Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Edgar Toll

Reimplementierung eines RCS-Clients zur Beschleunigung von Variantentests

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Edgar Toll

Reimplementierung eines RCS-Clients zur Beschleunigung von Variantentests

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Thomas Lehmann
Zweitgutachter: Prof. Dr. Martin Becke

Eingereicht am: 16. Juni 2017

Edgar Toll

Thema der Arbeit

Reimplementierung eines RCS-Clients zur Beschleunigung von Variantentests

Stichworte

Roboter, RCS Module, RRS Protokoll, RRS Interface, Variantentests, Protokollanalyse, …

Kurzzusammenfassung

In der Automobilindustrie herrscht heutzutage ein hoher Automatisierungsgrad. Die Anforderungen an die Roboter und die genutzten Roboterprogramme steigen immer weiter. In virtuellen Umgebungen werden daher Roboterprogramme simuliert und optimiert, um diesen Anforderungen nach zu kommen. So spielen beispielsweise die Einhaltung oder Verkürzung der Taktzeit eine Rolle, wie auch verschiedene, die Qualität beeinflussende Faktoren, welche in der Simulation geprüft werden. Ein Programmierer muss die ihm gegebenen Varianten testen, gegeneinander abwägen und gegebenenfalls einen geeigneten Mittelweg finden. Dabei benötigt dieser Prozess einen großen Teil der Zeit in der Erstellung und Optimierung von Roboterprogrammen.

Process Simulate ist eine roboterherstellerunabhängige Simulationssoftware von Siemens PLM. Zur möglichst realitätsnahen Simulation der Roboter in Process Simulate werden sogenannte RCS (Robot Controller Simulation) Module genutzt, die im Idealfall die Roboter Bewegung exakt abbilden. Das Interface der RCS Module ist roboterherstellerunabhängig über das RRS Protokoll spezifiziert.

Im Rahmen dieser Bachelorarbeit wird die aktuelle Verwendung des RRS Protokoll auf Möglichkeiten der Beschleunigung des Simulationsablaufes analysiert. Auf der Analyse basierend wird beschrieben, wie ein RCS Client implementiert wird, der die Durchführung von Variantentests gegenüber Process Simulate beschleunigen soll.

Edgar Toll

Title of the paper

Reimplementation of a RCS Client in order to speed up variation tests

Keywords

Robot, RCS Module, RRS Protocol, RRS Interface, Variation-Tests, Protocol Analysis, ...

Abstract

Nowadays the automotive industry has an increasing demand for more and more requirements on robot programs. Requirements such as cycle time or other factors that change the quality of the robot procedure are pre-checked within simulations. The process of probing these variants is currently taking a great amount of time within the creation or optimization of robot programs.

Process Simulate is a universal, robot-vendor independent software created by Siemens PLM. In order to simulate robots independently Process Simulate uses RCS (Robot Controller Simulation) modules. These RCS modules simulate the ideally exact robot movement. In order to specify the common interface for these RCS modules RRS was specified.

This thesis describes how to speed up simulations based on RCS Modules by re-implementing RCS clients. Based on the analysis an own RCS client is implemented that will run variation tests faster than Process Simulate.

Inhaltsverzeichnis

Listings

1 Einleitung

Durch den hohen Automatisierungsgrad in der Automobilindustrie kommen viele Roboter zum Einsatz. Eine zeitsparende Möglichkeit, Roboter zu programmieren, bietet die Off-Line Programmierung (OLP). Eine Methode der OLP ist die Computer-Aided Design (CAD) gestützte Off-Line Programmierung. Dafür existieren Computer-Aided Robotics (CAR) Tools wie *Process Simulate* von Siemens. Hierbei hat ein Programmierer ein CAD-Abbild einer realen Anlage und erstellt oder optimiert darauf basierend Roboterbahnen, die später als ein Roboterprogramm exportiert werden.

Die Anforderungen an die Roboterprogramme steigen immer weiter. So spielen beispielsweise die Taktzeit oder die Qualität des Operationsergebnisses, wie zum Beispiel einer Lackierung, eine Rolle. Der Programmierer muss sich dabei manuell an gewünschte Kriterien annähern und basierend auf seiner Berufserfahrung, zwischen diesen abwägen. Da mögliche Probleme mit einer dieser Varianten teilweise erst in der Simulation zu erkennen sind, müssen diese Varianten ausprobiert werden. Dafür werden Arbeitsstunden in die Umsetzung von Varianten gesteckt, die sich später nicht als praktikabel herausstellen.

In der Routennavigation passiert etwas ganz ähnliches, wie es sich der Roboterprogrammierer für seine tägliche Arbeit wünscht. Um bei der Auswahl der geeignetsten Route zum gewünschten Zielort zu unterstützen, simuliert das Navigationsgerät unterschiedliche Varianten der Routenführung und schlägt dem Nutzer eine Teilmenge geeigneter Varianten vor. In der Routennavigation ermittelt das Navigationsgerät beispielsweise die kürzeste Zeit zum Erreichen des Zielortes, die kürzeste Stecke um Ressourcen zu sparen und weitere Varianten. Die für den Nutzer am besten zutreffende Variante muss nur noch ausgewählt werden.

Dabei können eine Vielzahl von Optimierungszielen in der Robotik in Frage kommen. Geringe Taktzeit, gleichmäßige Abfahrgeschwindigkeit für bessere Lackierqualität oder auch weniger abrupte Fahrbewegungen für möglichst geringen Energiebedarf und Verlängerung der Lebenszeit der Roboter sind lediglich ein kleiner Teil des Möglichen.

Aktuell ist der Roboterprogrammierer in der Benutzung von *Process Simulate* darauf beschränkt, seine Anforderungen manuell umzusetzen und mit *Process Simulate* eine Simulation durchzuführen. Entspricht das Ergebnis nicht den gewünschten Kriterien, muss der Program-

mierer die Roboterbahnen iterativ verbessern. Das Optimieren einer Roboterbahn auf ein bestimmtes Kriterium kann dabei automatisch passieren, wie Olaf Bürger zur Glättung der Abfahrgeschwindigkeit untersucht [6].

Da die Simulationen jedoch eine gewisse Zeit benötigen und während dieser Zeit nicht weiter gearbeitet werden kann, werden die Simulationen aktuell nur am Ende eines Arbeitsschrittes durchgeführt. Die Routennavigation im Auto ist jedoch in der Lage dynamisch auf Änderungen einzugehen. Deaktiviert man beispielsweise Autobahnabschnitte, werden im Hintergrund bereits dazu passende Routen berechnet und vorgeschlagen, ohne dabei eine blockierende Hauptaktion darzustellen. So könnte auch die Arbeitsweise in der Simulation von Roboterpfaden optimiert werden und bereits während der Anpassung von Pfaden validiert werden oder Vorschläge unterbreitet werden, ohne bis zur Fertigstellung des Arbeitsschrittes zu warten. So kann der Roboterprogrammierer schnell erkennen, welche Aktionen zu Problemen führen und frühzeitig darauf reagieren, anstatt später aufwendig die Problemursache zu suchen.

Zur möglichst realitätsnahen Simulation benutzt *Process Simulate* sogenannte Roboter Controller Simulation (RCS) Module. Die RCS Module umfassen die Simulation der Roboter Kinematik. *Process Simulate* ist durch seine Architektur auf eine Simulation zur Zeit beschränkt. Aufgrund dessen müssen Varianten rein sequenziell in *Process Simulate* durchgeführt werden, wie es auch Olaf Bürger in seiner Arbeit [6] tut. Außerdem ist der Ressourcenbedarf einer *Process Simulate* Instanz durch die grafische Darstellung der CAD Daten vergleichsweise hoch.

Process Simulate erhält alle für die Durchführung der Bahnsimulation relevanten Daten durch die Interaktion mit den RCS Modulen. Daher lässt sich eine Bahnsimulation vollständig über die Nachbildung der Interaktion erreichen. Zur Kommunikation zwischen CAR Tool und RCS Modul wird das RRS (Realistic Robot Simulation) Protokoll [1] verwendet. Durch die Reimplementierung der RRS Interaktion in ein eigenes CAR Tool sollte sich die Durchführung von Varianten beschleunigen lassen.

1.1 Zielsetzung

In dieser Bachelorarbeit soll die aktuelle Verwendung von RRS in *Process Simulate* auf Möglichkeiten der Beschleunigung der Simulationsdauer untersucht werden. Darauf aufbauend wird ein Durchstich eines eigenen CAR Tools bzw. eines eigenen RCS Clients implementiert, der einzelne Varianten im Idealfall schneller als *Process Simulate* durchführen kann. Ausblickend soll die Reimplementierung nebenläufig eingesetzt werden können, um so zusätzliche Beschleunigung zu erreichen, die aktuell mit der nur rein sequenziell möglichen Simulation von Varianten in *Process Simulate* nicht erreicht werden kann.

Das eigene CAR Tool soll die Machbarkeit des Durchführens von mehreren, unterschiedlichen Varianten demonstrieren. Um möglichst realistische Simulationen durchzuführen wird eine Roboterbahn aus einem produktiv eingesetzten Roboterprogram als Basis für die Varianten verwendet. Zur Demonstration mehrerer, sich unterscheidender Varianten werden aus der Roboterbahn durch rudimentäre Anpassung eines Kriteriums unterschiedliche Varianten erzeugt.

Die Varianten sollen über den eigenen RCS Client unter Benutzung der RCS Module simuliert werden. Zur Simulation soll dasselbe RCS Modul verwendet werden, wie es auch *Process Simulate* zur Simulation des Roboterprogramms nutzt.

Der RCS Client soll dabei als eigenständige Komponente fungieren. Er soll nicht an Process Simulate zur Eingabe oder an ein spezielles RCS Modul eines bestimmten Herstellers gebunden sein.

Die Roboterprogramme aus dem produktiven Einsatz und das CAR Tool *Process Simulate* wurden von der Firma ICARUS Consulting GmbH zur Verfügung gestellt.

1.2 Struktur dieser Arbeit

In Kapitel 2 „Grundlagen“ werden die wichtigsten Grundlagen aus der Robotik und die grobe funktionsweise des RRS Protokolls erläutert.

Kapitel 3 „Analyse“ dient der Analyse der aktuell verwendeten RRS Interaktion und die Schritte dorthin, um die aktuelle RRS Interaktion analysieren zu können.

Kapitel 4 „Reimplementierung“ beschreibt die Reimplementierung eines Durchstichs eines RCS Clients und dessen Bewertung auf Erfüllung des Beschleunigungszieles.

Kapitel 5 „Fazit“ zieht ein Fazit zu dieser Arbeit und bietet einen Ausblick auf sich daraus ergebende Möglichkeiten.

2 Grundlagen

Da diese Arbeit im Rahmen der Bachelorprüfung Informatik erfolgt, wird grundsätzliches Wissen aus der Informatik vorausgesetzt und nicht weiter erklärt. Tiefergehendes Wissen wird, wenn nötig, zum jeweiligen Zeitpunkt kurz erläutert.

Um eine Implementierung eines Tools im Bereich der Robotik durchzuführen, müssen die grundlegenden Begrifflichkeiten geklärt werden. Das Wissen dazu stammt sowohl aus der RRS Spezifikation [1], als auch von der Firma ICARUS Consulting GmbH. Dazu werden die Grundlagen von Roboterpfaden in Abschnitt 2.1 „Roboterpfad“ erläutert. Darauf folgend wird in Abschnitt 2.2 „RRS Protokoll“ das grundlegende Prinzip des RRS Protokolls [1], soweit nötig, kurz erklärt.

2.1 Roboterpfad

Bei der Programmierung von Robotern werden die Fahrbewegungen immer in Roboterpfaden definiert. Ein Roboterpfad ist dabei eine Abfolge von Locations (Punkte mit Ausrichtung im Raum), die der Roboter nacheinander abfährt.

Dabei werden bestimmte Parameter gesetzt, um diese Bewegung genauer zu spezifieren. Diese werden teilweise für den Roboter, teilweise zu Beginn eines Pfades und teilweise zu einem bestimmten Pfadpunkt definiert. So kann die Bewegung beispielsweise als lineare Bewegung (LIN) oder als Point to Point (PTP) bewegung definiert werden. Bei einer linearen Bewegung wird das Roboter Werkzeug, das sogenannte TCPF (Tool Center Point Frame), linear zwischen den Pfadpunkten bewegt, um etwa einen Lackiervorgang durchzuführen. Die Fahrbewegung kann aber auch mit einer PTP Bewegung alle Roboterachsen auf ihre gewünschte, neue Position bringen, wobei sich das TCPF aber nicht linear bewegen wird. Dies wird unter Anderem für Anfahrpositionen genutzt, um das TCPF möglichst schnell in die Nähe der Karosse zu bringen.

Weitere Parameter können Maximalgeschwindigkeiten oder Überschleif (engl. flyby) sein. In Abbildung 2.1 ist bei P3 und P4 ein Überschleifradius eingezeichnet. Innerhalb des Überschleifradiuses kann der Roboter seine Bahn frei bestimmen und muss so nicht auf eine Geschwindigkeit von 0 reduzieren, um den Punkt exakt zu erreichen. Die Überschleifpunkte

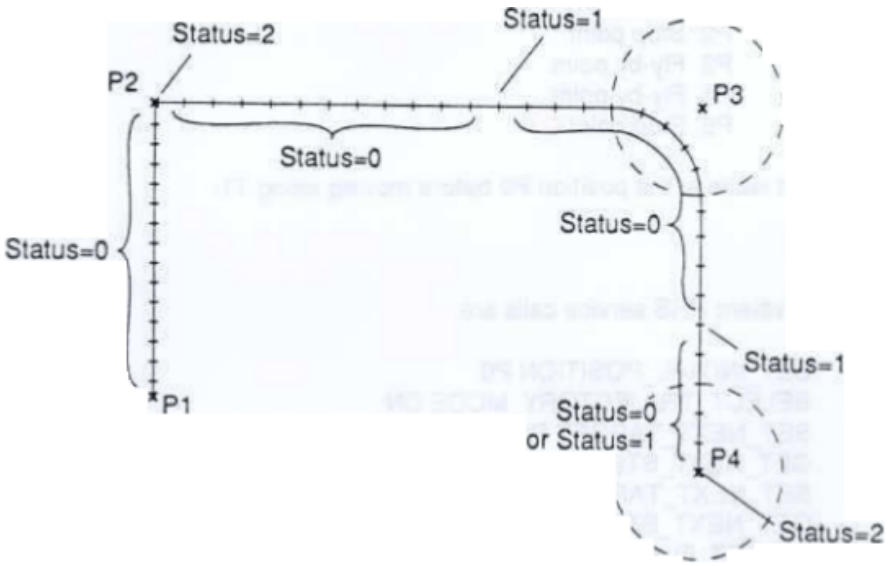


Abbildung 2.1: Ein Roboterpfad über die Locations P1 bis P4 [1]

werden unter anderem genutzt, um einen gleichmäßigen Bahnverlauf zu erreichen, wie er beispielsweise beim Lackieren benötigt wird.

2.2 RRS Protokoll

Um mit den RCS Modulen unterschiedlicher Hersteller zu interagieren, wird ein einheitliches Interface benötigt. Ein solches Interface definiert die RRS Spezifikation [1].

Das RRS Protokoll wurde zudem erweiterbar entworfen, was die Kompatibilität zwischen den RRS Versionen der ersten Generation sicherstellt. Die Spezifikation der zweiten Generation (RRS-2) [3] wird aktuell von roboterherstellerübergreifenden CAR Tools wie *Process Simulate* weder unterstützt, noch ist dies in Zukunft geplant. Daher wird in dieser Arbeit mit den RCS Modulen der ersten Generation gearbeitet.

Um die Dauer einer Simulation, die das RRS Protokoll verwendet, senken zu können, müssen die dafür verwendeten Prinzipien verstanden werden. Das RRS Protokoll ist wie ein Remote Procedure Call (RPC) Protokoll entworfen, wird jedoch in der Spezifikation nie so betitelt. In Unterabschnitt 2.2.1 „RPC“ ist die Herangehensweise eines RPC Protokolls und die daraus resultierenden Möglichkeiten der Beschleunigung dessen beschrieben. In Unterabschnitt 2.2.2 „Motion Konzept“ wird das Konzept hinter der Simulation von Roboterpfaden mit RCS Modulen erläutert.

2.2.1 RPC

Bei einem RPC Protokoll werden entfernte Prozeduren aufgerufen. Dies ist ein typischer Client-Server Ansatz von verteilten Systemen auf Request Response Basis. Die aufgerufenen Prozeduren können dabei auf dem selben PC, aber auch hunderte von Kilometern entfernt ausgeführt werden.

In der grundsätzlichen Form wird RRS jedoch nur als normaler C-Call in einer Dynamic Link Library (DLL) aufgerufen. Eine in der RRS Spezifikation vorgeschlagene Erweiterung bietet dabei den Aufruf der jeweiligen entfernten Prozedur über einen Shared Memory. Wie im Laufe dieser Arbeit klar wird, hat ABB seine RCS Module ins TCP/IP Netzwerk ausgelagert und erlaubt so auch die Benutzung über Rechnergrenzen hinweg. In der Spezifikation wird jedoch nie von RPC gesprochen.

Die Interaktion zwischen einem CAR Tool und einem RCS Modul wird dabei immer vom CAR Tool gesteuert. Das CAR Tool sendet eine entsprechende Nachricht, mit der eine entfernte Methode aufgerufen wird. Die Methode wird bearbeitet und die Antwort zurück an das RCS Modul geliefert. Benötigt das CAR Tool weitere Informationen, müssen die jeweiligen Methoden dazu aufgerufen werden. Das RCS Modul kann dabei nicht von sich aus eine Interaktion initiieren.

Unter anderem, um die Interaktion vom RCS Modul aus initiieren zu können, wurde die zweite Generation von RCS Modulen (RRS-2) [3] gestartet. Diese konnte sich aber bisher nicht roboterherstellerübergreifend durchsetzen.

Da das RCS Protokoll in dieser Arbeit unverändert benutzt und nur der Client optimiert werden soll, werden die Möglichkeiten der Beschleunigung durch das Protokoll beschränkt. So wird immer mindestens die Zeit zur Beantwortung einer Methode benötigt, die auch aktuell schon benötigt wird. Diese Zeit lässt sich nicht verkürzen.

Wenn die Methoden nicht sequenziell nacheinander ausgeführt werden müssen, kann die Ausführung nebenläufig an mehrere RCS Module geschehen. Dies ist aber durch den Aufbau des RRS Protokolls, welches stark stateful ist, unwahrscheinlich. Möglich ist jedoch das nebenläufige Durchführen von mehreren, kompletten Simulationen, wie beispielsweise zum Simulieren von Varianten.

Eine Möglichkeit zur Beschleunigung bietet das Verkürzen der Zeit zwischen zwei Methoden im CAR Tool. So kann das CAR Tool beispielsweise die nächste Methode bereits vorbereiten, während im RCS Modul die vorherige Methode noch ausgeführt wird. So kann nach dessen Beendigung die nächste Methode schneller angestoßen werden.

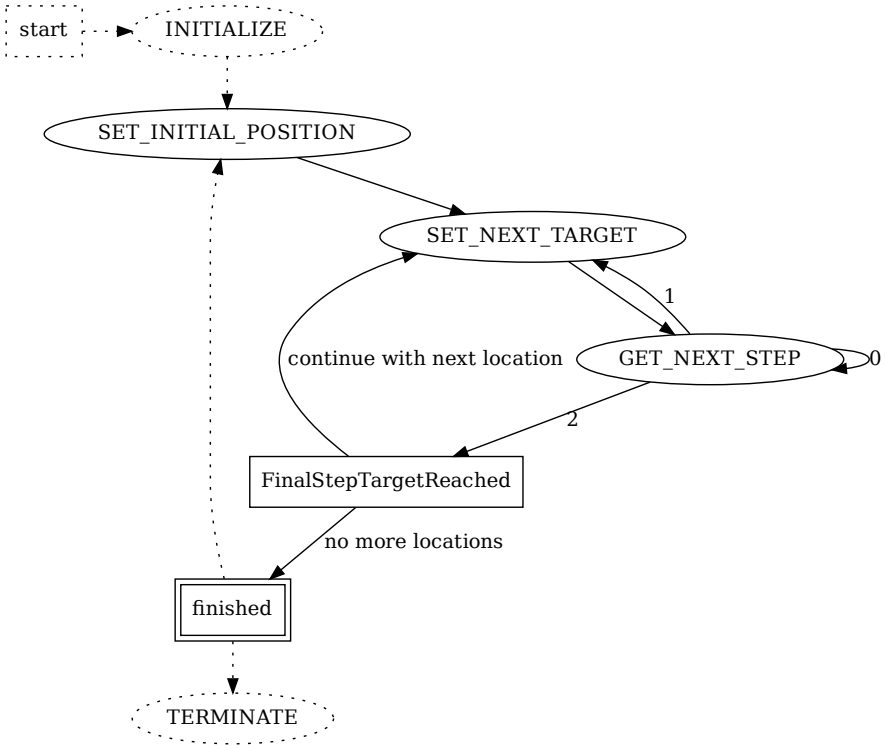


Abbildung 2.2: Motion Konzept des RRS Protokolls

Eine andere Möglichkeit bietet das Weglassen von Nachrichten bzw. dem Ausführen von Prozeduren. Dieser Ansatz verfälscht jedoch im Normalfall die Interaktion. Da die RCS Module stark stateful sind, werden viele Einstellungen zur Laufzeit gespeichert. Prozeduren, die nur bestimmte Einstellungen setzen, die bereits gesetzt wurden, sollten also unabhängig von der sonstigen Interaktion weggelassen werden können.

Zusätzlich zur Kommunikation kann jedoch möglicherweise auch die Interaktion optimiert werden, wofür im folgenden Abschnitt das Motion Konzept betrachtet wird.

2.2.2 Motion Konzept

Um die Interaktion mit dem RCS Modul zu beschleunigen, ist der Zweck der Nachrichten innerhalb der Interaktion von Bedeutung. Das dafür verwendete Motion Konzept ist in der RRS Spezifikation [1] beschrieben und hier kurz erläutert. Dabei wir der Roboterpfad mit einer bestimmten Abtastrate simuliert.

Der im Folgenden beschriebene Nachrichtenverlauf ist in Abbildung 2.2 dargestellt. Dabei sind mit den runden Elementen in der Abbildung die Methoden, die im RCS Modul aufgerufen werden, dargestellt. Die Rechteckigen symbolisieren Entscheidungen im CAR Tool. Existieren mehrere Möglichkeiten von einem Schritt zum Nächsten, ist der Pfeil für seine jeweilige Bedeutung beschriftet. Gestrichelt sind die Elemente, die nicht zwingend benötigt werden. So kann beispielsweise ein RCS Modul bereits initialisiert sein und muss daher nicht initialisiert werden.

Sobald das CAR Tool eine Simulation mit einem RCS Modul durchführen will, wird das RCS Modul gestartet. Darauf folgend nutzt das CAR Tool *INITIALIZE* um dem RCS Modul unter anderem den verwendeten Robotertyp und die Roboterkonfiguration zu übergeben, zu dem eine Simulation durchgeführt werden soll. Dabei gibt das RCS Modul ein Roboter Handle zurück, mit dem das CAR Tool in der folgenden Interaktion auf den Roboter verweist.

Soll eine neue Simulation gestartet werden, so wird mit *SET_INITIAL_POSITION* die Roboterachsstellung gesetzt. Von dieser Position wird die Simulation beginnen. Zu diesem Zeitpunkt können auch weitere Methoden benutzt werden, die die Simulation genauer spezifizieren. Nach der Initialisierung beginnt die Schleife.

Das CAR Tool übergibt dem RCS Modul mittels *SET_NEXT_TARGET* die erste Location im Pfad, die der Roboter anfahren soll. Danach fragt das CAR Tool nach einem Simulationsschritt, dessen Schrittweite sich aus der Abtastrate ergibt. Das RCS Modul berechnet die Position des Roboters nach der Schrittweite eines Simulationsschrittes und gibt die neue Positionierung des Roboters zurück. Außerdem wird dabei ein Status zurückgegeben, der aussagt, ob das RCS Modul weitere Targets, also Pfadlocations benötigt. Der jeweils zurückgegebene Status bei einem Simulationsschritt in dem Beispielpfad ist in Abbildung 2.1 eingezeichnet. Solange der Status 0 bleibt, wird die Simulation Schritt für Schritt fortgeführt.

Ändert sich der Status auf 1 oder 2, benötigt der Roboter eine weitere Location zur Weiterberechnung des Pfades. Ist der Status 2, so hat der Roboter in der Simulation sein Ziel erreicht. Dies kann das Ende der Simulation bedeuten, muss aber nicht, wie man an P2 in Abbildung 2.1 sieht.

Wird der Pfad mit Überschleifradien simuliert, so benötigt der Roboter im Verlauf der Simulation die Position der Location, die er nach der aktuellen Location anfahren soll. In diesem Falle wird mit Status 1 signalisiert, das eine weitere Location benötigt wird.

Das Setzen des Überschleifradiuses oder anderer Parameter wie der Maximalgeschwindigkeiten erfolgt vor dem Setzen eines neuen Targets. Da das RCS Modul stateful ist und diese Parameter beibehält, müssen nur Wertänderungen mitgeteilt werden.

3 Analyse

RCS Module werden bereits seit Jahrzehnten produktiv zur Simulation in der Roboter Off-Line Programmierung eingesetzt. Die aktuellste Version 1.3 [2] der RRS Spezifikation wurde 1997 veröffentlicht. Wie in Abschnitt 2.2 „RRS Protokoll“ bereits angesprochen, wurde das Protokoll erweiterbar gestaltet und könnte damit in einer weiterentwickelten Form im produktiven Einsatz verwendet werden. Um das Protokoll in seiner aktuell verwendeten Form zu verstehen und zu benutzen, muss das Ansprechen der RCS Module zuerst aus *Process Simulate* herausgelöst und analysiert werden. Dazu wird einer ersten Feststellung einer TCP/IP-Kommunikation während der Simulation mit einem RCS Modul nachgegangen, um die RRS Interaktion, die aktuell in Produktivprojekten statt findet, zu analysieren.

Eine Alternative stellt der Einsatz eines Man-in-the-Middle-Tools dar. Dabei wird ein eigenes RCS Modul für *Process Simulate* zur Verfügung gestellt, das sämtliche Datenflüsse an das produktiv verwendete RCS Modul weiter leitet.

Um zu erkennen, an welchen Punkten der Interaktion eine relevante Beschleunigung der Simulation möglich ist, wird das Zeitverhalten der Komponenten betrachtet. Basierend auf dem Zeitverhalten kann beurteilt werden, ob die angestrebte Beschleunigung der Interaktion sequenziell erreicht werden kann oder ob der Fokus auf nebenläufige Simulationen gelegt werden sollte.

Der Abschnitt 3.1 „TCP/IP“ betrachtet die in einer ersten Feststellung beobachtete TCP/IP-Kommunikation während einer Simulation. Abschnitt 3.2 „Sniffer/ MitM“ behandelt die erfolgreiche Erstellung eines Sniffing Tools oder auch Man-in-the-Middle-Tools zum Mitlesen der Interaktionen, sowie der daraus resultierenden Analyse der Nachrichten.

3.1 TCP/IP

In einer ersten Untersuchung der Interaktion zwischen *Process Simulate* und einem RCS Modul eines ABB Roboters durch die Firma ICARUS Consulting GmbH wurde bereits die Existenz einer TCP/IP-Verbindung mit Hilfe des Tools *Wireshark* [7] festgestellt, aber nicht weiter untersucht.

Im Rahmen dieser Bachelorarbeit konnte das Beobachten der TCP/IP-Kommunikation mit Hilfe von *Wireshark* nachgestellt werden. *Wireshark* bietet für die ABB-TCP/IP-Kommunikation bis zur Fertigstellung dieser Bachelorarbeit keinen offiziellen „Dissector“ zur Analyse des RRS Protokolls [8].

Zusätzlich wird mit Hilfe von *TCPView* [9] beobachtet, das nur ABB RCS Module eine Kommunikation über TCP/IP durchführen. *TCPView* ist ein Tool für Windows um offene Netzwerk Sockets und die Prozesse dazu anzuzeigen, ähnlich wie *netstat* dafür unter Linux genutzt wird.

RCS Module anderer Hersteller haben keine offenen Netzwerk Sockets. Somit wäre eine Reimplementierung eines RCS Clients auf TCP/IP Basis nur mit RCS Modulen von ABB nutzbar.

Da nicht alle Roboter Hersteller die Kommunikation über TCP/IP durchführen, wäre es möglich, dass ABB die RRS Kommunikation in einem eigenen Protokoll abstrahiert oder angepasst hat. Die beobachtete Interaktion könnte somit Zusatzelemente, zusätzlich zu der in der RRS Interface Specification [1] spezifizierten Interaktion, enthalten oder sich gänzlich unterscheiden. Somit können mögliche Erkenntnisse aus der Analyse der TCP/IP Interaktion nicht für RCS Module anderer Hersteller verwendet werden.

Um roboterunabhängig zu bleiben, wird der Ansatz der Analyse der TCP/IP Interaktion nicht verwendet.

3.2 Sniffer/ MitM

Als weiterer, alternativer Ansatz bietet sich das Eingreifen und Mitlesen der existierenden Kommunikation über einen „Man-in-the-Middle-Angriff“ an. *Process Simulate* kommuniziert dann mit dem „Sniffer“ und dieser mit dem RCS Modul. Dieser Aufbau kann in einem weiteren Schritt aufgetrennt werden und der Sniffer gegen die Reimplementierung ausgetauscht werden. Wird der Sniffer dabei bereits so aufgebaut, dass die Nachrichten eingelesen werden und mit einer eigenen Komponente wieder versendet werden, kann die Reimplementierung später dieselbe Komponente zur Kommunikation direkt verwenden.

Um den Sniffer zu realisieren, wird zuerst der bestehende Kommunikationsfluss analysiert und der ideale Angriffspunkt bestimmt, siehe Unterabschnitt 3.2.1 „Angriffspunkte“.

Mit Hilfe des Sniffers wird dann in Unterabschnitt 3.2.3 „Nachrichten Analyse“ die Interaktion für die Reimplementierung eines eigenen RCS Clients analysiert. Außerdem kann so die reale Interaktion mit der spezifizierten Interaktion verglichen werden, um mögliche Abweichungen oder Erweiterungen seit der RRS Spezifizierung ebenfalls in die Reimplementierung einfließen lassen zu können.

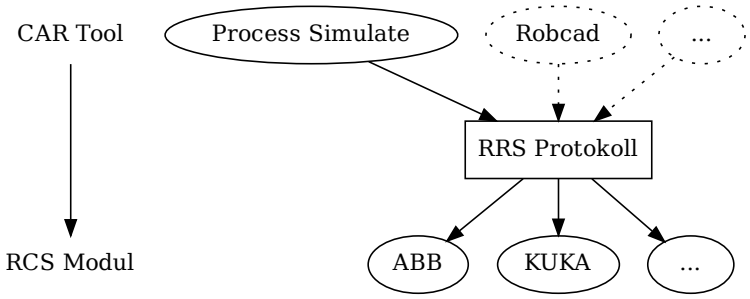


Abbildung 3.1: Grundlegender Aufbau aus CAR Tool und den RCS Modulen nach [1]

3.2.1 Angriffspunkte

Zur Auftrennung und Platzierung des „Sniffers“ wird zunächst der Nachrichtenfluss analysiert. Der Nachrichtenfluss von CAR Tools wie *Process Simulate* an die RCS Module wird in Abbildung 3.1 gezeigt.

Der Kommunikationsfluss von *Process Simulate* mit einem RCS Modul ist in Abbildung 3.2 aufgeführt. Dort ist auch der Unterschied zwischen dem in der RRS Spezifikation beschriebenen Verhalten und dem der realen, durch die Roboterhersteller ausgelieferten Module, aufgezeigt. Laut Spezifikation enthalten die RCS Module nur eine Library mit einem Einstiegspunkt (siehe Unterunterabschnitt 3.2.1.3 „C-Call rcsXXYY“). Das ausgelieferte RCS Modul enthält jedoch zusätzlich einen Wrapper, der über Shared Memory angesprochen wird (siehe Unterunterabschnitt 3.2.1.4 „Shared Memory“). Nur der Wrapper wird von *Process Simulate* verwendet und nicht die Library des RCS Moduls direkt.

Aus den betrachteten Ansätzen hat sich Unterunterabschnitt 3.2.1.4 „Shared Memory“ durchgesetzt und wird in Unterabschnitt 3.2.2 „Implementation“ verwendet.

3.2.1.1 TCP/IP

Wie bereits in Abschnitt 3.1 „TCP/IP“ beobachtet, findet innerhalb des ABB RCS Moduls eine TCP/IP Kommunikation statt. Diese ist aus den dort genannten Gründen nicht gut für ein weiteres Vorgehen geeignet. Allerdings kann, wenn keiner der anderen Angriffspunkte effektiv genutzt werden kann, die TCP/IP Kommunikation für den Sniffer verwendet werden, um die aktuelle RRS Interaktion mit der RRS Spezifikation zu vergleichen.

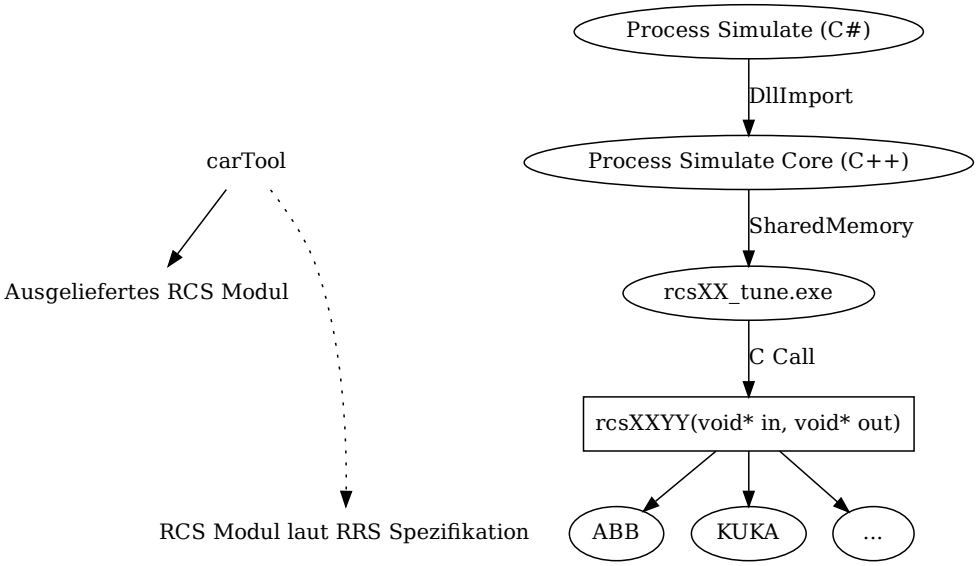


Abbildung 3.2: Nachrichtenfluss in *Process Simulate*

3.2.1.2 RRS DEBUG Option

In der RRS Spezifikation [1] ist die Möglichkeit gegeben, dem RCS Modul anzuweisen, ein Debug Log auf Dateibasis zu erstellen. Das Format des Logs wurde grundlegend spezifiziert, unterscheidet sich aber dennoch in Details zwischen den beobachteten RCS Modulen von KUKA und ABB. Es ist also keine einfache, roboterherstellerübergreifende Analyse möglich.

Außerdem bietet die RRS DEBUG Option keine Möglichkeit des Sendens von Nachrichten an das RCS Modul. Es wäre somit nur zur Analyse der Nachrichten hilfreich und wird damit zur Erstellung des Sniffers nicht betrachtet.

3.2.1.3 C-Call rcsXXYY

Der in der RRS Spezifikation [1] spezifizierte Weg, um ein RCS Modul anzusprechen, ist der Aufruf über einen C-Call. Die Library des RCS Moduls muss dabei genau einen Einsprungpunkt aufweisen, der dem Format in Listing 3.1 entspricht.

```
rcsXXYY(void* in, void* out)
```

Listing 3.1: Aufruf eines RCS Moduls über einen C-Call

Dabei steht XX für die zweistellige, klein geschriebene Abkürzung des Roboterherstellers und YY für die zweistellige Versionsnummer. Ein beispielhafter Funktionskopf für den Aufruf an ein ABB RCS Modul der Version 1 ist in Listing 3.2 gezeigt.

```
rcsab01(void* in, void* out)
```

Listing 3.2: Beispielhafter Aufruf eines ABB RCS Moduls über einen C-Call

Die Eingangsdaten werden als Type-Length-Value Byte-Paket mitgegeben und ein gewisser Speicherbereich für die Ausgabedaten reserviert. Die Größe des Speichers für die Ausgabedaten ist ebenfalls in den Eingangsdaten angegeben.

Laut der RRS Spezifikation soll der C-Call von vielen Programmiersprachen aus aufrufbar sein. Aus diesem Grund soll die Library des RCS Moduls auch genau einen Einsprungpunkt enthalten, um das Aufrufen weiter zu vereinfachen. Allerdings verwenden Programmiersprachen wie C# den Namen des Einsprungpunktes, der sich je nach Hersteller und Version unterscheidet, weshalb für die Benutzung des C-Calls für jeden Hersteller und Version eines RCS Moduls ein individueller Eintrag im Quellcode hinzugefügt werden muss.

Process Simulate verwendet den C-Call nicht direkt, sondern abstrahiert diesen über ein Shared Memory. Der Shared Memory ist genauer in Unterunterabschnitt 3.2.1.4 „Shared Memory“ beschrieben. Die Abstraktion sorgt dafür, dass *Process Simulate* nicht den C-Call als bindende Schnittstelle, sondern den Shared Memory nutzt. Ein Hersteller von RCS Modulen kann also eine andere Schnittstelle als den C-Call zu seinem RCS Modul verwenden, solange er das Shared Memory nutzt. Beispielsweise wurde beobachtet, das KUKA sich nicht an die RRS Spezifizierung des C-Calls hält. Listing 3.3 zeigt, dass KUKA 3 Stellen für den Herstellernamen und eine Stelle für die Version benutzt, statt jeweils 2 Stellen zu verwenden.

```
rcskrc1(void* in, void* out)
```

Listing 3.3: Ein nicht der RRS Spezifikation entsprechender C-Call eines KUKA RCS Moduls

3.2.1.4 Shared Memory

In der RRS Spezifikation [1], 3.2 „Integration Concept“, „The Two Module Concept“ wird die Möglichkeit erwähnt, dass CAR Tools den C-Call, genauer beschrieben in Unterunterabschnitt 3.2.1.3 „C-Call rcsXXYY“, mit Hilfe eines Shared Memory abstrahieren können. Ein Shared Memory ist ein geteilter Speicher, auf den mehrere Parteien lesend und schreibend zugreifen können. Um die Korruption des Speichers durch gleichzeitigen Zugriff zu vermeiden, wird in diesem Fall mit Hilfe von Semaphoren die aktive Partei signalisiert.

Ziel des Shared Memory Ansatzes ist laut RRS Spezifikation, weitere Kompatibilität mit Programmiersprachen zu gewährleisten, sowie die Kommunikation von CAR Tool und RCS

Modul zu entkoppeln. Das abstrahierende Programm, welches den Shared Memory ausliest und an das RCS Modul per C-Call gibt, wird im folgenden Wrapper genannt. Außerdem muss der Wrapper nicht jedes Mal kompiliert werden, wenn das CAR Tool entwickelt wird. Genauer wird auf das Shared Memory Konzept nicht in der RRS Spezifikation eingegangen.

Die Art und Weise, wie *Process Simulate* die Wrapper benutzt, ist undokumentiert und wurde im Rahmen dieser Bachelorarbeit analysiert. Die Wrapper werden, nicht wie in der RRS Spezifikation vorgeschlagen, vom CAR Tool bereitgestellt, sondern von den Herstellern der RCS Module mitgeliefert. Alle installierten Wrapper sind in einer rrs.xml innerhalb des *Process Simulate* Installationsverzeichnisses aufgelistet, die *Process Simulate* ausliest und für den Nutzer zur Auswahl anbietet. *Process Simulate* spricht die Wrapper immer mit denselben Aufrufparametern an, wie in Listing 3.4 zu sehen. Meistens endet der Dateiname des Wrappers auf „_tune.exe“. Die ersten beiden Parameter enthalten die Namen der Semaphoren, die zur Signalisierung des beschriebenen Speichers dienen. Der dritte Parameter enthält den Namen des Shared Memory. Der vorletzte Parameter enthält den Offset, ab dem im Shared Memory der Bereich für den Output dient. Der letzte Parameter enthält die Process ID (PID) des startenden Programms, also der PID von *Process Simulate*. Beispielsweise nutzt ABB die PID, um sich selbst zu beenden, sobald das Programm der PID beendet wurde. KUKA scheint die PID nicht zu nutzen.

```
<rcs-modul>_tune.exe "/semaphores/RRS_76732_0" "/semaphores/
   RRS_76732_1" "/sharedMem/RRS_76732" 4096 1695
```

Listing 3.4: Beispielhafter Aufruf des Shared Memory Wrappers aus *Process Simulate*

Die Semaphoren werden zur Signalisierung auf dem Shared Memory, ab wann der Inhalt gelesen werden kann, genutzt. In Abbildung 3.3 ist dieser Zyklus, basierend auf der durchgeführten Nachrichtenfluss-Analyse, skizziert.

Ein Vorteil der Nutzung des Shared Memory ist, dass es exakt dieselbe Schnittstelle wie *Process Simulate* zu den RCS Modulen darstellt. Alle RCS Module, die so mit *Process Simulate* arbeiten, werden über diesen Weg analysierbar sein. Außerdem werden dieselben Nachrichten verwendet, wie auch über den C-Call. Die Reimplementierung könnte also in Zukunft auch durch Austauschen der Shared Memory Komponente auf die möglicherweise schnellere Benutzung des C-Call geändert werden.

Die Implementierung eines Sniffers über das Shared Memory wird im folgenden Abschnitt beschrieben.

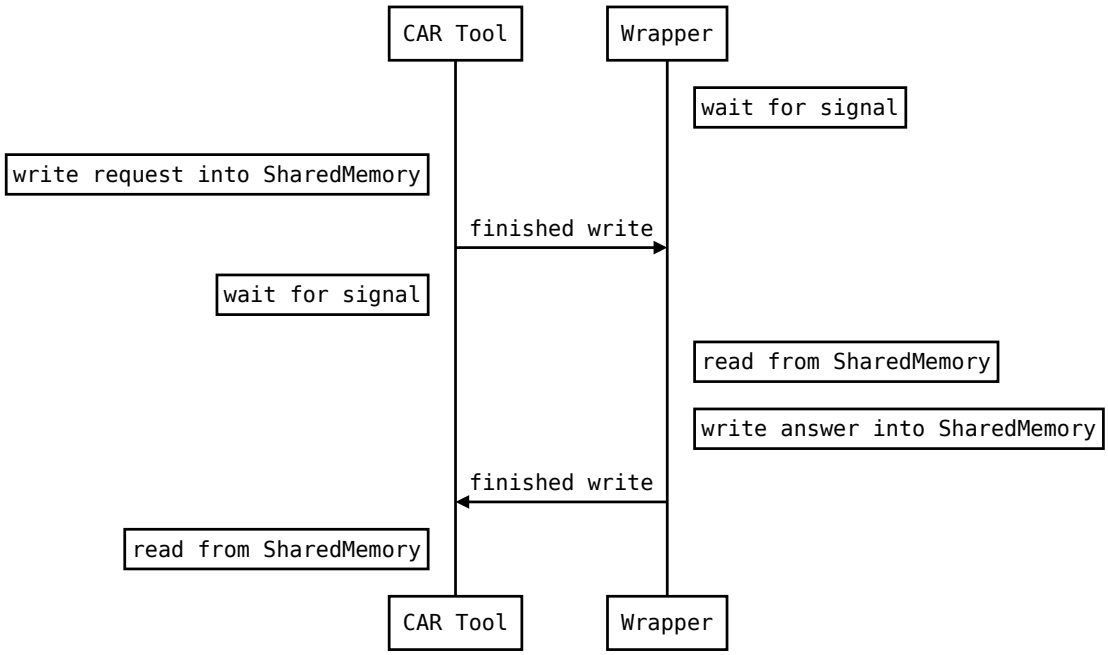


Abbildung 3.3: Synchronisierung des Schreib- und Lese-Zugriffs auf dem Shared Memory

3.2.2 Implementation

Um einen Sniffer in das produktiv eingesetzte *Process Simulate* zu integrieren, wird ein eigenes Programm erstellt. Das Programm muss dieselben Kommandozeilenparameter wie die Wrapper akzeptieren und wird in die rrs.xml innerhalb von *Process Simulate* wie ein RCS Modul eingetragen. Dem Roboter in *Process Simulate*, mit dem die Tests durchgeführt werden, wird der Sniffer als RCS Modul zugeordnet. Das eigentlich verwendete RCS Modul wird innerhalb des Sniffers hinterlegt und wird später vom Sniffer statt von *Process Simulate* gestartet.

Der Sniffer liest den Input, den *Process Simulate* byteweise sendet, ein und prüft mit Hilfe des mitgegebenen Opcode den Nachrichtentyp. Ist das Einlesen des Nachrichtentyps bereits im Sniffer implementiert, wird der Inhalt in das der Nachricht zugehörige Datenobjekt geschrieben. Das Senden an das RCS Modul wird von einer eigenen Komponente übernommen, die die Datenobjekte entgegen nimmt und daraus die Nachricht byteweise in den Speicher schreibt. Dies hat den Vorteil, dass die implementierte Komponente für das Senden an das RCS Modul später direkt in der Reimplementierung verwendet werden kann.

Im späteren Verlauf der Bachelorarbeit wurde dieses Verhalten angepasst und die Daten nicht mehr von Bytes in Datenobjekte und zurück gewandelt, da diese Umwandlung bei Programmierfehlern zu einer verfälschten Interaktion führen kann bzw. führte.

Beim Einlesen der Datenobjekte aus den Input Bytes wurde sich grundsätzlich auf die RRS Spezifikation des jeweiligen Nachrichtentyps (Opcode) verlassen. Wenn neue Nachrichten zum Protokoll hinzugefügt worden wären, müssten diese laut Spezifikation neue Opcodes größer 1000 verwenden.

Problematisch gestaltete sich die binäre Darstellung der in der RRS Spezifikation verwendeten Datentypen, da diese nicht explizit definiert sind. Die binäre Darstellung wurde mit Hilfe der bekannten binären Darstellungen (Byte Inhalt und Längen) per Ausschlussverfahren festgestellt. Dies waren zu Beginn Annahmen, da die meisten Datentypen beim beobachteten RCS Initialisierungsvorgang mit 0 gefüllt sind. Später bestätigten sich diese Annahmen, sobald der Inhalt während Simulationen mit den vermuteten Inhalten in *Process Simulate* überein stimmte. So wurde beispielsweise festgestellt, dass ein „real“ Datentyp ein 64 Bit Floating Point Wert und ein „real32“ ein Array von 32 „real“ Werten ist.

3.2.3 Nachrichten Analyse

Zur Analyse der Nachrichten wird der aktuell verwendete Ist-Stand mit der Spezifikation verglichen. Wird ein Nachrichtentyp nach der Spezifikation implementiert und es treten Fehler auf, die nicht in der Implementierung liegen, gibt es Differenzen zwischen Spezifikation und Ist-Stand. Da alle verwendeten Nachrichtentypen betrachtet und wenn nötig implementiert werden, um dessen Sinn zu verstehen, wird eine hohe Abdeckung der Analyse erreicht.

Überwiegend folgt *Process Simulate* bei den Nachrichten den in der RRS Spezifikation vorgegeben Abläufen der Interaktion. Aufgefallen sind dennoch einige Nachrichten:

Opcode 1201 *Process Simulate* sendet bei der Initialisierung einer neuen Simulation eine Nachricht mit dem Nachrichtentyp (Opcode) 1201. Dieser Opcode befindet sich laut Spezifizierung im Bereich der selbst definierten, hinzugefügten Methoden. Alle im Rahmen der Bachelorarbeit beobachteten RCS Module haben diese Nachricht mit dem Statuscode 'Not Implemented' beantwortet. Der Zweck dieser Nachricht ist daher unbekannt.

GET_RCS_DATA* und *MODIFY_RCS_DATA Ein Punkt der aufgefallen ist, sind die Parameter der Methoden *GET_RCS_DATA* und *MODIFY_RCS_DATA*. Diese Verwenden den Parameter „Storage“, der laut Spezifikation die möglichen Inhalte „InMemory“ und „OnDisk“ hat. Wenn der Parameter „OnDisk“ verwendet wird, kann dies möglicherweise Einfluss auf eine nebenläufige Durchführung haben, da die RCS Module sich gegenseitig die Daten korrupieren könnten. Im Falle vom untersuchten RCS Modul von ABB wird immer nur „InMemory“ ver-

wendet. Wird stattdessen „OnDisk“ mitgegeben, antwortet das RCS Modul von ABB mit dem Statuscode für „Storage not supported“.

Ansteuerung je nach Hersteller Ein weiterer beobachteter Punkt ist die teilweise unterschiedliche Benutzung der Nachrichten je nach Hersteller. Alle gefundenen Unterschiede sind im Rahmen der aktuellen Spezifizierung und würden eine präzisiere RRS Spezifizierung voraussetzen.

Beispielsweise verwendet KUKA fortlaufende IDs beim Hinzufügen eines neuen Targets. Ist bereits ein Target mit einer ID hinterlegt, tritt ein Fehler beim erneuten Versuch des Setzens auf. Bei ABB hingegen wird immer die TargetID 0 verwendet. Wird eine andere ID mitgegeben, tritt ein Fehler auf.

Diese Unterschiede in der Ansteuerung sind ebenfalls innerhalb von *Process Simulate* implementiert. Daher muss *Process Simulate* zumindest teilweise unterschiedliche Logik, je nach RCS Modul, verwenden.

Um die Nachrichten genauer zu verstehen, wird im folgenden Abschnitt der Zweck der Nachrichten analysiert bzw. aus der RRS Spezifikation nachgeschlagen. Danach wird in Unterunterabschnitt 3.2.3.2 „Zeitverhalten“ der Zeitbedarf der einzelnen Komponenten verglichen.

3.2.3.1 Nachrichten Zweck

Um einen minimalen Client zu erstellen, muss die kleinstmögliche Anzahl an Nachrichten versendet werden. Das kann beispielsweise das Starten und das Beenden umfassen. So kann sichergestellt werden, dass die grundlegende Kommunikation funktionsfähig implementiert ist und darauf basierend komplexere Interaktionen durchgeführt werden können.

Um nah an der produktiv eingesetzten Logik zu bleiben, wird eine Nachrichtensequenz mit Hilfe des Sniffers mitgeschnitten. Aus den mitgeschnittenen Binärdaten kann ein weiteres, im Rahmen dieser Bachelorarbeit erstelltes Analyse Tool menschenlesbare Logs erstellen. Diese werden zur Analyse der Nachrichten genutzt.

Zur Analyse der Nachrichten sind zwei Aspekte zu betrachten: Überblick über den Programmfluss und der genaue Inhalt der Daten. Dafür erstellt das Analyse Tool zwei Logs aus den Binärdaten: ein Kurzes, im Folgenden als ShortLog bezeichnetes Log, wie in Listing 3.5 zu sehen und ein langes, vollständiges Log.

Das ShortLog enthält die wichtigsten Informationen einer Nachricht, wie Nachrichtentyp (Opcode) und Rückgabestatus. Zusätzlich sind die relevantesten Informationen der spezifischen

Nachricht und dessen Antwort aufgeführt. Relevante, aber komplexer darzustellende Inhalte wie Positionsmatrizen werden ebenfalls weg gelassen, um die Übersicht zu behalten.

Informationen wie Startzeit und Sequenznummer sind nicht in der Nachricht selbst enthalten, sondern werden separat vom Sniffer aufgezeichnet. Die Sequenznummer beginnt dabei bei jeder aufgezeichneten Sequenz bei 0. Die Startzeit sowie genaueres Zeitverhalten der Nachrichten wird in Unterunterabschnitt 3.2.3.2 „Zeitverhalten“ genauer betrachtet und analysiert.

Außerdem wird ein langes, vollständiges Log erzeugt, das sämtliche Inhalte der Nachrichten enthält. Auf das vollständige Log wird in dieser Bachelorarbeit nicht weiter verwiesen, da nur der Zusammenhang der Nachrichten untereinander für das Vorgehen der Reimplementierung relevant ist. Der genaue Nachrichteninhalt ist nur für die korrekte Programmierung der Reimplementierung von Bedeutung.

Im ShortLog steht jede Zeile für eine Nachricht bzw. einen entfernen Methodenaufruf und dessen Antwort. Beispielsweise wird in Listing 3.5 Zeile 1 die Methode *INITIALIZE* am RCS Modul aufgerufen. Dies ist die erste Nachricht (Sequenznummer 000), die Kommunikation dieser Nachricht begann 85 ms nach der ersten Bereit Signalisierung an *Process Simulate* und das RCS Modul hat die Nachricht erfolgreich abgearbeitet (Successful).

```
1  00:00:00.0845836 000 INITIALIZE -> Successful
2  00:00:02.2356733 001 GET_ROBOT_STAMP -> Successful
3  00:00:02.2596821 002 SET_INITIAL_POSITION -> Successful
4  00:00:02.3057284 003 GET_MESSAGE 0 -> MessageNumberNotFound
5  00:00:02.3217340 004 GET_MESSAGE 1 -> Successful  NoSeverityGiven 120006:
6  00:00:02.3447445 005 GET_MESSAGE 2 -> Successful  NoSeverityGiven 71389:
7  00:00:02.3617487 006 GET_MESSAGE 3 -> Successful  NoSeverityGiven 120006:
8  00:00:02.3757557 007 GET_MESSAGE 4 -> Successful  NoSeverityGiven 120006:
9  00:00:02.3902849 008 GET_MESSAGE 5 -> Successful  NoSeverityGiven 120006:
10 00:00:02.4115159 009 GET_MESSAGE 6 -> Successful  NoSeverityGiven 120006:
11 00:00:02.4255221 010 GET_MESSAGE 7 -> Successful  NoSeverityGiven 120006:
12 00:00:02.4445279 011 GET_MESSAGE 8 -> Successful  NoSeverityGiven 120006:
13 00:00:02.4595340 012 GET_MESSAGE 9 -> Successful  NoSeverityGiven 120006:
14 00:00:02.4780506 013 GET_MESSAGE 10 -> Successful  NoSeverityGiven 120006:
15 00:00:02.4925900 014 GET_MESSAGE 11 -> Successful  NoSeverityGiven 120006:
16 00:00:02.5065970 015 GET_MESSAGE 12 -> Successful  NoSeverityGiven 120006:
17 00:00:02.5226018 016 GET_MESSAGE 13 -> Successful  NoSeverityGiven 71389:
18 00:00:02.5456398 017 GET_MESSAGE 14 -> Successful  NoSeverityGiven 120006:
19 00:00:02.5606437 018 GET_MESSAGE 15 -> Successful  NoSeverityGiven 120006:
20 00:00:02.5746489 019 1201 untyped! -> NotSupported
```

Listing 3.5: Log der Interaktion beim Initialisieren eines RCS Modules durch *Process Simulate*

Die Nachrichten der Sequenz in Listing 3.5 sind:

- *INITIALIZE* dient zum Initialisieren einer Roboterinstanz. Mitgegeben werden Informationen wie Roboterart oder der Pfad zum Maschinendaten Ordner und zurückgegeben wird das Roboter Handle, mit dem man den Roboter für weitere Nachrichten im RCS Modul referenziert. Diese Nachricht wird für eine Kommunikation zwingend benötigt.
- *GET_ROBOT_STAMP* dient laut der RRS Spezifikation [1] Abschnitt 3.4.1 zum Auslesen von Daten, die im CAR Tool angezeigt werden können, sonst aber nicht benötigt werden. Daher wird diese Nachricht nicht weiter betrachtet.
- *SET_INITIAL_POSITION* initialisiert die Position, auf der der Roboter für folgende Aktionen stehen soll. Diese Nachricht ist optional, stellt aber den State sicher, in dem sich das RCS Modul befindet. Für Simulationen sollte diese Nachricht verwendet werden.
- *GET_MESSAGE* ruft eine Message, die hinter einer ID hinterlegt ist, ab. Die Anzahl vorhandener Nachrichten ist in diesem Fall in der Antwort auf *INITIALIZE* angegeben worden. Das Abrufen von Messages verändert den State des RCS Moduls nicht. Zusätzlich dazu enthält der Message Content bei ABB nur Zahlen, die auf den ersten Blick nichts sagen (möglicherweise hilft die ABB Doku weiter). Deswegen wird diese Nachricht im weiteren Verlauf nicht weiter betrachtet.
- *1201* ist eine unbekannte Nachricht. „untyped!“ ist der Hinweis des entwickelten Analysetools, das die Typisierung der Binärdaten der Nachricht nicht implementiert wurde. Opcode 1201 ist in Unterabschnitt 3.2.3 „Nachrichten Analyse“ genauer beschrieben.

Eine Endnachricht der Interaktion wird nicht verwendet. Wird die Interaktion beendet, terminiert *Process Simulate* das RCS Modul.

Das Betrachten einer kompletten Simulation umfasst schnell mehrere tausend Nachrichten. Die Anzahl der Nachrichten hängt dabei unter anderem von der Anzahl der Locations im Pfad (*SET_NEXT_TARGET*) und der Realzeit, die der Roboter zum Durchfahren des Pfades benötigt, ab (*GET_NEXT_STEP*). Beim Setzen eines neuen Targets muss das Target konfiguriert werden (Geschwindigkeiten, Überschleif, usw.), wofür jeweils Nachrichten benötigt werden. Da der Pfad in Zeitschritten simuliert wird, ist die Anzahl der Nachrichten abhängig von der Gesamtdauer und Schrittweite. Das ABB RCS Modul hat die Schrittweite nicht änderbar auf $\approx$ 24.19 ms gesetzt. Dies ist die Zeit, die von einer *GET_NEXT_STEP* Nachricht simuliert wird. Beim Versuch diese mit der dazu passenden Nachricht anzupassen, liefert das ABB RCS Modul 'NOT_SUPPORTED' zurück.

Die Simulation mit einem ABB RCS Modul eines Roboterpfades über 25 Pfadpunkte, für den Roboter knapp 16 Sekunden benötigt, umfasste bereits 1120 Nachrichten. Dieser Pfad enthält zu Beginn und am Ende An- und Abfahrpunkte, sowie einen Abschnitt mit einer Linearbewegung. Die verwendeten Nachrichten im Testpfad, sowie deren Anzahl sind in Listing 3.6 aufgeführt.

```
001x INITIALIZE
001x RESET
001x GET_RCS_DATA
001x DEBUG
001x 1201
002x GET_ROBOT_STAMP
002x SET_INITIAL_POSITION
002x SET_JOINT_ACCELERATIONS
002x SET_JOINT_JERKS
005x SELECT_WORK_FRAMES
006x MODIFY_CELL_FRAME
016x GET_MESSAGE
018x MODIFY_RCS_DATA
023x DEFINE_EVENT
023x GET_EVENT
025x SET_NEXT_TARGET
025x SELECT_MOTION_TYPE
025x SET_CARTESIAN_POSITION_SPEED
025x SET_CARTESIAN_ORIENTATION_SPEED
026x SELECT_FLYBY_MODE
041x GET_FORWARD_KINEMATIC
052x SET_CONFIGURATION_CONTROL
138x SET_FLYBY_CRITERIA_PARAMETER
659x GET_NEXT_STEP
```

Listing 3.6: Gesendete Nachrichten von *Process Simulate* in einem ABB Roboterprogramm

- *RESET* wird am Ende der Interaktion genutzt, um den State der Instanz des Roboters im RCS Modul zurück zu setzen. Nachfolgende Interaktionen können das RCS Modul dann wieder so benutzen, wie direkt nach dem *INITIALIZE*. Diese Nachricht ist nicht essentiell, kann aber das Wiederverwenden eines bereits laufenden RCS Moduls beschleunigen.
- *GET_RCS_DATA* und *MODIFY_RCS_DATA* werden zum Setzen und Abrufen von Einstellungen im RCS Modul verwendet. Mit dem Abrufen dieser Einstellungen wird in diesem Fall der Wert von „AuxDataMap“ abgefragt. Die Antwort dieser Anfrage wird von den verwendeten ABB RCS Modulen nicht wie erwartet beantwortet und in Unterunterabschnitt 3.2.3.3 „Anomalien“ genauer betrachtet. Gesetzt werden Roboter Tool

spezifische Einstellungen. Diese werden zu einem späteren Zeitpunkt benötigt, um eine realitätsnahe Simulation zu erreichen, sind aber nicht essentiell.

- *DEBUG* wird in diesem Fall eingesetzt, um sämtliche Debug Log Optionen zu deaktivieren. Diese Nachricht wird nicht benötigt und deswegen nicht betrachtet.
- *SET_JOINT_ACCELERATIONS* und *SET_JOINT_JERKS* setzen die maximalen Beschleunigungs und Ruck Werte für die jeweiligen Roboter Achsen, wenn eine Änderung vorliegt. Diese Nachrichten sind nicht essentiell, werden aber für realitätsnahe Simulationen benötigt.
- *SELECT_WORK_FRAMES* und *MODIFY_CELL_FRAME* wählen und setzen die relativen Frames / Koordinatensysteme. Diese Nachrichten werden zwar nicht vorausgesetzt, da diese im Testpfad aber deutlich größer sind, als beispielsweise Korrektur Koordinatensysteme für Abweichungen einer realen Anlage, werden die Nachrichten voraussichtlich für eine minimal lauffähige Simulation benötigt.
- *DEFINE_EVENT* und *GET_EVENT* dienen zur Definition und Behandlung von in der RRS Spezifikation definierten Events. Mit Hilfe dieser Events können Punkte zwischen den *GET_NEXT_STEP* Schritten bestimmt werden. Da die Events nur zusätzliche Ausgaben produzieren und sich nicht auf die sonstige Interaktion auswirken sollten, werden diese in dieser Arbeit nicht betrachtet.
- *SET_NEXT_TARGET* wird zum Setzen des nächsten Ziels genutzt und daher zwingend benötigt.
- *SELECT_MOTION_TYPE*, *SET_CARTESIAN_POSITION_SPEED*, *SET_CARTESIAN_ORIENTATION_SPEED* und *SELECT_FLYBY_MODE* dienen zur genaueren Spezifizierung einer im Folgenden übergebenen Location (*SET_NEXT_TARGET*). Diese Nachrichten sind nicht essentiell, werden aber voraussichtlich für eine realitätsnahe Simulation benötigt. *SELECT_FLYBY_MODE* wird kurz vor Ende des Roboterpfades ein zusätzliches Mal benutzt, um dem RCS Modul das Ende des Pfades statt einer weiteren Location mit Überschleif zu signalisieren.
- *GET_FORWARD_KINEMATIC* wird verwendet, um aus den Achs-Werten in *Process Simulate* die benötigten Koordinaten für *SET_NEXT_TARGET* zu berechnen. Diese Methode wird häufiger als *SET_NEXT_TARGET* aufgerufen, da die genaue Position bestimmter Events ebenfalls mit Hilfe dieser Methode bestimmt wird. Da sowohl in *Process Simulate*

als auch Roboterprogrammen bereits die Koordinaten der Locations hinterlegt sind, wird diese Methode voraussichtlich nicht benötigt und wäre nur für Events von Bedeutung.

- *SET_CONFIGURATION_CONTROL* wird für jeweils zwei Configs pro Target aufgerufen. Außerdem werden die Methoden für das erste Target doppelt aufgerufen. Da die gesetzten Werte immer gleich sind, ist zu vermuten, das diese Methode insgesamt jeweils nur einmal pro Config für den Testpfad benötigt werden würde. Außerdem ist die Methode nicht essentiell, wird aber für eine realitätsnahe Simulation benötigt.
- *SET_FLYBY_CRITERIA_PARAMETER* setzt für jedes Target mit Überschleif (23 der 25 Locations) jeweils 6 Überschleif Parameter. Diese Methode ist nicht essentiell, wird aber in Verbindung mit *SELECT_FLYBY_MODE* benötigt.
- *GET_NEXT_STEP* wird für jeden Zeitschritt der Simulation aufgerufen und wird daher zwingend benötigt.

Für eine erste, minimale Durchstichimplementierung werden nur *SET_NEXT_TARGET* und *GET_NEXT_STEP* zur Simulation eines Pfades benötigt. Vermutlich müssen die Frames / Koordinatensysteme gesetzt werden, um einen Fortschritt der Simulation zu erreichen. Darauf aufbauend können Nachrichten zur Spezifizierung eines Targets genutzt werden, sodass die eigene Simulation der über *Process Simulate* gestarteten Simulation gleicht.

Generell wurde beobachtet, dass Nachrichten zur Spezifizierung eines Targets mit selbem Inhalt häufiger wiederholt werden. Da die RCS Module stateful sind, sollten diese die Auswirkungen der Nachrichten behalten. Es ist daher davon auszugehen, dass die exakt gleichen Nachrichten nicht wiederholt werden müssen und so Zeit gespart werden kann.

3.2.3.2 Zeitverhalten

Zur Messung des Zeitverhaltens wird an verschiedenen Stellen in der Kommunikation die benötigte Zeit gemessen. Um Vor- oder Nachteile der zugrunde liegenden CPU Taktung auszuschließen, wird die Zeit aller Kommunikationsteilnehmer auf einem System gemessen und relativ zueinander betrachtet. Die 3 gemessenen Zeiten sind im Sequenz-Diagramm in Abbildung 3.4 eingezeichnet und im Folgenden beschrieben.

Zeitbedarf RCS Modul Die hauptsächlich gemessene Zeit ist die Zeit, die das RCS Modul für die Durchführung benötigt. Der Timer zur Messung dieser Zeit wird als letzte Aktion vor dem Signalisieren des beschriebenen Shared Memorys gestartet und als erste Aktion nach der Signalisierung des lesbaren Shared Memorys gestoppt. Der Zeitbedarf des RCS Moduls

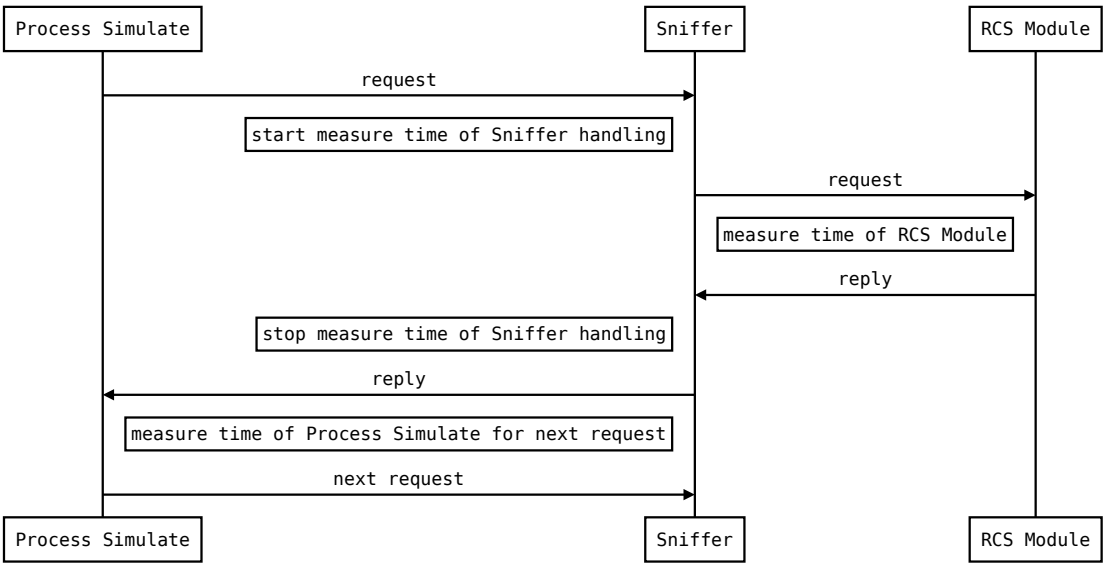


Abbildung 3.4: Messung des Zeitbedarfs von Methodenaufrufen am RCS Modul

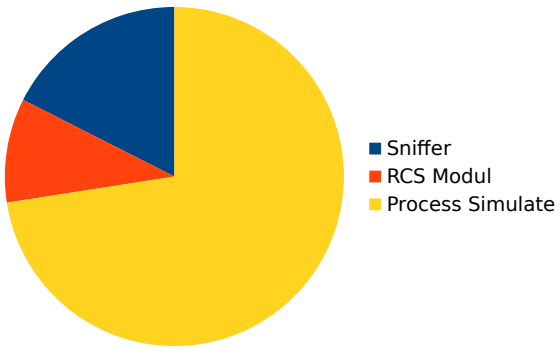


Abbildung 3.5: Zeitbedarf der Komponenten im Verhältnis zueinander

wird in dieser Arbeit als unvermeidbar und unveränderlich angenommen, da sich diese Arbeit auf die Beschleunigung genereller RCS Module nach der RRS Spezifizierung bezieht. Die Optimierung von RCS Modulen ist getrennt von dieser Arbeit zu betrachten. Daher wird die minimal benötigte Zeit für eine Simulation immer über dem Zeitbedarf des verwendeten RCS Moduls liegen.

Zeitbedarf Sniffer Zusätzlich wird die Zeit gemessen, die der Sniffer selbst benötigt (vom Start des Sendens bis zum Antworten an *Process Simulate*, abzüglich der Zeit des RCS Moduls). Mit Hilfe dieses Zeitbedarfs kann der Kommunikationsoverhead der Implementierung eingeschätzt werden.

Zu beachten ist das hier der Worst Case mit aktiviertem Logging gemessen wird. Der Sniffer speichert alle Nachrichten binär zur späteren Analyse und verwendet dafür etwa $\frac{4}{5}$ der Zeit im IOWait. Für den produktiven Einsatz kann das Logging deaktiviert werden.

Zeitbedarf *Process Simulate* Außerdem wird die Zeit gemessen, die *Process Simulate* zum Senden der darauf folgenden Nachricht benötigt. Dies beruht auf der Annahme, *Process Simulate* starte sequenziell mit der Generierung der nächsten Nachricht nach dem Empfangen des Ergebnisses der vorherigen. Wenn *Process Simulate* diese Generierung parallelisiert durchführt, dann ist dieser Wert nicht nutzbar. Da *Process Simulate* und besonders der *Process Simulate Core* selten parallelisiert und komplett Single Core basiert entwickelt wurde, ist dies jedoch unwahrscheinlich.

Ist der Zeitbedarf von *Process Simulate* im Verhältnis zum RCS Modul minimal, kann die angestrebte Reimplementierung keine signifikante Beschleunigung einer Simulation durch sequenzielle Aufrufe erreichen und ein größeres Optimierungspotenzial muss gefunden werden. Ein möglicher Ansatz mit größerem Optimierungspotenzial ist die nebenläufige Simulation der Varianten.

Fazit Zeitverhalten In Abbildung 3.5 ist das Verhältnis des Zeitbedarfs bei der Simulation eines Beispielpfades in *Process Simulate* zwischen den drei zuvor genannten Kommunikationsteilnehmern zu sehen. *Process Simulate* benötigt in der gesamten Kommunikation den größten Anteil der Zeit. Der Sniffer benötigt etwa das Doppelte des RCS Moduls an Zeit. Zu beachten ist aber, dass der Zeitbedarf der Kommunikationskomponente im Sniffer auf etwa $\frac{1}{5}$ gesenkt werden kann, wenn das Logging deaktiviert wird. Um die Kommunikation zu beschleunigen, sollte bevorzugt der Kommunikationsteilnehmer mit dem höchsten Zeitaufwand optimiert werden. Dies ist, wie zuvor bereits vermutet, *Process Simulate*.

Zusätzlich zu den verwendeten Nachrichten an das RCS Modul und dessen Relevanz für die Reimplementierung wurde nun aufgezeigt, dass *Process Simulate* die mit Abstand langsamste Komponente während der Simulation darstellt. Bevor auf die Reimplementierung in Kapitel 4 eingegangen wird, werden im Folgenden noch festgestellte Anomalien bei der Interaktion erläutert und in Unterabschnitt 3.2.4 „Retrospektive (Sniffer)“ das Vorgehen der Analyse mit dem Sniffer bewertet.

3.2.3.3 Anomalien

Während der Analyse sind an mehreren Stellen Nachrichten aufgefallen, die sich von der RRS Spezifikation oder anderen RCS Modulen unterscheiden. Nachrichten die sich je nach Roboter Hersteller / RCS Modul unterscheiden, aber der Spezifikation entsprechen, wurden bereits in Unterabschnitt 3.2.3 „Nachrichten Analyse“ erwähnt. Im Folgenden werden zwei Anomalien betrachtet, die von der RRS Spezifikation so nicht vorgesehen sind, aber während der Bearbeitung dieser Arbeit nie zu Fehlern oder Abstürzen führten.

NumberOfMessages Bei Methoden, die zusätzliche Textinformationen an das CAR Tool liefern können, ist in der Antwort vom RCS Modul ein Parameter „NumberOfMessages“ hinterlegt. Mit diesem Wert wird die Anzahl der Nachrichten, die abgerufen werden können, an das CAR Tool gemeldet. In der RRS Spezifikation Version 1.1 [1] ist jedoch nur an einer Stelle im Nebensatz definiert, dass die Nachrichten 1 basiert abgelegt werden.

Eine der Methoden, die zusätzliche Nachrichten hinterlegen kann, ist *INITIALIZE*. Die verwendeten RCS Module von ABB haben in den beobachteten Simulationen dabei NumberOfMessages = 16 zurück gegeben. Laut Spezifikation können nun vom CAR Tool die Nachrichten von 1 bis 16 abgefragt werden. *Process Simulate* fragt jedoch die Nachrichten von 0 bis 15 ab, erhält bei Nachricht 0 ’MessageNumberNotFound’ als Status zurück und die Nachricht 16 wird nie abgefragt. Das Phänomen kann in Listing 3.5 beobachtet werden. Dies scheint jedoch zu keinen Fehlern in der Simulation zu führen. Eine manuelle Anpassung der Nachrichten ID mithilfe des Sniffers auf die korrekten Werte, führe zu keinem anderen Verhalten.

Interessanterweise ist dieses Verhalten der 0 statt 1 basierten Nachrichten IDs in *Process Simulate* nur nach der *INITIALIZE* Methode zu beobachten. Nach allen anderen Methoden, bei denen es beobachtet wurde, ist dies korrekt und 1 basiert implementiert.

Das verwendete KUKA RCS Modul hat nach dem *INITIALIZE* Aufruf immer NumberOfMessages = 0 zurück gegeben, daher konnte dieser Fehler dort nie beobachtet werden. In Unterabschnitt 3.2.3 „Nachrichten Analyse“ wurde bereits festgestellt, dass sich die Implementierung in *Process Simulate* je nach RCS Modul teilweise unterscheidet. So könnte dies

ein Fehler in der Implementierung für ABB RCS Module sein, aber auch generell für alle RCS Module falsch implementiert worden sein.

Diese Anomalie wird im Laufe dieser Arbeit nicht weiter betrachtet. Nach Abschluss dieser Arbeit wird ein Bug Report von der Firma ICARUS Consulting GmbH, die diese Arbeit betreut, an Siemens gemeldet.

„AuxDataMap“ Um Daten im RCS Modul vom CAR Tool aus zu speichern oder zu modifizieren, existieren die beiden Methoden *GET_RCS_DATA* und *MODIFY_RCS_DATA*. Mit Hilfe dieser Methoden setzt *Process Simulate* in den beobachteten Simulationen mit einem ABB Roboter beispielsweise die Masse des am Roboter angebrachten Werkzeugs, dessen Schwerpunkt und dessen Koordinaten an das RCS Modul.

Während der Initialisierung des RCS Moduls fragt *Process Simulate* mit *GET_RCS_DATA* nach dem Wert von „AuxDataMap“. Das verwendete RCS Modul von ABB antwortet darauf mit 'Successful', allerdings enthält die Antwort, entgegen der Spezifikation, keine weiteren Parameter, die den Wert enthalten müssten. *Process Simulate* setzt die Simulation fort und scheint diese, nicht spezifikationskonforme Antwort zu ignorieren.

Um diese Anomalie genauer zu untersuchen, wurde der Wert von „AuxDataMap“ mit *MODIFY_RCS_DATA* gesetzt und erneut abgefragt. Das Schreiben des Wertes von „AuxDataMap“ ist möglich, das Auslesen danach enthält weiterhin eine leere Antwort mit dem Status 'Successful'.

Die Methoden *GET_RCS_DATA* und *MODIFY_RCS_DATA* haben einen Parameter, der die Art des verwendeten Speichers (InMemory oder OnStorage) angibt. Normalerweise antwortet das RCS Modul von ABB bei OnStorage mit einer Fehlernachricht mit dem Status 'StorageNotSupported'. Bei „AuxDataMap“ werden diese Nachrichten weiterhin mit einer leeren Nachricht mit dem Status 'Successful' beantwortet.

Vermutlich wurde im RCS Modul von ABB kurz nach dem Annehmen von *GET_RCS_DATA* eine Abbruchbedingung für „AuxDataMap“ eingerichtet, die eine leere, erfolgreiche Antwort verschickt. Der Grund hierfür ist unklar.

Diese Anomalie wird im Laufe dieser Arbeit ebenfalls nicht weiter betrachtet.

Als letzten Schritt vor der Umsetzung der Reimplementierung wird das Vorgehen der Analyse mit dem Sniffer bewertet.

3.2.4 Retrospektive (Sniffer)

Im Rückblick kann der Sniffer als Wahl zur Analyse, die Umsetzung des Sniffers und die Umsetzung der Analyse mit dem Sniffer bewertet werden.

Zur Analyse hat sich der Ansatz mit dem Sniffer als gut geeignet bestätigt. Dieser Ansatz kann zuverlässig und ohne das Anpassen anderer Kommunikationsteilnehmer, außer im vorgesehenen Rahmen der normalen Benutzung, analysieren und diese so im Produktivzustand bewerten.

Die Umsetzung des Sniffers wurde zu Beginn unter Einbeziehung der später in der Reimplementierung benötigten Kommunikationskomponente begonnen. Dafür wurden die Nachrichten aus den Binärdaten eingelesen und in ein menschenlesbares Format übersetzt. Das menschenlesbare Format wurde dann an die wiederverwendbare Kommunikationskomponente gegeben und binär gesendet. Durch dieses Vorgehen wurde die Kommunikationskomponente, die einen großen Teil der Reimplementierung umfasst, früh fertig gestellt. Der größte Nachteil dieses Vorgehens war jedoch das Übersetzen der Nachrichten, statt einer direkten Durchleitung. Durch Programmierfehler beim Einlesen im Sniffer oder in der Kommunikationskomponente wurde die Interaktion manipuliert und falsch fortgesetzt. Diese semantischen Fehler ließen sich nur durch das Nachvollziehen des Nachrichtenzwecks im Vergleich zur tatsächlichen Antwort herausfinden. Ein erkennen dieser Fehler im Quellcode war nur schwer möglich.

Daher wurde im Laufe dieser Arbeit der Sniffer umgestaltet, sodass dieser die Nachrichten direkt durchleitet und dabei loggt. Ein weiteres Analysetool liest später die Binärlogs ein, analysiert sie und erstellt so in einem zweiten, dem Sniffer nachgelagertem Schritt menschenlesbare Logs. Dieser Ansatz hat zusätzlich den Vorteil, bei neuen Versionen des Analysetools nicht die Kommunikation erneut laufen lassen zu müssen. Diese Änderung auf zwei separate Tools sollte in Zukunft von Beginn an ähnlich gehandhabt werden.

Die Analyse mit dem Sniffer durchzuführen war ebenfalls positiv. Die Zeitmessungen und Analysen der produktiv verwendeten Interaktionen lassen sich einfach durchführen. Das Logging der Binärdaten führt zu einer auch später noch nachvollziehbaren Kommunikation.

Basierend auf der Analyse des verwendeten RRS Protokolls kann nun mit der Reimplementierung, hin zu einem eigenen Client, begonnen werden.

4 Reimplementierung

Basierend auf dem im vorherigen Kapitel gesammelten Wissen, kann die Reimplementierung durchgeführt werden. Zur Durchführung dieser werden noch einige Fragen geklärt, die sich speziell auf die Reimplementierung beziehen und nicht auf das generelle Protokoll, wie es zuvor analysiert wurde.

In Abschnitt 4.1 „Eingangsdaten beschaffen“ werden die Wege, um an die Eingangsdaten der Simulation zu gelangen, diskutiert. Außerdem muss die Reimplementierung später auf Korrektheit der Simulationsergebnisse, sowie der Erfüllung des Beschleunigungsziels analysiert werden. Ansätze hierfür werden in Abschnitt 4.2 „Validierung der Reimplementierung“ betrachtet. Darauf folgend wird das Vorgehen bei der Implementierung in Abschnitt 4.3 „Vorgehen bei der Implementierung“ erläutert. In Abschnitt 4.4 „Bewertung der Reimplementierung“ wird der zuvor angedachte Ansatz zur Validierung angewendet und die Implementierung bewertet. Zum Schluss wird das Vorgehen bei der Reimplementierung in Abschnitt 4.5 „Retrospektive“ bewertet und es folgt ein Ausblick auf zukünftige Möglichkeiten für den neu geschaffenen RCS Client in Abschnitt 4.6 „Ausblick“.

4.1 Eingangsdaten beschaffen

Um realitätsnahe Simulationen, so wie sie aktuell mit *Process Simulate* durchgeführt werden, mit dem eigenen Client durchzuführen, müssen die selben Simulationsdaten wie aus *Process Simulate* verwendet werden können.

Dafür bieten sich die Roboterdownloads an. Als Roboterdownloads werden die Roboterprogramme bezeichnet, die auf den Roboter geladen (downloaded) und vom Roboter ausgeführt werden. Eine Alternative dazu stellt ein *Process Simulate* Plugin dar, welches die Daten aus *Process Simulate* ausliest und an den eigenen Client weiter reicht, statt über die *Process Simulate Core* Implementierung zu laufen.

Roboterdownload Der größte Vorteil der Roboterdownloads ist die Unabhängigkeit von *Process Simulate*, da die jeweilige Programmiersprache des Roboters genutzt werden kann. Ein Nachteil dabei ist aber, dass die Programmiersprachen sowohl vom Roboterhersteller abhängen,

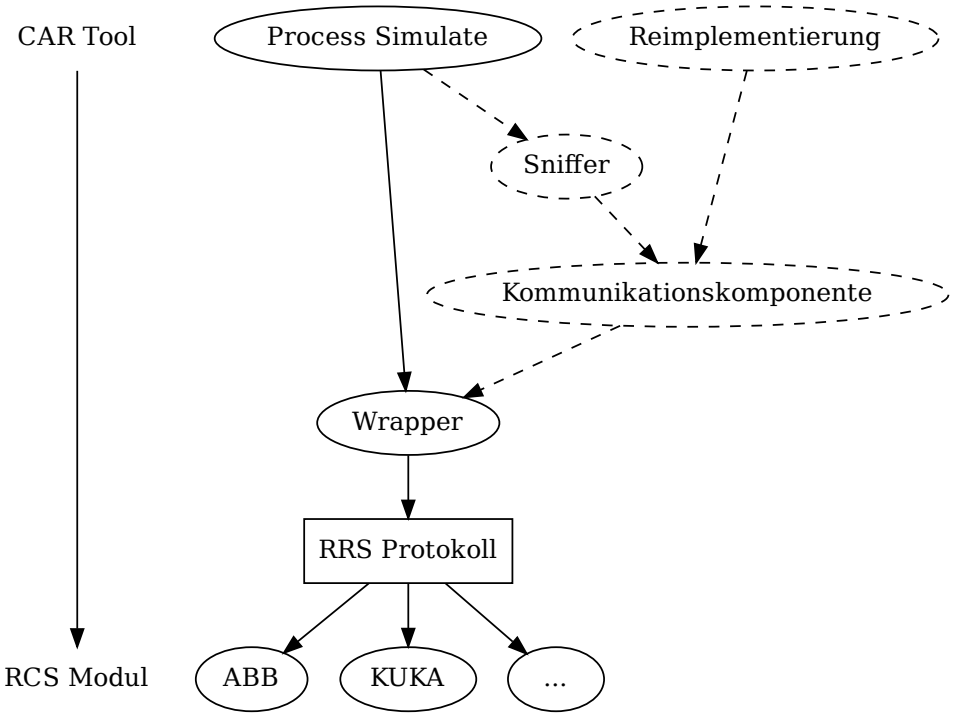


Abbildung 4.1: Aufbau der Reimplementierung und des Sniffers im Vergleich zu *Process Simulate*

aber auch dessen Syntax zwischen den einzelnen Versionen der Programmiersprachen / Roboter stark variieren kann. Die Empfehlung von der ICARUS Consulting GmbH hierfür war, mit einem ABB Programm die Durchstichimplementierung durchzuführen, da diese einen, im Vergleich zu anderen Herstellern, einfachen Aufbau haben.

Um die Machbarkeit dieses Ansatzes zu testen, wurde ein rudimentäres Tool zum Einlesen der in der Analyse hauptsächlich verwendeten Produktivpfade implementiert. Dabei wurde festgestellt, dass zum Einlesen nur der zwei meist genutzten Roboterprogramme bereits jeweils eigenständige Implementierungen benötigt werden.

Außerdem werden für viele Werte Variablen verwendet, die aus der Roboterconfig geladen werden müssten. Die Roboterconfig beinhaltet viele Dateien für unterschiedliche Zwecke, die nicht versionsübergreifend gleich sind. Die Roboter-Controller der zwei meist genutzten Roboterprogramme haben bereits eine unterschiedliche Dateistruktur, sodass das Auslesen der Werte der jeweiligen Variablen eine komplexere Einlesefunktion benötigt. In den untersuchten Roboterkonfigurationen wurde jedoch immer der intuitive Wert für den dazugehörigen Variablennamen verwendet. So beinhaltet beispielsweise die Variable für die Geschwindigkeit *v800*, die in mm/s angegeben wird, den Wert 800. Daher wird, um es einfach zu halten, der String des Variablennamen zum Wert geparst, statt den realen Wert aus der jeweiligen Roboterconfig einzulesen. Bei nicht auf diese Weise einlesbaren Werten wurden die relevanten Werte als Konstanten aus der jeweiligen Roboterconfig in den Quellcode übertragen.

***Process Simulate* Plugin** Anders als bei den Roboterdownloads bietet ein *Process Simulate* Plugin [5] keine vollständige Unabhängigkeit von *Process Simulate*. Ein deutlicher Vorteil ist aber die roboterunabhängige API mit Key Value Paaren von Einstellungen. Die Keys der Variablen sind zumindest bei allen untersuchten ABB Roboterprogrammen gleich. Die Variablen enthalten jedoch ebenfalls die Variablennamen, sodass diese, wie im vorherigen Abschnitt beschrieben, aus den Variablennamen angenommen werden. Durch die roboterunabhängige API ist dieser Ansatz deutlich schneller zu implementieren als das Einlesen der jeweiligen Roboterdownloads.

Da es in dieser Arbeit um die Interaktion und Kommunikation und deren Beschleunigung gehen soll und die Herkunft der Daten dafür nicht von Bedeutung ist, wird der Ansatz des *Process Simulate* Plugins verfolgt. Für eine Produktivimplementierung muss die Import Komponente jedoch ausgetauscht werden, um unabhängig von Process Simulate genutzt zu werden.

4.2 Validierung der Reimplementierung

Zur Validierung der Reimplementierungen des RCS Clients muss sowohl die semantische Korrektheit, als auch das Erreichen des Beschleunigungsziels überprüft werden. Da die Reimplementierung die selben Ergebnisse erzielen können soll, muss dies vergleichbar zu der Referenz von *Process Simulate* sein.

Um auf semantische Korrektheit zu überprüfen, wird das Simulationsergebnis mit der korrekten Simulation verglichen. Hierfür wird die jeweilige, vom Sniffer mitgeschnittene Simulation aus *Process Simulate* verwendet. Da *Process Simulate* bereits viele Jahre im produktiven Umfeld zur Simulation von Roboterprogrammen genutzt wird, wird von der Richtigkeit dieses Tools ausgegangen.

Da mit einer Minimalimplementierung begonnen wird, mehr dazu in Abschnitt 4.3 „Vorgehen bei der Implementierung“, wird sich sowohl der Inhalt, als auch der Umfang der Nachrichten von denen einer vollständigen Interaktion unterscheiden. Auch sorgen floating point Ungenauigkeiten für erschwerte Vergleichbarkeit der Logs. Um die Unterschiede zwischen den Simulationen sichtbar zu machen und diese damit vergleichen zu können, werden die Geschwindigkeiten, Beschleunigungen und der Ruck des Roboter Toolframes über den Simulationsverlauf der jeweiligen Simulationen geplottet. Die Plots werden verglichen und es wird versucht, Unterschiede möglicher, noch nicht genutzter Nachrichten zuzuordnen. Dabei wird die Erfahrung genutzt, welche Einstellungen, die mit jeweiligen Nachrichten gesetzt werden, welche Auswirkungen auf den Roboter in der Simulation haben bzw welche Auswirkungen ein Fehlen dieser Einstellungen haben kann und ob diese in den Plots erkennbar ist.

Um einen Referenzplot als Vergleichsbasis zu generieren, kann das *Process Simulate* Tool „Robot Viewer“, zu sehen in Abbildung 4.2, verwendet werden. Dieses wird im Umfeld von *Process Simulate* verwendet, um diese Werte für den Anwender bereitzustellen. So kann die Korrektheit der eigenen Plots sichergestellt werden.

Um das Erreichen des Beschleunigungszieles zu validieren, werden die zur Simulation benötigten Zeiten des jeweiligen CAR Tools miteinander verglichen. Da sowohl im Sniffer, als auch in der Reimplementierung die selbe Kommunikationskomponente, sowie das selbe RCS Modul verwendet wird, können die dafür gemessenen Zeiten vernachlässigt werden. Der Nachrichtenfluss der Reimplementierung ist in Abbildung 4.1 zu sehen. In dieser Arbeit entwickelte Tools und Nachrichtenflüsse sind gestrichelt eingezeichnet.

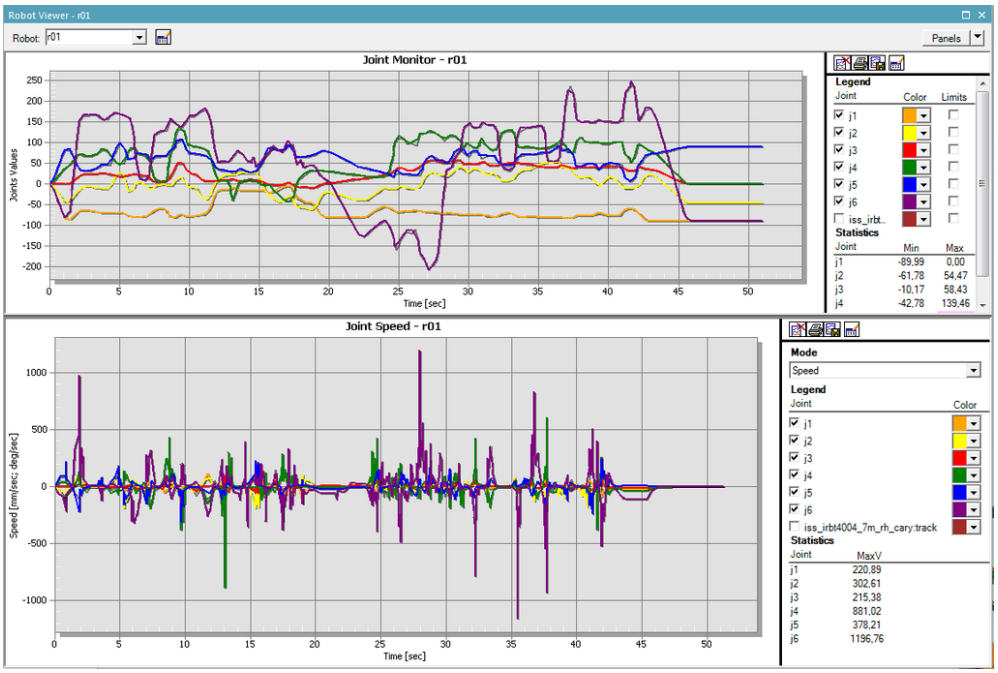


Abbildung 4.2: Das *Process Simulate* Tool „Robot Viewer“ in der Ansicht von Roboterachswerten und dessen Geschwindigkeiten im Verlauf einer Simulation

4.3 Vorgehen bei der Implementierung

Um früh semantische Fehler oder Abweichungen in der Simulation zu erkennen, wird der neue Client iterativ implementiert. Mit jedem Iterationsschritt wird dabei eine zusätzliche Schwierigkeit umgesetzt und damit die Abweichung zur Referenzsimulation reduziert.

Die Priorsierung der als nächstes zu implementierenden Funktion, erfolgt zum jeweiligen Iterationsschritt. Dabei wird versucht, Differenzen zu deuten und auf bestimmte, noch nicht umgesetzte Nachrichtentypen zurückzuführen. Die größten Abweichungen werden dabei zuerst behandelt.

Um die Abweichungen der eigenen Simulation zur Referenzsimulation erkennen zu können, werden die Geschwindigkeiten des Roboters, wie in Abschnitt 4.2 „Validierung der Reimplementierung“ beschrieben, geplottet. Dafür wird sowohl das Tool Center Point Frame (TCPF) in Abbildung 4.3 geplottet, als auch die Achsstellungen der 6 Roboterachsen (Joints 1 bis 6) in Abbildung 4.4. Für die Wahl der im jeweiligen Iterationsschritt umzusetzenden Schwierigkeit werden alle Plots betrachtet. Im Folgenden werden jedoch nur die ausschlaggebenden Plots gezeigt.

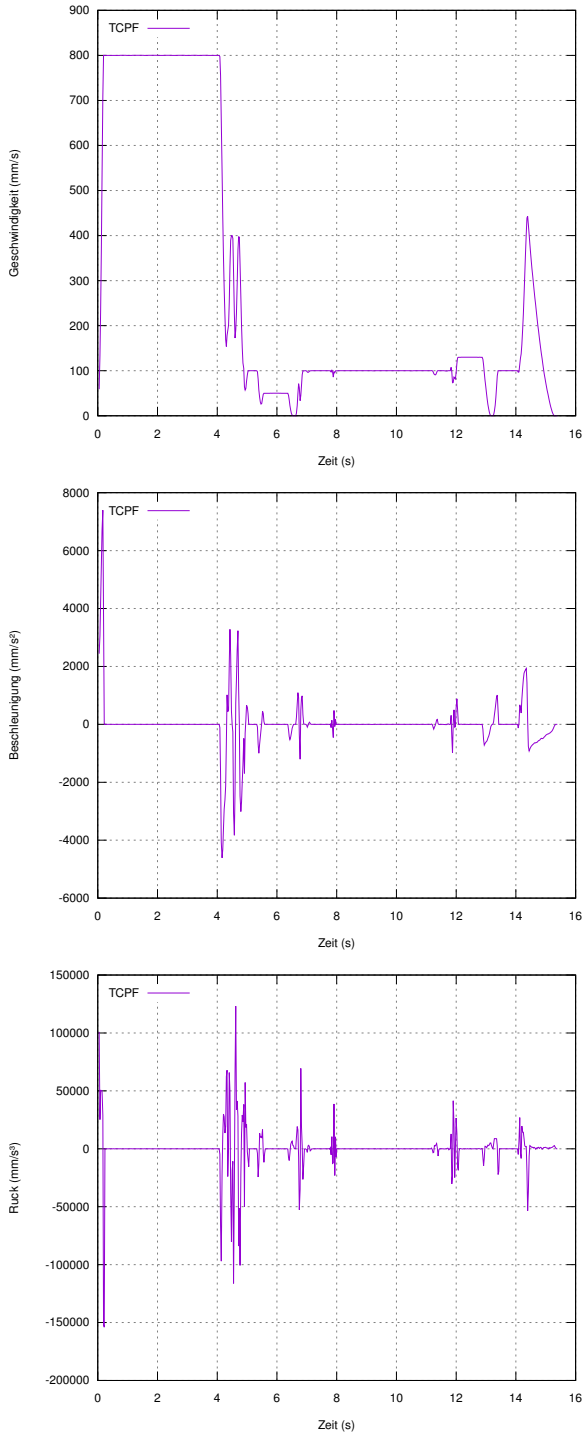


Abbildung 4.3: Geschwindigkeiten des Roboter Toolframes (TCPF) des Referenzpfades, aufgenommen mit dem Sniffer aus *Process Simulate*

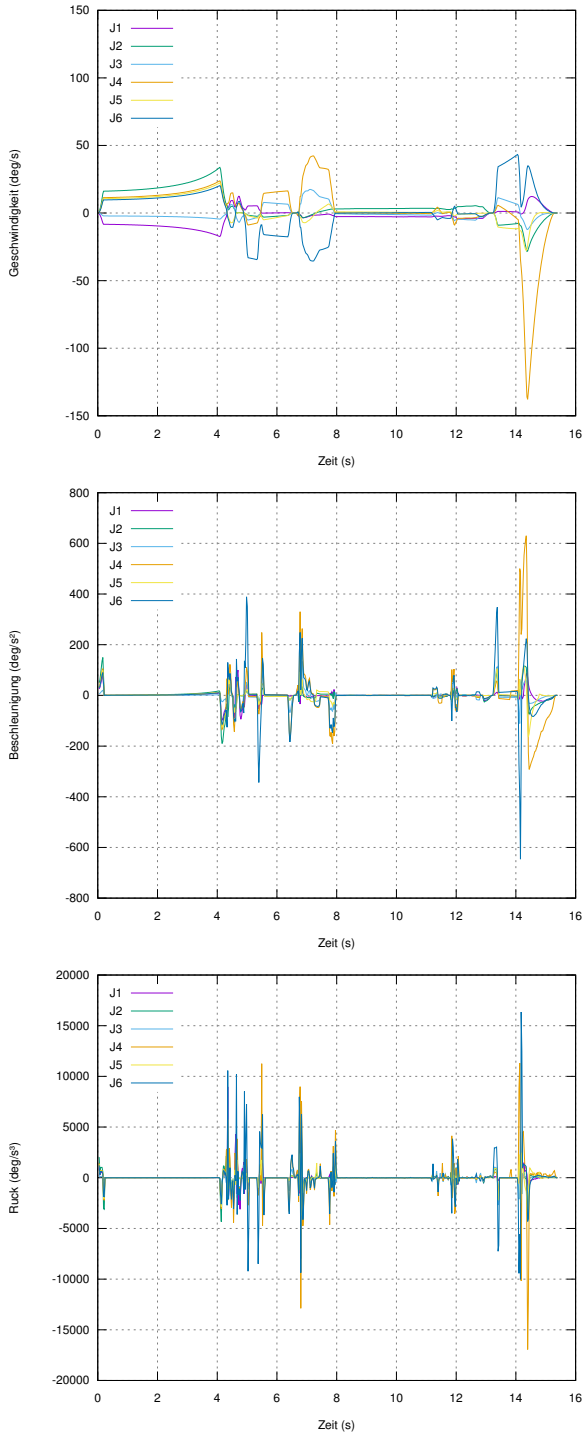


Abbildung 4.4: Geschwindigkeiten der 6 Roboter Achsen (Joints 1-6) des Referenzpfades, aufgenommen mit dem Sniffer aus *Process Simulate*

In Unterunterabschnitt 3.2.3.1 „Nachrichten Zweck“ werden die benötigten Nachrichten für die Reimplementierung analysiert. Diese sind nochmals in der Reihenfolge der Wichtigkeit für die Implementierung aufgeführt:

- Zur minimalen Implementierung wird nur *INITIALIZE* zwingend benötigt. Da nur mit der Initialisierung keine Roboterbewegung stattfinden wird, wird dieser Schritt später nicht genauer erwähnt.
- Um die Positionen der Pfadlocations zu setzen und diese anzufahren werden *SET_NEXT_TARGET* und *GET_NEXT_STEP* verwendet. Außerdem muss die initiale Position, von der der Roboter losfahren soll, mit *SET_INITIAL_POSITION* gesetzt werden. Da der Roboter eigene Bezugssysteme verwendet, müssen vermutlich auch die Bezugssysteme (Work Frames) gesetzt werden, weshalb vermutlich auch *SELECT_WORK_FRAMES* und *MODIFY_CELL_FRAME* benötigt werden.
- Um die Art der Roboterbewegung zu definieren, wird *SELECT_MOTION_TYPE* verwendet.
- Damit sich der Roboter an Geschwindigkeitslimits halten kann, die zum Beispiel für gleichmäßige Lackierungen notwendig sind, werden *SET_CARTESIAN_POSITION_SPEED* und *SET_CARTESIAN_ORIENTATION_SPEED* verwendet. Für die maximale Beschleunigung und den maximalen Ruck werden *SET_JOINT_ACCELERATIONS* und *SET_JOINT_JERKS* benötigt.
- Für den Überschleif (Flyby) werden *SELECT_FLYBY_MODE* und *SET_FLYBY_CRITERIA_PARAMETER* benötigt.
- *SET_CONFIGURATION_CONTROL* wurde bei der Beobachtung mit dem Sniffer immer gleich verwendet. Dies wird nach der minimalen Implementierung als Teil der Initialisierung hinzugefügt und nicht weiter erwähnt. Zu Testzwecken wurde die Nachricht bei jeder Pfadlocation gesendet, was keine Änderung hervorgerufen hat. Die Nachricht braucht also bei diesem Pfad nur zu Beginn gesendet werden.
- Mit *GET_RCS_DATA* und *MODIFY_RCS_DATA* wurden zu Beginn der Simulation lediglich die Tooldaten übermittelt (Masse, Masseschwerpunkt, …). Dies wird direkt zu Anfang nach der minimalen Implementierung als Teil der Initialisierung hinzugefügt und nicht weiter erwähnt. Zu Testzwecken wurden diese Nachrichten nach der vollständigen Implementierung entfernt und eine Abweichung war festzustellen. Demnach werden die Tooldaten vom RCS Modul berücksichtigt.

- *DEFINE_EVENT* und *GET_EVENT* werden in dieser Arbeit vernachlässigt.

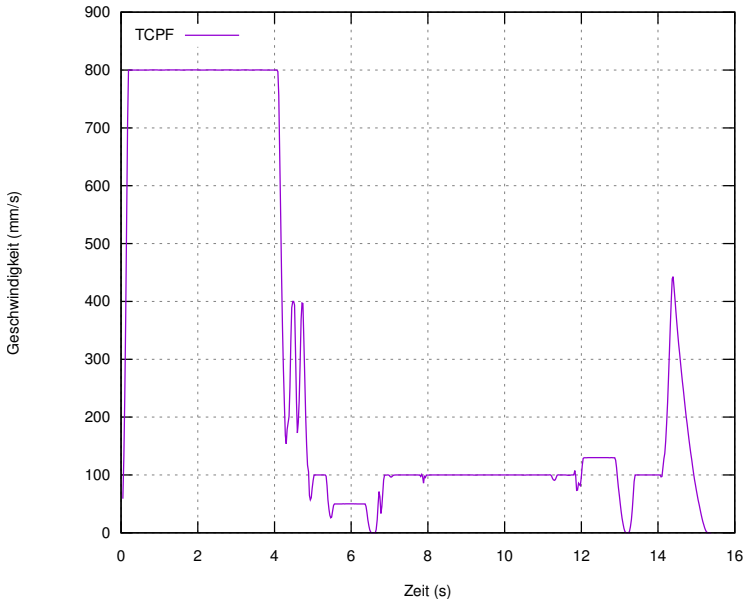
Iterationsschritt I Nachdem die minimale Implementierung problemlos funktionierte und somit die Initialisierung des RCS Moduls als Basis genutzt werden kann, werden im ersten Iterationsschritt die Positionen der Pfadlocations gesendet. In Abbildung 4.5 sind zu hohe Geschwindigkeiten im Vergleich zur Referenzsimulation zu sehen. Die maximalen Werte hierfür werden im nächsten Iterationsschritt gesetzt. Außerdem ist zu erkennen, dass zu jeder Pfadlocation die Geschwindigkeit auf 0 abfällt. Dies liegt am fehlenden Überschleif, da der Roboter jede Location exakt anfahren muss. Der Überschleif wird in einem späteren Iterationsschritt gesetzt, da dieser im Vergleich zu einfachen Maximalwerten eine komplexere Umsetzung erfordert.

Iterationsschritt II Im zweiten Iterationsschritt wird die maximale Geschwindigkeit sowohl des TCPF, als auch der Roboterachsen begrenzt. In Abbildung 4.6 sind die nun zur Referenzsimulation passenden Maximalgeschwindigkeiten erkennbar. Außerdem dauert die Simulation jetzt etwas länger als die Referenzsimulation. Dies liegt wie zuvor am fehlenden Überschleif. Außerdem ist nach diesem Schritt eine zu hohe Beschleunigung in Abbildung 4.7 zu erkennen.

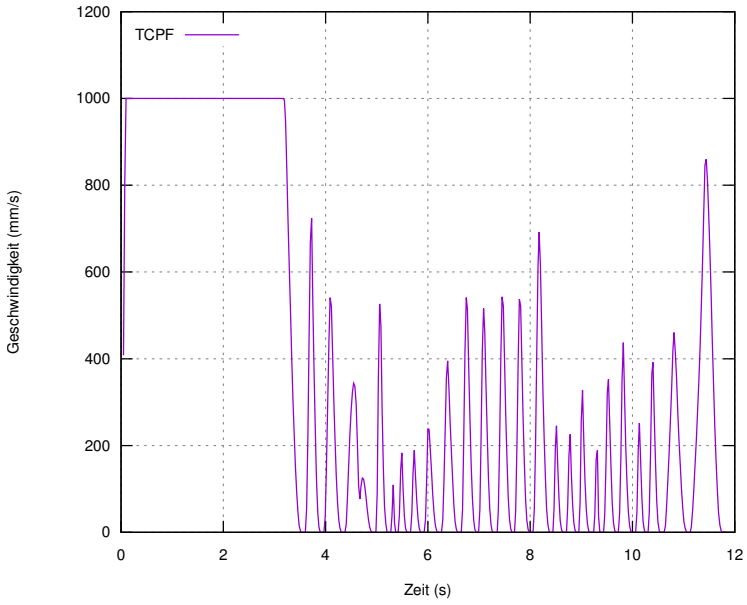
Iterationsschritt III Mit der Beschränkung der maximalen Beschleunigung und des maximalen Rucks sind diese Kurven in Abbildung 4.7 deutlich näher an der Referenzsimulation. Weiterhin ist der fehlende Überschleif in Abbildung 4.8 zu erkennen.

Iterationsschritt IV Im vierten Iterationsschritt werden dem RCS Modul ebenfalls zu jeder Pfadlocation die passenden Überschleifwerte geliefert. So fehlen nun in Abbildung 4.9 die Absenkungen der Geschwindigkeit auf 0 im Hauptteil des Roboterpfades. Die Kurve entspricht aber noch nicht der Referenzsimulation. Dies ist zum Beispiel an einem Einbruch der Geschwindigkeit etwa bei Sekunde 14 zu sehen. Außerdem benötigt der Roboter zum Abfahren des Pfades etwa eine Fünftelsekunde zum Abfahren länger als in der Referenzsimulation.

Iterationsschritt V Zum letzten Iterationsschritt wurde ein Fehler im *Process Simulate* Plugin behoben. Die Bezugssysteme relativ zum Roboter wurden mit einem Rechenfehler aus dem Weltkoordinatensystem, mit dem *Process Simulate* arbeitet, berechnet. Nach dieser Korrektur stimmen die Geschwindigkeitskurven der eigenen Simulation mit der der Referenzsimulation überein.

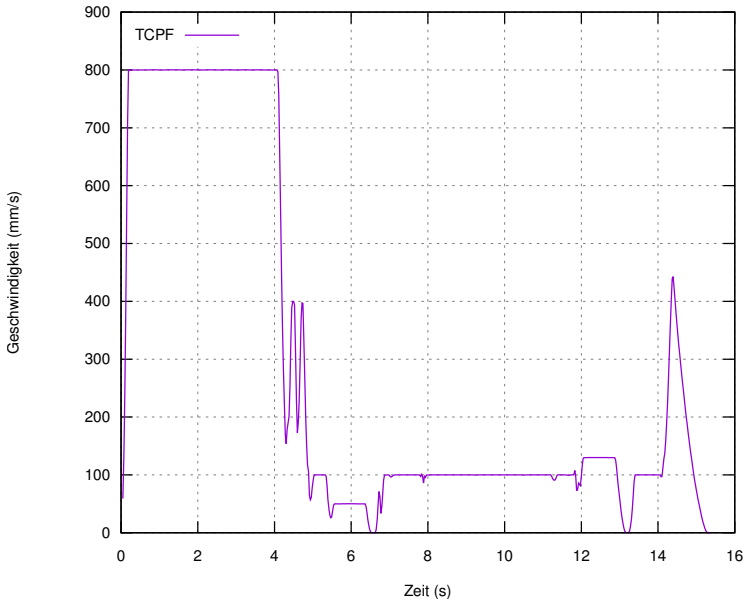


(a) Referenzgeschwindigkeitsverlauf

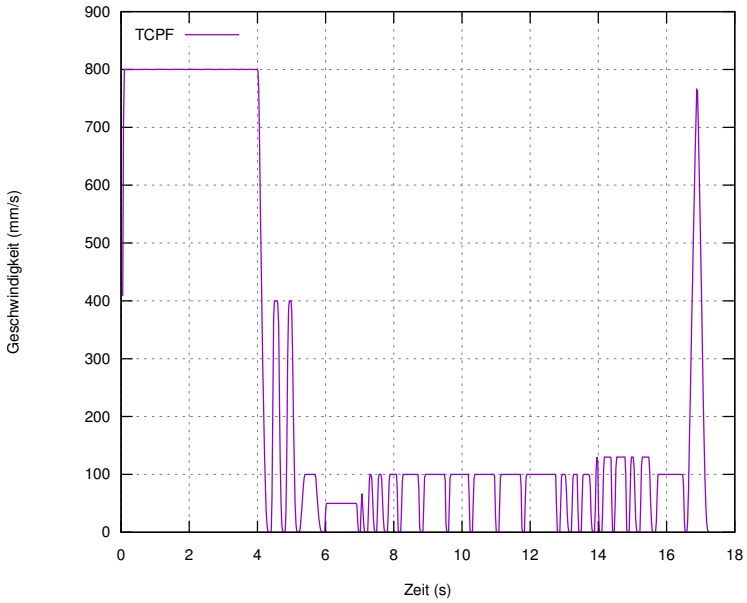


(b) nach Iterationsschritt I

Abbildung 4.5: Nur die Zielpositionen der Pfadlocations werden gesetzt

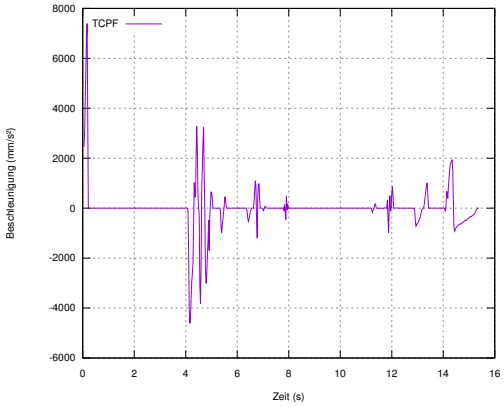


(a) Referenzgeschwindigkeitsverlauf

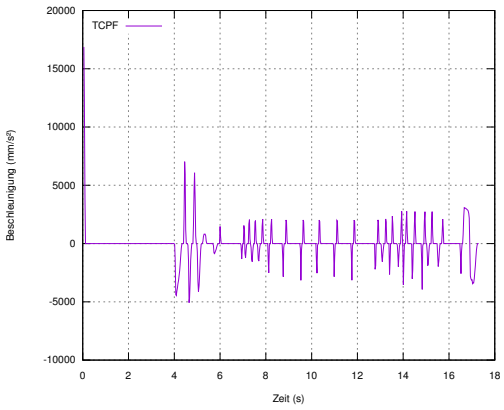


(b) nach Iterationsschritt II

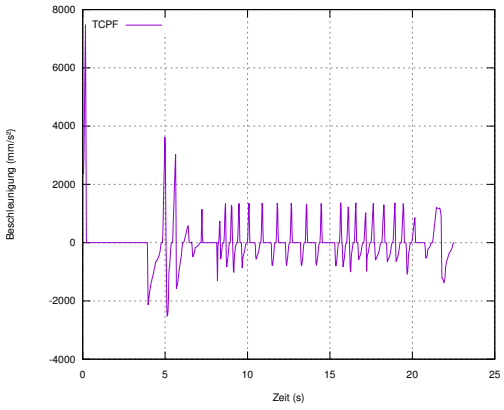
Abbildung 4.6: Die Art der Fahrbewegung und die Maximalgeschwindigkeit wurden zusätzlich angegeben



(a) Referenzbeschleunigung

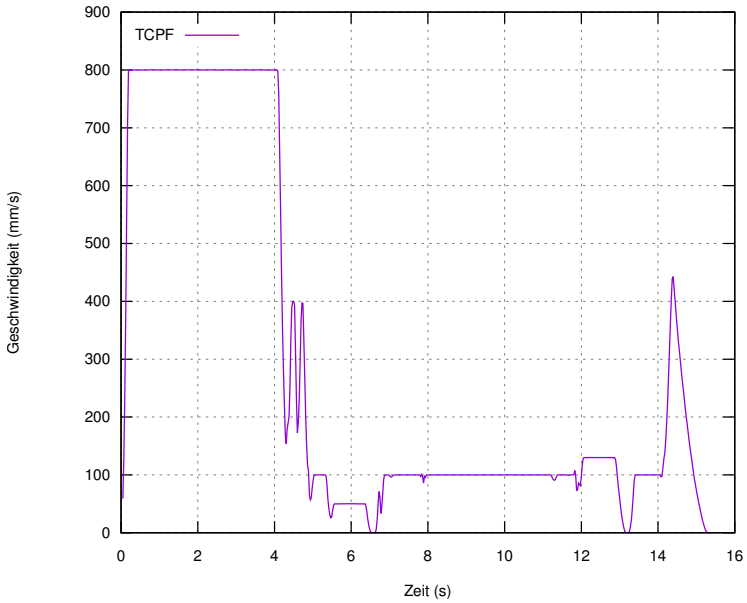


(b) nach Iterationsschritt II

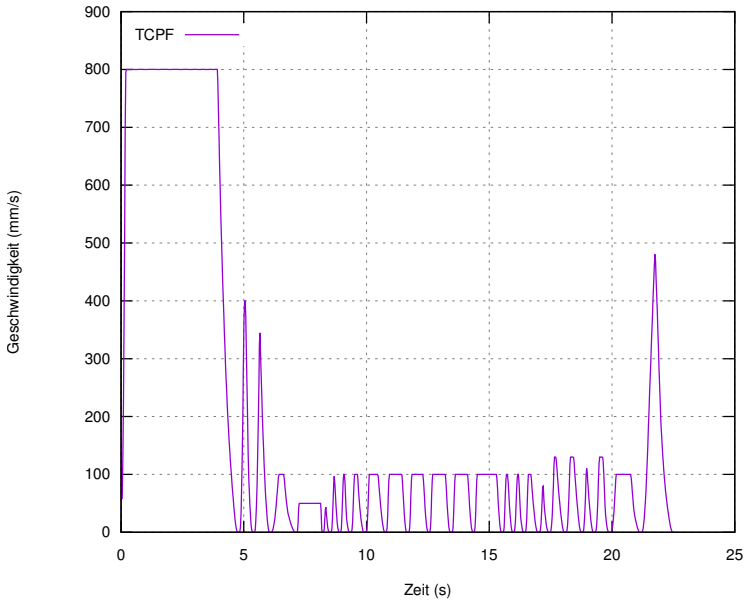


(c) nach Iterationsschritt III

Abbildung 4.7: Im Vergleich zur Referenzsimulation ist die Beschleunigung deutlich zu stark. Dies wurde mit Iterationsschritt III verbessert. (Auf Y-Achsskalierung achten)

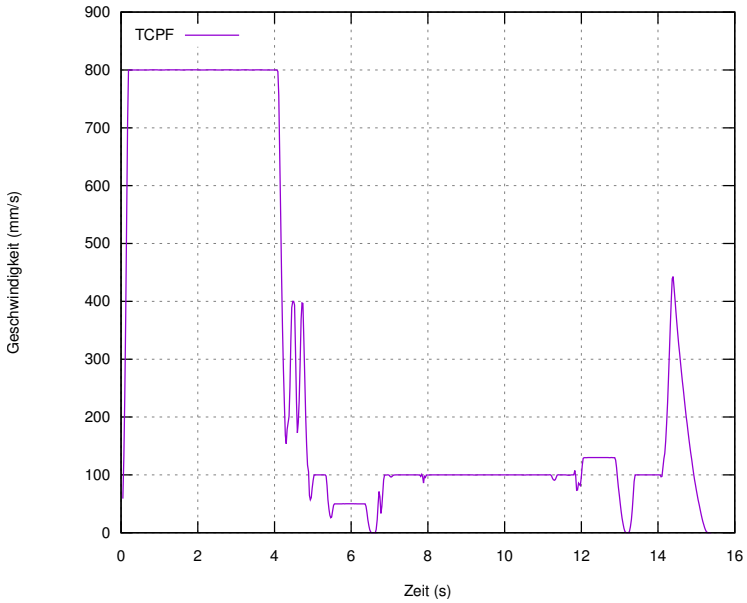


(a) Referenzgeschwindigkeitsverlauf

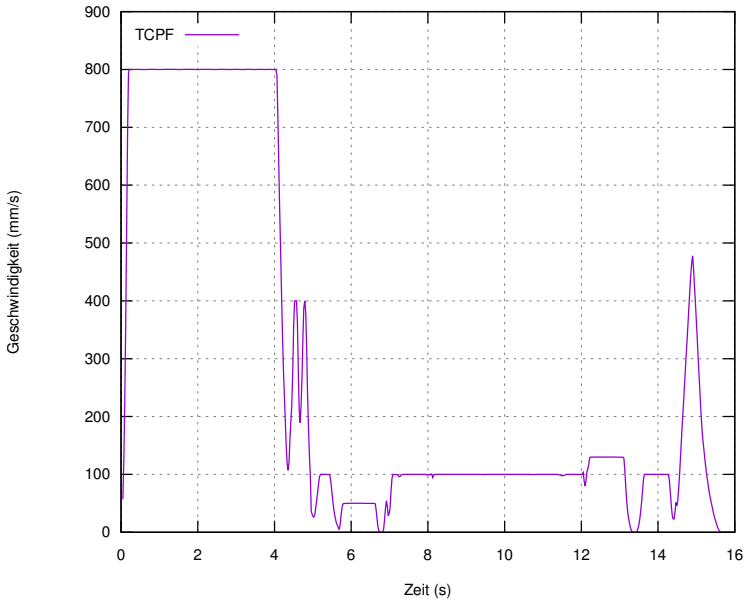


(b) nach Iterationsschritt III

Abbildung 4.8: Die maximale Beschleunigung und der maximale Ruck werden ebenfalls gesendet

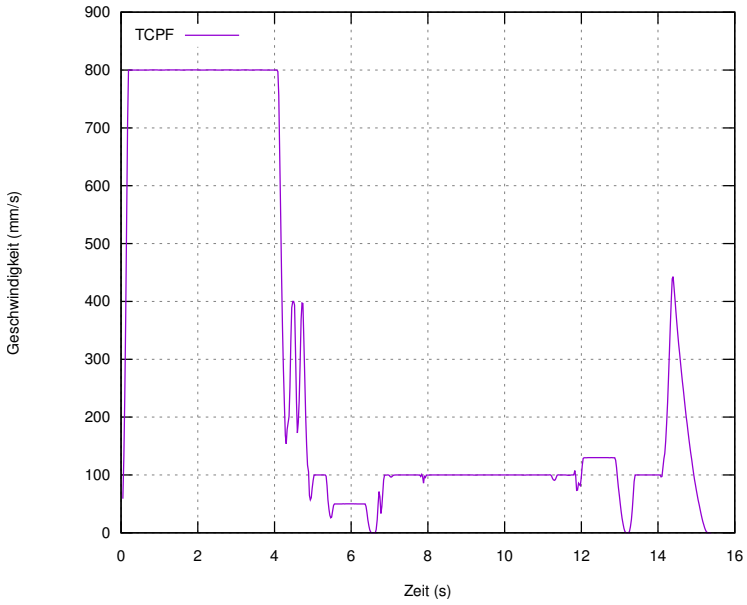


(a) Referenzgeschwindigkeitsverlauf

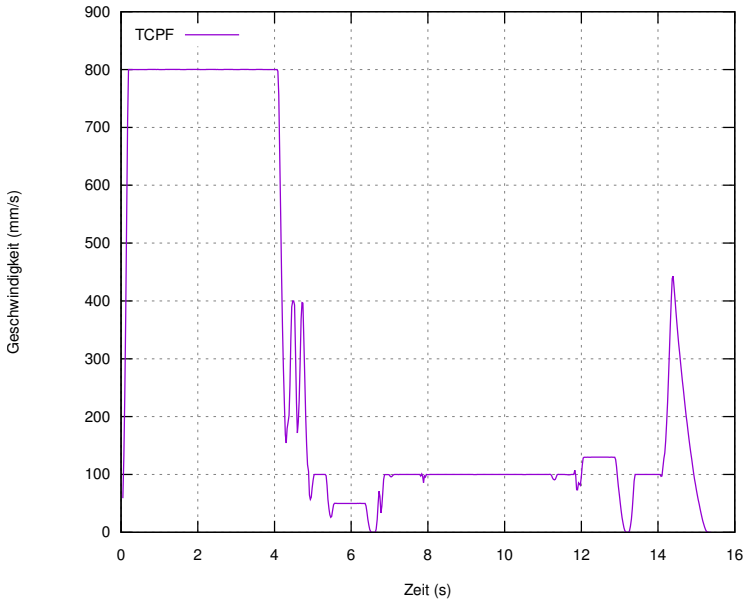


(b) nach Iterationsschritt IV

Abbildung 4.9: Wenn Überschleif mit angegeben wird, sinkt die Geschwindigkeit nicht an jeder Pfadlocation auf 0



(a) Referenzgeschwindigkeitsverlauf



(b) nach Iterationsschritt V

Abbildung 4.10: Fehler in der Einlese der Bezugssysteme aus *Process Simulate* wurde behoben

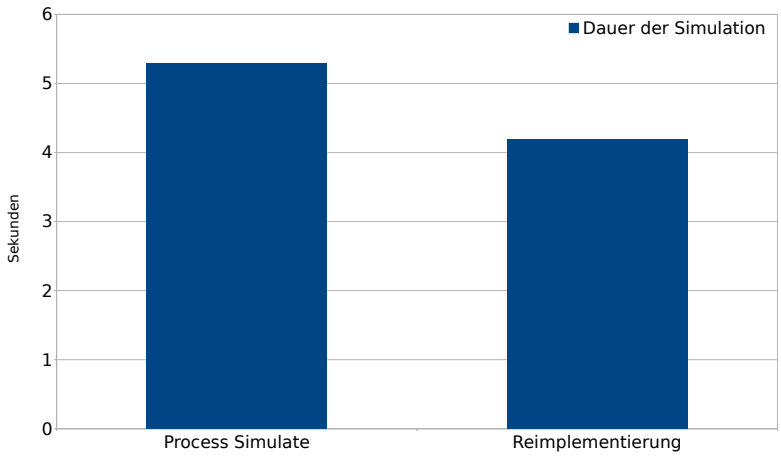


Abbildung 4.11: Vergleich der Dauer der Simulation des Referenzpfades von *Process Simulate* und der Reimplementierung

Wie in Unterunterabschnitt 3.2.3.1 „Nachrichten Zweck“ festgestellt, verwendet *Process Simulate* häufig die selben Nachrichten zum Einstellen von Parametern erneut, obwohl das RCS Modul diese Einstellungen beibehält. Dies wurde bei der Reimplementierung zum Sicherstellen einer korrekten Simulation genauso wiederholt. Zur Vollendung wurden diese, vermutlich überflüssigen Nachrichten entfernt. Das Simulationsergebnis wurde dadurch nicht beeinflusst.

Mit diesem Ansatz der Implementierung des eigenen Clients wurde bereits die semantische Korrektheit sichergestellt. Daher muss im Folgenden nur noch das Beschleunigungsziel überprüft werden, wie es in Abschnitt 4.2 „Validierung der Reimplementierung“ erläutert ist.

4.4 Bewertung der Reimplementierung

Um zu bewerten, ob die Reimplementierung schnellere Simulationen erlaubt, als sie mit *Process Simulate* durchgeführt werden können, muss das Zeitverhalten der Reimplementierung mit der von *Process Simulate* verglichen werden. Dazu wurde bereits in Unterunterabschnitt 3.2.3.2 „Zeitverhalten“ das Zeitverhalten von *Process Simulate* analysiert.

Die Simulation des Referenzpfades benötigt mit *Process Simulate* etwa 5,3 Sekunden (ohne grafische Aktualisierung). Mit der Reimplementierung werden dafür etwa 4,2 Sekunden benötigt. Die Messungen wurden jeweils 5 Mal wiederholt und der Median als Vergleichswert genutzt. Der Vergleich der Dauer ist in Abbildung 4.11 zu sehen. Dabei ist in beiden Fällen das Logging in der Kommunikationskomponente deaktiviert.

Da in beiden Fällen, der Analyse von *Process Simulate* und der Reimplementierung, die selbe Kommunikationskomponente, als auch das selbe RCS Modul verwendet wird, muss nur der

Zeitbedarf von *Process Simulate* mit dem der der Reimplementierung verglichen werden, ohne die anderen Komponenten zu beachten. *Process Simulate* benötigt im Durchschnitt für eine Nachricht in der Simulation etwa 2.8 ms. Die Reimplementierung benötigt im Durchschnitt nur jeweils 1,8 ms. Diese Beschleunigung lässt sich unter anderem dadurch erklären, dass die Reimplementierung im Gegensatz zu *Process Simulate* sämtliche Daten vor der Simulation mit dem *Process Simulate* Plugin einliest und während der Simulation nur noch senden muss. *Process Simulate* hingegen führt den Lese- und Bechnungsvorgang vermutlich während der Simulation durch. Der Einlesevorgang der Reimplementierung dauert jedoch nur etwa 30 ms, die im Verhältnis zu mehreren Sekunden Simulationsdauer minimal sind.

Ein großer Teil der Simulationsdauer konnte vor allem durch das Herausfiltern von redundanten Nachrichten gespart werden. *Process Simulate* versendet für die Simulation des Referenzpfades 1120 Nachrichten an das RCS Modul. Die Reimplementierung kommt mit 944 Nachrichten aus, was einer Reduzierung um 16 % entspricht. Würden die 26 Nachrichten für das Event Handling (*DEFINE_EVENT* und *GET_EVENT*), welches in dieser Arbeit nicht behandelt wird, ebenfalls verwendet werden, so entspräche dies einer Reduzierung um 11 %.

So verringert die Senkung der benötigten Nachrichtenanzahl, als auch die geringere Bearbeitungszeit die Dauer einer Simulation. Zusätzlich lässt sich die Reimplementierung nebenläufig nutzen. In einer prototypischen, nebenläufigen Implementierung wurden 15 Varianten des Referenzpfades simuliert. Dafür wurden 27 Sekunden benötigt, was etwa 1,8 Sekunden pro Simulation entspricht. Diese Verbesserung wird vermutlich durch das Betriebssystem Scheduling und der Verteilung auf unterschiedliche CPU Kerne erzeugt, da ein RCS Modul während seiner kompletten Laufzeit nicht die gesamten Systemkapazitäten benötigt.

In den folgenden Abschnitten wird die Vorgehensweise der Reimplementierung bewertet, sowie dessen mögliche Zukunft angesprochen.

4.5 Retrospektive

Beim Vorgehen der Reimplementierung hat vor allem die klare Trennung in Komponenten hilfreich. Das Einlesen der Eingangsdaten wurde mit dem *Process Simulate* Plugin gelöst und die Kommunikationskomponente wurde aus der Analyse übernommen. Da das *Process Simulate* Plugin klar zu Testzwecken entwickelt wurde, konnte dies viele Annahmen enthalten, die im produktiven Umfeld nicht möglich gewesen wären, aber für eine deutlich schnellere Implementierung gesorgt haben. Außerdem ist durch diese Trennung ein späterer Austausch von Komponenten leicht möglich. Die Wiederverwendung der Kommunikationskomponente führe zu einer leichten Vergleichbarkeit zwischen *Process Simulate* und der Reimplementierung.

Zudem war eine rudimentäre, nebenläufige Simulation von mehreren Varianten in kürzester Zeit implementierbar. Das von Beginn stark fokussierte Ziel der nebenläufig ausführbaren Reimplementierung hat dabei stark geholfen.

Nachteilhaft war die, in der RRS Spezifikation nicht klare Definition als RPC Protokoll. Aus diesem Grund wurde die Reimplementierung als Plattform zum Versenden der RRS Nachrichten entwickelt, statt es als RPC Abstraktion zu verwenden. Eine Entwicklung näher an der tatsächlichen RPC Struktur würde die Nutzbarkeit der Reimplementierung für Programmierer vereinfachen.

4.6 Ausblick

Für die Zukunft dieser Reimplementierung spielen zwei Aspekte eine Rolle: weitere Möglichkeiten der Beschleunigung der Simulationen und die Vervollständigung der Reimplementierung für die produktive Nutzung.

Möglichkeiten weiterer Beschleunigung Ein Ansatz um die Simulation mit den RCS Modulen noch weiter zu beschleunigen, ist das Herauskürzen der Wrapper. So könnte die Reimplementierung direkt über die C-Calls mit einem RCS Modul kommunizieren.

Ein anderer Ansatz wäre, den Initialisierungsprozess zeitlich entkoppelt, vorgelagert auszuführen. Ist beispielsweise bereits der genaue Roboter bekannt, wie es beim Starten eines Projekts in *Process Simulate* der Fall ist, kann eine Instanz der Reimplementierung im Hintergrund gestartet werden, die bereit für eine mögliche Simulation ist. Da das RCS Modul mit dem Referenzpfad etwa 2 Sekunden für den Aufruf von *INITIALIZE* benötigt, kann so die Dauer der Simulation (etwa 4 Sekunden) zum benötigten Zeitpunkt nahezu verdoppelt werden.

Produktiver Einsatz Um die Reimplementierung für den produktiven Einsatz nutzen zu können, müssen noch einige Funktionalitäten ergänzt werden, die im Rahmen der Durchstichimplementierung nicht umgesetzt wurden. So wurden beispielsweise die Events außer acht gelassen. Auch wurde die Reimplementierung nur im Zusammenhang mit ABB RCS Modulen verwendet. Zudem ist das Einlesen von Eingangsdaten nur sehr rudimentär vorhanden.

Außerdem sollte in einem Refactoring die aktuelle Verwendung auf Nachrichtenbasis in die Richtung der RPC geändert werden, um Entwicklern, die die Reimplementierung verwenden wollen, ein einfacheres Interface bereit zu stellen und die Weiterentwicklung der Reimplementierung zu vereinfachen.

5 Fazit

Die Reimplementierung konnte das angestrebte Ziel der Beschleunigung einer Simulation nicht nur sequenziell erreichen, sondern auch die Möglichkeit eröffnen, nebenläufige Simulationen durchzuführen. Im Vergleich zu der Simulation mit *Process Simulate* bieten sich daher neue Möglichkeiten, die mit den aktuell verfügbaren Tools nicht umsetzbar sind.

Darunter sind die bereits zu Beginn dieser Arbeit angestrebten Ziele, wie das Simulieren von vielen Varianten zur Optimierung von Pfaden mit der Reimplementierung statt eines *Process Simulate* Plugins. Ein Anwendungsfall wäre die Verbesserung der von Olaf Bürger in seiner Arbeit zur Glättung der Abfahrgeschwindigkeit [6] entwickelten Software. Auch weitere, für diesen Zweck entwickelte Software wäre auf Basis der Reimplementierung denkbar.

Dazu kamen aber auch neue Ziele, die während der Erstellung dieser Arbeit innerhalb der Firma ICARUS Consulting GmbH entstanden. Beispielsweise ist der Einsatz der Reimplementierung innerhalb von Assistenztools für die effektivere Arbeit mit *Process Simulate* denkbar. Auch ließe sich die Reimplementierung in Projekten einsetzen, die die Visualisierung von Roboterpfaden als Thema haben. So kann durch den Einsatz von Augmented Reality die Grenze zwischen Realität und Simulation weiter aufgelöst werden und Optimierungsprozesse an bestehenden Roboterprogrammen effizienter gestaltet werden. Das hier benötigte, direkte Feedback ist auf eine schnell ablaufende Simulation angewiesen.

Das verwendete Vorgehen mit Hilfe des Sniffers und der generalisierten Kommunikationskomponente erwies sich als größtenteils effektiv zur Erreichung des Ziels der Reimplementierung und dessen Validierung. Bei der Umsetzung des Sniffers mit der Kommunikationskomponente führten Programmierfehler jedoch zu Verfälschungen der Interaktion, weshalb der Sniffer im Laufe der Arbeit umgestellt wurde.

Literaturverzeichnis

[1] RRS-Owners, RRS Interface Specification, Version 1.1, 20. September 1995, http://www.realistic-robot-simulation.org/

[2] RRS-Owners, RRS-Interface Specification, Version 1.3, Berlin, 23. September 1997, http://www.realistic-robot-simulation.org/

[3] IPK Fraunhofer, "VRC-Specification Owners", VRC Interface Specification, Juni 2001, http://www.realistic-robot-simulation.org/

[4] Tecnomatix, Siemens PLM Solutions, Process Simulate, aufgerufen am 13.2.2017, https://www.plm.automation.siemens.com/en_us/products/tecnomatix/manufacturing-simulation/assembly/process-simulate.shtml

[5] Tecnomatix, Siemens PLM Solutions, Tecnomatix Engineering API, ausgeliefert mit der jeweiligen Tecnomatix [4] Installation, im Installationsverzeichnis unter emPower\TecnomatixSDKHelp.chm

[6] Olaf Bürger, ICARUS Consulting GmbH, Automatische Anpassung von Anfahrpunkten zur Glättung des Applikationsgeschwindigkeitsprofils von Roboterbahnen in einem Simulationsprogramm, 29.04.2016

[7] Gerald Combs and contributors, Wireshark, Version 2.2.5, 07.03.2017, https://www.wireshark.org/

[8] Gerald Combs and contributors, Wireshark Display Filter Reference, Version 2.2.5, 07.03.2017, aufgerufen am 20.03.2017, https://www.wireshark.org/docs/dfref/

[9] Microsoft Sysinternals, TCPView, Version 3.05, 25.07.2011, https://technet.microsoft.com/en-us/sysinternals/tcpview.aspx

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 16. Juni 2017 Edgar Toll