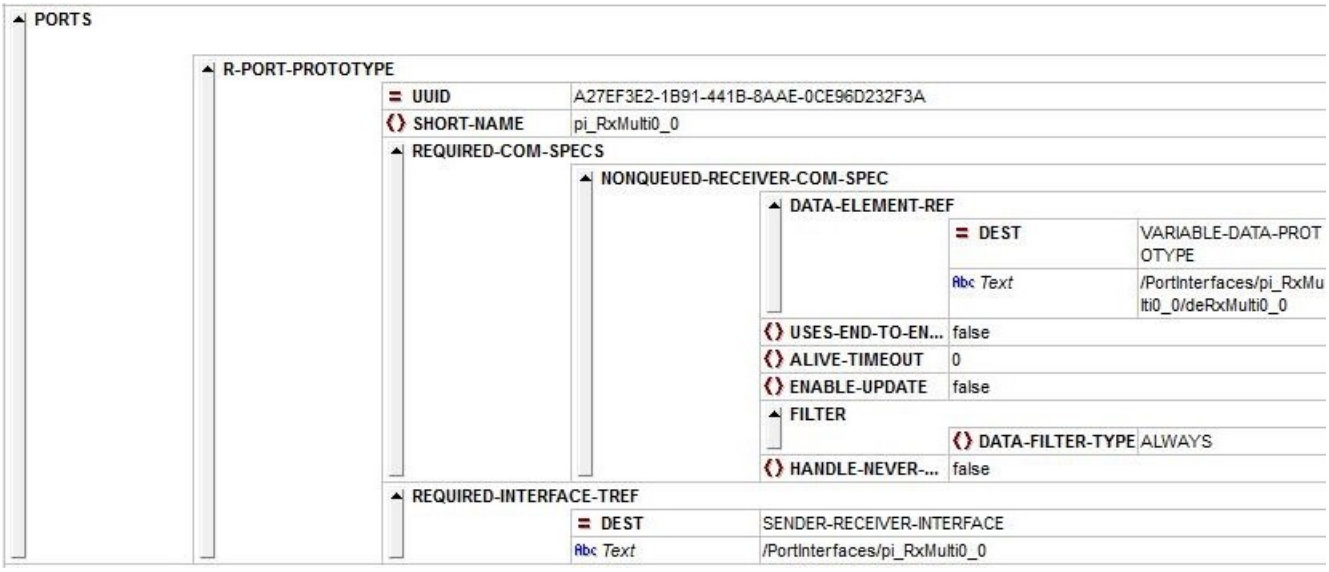


Abbildung A.1.: Beispiel für die Beschreibung eines Ports in einer SWC Beschreibung

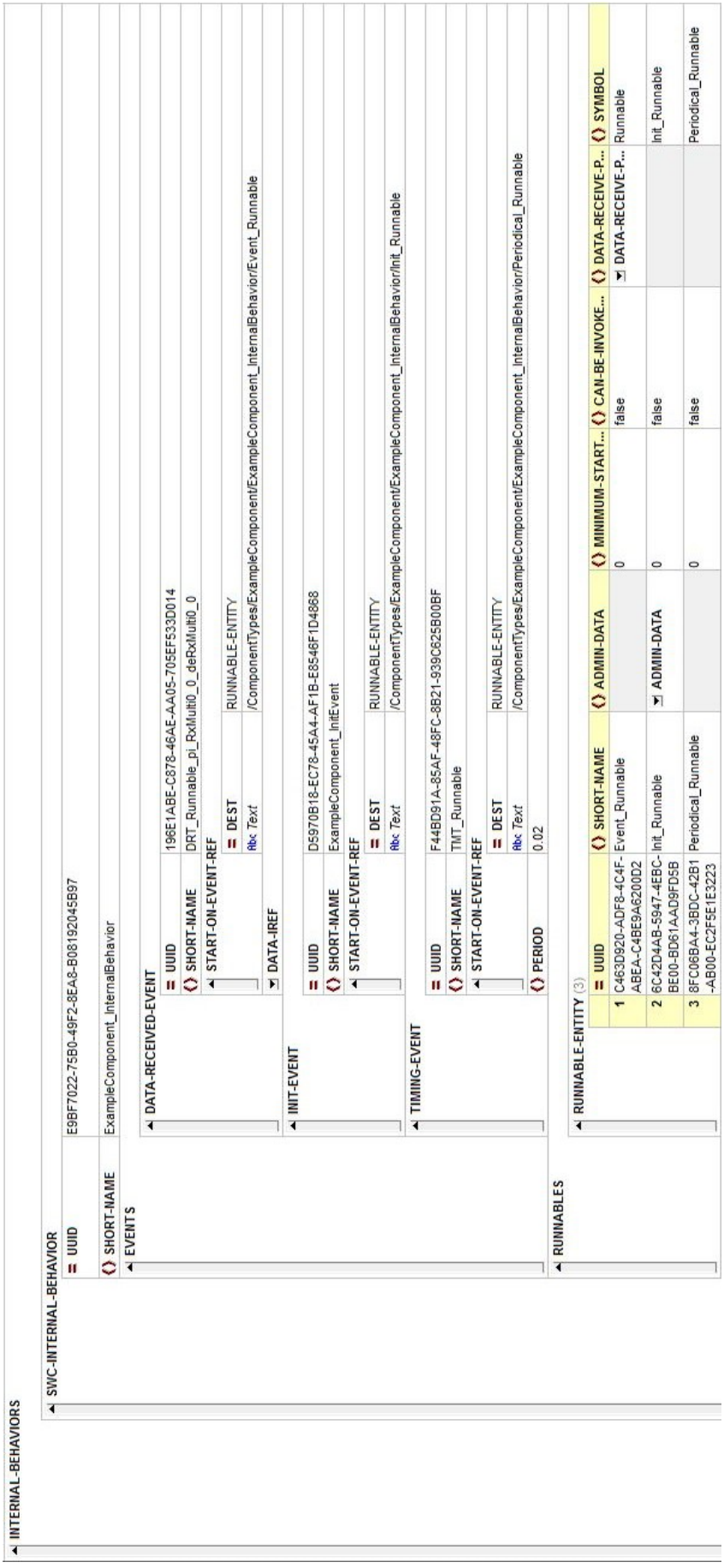


Abbildung A.2.: Beispiel für die Beschreibung von Runnables und RTEEvents in einer SWC Beschreibung

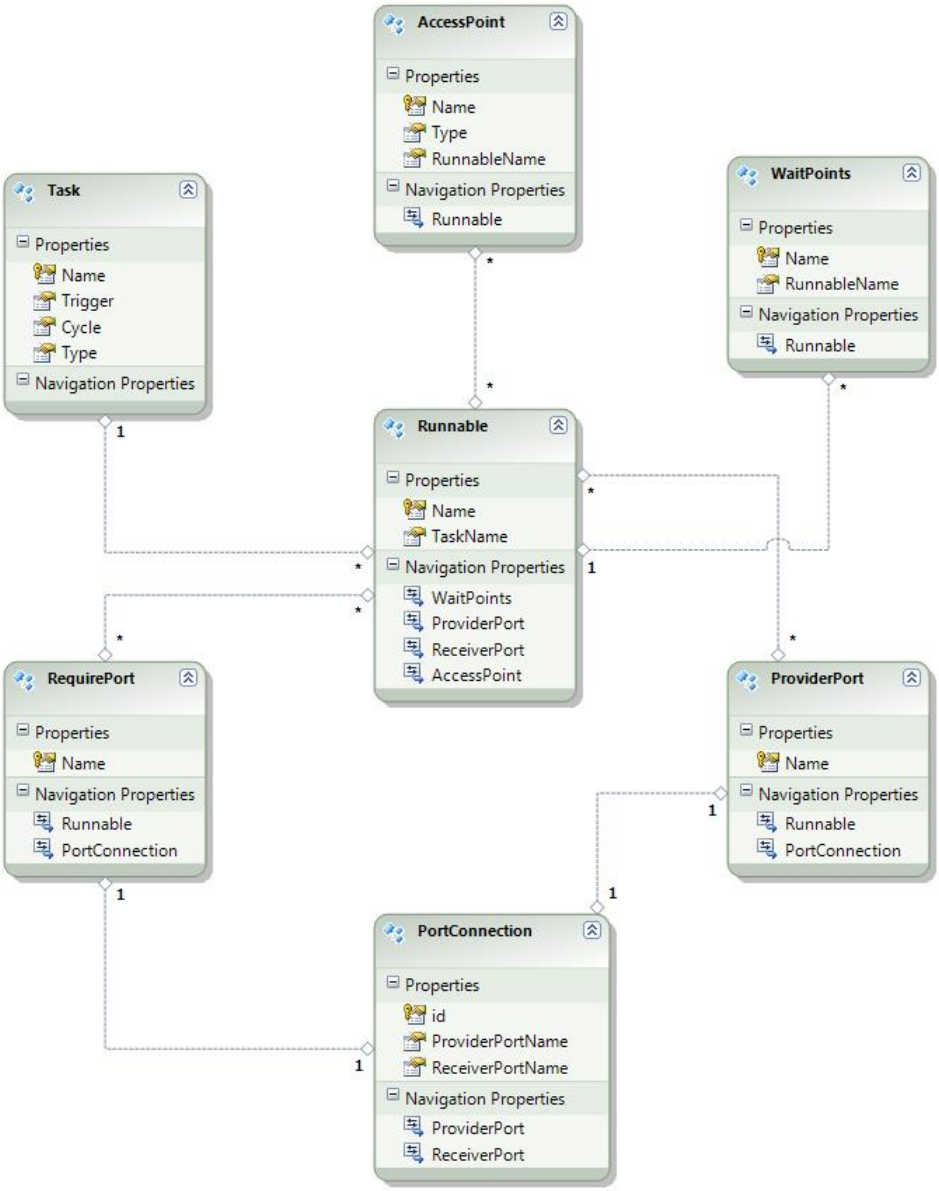


Abbildung A.3.: Datenmodel für die Implementierung des Runnable-Task-Mapping

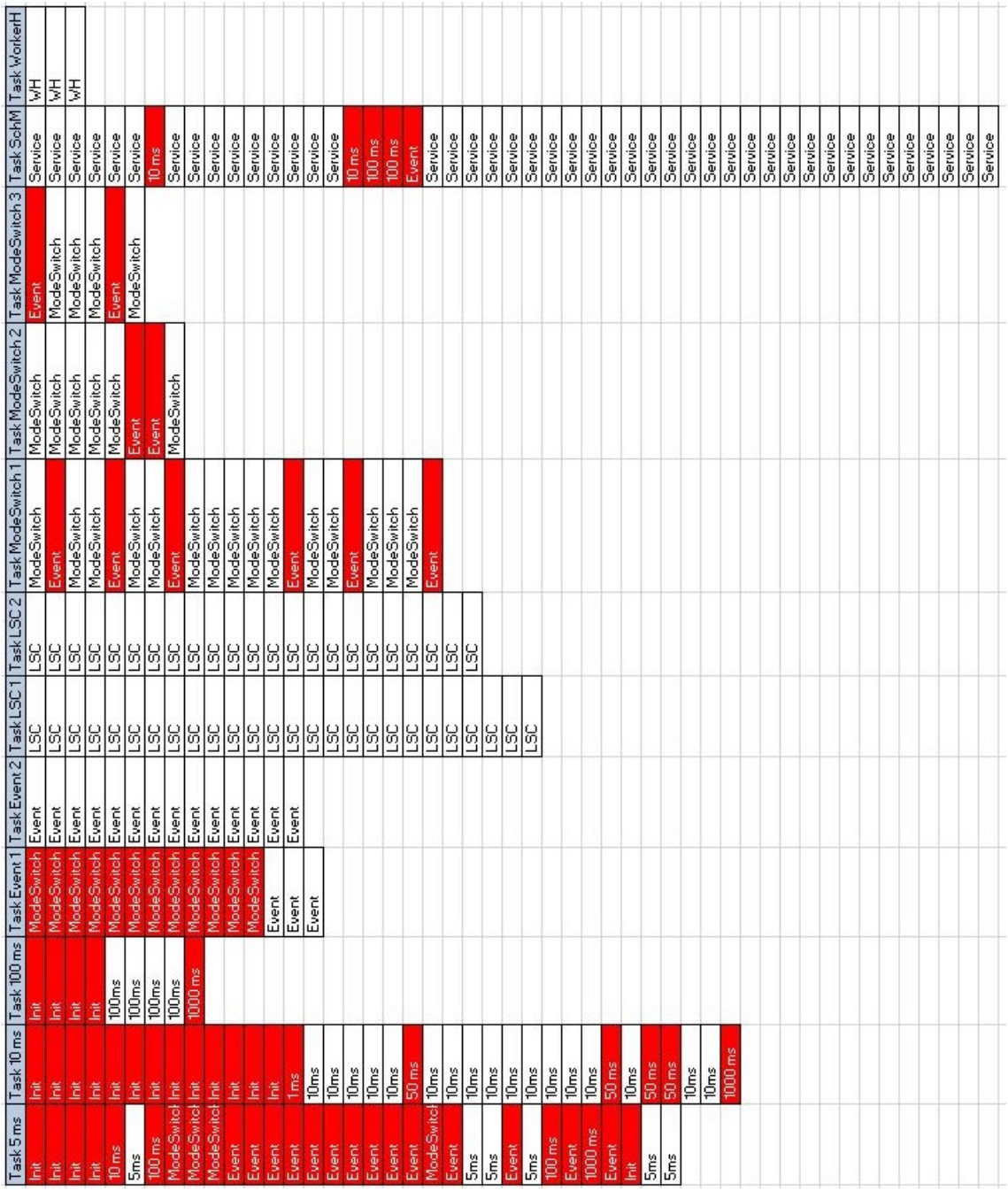


Abbildung A.4.: Runnable-Task-Mapping des Integrators. Die Zuordnungsfehler sind in Rot markiert.

Task 5 ms	Task 10 ms	Task 100 ms	Task Event 1	Task Event 2	Task LSC 1	Task LSC 2	Task ModeSwitch 1	Task ModeSwitch 2	Task ModeSwitch 3	Task SchM	Task WorkerH
Init	Init	Init	ModeSwitch	Event	LSC	LSC	ModeSwitch	ModeSwitch	Event	Service	WH
Init	Init	Init	ModeSwitch	Event	LSC	LSC	Event	ModeSwitch	ModeSwitch	Service	WH
Init	Init	Init	ModeSwitch	Event	LSC	LSC	ModeSwitch	ModeSwitch	ModeSwitch	Service	WH
Init	Init	Init	ModeSwitch	Event	LSC	LSC	ModeSwitch	ModeSwitch	ModeSwitch	Service	
10 ms	Init	100ms	ModeSwitch	Event	LSC	LSC	Event	ModeSwitch	Event	Service	
5ms	Init	100ms	ModeSwitch	Event	LSC	LSC	ModeSwitch	Event	ModeSwitch	Service	
100 ms	Init	100ms	ModeSwitch	Event	LSC	LSC	ModeSwitch	Event		10 ms	
ModeSwitch	Init	100ms	ModeSwitch	Event	LSC	LSC	Event	ModeSwitch		Service	
ModeSwitch	Init	1000 ms	ModeSwitch	Event	LSC	LSC	ModeSwitch			Service	
ModeSwitch	Init		ModeSwitch	Event	LSC	LSC	ModeSwitch			Service	
Event	Init		ModeSwitch	Event	LSC	LSC	ModeSwitch			Service	
Event	Init		ModeSwitch	Event	LSC	LSC	ModeSwitch			Service	
Event	Init		Event	Event	LSC	LSC	ModeSwitch			Service	
Event	1 ms		Event	Event	LSC	LSC	Event			Service	
Event	10ms		Event		LSC	LSC	ModeSwitch			Service	
Event	10ms				LSC	LSC	ModeSwitch			Service	
Event	10ms				LSC	LSC	Event			10 ms	
Event	10ms				LSC	LSC	ModeSwitch			100 ms	
Event	10ms				LSC	LSC	ModeSwitch			100 ms	
Event	50 ms				LSC	LSC	ModeSwitch			Event	
ModeSwitch	10ms				LSC	LSC	Event			Service	
Event	10ms				LSC	LSC				Service	
5ms	10ms				LSC	LSC				Service	
5ms	10ms				LSC					Service	
Event	10ms				LSC					Service	
5ms	10ms				LSC					Service	
100 ms	10ms									Service	
Event	10ms									Service	
1000 ms	10ms									Service	
Event	50 ms									Service	
Init	10ms									Service	
5ms	50 ms									Service	
5ms	50 ms									Service	
	10ms									Service	
	10ms									Service	
	1000 ms									Service	
										Service	
										Service	
										Service	
										Service	
										Service	
										Service	
										Service	
										Service	
										Service	
Legende										Service	
Regel 1										Service	
Regel 2										Service	
Regel 3										Service	

Abbildung A.5.: Runnable-Task-Mapping des Integrators. Die Optimierungsfehler sind farblich markiert.

Task_Init	Task_1ms	Task_5ms	Task_10ms	Task_50ms	Task_100ms	Task_1000ms	Task_Event_1	Task_Event_2	Task_LSC_1	Task_LSC_2	Task_ModeSwitch_1	Task_ModeSwitch_2	Task_SchM	Task_WorkerH
Init	1ms	5ms	10ms	50ms	100ms	1000ms	Event	Event	LSC	LSC	ModeSwitch	ModeSwitch	Service	special
Init		5ms	10ms	50ms	100ms	1000ms	Event	Event	LSC	LSC	ModeSwitch	ModeSwitch	Service	special
Init		5ms	10ms	50ms	100ms	1000ms	Event	Event	LSC	LSC	ModeSwitch	ModeSwitch	Service	special
Init		5ms	10ms	50ms	100ms		Event	Event	LSC	LSC	ModeSwitch	ModeSwitch	Service	
Init		5ms	10ms		100ms		Event	Event	LSC	LSC	ModeSwitch	ModeSwitch	Service	
Init		5ms	10ms		100ms		Event	special	LSC	LSC	ModeSwitch	ModeSwitch	Service	
Init			10ms		100ms		Event	special	LSC	LSC	ModeSwitch	ModeSwitch	Service	
Init			10ms		Event		Event	special	LSC	LSC	ModeSwitch	ModeSwitch	Service	
Init			10ms		Event		Event		LSC	LSC	ModeSwitch	ModeSwitch	Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC		ModeSwitch		Service	
Init			10ms				Event		LSC		ModeSwitch		Service	
Init			10ms				Event		LSC		ModeSwitch		Service	
Init							Event		LSC		ModeSwitch		Service	
Init							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
													Service	
													Service	
													Service	
													Service	
													Service	
													Service	
													Service	
													Service	
													Service	
													Service	
													Service	
													Service	

Abbildung A.6.: Automatische Lösung des Runnable-Task-Mapping. Zuordnungsfehler sind rot markiert.

Task_Init	Task_1ms	Task_5ms	Task_10ms	Task_50ms	Task_100ms	Task_1000ms	Task_Event_1	Task_Event_2	Task_LSC_1	Task_LSC_2	Task_ModeSwitch_1	Task_ModeSwitch_2	Task_SchM	Task_WorkerH
Init	1ms	5ms	10ms	50ms	100ms	1000ms	Event	Event	LSC	LSC	ModeSwitch	ModeSwitch	Service	special
Init		5ms	10ms	50ms	100ms	1000ms	Event	Event	LSC	LSC	ModeSwitch	ModeSwitch	Service	special
Init		5ms	10ms	50ms	100ms	1000ms	Event	Event	LSC	LSC	ModeSwitch	ModeSwitch	Service	special
Init		5ms	10ms	50ms	100ms		Event	Event	LSC	LSC	ModeSwitch	ModeSwitch	Service	
Init		5ms	10ms		100ms		Event	Event	LSC	LSC	ModeSwitch	ModeSwitch	Service	
Init		5ms	10ms		100ms		Event	special	LSC	LSC	ModeSwitch	ModeSwitch	Service	
Init			10ms		Event		Event	special	LSC	LSC	ModeSwitch	ModeSwitch	Service	
Init			10ms		Event		Event	special	LSC	LSC	ModeSwitch	ModeSwitch	Service	
Init			10ms				Event		LSC	LSC	ModeSwitch	ModeSwitch	Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC	LSC	ModeSwitch		Service	
Init			10ms				Event		LSC		ModeSwitch		Service	
Init			10ms				Event		LSC		ModeSwitch		Service	
Init			10ms				Event		LSC		ModeSwitch		Service	
Init							Event		LSC		ModeSwitch		Service	
Init							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
							Event		LSC		ModeSwitch		Service	
													Service	
													Service	
													Service	
													Service	
													Service	
													Service	
													Service	
													Service	
Legende													Service	
Regel 1													Service	
Regel 2													Service	
Regel 3													Service	

Abbildung A.7.: Automatische Lösung des Runnable-Task-Mapping. Die Optimierungsfehler sind farblich markiert.

Literaturverzeichnis

[AHS08] P. Achananuparp, X. Hu, X. Shen. „The Evaluation of Sentence Similarity Measures." In: *DaWaK*. Hrsg. von I.-Y. Song, J. Eder, T. M. Nguyen. Bd. 5182. Lecture Notes in Computer Science. Springer, 5. Sep. 2008, S. 305–316. ISBN: 978-3-540-85835-5. DOI: 10.1007/978-3-540-85836-2_29. URL: http://dblp.uni-trier.de/db/conf/dawak/dawak2008.html#AchananuparpHS08 (zitiert auf S. 26, 31, 32).

[AUT03] AUTOSAR. *AUTOSAR Development Partnership*. 2003. URL: https://www.autosar.org/ (besucht am 11. 05. 2017) (zitiert auf S. 7, 15, 17).

[AUT13a] AUTOSAR. 2013. URL: http://www.autosar.org/fileadmin/files/standards/classic/4-2/main/auxiliary/AUTOSAR_EXP_VFB.pdf (zitiert auf S. 18, 19).

[AUT13b] AUTOSAR. 2013. URL: http://www.autosar.org/fileadmin/files/standards/classic/4-2/software-architecture/rte/standard/AUTOSAR_SWS_RTE.pdf (zitiert auf S. 19, 20).

[BDM12] R. Burkard, M. Dell'Amico, S. Martello. *Assignment Problems, Revised Reprint:* Society for Industrial und Applied Mathematics, 2012. ISBN: 9781611972221. URL: https://books.google.de/books?id=FH6%5C_Bb82I%5C_UC (zitiert auf S. 33).

[Com17] J. Community. *Drools*. 2017. URL: https://www.drools.org/ (besucht am 16. 05. 2017) (zitiert auf S. 23, 48).

[CRF03] W. W. Cohen, P. Ravikumar, S. E. Fienberg. „A Comparison of String Metrics for Matching Names and Records". In: *KDD Workshop on Data Cleaning and Object Consolidation*. 2003. URL: https://www.cs.cmu.edu/afs/cs/Web/People/wcohen/postscript/kdd-2003-match-ws.pdf (zitiert auf S. 26, 31, 46).

[FF06] M. Fowler, M. Foemmel. *Continuous integration,* 2006. URL: http://www.martinfowler.com/articles/continuousIntegration.html (zitiert auf S. 7, 9).

[FNG+09] A. Ferrari, M. D. Natale, G. Gentile, G. Reggiani, P. Gai. „Time and memory tradeoffs in the implementation of AUTOSAR components". In: *2009 Design, Automation Test in Europe Conference Exhibition*. Apr. 2009, S. 864–869. DOI: 10.1109/DATE.2009.5090783 (zitiert auf S. 26, 27).

[Gal08] A. Gal. „Interpreting similarity measures: Bridging the gap between schema matching and data integration". In: *2008 IEEE 24th International Conference on Data Engineering Workshop*. Apr. 2008, S. 278–285. DOI: 10.1109/ICDEW.2008.4498332 (zitiert auf S. 25).

[Gmb16a] V. I. GmbH. „Agenda Distributed Agile Development“. 2016 (zitiert auf S. 9).

[Gmb16b] V. I. GmbH. „AUTOSAR in Practice“. 2016 (zitiert auf S. 31).

[Gmb17a] V. I. GmbH. *DaVinci Configurator Pro*. 2017. URL: https://vector.com/vi_davinci_configurator_pro_de.html (besucht am 15. 05. 2017) (zitiert auf S. 22).

[Gmb17b] V. I. GmbH. *DaVinci Developer*. 2017. URL: https://vector.com/vi_davinci_developer_de.html (besucht am 15. 05. 2017) (zitiert auf S. 22).

[Gro17] H. Group. *H2 Database Engine*. 2017. URL: http://www.h2database.com/html/main.html (besucht am 17. 05. 2017) (zitiert auf S. 23).

[KF09] O. Kindel, M. Friedrich. *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. Heidelberg: dpunkt, 2009. ISBN: 978-3-898-64563-8 (zitiert auf S. 15, 17, 19–22).

[KYM+09] F. Kluge, C. Yu, J. Mische, S. Uhrig, T. Ungerer. „Implementing AUTOSAR Scheduling and Resource Management on an Embedded SMT Processor“. In: *Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '09. Nice, France: ACM, 2009, S. 33–42. ISBN: 978-1-60558-696-0. URL: http://dl.acm.org/citation.cfm?id=1543820.1543828 (zitiert auf S. 20).

[Lev66] V. I. Levenshtein. „Binary codes capable of correcting deletions, insertions and reversals.“ In: *Soviet Physics Doklady* 10.8 (1966). Doklady Akademii Nauk SSSR, V163 No4 845-848 1965, S. 707–710 (zitiert auf S. 31).

[LLP+09] R. Long, H. Li, W. Peng, Y. Zhang, M. Zhao. „An Approach to Optimize Intra-ECU Communication Based on Mapping of AUTOSAR Runnable Entities“. In: *2009 International Conference on Embedded Software and Systems*. Mai 2009, S. 138–143. DOI: 10.1109/ICESS.2009.63 (zitiert auf S. 27, 36–41, 55, 61).

[MGR02] S. Melnik, H. Garcia-Molina, E. Rahm. „Similarity flooding: a versatile graph matching algorithm and its application to schema matching“. In: *Proceedings 18th International Conference on Data Engineering*. 2002, S. 117–128. DOI: 10.1109/ICDE.2002.994702 (zitiert auf S. 25, 32).

[MMAW99] N. McKeown, A. Mekkittikul, V. Anantharam, J. Walrand. „Achieving 100% throughput in an input-queued switch“. In: *IEEE Transactions on Communications* 47.8 (Aug. 1999), S. 1260–1267. ISSN: 0090-6778. DOI: 10.1109/26.780463 (zitiert auf S. 25, 29, 33).

[Nau09] N. Naumann. „Autosar runtime environment and virtual function bus“. In: *Hasso-Plattner-Institut, Tech. Rep* (2009), S. 38 (zitiert auf S. 17).

[PBKS07] A. Pretschner, M. Broy, I. H. Kruger, T. Stauner. „Software engineering for automotive systems: A roadmap“. In: *2007 Future of Software Engineering*. IEEE Computer Society. 2007, S. 55–71 (zitiert auf S. 7).

[Pow11] D. M. W. Powers. „Evaluation: From precision, recall and f-measure to ROC, informedness, markedness & correlation“. In: *Journal of Machine Learning Technologies* 2.1 (2011), S. 37–63 (zitiert auf S. 50).

[Pro17] T. A. G. Project. *Apache Groovy*. 2017. URL: http://groovy-lang.org/ (besucht am 16. 05. 2017) (zitiert auf S. 23).

[Ros03] R. G. Ross. *Principles of the Business Rule Approach*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0201788934 (zitiert auf S. 42).

[SAA10] A. Shaout, M. Arora, S. Awad. „Automotive software development and management“. In: *Computer Engineering Conference (ICENCO), 2010 International*. IEEE. 2010, S. 9–15 (zitiert auf S. 7).

[SDD+04] J. M. Stecklein, J. Dabney, B. Dick, B. Haskins, R. Lovell, G. Moroney. „Error cost escalation through the project life cycle“. In: (2004) (zitiert auf S. 7, 13).

[SDL09] J. Srinivasan, R. Dobrin, K. Lundqvist. „'State of the Art' in Using Agile Methods for Embedded Systems Development“. In: *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2009, Seattle, Washington, USA, July 20-24, 2009. Volume 2*. 2009, S. 522–527. DOI: 10.1109/COMPSAC.2009.186. URL: http://dx.doi.org/10.1109/COMPSAC.2009.186 (zitiert auf S. 7).

[TLD+14] R. Y. Takahira, L. R. Laraia, F. A. Dias, S. Y. Abraham, P. T. Nascimento, A. S. Camargo. „Scrum and Embedded Software development for the automotive industry“. In: *Management of Engineering & Technology (PICMET), 2014 Portland International Conference on*. IEEE. 2014, S. 2664–2672 (zitiert auf S. 7).

[Zee12] A. Zeeb. „Plug and Play Solution for AUTOSAR Software Components“. In: *ATZ elektronik, issue 1-2/2012*. 2012 (zitiert auf S. 10).

[ZG11] M. Zhang, Z. Gu. „Optimization issues in mapping AUTOSAR components to distributed multithreaded implementations“. In: *2011 22nd IEEE International Symposium on Rapid System Prototyping*. Mai 2011, S. 23–29. DOI: 10.1109/RSP.2011.5929971 (zitiert auf S. 26, 27, 36, 37).

[ZN12] H. Zeng, M. D. Natale. „Efficient implementation of AUTOSAR components with minimal memory usage“. In: *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. Juni 2012, S. 130–137. DOI: 10.1109/SIES.2012.6356578 (zitiert auf S. 27, 36, 37).

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AI AutomationInterface. 45

Apache Subversion SVN. 8

AUTOSAR XML ARXML. 17

BRMS Business-Rule-Management-System. 42

BSW Basissoftware. 8

Cfg Pro DaVinci Configurator Pro. 8

CI Continuous Integration. 7

ECU Electronic Control Unit. 27

MCU Microcontroller Unit. 49

OEM Original Equipment Manufacturer. 7

PM Port-Mapping. 8

RTE Runtime-Environment. 8

RTM Runnable-Task-Mapping. 8

SWC Softwarekomponente. 8

Vector Vector Informatik GmbH. 8

Alle URLs wurden zuletzt am 25. 05. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift