Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Automatische Tests von speicherprogrammierbaren Steuerungen durch die Ausführung modifizierbarer Aufnahmen des Maschinenverhaltens

Automatic tests of programmable logic controllers by executing modifiable recordings of the machine's behavior

Masterarbeit

im Studiengang Technische Informatik

von

Eugen Wagner

Erstprüfer: Prof. Dr. Joel Greenyer
Zweitprüfer: Prof. Dr. Kurt Schneider

Hannover, 29. Juni 2016

Abstract

Speicherprogrammierbare Steuerungen (SPS) werden heutzutage weltweit zur Automatisierung von Maschinen eingesetzt. Der stetig wachsende Bedarf an Automatisierung führt zu einer Transition von klassisch mechanischen kontrollierten Bewegungen, hin zu von Software kontrollierten. Dadurch erlebt die Software von SPSen seit Jahren einen immer größeren Zuwachs an zu bewältigenden Funktionen, was die Sicherstellung von hoher Software-Qualität erschwert. Um der Lage Herr zu werden, werden bereits vermehrt Versuche angestellt um Testtechniken aus dem Software Engineering in die Welt der Entwicklung von SPSen einzuführen, bislang jedoch nur mit mäßigen Erfolg.

Daher werden in dieser Masterarbeit vorhandene Softwarelösungen zur Testautomatisierung und Model Checking anhand erkannter Anforderungen überprüft. Es stellt sich heraus, dass vor allem aus Zeitgründen ein Mehraufwand im Entwicklungsprozess in der Automatisierungstechnik nicht zulässig ist. Diese Tatsache disqualifiziert den Großteil der aktuell vorhandenen Lösungen.

Unter Zuhilfenahme von Ideen aus anderen Gebieten, wird das *PLC Testcase Tool* vorgestellt. Ein Konzept, welches automatische Tests an Maschinen durch Aufnahme des Ein- und Ausgangsverhaltens der SPS ermöglicht. Hierfür wird während des Tests einer Funktion der Maschine, welcher ohnehin gemacht wird, der Ausgangszustand der SPS eingelesen und anschließend die Abfolge der Ein- und Ausgänge aufgenommen. Mit der dabei generierten *TestCaseRecording*-Datei ist ein Wiederabspielen inklusive Simulation des Maschinenverhaltens ohne die Maschine selbst möglich. Weiter ist der Testfall durch das Format lesbar und modifizierbar. Nach der Wiedergabe des Testfalls wird ein Protokoll erstellt welches den Testfall genau dokumentiert. Eine prototypische Implementierung wird demonstriert.

.

Eidesstattliche Erklärung

„Hiermit versichere ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen verwendet habe.“

Hannover, den 29. Juni 2016

_____

Eugen Wagner

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

IV. Gliederung der Arbeit

1. Einleitend wird über die generelle Arbeitsweise in der Automatisierungsindustrie Auskunft gegeben. Die Entwicklung bis heute wird kurz erläutert und es wird auf kommende Probleme verwiesen, welche von Autoren und Forschern aus dem Bereich der Informatik und Automatisierungstechnik prognostiziert worden sind. Die besonderen Probleme im Bezug auf Testen für Ingenieure, welche die Anlagen automatisieren, werden hierbei hervorgehoben und bilden die Problemstellung.
2. Die Grundlagen beschreiben alle Technologien und Techniken, die der Leser benötigt, um Entscheidungen und das vorgestellte Konzept verstehen zu können. Dabei wird angenommen, dass der Leser ein Student der Informatik bzw. der Technischen Informatik ist.
3. Mit dem Referenzbeispiel in diesem Kapitel soll dem Leser geholfen werden, die nachfolgenden Konzepte besser verstehen zu können.
4. Während der Anforderungsanalyse werden dem Leser zuerst die Aufgaben und der Aufgabenbereich eines Ingenieurs für Automatisierungstechnik deutlich gemacht um daraus die Bedürfnisse ableiten zu können. Zusammen mit den aus der Industrie kommenden Wünschen, ergibt sich daraus eine Anforderungsliste.
5. Da das Streben nach Verbesserung der Software-Qualität in der Automatisierungsindustrie nicht neu ist, wurden bereits einige Lösungen erarbeitet. Diese werden im Kapitel „Stand der Technik“ vorgestellt und anhand der erkannten Anforderungen bewertet.
6. Im Kapitel Konzept wird ein Vorschlag zur Lösung der in der Anforderungsanalyse erkannten Probleme beschrieben und ein prototypisches Software Werkzeug vorgestellt, welches dabei assistiert.
7. Anhand von Anwendungsbeispielen wird gleichzeitig die Funktionalität des Prototyps demonstriert. Es werden die Rahmenbedingungen eingehalten und erklärt um aufzuzeigen weshalb einige der benötigten Anforderungen nur schwer zu realisieren sind.
8. In einem Kapitel mit Fazit und Ausblick wird die Arbeit abgeschlossen und einige Vorschläge für künftige Erweiterungen des Tools gegeben.

V. Glossar

Ingenieur

Berufsbezeichnung. Wird in dieser Arbeit sowohl für männliche als auch weibliche Ingenieure verwendet. Hauptsächlich wird in dieser Arbeit auf den Ingenieur für Automatisierungstechnik referenziert.

Speicherprogrammierbare Steuerung

Eine speicherprogrammierbare Steuerung (SPS) ist ein Gerät, das zur Steuerung oder Regelung einer Maschine oder Anlage eingesetzt und auf digitaler Basis programmiert wird. Im Laufe dieser Arbeit wird SPS als synonym für Soft-SPS verwendet, da es in dieser Arbeit fast ausschließlich um Soft-SPS geht. Bei Soft-SPSen handelt es sich um auf Industrie PCs laufende Steuerungs-Software, welche -anders als eine Hardware-SPS- auf einem Betriebssystem läuft.

Anwendungssoftware

Mit Anwendungssoftware (auch Anwendungsprogramm, kurz Anwendung oder Applikation; englisch application software, kurz App) werden Computerprogramme bezeichnet, die genutzt werden, um eine nützliche oder gewünschte nicht systemtechnische Funktionalität zu bearbeiten oder zu unterstützen. Sie dienen der „Lösung von Benutzerproblemen“. (CLAUS & SCHWILL 2006)
Die Verwendung dieses Begriffs in dieser Arbeit bezieht sich stets auf Software, die nicht zur Steuerung von Maschinen gedacht ist und meist eine grafische Benutzeroberfläche besitzt.

Reaktive Systeme

Systeme welche im Laufzeitverhalten sehr stark von ihrer Umwelt abhängen, wie z.B. Speicherprogrammierbare Steuerungen. Ihr Verhalten hängt stark von den Positionen und momentanen Verhalten der an die Steuerung angeschlossenen Aktuatoren und Sensoren ab (KORMANN et al. 2012, S. 1615).

Virtuelle Inbetriebnahme

Ein Konzept, welches die Steuerung eines virtuellen 3D Modells einer Maschine mittels einer SPS ermöglicht. Damit soll in der Automatisierungstechnik die Entwicklungszeit verringert werden.

Capture/Replay Tool

Auch Testroboter genannt. Ein Software-Werkzeug, das alle vom Tester manuell durchgeführten Bedienschritte, wie zum Beispiel Tastatureingaben oder Mausklicks, während einer Testsitzung aufnimmt. Diese Bedienschritte werden vom Werkzeug in einem Testskript gespeichert. Danach kann der aufgezeichnete Testfall vom Tester automatisch wiederholt abgespielt werden. Verwendungszweck ist die Softwarequalitätssicherung.

PLCopenXML

Ein Standardformat in der Automatisierungstechnik, welches dazu dienen soll, SPS Programme verschiedener Hersteller miteinander kompatibel zu machen.

AutomationML

Ein Standardformat in der Automatisierungstechnik, welches dazu dienen soll, alle Bereiche der Automatisierung der verschiedenen Hersteller miteinander kompatibel zu machen.

1 Einleitung

Ausgelöst durch die dritte industrielle Revolution, welche am Ende des 20. Jahrhunderts mit der Digitalisierung von Computern eingeleitet wurde, ist die Elektronik ein fester Bestandteil des täglichen Lebens geworden, sodass mittlerweile beinahe alle Lebensbereiche des Menschen davon abhängig sind. Ebenfalls dadurch ausgelöst, arbeiten die meisten Fabriken heutzutage mit hoch automatisierten Maschinen, welche die Produktivität um ein Vielfaches steigern im Vergleich zur Herstellung durch menschliche Arbeitskräfte. Gleichzeitig wird die gleichbleibend hohe Qualität des Endprodukts garantiert und eventuell sogar gesteigert. Diese digitale Revolution an Produktionsmaschinen führte schließlich zur Erfindung der speicherprogrammierbaren Steuerung (fortan SPS genannt).

SPSen sind eingebettete Systeme deren Funktionalität von einfachen Mikrochips bis hin zu High-end Computern reichen. Entscheidend ist die Fähigkeit über verschiedene Bussysteme mit anderen Komponenten kommunizieren zu können. Das Anwendungsgebiet von SPSen liegt dabei vorrangig im Bereich der Regelung und Automatisierung von mechatronischen Systemen. Das heißt, dass z.B. auf Knopfdruck Antriebe gestartet werden, die daraufhin Bewegungen in Gang setzten, welche wiederum der SPS über Sensoren eine Rückmeldung geben und diese den nächsten Schritt des Ablaufplans befolgt. „Heute steigt der Bedarf an Automatisierung in allen technischen Bereichen“ (BERNS et al. 2010, S.147). Die damit einhergehende Funktionalitätserweiterung der Maschine führt zu einem Zuwachs der Verantwortung an die Steuerungssoftware. Gleichzeitig ist zu beobachten, dass es in den kommenden Jahren immer mehr eine Transition der Funktionalitäten einer Maschine weg von mechanischen in Richtung Software geben wird. Die Software wird die Hauptdisziplin für Innovation und somit eine der Hauptaspekte der generellen Systemqualität (vgl. KORMANN et al. 2012, S. 1615). Diese Verantwortung wird verkompliziert durch die allgemeine Haltung, dass Software heutzutage schneller und günstiger fertiggestellt werden muss, während eine ständig gleichbleibende Qualität erwartet wird (E.B. BLANCO VINUELA et al. 2014, S. 1258). Für die daraus resultierenden Probleme bedarf es Lösungen.

Ähnlich wie bei der Entwicklung von Anwendungssoftware, bei der das Klickverhalten des Benutzers auf der Benutzeroberfläche sehr unvorhersehbar ist und oft zu ungewollten Programmzuständen führen kann, ist die Umwelt bei automatisierten Maschinen mit Verfahrwegen oder schwenkenden Roboterarmen auch ein solcher unvorhersehbarer Faktor. Alles wird jeweils durch die Umwelt in Form eines Benutzers, in aller Regel mit Maus und Tastatur ausgerüstet, initiiert und überwacht. Dies ist ein wesentlicher Unterschied zu der Software einer SPS, welche automatisierte Maschinen steuert. Die Software reaktiver Systeme in der Automatisierungstechnik wie der SPS muss sich mit den gegebenen physikalischen Grenzen seiner Bauteile und dessen Problemen auseinandersetzen. Beispiele hierfür sind verrutschendes Material ohne feste Ortskenntnis auf einem Laufband oder ein Roboterarm, der sein Ziel nicht richtig greift. Solche Fehlverhalten bereiten beim Betrieb einer automatisierten Maschine weitreichende Probleme, denn SPSen übernehmen meist komplexe Steuerungsaufgaben, deren Defekt oder inkorrekte Funktionsweise zu ernsthaften Auswirkungen auf menschliches Leben führen kann. Somit ist „die Sicherung einer hohen Qualität dieser Systeme Pflicht“ (SCHOLZ 2005, S. 207).

Das Testen von System- und Softwarefunktionen sind sehr wichtige Mittel, um eine gewisse Qualität zu garantieren. Bei eingebetteten Systemen in der Automobil- und Raumfahrtindustrie werden Funktionstests an der Steuerungssoftware vor dem Feldversuch mit echter Hardware bereits seit Jahren standardmäßig durchgeführt. Jedoch hat sich dieses Testen der Software vor der Inbetriebnahme einer Maschine in der Industrie noch nicht flächendeckend durchsetzen können. Gründe dafür sind zum Einen der Zeitmehraufwand und die anfänglichen zusätzlichen Kosten für

den Einsatz von Tests. Und das obwohl die Mehrheit aller Fehler bei der Automatisierung bereits heute auf die Software der SPS zurückgeführt werden können (KORMANN et al. 2011).

Der Entwicklung von SPS Software fehlen heutzutage weiterhin viele der Software Engineering Best Practice Ansätze wie z.B. Unit Testing (E.B. BLANCO VINUELA et al. 2014). Doch durch die rapide wachsende Verantwortung der Software für die Systemqualität wird mittlerweile auch in der Automatisierungsindustrie das Streben nach Absicherung der Software erkennbar. Das Testen von SPSen zu vereinfachen und systematisch zu gestalten ist eine wichtige Problemstellung, welche es weiter zu untersuchen gilt (JAMRO 2015, 1120). Dieses Problem wird in dieser Masterarbeit adressiert.

1.1 Rahmenbedingungen

Diese Ausarbeitung ist im Rahmen einer Zusammenarbeit der Continental Reifen Deutschland GmbH im Bereich der Software Engineering Abteilung, genauer der Conti Machinery, entstanden. Die Conti Machinery ist die hauseigene Maschinenbaufabrik der Continental AG und für den Bau von Reifenkonfektionierungsmaschinen verantwortlich. Alle Arbeiten, vom Konzept einer Maschine, über die Konstruktionszeichnungen, dem Hard- und Softwareentwurf bis hin zur Inbetriebnahme der fertigen Maschine liegen im Zuständigkeitsbereich der Conti Machinery.

Bei den Maschinen handelt es sich um hoch automatisierte, komplexe und schnell produzierende Anlagen, welche zeitgleich viele mechanische Bewegungen durchführen. Diese Bewegungen werden von einer Großzahl an Sensoren erfasst und durch elektrische und pneumatische Antriebe in Bewegung gesetzt.

Kontrolliert wird der Prozess durch eine Speicherprogrammierbare Steuerung. Diese ist mit allen Aktuatoren und Sensoren verbunden und überwacht bzw. kontrolliert Anlagenzustand, sofern die erwähnten Komponenten korrekt arbeiten.

Die Logik hinter der SPS wird durch die Automatisierungs-Software TwinCat 2.0 von der Fa. Beckhoff Automation GmbH und Co. KG. programmiert.

Auf einem so genannten Industrie PC (IPC) laufend, kommuniziert die Software per Feldbus mit allen verbundenen Eingangs- und Ausgangskomponenten und kann somit die Maschine in Bewegung setzen. Die von der Conti Machinery verwendeten IPCs der Firma Beckhoff verfügen über ein Microsoft Windows Embedded Compact 7 Betriebssystem.

Die verwendeten Entwicklungsrechner arbeiten mit einem gewöhnlichen Microsoft Windows 7 Enterprise Betriebssystem auf dem die TwinCat-Automatisierungssoftware mitinstalliert ist. Des Weiteren ist zu beachten, dass jede bei der Conti Machinery entwickelte Anwendersoftware nach dem Internen Standard auf dem Microsoft.NET Framework basieren muss, d.h. in einer .NET-Programmiersprache, wie C#.NET oder Visual Basic.NET zu schreiben ist.

Es bestehen die folgenden limitierenden Randbedingungen:

- Verwendeter Entwickler-Laptop: HP ZBOOK G2
- Betriebssystem: Microsoft Windows 7 Enterprise
- Automatisierungssoftware: Beckhoff TwinCat 2
- Zu verwendende Programmiersprache: C# .NET

1.2 Recherche

Die Quellen zu dieser Arbeit stammen aus folgender Recherchearbeit:

Nach anfänglicher Einarbeitung in die Entwicklung in der Automatisierungstechnik wurden die Themen Test und Verifikation der Maschinen zusammen mit Ingenieuren der Software Engineering-Abteilung diskutiert. Auf dieser Grundlage startete die eigentliche Arbeit. Die Suchmaschine Google wurde genutzt um die Initiale Recherche nach bereits auf dem Markt verfügbaren Lösungen zum Thema „Steigerung der Software Qualität von speicherprogrammierbaren Steuerungen“ zu finden. Darauf aufbauend wurden die bekanntesten elektronischen Bibliotheken (ACM Digital Library, Elsevier Science Direct, IEEE Explore, Springer Link) nach verwandten Arbeiten und Vorstellungen von bereits implementierten Testwerkzeugen für SPSen durchsucht. Es handelt sich bei den Suchwörtern jeweils um die Einstiegspunkte zu dem jeweiligen Arbeiten bzw. Lösungen. Daher beschränken sich die Quellen nicht auf diese, während der Initialen Recherche gefundenen, Ergebnisse. Es wurden ebenfalls die Institute der jeweiligen wissenschaftlichen Einrichtungen und die Firmen auf weiterführende Recherche-Ergebnisse hin untersucht.

Die dabei verwendeten Suchwörter lauten: „PLC Testing“, „PLC Verification“, „PLC Model Checking“, „PLC Formal Verification“,, „Programmable Logic Controller Testing“, „PLC Test Suite“, „SPS Test Suite“, „SPS Testen mit HIL“, „SPS Testen“, „IEC 61131-3 Test“, „PLC HIL“, „PLC Hardware in the loop“, „Automatic PLC Testing“, „Testing in Automation with TwinCat“, „Testing in Automation Industry“, „Testen von SPS Systemen“, „Testen reaktiver Systeme“, „Twincat Test Software“, „IEC 61131-3 Test Software“, „PLC Plant Recording“, „Recording PLC“, „HIL Test für SPS“, „Automatisches Testen von SPS Steuerungssoftware“.

Die Quellen für die Grundlagen stammen aus Fachliteratur. Die genauen Abschnitte werden an den entsprechenden Stellen im Text durch Zitierung kenntlich gemacht.

2 Grundlagen

Dieses Kapitel soll dem Leser ein Grundverständnis über die Automatisierungstechnik und das Testen reaktiver Systeme geben. Es wird versucht aufkommende Fragen im Verlauf des Lesens bereits vorab zu klären, welche sich eine Studentin oder ein Student der Informatik bzw. der technischen Informatik während des Lesens der Ausarbeitung stellen würde. Dadurch soll gewährleistet werden, dass alle die in dieser Masterarbeit vorgestellten Entscheidungen bzw. Entscheidungswege, nachvollzogen werden können. Es werden lediglich die relevanten Aspekte der jeweiligen Technologien und Methoden erläutert, ohne dabei das Wesentliche durch unnötige Vertiefung zu verlieren.

2.1 Entwicklung von automatisierten Systemen

Das Ziel von Automatisierungssystemen ist es einen technischen Prozess weitestgehend ohne den Einfluss eines menschlichen Bedieners selbstständig ablaufen zu lassen. Dies stellt eine Reihe von Anforderungen an das automatisierte System, die von den Eigenschaften des technischen Prozesses, des technischen Systems, Sicherheits- und Umweltfaktoren sowie von den gesetzlichen Bestimmungen abhängen. Daher sind am Entwicklungsprozess einer Maschine eine Vielzahl an Entwicklern aus allen technischen Bereichen beteiligt. Soweit möglich wird sich in dieser Arbeit auf den Softwareaspekt konzentriert.

Um die Fertigungsprozesse in der Automatisierungstechnik unter Kontrolle zu halten, werden verschiedene Arten der Prozesssteuerungen eingesetzt. Sie sind auf Abbildung 2-1 zu sehen.

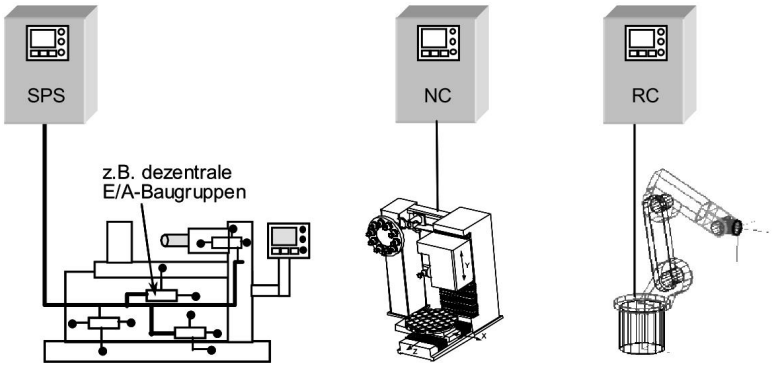


Abbildung 2-1 Prozesssteuerungen der Automatisierungstechnik (WÖRN & BRINKSCHULTE 2005, S. 20)

Die Speicherprogrammierbare Steuerung links im Bild ist für den übergeordneten Ablauf des Prozesses verantwortlich und kann auch die beiden nebenliegenden Einheiten über definierte Schnittstellen befehligen. Für die Bewegungssteuerung (Motion Control) von Achsen oder Werkzeugmaschinen werden so genannte Numeric Controls (NCs) eingesetzt. Auf der rechten Seite dargestellt werden für Robotersteuerungen entsprechend Roboter Controls genutzt. Letztere sind für diese Arbeit nicht relevant, die wesentlich Aspekte der beiden anderen werden im Folgenden näher beschrieben.

Durch die zunehmende Verlagerung von Hardware-Funktionalitäten in die Software ist der Anteil der Softwareentwicklung am gesamten Entwicklungsaufwand stetig gestiegen. Die Entwicklung der

Steuerungssoftware stellt somit einen wesentlichen Teil des Automatisierungsprojektes dar. Dabei sind sowohl die Vorgehensweisen als auch die verwendeten Programmiersprachen historisch gewachsen. In den Anfängen bedeutete die Erstellung einer Steuerungssoftware die Entwicklung von Datenstrukturen und Algorithmen, die eine spezifische Anforderung erfüllen sollten. Dabei konnten die meisten Probleme durch eine Reihe von Schritten oder Prozeduren gelöst werden. Dieser Ansatz, meist als imperative oder prozedurale Programmierung bezeichnet, ist jedoch nur bedingt für kleine, wenig komplexe Probleme effektiv anwendbar. Mit steigender Komplexität der zu automatisierenden Systeme ist auch der notwendige Aufwand gestiegen um derartige Software zu Erstellen, die Entwicklungsschritte und Inhalte vollständig zu dokumentieren, zu warten oder zu erweitern. Daher wird seit einigen Jahren versucht, auch objektorientierte Konzepte einzuführen. Dies ist in der Automatisierungstechnik nicht ohne Hindernisse realisierbar, da Abstraktionen im Quellcode von den Bauteilen der Maschine unvorteilhaft sein können. So ist zu berücksichtigen, dass das Wartungspersonal zum Lösen von Fehlern häufig in den Quellcode der Maschine schauen muss und möglicherweise nicht mit dem Paradigma der Objektorientierten Programmierung vertraut ist. Daher ist ein möglichst leicht verständlicher Aufbau des Codes und möglichst hohe Korrelation zwischen gewählten Variablentitel zu den Hardwarebestandteilen der Anlage ein angestrebtes Ziel (OVATMAN et al. 2014, S.3).

Das Vorgehensmodell bei der Planung von Automatisierungssystemen nach Seitz (MATTHIAS SEITZ 2012, S.214) sagt, dass die Realisierung der Hard- und Software anhand der Pflichtenheftvorgaben erfolgen soll. Die Steuerungshardware wird gemäß dem Hardwarestrukturplan beim Hersteller bestellt und nach Lieferung im Schaltraum aufgebaut. Die einzelnen Komponenten der Steuerungshardware müssen somit vom Anwender nicht erneut geprüft werden, da sie beim Hersteller umfangreiche Qualitätsprüfungen durchlaufen und sich durch ihre weite Verbreitung im Betrieb bewährt haben. Somit ist hinsichtlich der Hardware lediglich zu prüfen, ob sie vollständig ist und alle Module korrekt installiert wurden.

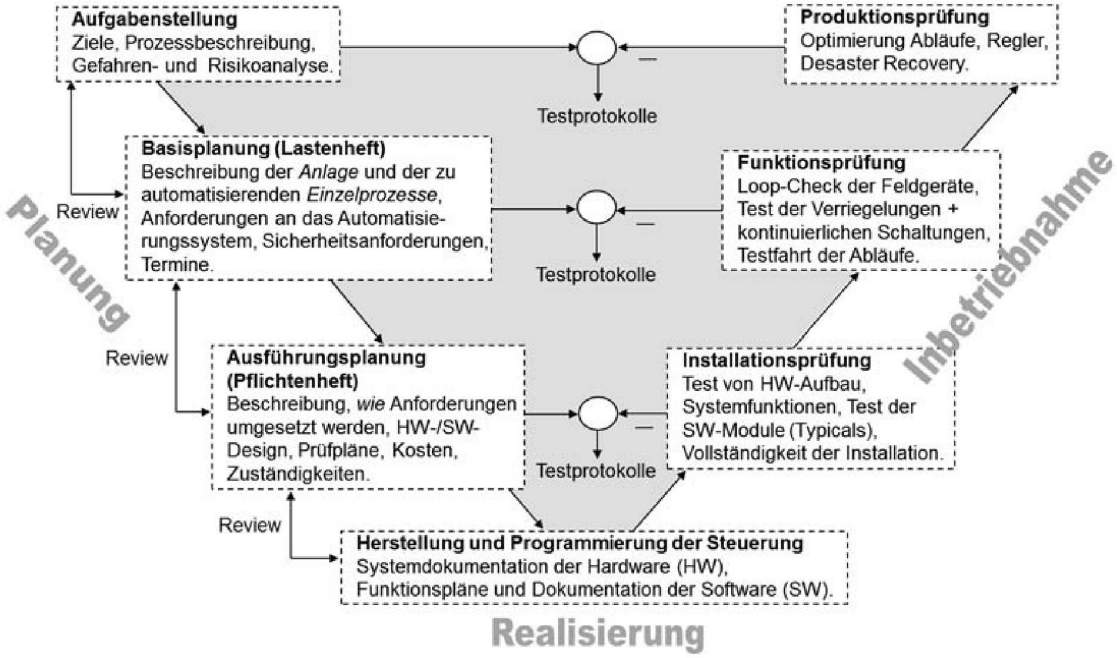


Abbildung 2-2 Vorgehensmodell bei der Planung von Automatisierungssystemen(MATTHIAS SEITZ 2012, 211)

Die Abbildung 2-2 beschreibt das ideale Vorgehensmodell mit Berücksichtigung der Besonderheit des Entwicklungsprozesses im Maschinen- und Anlagenbau, bei der stets parallele

Entwicklungsprozesse in den verschiedenen Disziplinen (Maschinenbau, Elektrotechnik, Softwaretechnik) herrschen.

Meist wird mit der mechanischen Konstruktion begonnen, mit der Planung der elektrotechnischen Komponenten fortgeführt und schließlich die Software entwickelt. Dies führt dazu, dass Integrationstests der Software meist erst durchgeführt werden, wenn an den mechanischen und elektrotechnischen Komponenten kaum oder bis gar keine Änderungen mehr möglich sind, da dies zu hohe Kosten verursachen würde. Somit muss die Software auf bereits festgelegte mechanische und elektrotechnische Lösung angepasst werden, um geforderte Funktionalitäten zu realisieren. Dies führt zu einer Steigerung der Softwarekomplexität. (RÖSCH S. et al. 2013, S. 41).

2.2 Speicherprogrammierbare Steuerungen

Speicherprogrammierbare Steuerungen sind Geräte mit Mikroprozessoren, welche wie eine Blackbox betrachtet werden können. Ausgangssignale werden mittels Programm und entsprechenden Eingangssignalen geschaltet. In der Regel werden SPSen zur Steuerung bzw. Regelung verschiedenster Prozesse eingesetzt welche zum Großteil Aktuatoren und Sensoren ansteuern, die daraufhin mechanische Bewegungen umsetzen bzw. Erkennen, ob ein Ereignis in der Maschinenumgebung eingetreten ist, wie beispielsweise eine überschrittene Lichtschranke.
Ein wesentlicher Unterschied zwischen dem Programmablauf in der konventionellen Datenverarbeitung und bei SPS-Systemen ist die Art der Abarbeitung des Programmcodes. Während herkömmliche Softwaresysteme meist ereignisgesteuert arbeiten und bei Auftreten eines Ereignisses eine Methode ausführen, werden SPS-Programme immer wieder zyklisch abgearbeitet. Zu Beginn eines Zyklus werden die Zustände der Eingänge in das Prozessabbild der Eingänge kopiert. Anschließend wird das Programm unter Verwendung des Prozessabbildes abgearbeitet. Ausgänge werden dabei nicht direkt gesetzt, sondern ebenfalls in das Prozessabbild geschrieben. Am Ende des Zyklus wird das Prozessabbild auf die physikalischen Ausgänge übertragen.

Abbildung 2-3 zeigt die prinzipielle Arbeitsweise einer SPS. Es handelt sich um ein typisches EVA-Prinzip. Einmal gestartet werden zyklisch alle Eingänge eingelesen, die Eingaben mit Hilfe des Programms in der SPS verarbeitet und anschließend entsprechende Ausgänge geschaltet. Dabei werden die Ressourcen des SPS-Arbeitsspeichers in Anspruch genommen.

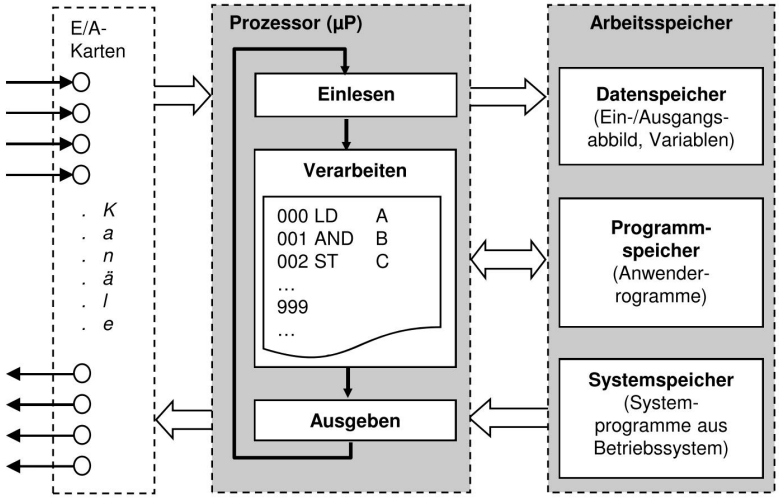


Abbildung 2-3 Signalverarbeitung und Arbeitsweise einer SPS (MATTHIAS SEITZ 2012, S. 28)

Bei den „E/A Karten“ in der Abbildung handelt es sich um Steckkarten, welche komfortabel ein- und ausgebaut werden können und mittels eines Feldbus mit der SPS kommunizieren können. Sie sind mit Feldgeräten verbunden. Das sind solche, welche der SPS die Eingangssignale liefern, bzw. von der SPS kontrolliert werden, also Aktuatoren und Sensoren. Diese Entwicklung hilft, die Kommunikation mit den Ein- und Ausgängen der SPS komfortabler auswerten zu können. Dank der sogenannten Feldbustechnik wird auch der Verdrahtungsaufwand minimiert. Die Feldgeräte und SPS werden zusammen an eine Busleitung angeschlossen. Voraussetzung für eine Feldbusankopplung ist aber, dass die Feldgeräte über einen Mikrocontroller mit busfähiger Schnittstelle verfügen. Dies ist bei den meisten älteren Feldgeräten nicht der Fall (MATTHIAS SEITZ 2012, S. 33).

2.2.1 Sensoren und Aktuatoren

Wie bereits erwähnt, werden die Ein- und Ausgangssignale über die E/A-Karten an die SPS weitergeleitet. Es handelt sich meistens um ein Master-Slave System, bei dem mehrere E/A Karten hintereinander angeschlossen sind und untereinander wiederrum mit einem weiteren „internen Bus“ kommunizieren. Zur SPS muss nur der Master kommunizieren. Die folgenden Beispiele zeigen weshalb die Ansteuerung und das Auslesen dieser Karten so komfortabel sind.

Durch den Aufbau einer solchen Karte, welche in Abbildung 2-3 zu sehen ist, kann ein kompletter Motor mit nur einem einzigen binären Signal gesteuert werden. Für die SPS ist es dabei irrelevant, was für ein Feldgerät an dem Ausgang angeschlossen ist, da es sich aus der SPS Sicht nur um eine Boolesche Variable handelt, also entweder TRUE = Motor an oder FALSE = Motor aus. Für den korrekten Anschluss an die benötigte Stromversorgung und Platzierung ist der Automatisierungsingenieur selbst verantwortlich.

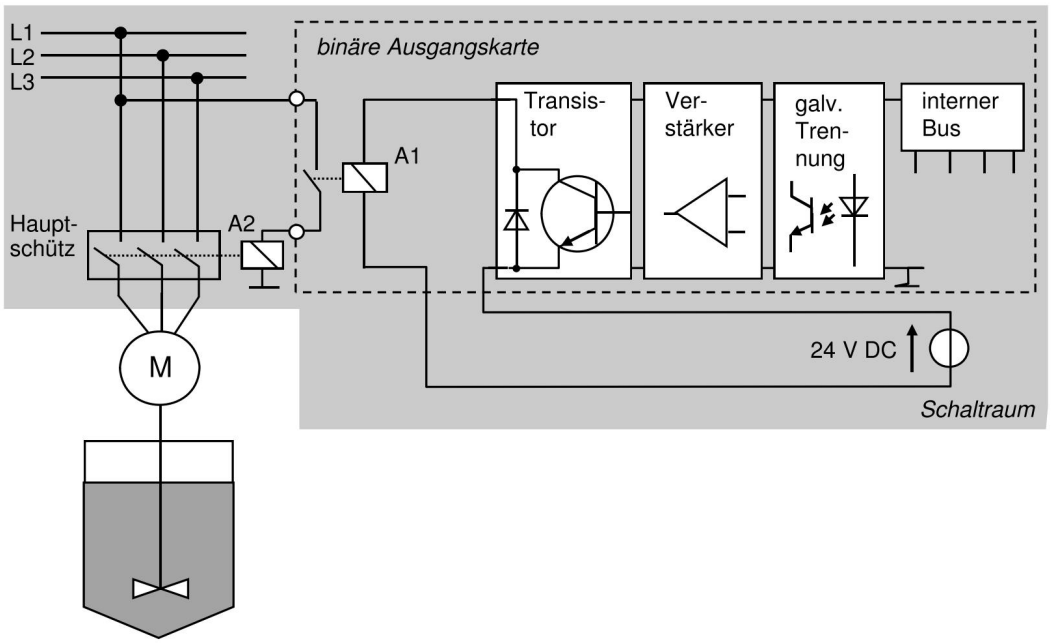


Abbildung 2-4 Beispiel eines digitalen Ausgangs einer SPS (MATTHIAS SEITZ 2012, S. 30)

Im Gegenteil dazu werden bei einer digitalen Eingangskarte die Signale für die SPS geschaltet, wie es Abbildung 2-5 beschreibt. Dabei wird durch eine Trigger-Stufe verhindert, dass der digitale Eingang bei kleinsten Signaländerung zu feinfühlig togglet.

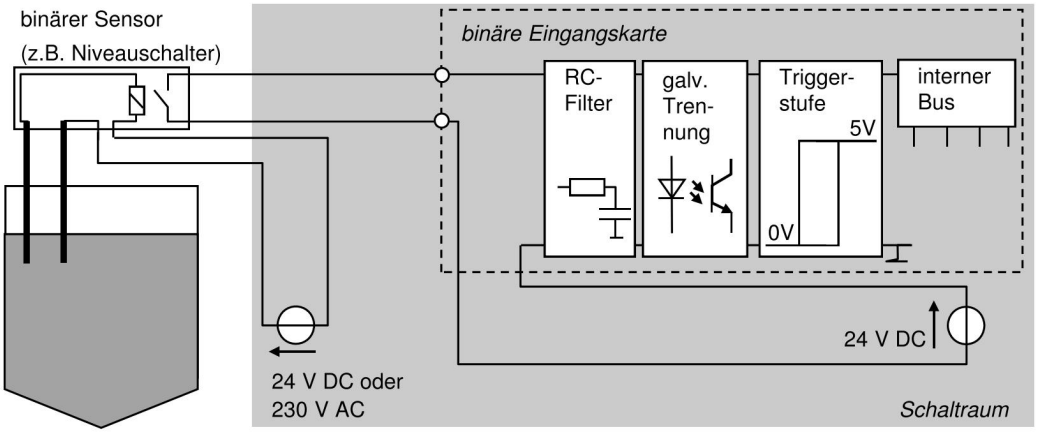


Abbildung 2-5 Beispiel einer digitalen Eingangskarte einer SPS (MATTHIAS SEITZ 2012, S. 32)

Die analoge Eingangskarte verfügt in der Regel über einen konfigurierbaren Wertebereich. Durch den internen Analog/Digital-Wandler wird der SPS nur ein Integer-Wert über den Feldbus übertragen.

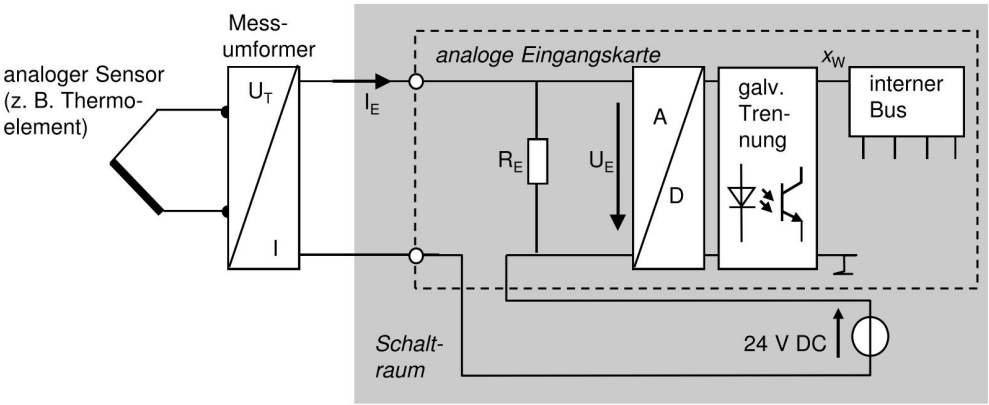


Abbildung 2-6 Analoge Eingänge einer SPS(MATTHIAS SEITZ 2012, S. 31)

Ähnlich zu den digitalen Karten, verhält es sich auch mit den analogen. Die Ansteuerung wird durch einen simplen Datentypen wie z.B. Integer ermöglicht, da durch den internen Analog/Digital-Wandler in der Ausgangskarte das Signal rückgewandelt wird. Das Beispiel in Abbildung 2-7 zeigt die interne Verschaltung einer solchen Karte an einer Lüfter-Regelung.

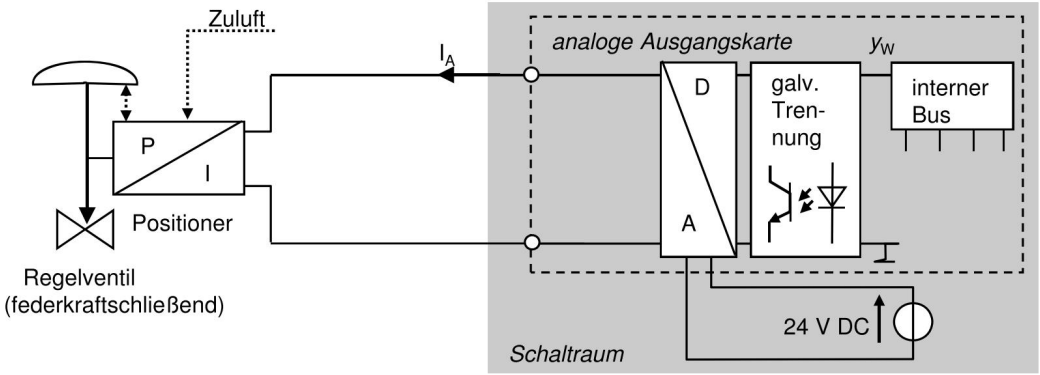


Abbildung 2-7 Beispiel eines analogen Ausgangs einer SPS (MATTHIAS SEITZ 2012, S. 34)

2.2.2 Nummeric Controls

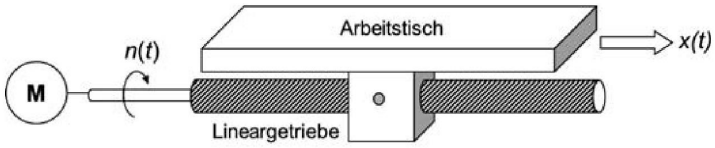


Abbildung 2-8 Linearachse für einen Arbeitstisch (MATTHIAS SEITZ 2012, S. 175)

Die in Abbildung 2-8 dargestellte Linearachse wird von einem Motor angetrieben und bewegt den Arbeitstisch über das Gewinde in x-Richtung. Der Motor wird so konfiguriert, dass er anhand der Motordrehzahl über Zeit der SPS mitteilen kann, wie weit die Achse verfahren ist. Die Ansteuerung von Achsen stellt einen Sonderfall in der Kommunikation mit Feldgeräten dar, da sie über sogenannte Numeric Controls (NC) realisiert wird.

Die Ansteuerung einer Achse erfolgt zwischen SPS und NC erfolgt auf zwei Arten:

- Als Zyklisches Interface von der SPS zur NC (Sollwerte etc.) und von der NC zur SPS (Istwerte etc.): Austausch der benötigten Informationen zu jedem SPS-Zyklus, über das zyklische Prozessabbild (Ein/Ausgänge).
- Als Funktionsbaustein: Es werden NC-Funktionsbausteine (NC-FB) bereitgestellt, in denen jeweils thematisch verwandte Funktionalitäten zusammengefasst sind. Die NC-FBs sind als Firmware-Bausteine implementiert, d.h. sie sind Teil der Steuerungssoftware und in ihrem Verhalten fest definiert. Die Bausteine verfügen über Ein- und Ausgänge, deren Datentypen ausschließlich Standard-Datentypen sind.

Industrielle Motion-Control-Systeme, wie beispielsweise das TwinCAT-NC-Paket von Beckhoff, bieten standardisierte Funktionsbausteine zur Bewegungssteuerung von Gleichlauf- und Positioniersystemen. Diese sind größtenteils von der PLCopen genormt und nach IEC 61131 in SPS-Programmen einsetzbar. (MATTHIAS SEITZ 2012, S. 174).

Ein Beispiel für solch einen Baustein ist in der folgenden Abbildung zu sehen.

Abbildung 2-9 MC_MoveAbsolute Baustein zur Kontrolle von Achsen(BECKHOFF AUTOMATION GMBH & CO. KG 2016a)

2.2.3 Arten von SPSen

Die drei verschiedenen Aufbauarten werden bei SPSen unterschieden nach Hardware-SPS, Slot-SPS und Soft-SPS. Letztere ist für die vorliegende Ausarbeitung wichtig, da es im Gegensatz zu den zwei erstgenannten SPSen nicht um eingebettete Systeme. Eine Soft-SPS ist dagegen reine Software und nutzt die Hardware des Host-PCs. Über eine Netzwerkanbindung kann diese mit den Feldgeräten kommunizieren (MATTHIAS SEITZ 2012, 26 f.). Die Vorteile der SPS im PC ergeben sich hauptsächlich dadurch, dass die rasante Entwicklung der PC-Leistung für SPSen genutzt

werden kann. Somit können heutige Soft-SPSen höhere Verarbeitungsgeschwindigkeiten erreichen als Hardware-SPSen. Damit werden anstatt von Zykluszeiten im Millisekunden-Bereich Zykluszeiten von wenigen Mikrosekunden erreicht. Ebenfalls können die IPCs gleichzeitig zur Steuerung, Entwicklung und Prozessvisualisierung verwendet werden. Dadurch ergeben sich preisgünstigere, einfachere und durchgängige Systemstrukturen, mit denen der Anwender gewohnt ist umzugehen. Das System wird offener, da der Datenaustausch auf einer einheitlichen Plattform unter Windows standardisiert wird. Somit wird die Ankopplung von Bedien- und Beobachtungssystemen sowie von übergeordneten Planungssystemen an die SPS vereinfacht. (MATTHIAS SEITZ 2012, 27)

2.2.4 Program Organisation Units

Bevor eine Übersicht der Programmiersprachen gegeben wird, soll kurz auf das zugrunde liegende Konzept der IEC 61131-3 eingegangen werden.
Steuerungsprogramme bestehen aus einzelnen, funktionell voneinander unabhängigen Bausteinen, so genannten Programm-Organisationsbausteinen (POE), oder auf englisch: Program Organisation Units (POU). Sie können getrennt voneinander erstellt und bearbeitet werden. Ebenfalls können sie zu einem gewissen Grad ineinander verschachteltet werden (JAMRO 2015, 1120). Die IEC 61131-3 definiert drei Klassen von Programm-Organisationseinheiten, die verschiedene Eigenschaften aufweisen: Funktionen, Funktionsbausteine und Programme (nach Seitz(MATTHIAS SEITZ 2012, 52)):

- *Funktionen*: Eine Funktion gemäß IEC 61131-3 ist eine POE, die beliebig viele Eingangsparameter hat und genau ein Ergebnis-Element zurückliefert. Ein Beispiel stellt die Funktion sin(x) dar. Für Funktionen gelten bestimmte Einschränkungen. So dürfen Funktionen keine interne Speicherfähigkeit besitzen, keine globalen Variablen lesen oder schreiben und aus Funktionen heraus dürfen keine Funktionsbausteine aufgerufen werden. Nur so kann sichergestellt werden, dass die Forderung, bei gleichem Satz von Eingangsvariablen stets dasselbe Ergebnis zu liefern, erfüllt wird. Rekursive Aufrufe einer Funktion sind nach IEC 61131 nicht möglich. Eine Funktion hat im Unterschied zu einem Funktionsbaustein *kein* Gedächtnis. Damit ist es Funktionen nicht erlaubt, interne Zwischenwerte zu speichern. (MATTHIAS SEITZ 2012, 52)
- *Funktionsbausteine und Programme:* Beide unterscheiden sich nur geringfügig voneinander. Beide POEs können beliebig viele Ein- und Ausgangsparameter haben und interne Daten besitzen, die nach einem Aufruf erhalten bleiben und beim nächsten Aufruf wieder genutzt werden können. Die Ausgangswerte eines Funktionsbausteins oder eines Programms können somit nicht nur von den Eingangsparametern, sondern auch von dem internen Zustand der POE abhängen. Die Eigenschaft ein Gedächtnis besitzen zu können, sind ein wesentliches Unterscheidungsmerkmal zu Funktionen, wodurch Funktionalitäten wie Zähler- oder Timer-Bausteine realisiert werden können. Funktionsbausteine und Programme können mehrfach instanziiert werden. Jede Instanz besitzt einen eigenen Bezeichner und eine eigene Datenstruktur. Der wesentliche Unterschied zwischen diesen beiden POEs ist die Eigenschaft, dass Programme weder von anderen Programmen noch von Funktionsbausteinen aufgerufen werden dürfen. Programme werden ausschließlich von Ressourcen aufgerufen, können explizit einer Task zugeordnet werden oder laufen implizit in der Hintergrundtask.

2.2.5 IEC 61131-3

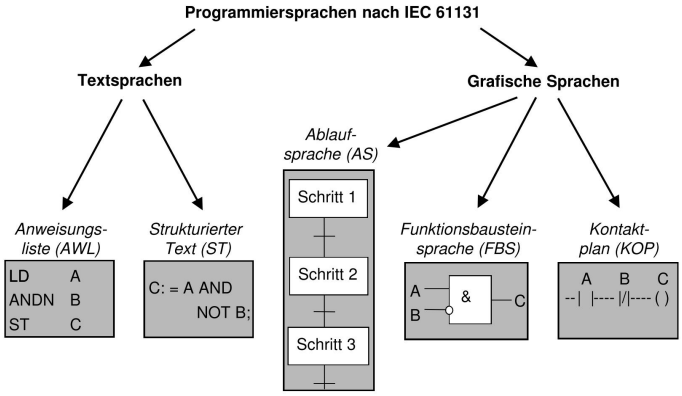
Der Standard IEC 61131 der Internationalen Elektrotechnischen Kommission (IEC) standardisiert SPSen. Im dritten Teil des Standards werden die Programmiersprachen definiert. Obwohl es

mittlerweile auch alternativen gibt, wird nach der ARC Industry Advisory Group die Nutzung des Standards bei SPSen in der industriellen Praxis zumindest in den kommenden fünf bis zehn Jahren etabliert bleiben (BECKERT et al. 2015, 242). Gleichzeitig werden Automatisierungssysteme durch die heute verfügbare Hardware mit zum Teil stark unterschiedlichen Programmiersystemen immer heterogener. Um Automatisierungssysteme auch weiterhin mit hoher Qualität und unter angemessenem Einsatz von Ressourcen entwickeln zu können, wurde eine Standardisierung der Programmiersprachen eingeführt.

Um die Echtzeit garantieren zu können, muss jeglicher Speicher schon zum Start des Programms feststehen und es gibt keine dynamische Speicherreservierung. Damit ist auch der Sprachumfang eingeschränkt. Es gibt neben einfachen Typen (Byte, Word, Dword...) und Strings mit fester Länge noch Strukturen die aus einfachen Datentypen oder anderen Strukturen zusammengesetzt sind (SULFRIAN).

Die IEC 61131-3 unterscheidet zwischen textuellen Sprachen, wie der Anweisungsliste (AWL) und dem Strukturierten Text (ST) und grafischen Sprachen, wie der Ablaufsprache (AS), der Funktionsbausteinsprache (FBS) und dem Kontaktplan (KOP). Die Sprachen können wie folgt charakterisiert werden:

- Die Anweisungsliste (AWL) ist eine Assembler-ähnliche Sprache, bei der die einzelnen Anweisungen, ähnlich einer Maschinensprache, listenartig untereinander aufgeführt werden.
- Strukturierter Text (ST) stellt im Gegensatz zur AWL eine Pascal-ähnliche Hochsprache dar. Leistungsfähige Schleifenprogrammierung, mathematische Funktionen, Iteration sowie konditionierte Befehle sind die herausragenden Bestandteile dieser Sprache. Sie ist vielmehr als Ergänzung statt als Ersatz für AWL zu sehen.
- Die Ablaufsprache (AS) definiert eine POE als eine Reihe von Schritten und Transitionen (Übergangsbedingungen), die durch gerichtete Verbindungen miteinander verbunden sind. Dadurch ist die Ablaufsprache eher zur Realisierung übergeordneter Abläufe geeignet.
- Die Funktionsbausteinsprache (FBS) ist eine grafische Sprache, die den Grundgedanken funktionsorientierter logischer Ablaufketten widerspiegelt. Eine Programm Organisationseinheit in der Funktionsbausteinsprache besteht aus Netzwerken logisch verknüpfter Bausteine. Funktionsbausteine haben definierte Ein-/Ausgänge und können von Entwicklern wie Blackboxes angesehen werden (OVATMAN et al. 2014, 4)
- Der Kontaktplan (KOP) ist eine grafische Programmiersprache, die sich am Stromlaufplan elektrischer Schaltungen orientiert. Zwischen zwei vertikal verlaufenden Stromschienen werden die einzelnen Teile einer Schaltung (hauptsächlich Schalter, Spulen und Funktionsbausteine) als Netzwerk dargestellt.



- **Abbildung 2-10 Programmiersprachen es IEC 61131-3 Standards (MATTHIAS SEITZ 2012, S. 60)**

- *VAR*: Lokale Variablen sind nur innerhalb der Programmorganisationseinheit gültig, in der sie deklariert wurden.
- *VAR_GLOBAL*: Globale Variablen gelten in allen Programmen, Funktionen und Funktionsbausteinen.
- *VAR_INPUT*: Durch Eingangsvariablen werden Werte in Funktionen oder Funktionsbausteine hineingeschrieben.
- *VAR_OUTPUT*: Durch Ausgangsvariablen werden Werte von Funktionsbausteinen ausgegeben.
- *VAR_IN_OUT*: Im Gegensatz zu reinen Eingangsvariablen können Ein- und Ausgangsvariablen innerhalb des Funktionsbausteins verändert und dann ausgegeben werden.
- *VAR_RETAIN*: Diese Variablen behalten ihren Wert, wenn die SPS aus- und wieder eingeschaltet wird.
- *VAR_PERSISTENT*: Persistente Variablen behalten ihren Wert, wenn die Software erneut in die SPS geladen wird.

Von Matthias Seitz (MATTHIAS SEITZ 2012, S.62) wird folgender Einsatz der Programmiersprachen empfohlen:

- Prozessabläufe lassen sich sehr gut als Schrittketten beschreiben und werden deshalb in der Ablaufsprache AS programmiert,
- Ansteuerprogramme für Aktuatoren und Sensoren bestehen zumeist aus Verknüpfungslogik und werden deshalb in der Funktionsbausteinsprache FBS programmiert,
- Anwender-Funktionsbausteine und -Funktionen, die das Wartungspersonal in der Regel ohnehin als Black-Box ansieht, können effizient in Textsprachen wie AWL oder ST programmiert werden.
- Da besonders die übergeordneten Prozessabläufe in dieser Masterarbeit interessant sind, wird im Folgenden näher auf die Ablaufsprache eingegangen:
- Der Ablauf vieler Prozesse - insbesondere der von Fertigungsprozessen - kann determiniert als eine Abfolge eindeutiger Zustände beschrieben werden. Es ist bekannt, wann ein bestimmter Zustand des Prozesses auftritt und welche Zustände von dort aus nachfolgend eingenommen werden können.
- Wie in der Mitte der Abbildung 2-10 zu sehen ist, sind die entscheidenden Elemente der Ablaufsprache sogenannte Schritte (mit Aktionen) und Transitionen (logische Bedingungen für den Übergang vom vorhergehenden zum nächsten Schritt). Schritte und Transitionen werden durch die Wirkverbindungen verknüpft. Dabei sind die Grundregeln dazu, dass Schritte und Transitionen sich immer abwechseln. Eine Wirkverbindung kann also nur einen Schritt mit einer Transition oder eine Transition mit einem Schritt verbinden, niemals aber Schritt mit Schritt oder Transition mit Transition. Wirkverbindungen wirken immer von oben nach unten bzw. von links nach rechts. Werden andere Wirkrichtungen benötigt, müssen sie durch einen Pfeil gekennzeichnet werden. Jeder Graph der Ablaufsprache (Auch Grafcet oder Schrittkette genannt) hat mindestens einen Initialisierungsschritt (doppelt umrandet), mit dem definiert wird, mit welchem Schritt die Schrittkette beginnt.
- Die meisten Grafcets sind geschlossene Grafcets, haben also einen Sprung vom Ende zurück zum Anfang. Dies ergibt sich aus dem Zweck des Grafcet, der Darstellung von Ablaufsteuerungen in der Fertigung: Um mehrere Teile fertigen zu können, müssen sich Schrittketten wiederholen.
- Die Transition liefert ein boolesches Signal, also nur TRUE oder FALSE ('1' oder '0'). Boolesche Verknüpfungen können sowohl „mathematisch“ als auch graphisch dargestellt werden.

Die meisten SPS Programme machen viel Gebrauch von globalen Variablen. Erstens, um die Schnittstelle zur externen Hardware zu definieren, indem die sie mit den physikalischen Adressen der Sensoren bzw. Aktuatoren verlinkt werden. Zweitens, weil sie fast die einzige Möglichkeit darstellen um zwischen Tasks zu kommunizieren und diese zu synchronisieren. Es existieren zwar Konzepte zum Synchronisieren von Tasks, jedoch werden diese selten verwendet, da das reservieren einer Task und das damit verbundene Abwarten für eine andere Task zu nicht-vorhersehbaren Zykluszeiten einer Task führen würde. (PRAHOFER et al. 2011, 643)

2.2.6 TwinCat

TwinCat ist ein Soft-SPS System, welches jeden Computer als SPS System einsetzbar macht. In der Regel läuft es als Task mit hoher Priorität auf einem Windows Betriebssystem. Es werden Zykluszeiten eingestellt welche vom TwinCat System eingehalten werden.

2.2.6.1 Die ADS Schnittstelle

Die Automation Device Specification (ADS) beschreibt eine geräte- und feldbusunabhängige Schnittstelle, welche die Kommunikation zwischen den ADS Teilnehmern (Geräte) regelt.

Ein Objekt, welches die ADS-Schnittstelle implementiert hat und "Server-Dienste" anbietet, wird als ADS-Device bezeichnet. Die SPS selbst ist somit auch ein SPS Device im ADS System.

Die Systemarchitektur von TwinCAT erlaubt es, einzelne Teile der Software (z.B. TwinCAT PLC, TwinCAT NC, ...) als eigenständige Geräte (Devices) zu betrachten: Für jede Aufgabe gibt es ein Softwaremodul (Server oder Client). Der Nachrichtenaustausch zwischen diesen Objekten wird über eine einheitliche ADS-Schnittstelle vom Message-Router abgewickelt. Dieser verwaltet und verteilt alle Nachrichten im System und im Netzwerk (TCP/IP). Der TwinCAT Message-Router existiert auf jedem TwinCAT-PC und auf jedem Busklemmen-Controller. Somit können alle TwinCAT-Server und Client-Programme Befehle und Daten austauschen.

2.3 Software Test

Das Testen von Software ist eine Aktivität, welche durch das Ausführen von Programmen Fehler finden soll. Weiterhin dient es zum Vergleich des Ist- und Sollverhalten. Nach Dijkstras berühmtem Diktum kann das Testen von Software allein nur die Anwesenheit von Fehlern zeigen, nicht die Abwesenheit bescheinigen (PRETSCHNER). Daher sollte man versuchen, die Tests möglichst klug zu planen, um so viele Aspekte wie möglich in so wenigen Testfällen wie nötig abzudecken. Ein vollständiges Testen einer Software ist zeitlich nicht zu realisieren: Im Buch "Software Qualität" von Kurt Schneider wird anhand des Beispiels einer Funktion mit drei Eingangsparametern verdeutlicht, dass ein komplettes Austesten einer Software unmöglich ist: Die Parameter haben jeweils 2^{16} mögliche Werte, somit insgesamt 2^{48} mögliche Werte, was zu 281.474.976.710.656 verschiedene Parameterbelegungen führt. Bei einer optimistischen Zeiteinschätzung von 100 Testläufe pro Sekunde, würde das komplette Austesten rund 89.255 Jahre dauern (SCHNEIDER 2007).

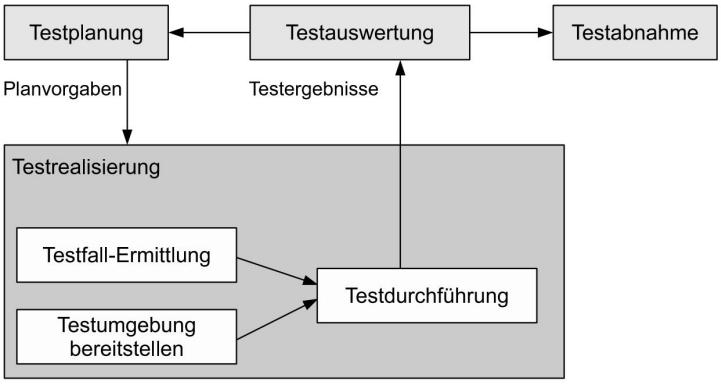


Abbildung 2-11 Kompletter Testablauf nach Gessler (GESSLER 2014, S. 194)

2.3.1 Arten von Tests

Es existiert eine Vielzahl an Variationen von Testarten und –strategien, um ein gewisses Maß an Vertrauen bezüglich Korrektheit der Funktion einer Software herzustellen. Die beiden im Folgenden erwähnten sind die einzig an dieser Stelle relevanten, da der Fokus dieser Arbeit auf die Durchführung dynamischer Tests liegt.

Alle dynamischen Testtechniken besitzen nach Vigenshow (VIGENSCHOW 2010) die folgenden gemeinsamen Merkmale:

1. Die übersetzte, ausführbare Software wird mit konkreten Eingabewerten versehen und ausgeführt.
2. Es kann in der realen Betriebsumgebung getestet werden.
3. Es handelt sich um Stichprobenverfahren.
4. Sie können die Korrektheit der getesteten Software nicht beweisen.

2.3.1.1 Blackbox Tests

Mit verschiedenen Testdaten und Testszenarien wird beim Blackbox-Test geprüft inwieweit das zu testende System die vorgegebenen Anforderungen erfüllt. Es wird somit gegen die zuvor festgelegten Anforderungen an die Software getestet. Je nach System und Art des Tests sind die Anforderungen dabei von unterschiedlicher Natur. Beim Modultest wird z.B. gegen die Modulspezifikation getestet, beim Schnittstellentest gegen die Schnittstellenspezifikation und beim Abnahmetest gegen die fachlichen Anforderungen, wie sie etwa in einem Pflichtenheft niedergelegt sind. Der Tester hat keine Kenntnis über den zu Grunde gelegten Quellcode des Programms (VIGENSCHOW 2010).

Bei solchen dynamischen Tests wird die Ausführbarkeit der Software vorausgesetzt, also sollten zumindest die zu testenden Teile bereits realisiert sein. Grundprinzip der dynamischen Verfahren ist die Ausführung der zu testenden Software mit systematisch festgelegten Eingabedaten (Testfälle). Für jeden Testfall werden zu den Eingabedaten auch die erwarteten Ausgabedaten angegeben (Sollwerte). Treten beim Vergleich der Ist- und Sollwerte Abweichungen auf, so liegt ein Fehler in der umgesetzten Software vor. Die einfachste Form des dynamischen Tests ist die Ausführung der zu testenden Software mit Eingaben, die eine testende Person unmittelbar und üblicherweise nicht reproduzierbar erzeugt.

2.3.1.2 Whitebox Test

Strukturorientierte Whitebox-Tests bestimmen Testfälle auf Basis des Softwarequellcodes. Dem Prüfer sollte demnach der Quellcode zugänglich sein. Software-Module enthalten zu verarbeitende

Daten und Kontrollstrukturen, welche die Verarbeitung der Daten steuern. Entsprechend werden Tests unterschieden nach kontrollfluss-basierten Tests und solchen mit Datenzugriffen als Grundlage.
Kontrollflussorientierte Whitebox-Tests sind auf logische Ausdrücke der Implementierung bezogen, während datenflussorientierte Kriterien sich auf den Datenfluss der Implementierung konzentrieren. Genau genommen konzentrieren sie sich auf die Art und Weise in welcher Weise die Werte mit ihren Variablen verbunden werden und wie diese Anweisungen die Durchführung der Implementierung beeinflussen. (VIGENSCHOW 2010)

2.3.2 Automatisches Testen

Automatisches Testen bedeutet, dass die gefunden Testdaten jedes Testfalls nicht von einer realen Person manuell in das zu testende Programm eingegeben werden, sondern dieses automatisch abläuft. Dadurch werden die Tests reproduzierbar und im Falle einer guten Umsetzung leicht zu parametrieren. Somit sind die Tests vor allem wertvoll beim Ausführen von Regressionstests, also beim Vergleich einer neueren Version der Software zur Vorversion oder zur Dokumentation über getestete Eigenschaften der Software.

Dennoch können automatische Tests manuelle nicht ersetzen. Auch nach der Durchführung von automatischen Tests werden immer noch manuelle Tester benötigt, da deren analytische Fähigkeiten nicht automatisiert werden können. Daher dürfen automatische Tests nicht als Ersatz für manuelle Tests gedacht werden, sondern eher als sich gegenseitig begünstigende Maßnahmen (E.B. BLANCO VINUELA et al. 2014, S. 1259).

2.3.3 Hardware in the loop Simulation

Beim simulierten Testen via Hardware-in-the-Loop (HIL) wird das zu testende System (Entwicklungssystem) über simulierte Modelle getestet. Im Vergleich zur Test-Hardware (physikalischen Implementierung) hat die Simulation in Bezug auf Test- und Fehlersuche einige Vorteile. Einer der wichtigsten Vorteile sind die Steuer- und Beobachtbarkeit zur Laufzeit. Unter Steuerbarkeit versteht sich die Fähigkeit des Systems zu Steuern. Dies kann sowohl die Zeit (Simulationsstart/-stop) als auch die Daten (Eingangsdaten oder interne Größen) betreffen. Unter Beobachtbarkeit wirddie Möglichkeit verstanden Werte des Systems zu überprüfen. Die Simulation benötigt zur Einrichtung deutlich weniger Zeit als die physikalische Implementierung. (GESSLER 2014, S. 196)

Das simulierte Modell der Umgebung wird dabei meist von einem weiteren Echtzeitrechner, dem sogenannten HIL-Simulator, übernommen. Es ist jedoch auch möglich, den Simulator auf dem gleichen System wie das Testsystem laufen zu lassen, falls das Testsystem es erlaubt (z.B. eine Soft-SPS). Der HIL-Simulator stellt Ein- und Ausgabeschnittstellen zur Verfügung um dem Entwicklungssystem eine korrekte Kontaktierung mit seiner Umgebung zu simulieren. So stellt der HIL-Simulator diejenigen Werte dar, die das Testsystem im Normalbetrieb durch Sensoren erfassen würde. Dem Testsystem wird somit simuliert, dass es in seiner realen Umgebung im Normalbetrieb befindlich sei. Es herrschen ideale Testbedingungen wie aus einem Labor.
Der HIL-Simulator simuliert wiederkehrende Abläufe um das Verhalten der Entwicklungsversion zu testen. Kommt es bei der aktuellen Version zu einem Fehler, so ist nicht nur die Reaktion der Entwicklungsversion bekannt, sondern auch die Gegebenheiten der simulierten Umgebung, die zu dem Fehler führten. In einem zweiten HIL-Test mit verbesserter Software wird dann zunächst genau dieser Testfall durchlaufen, um die aktualisierte Entwicklungsversion mit ihrer Vorversion zu vergleichen und die Funktionstüchtigkeit sicher zu stellen. Diese Testfälle können auch

automatisiert gefunden werden und stellen somit eine theoretisch unendliche Anzahl an Testfällen zur Verfügung, dessen Parameter beinahe beliebig fein eingestellt werden können und dessen Wiederholung präzise durchgeführt werden kann.

2.3.4 Virtuelle Inbetriebnahme

Die virtuelle Inbetriebnahme (VIBN) stellt eine Erweiterung der Simulation mittels HIL dar. Diese Form der Simulation hat besonders hohen Stellenwert in der Automatisierungsindustrie. Hier wird neben einem Modell des Maschinenverhaltens ebenfalls ein 3D Modell der Maschine bereitgestellt, welches interaktiv vom Tester oder Entwickler manipuliert werden kann. Alle Bewegungen im Prozess werden sofort sichtbar und der Entwickler kann dementsprechend eine Beurteilung über die geschriebene Funktion machen. Der Konfigurationsaufwand für diese beiden Modelle ist enorm, weshalb viel geforscht wird um den Aufwand zu reduzieren.

2.4 Model Checking

Im Gegensatz zu den zuvor beschriebenen Testtechniken handelt es sich beim Model Checking um eine Technik zur Verifikation von Software. Eine aussagekräftige Grafik, welche den Unterschied erklärt, findet sich im Buch „Entwicklung eingebetteter Systeme“ und ist in Abbildung 2-12 zu sehen.

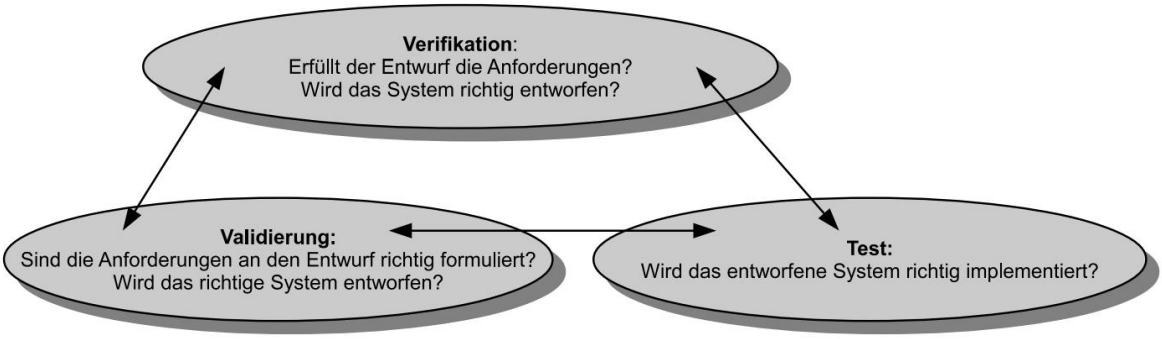


Abbildung 2-12 Einordnung der Begriffe: Verifikation, Validierung und Test (GESSLER 2014, S.192)

Peter Scholz hat eine Erläuterung zum Thema Model Checking in seinem Buch „Eingebettete Systeme“ (SCHOLZ 2005, 204 f.) beschrieben. Die nachfolgenden Passagen zu Model Checking sind aus dem Buch übernommen.

Model Checking wird sowohl im Bereich der Hardwareverifikation wie auch zunehmend im Bereich der Softwareverifikation verwendet. Es ist ebenfalls Logik-basiert und eignet sich besonders zur vollautomatischen Verifikation von Eigenschaften (z. B. Sicherheitseigenschaften) reaktiver Systeme, wie eine SPS. Bei der im Bereich des Model Checkings verwendeten Logik handelt es sich um Varianten der normalerweise im Rahmen eines selbst naturwissenschaftlichen Studiums in der Regel eher selten gelehrten temporalen bzw. modalen Logik. Um Logik-basierte Verifikationssysteme verstehen und benutzen zu können, ist eine gewisse Vertrautheit mit den zu Grunde liegenden formalen Logiken und Kalkülen erforderlich.

Beim Model Checking handelt es sich um Entscheidungsverfahren, um zu prüfen, ob eine gegebene Kripke-Struktur ein Modell für eine angegebene temporallogische Formel ist. Für diese Prüfung ist in der Regel eine erschöpfende Durchsuchung aller Systemzustände erforderlich. Damit liegt die

größte Herausforderung des Model Checkings auf der Hand, nämlich die automatische Verifikation einer in vielen Fällen „explodierenden“ Anzahl von Zuständen. Meist werden sicherheitskritische Aspekte überprüft. Um solche Eigenschaften formal mittels Model Checking nachweisen zu können, müssen sie zunächst unter Zuhilfenahme einer geeigneten formalen Sprache spezifiziert werden. Klassische Logik ist ungeeignet, um die Dynamik veränderlicher (ggf. nichtdeterministischer) Systeme zu beschreiben.

Grundsätzlich wird bei temporallogischen Aussagen zwischen sogenannten Safety- sowie Liveness-Eigenschaften unterschieden. Eine Safety-Eigenschaft (engl. safety property) dient dazu, zu jedem Zeitpunkt und für jede (ggf. nicht-deterministische) Verhaltensalternative die Absenz von unerwünschten Zuständen zu vermeiden, vgl. Aussage. Im Englischen wird dies oft prägnant durch „nothing bad will happen“ zusammengefasst. Eine Liveness-Eigenschaft dagegen stellt sicher, dass irgendwann in einem (ggf. nicht-deterministischen) Ausführungspfad des Systems ein gewünschter Zustand eintritt, siehe. Dies fasst man im Englischen oft kurz mit „eventually, something good will happen“ zusammen. Temporale Logiken stellen hierfür adäquate Formalismen zu solchen Spezifikation dar.

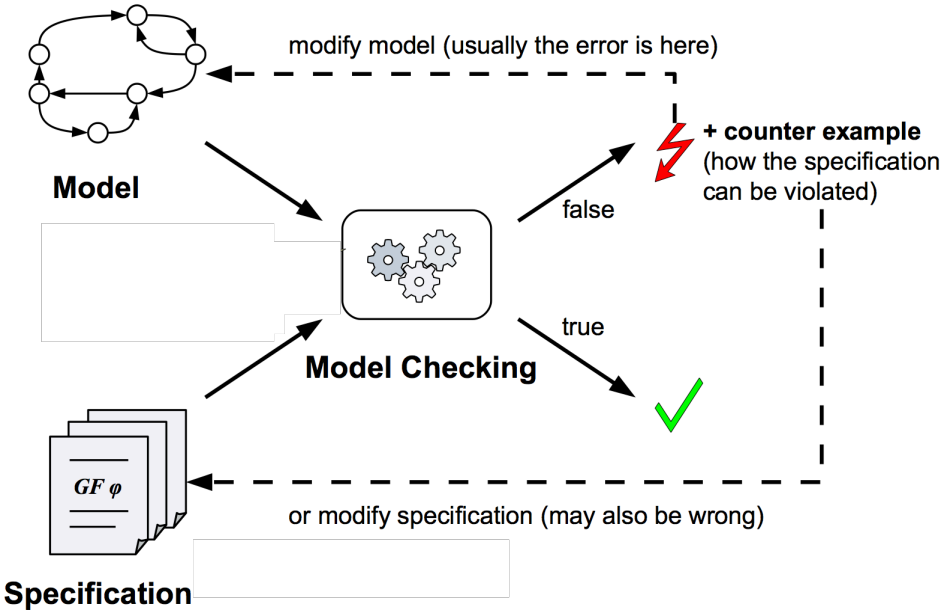


Abbildung 2-13 Model Checking als Blackbox[1]

In der Abbildung 2-13 ist das Vorgehen beim Model Checking nachvollziehbar illustriert. Das Model, welches in Form von z.B. einer Kripke Struktur vorliegt, wird zusammen mit einer Spezifikation (temporalen Logik wie der Computational Tree Language (CTL)) in ein Model Checking Werkzeug gegeben, welches daraufhin alle Zustände des Modells auf die eingegebenen Spezifikationen überprüft und anschließend als Ausgabe bescheinigt ob die Spezifikation dem Modell entspricht oder nicht. Wenn nicht, wird ein Gegenbeispiel in Form einer Folge von Möglichen Schritten im Modell präsentiert, welche die Spezifikation verletzen.

[1] Aus der Vorlesung „Formale Methoden im Software Engineering“ von Prof. Dr. Joel Greenyer (Leibniz Universität Hannover) im Wintersemester 2014/2015

3 Beispiel: Stitcher

Ein qualitatives Beispiel, welches bewusst einfach gehalten ist, wird in diesem Kapitel präsentiert. Es soll dem Leser die Möglichkeit geben, die Arbeitsweise einer SPS in seiner Umwelt zu verstehen. Gleichzeitig soll es als ein durchgehendes Beispiel dienen, dass dem Leser eine Referenz gibt, um im Folgenden die Vorzüge der alternativen Modellierungsartenbesser nachvollziehen zu können.

Der sogenannte Stitcher (deutsch: Hefter) wird bei einer Reifenkonfektionsmaschine dafür verwendet, zwei Lagen Gummi für einen Reifen aneinander zu heften. Dies wird erreicht in dem die Materialien so dicht aufeinander gepresst werden, dass sie nach dem Pressvorgang kaum mehr voneinander zu trennen sind.

Der Stitcher besteht im Wesentlichen aus einer Achse mit gegenläufigen Schlitten auf welchen die „Stitcher“ befestigt sind. Diese Vorrichtung kann hydraulisch über den Hubweg hoch und runter gefahren werden.

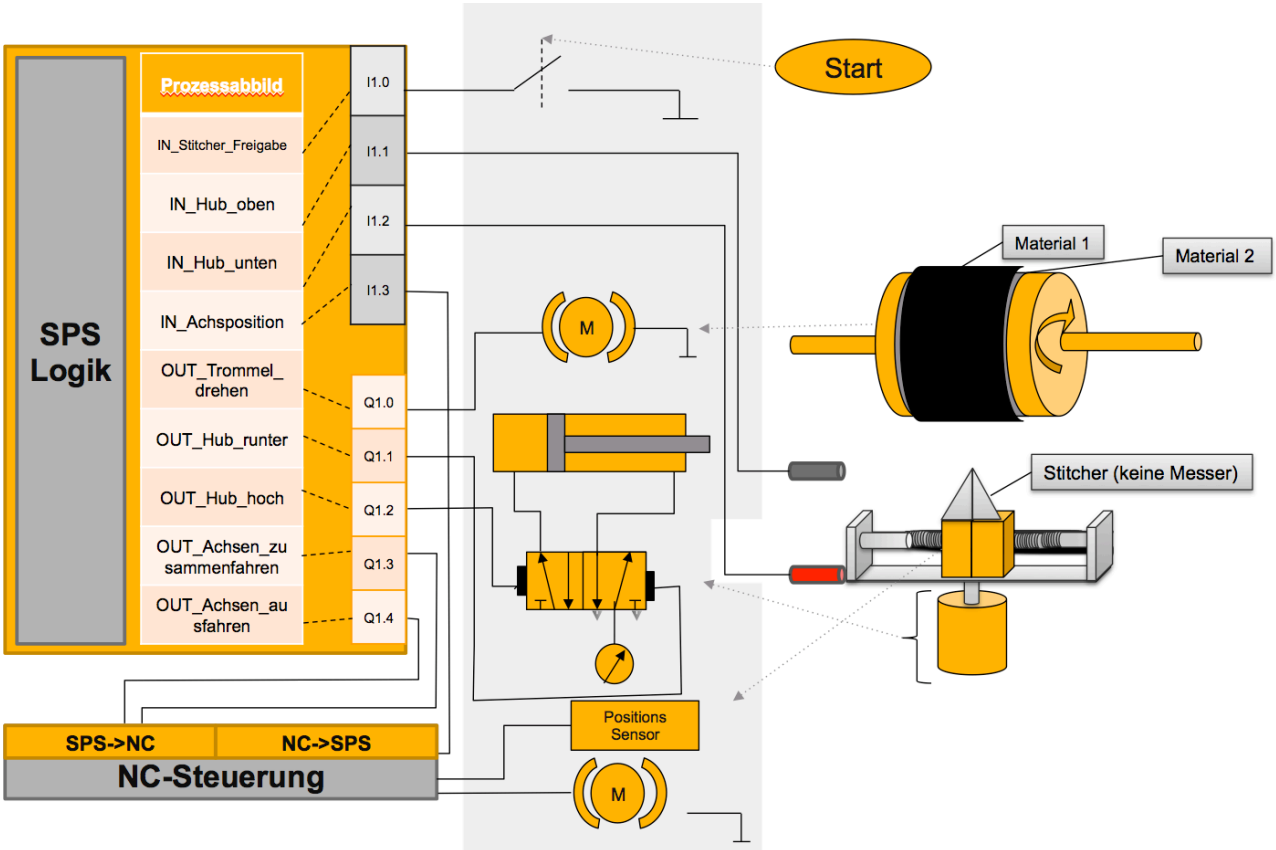


Abbildung 3-1 Stitcher Beispiel aus SPS Sicht, technischer Sicht und als Grafik

Die Abbildung 3-1 zeigt das System qualitativ aus mehreren Ansichten. Die SPS und eine NC-Steuerung für die Achse sind links auf dem Bild zu sehen und rechts daneben in grau unterlegt befindet sich eine beispielhafte technische Realisierung der Ansteuerung der Bauteile. Die grauen Pfeile zeigen jeweils welche technische Realisierung hinter dem Bauteil steht.

Rechts ist der Stitcher zusammen mit einem Startknopf zu sehen, welcher zum Starten des Arbeitszyklus gedacht ist und über einen simplen Kontakt realisiert wird. Die Welle der Trommel wird von einem Motor angetrieben. Der Hub besteht aus einem elektrisch angesteuerten 3/2 Wegeventil welches den Pneumatik-Zylinder durch Umschalten der Druckluftzuführung hoch und

runter fahren lässt. Die Sensoren für die Feststellung der Höhe des Stitchers sind simpel ausgeführte digitale Sensoren. Die Achse wird separat über eine NC-Steuerung kontrolliert. Diese ist ebenfalls mit den Ein- und Ausgängen der SPS verbunden. Technisch realisiert wird die Achse von einem Achsmotor mit Positionssensor, welcher die Ist-Position der Achse bestimmt.

Auf der linken Seite ist das Prozessabbild der Ein- und Ausgänge der SPS zu sehen. Hier werden die Variablen der SPS Logik mit physikalischen Ausgängen der Ausgangskarten der SPS verlinkt.

Die Positionen, die der Stitcher einnehmen kann, werden in Tabelle 3-1 Positionen des Stitchers mit einer kurzen Beschreibung illustriert.

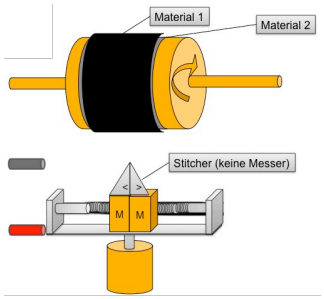
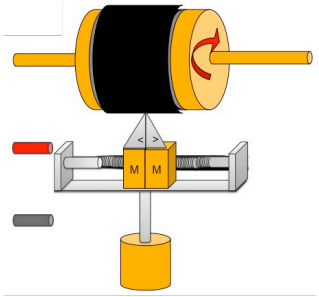
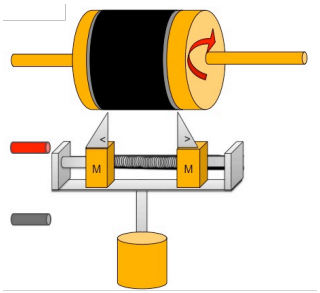
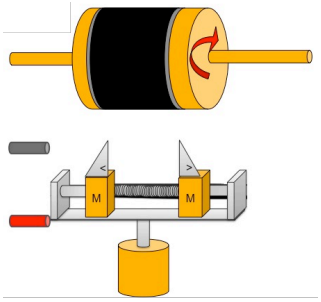
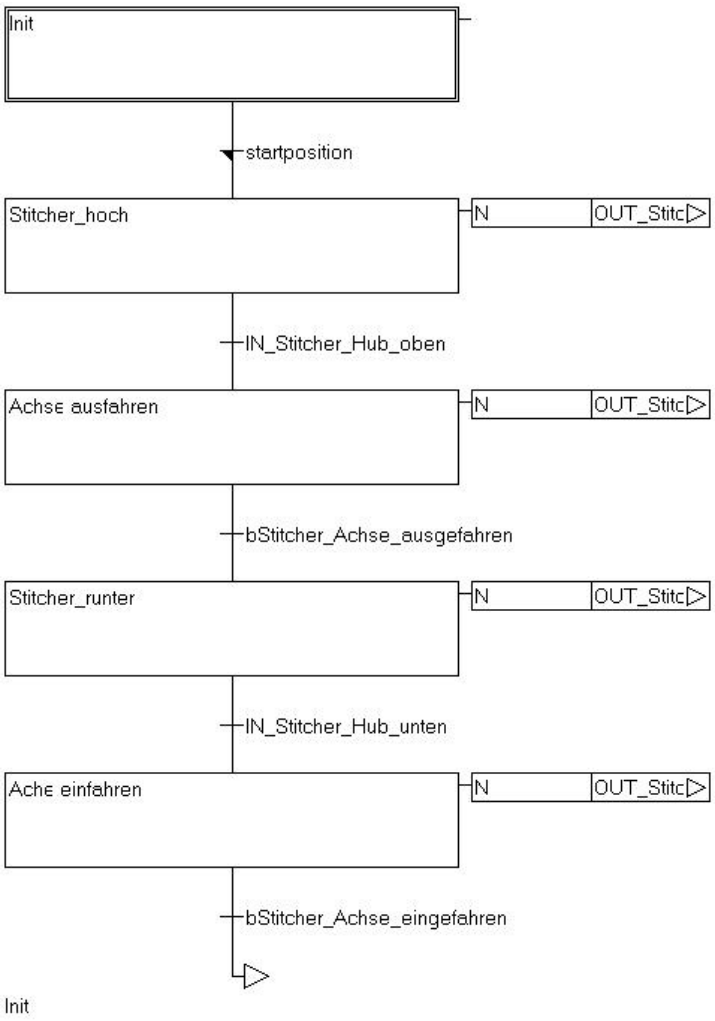
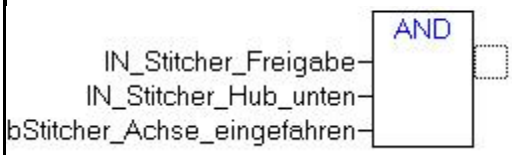
Nr.	Abbildung	Beschreibung
1		Die Trommel mit der Karkasse und den sich darauf befindlichen Materialien fängt an zu drehen. Der Stitcher, welcher sich in der Grundstellung befindet, startet den Prozess mit einer Aufwärtsbewegung.
2		Oben angekommen, befindet sich der Stitcher in der Arbeitsposition Das Ausfahren der Achsen und der darauf befindlichen Platten beginnt.
3		Die beiden Servomotoren auf der Achse bewegen sich voneinander weg, um dabei die beiden übereinander liegenden Materialien stärker aneinander zu heften.
4		Nachdem das geschehen ist, fährt der Stitcher wieder herunter. Unten angekommen, fahren die beiden Servomotoren schließlich zusammen und der Stitcher befindet sich in der Ausgangsposition.

Tabelle 3-1 Positionen des Stitchers

Programmiertechnisch wird der Stitcher in der bereits vorgestellten Ablaufsprache realisiert. Der kommentierte Graph soll helfen, die Implementierung zu verstehen.

3.1 Stitcher in Ablaufsprache

<table>
<tr><td rowspan="7"> 
</td><td>Der Initial-Schritt hat hier keine Auswirkung auf die Steuerung.</td></tr>
<tr><td>Die Transition von Initialschritt zum „Stitcher hoch“ Schritt wird erreicht durch die Logik, welche sich hinter startposition verbrigt:
 

Erst wenn diese erfüllt ist, wird der nächste Schritt ausgeführt.</td></tr>
<tr><td>Der Schritt „Stitcher_hoch“ triggert das Ausgangssignal OUT_Stitcher_hoch, solange es aktiv ist, d.h. bis die nächste Transitions-Bedingung nicht erfüllt ist.</td></tr>
<tr><td>Sobald der Sensor hinter IN_Sticher_Hub_oben ausgelöst hat, wird durch Setzten des Ausgangs OUT_Stitcher_ausfahren die Achse zum Ausfahren bewegt.</td></tr>
<tr><td>Die Variable bStitcher_Achse_ausgefahren wird von der Achssteuerung des Programms gesetzt und leitet mit OUT_Stitcher_runter das Herunterfahren des Hubs ein.</td></tr>
<tr><td>Sobald auch die Achse zusammengefahren ist, gilt der Ablauf als beendet und es wird in den Initial-Schritt gesprungen.</td></tr>
</table>

Tabelle 3-2 Stitcher Beispiel in IEC 61131-3 Ablaufsprache

4 Anforderungsanalyse

Heide Balzert beschreibt die Anforderungsanalyse eines zu entwickelnden Systems als eine der anspruchsvollsten Tätigkeiten der Softwareentwicklung, da der Auftraggeber für gewöhnlich nur eine Idee davon hat was das System können soll aber kein komplettes Model des Systems „im Kopf“ hat (vgl. BALZERT 2005, S.8). Die gestellten Anforderungen sind in der Regel unklar, sehr fallorientiert und teils sogar widersprüchlich. „Die Aufgabe des Systementwicklers ist es aus diesen nicht eindeutigen Wünschen ein konsistentes System zu erstellen, das den Vorstellungen des Auftraggebers entspricht“.

4.1 Die Ist-Situation

Der Ingenieur in der Automatisierungstechnik hat durch seine in der Regel sehr guten Kenntnisse über den Fertigungsprozesses einer Maschine seines Fachbereich sehr schnell eine Idee davon, was das Ergebnis eines zu automatisierten Arbeitsschritts bei gegebenen Eingang sein soll und wie dieser programmiertechnisch erreicht werden kann. Wie genau das Ergebnis in Code umgesetzt wird und wie zu beurteilen ist über die Korrektheit des geleitsteten - ist zum Großteil der rein subjektiven Einschätzung des Ingenieurs selbst überlassen. Während es für die eingebetteten Systeme in der Automobilindustrie mittlerweile bereits eigene Abteilungen für das Testen und Verifizieren des korrekten Verhaltens von Software gibt, verlangte es die Situation in der Automatisierungsindustrie noch nicht diese Extrainvestition zu tätigen, da die Maschinensoftware bislang noch nicht die heutigen Ansprüche an die Software gestellt hat. Anders als bei fahrenden Autos, kann nämlich die automatisierte Maschine durch spezielle Sicherheitsvorkehrungen, welche strikt und systematisch getestet werden, wesentlich einfacher zum Stillstand oder in einen für den Menschen sicheren Zustand versetzt werden.

In der industriellen Praxis wird sich in erster Linie um die Grundfunktionalität bemüht, also das Verhalten der Maschine ohne Komplikationen (vgl. KORMANN et al. 2011). Diese Grundfunktionalität wird in aller Regel schnell erreicht. Viel interessanter ist jedoch die Frage, was passiert, wenn z.B. ein Sensor nicht auslöst oder ein Aktuator sich falsch verhält, also Sonderfälle und generelle Fehlverhalten der Hardware zu betrachten. Das Verhalten der SPS bei Sonderfällen wird aus Erfahrungswerten abgeleitet und programmiert oder erst nachträglich erweitert, wenn der Fall eintritt.

Ob auf solche Situationen während der Implementierung Rücksicht genommen wurde, erschließt sich einem Dritten ohne die Informationen des Entwicklungsingenieurs nur sehr schwer durch das Lesen des Programmcodes oder entsprechendes manuelles Testen.

Ein Test der Software im Zusammenhang des kompletten Systems kann erst mit der aufgebauten Maschine erfolgen. Ein Testen ohne die Maschine ist nicht ohne unterstützende Software möglich. Eine derartige Software, welche das Verhalten der Bewegungen aller Maschinenteile um die SPS herum beschreibt, wird Hardware-in-the-Loop (HIL) Software genannt, da die Hardware in einer Schleife als Software abgearbeitet wird und damit das reale System testet.

Dieser Mangel an Tests ist vielen Faktoren geschuldet. Ein besonders wichtiger Aspekt ist die Zeit: Bis die Vielzahl an zusammenarbeitenden mechanischen, pneumatischen und elektrischen Komponenten aufgebaut, verdrahtet und inspiziert wurden, vergeht bereits eine Menge Zeit und Mühe. Erst darauf folgend kann die Inbetriebnahme einer Maschine gestartet werden. Diese

beinhaltet neben dem Testen der Maschine auch alle Teile korrekt zu konfigurieren, um diese dann zusammen im Takt die korrekten Bewegungen in die vorgegebene Richtungen machen zu lassen. Das Ziel als Ergebnis ein fertiges Produkt herzustellen, wie es z.B. einen Reifen sein könnte; und zwar so automatisch wie möglich.

Die hohe Nachfrage und der hohe Durchsatz an Automatisierung für Maschinen begrenzt zunehmend die zur Verfügung stehende Zeit um an den Maschinen zu testen und neu-entwickelte Funktionalitäten zu evaluieren, da die Maschine schnellstmöglich dem Kunden übergeben werden muss und die nächste Maschine für die Inbetriebnahme bereits wartet.

Daher ist es von großem Interesse, die Zeit des Ingenieurs je Maschine zum Test neuer Funktionalität möglichst kurz zu halten. Mit geschätzten 50% der Entwicklungskosten stellt das Testen der Maschine eine der wohl teuerste Aufgabe dar (HASSAPIS 2000, S. 345). Dabei können meistens keine HIL Umgebungen assistieren, da sie für jedes einzigartige Projekt neu entwickelt werden müssen und aus diesem Grund häufig nicht vorhanden sind.

Die Arbeit, das Verhalten der Maschine für eine HIL-Simulation zu modellieren, kann ziemlich mühsam und zeitraubend sein und wird derzeit als die größte Hürde betrachtet, die den Ausbleib von Simulation im Entwicklungsprozess rechtfertigt. In ihrer Publikation "Why Do Verification Approaches in Automation Rarely Use HIL-Test?" (SCHETININ et al. 2013, S. 1428) geht eine Gruppe aus Deutschland der Frage nach, weshalb der Einsatz von Hardware-in-the-loop Tests in der Automatisierungsindustrie nicht weiter verbreitet ist. Sie erkennen, dass der Benefit des Testens die finanziellen Mehrbelastungen und den zusätzlichen Entwicklungsaufwand fürs Testen nicht erkannt wird und dass die meisten Lösungen stark herstellergebunden sind.

Ebenfalls ist ein weiterer Grund, dass das benötigte Know-How für die Erstellung und die richtige Nutzung der Simulationsmodelle zu meist nicht vorhanden ist. (MATHIAS OPPELT, 1). Im Allgemeinen wird fälschlicherweise angenommen, dass sich der anfängliche Aufwand und die Kosten nicht lohnen würden. Eine genauere Untersuchung hat ergeben, dass 20% der Entwicklungszeit für das Erstellen von Modellen des Maschinenverhaltens akzeptabel wäre. (SUSANNE ROESCH et al. 2014).

Zurzeit werden Tests häufig lediglich auf manuelle Tests vor der Maschinenabnahme reduziert. Das bedeutet, dass die SPS erst in Zusammenspiel mit der Maschine getestet werden kann. Dies ist häufig sehr zeit- und kostspielig, denn es bedeutet, dass auch die kritischen Pfade des Entwicklungsprozesses davon betroffen sind (vgl. MATHIAS OPPELT, S.1; JAMRO 2015, S.1119).

Die Ergebnisse einer Umfrage über aktuelle Test-Praktiken in der Automatisierungsindustrie ist in Tabelle 4-1 zu sehen:

Aspect	Summary
How is testing applied today?	▪ Partially black box tests of function blocks ▪ Tests in combination with visualization (process simulation by manually setting inputs) • Almost no structural testing
Who creates and administrates test cases?	▪ Developers ▪ Test cases are manually created out of requirements, specification or functional description of hardware / software modules ▪ There is no extra testing department • Partially there are no software test cases at all
How much time is used for testing?	▪ There is no explicit time for testing • "Testing" is mostly considered to be part of software construction
What needs to be tested?	▪ Straight forward functionality ▪ Likely and dangerous fault situations ▪ Emergency shutdown ▪ Scenario-based testing ▪ Considering physical effects • Boundary values (Throughput, Speed)

Tabelle 4-1 Ausgewählte Fragen und Antworten aus einer Umfrage zum Thema Testen in der Automatisierungsindustrie (KORMANN et al. 2011)

Beim Testen der Software von SPSen scheint keine Systematik erkennbar zu sein. Die Tests werden nur vereinzelnd manuell von den Entwicklern selbst durchgeführt und dabei ohne vollständige Dokumentation.

Historisch betrachtet, war die Kenntnis über aktuelle Entwicklungsmethoden im Software Engineering nicht von großer Relevanz. Steuerungen konnten mit einer überschaubaren Anzahl von Schaltungen, mittels boolescher Algebra, realisiert werden. Da die Software nun jedoch immer mehr an Bedeutung in der Automatisierung gewinnt, wird neuerdings in diesen Bereich auch vermehrt auf Inhalte des Software Engineerings Wert gelegt.

Aufgrund der sich erst zu etablierenden Methoden aus dem Software Engineering sind Werkzeuge für z.B. Unit-Tests, wie der später beschriebene CodeSys Test Manager (3S SMART SOFTWARE SOLUTIONS GMBH), eher unbekannt und die Einführung von Test-getriebener Entwicklung würde eine Beschreibung des Komplettsystems benötigen, somit Umwelt mit eingeschlossen.

Einige Lösungen zu diesen Problemen sind bereits adressiert. Heute bieten einige Hersteller bereits eine Matlab/Simulink-Schnittstelle und somit umfangreiche Möglichkeiten zur Simulation von Maschinen und Anlagen. Damit können sowohl Regler als auch Steuerungen komfortabel entworfen werden. Die zugehörigen Matlab/Simulink-Programme werden dafür automatisch in SPS-Code übersetzt(MATTHIAS SEITZ 2012).

Der Trend scheint momentan zumindest das Zeitproblem mittels so genannter virtueller Inbetriebnahme zu lösen, also einer 3D Simulation der Maschine, welche das Verhalten der Umwelt für die Handlungen der SPS einprogrammiert bekommt. Das wäre zumindest eine Lösung für die Entwicklung neuer Funktionen, um bereits im Vorfeld ohne die physikalische Maschinenhardware testen zu können. Damit ist jedoch das Problem der nicht dokumentierten und nicht reproduzierbaren Tests genüge getan, welche die Funktionen der Maschinen attestieren.

4.2 Anforderungen

Durch Auswerten der Ist-Situation können die wesentlichen Anforderungen wie folgt grob zusammengefasst werden:

ANF1 Mehr Dokumentation

- Um vorzubeugen, dass nicht genau bekannt ist, welche Funktionen verfügbar sind und welche Funktionen getestet worden sind.

ANF2 Kombinierbarkeit mit virtueller Inbetriebnahme

- Da diese Form der Simulation als die einzige angesehen wird, für die sich die Modellierung des Maschinenverhaltens lohnt, muss eine Lösung auch mit dieser in Zukunft verwendeten Technik kompatibel sein.

ANF3 Zeit an Maschine reduzieren

- Da ein wesentliches Problem darin besteht, dass die Zeit an den Maschinen sehr stark limitiert wird aufgrund der hohen Nachfrage an Automatisierung.

ANF4 Mehraufwand möglichst gering halten

- Da die Vielfalt der Aufgaben der Entwickler Mehraufwand so gut wie nicht zulässt, darf die Lösung diesen möglichst nicht verursachen.

5 Stand der Technik

In diesem Kapitel wird der aktuelle Stand der Technik im Bereich Software-Qualitätssteigerung von SPSen präsentiert. Es haben sich zwei Lösungen als die vielversprechendsten etabliert, was die Anzahl an Publikationen zeigt. Die Erste ist das Automatische Testen der Software von SPSen, meist in Form von Unit Testing der POEs. Zweiteres handelt von einer Lösung für automatisches Model Checking der SPS. Was jeweils für und gegen die im Folgenden vorgestellten Lösungen spricht wird in einer Pro/Contra-Tabelle festgehalten. Die Erfüllung der in 4.2 festgehaltenen Anforderungen wird mittels (+)-Zeichen für „Erfüllt“ und (-)-Zeichen für „Nicht Erfüllt“ illustriert. Dabei gelten folgende Eigenschaften als zwingend zu erfüllende Bedingung für das Werkzeug:

Test Werkzeug			
ANF1	ANF2	ANF3	ANF4
Dokumentation über durchgeführte Testsequenz und Ausgang	Modellierung von Testfall-spezifischen Maschinenverhalten oder keine Modellierung des Maschinenverhaltens	Testausführung ohne Maschine möglich	Möglichst wenig Aufwand für den Entwickler

Tabelle 5-1 Zu erfüllende Eigenschaften zum Erfüllung der Anforderungen für Test Werkzeuge

Model Checking Werkzeug			
ANF1	ANF2	ANF3	ANF4
Dokumentation über geprüfte Eigenschaften in natürlicher Sprache	*(Kein Grund für nicht bestehen)*	Alle IEC 6113-3 werden unterstützt	Integrierte Hilfe zum Formulieren von formalen Spezifikationen

Tabelle 5-2 Zu erfüllende Eigenschaften zum Erfüllung der Anforderungen für Model Checking Werkzeuge

Allerdings haben die meisten der heute verfügbaren Werkzeuge einen gemeinsamen Faktor, welcher eine Investition in diese Lösung verhindert. Dabei handelt es sich um den Wunsch nach einer virtuellen Maschineninbetriebnahme Software, welche die vorhandenen CAD Modelle problemlos in eine interaktiv bedienbare 3D Simulation der Maschine überführt. Der Konsens ist, dass für diese mächtige Art der Simulation der Mehraufwand für die Implementierung des gesamten Maschinenverhaltens gerechtfertigt wäre.

5.1 Virtuelle Inbetriebnahme

Die Einführung einer virtuellen Inbetriebnahme in den Entwicklungsprozess ist das Ziel, welches in großen Teilen der Automatisierungsindustrie verfolgt wird. Auch der Verein Deutscher Ingenieure[2] (VDI) empfiehlt den Einsatz einer virtuellen Inbetriebnahme in der Richtlinie 4499 (BRACHT et al. 2011). Es sind bereits viele Lösungsansätze und kommerziell verfügbare Alternativen vorgestellt

[2] https://www.vdi.de/

worden. Nichtsdestotrotz hat sich im Test bis zum heutigen Zeitpunkt keine Lösung für hochkomplexe Maschinen bewehren können.

Anfänglich ist das Modellieren und Konfigurieren der 3D Simulation und des Maschinenverhalten eine sehr teure und komplizierte Maßnahme. Laut Wünsch (WÜNSCH 2008) würde sich dieses jedoch schnell amortisieren, sofern die vermiedenen Kosten zur Fehlerbeseitigung und das frühere Erreichen der Gewinnzone betrachtet werden würde. Im Rahmen seiner Dissertation über Untersuchungen zur Methodik virtueller Inbetriebnahme, erkannte Wünsch, dass die VIBN viele Vorteile bringen würde:

- Höhere Motivation der Ingenieure, da sie ihre Software bereits in einer frühen Projektphase testen können und zusätzlich ein visuelles Feedback bekommen
- Geringere Reisekosten und familienfreundliches Arbeiten durch verkürzte Inbetriebnahme vor Ort
- Kapazitätstests einfach möglich
- Beschädigungen der realen Anlage werden vermieden
- Unproduktives Warten wird vermieden, da die Steuerung unabhängig von anderen Bereichen getestet werden kann
- Alternative Steuerungskonzepte können gefahrlos am Modell getestet werden

Im Prinzip erfüllen viele auf dem Markt verfügbare Software für die virtuelle Inbetriebnahme den Großteil der zuvor genannten Anforderungen und Kriterien. Mit den zusätzlichen Punkten von Matthias Oppelt und Leon Urbas (MATHIAS OPPELT, S. 6) ergeben sich die folgenden zusätzlichen Anforderungen:

- Automatische Überführung der Computer Aided Modelling Design (CAD)-Modellen der Maschinen-Konstriktionszeichnungen in die 3D Simulation
- Assistierte Konfiguration der Simulationsbewegungen
- Ausführbarkeit mit der echten SPS
- Ein skalierbares Simulationsmodell
- Effizientes Simulationsdesign mit Vorlagen und Interfaces
- Management für Testfälle und Dokumentation

Nach aktuellem Kenntnisstand erfüllt keines der verfügbaren Lösungen alle diese Anforderungen. Es werden jedoch mehrere vielversprechende Alternativen auf ihre Tauglichkeit und Wirtschaftlichkeit geprüft. Beispiele hierfür sind SimulationX[3], industialPhsics[4], WinMOD[5], ISGvirtous[6] oder Simulink 3D Simulation[7]. TwinCat selbst bietet in seinem Produktportfolio zwar mit TC3 Ethercat Simulation Tool[8] eine Lösung für die virtuelle Inbetriebnahme an, es wird jedoch keine Lösung für eine 3D Simulation angeboten. Alle der genannten Lösungen erlauben eine Beschreibung des Maschinenverhaltens in IEC 61131-3 Sprachen.

[3] https://www.simulationx.de/simulationssoftware/simulationsansatz.html
[4] http://www.machineering.de/sps-simulation.html
[5] http://www.winmod.de/de/
[6] http://www.isg-stuttgart.de/de/isg-virtuos/informationen.html
[7] http://de.mathworks.com/products/3d-animation/
[8] https://www.beckhoff.com/english.asp?twincat/twincat_simulation_manager.htm

5.2 Option: Automatisches Testen

Das automatische Testen von System und Software ohne dabei die Hardware zu benötigen ist ein sehr wichtiges Mittel, um die Software-Qualität zu steigern. Bei eingebetteten Systemen aus der Automobil und Raumfahrtindustrie werden solche Tests seit langer Zeit praktiziert, jedoch hat sich das Testen der Software von automatisierten Maschinen in der Industrie bislang noch nicht durchsetzten können. Wie bereits erläutert sind laut Benjamin Kormann (KORMANN et al. 2011) die Hautgründe hierfür der Mehraufwand und die anfänglich zusätzlichen Kosten für den Einsatz dieser Art von Tests.

Die Natur von automatischen Tests legt direkt nah, dass sie ohne Maschine getestet wird. Solche Tests helfen nicht nur zur Steigerung der Software Qualität sondern motivieren zusätzlich auch die Entwickler durch eine schnelle Bestätigung über funktionierende Testfälle. Es ist erstaunlich, dass automatische Testkonzepte, wie Unit Testing, nicht verbreitet sind in der SPS Welt, obwohl sie laut Jamro (JAMRO 2015, S.1119) generell als sehr hilfreich angesehen werden. Weiterhin lag der Einsatz von Unit-Tests und testgetriebene Entwicklung für den Bereich der SPSen bislang nur im Fokus von einigen wissenschaftlichen Arbeitsgruppen. Wirklich eingesetzt wird das Konzept nur in der Anwendungsentwicklung.

Nur an der Maschine zu testen kann neben dem Zeitverlust ebenfalls zu erheblichem Schaden an der Maschine führen. (A. BECKMANN et al., S. 288). Zurzeit wird das Testverhalten häufig zusammen in den Steuerungscode integriert, was zu einer sehr schlechten Lesbarkeit des gesamten Codes führt. Dies ist sehr ungünstig, da dieser besonders im Bereich der Automatisierung von vielen anderen Personen neben dem Entwickler gewartet und verstanden werden muss (END USER COMPUTING SERVICES 2009).

In der Welt des Software Engineerings von Desktop-Anwendungen haben sich über die letzten Jahre hinweg viele Praktiken durchgesetzt, welche Entwicklung von Software-Systemen schneller, effektiver und robuster machten. Zum Beispiel die testgetriebene Entwicklung oder die Verwendung von bewährten Mustern, um bestimmte, wiederauftretende Probleme effizient zu lösen (RAMLER et al. 2014). Insbesondere für Software-Test haben sich bereits etliche Test-Frameworks etabliert, die zusammen beinahe die Gesamtheit aller Hochsprachen unterstützen. Nun werden SPSen generell in speziell-angepassten Sprachen (Aus der IEC 61131-3 Norm) programmiert, welche bislang noch nicht zufriedenstellend bei diesem Schritt unterstützt werden. Die Entwicklung dorthin ist jedoch sehr ambitioniert, was die Anzahl der wissenschaftlichen Publikationen andeutet.

Eine Implementierung für die Beschreibung des Maschinenverhaltens für einen Testfall könnte in IEC 61131-3 so aussehen (am Stitcher Beispiel):

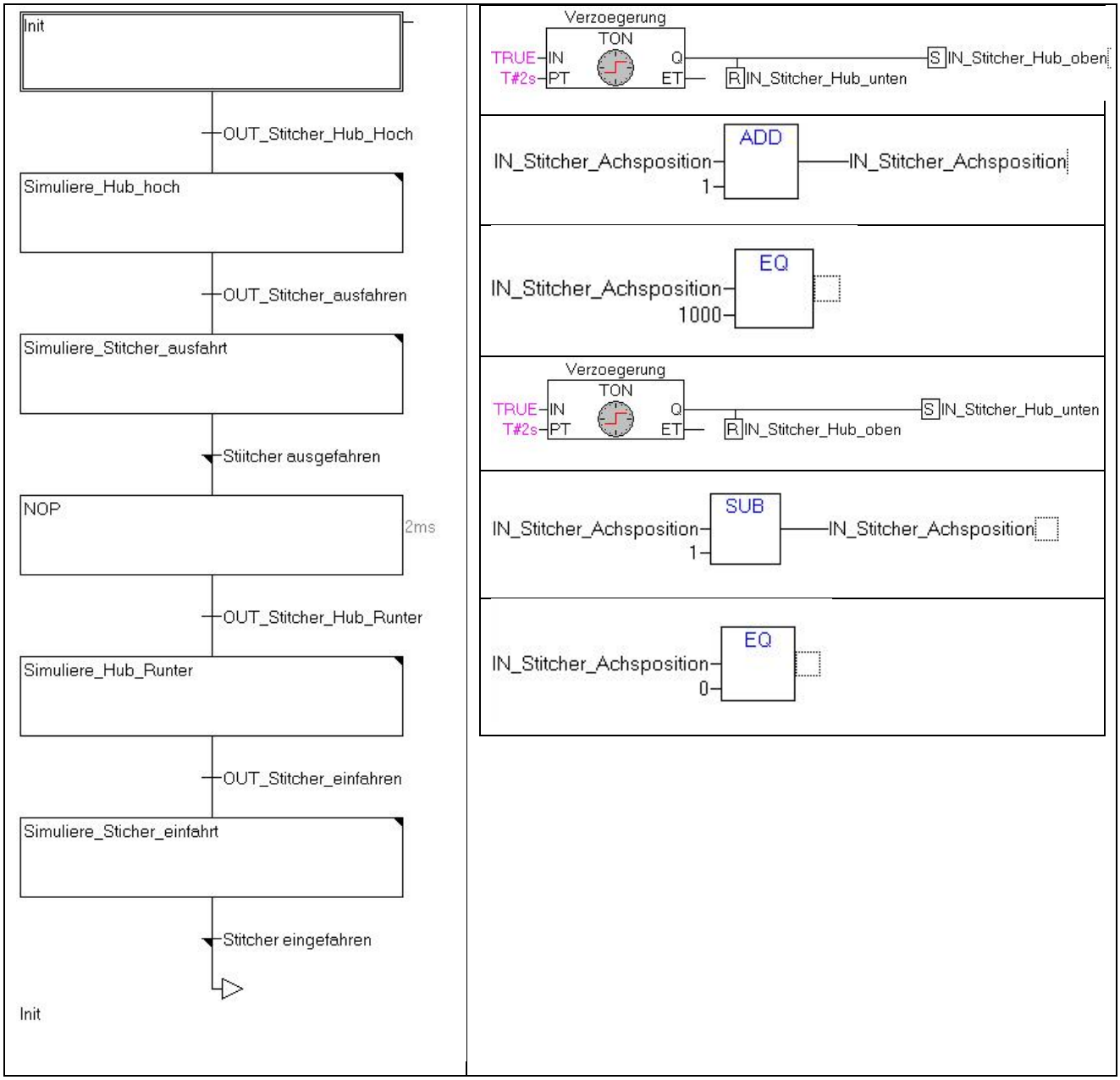


Tabelle 5-3 Maschinenverhalten für Testfall beschrieben in IEC 61131-3

Die linke Tabelle zeigt die Aktions-Funktionsbausteine welche sich hinter den Schritten und Transitionen verbergen. Bausteine die Aktionen durchführen, sind mit einem kleinen Dreieck in der rechten oberen Ecke versehen und bei Schritten mit einem kleinen Dreieck links neben dem Transition-Symbol gekennzeichnet. Die zeitliche Reihenfolge ist in der Schrittkette links abgebildet.

Diese Implementierung kann zusammen mit der in 3.1 vorgestellten Stitcher Implementierung ohne die Anwesenheit von echter Hardware ausgeführt werden und würde somit das System vervollständigen. Da eine SPS oft auch zeitliche Eigenschaften einhalten muss, wird der Stitcher Implementierung das Warten mittels Einschaltverzögerung vorgespielt bis der Hub hochfährt.

Natürlich ist so noch kein Testfall mit Dokumentation erschaffen. Die jeweiligen Lösungen bieten für die Auswertung noch weitere Funktionsbausteine an. Das Beispiel soll nur eine Idee dafür sein, wie es in der Praxis getestet werden kann.

5.2.1 Spike Prototype

Im Rahmen der Recherche ist nur ein einziges kommerziell erhältliches Produkt gefunden worden, welches für das automatische Testen von TwinCat SPS Systemen verfügbar ist: Spike Prototype von HAL Software[9]. Dieses Tool ist primär zum Erstellen von Prototypen zur Präsentation gedacht; allerdings unterstützt es in fast allen Phasen des Engineering Prozesses.

ANF1	ANF2	ANF3	ANF4
+	+	+	−
Was spricht dafür		Was spricht dagegen	
• Sehr gut zum Erstellen erster Prototypen • Dokumentiert automatisch ausgeführte Testfälle		• Modellierung von Tests in der Programmiersprache Phyton	

5.2.2 CPTest+

Einige Ausarbeitungen stellen Domain Specific Languages (DSLs) vor, welche den Fokus auf das Testen von speicherprogrammierbaren Steuerungen legen. Der aktuellste Forschungsstand zu diesen Thema ist in der Publikation „POU-Oriented Unit Testing of IEC 61131-3“ von Marcin Jamro zu finden, einem Mitglied des Institute of Electrical and Electronics Engineers (IEEE (JAMRO 2015).

Seine CPTest+ Sprache hat einen umfangreichen Befehlssatz durch welchen Funktionsblöcke zum Testen von Programmen des IEC 61131-3 Standards generiert und in den Code integriert werden. Dabei können Mock-Objekte erzeugt und viele nützliche Eigenschaften überprüft werden. Der Fokus liegt bei seiner Arbeit auf Unit Tests an POEs mit dem Ziel, dass mithilfe seiner Testsprache die Testgetriebene Entwicklung erleichtert wird.

Ein bekanntes Problem ist, dass sich Fehler einer POU auf andere POUs auswirken können. Deshalb können Fälle auftreten in denen ein Testfall nicht bestanden wird, obwohl das POU korrekt ist.

ANF1	ANF2	ANF3	ANF4
−	+	+	−
Was spricht dafür		Was spricht dagegen	
• Testfälle werden in einem für Ingenieure sehr verständlichen Formalismus geschrieben, der speziell zum Testen ausgelegt		• POU Fehlerfortpflanzung • Nur für CPdev Entwicklungsumgebung • Noch sehr experimentell	

5.2.3 XFEL Lösung

Am Forschungszentrum European x-ray free electron laser (XFEL) wurde in der Ausarbeitung “Automated Verification Environment for TwinCat PLC” ein Framework zum Testen von TwinCat Systemen vorgestellt. (A. BECKMANN et al., S. 288).

[9] http://hal-software.com/spike-prototype/

ANF1	ANF2	ANF3	ANF4
+	+	+	−
Was spricht dafür		Was spricht dagegen	
• Erfolgreich bei großen Projekten getestet worden sind • Dokumentiert automatisch ausgeführte Testfälle		• Modellierung von Testfällen in der Programmiersprache Python	

5.2.4 Modellbasierte Fehlereinpflanzung

In einem Paper werden gewollte Fehlereinpflanzungen zu Testzwecken eingesetzt. Interessant hierbei ist, dass es sich um ein modellbasiertes Test-Framework handelt, bei welchen UML Sequenzdiagramme via Drag and Drop umgestaltet werden können. Solche Techniken werden in der Automobilindustrie schon länger erfolgreich eingesetzt, jedoch noch nicht in der Automatisierung (SUSANNE ROESCH et al. 2014).

ANF1	ANF2	ANF3	ANF4
−	+	+	−
Was spricht dafür		Was spricht dagegen	
• Einfache Modellierung und Änderung von Testfällen		• Nicht auf großen Projekten erprobt • Noch sehr experimentell	

5.2.5 Der Codesys Test Manager

Mit dem CODESYS Test Manager stellt die 3S Software Solutions GmbH den bislang einzigen vollständig in die Benutzerumgebung integrierten Test Manager zur Verfügung. Er dient zur Erstellung, Durchführung und Auswertung von Tests. Der Testcode für jeden Testfall wird komplett in IEC 61131-3 erstellt und wird zusammen mit der Steuerung in Echtzeit abgearbeitet. Ebenfalls ist eine Testfallverwaltung mit integriert und alle Tests können automatisch abgespielt werden.

Die Modellierung von mehreren Testfällen würde viel Zeit in Anspruch nehmen und die Modelle der Umwelt sind für den jeweiligen Testfall maßgeschneidert. Das bedeutet, unter der Prämisse der Lauffähigkeit der späteren virtuellen Inbetriebnahme auf Basis einer IEC 61131-3 Simulation, dass dies für die Umwelt nicht einsetzbar wäre. (3S SMART SOFTWARE SOLUTIONS GMBH)

ANF1	ANF2	ANF3	ANF4
+	+	+	−
Was spricht dafür		Was spricht dagegen	
• Modellierung von Tests in IEC 61131-3 Programmiersprachen • Vollintegriert in Benutzerumgebung		• Nur für CodeSys • Nicht eindeutig geklärt, ob die Testmodelle bei der VIBN eingesetzt	

	werden können.

Zusammenfassend lässt sich zu den Lösungen sagen, dass sie alle gut durchdacht sind und hilfreich in der Entwicklung wären. Einige davon sind zunächst sehr experimentell und stellen derzeit noch keine Testversion zur Verfügung. Somit ist es schwierig ein genaues Bild der Gesamtsituation zu erhalten. Mit der Idee einer zukünftig eingeführten virtuellen Inbetriebnahme sind jedoch noch einige Punkte zu klären: Unklar ist, ob die Modelle für das Maschinenverhalten, welche implementiert werden müssen, in der VIBN eingesetzt werden können.

5.3 Option: Model Checking

Algorithmische Verifikation, oder auch Model Checking, ist eine Technik, die Algorithmen benutzt, um ein globales Model des Gesamtsystems auf die Einhaltung von formal definierten Anforderungen zu überprüfen. (E.B. BLANCO VINUELA et al. 2014, S.1259).
Es gibt mehrere Möglichkeiten, wie etwa Simulation, Testen oder formale Verifikation, um die Korrektheit eines eingebetteten reaktiven Softwaresystems zu überprüfen. Simulation und Testen sind die derzeit in der Praxis am weitesten verbreiteten Ansätze. Eine hundertprozentige Korrektheit der Spezifikation mit diesen Methoden nachzuweisen, ist allerdings so gut wie unmöglich. Die vollumfängliche Übereinstimmung des Softwareverhaltens mit der Anforderungsspezifikation kann nur durch formale Verifikation sichergestellt werden (SCHOLZ 2005, 204).

Mit Testen allein ist es für gewöhnlich unmöglich, alle existenten Kombinationen von Pfaden bzw. Zuständen zu testen, welche die Controller-Software einschlagen könnte. Mit formaler Verifikation ist das möglich. Somit können theoretisch Fehler schon bereits vor der Implementation während der Entwurfsphase eliminiert werden. Die generelle Idee ist es, ein formales Modell des realen Systems zu erstellen und dieses mittels Model Checker Tools gegen formal definierte Anforderungen zu prüfen, welche bestimmte Eigenschaften des realen Systems beschreiben(E.B. BLANCO VINUELA et al. 2014, S. 1259).

In der folgenden Tabelle werden die Vor- und Nachteile der beiden Methoden „formale Verifikation“ und „Automatischer Test“ gegenübergestellt:

	Advantages	**Disadvantages**
Automatic Testing	Testing with the real system Technology is available Reduce human errors Reusable for different PLC	Sophisticated maintenance High price for new test case Black box testing Difficult to find the source of the problem
Model Checking	Explores all the combinations Earlier bug detection Avoid human errors Complexity hidden by the generation tools Counter-examples to find the source problem	Verification of a system model Need of automatic generation tools Need to prove the transformations State space explosion Applying abstractions techniques is not trivial

Tabelle 5-4 Vor- und Nachteile beim Verwenden von Automatischen Tests und Model Checking (E.B. BLANCO VINUELA et al. 2014, S. 1261)

Das Forschung-Team, welches diese Tabelle (Tabelle 5-4) aufgestellt hat, schrieb weiterhin, dass Automatisches Testen ein besser umsetzbarer praktischer Ansatz ist, aber die Absicherung mittels

formaler Verifikation eine gute Aussicht für die Zukunft wäre (E.B. BLANCO VINUELA et al. 2014, S.1261). Die Lösung dieses Teams vom CERN[10] wird im späteren Verlauf vorgestellt.

In einer Untersuchung von Model Checking Praktiken (OVATMAN et al. 2014, S. 3) haben die Beteiligten drei Gründe für die hervorragende Eignung des Model Checkings für SPSen genannt:

1. Der SPS Code lässt sich auf aussagenlogische Formeln und Zustandsübergangsautomaten überführen.
2. SPS Programme laufen grundsätzlich parallel ab.
3. SPS Programme werden meistens in Echtzeitsystemen verwendet. Dies macht die formale Verifikation noch bedeutender, da solche Systeme meist sicherheitsrelevant sind.

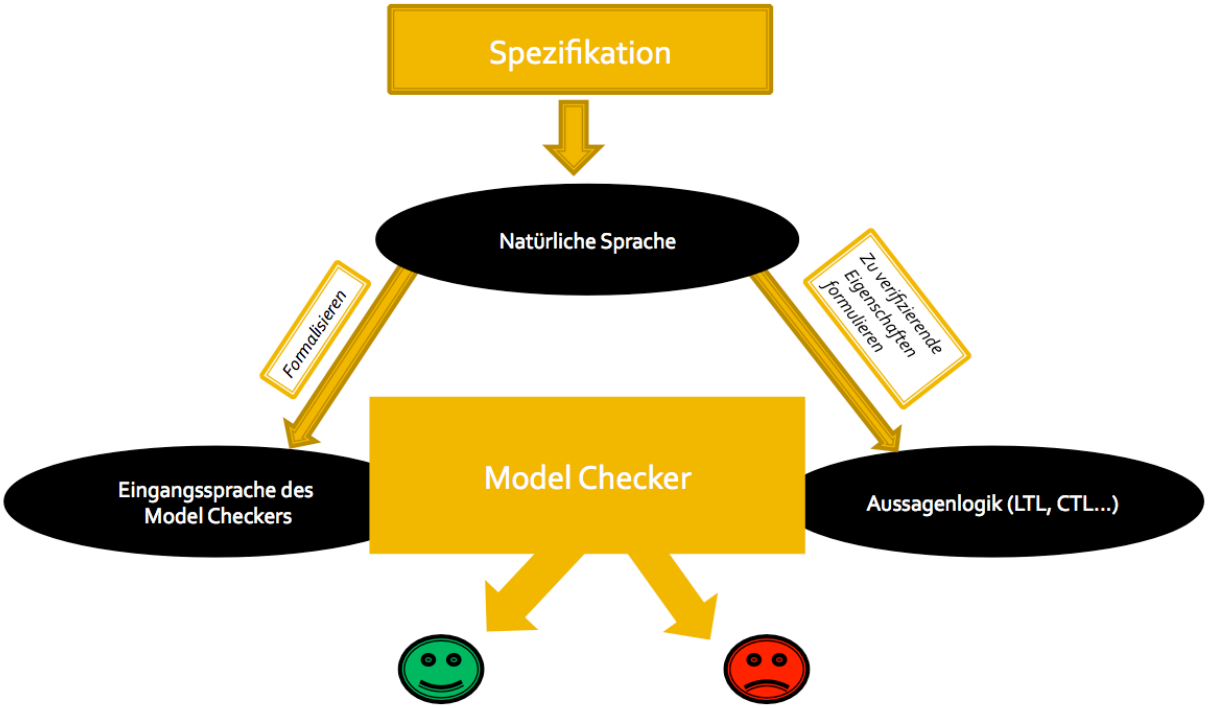


Abbildung 5-1 Prinzipielles Vorgehen zum Formalisieren beim Model Checking

In Abbildung 5-1 ist der üblich durchzuführende Prozess zu sehen, welcher für eine formale Verifikation nötig ist. Es geht hierbei also im ersten Moment nicht um Verifikation eines laufenden Programms sondern um ein Modell davon. Aus Grundlage des Modells wird dann anschließend die Implementierung vorgenommen.

Das Prinzip müsste im Entwicklungsprozess demnach so integriert werden, dass die Spezifikation, welche beispielsweise in einer Besprechung festgelegt wurde, vom Ingenieur in die Eingangssprache (Modelle) eines Model Checkers überführt wird. Dies kann sich je nach Model Checker mehr oder minder schwer gestalten, benötigt in jeden Fall Übung. Weiterhin müssen die Eigenschaften in einer temporalen Logik wie LTL spezifiziert werden. Ein Schritt mit viel Mehraufwand, welcher in der Praxis nicht gerne gegangen wird.

Um das Problem besser verstehen zu können soll ein Beispiel in LTL helfen:

[10] https://home.cern/

Wenn eine Anforderung beim Stitcher lauten würde „Der Stitcher darf niemals in der Arbeitsposition zusammenfahren“ würde der Entwicklungsingenieur in der temporalen Logik LTL die folgende Formel definieren:

```
1.  G( IN_Hub_oben -> NOT OUT_Achsen_zusammenfahren)
```

Die Einzelheiten der LTL Logik sollen an dieser nicht weiter erläutert werden. Übersetzt werden kann diese Formel etwas verständlicher als:

„Es ist immer der Fall, dass wenn IN_Hub_oben wahr ist, OUT_Achsen_zusammenfahren unwahr ist“.

Wie man die formale Verifikation in die Softwareentwicklung von SPSen in die Praxis einführen kann, wird bereits erforscht. Es sind dabei mehrere Konzepte für eine Lösung entstanden. Die generellen Ideen, die dabei verfolgt werden, lassen in der folgenden Abbildung zusammenfassen:

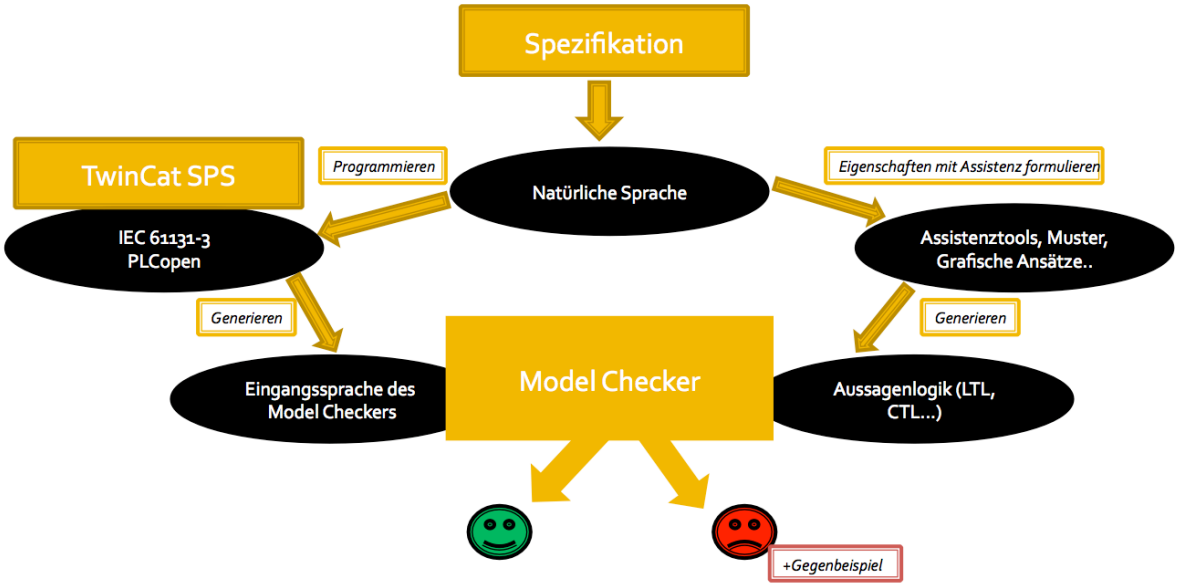


Abbildung 5-2 Vereinfachtes Vorgehen zum Model Checking bei SPSen

- Automatische Überführung von SPS Code in die Eingangssprache eines Model Checkers
- Vereinfachte Formulierung von formalen Spezifikationen.

Anstatt die Modellierung von formalen Modellen für einen Model Checker selbst zu erstellen, werden diese aus programmierten SPS Code transformiert. Der Entwickler selbst hat also keine zusätzliche Aufgabe, da die Software wie gewohnt geschrieben wird. Komplizierter ist es allerdings mit den zu überprüfenden Eigenschaften. Diese müssen nämlich aus der Spezifikation abgeleitet werden. Nachfolgend werden die erfolgversprechendsten Lösungen präsentiert.

5.3.1 Lösungen für die Formulierung von Spezifikationen

Die Eigenschaften, welche aus den Spezifikationen für den Model Checker in temporale Logiken überführt werden müssen, sind meist schwer zu erfassen. Temporale Logiken sind nicht sehr gut geeignet für den Einsatz in der Automatisierungstechnik, da die Ingenieure in der Regel keinen Informatik-Hintergrund haben (OSKAR LJUNKRANTZ et al. 2011, S. 1). Es werden im Folgenden die verschiedenen Möglichkeiten zur assistierten Formulierung von formalen Spezifikationen aufgezeigt. Da es sich mehr um ein Assistenzwerkzeug handelt, wird auf eine Bewertung über Erfüllung der Anforderungen verzichtet.

5.3.1.1 *AUTILI et al. 2015*

In einem an der Swinburn University of Technology entwickelten Werkzeug mit dem Namen *PSPWizard*[11] wird bei der Zusammensetzung von formalen Spezifikationen in vielen temporalen Logiken geholfen. Das Werkzeug bedient sich hierfür aus einem Katalog von Mustern, welcher Eigenschaften und Verhalten auflistet, die häufig in verschiedenen Branchen spezifiziert wurden und werden. Der Katalog ist bereits sehr umfangreich und ist benutzerspezifisch erweiterbar.

Es werden durch gezielte Fragen eine formale Spezifikation in englischer Sprache und gleichzeitig im gewünschten Formalismus zusammengestellt. Die folgende Abbildung zeigt die Benutzeroberfläche und gibt einen Überblick über die Idee und die Funktionalität.

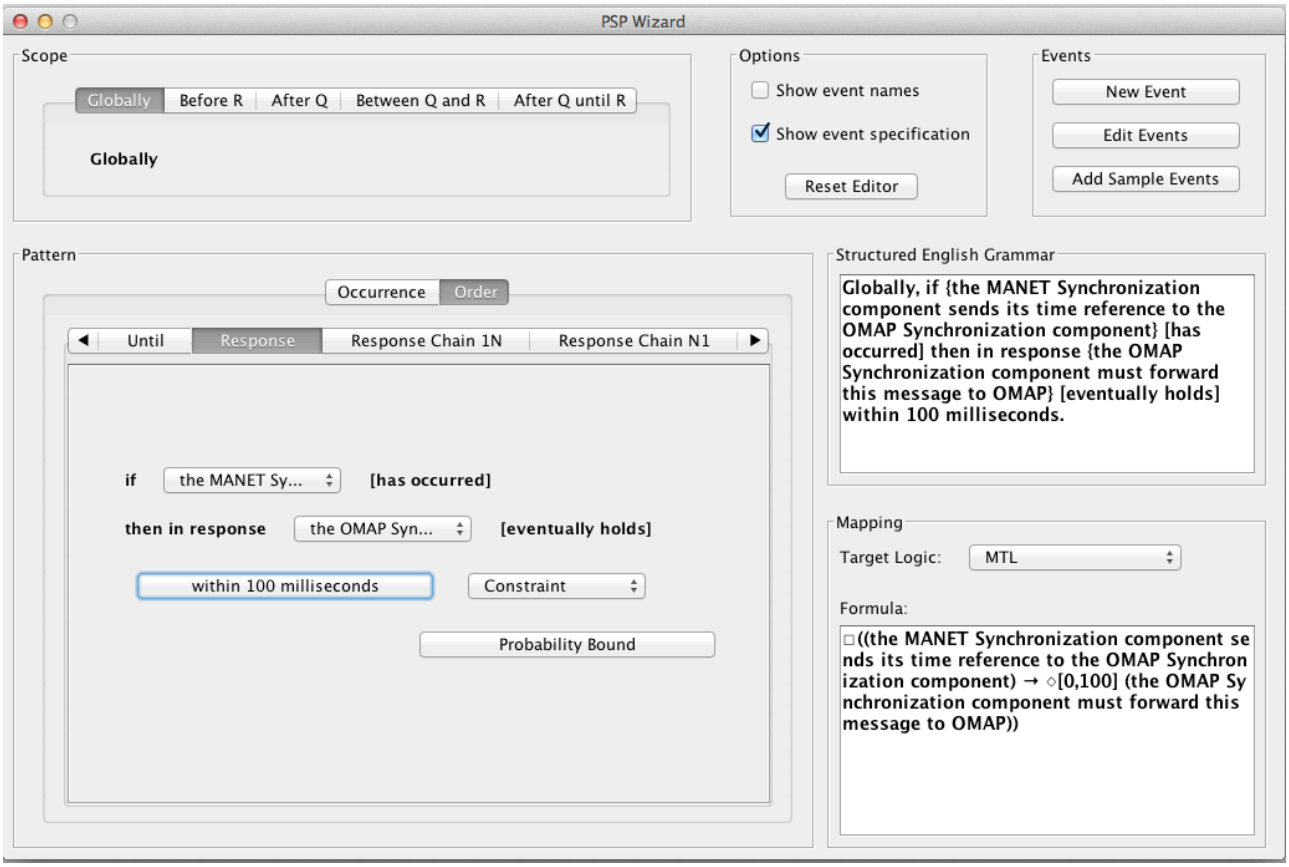


Abbildung 5-3 Benutzeroberfläche des PSP Wizard Werkzeugs

Eine ähnliche auf Mustern basierende Lösung wird in einer weiteren Publikation vorgestellt (FERANANDEZ ADIEGO et al. 2014). In dieser Arbeit wird nur die Lösung der Model-Transformation dieser Lösung vorgestellt.

Was spricht dafür	Was spricht dagegen
• Unterstützt das Formulieren von Formeln in eine Vielzahl von temporalen Logiken • Kann mit eigenen Mustern erweitert werden	• *(Keine Negativ Punkte)*

[11] http://www.ict.swin.edu.au/personal/mlumpe/PSPWizard/

5.3.1.2 REMENSKA et al. 2014

Das Werkzeug PASS (Property ASSistant), welches Teil des mCRL2[12] Toolset der gleichnamigen formalen Spezifikationssprache ist, verwendet UML Diagramme für die Beschreibung von Eigenschaften zur formalen Verifikation. Die Modelle werden transformiert in die temporale Logik µCalculus.

Der Einsatz von UML Diagrammen ist sehr interessant, es gibt allerdings kein Werkzeug welches IEC 61131-3 Implementierungen in die Eingangssprache des mCRL2 Model Checkers überführt.

Was spricht dafür	Was spricht dagegen
• Modellierung mittels UML Diagrammen ist ein sehr interessanter Ansatz	• µCalculus wird sehr selten bei Model Checkern als temporale Logik verwendet • Keine Transformationslösung für die mCRL2 Spezifikationssprache

5.3.1.3 Oscar Ljungkrantz et al. 2011

Eine speziell auf die Bedürfnisse der SPS-Entwickler angepasste formale Spezifikationssprache stellt ST-LTL der Chalmers University of Technology (Göteberg, Schweden) dar. Anhand einer Sprache, die an der IEC 61131-3 Structured Text angelehnt ist, werden hier die zu verifizierenden Eigenschaften beschrieben. Das LTL Beispiel mit dem dieses Kapitel eingeleitet wurde, würde in ST-LTL wie folgt aussehen:

```
2. Spec := ALWAYS(IN_Hub_oben -> NOT OUT_Achsen_zusammenfahren)
```

Was spricht dafür	Was spricht dagegen
• Sprache die speziell für Ingenieure in der Automatisierungstechnik entworfen wurde	• Nicht öffentlich verfügbar • Nur für LTL

5.3.2 Lösungen für die Automatische Umformung von SPS Code

Bei der Umsetzung dieses Ablaufs ist die automatische Übersetzung des IEC 61131-3 Codes in die formalen Modelle eine der schwerwiegenderen Probleme. Weiter sind Timing Aspekte dabei zu berücksichtigen (OVATMAN et al. 2014, S. 19). Bei den formalen Modellen handelt es sich in diesem Fall um die Eingangssprache eines Model Checkers. Aus heutiger Sicht, kann behauptet werden, dass SMV[13] und UPPAAL[14], die meistverwendeten Model Checking Werkzeuge sind wenn es um SPSen geht. Der Hauptunterschied zwischen SMV und UPPAL ist die Möglichkeit Echtzeit-Clock in das Model fürs Model Checking zu integrieren in UPAALL. Andererseits ist dafür das uneingeschränkte Model Checking von Zeiteigenschaften in SMV möglich. (OVATMAN et al. 2014, S. 8).

Am meisten wird die IEC 61131-3 Sprache Structured Text zur Transformation in die Eingangssprache eines Model Checkers verwendet, da fast alle anderen Sprachen im Bereich der SPSen in diese C-ähnliche Sprache übersetzt werden können. (OVATMAN et al. 2014, S.4).

[12] http://www.mcrl2.org/
[13] http://nusmv.fbk.eu/
[14] http://www.uppaal.org/

Es existieren allerdings auch etwas ausgereiftere Werkzeuge, welche folgend vorgestellt werden.

5.3.2.1 THRAMBOULIDIS et al., 2011

In diesem an der Saarland Universität (Saarbrücken, Deutschland) und der University of Patras (Griechenland) entwickelten Werkzeug werden Transformationen vom PLCopenXML[15] Format in die Eingangssprache des UPPAAL Model Checkers transformiert. Das Werkzeug ist aktuell noch nicht verfügbar.

ANF1	ANF2	ANF3	ANF4
−	+	−	+
Was spricht dafür		Was spricht dagegen	
• Durch das PLCopenXML Format ist es Herstellerunabhängig		• Noch nicht öffentlich verfügbar • Nur für Funktionsbausteine	

5.3.2.2 FERANANDEZ ADIEGO et al. 2014

Eine Lösung zur formalen Verifikation der SPS Anlagen am Teilchenbeschleuniger CERN in der Schweiz stellt das Werkzeug PLCverif[16] dar und ist die bisher gründlichste Umsetzung. Es handelt sich um ein Eclipse[17] basiertes Werkzeug, welches die beiden Hauptprobleme (siehe Abbildung 5-2) zu lösen versucht und bereits an großen Steuerungen getestet worden ist. Es kann theoretisch aus allen IEC 61131-3 Sprachen in ein Zwischenmodell („intermediate model“ in Abbildung 5-4) transformiert werden. Dieses wird daraufhin mit weiteren Tools bearbeitet um somit das Modell schlanker für die formale Verifikation zu gestalten. Dieses Zwischenmodell kann nun einfacher in die Eingangssprachen verschiedener Model Checker überführt werden. Dieser Schritt wird damit begründet, dass sich bislang noch kein Model Checker als der geeignetste für den Bereich durchgesetzt hat.

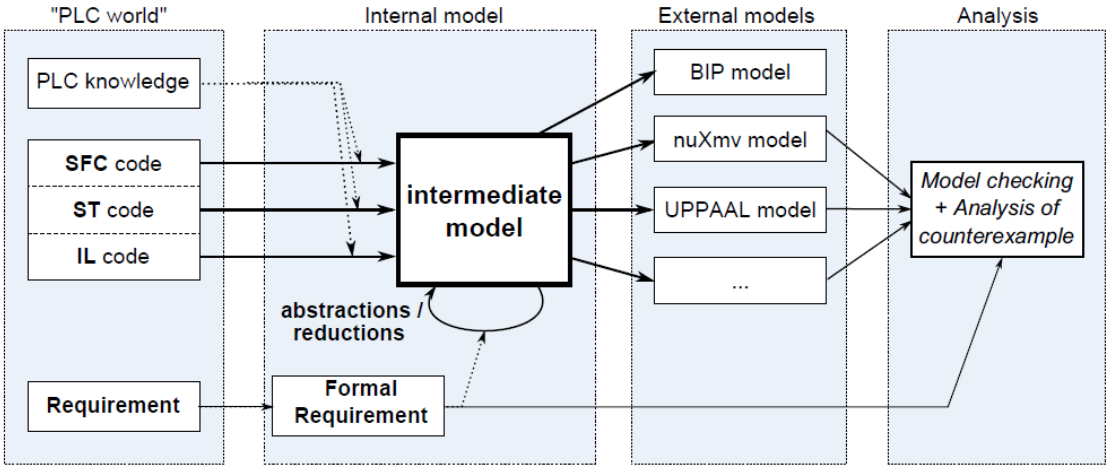


Abbildung 5-4 Vorgehen zur formalen Verifikation der SPSen im CERN

[15] http://www.plcopen.org/pages/tc6_xml/
[16] https://wikis.web.cern.ch/wikis/display/EN/PLCverif
[17] http://www.eclipse.org

ANF1	ANF2	ANF3	ANF4
➖	➕	➕	➕

Was spricht dafür	Was spricht dagegen
• Alle Sprachen der IEC 61131-3 werden theoretisch unterstützt. • Zukunftsorientiert durch keine Festlegung auf einen Model Checker • Integrierte Unterstützung zum Schreiben von formalen Spezifikationen • Getestet an Anlagen in Industriegröße	• Bislang nur für Siemens SPS • Noch nicht öffentlich verfügbar

5.3.2.3 BIALLAS et al. 2012

Ebenfalls eine Gesamtlösung bietet das Werkzeug Arcade.PLC[18] der Rheinisch-Westfälischen Technische Hochschule(RWTH) Aachen an. Analog zu PLCverif handelt es sich um ein Eclipse-basiertes Werkzeug. Anders als andere Model Checker führt Arcarde.PLC das Model Checking direkt an dem PLC Code durch, wodurch Gegenbeispiele direkt auf den Code zurückgeführt werden können. Die folgende Abbildung beschreibt die innere Zusammensetzung des Werkzeugs:

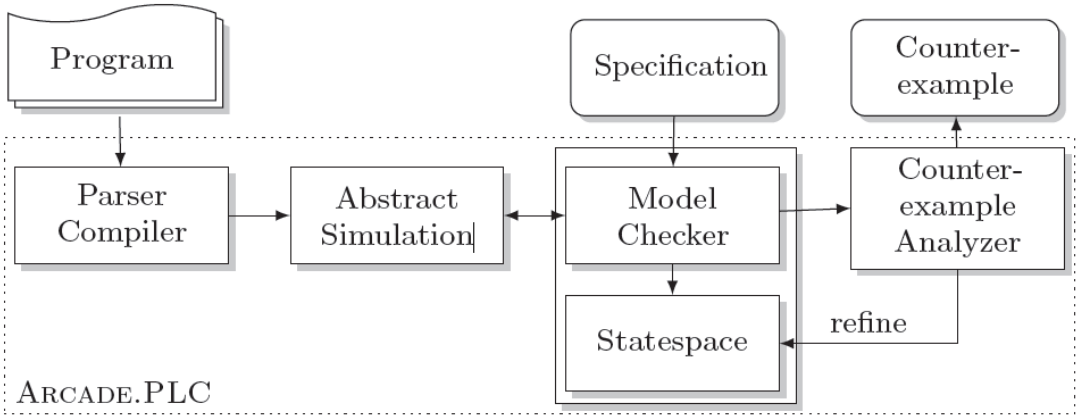


Abbildung 5-5 Arbeitsweise des Arcade.PLC Tools der RWTH Aachen

ANF1	ANF2	ANF3	ANF4
➕	➕	➕	➖

Was spricht dafür	Was spricht dagegen
• Öffentlich verfügbar • Gegenbeispiele in SPS Code • Getestet an Anlagen in Industriegröße	• Auf Siemens SPS spezialisiert • Keine Unterstützung für Ablaufsprache

[18] https://arcade.embedded.rwth-aachen.de

Spezifikationen können mittels CTL, Past Time LTL und Safety Automata formuliert werden. Es existiert eine Managementfunktion für die Spezifikationen, jedoch noch keine assistierte Möglichkeit diese zu formulieren.

Nach bestem Kenntnisstand handelt es sich um das einzige öffentlich verfügbare Werkzeug zum Model Checking von SPSen.

5.4 Fazit

Alle der hier vorgestellten Werkzeuge bieten gute Lösungen für ihr Ziel an. Unter Betrachtung der Anforderungen, vor Allem der Aspekte des Mehraufwands, sind die meisten Werkzeuge zum automatischen Testen eher ungeeignet. Sie verursachen entweder Mehraufwand in Form von doppelt zu bewältigender Arbeit oder viel Lernaufwand z.B. für das Lernen einer neuen Programmiersprache.

Im Bereich der formalen Verifikation liegt das Problem in der Transformation der in IEC 61131-3 geschriebenen Implementierungen. Diese ist wichtig um nicht nur künftige Software zu verifizieren, sondern auch vorhandene. Durch die assistierte Formulierung von Eigenschaften und der gleichzeitigen Dokumentierung dieser wird im Endeffekt Mehraufwand vermieden. Momentan ist somit ein erprobtes und verfügbares Werkzeug nur für Systeme SPS der Firma Siemens verfügbar.

6 Lösungskonzept

Dieses Kapitel stellt eine Lösung vor, welche die erkannten Anforderungen an ein System zur Steigerung der Software Qualität in der Automatisierungsindustrie erfüllen würde. Weiterhin werden die in Kapitel0 festgehaltenen Rahmenbedingungen eingehalten.

Ein Model Checking Werkzeug würde sich als ein sehr gutes und hilfreiches Mittel anbieten, um die Software Qualität in der SPS Welt zu steigern. Durch das einmalige Aufschreiben von formalen Spezifikationen, optimaler Weise assistiert, könnten verschiedenste Eigenschaften der Maschine formuliert werden. Diese Eigenschaften könnten auch an späteren Softwareversionen überprüft werden, da die automatische Transformation in die Eingangssprache eines Model Checkers dies ermöglichen würde. Da für eine formale Verifikationslösung der Schritt der Model Transformation fehlt, wird hierfür eine Lösung benötigt. Im Rahmen einer 6 monatigen Masterarbeit würde eine solche Transformationslösung für alle Sprachen der IEC 61131-3 aller Voraussicht nach nicht zufriedenstellend lösbar sein.

Um den gestellten Anforderungen gerecht zu werden, wurde sich aus der Testwerkzeug-Landschaft mechatronischer Systeme anderer Branchen bedient. Gemeinsam mit Hinweisen der genannten, aktuellen Literatur ist daraus das Konzept des *PLC Testcase Tool* (kurz PLCTT) entstanden.

6.1 Lernen aus anderen Branchen

Anforderung Nr. 4 (**ANF4**) verlangt, dass der Mehraufwand für den Entwickler gering sein soll. Keines der Test-Werkzeuge, die im vorangegangenen Kapitel vorgestellt wurden, erfüllt diese Forderung. In anderen Branchen wie der Anwendungsentwicklung gibt es Werkzeuge die dazu in der Lage sind.

6.1.1 Lernen aus der Automobilindustrie: BTC EmbeddedTester

Das Framework BTC Embedded Tester der Firma BTC Embedded Systems, welches oft in der Automobilindustrie verwendet wird, erlaubt neben Automatischen Tests und formaler Verifikation auch das Erstellen von Testfällen durch nachträgliches Evaluieren der Ergebnisse. Das Hilfswerkzeug TestVectorEditor[19] hilft bei dieser effizienten Testfallerstellung indem es dem Entwickler erlaubt das Steuergerät mit den Testdaten zu stimulieren und anschließend erst die entsprechenden Reaktionen zu bewerten. Dadurch verhilft es dem Entwickler quasi zur Erstellung eines reproduzierbaren Testfalls durch eine ohnehin notwendige Tätigkeit: Prüfen der programmierten Funktionen auf Richtigkeit.

6.1.2 Lernen aus der Anwendungssoftwareentwicklung: Capture/Replay Tools

Capture and Replay Werkzeuge (auch Testroboter genannt) werden zur Softwarequalitätssicherung bei der Entwicklung von für grafischen Benutzeranwendungen benutzt und tragen dort zur Testautomatisierung bei. Sie bieten die Möglichkeit, die Interaktion des Benutzers aufzunehmen, diese zu speichern (Capture) und zu einem späteren Zeitpunkt wieder abspielen zu lassen (Replay). Diese sogenannten Capture/Replay Tools nehmen dabei nicht primitiv den Bildschirmverlauf auf,

[19] https://www.btc-es.de/index.php

sondern registrieren die auftretenden Events, wie beispielsweise einen Mausklick auf einen beliebigen Button (SCHNEIDER 2007).

Ein Beispielwerkzeug ist QF-Test[20] für die Entwicklung in der Programmiersprache Java.

6.2 Dokumentieren

Neben effizient gewählten Testfällen sollten weitere Kriterien beim Testen beachtet werden: Zum Beispiel dass Testfälle komplett reproduzierbar sein sollten. Dem Tester sollte bewusst sein, dass zum Testen mehrere Artefakte gehören: Der Prüfling oder das "Test-System", die Testfallumgebung, die Spezifikation, die Testdaten und entstandenen Testresultate. Schneider sagt, dass streng genommen die gesamte Testumgebung mit Testfall mitprotokolliert und mitverwaltet werden soll, d.h. sogar der Compiler in der Version während des Tests. Pragmatisch gesehen sollte man sich jedoch auf die wichtigsten Daten zum Testfall beschränken (SCHNEIDER 2007). Dem Prüfling gilt der gesamte Fokus beim Testen. Die Tests welche durchgeführt werden, sollten sich genau auf den gleichen Prüfling beziehen, um später die Testsituation genau reproduzieren zu können. Daher sollte der Prüfling auch niemals während eines laufenden Tests geändert werden, nicht mal zur Fehlerkorrektur. (SCHNEIDER 2007). Ähnlich appelliert auch Gessler (GESSLER 2014, S. 195) die Testfälle professionell handzuhaben, zu dokumentieren und für spätere Testwiederholungen reproduzierbar zu sichern (bekannt als Regressionstest).

Somit steht fest, dass es für einen ordentlichen Test mit Testfällen nicht ausreicht, wenn ein Entwickler sich Eingangsbelegungen konstruiert und für sich selbst die Ergebnisse dokumentiert. Kurt Schneider hält folgende Mindestvoraussetzungen an Testdokumentation fest:

- Eine ID, an der sich der Testfall schnell und eindeutig identifizieren lässt.
- Die Testdaten, welche Eingaben und Parameter umfassen.
- Vorbereitende Schritte, um die Testumgebung in den benötigten Ausgangszustand zu versetzen. (Datenbank füllen, Sensorsignale simulieren).
- Eine Beschreibung, was zur Durchführung des Tests zu tun oder aufzurufen ist.
- Die Sollresultate.

Die Anforderung Nr.1 (ANF1) verlangt, eine Dokumentation über die Ausgänge der Testfälle und deren Beschreibung.

6.3 Unabhängigkeit von Hardware

Das modellbasierte Testen von SPSen, wie es in 5.2.4 (KORMANN et al. 2011) vorgestellt wird, erlaubt das Testen ohne dabei die Maschine zu benötigen. Dabei werden die Tests so modelliert, dass sie leicht Änderungen erlauben, um das Manipulieren von falschen Signalen zu ermöglichen. Es wird richtig erkannt, dass damit die Testfälle direkt als Dokumentation dienen können. Weiterhin ist die Zurückverfolgbarkeit des Grunds eines Fehlverhaltens eine nützliche Anforderungen an ein Testsystem die beschrieben wird.

[20] https://www.qfs.de

6.4 Grobes Konzept

Die Idee stammt aus einer Kombination von Capture and Replay Tool, dem BTC Embedded Tester, Modellbasierter Fehlereinpflanzung (SUSANNE ROESCH et al. 2014) und einem sogenannten deterministischen Replay Debugging Ansatz (PRAHOFER et al. 2011, S.650), bei dem Aufnahmen zu Offline-Debugging-Zwecken verwendet werden. Dadurch soll ungewöhnliches Verhalten durch Aufnahme der Abläufe und anschließende Präsentation aus verschiedenen Sichten aufzeigt werden.

Das Konzept wird nun anhand der folgenden Abbildung erläutert. Anhand von Markierungen, welche die Reihenfolge der Abarbeitung in den beiden Phasen „Aufnahme“ und „Wiedergabe + Evaluation“ zuweisen, wird die prinzipielle Abarbeitung in der richtigen Reihenfolge gezeigt wie es die Auflistung unterhalb der Abbildung ausformuliert:

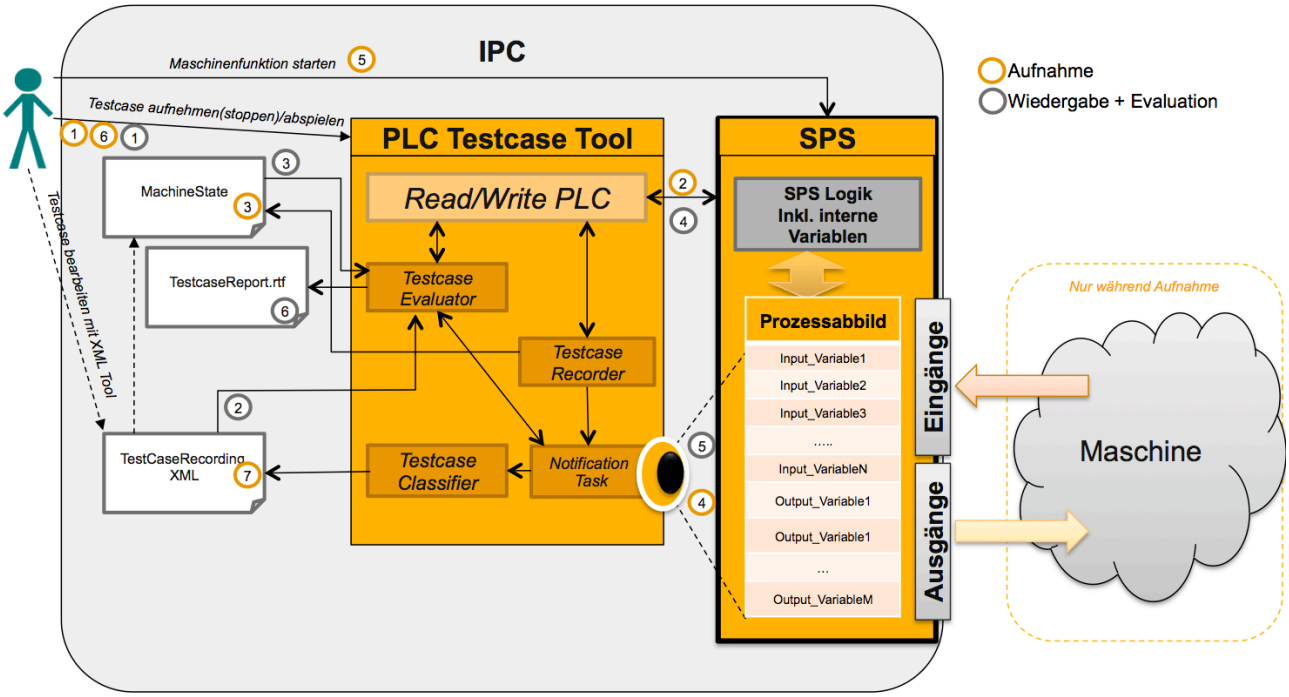


Abbildung 6-1 System-Übersicht über den groben Aufbau des PLC TestCase Tools

<u>Aufnahme</u>

1. Der Anwender ist mit einer laufenden SPS verbunden, die an einer Maschine angeschlossen ist und startet die Testcase Aufnahme.
2. Das PLCTT liest nun alle internen Merker- und Eingangsvariablen aus der SPS aus.
3. Der Maschinenzustand wird serialisiert[21] und in einer Extra-Datei gespeichert.
4. Nun gibt das PLCTT dem Benutzer die Rückmeldung, dass die Aufnahme läuft. Die *Notification Task* überwacht die Ein- und Ausgänge der SPS auf Wert-Änderungen. Jede Änderung wird registriert und mit einem Zeitstempel gespeichert.
5. Der Benutzer startet nun die zu testende Funktion.
6. Nach Beendigung der Funktion, stoppt der Benutzer den Testfall und gibt eine Beschreibung und einen Namen in das System ein.
7. Die aufgenommenen Änderungen werden nun zu einem Testfall zusammengefasst und in eine *TestCaseRecording*-Datei gespeichert, bei dem die Ausgangssignale die zu Evaluierenden sind und die Eingangssignale zu simulierende Signale sind.

[21] https://de.wikipedia.org/wiki/Serialisierung

Wiedergabe + Evaluation

1. Der Anwender ist mit einer laufenden SPS ohne angeschlossene Maschine verbunden und will einen Testfall starten.
2. Der Anwender wählt eine *TestCaseRecording*-Datei aus.
3. Das PLCTT liest aus der *TestCaseRecording*-Datei den Zustand der SPS vor Testfallstart aus (*MachineState*-Datei und schreibt diese in den Programmspeicher.
4. Die SPS wird gestoppt und der Zustand in Form von allen Eingangsvariablen und internen Merker Variablen in die SPS geschrieben. Die SPS befindet sich nun erneut im Ausgangszustand vor dem aufgenommenen Testfall.
5. Nun wird die SPS gestartet und es werden die Eingangssignale für die SPS nach der vorgegebenen Reihenfolge und den entsprechenden Zeitabständen in die SPS geschrieben. Gleichzeitig werden die nun von der SPS gesetzten Ausgangssignale evaluiert.
6. Nach Beendigung des Testfalls durch ein nicht erwartetes Verhalten, wie eine zu späte Reaktion der SPS oder eine falsche Reihenfolge, wird der Testfall beendet. Ein Report über den Testfall wird erzeugt, welcher die gesetzten Eingangssignale und die erhaltenen Reaktionen der SPS auflistet und bewertet anhand eines Abgleichs mit der *TestCaseRecording*-Datei.

Die Eigenschaften, welche das PLCTT hat und dadurch die Anforderungen erfüllt, werden in der Tabelle 6-1 ersichtlich.

ANF1	• Die Dokumentation enthält eine feste ID die sich aus Datum und Uhrzeit der Testdurchführung im Format YYYYMMDDHHMM zusammensetzt. • In der Datei wird beschrieben, was getestet wird. • Alle Wertevariablen der Ein –und Ausgänge, die erwartet werden, sind in der richtigen Reihenfolge mit Soll-Wert und Ist-Wert gelistet. • Ein Verweis auf die *MachineState*-Datei, welchen den Zustand der SPS wieder in den herrschenden während der Aufnahme versetzen kann, stellt die Ausgangsbedingungen wieder her.
ANF2	Da das simulierte Verhalten auf einen Testfall zugeschnitten ist, kann es problemlos auch mit einer virtuellen Maschine verwendet werden. Es sollte jedoch beachtet werden, dass die HIL Funktionalität der virtuellen Maschine während der Ausführung eines Testfalls auszuschalten ist, da diese sich sonst möglicherweise gegenseitig behindern würden.
ANF3	Da das Testverhalten die Eingangssignale in eine Maschine selber schreibt und somit eine Art HIL Verhalten zeigt, ist eine Ausführung von Tests ohne Maschine oder einer anderen HIL Umgebung möglich.
ANF4	Dadurch, dass Aufnahmen als Testfälle verwendet werden ist der Aufwand auf ein Minimum reduziert. Die aufgenommen Abläufe werden vom Ingenieur ohnehin zur Prüfung abgefahren.

Tabelle 6-1 Begründungen für die Erfüllung der Anforderung durch das PLCTT

Somit wären alle Anforderungen an eine Lösung zur Steigerung der Software Qualität einer SPS erfüllt.

Im Folgenden werden Einzelheiten des PLCTT erläutert und bestehende Probleme erklärt.

6.5 Einzelheiten des Konzepts

Wie bereits aus der vorherigen Erklärung hervor ging, gibt es im Konzept zwei verschiedene Anwendungsfälle: *Testfall aufnehmen* und *Testfall prüfen*. Beide erzeugen als Ergebnis Dokumente. Ein Szenario in dem ein solches Werkzeug zum Einsatz kommen könnte, zeigt die folgende Abbildung.

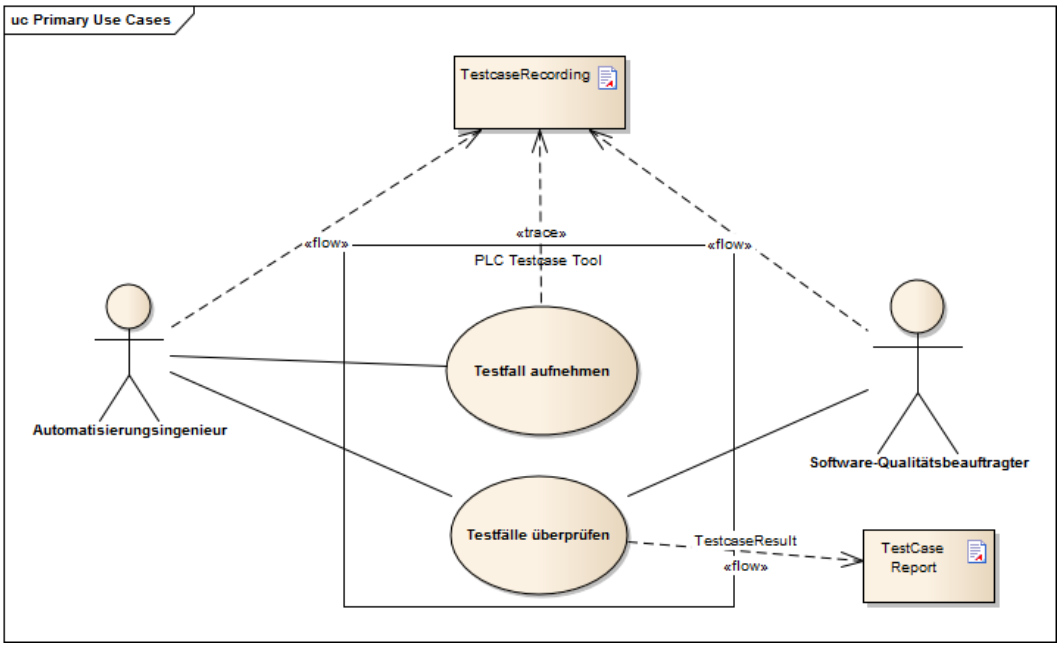


Abbildung 6-2 Use Case Diagramm des PLC Testcase Tools

Der Automatisierungsingenieur testet die Funktionen der Maschine und erzeugt dabei *TestCaseRecording*-Dateien. Diese sind aufgrund ihres Formats XML sehr leicht und komfortabel mit Hilfe von Werkzeugen wie z.B. XMLSpy[22] grafisch bearbeitbar. So können auch sogenannte „Was-wäre-wenn"-Testfälle unkompliziert aus den Dateien erstellt werden. Das Format für die Aufnahmen wird später genauer erläutert. Der Software-Qualitätsbeauftragte könnte in Form eines zweiten Ingenieurs auftreten. Dieser kontrolliert die durchgeführten Tests und ist dadurch in der Lage, die durchgeführten Tests zu verifizieren durch ein vier Augen Prinzip.

6.5.1 Testfall aufnehmen

Diese Funktion ist an den Ablauf in der Realität angelehnt: Auf Knopfdruck wird eine Funktion gestartet die vom PLCTT in allen Schritten erfasst wird. Aufgrund des schnellen Auslesens, welches die ADS Schnittstelle ermöglicht, wird jeder Signalwechsel der SPS erfasst. Der Ablauf wird dann in ein modifizierbares Format abgelegt. Anders als beim BTC EmbeddedTester wird hier von Beginn an angenommen, dass alle Ausgänge richtige Reaktion auf die stimulierten Eingänge zeigten. Es besteht allerdings auch die Möglichkeit die Ein- und Ausgänge entsprechend der Auffassung zu modifizieren, je nachdem ob ein Ausgang richtig oder falsch reagierte.

[22] http://www.altova.com/de/xmlspy.html

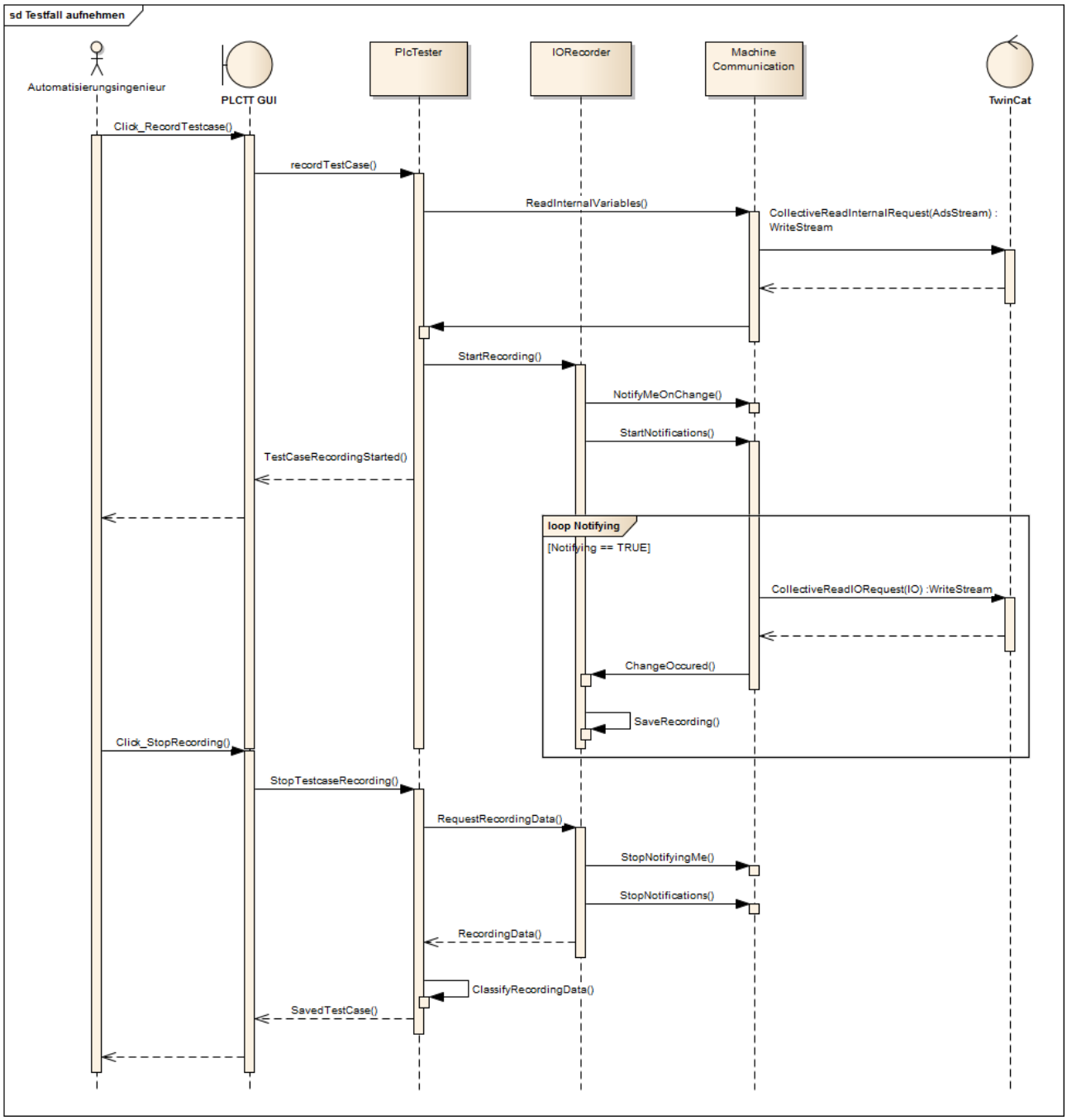


Abbildung 6-3 Sequenzdiagramm der Aufnahme Funktion des PLCTT

Zentraler Punkt in dem Ablauf ist die Notifikations-Funktion welche an dieser Stelle etwas genauer beschrieben werden soll. Einmal gestartet durch die *PLC Tester* Komponente läuft dieser als separater Task und fragt mittels SUM_WRITE Befehl, den die TwinCat ADS-Schnittstelle anbietet, alle Ein und Ausgänge ab. Eine solche Abfrage, die mehrere tausende Variablenwerte umfassen kann, ist relativ schnell beendet und hat in Versuchen kein Wertwechsel verpasst, was bedeutet, dass die benötigte Erfassungszeit stets unter der Zykluszeit der SPS lag.

Werden Wertwechsel detektiert, werden diese als Event an den *IORecorder* weitergeben, der sich vorher für Notifikationen angemeldet hat. Der *IORecorder* speichert diese ab und gibt sie dem *TestCaseClassifier* zur Entscheidung in welcher Reihenfolge und wie lange nacheinander diese Auftreten können bzw. ausgelöst werden.

Das TwinCat Interface selbst bietet auch einen Notifikations-Service an, welcher sich jedoch auf Grund der Limitierung von TwinCat auf 550 Variablen in der Praxis als wenig effizient gezeigt hat.

6.5.2 Testfall abspielen und Evaluieren

Um reproduzierbare, automatische Tests zu ermöglichen, muss die Maschine vorher in die passende Konfiguration und den passenden Zustand gebracht werden (RAMLER et al. 2014). Mit einem bekannten Anfangszustand kann dies erreicht werden. Von dort an kann die SPS deterministisch wiedergegeben werden ohne eine Verbindung zur originalen Umgebung zu benötigen (PRAHOFER et al. 2011, S. 641). Mit diesem Kerngedanken wird der Ablauf von Tests im PLCTT ermöglicht.

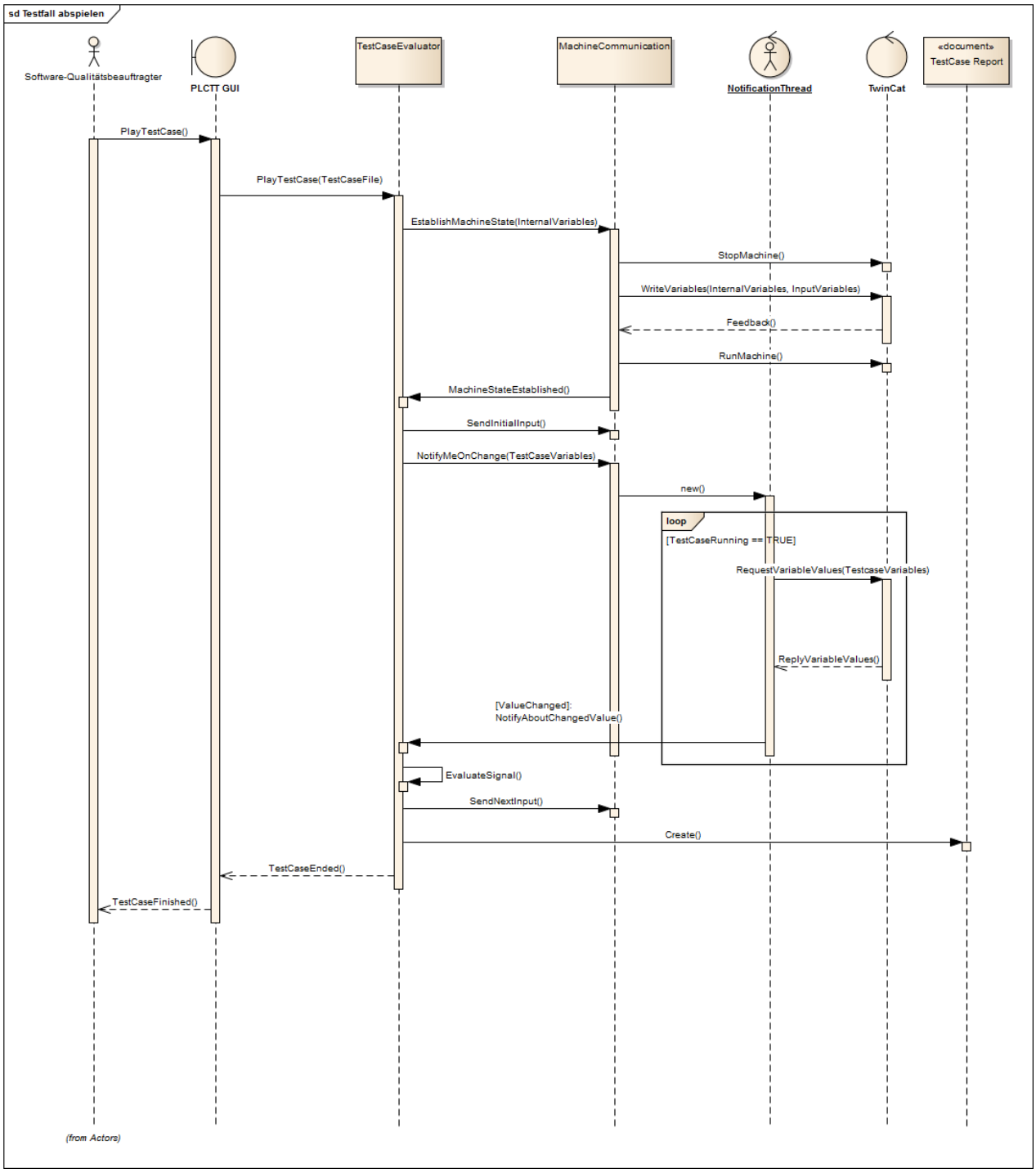


Abbildung 6-4 Sequenzdiagramm der Wiedergabe und Evaluationsfunktion des PLCTT

Die Evaluation und gleichzeitige Simulation im PLCTT wird mit der Hilfe von zwei parallel ablaufenden Threads[23] erledigt: Ein Task für die Eingangssignal-Simulation (Eingänge der SPS) und der andere für die Evaluation des Ausgangssignals. Die Idee ist, dass beide in einer Endlosschleife laufen und auf die globale Aktionsliste aus der TestCaseRecording-Datei des Tests

[23] https://en.wikipedia.org/wiki/Thread_(computing)

zugreifen und abfragen. Diese dient also als eine Art Synchronisation der beiden Threads. Beide können fünf verschiedene Zustände einnehmen: Prüfend, Wartend, Zustimmend, Stoppend, Abbrechend, dessen Bedeutung nachstehend erklärt wird:

Zustände des Thread für erwartete Ausgangssignale (*AssertedAction*):

- **Prüfend**
  - Prüfen ob das nächste Signal der Aktionsliste eine *AssertedAction* ist.
- **Wartend**
  - Nach **Prüfend**
  - Timer für die nächste *AssertedAction* laufen lassen.
- **Zustimmend**
  - Nach **Wartend**
  - Das erwartete Signal ist eingetreten. Signal auf globaler Aktionsliste abhacken.
- **Stoppend**
  - Entweder Timer abgelaufen nach **Wartend,**
  - Oder letztes Signal in der Aktionsliste wurde ausgeführt.
- **Abbrechend**
  - Der Andere Thread hat gestoppt.

Zustände des Thread für simulierte Eingangssignale (*EnvironmentStep*):

- **Prüfend**
  - Prüfen, ob das nächste Signal der Aktionsliste ein *EnvironmentStep* ist.
- **Wartend**
  - Nach **Prüfend**
  - Timer für die nächste *EnvironmentStep* laufen lassen.
- **Zustimmend**
  - Nach **Wartend**
  - Die Variable wird auf den Wert, der sich hinter dem *EnvironmentStep* steht gesetzt.
- **Stoppend**
  - Kann nicht eingenommen werden.
- **Abbrechend**
  - Der Andere Thread hat gestoppt.

Die Länge des Testcase hängt von der Länge der Aufnahme aus der *TestCaseRecording*-Datei ab. Erst nach Ablauf des Testfalls wird der TestCaseReport erzeugt wodurch der Testfall dokumentiert ist um möglicherweise Modifikationen wie Schneiden der Länge am Test zuvor zu ermöglichen

6.6 Die TestCaseRecording-Datei

Die Grundidee für das Aufnahme-Format stammt aus den modelbasierten Test-Ansatz mit UML Diagrammen (SUSANNE ROESCH et al. 2014).

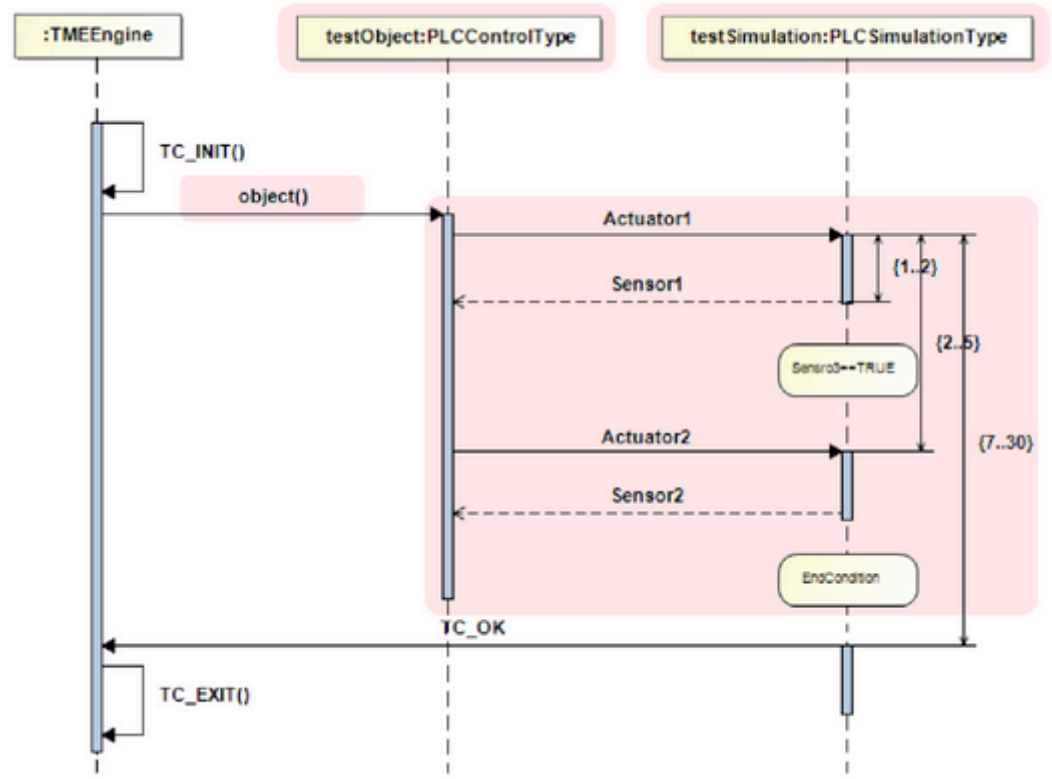


Abbildung 6-5 Ablauf eines Modelbasierten Testfalls an für eine SPS (SUSANNE ROESCH et al. 2014)

Ähnlich wird in der *TestCaseRecording*-Datei auch ein Anfangszustand beschrieben und danach Sensoren simuliert und Aktuatoren überwacht. Auch eine Zeitüberwachung ist vorhanden, wie bereits bei der Erläuterung der Wiedergabe-Funktion in 6.5.2. beschrieben wurde.

Das XML basierte Format der *TestCaseRecording*-Datei ist dafür vorgesehen, die Aufnahmen zu strukturieren und sie mit den entsprechenden Informationen und zu testenden Eigenschaften für die Wiedergabe und Evaluation zu versehen. Es wurde das XML Format gewählt, weil dieses einfach zu bearbeiten und umzusetzen ist und für die Implementierung eines Prototypen sehr gut eignet. Sie besteht aus folgenden Elementen die folgende Informationen enthalten:

TestCase	Übergeordnetes Element.
Name	Name des Testfalls.
Description	Beschreibung, was der Testfall zeigen soll.
MachineState	Dateiname der Maschinen-Zustandsdatei, welche sich im gleichen Ordner befinden muss.
ActionList	Übergeordnete Liste mit den durchzuführenden Schritten.
EnvironmentStep	Zu simulierender Schritt.
AssertedAction	Ein Event/Aktion die von der SPS als Reaktion erwartet wird.
VarName	Der Name der Variable in der SPS. Muss nicht unbedingt eine Ein-/Ausgangs-Variable sein. Auswertung klappt auch mit internen Variablen.
Value	Der erwartete Wert, bzw. der zu simulierende Wert der Variable.
WaitingTime	Die Wartezeit in Millisekunden bis diese Variable den Wert erreicht haben soll, bzw. wann dieser simuliert werden soll. Referenz ist hierbei immer der

	vorangegangene Wert. Es handelt sich um eine optionale Angabe. Bei *AssertedActions* wird bei Auslassen keine Zeit abgewartet, da lediglich die Reihenfolge relevant ist. Bei zu *EnvironmentSteps* wird der Wert sofort gesetzt.
ToleranceTime	Ein optionaler Wert, der bei einer Aufnahme hinzugefügt wird. Die Toleranzzeit ist eine Zeitangabe in Millisekunden. Weitere Erläuterung in Abschnitt 6.6.1.

Tabelle 6-2 Elemente des TestCaseRecording-Formats

Eine Beispiel *TestCaseRecording*-Datei für den Stitcher könnte wie folgt aussehen:

```
3.  <TestCase>
4.      <MachineState>MachineState_201606262308</MachineState>
5.      <Name>Stitcher Normalfunktion</Name>
6.      <Description>Der Stitcher fährt nach Drücken des Startknopfs
    den normalen Ablauf ab
7.      </Description>
8.      <ActionList>
9.
10.         <EnvironmentStep>
11.             <VarName>IN_Stitcher_Freigabe</VarName>
12.              <Value>true</Value>
13.              <WaitingTime>1775.42040000000009</WaitingTime>
14.         </EnvironmentStep>
15.
16.         <AssertedAction>
17.             <VarName>OUT_Hub_hoch</VarName>
18.             <Value>true</Value>
19.             <WaitingTime>14.649000000000001</WaitingTime>
20.             <ToleranceTime>31.617600000000001</ToleranceTime>
21.         </AssertedAction>
22.
23. <!—Überspringe Teil……..->.
24.
25.         <EnvironmentStep>
26.             <VarName>IN_Achsposition</VarName>
27.             <Value>0</Value>
28.             <WaitingTime>2.9</WaitingTime>
29.         </EnvironmentStep>
30.      </ActionList>
31. </TestCase>
```

6.6.1 Toleranzzeit

Es gibt Fälle mit schwer bestimmbaren Reihenfolgen für die Ausgangssignale einer SPS. Die genaue Bestimmung der Zeit und somit die Evaluation der richtigen Reihenfolge gestaltet sich schwierig durch die zyklische Abarbeitung der SPS und dem Überwachungsprozess, welcher nebenher die Wertänderungen überwacht. In einem Szenario, in dem zwei Ausgangsvariablen einen

Wertwechsel mit wenigem Millisekunden Abstand durchführen, kann es zu Problemen kommen. Welche Wertänderung am Ausgang als erstes erkannt wird, ist nicht wirklich vorhersehbar solange der Thread nicht synchron mit dem Zyklus der Maschine arbeitet, sprich die Werte immer wieder in der gleichen Reihenfolge während der Aktivität der SPS ausliest. Das führt zu folgender Situation (Abbildung 6-6 und Abbildung 6-7), wenn die Zykluszeit der Soft-SPS auf 10ms mit einer 80/20 Verteilung mit dem Betriebssystem zugunsten der SPS eingestellt ist, was ein sehr realistischer Wert ist.

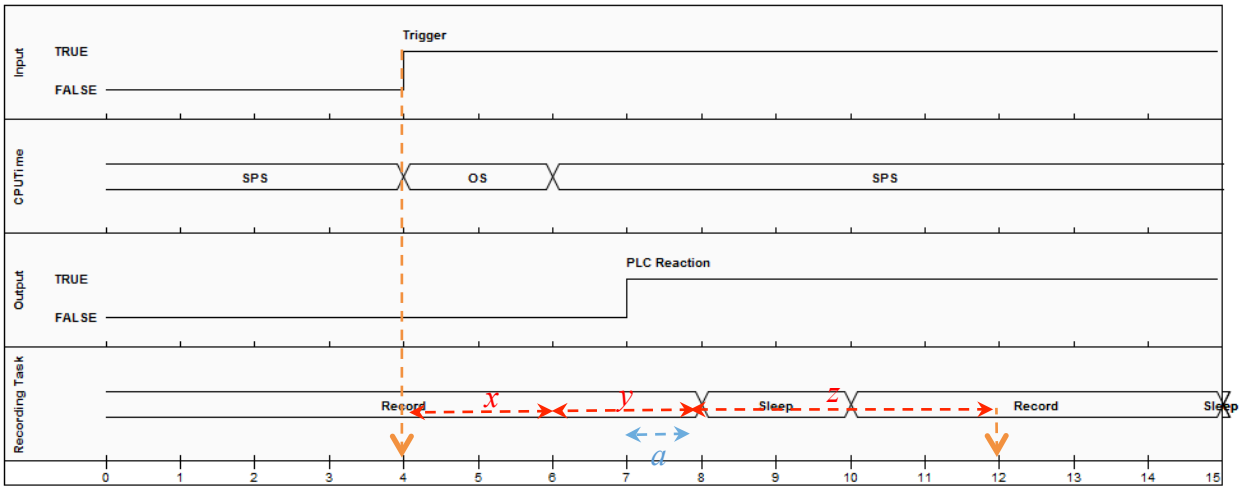


Abbildung 6-6 Timing Diagramm für das Worstcase Scenario für die Aufnahme von Testfällen

In dem Worstcase-Szenario wird ein Sensor-Inputwert bei 4ms ausgelöst.

- Das Betriebssystem hat soeben die Prozessorzeit bekommen und die SPS reagiert auf dieses Eingangssignal erst 3ms später.
- Zu dieser Zeit ist die Aufnahme-Task aktiv. Da sie jedoch in einer Schleife die Werte aller Variablen nacheinander liest kann es vorkommen, dass der Wert nicht erfasst wird.
- Für den nächsten Durchlauf bekommt die Aufnahme-Task keine Prozessorzeit zugewiesen. Die Aufnahme Task besitzt keine Echtzeiteigenschaften wie die Task der Soft-SPS, sondern bekommt vom Betriebssystem Ressourcen zugewiesen. Daher ist nicht absehbar zu welchem Zeitpunkt diese erneut startet.
- Nachdem die Aufnahme-Task wieder aktiv ist, braucht sie 2ms um in der Schleife zur entsprechenden Abfrage zu kommen. Nach $x+y+z$ = 6ms ist der Wert registriert.

Da diese Größen immer variieren, gilt es nun zu ermitteln, wie diese Größen dynamisch bestimmt werden können.

Die Dauer des letzten Auslese-Zyklus a und die Dauer vom letzten Gesamtdurchlauf aller Variablen bis zum Gesamtdurchlauf, bei dem die Änderung detektiert worden, ist z.

x bezeichnet die Zeit, die das Betriebssystem pro Zyklus zur Verfügung gestellt bekommt.

Somit bleibt y die einzige Variable, welche nicht bestimmbar ist. Durch empirische Versuche stellte sich die Faustformel $y=2*a$ bislang als praktikabel dar.

Es ergibt sich:

$$t_{\text{WorstCase}} = x+z+2*a$$

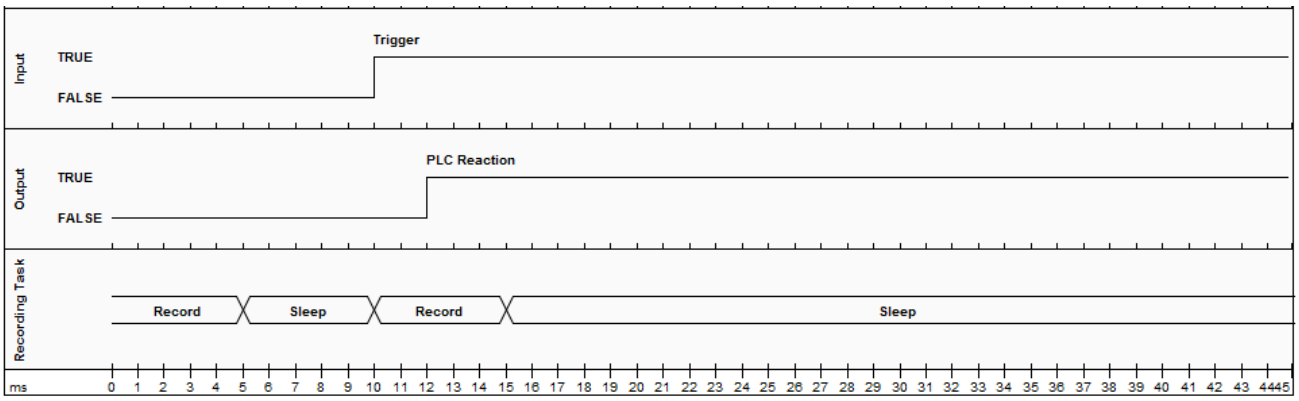


Abbildung 6-7 Bestcase Szenario für die Aufnahme von Testfällen

Im Bestcase Szenario kann theoretisch der Wert sofort innerhalb von dem Bruchteil einer Millisekunde erfasst werden. Der Sensorwert wird hier direkt von der aktiven SPS erfasst und dann direkt von der aktiven Aufnahme-Task.

Es ergibt sich also:

$$t_{BestCase} = 0s$$

Wenn das Beispiel weiter verfolgt wird, bedeutet das, dass ein gesetztes Ausgangssignal während einer Aufnahme immer eine Ungewissheit von 6ms aufweist. Wenn nun mehre Ausgänge zeitgleich oder in kurzen Zeitabständen nacheinander gesetzt werden, kann das PLCTT in der Evaluation nicht zu 100% bestimmen, welches Signal zuerst erwartet wird. Daher wurde das Konzept der Toleranz-Zeit nötig.

$$t_{Toleranz} = t_{WorstCase} - t_{BestCase} = x + z + 2 * a$$

Der Wert für x muss dem PLCTT über eine Konfigurationsdatei mittgeteilt werden.

6.6.2 Motion Contol

Das Erstellen von Testfällen mit Achsansteuerungen ist zwar möglich, aber weniger vorteilhaft mit der aktuellen XML Form der Modellierung. Die folgende Kurve über den Werteverlauf einer Achse macht das Problem deutlich:

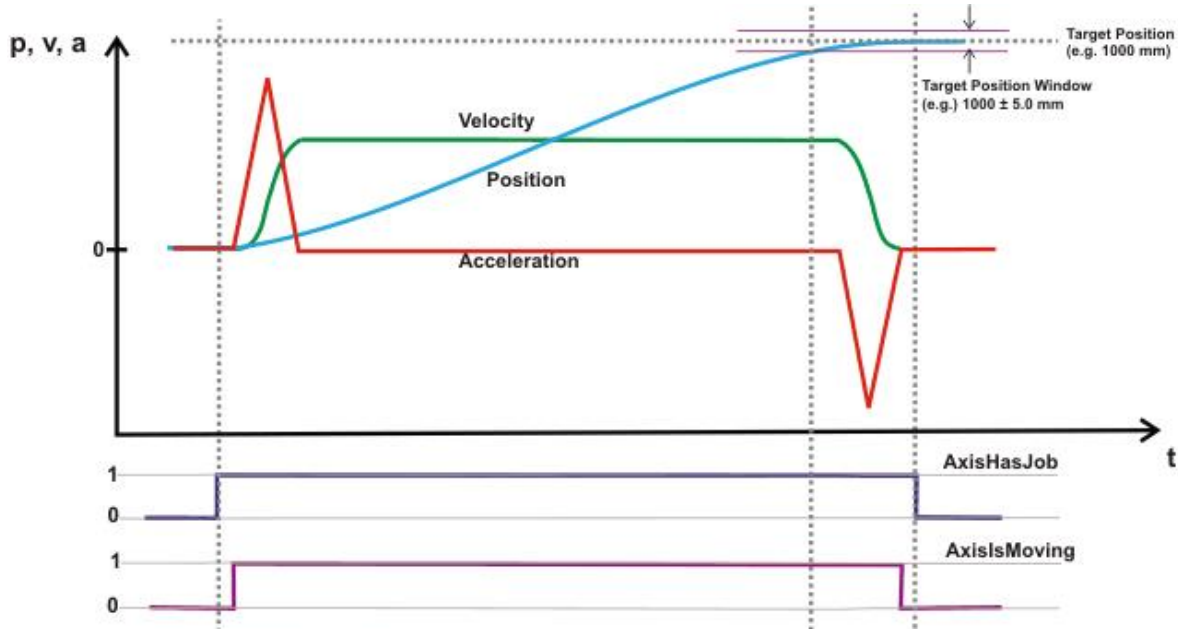


Abbildung 6-8 Diagramm über den Wertverlauf einer Achse

Angeschlossen an eine SPS wird der Wert für die aktuelle Position an den Steuerungsbaustein der Achse in der SPS gesendet. Dieser Wert, z.B. der Positionswert, ist sehr dynamisch und ändert sich häufig. Wenn zu einem Zeitpunkt die Eingänge der SPS überwacht werden, werden die kontinuierlich gesendeten Werte, im Endeffekt *EnvironmentAction*s, in die Testcase-Recording-Datei geschrieben. Die Übersichtlichkeit und Lesbarkeit der Datei wird somit gestört.

Ebenfalls muss beachtet werden, dass Achsbausteine ohne Hardware-Achsen ständige Wechsel der Modi durchführen. Das Deaktivieren der Hardware wird somit vor einer Wiedergabe notwendig.

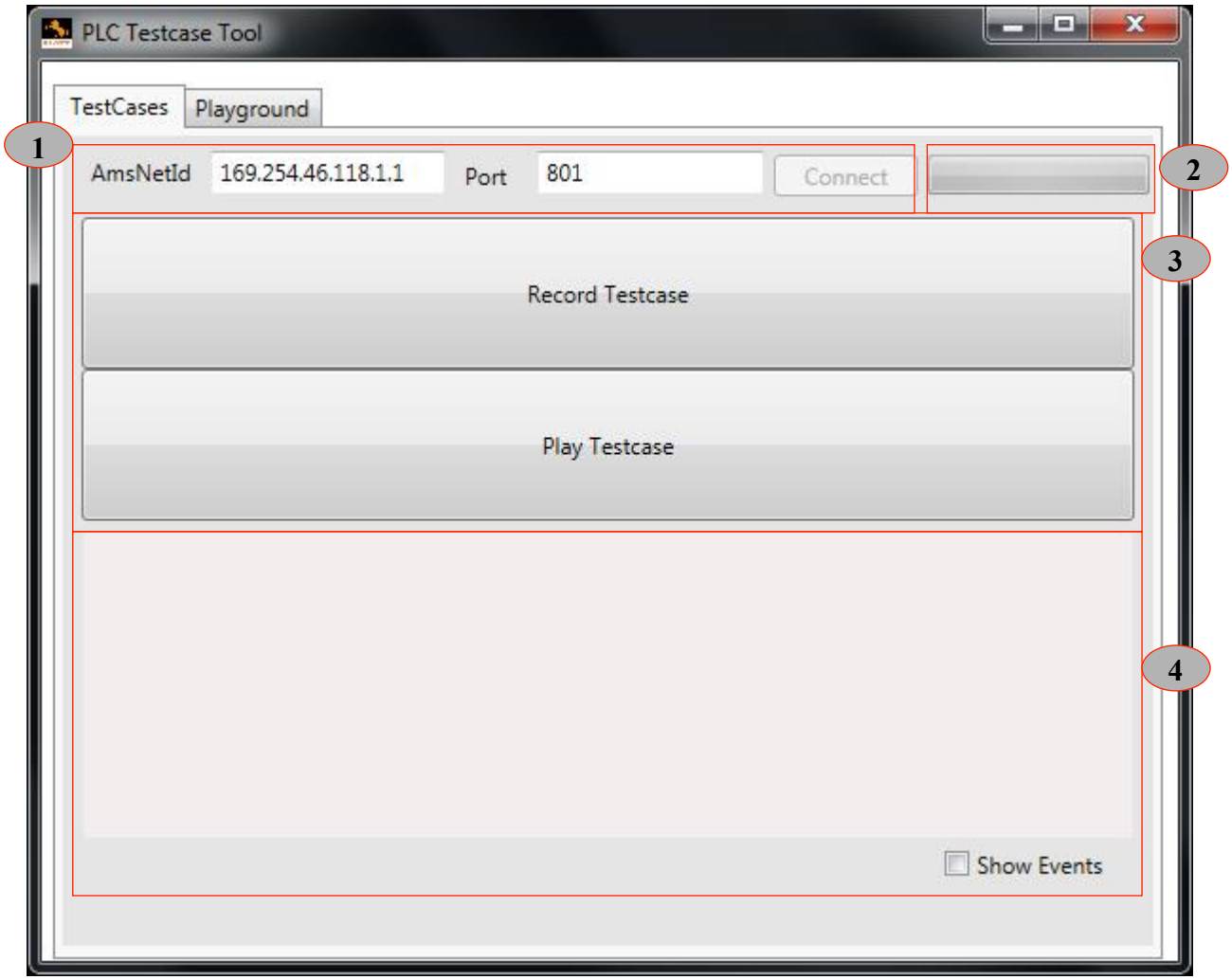
6.7 Testfall Report

Der Testfallreport ist eindeutig identifizierbar zu gestallten und verfügt daher über eine feste ID die sich aus Datum und Uhrzeit der Testdurchführung im Format *YYYYMMDDHHMM* zusammensetzt. Der Name und die Beschreibung des Testfalls aus der TestCaseRecording-Datei werden in den Report übernommen. Die genaue Abfolge mit Ist- und Sollzeit der AssertedActions sind aufgelistet, genau wie die simulierten Signale.

7 Anwendungsbeispiele

In diesem Kapitel soll anhand von Beispielen die Funktionalität des realisierten Prototyps gezeigt werden. Als Grundlage dafür soll das Stitcher-Beispiel dienen.

In der folgenden Abbildung ist die grafische Benutzeroberfläche des PLC Testcase Tools zu sehen. Die Konfigurationseinstellungen werden mittels Konfigurationsdatei festgelegt.



1. Die AmsNetId ist eine Art IP Adresse, die ein ADS Gerät eindeutig identifiziert. Die Soft-SPS TwinCat ist ein solches Gerät. Durch das Klicken auf „Connect“ wird sich auf die SPS gekoppelt.
2. Die Progressbar zeigt an, wie weit fortgeschritten Ladevorgänge sind.
3. Das Hauptfeld bietet die beiden Funktionen „Record Testcase“ (dt. Testfall aufnehmen) und „Play Testcase“ (dt. Testfall wiedergeben) an, welche jeweils die entsprechende Funktion starten.
4. Optionale Konsole für die Darstellung über erfasste Events.

Sobald die Verbindung aufgebaut ist, lassen sich die beiden Funktionen „Testfall aufnehmen“ und „Testfall abspielen“ starten.

7.1 Testfall aufnehmen

Abbildung 7-1 zeigt in einem Sequenzdiagramm am Stitcher-Beispiel den Ablauf während einer Testfallaufnahme. Anschließend werden die entsprechenden Bilder der Benutzeroberfläche gezeigt.

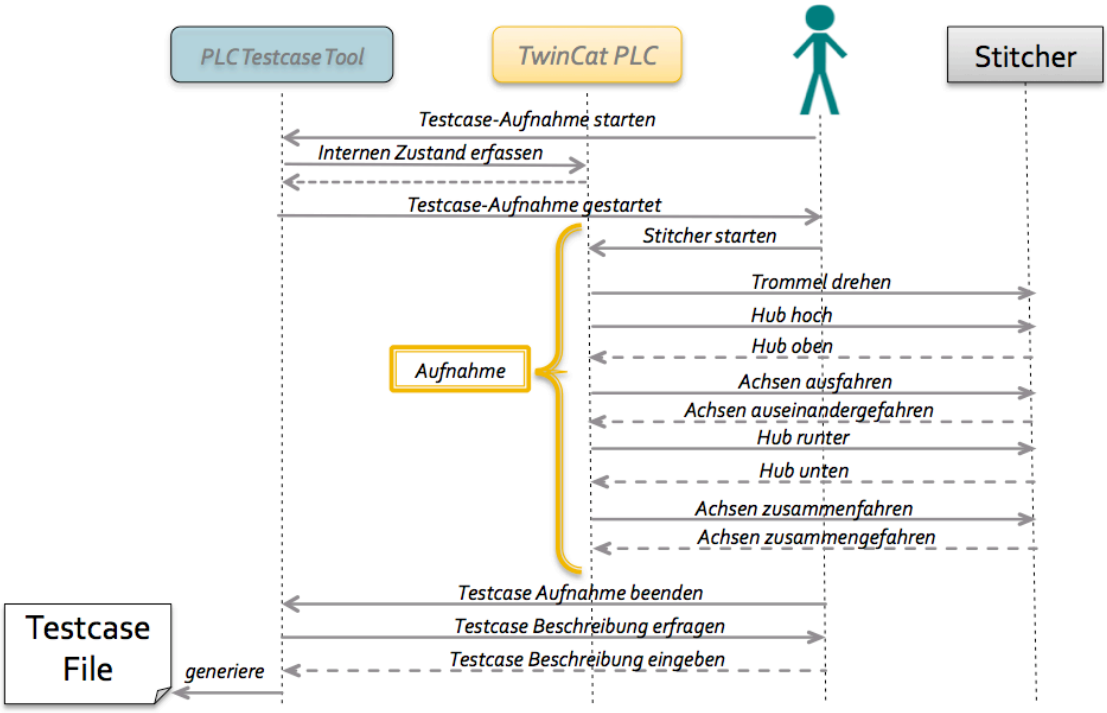


Abbildung 7-1 PLCTT: Testfall aufnehmen am Stitcher Beispiel

Beim Klicken auf „Testfall aufnehmen“ wird der Zustand der SPS gesichert und die Aufnahme gestartet. Nun lässt dich durch selbigen Button die Aufnahme beenden, wenn die Funktion für den Testfall abgelaufen ist:

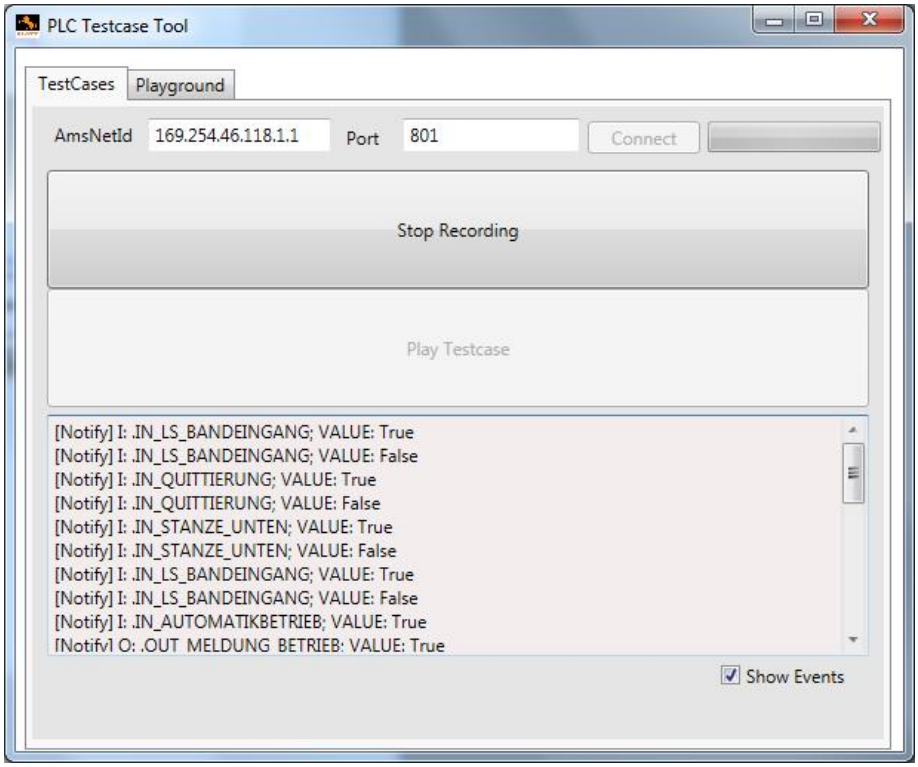


Abbildung 7-2 GUI des PLCTT während der Aufnahme Phase

Sobald die Aufnahme gestoppt wurde, wird der Tester nach einem Namen und einer Beschreibung des so eben ausgeführten Testfalls gefragt:

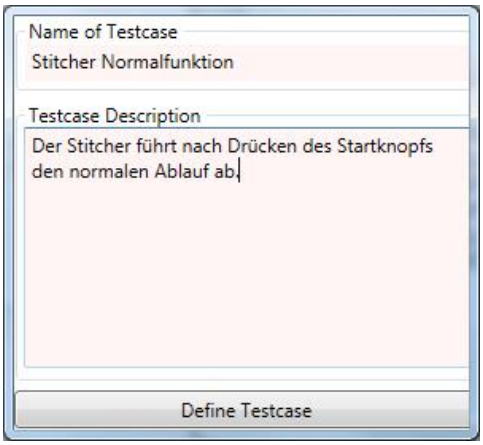


Abbildung 7-3 Abfrage nach Name und Beschreibung eines Testfalls

Die Struktur der dabei generierten TestCaseRecording-Datei wurde bereits in Abschnitt 6.6 am gleichen Beispiel gezeigt. In einem entsprechenden Werkzeug wie XMLSpy kann die Datei wie folgt betrachtet werden:

Abbildung 7-4 TestcaseRecording Datei in XmlSpy

Im Gegensatz zur XML Ansicht ist die Modifikation der Datei viel leichter gestaltbar. Ganze Blöcke können verschoben und kopiert werden. Mit Hilfe einer XML Schema-Datei lässt sich sogar die syntaktische Korrektheit nachweisen. Die Variablennamen, Werte und Wartezeiten (Tabelle 6-2) können problemlos editiert werden.

7.2 Testfall wird abgespielt

Auf Grundlage der Aufnahme kann nun der Testfall abgespielt werden. Das folgende Sequenzdiagramm beschreibt den entsprechenden Ablauf:

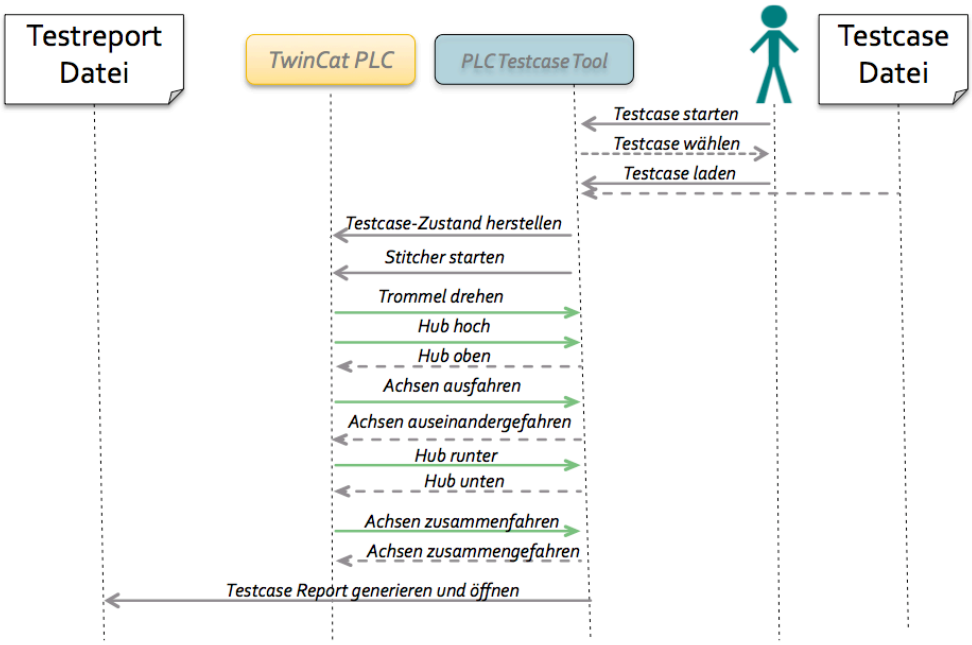


Abbildung 7-5 PLCTT: Erfolgreicher Testfall-Ablauf am Stitcher Beispiel

Als Resultat wird ein Testfall Report generiert, welcher folgendermaßen aussieht:

Testcase Result: Passed
Date: 27.06.2016 13:38

Name:Stitcher Normalfunktion
Description: Der Stitcher fährt nach Drücken des Startknopfs den normalen Ablauf ab

Member Name	Value	Trigger Time
IN_Stitcher_Freigabe	True	2114
OUT_Stitcher_hoch	True	5.862
IN_Stitcher_unten	False	1517
IN_Stitcher_oben	True	6.34
OUT_Stitcher_hoch	False	8.793
OUT_Stitcher_ausfahren	True	17.586
IN_Achsposition	1	3
IN_Achsposition	2	3
OUT_Hub_runter	True	0.977

Abbildung 7-6 Testfall Report nach bestandenen Test, Zur Demonstration gekürzt

Der resultierende Testfall Report hat die nachfolgend beschriebenen Eigenschaften:

7.3 Testfall wird mit Fehlverhalten bearbeitet

Zu Demonstrationszwecken wird nun der Ablauf in der TestCaseRecording-Datei modifiziert. Ein Fehlverhalten der SPS soll damit ausgelöst werden. Es wird nun suggeriert, dass nach dem simulierten IN_Stitcher_Freigabe-Signal die Variable fälschlicherweise *OUT_Stitcher_hoch* auf den Wert *false* gesetzt wird anstatt auf *true*.

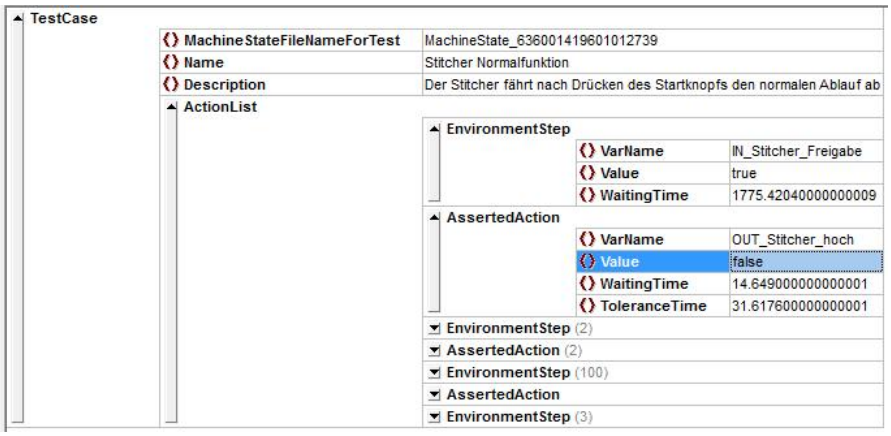


Abbildung 7-7 Bearbeiten des Stitcher Testfalls in der TestCaseRecording-Datei

Es werden zur Verbesserung der Übersicht nur die Ein- und Ausgänge aufgenommen aber die der Testauswertung könnte theoretisch mit allen Variablen im SPS-Programm erfolgen.

Ein nicht-bestandener Testfall Report sieht folgendermaßen aus:

Testcase Result: Failed

Date: 27.06.2016 14:02
Testcase: RecordedStitcherTc.XML

Name:Stitcher Normalfunktion
Description: Der Stitcher fährt nach Drücken des Startknopfs den normalen Ablauf ab

Member Name	Value	Trigger Time
IN_Stitcher_Freigabe	True	952.2
OUT_Stitcher_hoch	True	Not Arrived

Abbildung 7-8 Testfall Report eines nicht bestandenen Testfalls

8 Fazit und Ausblick

Die Automatisierungsindustrie sieht sich aufgrund des zunehmenden Automatisierungsbedarfs mit neuen Problemen konfrontiert, wie der geringen Zeit um Maschinenfunktionen zu testen und diese zu dokumentieren. Die straffe Zeitplanung in der Branche erlaubt es zumeist nicht aufwändige Testverfahren in den Entwicklungsprozess einzuführen, weshalb die wenigen momentan auf dem Markt verfügbaren Lösungen den Bedürfnissen der Industrie noch nicht entsprechen.

Automatische Funktionstests an speicherprogrammierbaren Steuerungen ohne Hardwarenotwendigkeit werden zunehmend wichtiger. Daher hat sich in der Automatisierungsindustrie die Idee durchgesetzt in Zukunft eine virtuelle 3D-Maschine zum Testen einzusetzen. Diese soll sich durch Konfiguration analog zu einer echten Maschine verhalten und kann mit der SPS gesteuert werden. Damit wäre das Problem der Hardwareunabhängigkeit weitestgehend gelöst, die Durchführung automatischer Tests zunächst noch nicht.

Eine sehr komfortable Lösung automatisierte Tests zu erstellen ohne dabei viel Mehraufwand für Entwickler darzustellen, stellen Capture/Replay Tools dar. Sie werden in der Entwicklung von Anwendungssoftware mit Benutzeroberfläche häufig verwendet um das Klicken auf Objekte auf Benutzeroberflächen zu simulieren. Das Aufnehmen von Aktionen, um daraus Testfälle zu erzeugen, ist eine in der Anwendungsentwicklung gängige Methode um schnell automatisierte Testfälle zu erzeugen. Die Grundlegende Idee für das erarbeitete Konzept ist davon abgeleitet. Die Behauptung ist, dass sich das Ein- und Ausgangsverhalten einer Maschine abstrahieren lässt auf das Klickverhalten eines Anwenders, welches an einer grafischen Benutzeranwendung arbeitet und die Reaktion des Anwenders wiederspiegelt. Durch das Kombinieren mit weiteren Ideen aus der Forschung rund um das Thema Qualitätssteigerung der Software von SPSen, wie das Replay Debugging (PRAHOFER et al. 2011) für das Abspielen ab einem bestimmten Zustand und die modelbasierte Fehlereinpflanzung (SUSANNE ROESCH et al. 2014) für den interessanten Anwendungsfall, ist das Konzept des *PLC TestCase Tools* entstanden.

Das XML-basierte *TestCaseRecording*-Format, welches die Testfälle beschreibt und aus den Aufzeichnungen der Abläufe generiert wird, kann durch XML Werkzeuge komfortabel bearbeitet werden und die Syntax mittels XML Schemas überprüft werden. Die optionale Angabe von Werten erlaubt es problemlos Testfälle bei Bedarf auch ohne vorheriges Aufnehmen zu formulieren. Das PLCTT wird entsprechend auf fehlende Werte reagieren: Ist z.B. kein Zeitwert für ein erwartetes Ausgangssignal der SPS angegeben, ist nur die Reihenfolge relevant.

Dieser neue und experimentelle Ansatz Speicherprogrammierbare Steuerungen zu Testen bedarf noch einer gründlicheren Prüfung. Erste Tests mit den Prototypen verliefen weitestgehend problemlos. Verbesserungsbedarf an verschieden Stellen des Konzepts sind jedoch zu erkennen.

Vor Allem beim Aufbau *TestCaseRecording*s. Der Overhead welches durch das XML Format entsteht, kann bei der Aufnahme komplexer Maschinen ziemlich stark gewichten. Es wäre hier eine schlankere Art des Formalismus interessant, wie es beispielsweise in der erwähnten *CPTest+* Sprache verwendet wird. Ein wichtiger Punkt wäre auch das Erkennen von Motion Control Bausteinen und ein komprimiertes Beschreiben von Verhalten, anstatt jeden Signalwechsel einzeln aufzunehmen. Theoretisch würde das ADS Interface das Erkennen dieser Bausteine leicht ermöglichen.

Weiterhin könnte anstelle der Toleranzzeitangaben eine Gruppenbildung für alternativ mögliche Signale erfolgen. Dabei müssten zwar ebenfalls alle Signale eintreffen, doch die Reihenfolge wäre egal. Dieser Schritt wird aktuell im PLCTT selbst durchgeführt. Allerdings wäre es auch als Konzept für eine Erweiterung der *TestCaseRecordings* geeignet.

Da es prinzipiell möglich ist *Time*-Variablen in der SPS zu manipulieren, könnte auch über eine Erweiterung nachgedacht werden, welche es erlaubt, Wartezeiten zu überspringen, um so die unter Umständen lange Zeit zwischen Signalen zu überspringen.

A. Anhang

A1. Datentypen in TwinCat

Table 1 Datentypen in TwinCat

Standard Datentyp	Benutzerdefinierte Datentypen
BOOL	ARRAY (Felder, Arrays)
BYTE	POINTER (Zeiger)
WORD	ENUM (Aufzählungstyp)
DWORD	STRUCT (Strukturen)
SINT	ALIAS (Referenzen)
USINT	Unterbereichstypen
INT	
UINT	
DINT	
UDINT	
LINT (Signed 64 Bit Integer, wird aktuell von TwinCAT nicht unterstützt)	
ULINT (Unsigned 64 Bit Integer, wird aktuell von TwinCAT nicht unterstützt)	
REAL	
LREAL	

Für eine übersichtliche Programmstrukturierung hat der Anwender ebenfalls die Möglichkeit, eigene Anwender-Datentypen mit dem Schlüsselwort TYPE zu definieren. Anwender-Datentypen können für verschiedene Zwecke verwendet werden auf die hier nicht näher eingegangen wird.

A2. Vergleich der Datentypen von TwinCat und .NET-Programmiersprachen

Table 2 Vergleich der Datentypen von TwinCat und .NET

System Manager	IEC61131-3	Entsprechender .NET Typ	C# Keyword	Visual Basic Keyword
BIT	BOOL	*System.Boolean*	*bool*	*Boolean*
BIT8	BOOL	*System.Boolean*	*bool*	*Boolean*
BITARR8	BYTE	*System.Byte*	*byte*	*Byte*
BITARR16	WORD	*System.UInt16*	*ushort*	-
BITARR32	DWORD	*System.UInt32*	*uint*	-
INT8	SINT	*System.SByte*	*sbyte*	-
INT16	INT	*System.Int16*	*short*	*Short*
INT32	DINT	*System.Int32*	*int*	*Integer*
INT64	LINT	*System.Int64*	*long*	*Long*
UINT8	USINT	*System.Byte*	*byte*	*Byte*
UINT16	UINT	*System.UInt16*	*ushort*	-
UINT32	UDINT	*System.UInt32*	*uint*	-
UINT64	ULINT	*System.UInt64*	*ulong*	-
FLOAT	REAL	*System.Single*	*float*	*Single*
DOUBLE	LREAL	*System.Double*	*double*	*Double*

A3. ADS Schnittstelle

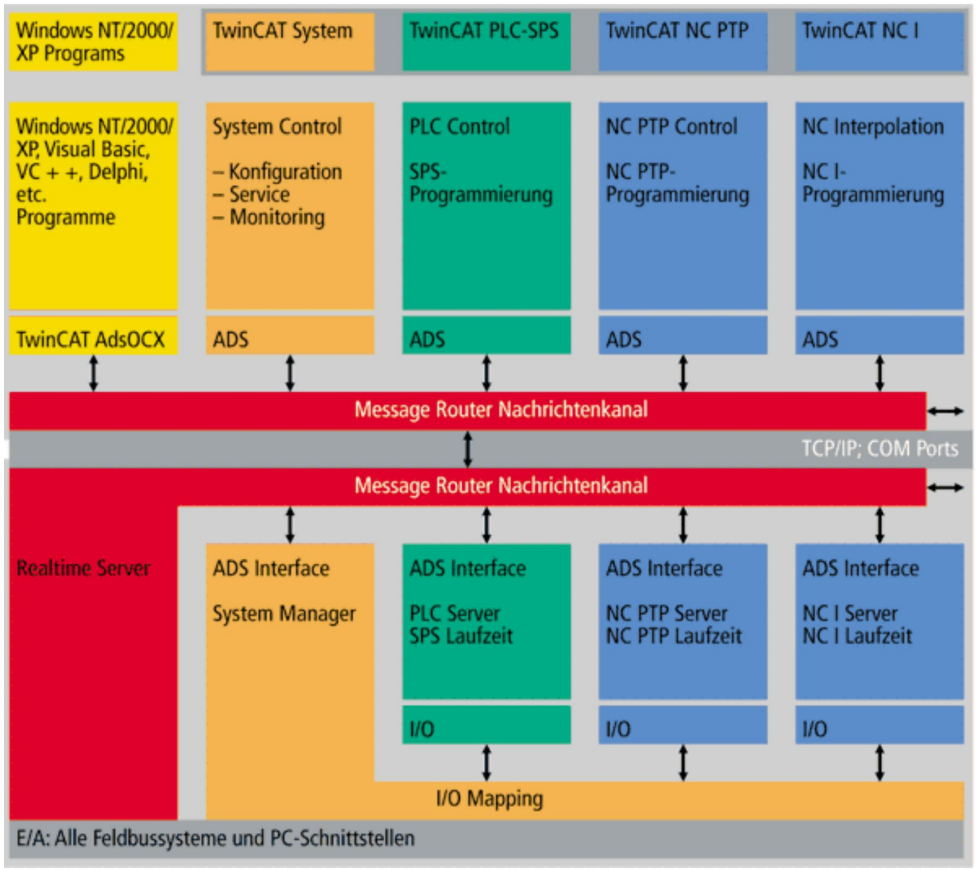


Figure 1 Kommunikation der TwinCat ADS Schnittstelle

B. TwinCat ADS Device Konzept (BECKHOFF AUTOMATION GMBH & CO. KG)

B1. ADS Reservierte Index Gruppen für das .NET-Interface

```
PlcRWMX = 16416,
PlcRWMB = 16416,
PlcRWRB = 16432,
PlcRWDB = 16448, Hex:4040
SymbolTable = 61440,
SymbolName = 61441,
SymbolValue = 61442,
SymbolHandleByName = 61443,
SymbolValueByName = 61444,
SymbolValueByHandle = 61445,
SymbolReleaseHandle = 61446,
SymbolInfoByName = 61447,
SymbolVersion = 61448,
SymbolInfoByNameEx = 61449,
SymbolDownload = 61450,
SymbolUpload = 61451,
SymbolUploadInfo = 61452,
SymbolNote = 61456,
IOImageRWIB = 61472,
IOImageRWIX = 61473,
IOImageRWOB = 61488,
IOImageRWOX = 61489,
IOImageClearI = 61504,
IOImageClearO = 61520,
DeviceData = 61696,
```

B2. "Index-Group/Offset" Spezifikation für allgemeine SPS-ADS-Dienste

In dieser Gruppe befinden sich auch Dienste für den Zugriff auf den SPS-Prozessdatenbereich der Merker.

Index Group	Index Offset	Zugriff	Datentyp	Phys. Einheit	Def.-Bereich	Beschreibung	Anmerkung
0x00004020	0x00000000-0x0000FFFF	R/W	UINT8[n]			**READ_M - WRITE_M** SPS-Memory-Bereich(%M-Feld). Offset ist Byteoffset	
0x00004021	0x00000000-0xFFFFFFFF	R/W	UINT8			**READ_MX - WRITE_MX** SPS-Memory-Bereich(%MX-Feld). Das Low-Word des Index-Offsets ist der Byteoffset. Der Index-Offset enthält die Bitadresse, die sich aus Bytenummer*8+Bitnummer errechnet.	
0x00004025	0x00000000	R	ULONG			**PLCADS_IGR_RMSIZE** Bytelänge des SPS-Prozessabbildes des	

						Memory-Bereiches	
0x00004030	0x00000000-0xFFFFFFFF	R/W	UINT8			**PLCADS_IGR_RWRB** Retain-Datenbereich. Offset ist Byteoffset	
0x00004035	0x00000000	R	ULONG			**PLCADS_IGR_RRSIZE** Bytelänge des Retain-Bereiches	
0x00004040	0x00000000-0xFFFFFFFF	R/W	UINT8			**PLCADS_IGR_RWDB** Daten-Bereich. Offset ist Byteoffset	
0x00004045	0x00000000	R	ULONG			**PLCADS_IGR_RDSIZE** Bytelänge des Daten-Bereiches	

B3. "Index-Group/Offset" Spezifikation für TwinCAT ADS-Systemdienste

Dieser Abschnitt umfasst diejenigen ADS-Dienste, die bei jedem TwinCAT-ADS-Gerät identische Bedeutung und Wirkung haben. In dieser Gruppe befinden sich auch Dienste für den Zugriff auf die SPS-Prozessdaten der Ein- und Ausgänge.

Index Group	Index Offset	Zugriff	Datentyp	Beschreibung	Anmerkung
0x0000F003	0x00000000	R&W	W: UINT8[n] R: UINT32	**GET_SYMHANDLE_BYNAME** Dem in den Write-Daten enthaltene Namen wird ein Handle (Kennwert) zugewiesen und dem Aufrufer als Ergebnis in den Read-Daten zurückgereicht.	
0x0000F004	0x00000000			Reserviert	
0x0000F005	0x00000000-0xFFFFFFFF =symHandle	R/W	UINT8[n]	**READ_/WRITE_SYMVAL_BYHANDLE** Den Wert, der durch 'symHdl' identifizierten Variable, lesen oder der Variablen einen Wert zuweisen. Der ‚symHdl' muss vorher durch den GET_SYMHANDLE_BYNAME-Dienst ermittelt worden sein.	
0x0000F006	0x00000000	W	UINT32	**RELEASE_SYMHANDLE** Die in den Write-Daten enthaltene Kennzahl (Handle) für eine abzufragende benannte SPS-Variable wird freigegeben.	
0x0000F020	0x00000000-0xFFFFFFFF	R/W	UINT8[n]	**READ_I - WRITE_I** SPS-Prozessabbild der physikalischen Eingänge(%I-Feld). Offset ist Byteoffset.	
0x0000F021	0x00000000-0xFFFFFFFF	R/W	UINT8	**READ_IX - WRITE_IX** SPS-Prozessabbild der physikalischen Eingänge(%IX-Feld). Der Index-Offset enthält die Bitadresse, die sich aus Bytenummer*8+Bitnummer errechnet.	

0x0000F025	0x00000000	R	ULONG	**ADSIGRP_IOIMAGE_RISIZE** Bytelänge des SPS-Prozessabbildes der physikalischen Eingänge.	
0x0000F030	0x00000000-0xFFFFFFFF	R/W	UINT8[n]	**READ_Q - WRITE_Q** SPS-Prozessabbild der physikalischen Ausgänge(%Q-Feld). Offset ist Byteoffset.	
0x0000F031	0x00000000-0xFFFFFFFF	R/W	UINT8	**READ_QX - WRITE_QX** SPS-Prozessabbild der physikalischen Ausgänge(%QX-Feld). Der Index-Offset enthält die Bitadresse, die sich aus Bytenummer*8+Bitnummer errechnet.	
0x0000F035	0x00000000	R	ULONG	**ADSIGRP_IOIMAGE_ROSIZE** Bytelänge des SPS-Prozessabbildes der physikalischen Ausgänge.	
0x0000F080	0x00000000-0xFFFFFFFF = n (Anzahl der internen (Sub-)Befehle)	R&W	W: (n * ULONG[3]) := **IG1, IO1, Len1, IG2, IO2, Len2, ..., IG(n), IO(n), Len(n)** R: (n * ULONG) + UINT8[Len1] + UINT8[Len2] + ..., + UINT8[Len(n)] := **Result1, Result2, ..., Result(n), Data1, Data2,..., Data(n)**	**ADSIGRP_SUMUP_READ** Die Write-Daten enthalten eine Liste von mehreren, separaten AdsReadReq(IG, IO, Len, Data) quasi als "Sammel-Lesebefehl". Dem Aufrufer wird in den Read-Daten das Ergebnis der Sammelanfrage zurückgereicht. Dabei werden zuerst alle Rückgabewerte aufgelistet, anschließend folgen die angefragten Daten.	SPS / IO ab TwinCAT v2.10 Build >= 1324
0x0000F081	0x00000000 - 0xFFFFFFFF = n (Anzahl der internen (Sub-)Befehle)	R&W	W: (n * ULONG[3]) + UINT8[Len1] + UINT8[Len2] + ..., + UINT8[Len(n)] := **IG1, IO1, Len1,**	**ADSIGRP_SUMUP_WRITE** Die Write-Daten enthalten eine Liste von mehreren, separaten AdsWriteReq(IG, IO, Len, Data) quasi als "Sammel-Schreibbefehl". Dem Aufrufer wird in den Read-Daten das Ergebnis der Sammelanfrage (die Rückgabewerte) zurückgereicht.	SPS / IO ab TwinCAT v2.11 Build >= 1550

			IG2, IO2, Len2, ..., IG(n), IO(n), Len(n), Data1, Data2, ..., Data(n) R: ULONG[n] := **Result1, Result2, ..., Result(n)**		
0x0000F082	0x00000000 - 0xFFFFFFFF = n (Anzahl der internen (Sub-)Befehle)	R&W	W: (n * ULONG[4]) + UINT8[WriteLen1] + UINT8[WriteLen2] + ..., + UINT8[WriteLen(n)] := **IG1, IO1, ReadLen1, WriteLen1, IG2, IO2, ReadLen2, WriteLen2, ..., IG(n), IO(n), ReadLen(n), WriteLen(n), WriteData1, WriteData2, ..., WriteData(n)** R: (n * ULONG[2]) + UINT8[ReturnLen1], + UINT8[ReturnLen2] + ..., + UINT8[ReturnLen(n)] := **Result1, ReturnLen1, Result2, ReturnLen2, ...,**	**ADSIGRP_SUMUP_READWRITE** Die Write-Daten enthalten eine Liste von mehreren, separaten AdsReadWriteReq(IG, IO, readLen, writeLen, writeData) quasi als "Sammel-Schreib/Lesebefehl". Dem Aufrufer wird in den Read-Daten das Ergebnis der Sammelanfrage zurückgereicht. Dabei werden zuerst alle Rückgabewerte und Return-Längen aufgelistet, anschließend folgen die angefragten Daten.	SPS / IO ab TwinCAT v2.11 Build >= 1550

			Result(n), ReturnLen(n), ReadData1, ReadData2, ..., ReadData(n)		

B4. Standardfunktionen nach IEC 61131-3

Table 3 Standardfunktionen nach IEC61131-3

Boole'sche Funktionen	
AND	UND-Verbindung
NOT	Negation (NICHT)
OR	ODER-Verbindung
XOR	EXOR-Verbindung

Arithmetische Funktionen	
ADD	Addierer
DIV	Divisor
EXPT	Potenzierung
MOD	Modulo-Division
MUL	Multiplizierer
SUB	Subtrahierer

Funktionen zur Typumwandlung
BYTE_TO_WORD
BYTE_TO_INT
BYTE_TO_REAL
UINT_TO_INT
INT_TO_UINT
DINT_TO_INT
INT_TO_DINT
INT_TO_BYTE
INT_TO_WORD
INT_TO_REAL
REAL_TO_BYTE
REAL_TO_WORD
REAL_TO_INT
TIME_TO_DINT
DINT_TO_TIME

Bitfolge-Funktionen	
ROL	Links rotieren
ROR	Rechts rotieren
SHL	Links schieben
SHR	Rechts schieben

Funktionen für Vergleich	
EQ	gleich (equal)
NE	ungleich (unequal)
LT	kleiner als (less than)
LE	kleiner gleich (less or equal)
GT	größer als (greater than)
GE	größer gleich (greater or eq.)

Numerische Funktionen	
ABS	Absolutwert
ACOS	Arccos
ASIN	Arcsin
ATAN	Arctan
COS	Cosinus
EXP	Exponent
LN	Natürlicher Logarithmus
LOG	Logarithmus zur Basis 10
SIN	Sinus
SQRT	Quadratwurzel
TAN	Tangens

Funktionen für Auswahl	
LIMIT	Begrenzung
MAX	Maximum
MIN	Minimum
SEL	binäre Auswahl

B5. Studie über die Verwendung von Simulation in der Automatisierungsindustrie

Table 4 Studie über die Verwendung von Simulation in der Automatisierungsindustrie Aus (KORMANN et al. 2012, S. 1616).

	Aspect		Machine/Plant Automation			
Current Situation	Used Programming Languages		IEC 61131-3 (mixture)		C/C++	
	Complexity of Control Software		mostly unable to pinpoint		no metrics for graphical languages	
	Development Process Model		no entry			
	Allocated Time for Testing		no explicit time for testing	considered to be part of software construction		usage of libraries
	Tool support	Modeling	Matlab / Simulink		COMOS	
		Version Control System	SVN		ENI	
		Bug Tracking	no entry			
		Configuration Management	no entry			
		Testing	None	Inhouse Software		
		Deployment	no entry			
		Requirements	Microsoft Office			
	Type of Testing		mainly function tests (blackbox)			
	Documented evidence of conformity		Internal: Checklist, Code Review	Video Taped Tests	Generally Performance Tests	Emergency Stop Tests
	Point in time for testing		Manual function test in testing environment (blackbox) during development		Manual interaction with functions by manipulating values within visualization (dry run)	
	Test Coverage		No need to be done, since not demanded			
	Testing resources		No test team available	Diverse PLCs (different vendors) are available for testing		
	Representation of test cases		description of operational sequence		State Machines	Textual Description
	Additional work for simulation model		Project point of view: not accepted		Only accepted, if benefit elsewhere	
	Information in simulation model		user interaction		colission information (robots)	
			state	boundary values	no existence of explicit simulation models	
	Frequent faults in control software		Missing reaction in undefined machine states (insufficient process description)			wrong I/O addresses
			permanent reimplementation of already implemented functionality			no standardization in software
			wrong interlocking	negative logic	missing notification on HMI	
	Derivation of test cases		requirements	specification	Function description of software / hardware modules	
	QA processes for software testing		No standardized process		Internal Code Reviews	
	Typical test scenarios		Simulation of machine faults	maloperation	Emergency Stop	Depends on project
	Are specified test cases executable		partially	Textual description for commissioning	No defined test cases after software construction	
Testing Needs	Granularity of Testing		Software element (FB) -> module -> machine -> plant		Software element during development	machines and modules
	Importance of Test scenarios		very important (plants become more complex)		depends on project and effort	
	Importance of testing fault scenarios		very important (clarification of occuring faults)		important (known and likely faults)	
	Importance of testing demanded functionality		very important		important	
	Which fault situations should to be tested		Known and occurring fault situations		known faults of serious consequences	
	Support for fault diagnosis of failed test cases		state of software and trace		Complete image of I/O	
	Importance of testing on boundary values		very important (situations, which endanger machine/operator)			maximum throughput

B6. Vergleich der Testverfahren der Automobilindustrie mit der Automatisierungsindustrie

Der Vergleich der beiden Industrien, welcher auf Basis einer umfänglichen Studie beruht, zeigt, dass die momentane Situation im Bereich Testen von Software in der Automatisierungsindustrie noch weit von dem entfernt ist, was in der Automobil/Luftfahrtindustrie bereits gut etabliert ist. (KORMANN et al. 2012, S. 1617).

Table 5 Vergleich beim Testen (KORMANN et al. 2012, 1617)

Machine / Plant Automation	Aerospace / Automotive
No allocated time for testing	High testing effort (40-80 %)
Manual functional tests (no regression testing)	High degree of automation (regression testing)
Missing resources for independent test specification	Separate testing departments
Currently no (statutory) documentation necessary	Documented evidence of testing conformity
Main focus: functional tests, fault situations, boundary values	Functional and nonfunctional tests
Almost no executable test cases	Requirements coupled testing (including tool support)
Lack of tool support	Tool support for blackbox and whitebox testing

Vergleich verschiedener DSLs für das Testen von SPSen

Table 6 Vergleich ausgewählter DSLs zum Testen von SPSen

Characteristic of the analyzed approach	Hametner *et al.* [26]	Krapfenbauer *et al.* [29]	Jamro and Trybus [35]	Current proposed CPTest+ approach
Test language	Keyword-driven approach	Python dynamically typed language	Preliminary version of CPTest+	Comprehensive version of CPTest+
Supported instructions	Keywords from the IEC 61131-3 norm[5]: `startConn`, `force`, `set`, `sleep`, `stopConn`, `get value`, `script`, `check`	A set of methods, such as `setData`, `setEvent`, `getEvent`, `doAssert`	`SET`, `RESET`, `ASSIGN`, `WAIT`, `LOG`, `ASSERT`	`SET`, `RESET`, `ASSIGN`, `WAIT`, `LOG`, `ASSERT`, `PAR`, `BEFORE`, `AFTER`, `INCLUDE`, `ANALOG`, `MOCK`
Supported standard	IEC 61131-3 and IEC 61499	IEC 61499	IEC 61131-3	IEC 61131-3
Abstraction level	High	Medium	High	High
Amount of code	Small	Medium	Small	Small
Require programming skills	Yes	No	Yes	Yes
Basic functional testing	Yes	Yes	Yes	Yes
Parameterized extension	No information	No information	No	Yes
Test fixtures	Other meaning	Yes	No	Yes
Test inclusions	No information	No information	No	Yes
Extension for analog signals	No information	No information	No	Yes
Mock objects	No detailed inform.	No detailed inform.	No	Yes
Complex test suites	No information	No information	No	Yes
IEC 61131-3-based unit tests	No	No	No	Yes
Support for performance tests	No information	No information	Partial	Yes [36], [37]

B7. Übersicht aktueller Recherche bezüglich automatischer Transformation von SPS Code für Model Checker

Table 7 Übersicht von formalen Verfikationslösungen für SPS Programme in ST (OVATMAN et al. 2014, S. 12)

	App. area	Syst. size	Performance	Auto. level	Formal spec.
[59]	Nuclear plant	16 blocks 7 vars	N/A	Semiauto.	N/A
[20]	Hydrogen gen. Unit	19 blocks 12 vars	N/A	Automatic	N/A
[115]	Nuclear plant	1,500 blocks 1,000 vars	N/A	Automatic	Manual
[89]	Railway interlock	100 vars	Minutes	Automatic	Manual
[103]	Safety app.	6 blocks	N/A	Automatic	N/A
[29,30]	Train control	30 blocks	<1 s	Automatic	Automatic
[57]	Nuclear plant	20,000 blocks 9,000 vars	N/A	Automatic	N/A
[87]	N/A	N/A	N/A	Manual	Manual

Table 8 Übersicht von formalen Verifikationslösungen für SPSen mit Funktionsblockdiagrammen (OVATMAN et al. 2014, S. 12)

	App. area	Syst. size	Performance	Auto. level	Formal spec.
[94]	Screw conveyor	74–93 vars	Minutes	Semiauto.	Manual
[106]	Chemical plant	24–93 vars	Minutes	Manual	Manual
[96]	N/A	N/A	N/A	Automatic	Manual
[101]	Machining line	30 vars	Seconds–hours	Automatic	Manual
[119]	Pumping line	39 vars	Minutes	Semiauto.	Manual
[8]	N/A	23–31 places	N/A	Automatic	Automatic
[99]	Gas burning equipment	N/A	N/A	Manual	Manual
[83]	Pinion identifier	N/A	Seconds	Manual	Manual
[33]	Pneumatic	N/A	Seconds	Automatic	Manual

B8. Meta Modell der PLCopenXML

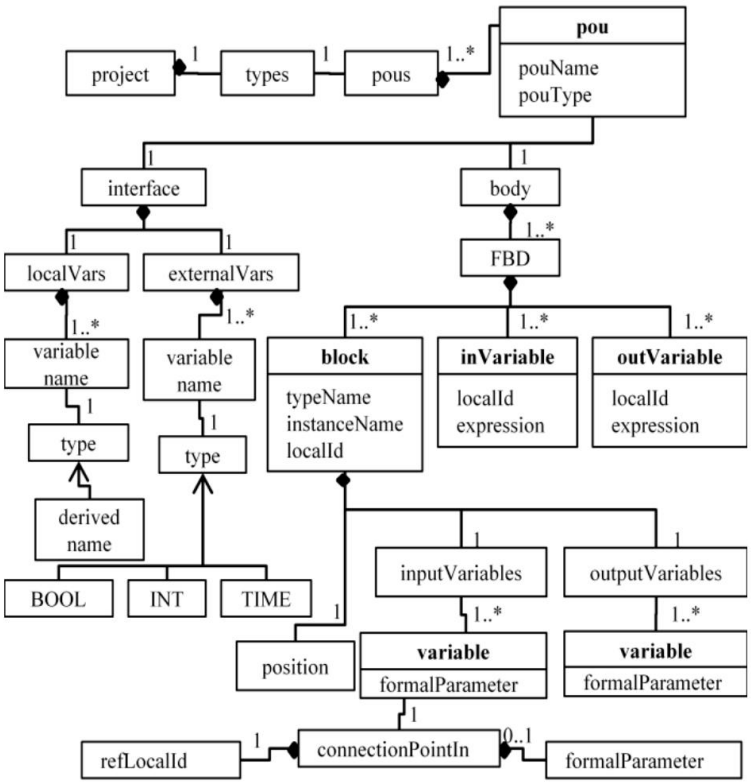


Figure 2 Meta Modell des PLCopenXML Standards (THRAMBOULIDIS et al., 486)

B9. Model-To-Model Transformation von PLCOpenXML für den UPPAAL Model Checker

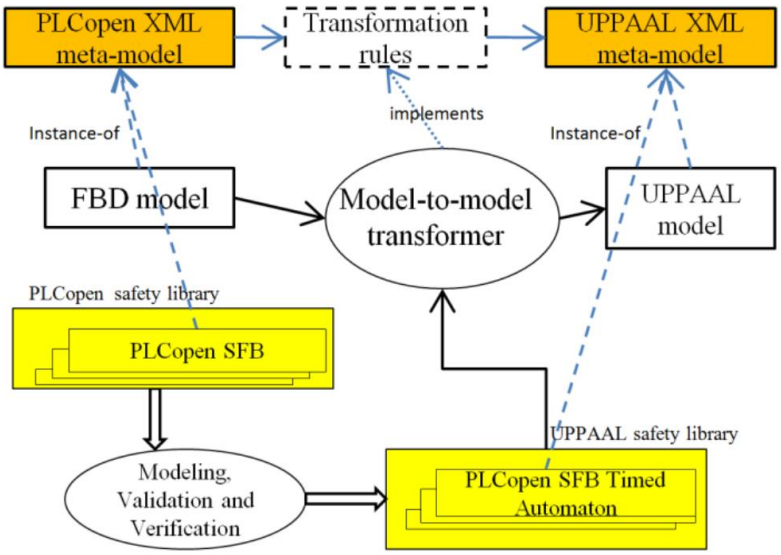


Fig. 4. The proposed transformation process.

Figure 3 Model-To-Model Transformation von PLCOpenXML für den UPPAAL Model Checker (THRAMBOULIDIS et al., S. 485)

C. Literatur

3S SMART SOFTWARE SOLUTIONS GMBH, Automatisiertes Testen von Steuerungsapplikationen/-bibliotheken. CODESYS Test Manager.

A. BECKMANN [XFEL.EU, HAMBURG & GERMANY], Automated Verification Environment for TwinCAT PLC Programs.

AUTILI, M., GRUNSKE, L., LUMPE, M., PELLICCIONE, P. & TANG, A. (2015), Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar. In: IEEE Transactions on Software Engineering, 1, doi: 10.1109/TSE.2015.2398877.

BALZERT, H. (2005), Lehrbuch der Objektmodellierung. Analyse und Entwurf mit der UML 2 ; mit CD-ROM und e-learning-Online-Kurs. Elsevier Spektrum Akad. Verl., Heidelberg.

BECKERT, B., ULBRICH, M., VOGEL-HEUSER, B. & WEIGL, A. (2015), Regression Verification for Programmable Logic Controller Software. In: BUTLER, M., CONCHON, S. & ZAÏDI, F. (Hrsg.), Formal Methods and Software Engineering. Springer International Publishing, Cham, 234–251. 10.1007/978-3-319-25423-4.

BECKHOFF AUTOMATION GMBH & CO. KG, Einführung ADS. http://infosys.beckhoff.com/index.php?content=./content/1031/tcadscommon/html/tcadscommon_introads.htm (11.06.2016).

BECKHOFF AUTOMATION GMBH & CO. KG, Manual Matlab / Simulink.

BECKHOFF AUTOMATION GMBH & CO. KG, Statusmeldungen einer PTP-Achse. Meldungen des zyklischen Achsinterface der Funktionsbausteine NC und MC_xxx. http://infosys.beckhoff.com/index.php?content=./content/1031/tcplclibsnc/html/tcplclibnc_ncsignals.htm&id=.

BECKHOFF AUTOMATION GMBH & CO. KG (2016a), MC_MoveAbsolute. Bausteinbeschreibung. https://infosys.beckhoff.com/index.php?content=./content/1031/tcplclib_tc2_mc2/27021597834317707.html&id= (10.06.2016).

BECKHOFF AUTOMATION GMBH & CO. KG (2016b), Vergleich Datentypen. http://infosys.beckhoff.com/index.php?content=./content/1031/tcsystemmanager/basics/tcsysmgr_datatypecomparison.htm&id= (10.06.2016).

BERNS, K., BERND, S. & TRAPP, M. (2010), Eingebettete Systeme. Systemgrundlagen und Entwicklung eingebetteter Software. Vieweg+Teubner Verlag / GWV Fachverlage GmbH Wiesbaden, Wiesbaden.

BIALLAS, S., BRAUER, J. & KOWALEWSKI, S. (2012), Arcade.PLC: A Verification Platform for Programmable Logic Controllers. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ACM, 338–341.

BRACHT, U., GECKLER, D. & WENZEL, S. (2011), Digitale Fabrik, Berlin, Heidelberg. https://www.vdi.de/uploads/tx_vdirili/pdf/1767683.pdf (23.06.2016).

CAVADA, R. & ALESSANDRO CIMATTI, GAVIN KEIGHREN, EMANUELE OLIVETTI, MARCO PISTORE, MARCO ROVERI, NuSMV 2.6 Tutorial.

CAVADA, R., CIMATTI, A., JOCHIM, C. A., OLIVETTI, E., PISTORE, M., ROVERI, M. & TCHALTSEV, A. (2010), NuSMV 2.6 User Manual.

CLAUS, V. & SCHWILL, A. (2006), Duden Informatik A - Z. Fachlexikon für Studium, Ausbildung und Beruf. Dudenverl., Mannheim.

E.B. BLANCO VINUELA, B. FERNÁNDEZ ADIEGO, A. MEREZHIN [CERN, GENEVA & SWITZERLAND] (2014), Proceedings of the 14th International Conference on Accelerator & Large Experimental Physics Control Systems [also] ICALEPCS2013, San Francisco, California, October 6-11 2013. JACoW, Geneva.

END USER COMPUTING SERVICES (2009), ETFA 2009. 14th IEEE International Conference on Emerging Technologies and Factory Automation : September 22-26, 2009 : University of

Balearic Islands, Mallorca, Spain : 2009 IEEE Conference on Emerging Technologies & Factory Automation : ETFA 2009 PROGRAM. IEEE, [Piscataway, N.J.].

ETFA 2012. Proceedings of 2012 IEEE 17th International Conference on Emerging Techologies & Factory Automation : September 17-21, 2012, Kraków, Poland (2012). IEEE, Piscataway, NJ.

FARINES, J.-M., QUEIROZ, M. H. de, DA ROCHA, V. G., CARPES, A. M. M., VERNADAT, F. & CREGUT, X., A model-driven engineering approach to formal verification of PLC programs. In: Factory Automation (ETFA 2011), 1–8.

FERNANDEZ ADIEGO, B., GONZALEZ SUAREZ, V. & BLANCO VINUELA, E. (2014), Bringing Automated Formal Verification to PLC Program Development. Oviedo U.

FESTO AG & CO. KG (2016), Elektrische Linearantriebe. Übersicht elektrischer Achsen und Antriebe.

GESSLER, R. (2014), Entwicklung eingebetteter Systeme. Vergleich von Entwicklungsprozessen für FPGA- und Mikroprozessor-Systeme Entwurf auf Systemebene. Springer Vieweg, Wiesbaden.

HASSAPIS, G. (2000), Soft-testing of industrial control systems programmed in IEC 1131-3 languages. In: ISA Transactions, 39 (3), 345–355, doi: 10.1016/S0019-0578(00)00029-X.

ICIT 2013-2013 IEEE International Conference on Industrial Technology (ICIT). Proceedings : Pavilion - Clock Tower Conference Centre, Cape Town, South Africa, 25-28 February, 2013 (2013). IEEE, Piscataway, NJ.

JAMRO, M. (2015), POU-Oriented Unit Testing of IEC 61131-3 Control Software. In: IEEE Transactions on Industrial Informatics, 11 (5), 1119–1129, doi: 10.1109/TII.2015.2469257.

KORMANN B., B. V.-H. (2011), Automated Test Case Generation Approach for PLC Control Software Exception Handling using Fault Injection.

KORMANN, B., TIKHONOV, D. & VOGEL-HEUSER, B. (2012), Automated PLC Software Testing using adapted UML Sequence Diagrams. In: IFAC Proceedings Volumes, 45 (6), 1615–1621, doi: 10.3182/20120523-3-RO-2023.00148.

MATHIAS OPPELT, L. U., Integrated Virtual Commissioning an essential Activity in the Automation Engineering Process.

MATTHIAS SEITZ (2012), Speicherprogrammierbare Steuerungen für die Fabrik- und Prozessautomation. In: Speicherprogrammierbare Steuerungen für die Fabrik- und Prozessautomation. Carl Hanser Verlag GmbH & Co. KG.

OVATMAN, T., ARAL, A., POLAT, D. & ÜNVER, A. O. (2014), An overview of model checking practices on verification of PLC software. In: Software & Systems Modeling, doi: 10.1007/s10270-014-0448-7.

OSCAR LJUNGKRANTZ, KNUT ÅKESSON, MARTIN FABIAN & CHENGYIN YUAN (2011), A Formal Specification Language for PLC-based Control Logic.

PRAHOFER, H., SCHATZ, R., WIRTH, C. & MOSSENBOCK, H. (2011), A Comprehensive Solution for Deterministic Replay Debugging of SoftPLC Applications. In: IEEE Transactions on Industrial Informatics, 7 (4), 641–651, doi: 10.1109/TII.2011.2166768.

PRETSCHNER, W. A., Zum modellbasierten funktionalen Test reaktiver Systeme.

RAMLER, R., PUTSCHÖGL, W. & WINKLER, D. (2014), Automated testing of industrial automation software: practical receipts and lessons learned. In: NAIR, A. R., PRÄHOFER, H., ZOITL, A., JETLEY, R., DUBEY, A. & KUMAR, A. (Hrsg.), the 1st International Workshop, 7–16.

REMENSKA, D., WILLEMSE, T. A. C., TEMPLON, J., VERSTOEP, K. & BAL, H. (2014), Property Specification Made Easy: Harnessing the Power of Model Checking in UML Designs. In: HUTCHISON, D., KANADE, T., KITTLER, J., KLEINBERG, J. M., KOBSA, A., MATTERN, F., MITCHELL, J. C., NAOR, M., NIERSTRASZ, O., PANDU RANGAN, C., STEFFEN, B., TERZOPOULOS, D., TYGAR, D., WEIKUM, G., ÁBRAHÁM, E. & PALAMIDESSI, C. (Hrsg.), Formal Techniques for Distributed Objects, Components, and Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, 17–32. 10.1007/978-3-662-43613-4.

RÖSCH S., TIKHONOV, D. & VOGEL-HEUSER, B. (2013), Testen in der Automatisierungstechnik. Anforderungen und Lösungsansätze. In: VOGEL-HEUSER, B. (Hrsg.), Engineering von der Anforderung bis zum Betrieb. Kassel University Press, Kassel, 41–50.

SCHETININ, N., MORIZ, N., KUMAR, B., MAIER, A., FALTINSKI, S. & NIGGEMANN, O., Why do verification approaches in automation rarely use HIL-test? In: 2013 IEEE International Conference on Industrial Technology (ICIT 2013), 1428–1433.

SCHOLZ, P. (2005), Softwareentwicklung eingebetteter Systeme. Grundlagen, Modellierung, Qualitätssicherung. Springer, Berlin.

SCHNEIDER, K. (2007), Abenteuer Softwarequalität. Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement. dpunkt-Verl., Heidelberg.

SULFRIAN, A., Modultests auf Speicherprogrammierbaren Industriesteuerungen.

SUSANNE ROESCH, DMITRY TIKHONOV, DANIEL SCHÃ¼TZ & BIRGIT VOGEL-HEUSER (2014), Model-Based Testing of PLC Software: Test of Plants' Reliability by Using Fault Injection on Component Level.

THRAMBOULIDIS, K., SOLIMAN, D. & FREY, G., Towards an automated verification process for industrial safety applications. In: 2011 IEEE International Conference on Automation Science and Engineering (CASE 2011), 482–487.

VIGENSCHOW, U. (2010), Testen von Software und Embedded Systems. Professionelles Vorgehen mit modellbasierten und objektorientierten Ansätzen. dpunkt.Verl., Heidelberg.

WÖRN, H. & BRINKSCHULTE, U. (2005), Echtzeitsysteme. Grundlagen, Funktionsweisen, Anwendungen ; mit 32 Tabellen. Springer, Berlin.

WÜNSCH, G. (2008), Methoden für die virtuelle Inbetriebnahme automatisierter Produktionssysteme. Techn. Univ., Diss.--München, 2007. Utz, München.