Modultests auf speicherprogrammierbaren Industriesteuerungen

Alexander Sulfrian
Bachelorarbeit

Berlin, 14. Januar 2011
mit Unterstützung durch Weber Maschinenbau GmbH

Inhaltsverzeichnis

1 Einleitung **5**
1.1 Motivation . . . 5

2 Grundlagen **7**
2.1 Testen . . . 7
2.2 Unterscheidung von Tests . . . 8
2.2.1 Black-Box-Tests . . . 8
2.2.2 White-Box-Tests . . . 8
2.2.3 Grey-Box-Tests . . . 9
2.3 Unittests . . . 9
2.3.1 Was wird getestet? . . . 9
2.3.2 Wie werden Tests geschrieben? . . . 10
2.3.3 Wann wird getestet? . . . 11
2.3.4 Wie wird getestet? . . . 12

3 EPAS-4 **15**
3.1 CoDeSys . . . 15
3.1.1 Programmierung in Zyklen . . . 15
3.1.2 IEC-61131-3 . . . 16
3.1.3 Projektformat . . . 17
3.1.4 Versionsverwaltung mit ENI . . . 17
3.2 Automatisierbarkeit . . . 18
3.2.1 Makros . . . 18
3.2.2 CMD-Befehle . . . 19
3.2.3 TypeLibrary . . . 20
3.2.4 Simulation von Mauseingaben . . . 22
3.3 Testumgebung: PacUnitTest . . . 22
3.3.1 Funktionen . . . 22
3.3.2 Programmierung von Testfällen . . . 23
3.4 Automatisierung von Tests . . . 24
3.4.1 Infrastruktur für die Tests . . . 24
3.4.2 Ausführung der Tests . . . 25
3.5 Ausblick . . . 26

4 SoMashine Motion **27**
4.1 CoDeSys 3 . . . 27

4.1.1 Software . . . 27
4.1.2 Hardware . . . 29
4.2 Versionsverwaltung . . . 29
4.3 Unittests in EPAS-5 . . . 30
4.3.1 ETest . . . 30
4.3.2 Automatisierung . . . 31

5 TwinCAT 3 **33**
5.1 Aktuelle Situation . . . 33
5.2 Kombinierte Entwicklung in der nächsten Version . . . 33
5.2.1 Vorteile . . . 35
5.2.2 Nachteile . . . 35
5.3 Fazit . . . 36

A Quellcode **37**
A.1 ProjectBuilder . . . 37
A.2 Network PutExecuter . . . 41

B Literaturverzeichnis **49**

C Abbildungsverzeichnis **51**

D Selbständigkeitserklärung **53**

1 Einleitung

1.1 Motivation

Tests gehören in den Hochsprachen schon lange zu dem Standardrepertoire in der Softwareentwicklung. Kent Beck entwickelte schon vor dem Jahre 1997 als erster das Konzept von vielen aktuellen Testumgebungen für Hochsprachen. 1999 veröffentlichte er seine Gedanken und Ausarbeitungen und stellte SUnit für Smalltalk vor [Bec99]. Zusammen mit Erich Gamma entwickelte er parallel dazu seit 1997 an einer Portierung seiner Umgebung auf die Java Plattform [Fow10]. Im Jahre 1998 veröffentlichte er diese bis heute erfolgreiche Testumgebung JUnit[1]. SUnit sowie JUnit stellten in den folgenden Jahren ein Vorbild für viele andere Programmiersprachen dar und es entstanden praktisch für alle Hochsprachen Portierungen. Selbst für exotische Sprachen wie zum Beispiel Haskell (HSUnit) oder MATLAB (mlUnit) sind heute Testumgebungen nach dem Vorbild von SUnit entstanden. Heute fasst man alle diese Portierungen unter dem gemeinsamen Namen xUnit zusammen. Der Name leitet sich aus dem Namen JUnit ab. Wobei das "x"für eine der vielen Programmiersprache steht, für die ein solches Testframework erhältlich ist. Außerdem ist xUnit aber auch eine Anspielung auf das Verhältnis zu Ëxtreme Programming"(auch XP, oder XProgramming). Die Testframeworks legten einen wichtigen Grundstein um die testgetriebene Softwareentwicklung zu ermöglichen.

Mit den Testumgebungen war die Basis für erfolgreiches Testen gelegt. In Folge dessen entstanden viele weitere Tools, die helfen sollten, die Software möglichst komplett und einfach Testen zu können. Es gibt zum Beispiel Hilfsmittel, die auswerten können, welche Bereiche des Quellcodes von den Tests durchlaufen und somit getestet wird. Außerdem gibt es Hilfsmittel um einzelne Programmteile einfacher zu testen im dem man andere benötigte aber nicht oder nur schwer kontrollierbare Teile durch so genannte "Mockobjekteäustauscht, deren Zustand man steuern kann.

Alle diese Hilfsmittel sind entstanden, damit es so einfach wie möglich wird Tests für eine Software zu schreiben.

Die eben beschriebenen Entwicklungen gelten jedoch fast nur Hochsprachen. Im Maschinenbau wird meistens keine Standardhardware eingesetzt. Die seit geraumer Zeit dort eingesetzten Speicherprogrammierbaren Steuerungen (kurz SPS oder PCL[2]) werden fast

[1] JUnit Version 1.0 wurde am 28.02.1998 veröffentlich: http://www.eg.bucknell.edu/ cs475/F2000-S2001/hyde/JUnit/README.html

[2] vom englischen "Programmable Logic Controller"

ausschließlich in speziell dafür angepassten Sprachen programmiert. Ein weiteres Problem ist auch, dass im Maschinenbau bisher vorwiegend Ingenieure eingesetzt wurden. Die Ingenieure, die früher die fest verdrahteten Schaltkreise mit Logikbausteinen erstellten, haben sich an die neue Art der Steuerung und Regelung von Anlagen angepasst und programmieren jetzt die Steuerungen. Die Entwicklung von Grundsätzen, die in der Softwareentwicklung schon lange bekannt sind, muss dort quasi erneut durchlaufen werde.

Nachdem nun einige Hersteller die Notwendigkeit einer Testumgebung erkannt haben, soll diese Arbeit den aktuellen Zustand und die Entwicklung dieser Umgebung in drei unterschiedlichen Entwicklungsumgebungen untersuchen. Dabei sollen die Unterschiede der Testsysteme und die daraus resultierenden Vorteile und Nachteile für die praktische Verwendung aufgezeigt werden. Zuvor soll noch einmal auf die Grundlagen des Unittesting unabhängig der eingesetzten Entwicklungsumgebung eingegangen werden.

2 Grundlagen

2.1 Testen

Testen ist eine der grundlegenden Techniken der Softwareentwicklung. Jedes Programm wird irgendwann mindestens einmal manuell getestet und zwar durch den Endkunden. Sogar der Aufruf des Compilers ist schon ein Test auf korrekte Syntax.

Spricht man aber von Tests in der Software Entwicklung versteht man darunter eigentlich Quelltext, der dazu dient die geschriebenen Funktionalitäten der Programms automatisch zu überprüfen und die Korrektheit zu sichern.

Andrew Hunt und David Thomas formulierten in ihrem Buch *Der Pragmatische Programmierer* den Tipp:

> Test Early. Test Often. Test Automatically.[HT99]

Damit formulieren sie drei wichtige Grundsätze: Sobald Quellcode vorhanden ist, sollten Test entstehen, damit kleine Fehler nicht unbemerkt zu großen Fehlern werden können. Außerdem sind kleine Fehler weitaus schneller zu beheben als große Fehler, die ein ganzen Teil der Anwendung betreffen. Damit Fehler allerdings gefunden werden können, müssen die Tests nicht nur existieren, sondern auch häufig (beispielsweise nach jeder Änderung in einem Versionsverwaltungssystem oder jede Nacht) ausgeführt werden, damit die Änderungen die zu dem Fehler möglichst genau eingegrenzt werden können und die Fehlerbehebung vereinfachen. Dies führt auch schon zum dritten Teil des Tipps: automatisches Testen. Niemand möchte sich stundenlang damit beschäftigen die Tests auszuführen. Außerdem dürfen sie nicht vergessen werden. Deshalb sollten Tests automatisch anhand von den schon genannten Faktoren ausgeführt werden. Dadurch können sie nicht vergessen werden und die Entwickler brauchen sich nur um die Fehlerbehebung kümmern und nicht um die Ausführung der Tests.

Die Erfahrung zeigt, dass Tests vernachlässigt werden wenn es keine gute Unterstützung des Entwicklers durch die Entwicklungsumgebung oder andere automatisieren Systeme gibt. Dies führt dazu, dass Entwickler den Mehrwert der Tests nicht schätzen und dann teilweise sogar keine Tests schreiben.

Einige gehen sogar noch weiter als *Der Pragmatische Programmierer*. In seinem Buch über des *Extreme Programming* beschreibt Ron Jeffries Anfang dieses Jahrtausends eine bis dato nicht da gewesene Weise der Softwareentwicklung. Die Technik des *Extrem Programmings* stellt die eigentliche Programmierung in den Vordergrund und stellt formale

Zwänge und Vorgaben hinten an. Damit werden die Anforderungen des Zielproduktes in kleinen Schritten erfüllt. Um dabei sicher zu sein, dass jede dieser kleinen Schritte das richtige Ergebnis erbracht und keine vorherigen Schritte beschädigt hat sind kontinuierliche Tests hier auch unentbehrlich. Jeffries geht sogar so weit, dass er beschreibt, welche Vorteile es hat die Testfälle vor dem eigentlichen Quellcode zu entwickelt.[Jef01]

Bei allen Betrachtungen zu unterschiedlichen Tests darf man aber einen wichtigen Fakt nicht vernachlässigen, den der Turing Award Preisträger Edsger W. Dijkstra schon im Jahre 1970 festgestellt hat:

> Program testing can be used to show the presence of bugs, but never to show their absence![Dij70]

Er betonte damit besonders, dass egal wie gut die Tests auch sein mögen, man kann immer nur Fehler in einer Software finden aber niemals zeigen, dass eine Software keine Fehler hat. Um zu zeigen, dass eine Software keine Fehler hat, muss man sich anderen Techniken zuwenden und beispielsweise die Korrektheit von einem Programm mit dem Hoare-Kalkül beweisen.

2.2 Unterscheidung von Tests

Es gibt verschiedene Arten von Tests. Eine Möglichkeit Tests zu unterscheiden, ist anhand der Menge an Informationen die der Testautor über das zu testende System hat. Danach unterscheidet man die Black-Box-Tests, White-Box-Tests und eine neue Mischform die Grey-Box-Tests.

2.2.1 Black-Box-Tests

Black-Box-Tests werden für ein unbekanntes System entwickelt. Es ist lediglich bekannt welche Eingaben das System akzeptieren soll und welche Rückgaben es bei korrekter Funktionsweise liefern soll. Dabei wird sehr gut das Gesamtsystem getestet und es besteht nicht die Gefahr, dass bestimmte Schwachstellen in der Implementierung absichtlich nicht getestet werden, da diese ja nicht bekannt sind. Im Gegensatz dazu steht allerdings ein erhöhter organisatorischer Aufwand, da die Tests nicht von dem Entwickler geschrieben werden dürfen, da dieser mit den internen Details vertraut ist. Außerdem muss für einen guten Black-Box-Test eine möglichst genaue Spezifikation vorhanden sein.

2.2.2 White-Box-Tests

Im Gegensatz zu den Black-Box-Tests weiß bei einem White-Box-Test der Entwickler genau was er testet und kennt alle internen Details der Implementierung. Mit Hilfe von Hilfsmitteln können White-Box-Tests teilweise sogar automatisch erzeugt werden (es

könnten zum Beispiel Eingaben so generiert werden, dass alle Verzweigungen in dem zu testenden Quellcode jeweils einmal wahr und einmal falsch werden) und die Qualität der Tests kann ermittelt werden. Im Allgemeinen sind einzelne kleine Module besser mit White-Box-Tests testbar. Da der Entwickler des Tests sowieso über genau über den internen Aufbau Bescheid wissen muss, kann der eigentliche Softwareentwickler White-Box-Tests parallel zur Arbeit am eigentlichen Quellcode schreiben.

2.2.3 Grey-Box-Tests

Besonders beim *Extreme Programming* versucht man die Vorteile der Black-Box-Tests und der White-Box-Tests zu verbinden. Dazu werden die Tests von dem eigentlichen Entwickler geschrieben bevor die eigentliche Implementierung erfolgt. Dadurch wird der hohe Aufwand für einen Black-Box-Tests weites gehend vermieden aber trotzdem läuft man nicht so schnell Gefahr auf Grund der Informationen zu der internen Funktionsweise bestimmte Fälle nicht oder nur Unzureichend zu testen. Allerdings sollen die Grey-Box-Tests nicht die Black-Box-Tests ersetzen. Diese benötigt man immer noch um das Gesamtsystem zu testen, da obwohl jede einzelne Komponente funktioniert noch lange nicht das Gesamtsystem funktionieren muss. Die Grey-Box-Tests sollte man ehre als Erweiterung und Ergänzung zu den White-Box-Tests sehen.

2.3 Unittests

Die am häufigsten angewendete Form der Tests sind die Modultests oder auch Unittests. Unittests sind White-Box-Tests oder Grey-Box-Tests von möglichst kleinen Modulen des Projektes.

2.3.1 Was wird getestet?

Die erste Frage mit der man sich in Bezug auf Unittests beschäftigen sollte, ist die Frage: *"Was ist eigentlich eine Unit?"*

Eine Unit, die mit einem Unittest getestet werden soll, ist ein möglichste kleines abgeschlossenes Modul einer Anwendung. Dies können einzelne Funktionen oder Gruppen von Funktionen, im objektorientierten Sinne auch Klassen oder zusammengehörende Klassen eines Designmusters sein.

Dies kleinen Module sollten so gestaltet sein, dass sie eine möglichst kleine und definierte Schnittstelle nach Außen haben. Bei einem Test müssen möglichst alle äußeren Einflüsse ersetzt werden, damit diese Einflüsse kontrollierbar sind und die Tests ein definiertes Ergebnis besitzen. Diese äußeren Einflüssen können beispielsweise eine Datenbankverbindung sein oder sogar die Netzwerkverbindung zwischen zwei Programmteilen aber

auch andere Module im Programm selber können die definierte Testausführung behindern.

2.3.2 Wie werden Tests geschrieben?

Es gibt viele Entwickler, die das Schreiben von Tests als unnütze Arbeit ansehen, bei der man den kompletten Sourcecode einer Funktion nochmal im Tests möglichst auf einem anderen Weg implementieren muss. Mit dieser Einstellung wird wird die Erstellung von Tests meistens ans Ende der Entwicklung verschoben und verläuft dann im Sand. Dabei muss das schreiben von Tests gar nicht so aufwendig sein.

Das Schreiben von Tests kann schon bei der Implementierung der eigentlichen Funktion unterstützt werden. Für eine einfache Erstellung von Tests sollte eine Funktion möglichst wenig externe Abhängigkeiten direkt aufrufen. Aufrufe an beispielsweise Datenbanken sollten so gekapselt sein, dass sie bei einem Test einfach mit einem entsprechenden Testmodul durchgeführt werden können, dass vordefinierte Ergebnisse zurück liefert.

Der einfachste Fall von einem Test prüft bei einer erwarteten Eingabe ob das Ergebnis korrekt ist. Dabei können nie alle Eingaben überprüft werden. Daher sollte der Tests sich auf Extremwerte konzentrieren.

Zu den positiven Tests, sollten wenn möglich auch negative Tests durchgeführt werden. Dabei soll überprüft werden, ob das zu testende Objekt sich bei fehlerhaften Eingaben auch definiert verhält.

Eine häufige Frage von Entwicklern ist immer wieder: *Wie viele Test muss ich denn schreiben?* Diese Frage kann nicht so einfach beantwortet werden. Es gibt schon einige Theorien, an welchen Kriterien man den Stand der Tests messen kann. Eine davon ist beispielsweise die Codeabdeckung. Glenford J. Myers beschreibt in seinem Buch: *The art of software testing* ebenfalls verschiedene Kriterien zum Einschätzen der Testqualität. Der Kriterium der Codeabdeckung beschreibt dabei, dass jede Quellcodezeile mindestens einmal durch Tests ausgeführt werden muss. Er beschreibt, dass bei Bedingungen in Quellcode eine einmalige Ausführung möglicherweise nicht reicht um Fehler zu finden. Somit kommt er zu dem Schluss:

> In other words, the statement-coverage criterion is so weak that it generally is useless.[MBTS04]

Das Kriterium der Codeabdeckung kann ihm nach, also meistens nur zur Präsentation einer Zahl für den Vorgesetzten dienen. Eine geringe Codeabdeckung spricht zwar für schlechte Tests, aber andersherum heißt eine hohe Abdeckung nicht dass die Qualität der Tests auch gut ist.

Als perfektes Kriterium beschreibt Myers die komplette Abdeckung aller möglichen Programmabläufe. Damit wäre sichergestellt, dass die Funktion in allen Fällen richtige Ergebnisse liefern würde. Allerdings ist dieses Ziel unrealistisch in modernen Programmen. Durch Schleifen oder Rekursion kann es unendlich viele Programmabläufe in Abhängigkeit von den Eingaben geben.

Man benötigt also ein Kriterium, dass in der Mitte zwischen Abarbeitung aller Programmabläufe und nur die Ausführung jeder Quellcodezeile. Ein Kriterium, was diese Anforderung erfüllt, ist das Kriterium der Pfadabdeckung. Dabei muss jede Bedingung innerhalb des zu testenden Moduls einmal wahr und einmal falsch gewesen sein, damit alle Verzweigungen getestet werden können. Tests nach diesem Kriterium können sogar automatisch generiert werden. Ein Generator könnte die Eingaben anhand der Entscheidungen automatisch auswählen und der Entwickler müsste nur noch die erwarteten Ausgaben zuweisen. Das einzige Problem dabei ist, dass sich die Möglichkeiten mit welchen Eingaben ein Modul aufgerufen werden soll, mit jeder Entscheidung innerhalb dieses Moduls, exponentiell erhöht und es damit ähnlich wie bei der kompletten Pfadabdeckung nicht praktisch ist alle Möglichkeiten zu testen.

Somit bleibt festzuhalten, dass der Entwickler selber entscheiden muss wie viele Tests er schreibt. Eine Analyse der Erfüllung von verschiedenen Kriterien kann dabei helfen. Dabei ist allerdings wichtig, dass von der Sprache und der Testumgebung geeignete Werkzeuge angeboten werden um zu ermitteln welche Quellcodezeilen mit welchen Werten in bestimmten Variablen ausgeführt wurden.

Einen Tipp für das Schreiben von Tests gibt es allerdings noch. Jedes Mal wenn ein Fehler von einem Tester oder Kunden gefunden wird, sollte genau für diesen Fehler ein Test geschrieben werden.[HT99] Damit wird sichergestellt, dass ein Feher der einmal aufgetreten ist nicht ein weiteres mal auftritt.

2.3.3 Wann wird getestet?

Getestet werden sollte nach jeder Änderung am Quelltext. Dies lässt sich beispielsweise relativ einfach mit einem Versionsverwaltungssystem realisieren. Dabei ist es entweder möglich, dass ein spezieller Server (ein so genannter Continuous Integration) die Versionsverwaltung ständig nach neuen Versionen abfragt und dieser dann bei einer neuen Version ein vollständiges Bauen des Projektes auslöst und die vorhandenen Tests ausführt. Eine andere Möglichkeit kann sein, dass spezielle Funktionen der Versionsverwaltung (Hooks[1]) genutzt werden um dem Server mitzuteilen dass eine neue Version bereit steht.

Bei einigen Projekten ist die Zeit, die das Projekt zur Übersetzung und zur Ausführung der Tests benötigt, großer als die durchschnittliche Zeit bis eine neue Sourcecodeänderung eingepflegt wird. In diesem Fall gibt es zwei Möglichkeiten. Einerseits kann man überlegen

[1] Hooks sind Funktionen die vor oder nach bestimmten Aktionen aufgerufen werden und das Verhalten von Programmen beeinflussen können.

nur einen neuen Test anzustoßen, wenn ein älterer schon beendet ist. Andererseits ist es in einem solchen Fall auch möglich nicht bei jeder Änderung die Tests auszuführen sondern regelmäßig jede Nacht, wenn besonders wenige Änderungen stattfinden sollten. Die zweite Möglichkeit bietet einige Nachteile gegenüber der ersten Möglichkeit. Durch die verzögerte Ausführung der Tests ist es dann nicht mehr eindeutig festzustellen, welche Änderung genau zu dem Fehler geführt hat. Außerdem wird es dadurch unmöglich, den Entwickler, dessen Änderung zu dem Fehler geführt hat direkt zu benachrichtigen. Das setzt höhere Ansprüche an Disziplin. Jeder Entwickler muss selbst am Morgen das Ergebnis der Tests überprüfen und eventuelle Fehler beseitigen. Werden Tests aber direkt nach einer neuen Version des Quellcodes ausgeführt, kann der Entwickler gleich über Fehler benachrichtigt werden, diese kann nicht vergessen eventuelle Fehler anzusehen und kann diese dann eventuell auch schneller wieder reparieren da der Gedankengang der zu dem Quellcode geführt hat, noch nicht so weit zurückliegt.

Wichtig bei der Ausführung der Tests ist, wie schon dem *pragmatischen Programmierer* empfohlen wurde, dass die Test auf jedem Fall automatisch ausgeführt werden. Wenn nur eine manuelle Ausführung möglich ist, wird es irgendwann auftreten, dass die Ausführung über längere Zeit vernachlässigt wird (sei es durch Stress der Entwickler oder einfaches vergessen). Bei der nächsten Ausführung können dann Fehler auftreten, die schwer mit einer Änderung in Verbindung zu bringen sind und damit auch schwerer zu beheben sind.

Auch in eine solchen Fall ist der Vorteil von Test natürlich nicht gleich total verloren. Es ist immerhin noch Möglich einen Fehler zu finden, den man ohne Tests vielleicht erst nach der Auslieferung beim Kunden festgestellt hätte. Aber die Tests haben einen noch viel größeren Nutzen wenn eindeutig festgestellt werden kann welche Änderung zu dem Fehler geführt hat und die Fehler damit schnell behoben werden können.

Es gibt auch Testumgebungen die nur aus einer Funktion in der Programmiersprache selber bestehen. Dieses ist zum Beispiel die Funktion `assert()` in C. Diese Funktion dient zum Testen von Vor- und Nachbedingungen während der Ausführung des Programms. Viele Entwickler nutzen dies Funktion nur während der Entwicklung und nutzen die Möglichkeit diese Überprüfungen in dem fertigen Produkt zu entfernen. Dies ist allerdings nicht empfehlenswert. Selber durch intensives Testen kann man nicht alle Fehler während der Entwicklung finden. Deshalb geht das Testen auch beim Kunden weiter und dort hilft jede Meldungen, die auf auf den Fehler hinweist.[HT99]

2.3.4 Wie wird getestet?

Je nach der eingesetzten Testumgebung besteht das Testen meistens aus dem Ausführen des Programms mit einem speziellen Parameter, das übersetzen des Quellcodes mit speziellen Parametern oder die Ausführung eines Werkzeugs der Testumgebung im dem Quellcodeverzeichnis.

Das Testen beginn meist damit, dass die aktuelle Version alle genötigten Quellcodedateien aus einer zentralen Versionsverwaltung abgerufen werden. Danach werden alle benötigten Bibliotheken und Voraussetzungen von dem Projekt bereitgestellt und müssen dabei gegebenenfalls erneut übersetzt werden. Sind dann alle Voraussetzungen vorhanden wird damit begonnen das Projekt zu übersetzen und Datei, die ausgeliefert werden könnte erzeugt (dies ist schon ein erster Test auf syntaktische Korrektheit und auf Kompatibilität mit einem anderen System als der Rechner des Entwicklers). Erst danach sollte damit begonnen werden die Dateien, die für die Testausführung benötigt werden zu erzeugen.

Abhängig von der Testumgebung ist es manchmal möglich nur einzelnen Tests auszuführen. Dies ist bei der testgetriebenen Entwicklung hilfreich, da nach der Entwicklung nur der einzelne Test ausgeführt werden muss um die Funktion zu überprüfen. Unabhängig davon, ob die Möglichkeit (wie beispielsweise bei JUnit, im Gegensatz zu CPPUnit) besteht ist bei der automatisierten Testausführung darauf zu achten, dass immer alle Tests ausgeführt werden, die im aktuellen Projekt vorhanden sind. Es kann nicht ausgeschlossen werden, dass eine Änderung in einem Bereich des Projektes durch eine Nebenbedingung auch den Tests in einem ganz anderen Bereich des Projektes beeinflusst.

Außerdem ist darauf zu achten, dass die Testausführung nicht hängen bleibt. Gute Testumgebungen können beispielsweise die maximale Zeit die ein Test benötigt begrenzen. Sollte dies nicht möglich sein, sollte zumindest bei der automatischen Ausführung der Server eine maximale Beschränkung eingebaut haben und bei Überschreitung den Test als Fehler markieren. Einige Continuous Integration Server können diese Zeit sogar dynamisch an die Zeit, die Tests bisher benötigt haben anpassen.

Hilfreich für die Auswertung der Tests ist es, wenn die Testumgebung ein standardisiertes Ausgabeformat anbietet. Dies kann das Testsystem dann automatisch auswerten, bei Fehlern die Entwickler benachrichtigen und die Ergebnisse beispielsweise auf einer Internetseite aufbereitet anzeigen. Für dieses Zweck hat sich das XML-Format von JUnit durchgesetzt, dass auch schon von vielen anderen Testumgebungen unterstützt wird. Im Lieferumfang der Testumgebung gibt es meistens XSLT-Scripte, mit denen man ein XML Dokument (das zur maschinellen Auswertung gedacht ist und nicht so einfach von einem Entwickler überblickt werden kann) schön auf mehreren HTML-Seiten anzeigen lassen kann. Dank der guten Unterstützung für XML in diversen Programmiersprachen ist es damit aber auch kein Problem, die Ergebnisse automatisch auszuwerten.

3 EPAS-4

EPAS-4 ist die aktuelle Entwicklungsumgebung der Firma Schneider Electric (ehemals Elau). Zusammen mit den Steuerungen bildet EPAS-4 die aktuelle Plattform PacDriveM.

3.1 CoDeSys

Diese Entwicklungsumgebung basiert auf einem System der Firma 3s. Die Firma 3s stellt ein Entwicklungsumgebung für Maschinensteuerungen her. Diese wird an Hersteller solcher Steuerungen verkauft und von diesen dann an die konkrete Steuerung angepasst und mit kleinen Erweiterungen versehen. Über 250 Firmen nutzen die CoDeSys Entwicklungsumgebung als Basis für die Programmierung ihrer Hardware.

EPAS-4 basiert auf der Version 2.4 von CoDeSys. Diese Software ist mittlerweile einige Jahre alt. Auch von EPAS-4 hat es seit 4 Jahren fast keine Updates mehr gegeben. Es ist nicht zu erwarten, dass es für die aktuelle Version noch Updates geben wird, da seit 3 Jahren die nächste Version in Entwicklung ist und mittlerweile auch Hardware zur Verfügung steht. Die Benutzerobfläche sieht dem Alter entsprechend nach Windows 3.1 aus (siehe Abbildung 3.1).

3.1.1 Programmierung in Zyklen

Das Programmiersystem zur Programmierung der Steuerungen ist nicht mit der klassischen Programmierung auf einem PC zu vergleichen. Die Steuerung besitzt ein Echtzeitbetriebssystem und führt eine oder mehrere Funktionen in festen Intervallen, die bis zu 2ms kurz sein können, immer wieder aus. Sollte die Ausführung der Funktion mal länger brauchen wird entweder die Ausführung abgebrochen oder das komplette Programm auf der Steuerung stürzt ab.

Damit die Programm Ausführung bei jedem Aufruf weiter gehen kann, werden an vielen Stellen in der Programmierung Statusmaschinen verwendet. Dies sind meistens switch-Anweisungen die verschiedene Werte für einen Status durchlaufen können und für jeden Wert des Status gibt es bestimmte Bedingungen unter denen er in einen anderen Status wechselt. Vorteil bei dieser Weise zu programmieren ist auch, dass man beim Ausführen später relativ einfach sieht, in welchem zustand sich das Programm befindet und auf welche Ereignisse gerade gewartet werden.

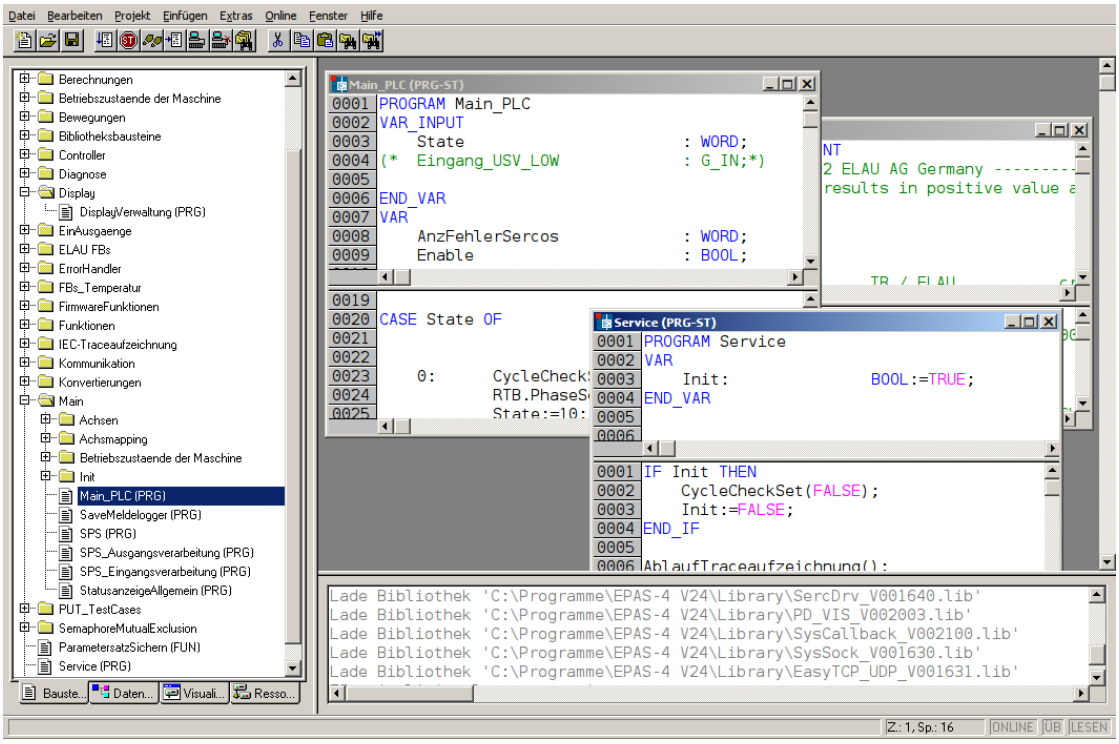

Abbildung 3.1: Oberfläche von EPAS-4: Multi-Dokument-Interface für die Bearbeitung von mehreren Bausteinen parallel

3.1.2 IEC-61131-3

Die Sprachen in denen programmiert werden kann, sind standardisiert. Der Standard IEC-61131-3 beschreibt ganz genau welche Datentypen es gibt und wie die verschieden Quellcodeobjekte aussehen.

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 feste Länge noch Strukturen die aus einfachen Datentypen oder anderen Strukturen zusammengesetzt sind.

In der CoDeSys Version 2.3 sind leider keine objektorientierte Programmierung möglich. Es gibt keine Klassen und keine Vererbungshierarchie.

Die Syntax der strukturierten Text Sprache ähnelt sehr stark Pascal. Neben dieser Sprache ist die Programmierung noch in einem Assembler ähnlichen Dialekt möglich oder in verschiedenen graphischen Programmiersprachen die in einigen Fällen ein übersichtliches Programmieren ermöglichen. Dadurch dass allerdings von diese Sprachen sehr schlecht

die Versionsgeschichte gespeichert werden kann, wird heute hauptsächlich Strukturierter Text eingesetzt.

Im Gegensatz zu anderen Sprache die Klassen, Interfaces und Funktionen bieten, ist in IEC-61131-3 nur definiert, dass Funktionen, Funktionsblöcke und Programme unterstützt werden.

Funktionen sind dabei nicht mit den Funktionen aus andern Programmiersprachen zu verwechseln. Hier sind Funktionen nebeneffektfrei und können deshalb nicht auf globale Variablen zugreifen und haben keinen internen Status.

Funktionsblöcke sind ähnlich wie Klassen. Sie können eigene Variablen haben. Allerdings können Funktionsblöcke keine Methoden haben sondern sind selber nur eine Funktion die mit Eingabe- und Ausgabeparametern aufgerufen werden kann. Bevor die allerdings möglich ist, muss eine Instanz von dem Funktionsblock erzeugt werden. Jede Instanz hat ihre eigenen Variablen. Es gibt keinen gemeinsamen Bereich an Variablen für alle Funktionsblöcke vom gleichen Typ.

Programme sind ähnlich wie Funktionsblöcke. Sie können eigene Variablen definieren und müssen aufgerufen werden. Allerdings gibt es von Programmen immer nur eine Instanz und man muss diese nirgends definieren.

Sichtbarkeitsbereiche gibt es kaum. Nur bei der Einbindung von externen Bibliotheken kann man auf interne Variablen leider nur lesend zugreifen.

3.1.3 Projektformat

Projektdateien werden in EPAS-4 und CoDeSys in einem binären Format abgespeichert. Daher ist es leider nicht möglich ohne die Entwicklungsumgebung einzelne Bausteine zu betrachten oder zu ändern. Intern ist das ganze Projekt in einem Baum eingeordnet. Unter der Wurzel sind verschiedene Knoten für verschiedene Elemente im Programm. Die Bausteine die Code enthalten sind so logisch von den Datentypen und den Konstanten und globalen Variablen getrennt. Auch die Einstellungen für die Steuerung sind extra abgelegt.

Alle Bausteine werden intern in einem XML-Format abgelegt. Leider hat man als Benutzer auf dieses Format keinen Zugriff.

3.1.4 Versionsverwaltung mit ENI

Da das Projekt in einer binär Datei gespeichert wird, kann man außerhalb der Entwicklungsumgebung keine Versionverwaltung sinnvoll nutzen. Auch die Zusammenarbeit an einem Projekt ohne Unterstützung durch die Entwicklungsumgebung wäre unmöglich.

Damit diese Probleme zumindest ein wenig gemildert werden, ist in EPAS-4 eine Versionsverwaltung integriert. Das so genannte *Engineering Interface* biete an die Bausteine des Projektes auf einem zentralen Server[1] abzulegen und bestimmte Bausteine als gemeinsame Bausteine allen zur Verfügung zu stellen. Dabei setzt ENI nicht wie zum Beispiel Subversion das Copy-Modify-Merge Verfahren ein sondern das ältere Lock-Modify-Unlock. Dabei kann im Gegensatz zum Verfahren von Subversion immer nur einer an jeder Datei arbeiten und die Datei bleibt bis der Benutzer diese wieder freigibt für andere Benutzer nur lesbar.

Leider kann der ENI-Client, der in EPAS-4 eingebaut ist, nur sehr wenige Funktionen einer Versionverwaltung. Es ist nicht möglich herauszufinden welche Version der Datei man lokal vorliegen hat und ob es eine neuere Version gibt. Dies ist nur möglich wenn man gleich alle Änderungen in seine lokale Version einfügt.

Neuere Versionsverwaltungssysteme wie zum Beispiel auch Git machen es dem Benutzer explizit schwer, Daten zu vernichten. Bei ENI kann man einzelne Bausteine von der Versionsverwaltung. Beim späteren wieder Einfügen muss man sich entscheiden ob man alle lokalen Änderungen verlieren möchte oder die Version auf dem Server mit der lokalen Version überschreibt. Dabei ist es allerdings nicht möglich die Unterschiede zwischen den drei Versionen anzuzeigen und ob ein anderer Benutzer die Datei inzwischen editiert hat.

Wenn man dazu noch bedenkt, dass es ohne Netzwerkverbindung zum ENI-Server nicht möglich ist einen Baustein zu locken. Will man beispielsweise beim Kunden vor Ort eine Änderung einpflegen hat man nur die Möglichkeit einen Baustein zu bearbeiten, wenn man das komplette Projekt oder den einzelnen Baustein aus der Versionsverwaltung entfernt. Das Einpflegen der so gemachten Änderungen ist dann schwer. Man muss das Projekt mit der Ausgansversion vergleichen und zur Sicherheit alle geänderten Bausteine manuell in die alte Version einpflegen.

3.2 Automatisierbarkeit

Wie schon in den vorherigen Kapitel erläutert, ist es für einen erfolgreichen Einsatz eines Testsystems in einer Entwicklungsumgebung nötig, dass dieses System automatisch bedient werden kann.

3.2.1 Makros

Die Entwicklungsumgebung erlaubt es in einem Untermenü Makros anzulegen. Diese Makros sind Kombinationen aus vordefinierten Befehlen, die in einer bestimmten Rei-

[1] Dabei wird für die eigentliche Ablage der Bausteine ein Visual SourceSafe Server als Backend benutzt. Allerdings werden nicht alle Funktionen von diesem SourceSafe Server an den Benutzer der ENI-Clienten weiter gegeben.

henfolge ausgeführt werden könnten.

Dafür existieren verschiedene Befehle. Einerseits gibt es Befehle, die das Verhalten der Entwicklungsumgebung an sich beeinflussen. Es kann zum Beispiel eine automatische Bestätigung oder Ablehnung von Dialogfenstern aktivieren. Außerdem kann man natürlich die Projekte selber beeinflussen. Dazu ist es natürlich erst einmal möglich die Projekte zu öffnen und zu speichern. Weiterhin ist auch das Exportieren und Importieren von Bausteinen möglich. Und als eine dritte Möglichkeit stehen Befehle zur Verfügung die zur Kommunikation mit der Steuerung dienen. Es kann dabei eine Verbindung zu Steuerung aufgebaut werden, Programme übertragen und ausgeführt werden oder Variablen ausgelesen und geschrieben werden.

Ein großer Nachteil dieser Makrobefehle ist es, dass es keine Möglichkeit gibt, auf ein Ereignis in der Steuerung zu warten und dann auf eine bestimmte Art und Weise zu behandeln. Die einzige Möglichkeit die lineare Ausführung zu unterbrechen, ist der so genannte `delay <duration>` Befehl. Dieser Befehl wartet immer die angegebenen Anzahl an Millisekunden und führt dann die Befehle nach diesem Befehl aus.

Wie schon bemerkt, kann man mit den Makros leider nicht auf Ereignisse in der Steuerung reagieren. Für eine eventuelle Testautomatisierung müsste demzufolge der Test mindestens so lange warten, so lange die längste Ausführung der Tests dauern könnte, bevor der Status ausgelesen werden könnte und in eine Datei geschrieben werden könnte.

Die Makros nur manuell ausführbar. Durch einen klick auf den entsprechenden Menüeintrag (der nach der Erstellung eine Makros angezeigt wird) werden die im Makro hinterlegten Befehle ausgeführt.

Im Hause Weber Maschinenbau werden Makros im Moment vorwiegend dafür eingesetzt um das Projekt beziehungsweise einzelne Bausteine in externe Dateien exportiert werden. Dies dient teilweise zur Archivierung auf einem SVN-Server. Allerdings sind ähnliche Schritte für die Erstellung einer Bibliothek nützlich.

3.2.2 CMD-Befehle

Die CMD-Befehle sind eng mit der Makrosprache verbunden. Die gleichen Befehle aus den Makros können beim Start als Kommandozeilenparameter angegeben werden. Diese Befehle werden dann nach dem Starten direkt ausgeführt.

Aber die prinzipiellen Probleme bleiben. Es kann damit nicht direkt auf Ereignisse reagiert werden und nachdem die Befehle einmal an EPAS-4 gesendet wurden, kann man keine Beeinflussung mehr vornehmen. Ein weiteres Problem ist, dass einige Funktionen nicht durch dieses Schnittstelle ausgeführt werden können. (Beispielsweise Aktionen in der Visualisierung des Steuerungsprogramme. Siehe dazu auch Abschnitt 3.3.)

3.2.3 TypeLibrary

Eine *Type Library* ist ein Teil der COM (Component Object Model) von Microsoft und dient zur Interprozesskommunikation. Das COM stellt unter Windows ein System dar um unabhängig von der Programmiersprache Funktionen in einer Anwendung oder eine DLL auszuführen und über eine objektorientierte Schnittstelle auf die Anwendungen zuzugreifen.[Cor10]

Eine solche Schnittstelle ist zum Beispiel von den Office-Produkten von Microsoft bekannt. Dort kann man aus einer anderen Programmiersprache beispielsweise eine Exel-Arbeitsmappe erstellen und die einzelnen Zellen bearbeiten.

Würde eine solche Schnittstelle für EPAS-4 existieren, könnte man vermutlich alle Funktionen aus einem eigenen Programm heraus aufrufen und könnte damit die Beschränkungen des Makrosprache umgehen.

Epas-4.tbl

Bei genauerer Betrachtung des Programmverzeichnisses fällt auf, dass es tatsächlich eine TypeLibrary gibt. Schaut man sich aber den Inhalt der binären `Epas-4.tbl` mit einem kostenlosen Hilfsprogramm aus dem Windows ResourceKit von Microsoft an, stellt man fest, dass diese TypeLibrary von CoDeSys stammt. Sogar alle GUIDs (globally unique identifier) der einzelnen Klassen stimmen mit denen von CoDeSys überein.

Probleme

Daraus ergeben sich mehrere Probleme. Einerseits ist es nicht einfach möglich (falls mehreren Produkte installiert sind, die auf CoDeSys 2.x basieren) zu entscheiden, welches Programm gestartet werden soll. Dies könnte man realisieren in dem man den Verweis von der GUID in der Registry von Windows auf eine andere Datei verweisen lässt. Allerdings sind alle Änderungen in der Registry immer etwas gefährlich. Bei einem Versuch dies durchzuführen, ist die Installation von EPAS-4 in der Art beschädigt wurden, dass keine Projekte mehr durch Doppelklick auf die Projektdatei geöffnet werden konnten und EPAS-4 in diesem Fall eine Neuinstallation gewünscht hat.

Ein weiteres Problem, dass diese TypeLibrary von CoDeSys stammt, besteht darin, dass die Entwickler von EPAS-4 sich über diese Funktionalität nicht bewusst waren. Mit der TypeLibrary sollten alle Funktionen, die auch mit den Makrobefehlen ausführbar sind, durchgeführt werden können. Leider stößt man beim Testen der Funktionalitäten (eine Dokumentation ist sowohl bei 3s als auch bei Elau nicht vorhanden) immer wieder auf Probleme.

Einige dieser Probleme sind eher subtiler Natur andere fallen schnell auf. Die folgenden beiden Codestücke dienen beiden dazu ein Projekt zu übersetzen:

```
1    // 1
2    IProject.compile();
3
4    // 2
5    IProject.GetBatch().Execute("compile");
```

Führt man die Variante 1 aus, wir das Programm wie erwartet übersetzt. Leider bleibt auch nach Beendigung des Programms, dass die COM-Schnittstelle benutzt hat, die EPAS-4 Anwendung, die im Hintergrund gestartet wurde, laufen und wird nicht beendet. Nach einigen Programmausführungen hab man dann mehrere eigentlich beendete EPAS-4 Prozesse im Hintergrund, die man nur noch mit einem Taskmanager beenden kann. Führt man nun allerdings Variante 2 aus dem obigen Quelltext aus, wird ebenfalls das Projekt übersetzt und die Anwendung beendet sich auch wieder sauber.

Scheinbar sind sich die Entwickler über dieses Problem bewusst. In einer C#-Erweiterung für die nächste Version der Entwicklungsumgebung (zur Konvertierung von alten Projekten) kann man beispielsweise mit dem *Reflector*[2] Quellcode finden, der dazu dient dieses Verhalten zu reparieren. Bei Benutzung dieser COM-Schnittstelle wird durch einen Vergleich der Prozessliste vor und nach dem Start der COM-Prozess ermittelt und nachdem die COM-Schnittstelle benutzt wurde, wird der Prozess einfach getötet.

Ein etwas auffälliger Fehler tritt auch auf, wenn man versucht einige der Makrobefehle zu benutzen. Die COM-Schnittstelle ermöglicht, wie oben schon gesehen die Ausführung von Makrobefehlen: `IProjekt.GetBatch().Execute(cmd);`. Leider funktionieren nur einige Makrobefehle. Solche zum Einloggen auf die Steuerung, zum Übertragen des fertigen Projektes, zum Auslesen oder Schreiben von Variablen werden entweder mit einer Fehlermeldung abgelehnt oder haben einfach keinen Effekt. Damit ist auch mit dieser Methode leider keine sinnvolle Automatisierung von Tests möglich.

ProjectBuilder

Der ProjectBuilder ist eine Beispielanwendung in C# für welche die TypeLibrary verwendet wird. Es erwartet als Kommandozeilenargument eine EPAS-4 Projektdatei (die nicht mit dem ENI-Server verbunden ist[3]), öffnet dann das Projekt, startet die Übersetzung und gibt am Ende die Fehler aus. Der Quellcode ist im Anhang unter A.1 zu finden. Im Quellcode ist ein weiteres Problem zu sehen. Die Ausgabe von EPAS-4 ist nicht direkt zugänglich sondern wird nur ein eine Datei geschrieben. Dort wird die Ausgabe dann in einem zweiten im Hintergrund laufenden Thread ausgelesen und ausgewertet.

[2] Der Reflector ist ein Programm, dass übersetzte Assemblies der DotNet-Plattform wieder in ihren Sourcecode zurück übersetzen kann.

[3] Wird der ENI-Server verwendet, werden leider bei jeder Benutzung der TypeLibrary die Benutzerdaten für den ENI-Server abgefragt. Das sich öffnende Dialogfeld, kann leider nicht unterdrückt werden und stört die automatische Ausführung.

3.2.4 Simulation von Mauseingaben

Die einfachste Möglichkeit ein fremdes Programm zu steuern, ist über die Simulation von Tastendrücken und Mausklicks. Diese Möglichkeit wird im Hause Weber Maschinenbau schon eingesetzt. Ursprünglich wurde das Programm AutoIt[4] verwendet um bestimmte oft benötigte Aufgaben zu automatisieren. In Anlehnung an dieses Tool wurde in jüngster Vergangenheit ein neues Werkzeug in C# geschrieben, dass genau wie AutoIt die Menüs von EPAS-4 mittels der Tastenkombinationen steuert.

Dieser pragmatische Ansatz hat zwar den Vorteil komplett von funktionierenden Schnittstellen unabhängig zu bleiben, allerdings bringt der Ansatz auch Nachteile mit sich. Einerseits müssen alle Installationen, die mit diesem Werkzeug bedient werden sollen, ähnliche Einstellungen haben. Beispielsweise werden auch Makros ausgeführt. Da das Makro-Menü aber davon abhängig ist, wie der Benutzer die Makros eingerichtet hat, ist es leider nur sehr schwer möglich eine allgemeine Tastensteuerung dafür zu finden. Ein weiterer Nachteil dieses Ansatzes ist auch, dass man keinen Zugriff auf die Ausgaben von EPAS-4 hat. Man kann damit beispielsweise nicht überprüfen ob das Projekt fehlerfrei übersetzbar ist und im Fehlerfall werden die Tastaturkommandos einfach weiter ausgeführt.

3.3 Testumgebung: PacUnitTest

Die Testumgebung in EPAS-4 ist nachträglich hinzugefügt wurden. Es besteht aus einer Bibliothek von Elau und einer Datei mit Quellcode zum Importieren in das eigene Projekt.

3.3.1 Funktionen

Das PacUnitTest-System ist im Funktionsumfang vergleichbar mit ähnlichen Unittestumgebungen im Umfeld von Hochsprachen. Man kann mehrere Testsuiten anlegen und in jeder von diesen können mehrere Testfälle sein. Jeder Testfall kann entweder erfolgreich beendet werden oder einen Fehler zurück liefern. PacUnit hat aber keine Funktionen um hängende Tests zu erkennen.

Die Testbibliothek enthält eine grafische Visualisierung, welche die komplette Steuerung des Testsystems erlaubt. Es können Test ausgewählt werden, die ausgeführt werden sollen, die Tests dann durchgeführt werden, nähere Informationen zum Testverlauf angezeigt werden und der Export der Ergebnisdatei ausgelöst werden (siehe Abbildung 3.2).

Die Ergebnisdatei ist ähnlich wie bei JUnit eine XML-Datei. Es wird ein XSL-Stylesheet zur Verfügung gestellt, damit man sich das Ergebnis im Browser anschauen kann. Außer

[4] http://www.autoitscript.com/autoit3/index.shtml

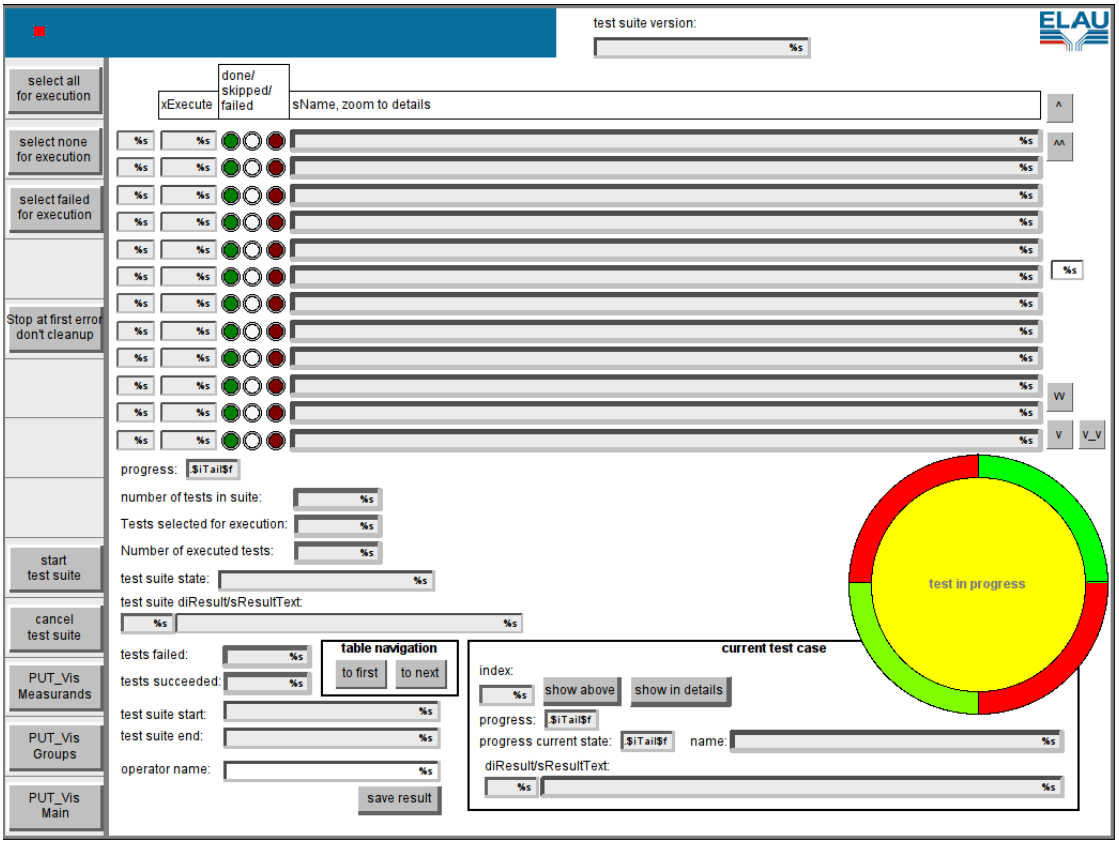

Abbildung 3.2: Visualisierung zur Steuerung der Tests. Oben werden die verschiedenen Testfälle und ihr Erfolg oder Misserfolg aufgelistet und unten befinden sich Buttons zur Steuerung.

in der Ergebnisdatei kann man auch das Ergebnis jedes einzelnen Test in der Visualisierung der Testbibliothek ansehen.

3.3.2 Programmierung von Testfällen

Um einen Testfall anzulegen importiert man zu erst die Vorlage aus der vorhandenen Datei. Damit wir ein Beispieltest angelegt. Dieser `PUT_FB_PUTEST_ExampleTest` wird nun kopiert und an die eigenen Bedürfnisse angepasst. Dafür wird dem Baustein ein neuer zu den geplanten Tests passender Name vergeben und die Beschreibung in der Deklaration im Baustein angepasst.

Damit die Testfälle, die in diesem Baustein entstehen sollen, auch ausgeführt werden, muss der Baustein beim Testsystem registriert werden. Dazu wird eine Instanz von dem gerade erzeugen Test angelegt und an eine spezielle Funktion der Bibliothek übergeben.

Da `PUT_FC_RegisterTestCase` noch mehr Parameter erfordert, ist er hilfreich einen bereits vorhandenen Aufruf im Baustein `SR_PacUnitTestCases` zu kopieren.

Nach dem die Registrierung vollendet ist, können nun die eigentlichen Tests erzeugt werden. Dazu wurde der Prozess schon in mehrere Abschnitte eingeteilt, in die jeweils eigener Quellcode eingefügt werden kann. Mit der Einteilung in Prepare (wird vor dem Tests ausgeführt), Execute (hier stehen die eigentlichen Tests), CleanUp (wird nach der erfolgreichen Beendigung der Tests ausgeführt) und Finally (wird immer nach den Tests ausgeführt) lassen sich beliebige Vorbedingungen für einen Test schaffen und nach den Tests wieder aufräumen.

Die eigentlichen Tests sind noch einmal etwas schwieriger zu erstellen:

```
1  IF (something bad happened) THEN
2          etState         := PUT_GE_TCS_ERROR;
3          diResult        := -10  (* select an appropreate result <= -10 *)
4          sResultText     := 'something bad happened';
5          RETURN;
6  END_IF
```

Die eigentlichen Tests lassen sich durch eine solche if-Abfrage implementieren. Wenn man einen Fehler feststellt muss man 3 Variablen zuweisen. Man sieht, dass man den Status des Tests manuell in den Fehlerstatus setzen muss. Weiterhin muss man das Ergebnis des Tests auf einen eindeutigen Wert kleiner gleich -10 setzen. Damit wird beim Auftreten des Fehlernachricht dem richtigen Testfall zugeordnet. Der ResultText ist freiwählbar und wird bei fehlschlagen des Tests sowohl in der Visualisierung als auch in der XML-Datei angezeigt. Die letzte Aktion ist Return damit die weiter Ausführung der Tests abgebrochen wird.

Leider gibt es keine zur Funktion, die ähnlich einfach wie `assertTrue()` von JUnit verwendet werden kann. Besonders die Bedingung, dass die Ergebnisse der Tests eindeutig sein müssen, ist bei der Entwicklung von Tests relativ aufwendig.

3.4 Automatisierung von Tests

Wie im Abschnitt 2.1 beschrieben, sollte das Ziel sein, die Tests automatisch entweder in jeder Nacht oder sogar noch besser nach jeder Änderung am Sourcecode auszuführen. Daraus ergeben sich mehrere Probleme für die Testumgebung in EPAS-4.

3.4.1 Infrastruktur für die Tests

Durch das binäre Format der Projektdatei, kann man selbst bei der Ausführung der Tests nach jeder neuen Version in der Versionsverwaltung die exakte Änderung nicht mehr nachvollziehen. Dies kann man damit umgehen, dass bei jeder Änderung die lesbare Exportdatei mit in die Versionsverwaltung eingepflegt wird. Aber auch dies stellt wieder Anforderungen an den Benutzer.

Weiterhin gibt es keine Möglichkeit die Steuerungsprogramme virtuell ohne die entsprechende Steuerung auszuführen. Der Computer, der die Tests ausführen soll, benötigt also dauerhaft Zugriff auf eine entsprechende Steuerung. Da diese vermutlich über das normale Firmennetzwerk mit dem Integrationsserver kommunizieren muss (diese Steuerung hat ja in den seltensten Fällen im Serverraum platz), müssen entsprechende Vorkehrungen getroffen werden, dass der Server keine andere Steuerung zufällig erreichen kann. Bei Tests können durchaus physikalische Ausgänge gesetzt werden und so vielleicht ungewollt eine Maschine in der Firma gesteuert werden. Diese Sicherung, dass keine andere Steuerung erreichbar ist, kann entweder durch eine logische Aufteilung des Netzwerks in kleinere virtuelle Netzwerke (VLAN) geschehen oder durch eine komplette Trennung, so dass der Server nur noch über ein VPN auf die Steuerung zugreifen kann.

Eine letzte Schwierigkeit für den Integrationsserver ist die Verteilung der benötigten Bibliotheken. Diese werden von EPAS-4 anhand der Änderungszeit identifiziert. Deshalb können benötigte Bibliotheken nicht einfach in einer Versionsverwaltung gespeichert werden. Die meisten der heute verwendeten Versionverwaltungssysteme können leider nicht die Änderungszeit der Datei speichern. Daher würde dann bei jedem Start des automatischen Testwerkzeuges mehrere Warnungen erscheinen, dass die Version der Bibliotheken nicht übereinstimmt.

3.4.2 Ausführung der Tests

Das Hauptproblem bei der Automatisierung der Tests ist aber eindeutig die Ausführung selber. Nach umfangreichen Kontakt mit dem Hersteller, der nach seinen Angaben ein Tool zur automatischen Ausführung von Tests bei der Entwicklung von Bibliotheken selber einsetzt, könnte uns aber keine Version von diesem Tool zur Verfügung gestellt werden. Als grobe Schätzung wurde lediglich versprochen, dass in etwa einem Jahr ein solches Tool verfügbar wäre, nachdem es alle Test und Dokumentationsabteilungen durchlaufen hat.

Da die Steuerung der Tests in der Entwicklungsumgebung durch eine Visualisierung erfolgt (also einen grafischen Bedienelement) kann man die Ausführung leider nicht mit dem CMD-Befehlen auslösen. Als einziger Ausweg die Tests automatisch auszuführen wurde vom Hersteller der Entwicklungsumgebung eine eigene Netzwerkschnittstelle auf der Steuerung selber empfohlen.

Schon die Tests von der Steuerung auszuführen ist allerdings ein Problem. Die entsprechenden Variablen sind nur zum Lesen freigegeben. Aber diese Beschränkung kann man mit einem Zeiger auf diese Variablen umgehen:

```
1    pxTmp : POINTER TO BOOL;
2
3    pxTmp := ADR(PUT_SR_PacUnitTest.xStart);
4    pxTmp^ := TRUE;
```

Obwohl `PUT_SR_PacUnitTest.xStart` als interne Variable deklariert ist, kann man mit dem Umweg über einen Zeiger auf diese Variable schreiben.

Mit eine Bibliothek direkt von Elau ist es relativ einfach einen Netzwerkserver zu programmieren. Mit Hilfe von nur 3 Funktionen ist der Empfang und das Senden von Nachrichten möglich. Der vollständige Quellcode für einen Netzwerkserver ist im Anhang unter A.2 zu finden. Der Server macht es möglich über ein einfaches textbasiertes Protokoll Tests zu starten und über den Erfolg oder Misserfolg zu erfahren.

Bisher sind nur 3 Befehle implementiert. `start` startet die Testausführung. Sobald die Tests beendet sind, wird eine Nachricht mit der Anzahl der erfolgreichen und fehlgeschlagenen Tests zurückgeschickt.

Ein weitere Befehl `notify` dient dazu, dass ein Client nach Testausführung benachrichtigen lassen kann. Dieser Client erhält solange Benachrichtigungen, wie er die Verbindung aufrecht erhält.

Der letzte Befehl ist `quit` und danach wird die Verbindung vom Server getrennt und alle Benachrichtigungen für den Client zurückgesetzt.

Nach der Ausführung wird auch gleich die entsprechende Ergebnisdatei auf der Steuerung gespeichert. Diese ist dann mit FTP[5] abrufbar und kann mit der entsprechenden XSL-Datei verarbeitet werden.

3.5 Ausblick

Nun existiert eine Möglichkeit von entfernt die Tests auf einer Steuerung auszuführen und das Ergebnis abzufragen. Damit jetzt produktiv automatisiert Tests auf der Steuerung ausgeführt werden können müssten noch die anderen Probleme der Infrastruktur beseitigt werden. Dann sollte es relativ einfach sein, einen Client zu schreiben, der bei einer Änderung in der Versionverwaltung den TestController mit dem Netzwerkserver in das Projekt einfügt, dann das Projekt auf die Steuerung spielt und dann die Tests ausführt.

Allerdings bleibt festzuhalten, dass von der Entwicklungsumgebung keine Unterstützung für Tests gegeben wird und die automatische Ausführung von Tests nur mit einigen Tricks möglich ist.

[5] Das FTP wird durch das Betriebssystem auf der Steuerung bereitgestellt. Bei diesem vxWorks sieht die Auflistung des Verzeichnisinhaltes anders aus und darum haben leider einige FTP-Clients Probleme.

4 SoMashine Motion

SoMashine Motion[1] ist die nächste Version der Entwicklungsumgebung von Elau. Elau wurde im Jahre 2008 ein Unternehmen von Schneider Electric. Seitdem wird sowohl am Standort von Elau in Marktheidenfeld in Deutschland als auch am Standort von Schneider Electric in Frankreich je eine Entwicklungsumgebung auf Basis von CoDeSys 3, dem Nachfolger von CoDeSys 2.4, entwickelt.

Außer der Entwicklungsumgebung wird auch noch die Workbench mitgeliefert. Diese Oberfläche bündelt alle Hilfsmittel und die neue und alte Entwicklungsumgebung und erlaubt das Wechseln zwischen den Anwendungen um ein Projekt mit mehreren Tools zu bearbeiten[2] (siehe Abbildung 4.1).

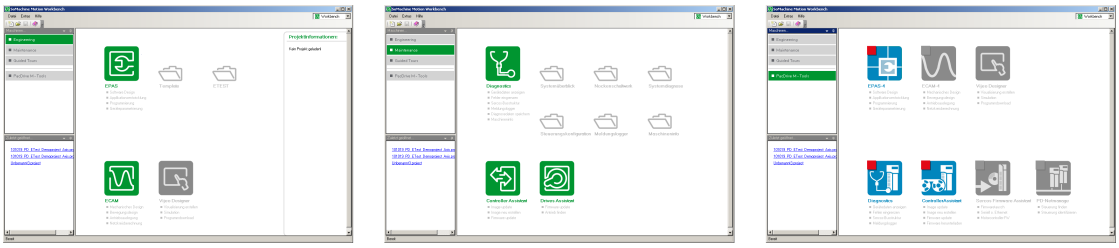

Abbildung 4.1: EPAS-5 Workbench zum Starten der Entwicklungsumgebung, allen Hilfsprogrammen und sogar der alten Entwicklungsumgebung

4.1 CoDeSys 3

4.1.1 Software

CoDeSys 3 ist das Nachfolger der erfolgreichen CoDeSys 2.4 Plattform. Die Version 3 basiert auf dem Interface von Visual Studio 2008. Die Oberfläche sieht damit wesentlich modernen aus und unterstützt automatisch viele Funktionen die mittlerweile bei Entwicklungsumgebungen selbst verständlich sind. Zum Beispiel können die verschiedenen

[1] SoMashine Motion ist das komplette Softwarepaket. Die Entwicklungsumgebung einzeln wird EPAS-5 genannt.

[2] Es kann zum Beispiel nötig sein, bestimmte Parameter der Motoren mit einem extra Werkzeug zu berechnen. Die Werte können dann einfach durch das Wechseln zur eigentlichen Entwicklungsumgebung in das Projekt übernommen werden.

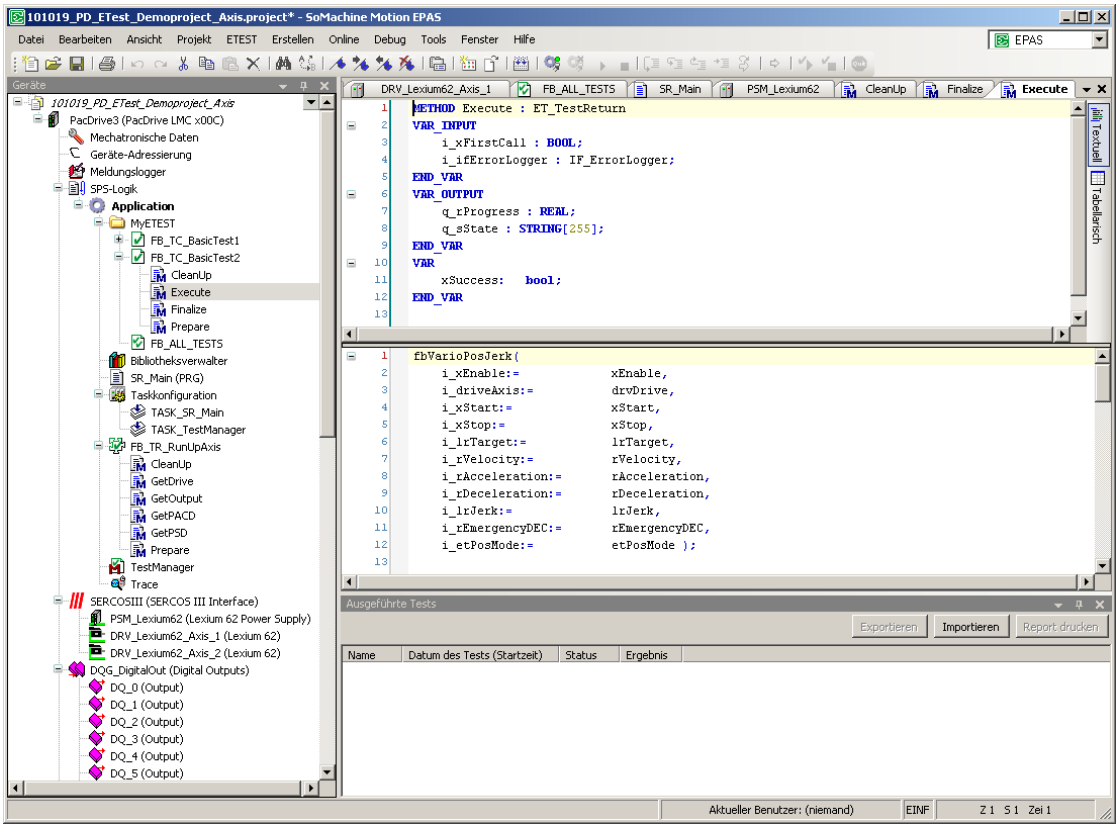

Abbildung 4.2: Oberfläche von EPAS-5 auf Basis von Visual Studio 2008

Fenster der Entwicklungsumgebung beliebig gruppiert und angeordnet werden. Im Gegensatz zu dem MDI[3] von CoDeSys 2.4 werden außerdem verschiedene Quellcodes in verschiedenen Reitern geöffnet. Außerdem funktioniert die automatische Vervollständigung in einer angemessenen Geschwindigkeit.

CoDeSys 3 unterstützt neue Spracherweiterungen für die IEC-61131-3 Programmiersprache. Diese objektorientierten Erweiterungen erlauben das erste Mal eine Programmierung mit Interfaces und Methoden und Vererbung ähnlich den objektorientierten Hochsprachen (siehe `METHOD` in Abbildung 4.2). Allerdings werden nicht alle Funktionen unterstützt. Beispielsweise können nur Interfaces vererbt werden und keine Klassen. Damit sind in der Vererbungshierarchie keine Implementierungen möglich.

Die CoDeSys 3 Plattform ist wesentlich offener als der Vorgänger. Durch die komplette Programmierung in DotNet und die Möglichkeit an fast allen Stellen mit Erweiterungen einzugreifen, können wesentlich mehr Dinge von Drittanbietern geleistet werden.

In CoDeSys 3 wird weiterhin auf dem Echtzeitsystem programmiert und die Funktionen

[3] Multi-Dokument-Interface: mehrere kleine Fenster in dem Hauptfenster der Anwendung

werden zyklisch aufgerufen.

4.1.2 Hardware

Schon seit 3 Jahren werden auf Messen regelmäßig CoDeSys 3 Testversionen von 3s verteilt. Allerdings gibt es noch keine richtige Version. Dies kann unter anderem damit zusammen hängen, dass mit dem Umstieg auf CoDeSys 3 auch ein Umstieg auf eine neue Hardware vollzogen werden muss. Außerdem können bestehende Projekte nur teilweise auf die neue Plattform konvertiert werden.

Hinzukommt, dass CoDeSys 3 einen neuen Bus zur Kommunikation zwischen Steuerung und den Motoren vorsieht. Bei CoDeSys 2.4 wurde bisher *SERCOS 2* für diesen Zweck verwendet (zwei Lichtwellenleiter). Mit CoDeSys 3 und *SERCOS 3* (CAT7 Ethernetkabel) werden damit auch neue Regler für die Motoren notwendig.

Schneider Electric kann mittlerweile lauffähige Steuerungen und Servoregler sowie die dazugehörigen Motoren liefern. Auch wenn die Software sich noch im Teststadium befindet können damit schon Programme entwickelt werden. Außerdem kann eventuelle noch Einfluss auf die weitere Entwicklung der Umgebung ausgeübt werden.

4.2 Versionsverwaltung

Auch mit SoMashine Motion ist die Versionsverwaltung nicht einfach. die Projekte werden in komprimierten ZIP-Archiven gespeichert, wobei die eigentlichen Dateien darin erneut verschlüsselt oder komprimiert sind. Damit können die einzelnen Bausteine des Projektes wieder nur nach einem Export in einer Versionsverwaltung abgelegt werden.

Intern ist aktuell ebenfalls keine Versionierung möglich. Der ENI-Server ist veraltet und wird entfernt und die neuen Erweiterungen für Subversion oder anderen Systeme sind noch in Entwicklung und noch nicht verfügbar.

Zusätzlich wird die Verwaltung von Bibliotheken erschwert. Schon früher konnten diese nicht einfach in der Versionsverwaltung abgelegt werden (Problem mit dem Änderungsdatum der Datei). In der neuen Version müssen alle verwendeten Bibliotheken in einer Verwaltung installiert werden (siehe Abbildung 4.3). Nach einem automatischen Abrufen aus einer Versionsverwaltung sind die Bibliotheken also unbekannt und noch nicht zu verwenden. Das Änderungsdatum spielt dann keine Rolle mehr, aber die Bibliotheken können zur Zeit nicht automatisch installiert werden.

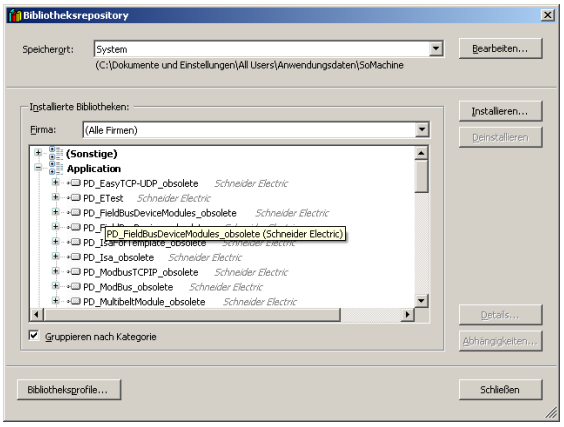

Abbildung 4.3: Zentrale Verwaltung von Bibliotheken

4.3 Unittests in EPAS-5

4.3.1 ETest

Im Gegensatz zu *EPAS-4* hat sich *EPAS-5* auf dem Gebiet der Tests wesentlich weiter entwickelt. In *EPAS-4* wurden die Tests noch als Bibliothek realisiert und liefen komplett auf der Steuerung ab. In *EPAS-5* wurden in die Entwicklungsumgebung die Erweiterung ETest eingebaut. Damit kann man in der Entwicklungsumgebung Testfälle anlegen (man muss keine Bausteine mehr importieren und dann kopieren). In diesen Testfällen sind dann die einzelnen Status (genau wie in der letzten Version: Prepare, Execute, Cleanup und Finally) schon als Methoden angelegt.

Die Testfälle an sich werden ähnlich programmiert, wie die PUT-Testfälle in der Vorgängerversion. Allerdings gibt es mittlerweile eine `Assert`-Funktion die es ermöglicht einen Fehler bei einer falschen Bedingung auszugeben. Dabei müssen dann keine Nummern für die Fehler vergeben werden sondern es kann einfach eine Fehlermeldung übergeben werden.

Außerdem kann man weiter Bausteine anlegen die mit einem Prepare und einem Cleanup benötigte Ressourcen für einen Test initialisieren und das Programm in einen definierten Zustand bringen. Diese Bausteine kann man dann mit einzelnen Testfällen verknüpfen, so dass diese immer vor und nach dem Test automatisch aufgerufen werden.

Auch die Ausführung der Tests erfolgt nicht mehr über eine Visualisierung auf der Steuerung. Die Testfälle können in der Entwicklungsumgebung zu Testreihen zusammengefasst werden und dann ausgeführt werden. Man sieht dann sofort den Erfolg oder Misserfolg der Test und kann eine HTML-Datei exportieren.

4.3.2 Automatisierung

Durch die starke Integration in die Entwicklungsumgebung sind die Tests leider sehr schwer automatisierbar. Die Möglichkeit eine Netzwerkkommunikation zu schreiben und dann die Tests von der Steuerung aus zu starten und auszuwerten ist leider unmöglich.

Außerdem ist leider die Automatisierung über die bekannten Schnittstellen entfernt wurden. Es gibt keine TypeLibrary mehr und auch die CMD-Schnittstelle wurde entfernt.

In dem Quellcode diverser Teile der Entwicklung wird eine Batch-Schnittstelle angelegt und Befehle zu dieser hinzugefügt. Leider kann auf diese Schnittstelle noch nicht zugegriffen werden. Da sich CoDeSys 3 immer noch in der Entwicklung befindet und ständig durch Kundenwünsche um neue Funktionen erweitert wird, sind leider noch nicht alle vorbereiteten Funktionen auch benutzbar.

Aktuell bestünde nur die Möglichkeit eine eigene Erweiterung für die Entwicklungsumgebung zu schreiben und in dieser Erweiterung ein Interface zur Steuerung für eine externe Anwendung zu schaffen. Dies wäre allerdings mit sehr viel Aufwand verbunden. Zumindest für Endkunden ist leider die API von CoDeSys 3 nicht dokumentiert. Mit Hilfe des Reflectors kann man zwar den Quelltext vieler Teile der Entwicklungsumgebung wiederherstellen, aber der Aufwand aus unkommentierten Quelltext eine API herauszufinden ist trotzdem noch hoch. Außerdem sind in einigen Teilen sogar die Variablen und Funktionsnamen unkenntlich gemacht wurden.

Laut Aussage von Elau ist bei ihnen intern eine Schnittstelle vorhanden. Diese ist allerdings nicht für Endkunden verfügbar und bereitet immer wieder Probleme, da sich die Schnittstellen der Entwicklungsumgebung ständig ändern. Außerdem wird diese Schnittstelle nicht mehr weiterentwickelt, sondern nur noch an die Veränderungen der Entwicklungsumgebung angepasst.

Solange also die Schnittstelle von der Firma 3s selber in der Entwicklungsumgebung nicht funktioniert und nicht dokumentiert ist, besteht keine Möglichkeit automatisiert Tests auszuführen.

5 TwinCAT 3

TwinCAT 3 ist die neue Version der Entwicklungsumgebung von der Firma Beckhoff. Die Firma Beckhoff verfolgt einen anderen Ansatz als viele andere Hersteller für Industriesteuerungen. Bei Beckhoff gibt es keine separate Hardwaresteuerung mehr. Alle Aufgaben zum Steuern und Regeln übernimmt eine Softwaresteuerung (Soft-SPS) auf einem Industrie-PC unter WindowsXP embedded. Dazu werden die Ein- und Ausgänge die sonst für eine Steuerung typisch sind, über verschiedene externe Bussysteme angeschlossen.

Das Industrie-PC Portfolio der Firma reicht dabei von kleinen Hutschienen-PCs mit einer Intel Atom CPU mit einem Rechenkern bis zu großen PCs für den Schaltschrank mit Dual-Core CPU. Dabei ist keine Standard-PC Hardware im Einsatz, sondern sämtliche Motherboards sind an die entsprechende Umgebung angepasst.

5.1 Aktuelle Situation

Die aktuellen Version der Entwicklungsumgebung TwinCAT 2.x basiert wie EPAS-4 auf der CoDeSys-2.x Software der Firma 3s. Im unterschied zu EPAS-4 war hier allerdings die komplette Entwicklung darauf auf die Soft-SPS abgestimmt. Man benötigte keine zum Testen von Programmen keine zusätzliche Hardware. Denn auf dem Entwicklungsrechner lief die gleiche Umgebung wie auf den *Steuerungen* von Beckhoff.

Da TwinCAT 2 auf der gleichen Software wie EPAS-4 aufsetzt gelten hier die gleichen Nachteile. Die Entwicklungsumgebung hat einen sehr eingeschränkten Editor, der Quellcode wird binär in der Projektdatei abgelegt und kann daher sehr eingeschränkt mit einem alternativen System versioniert werden und die Unterstützung für Tests ist mangelhaft.

5.2 Kombinierte Entwicklung in der nächsten Version

Die neue Version TwinCAT 3 ist im Gegensatz zur Vorgängerversion eine komplette Neuentwicklung. Dabei ist Beckhoff einen anderen Schritt gegangen als andere Hersteller für Steuerungen. TwinCAT 3 basiert nicht mehr direkt auf CoDeSys. Die Basis für die Entwicklungsumgebung bildet Visual Studio 2010 von Microsoft. Darin enthalten ist eine Erweiterung für IEC-61131-3 Quellcode. Die Philosophie von Beckhoff ermöglicht

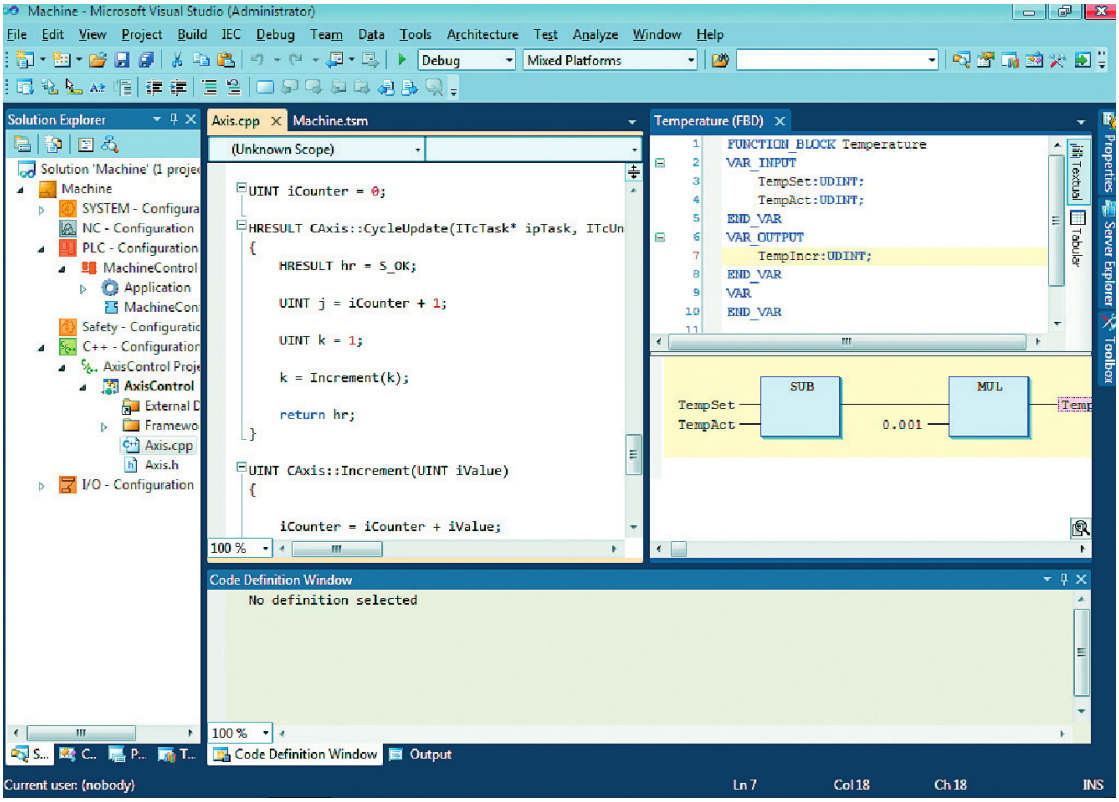

Abbildung 5.1: Oberfläche von TwinCAT 3 auf Basis von Visual Studio 2010

es das Steuerungsprogramm und die Visualisierung zur Bedienung auf einem PC laufen zu lassen. Dabei ist es möglich speziellen Programmteilen bestimmte CPU-Kerne zuzuweisen und somit die Echtzeitfähigkeit der Steuerung zu garantieren.

Außerdem ist es möglich die Steuerung von Beckhoff nicht nur nach IEC-61131-3 zu programmieren, sondern auch mit C/C++ (siehe Abbildung 5.1). In TwinCAT selber kann man sogar mit alle Sprachen die von Visual Studio unterstützt werden programmieren. Die Echtzeitfähigkeit ist allerdings nur für IEC-61131-3 und C/C++ garantiert. Die anderen Sprachen der DotNet-Plattform wie C# oder VisualBasic können für das Bedieninterface verwendet werden, bei dem keine Echtzeitfähigkeit gefordert wird.

Das System TwinCat 3 wurde schon Ende 2009 auf der SPS/IPC/Drives[1] vorgestellt. Auch dieses Jahr auf der SPS/IPC/Drives gab es wieder Präsentationen. Allerdings konnte Beckhoff bisher kein Testsystem zur Verfügung stellen. Deshalb müssen die Betrachtungen hier leider auf der theoretischen Ebene bleiben.

[1] Messe für Automatisierung in Nürnberg

5.2.1 Vorteile

Die Vorteile des TwinCAT 3 Systems liegen auf der Hand. Durch die Entwicklung von Steuerungsprogramm und Visualisierung in einem Projekt mit einer Entwicklungsumgebung ist es einfacher konsistente Versionen zu erzeugen. Wird bei beispielsweise bei dem Programm der Steuerung eine Variable umbenannt, kann das Visual Studio 2010 diese Umbenennung auch in den anderen Programmen vollziehen (Refactoring).

Da außerdem ein komplettes Visual Studio verwendet wird, dass nur mittels Plugins und Bibliotheken für die jeweiligen Sprachen erweitert wird, können auf den kompletten Funktionsumfang von Visual Studio zurückgegriffen werden. Eine Integration einer Versionsverwaltung ist dann kein Problem mehr, da für Visual Studio schon Plugins für diverse Systeme existieren.

Ein weiterer Vorteil der kombinierten Entwicklung von Steuerungsprogramm und Visualisierung in einer Umgebung ist der Datenaustausch zwischen diesen beiden Ebenen. Beckhoff hat den Vorteil gegenüber anderen Herstellern, dass die Visualisierung sich auf dem gleichen Rechner befindet wie die Steuerung. Dadurch ist der Datenaustausch relativ einfach. Musste man bei herkömmlichen Systemen ein Netzwerkprotokoll zum Austausch verwenden und konnte man damit die objektorientierten Eigenschaften nicht verwenden, ist die bei Beckhoff einfach. Beckhoff bietet für die DotNet Plattform Bibliotheken an um auf die Werte in dem Steuerungsprogramm sowohl lesend als auch schreibend drauf zu greifen zu können. Dabei kann die Variable in der Steuerung über das Objekt, dass diese Variable in der Steuerung verwaltet angesprochen werden. Damit ist eine sehr gute Semantik möglich.

Für Tests ergeben sich auch ganz neue Möglichkeiten. Durch die mögliche Verwendung von C/C++ kann man auf Unittest Werkzeuge aus dieser Welt zurückgreifen und ist nicht auf die Werkzeuge von Beckhoff oder sogar 3s angewiesen. Außerdem ist das Testen einfacher, da die Soft-SPS einfach auf dem Testrechner installiert werden kann und für einfache Tests keine weiter Hardware erforderlich ist (für komplexe sind dann eventuell noch Ein- und Ausgänge erforderlich).

Ein weiterer Vorteil stellt die Fähigkeit der Bedienung von Visual Studio über die Kommandozeile dar. Bisher sind alle Funktionen zum Übersetzen der Quellcodes auch von der Kommandozeile aus verfügbar. Es ist zu hoffen dass die Erweiterungen für den IEC-61131-3 Quellcode auch von der Kommandozeile aus zu bedienen sind. Dies würde eine sehr gute Automatisierbarkeit bedeuten.

5.2.2 Nachteile

Auf der anderen Seite steht den Vorteilen des Systems auch ein kleiner Nachteile gegenüber.

Die kombinierte Entwicklung von Visualisierung und Steuerung in einer Umgebung ist darauf ausgelegt, dass die Visualisierung auch wirklich dort entwickelt wird. Damit ist man auf die DotNet Plattform für die Visualisierung festgelegt. Sollte man hingegen eine Visualisierung in Java oder einem ganz anderen System entwickeln wollen (beispielsweise eine Webanwendung), bringen die Bibliotheken von Beckhoff nur wenig. Man müsste dann wieder anfangen eine Netzwerkanbindung dazwischen zu setzen. Allerdings wird Beckhoff weniger Bibliotheken zur Verfügung stellen um direkt auf der Steuerung eine Netzwerkkommunikation zu realisieren. Die einfachste Lösung wäre dann vermutlich eine Anwendung auf der DotNet Plattform zu schreiben, welche die Weiterleitung und Übersetzung der Nachrichten zur Steuerung übernimmt.

5.3 Fazit

Wenn man seine Visualisierung sowieso auf der DotNet Plattform betreibt oder kein Problem mit dem zusätzlichen Aufwand der Entwicklung einer Netzwerkkommunikation hat, ist TwinCAT 3 eine sehr viel versprechende Umgebung die viele Vorteile besitzt und durch die Möglichkeit der Benutzung von C/C++ auch eine gute Umgebung für Unittests bietet.

A Quellcode

A.1 ProjectBuilder

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Threading;
using System.Runtime.InteropServices;
using CoDeSys;

namespace ProjectBuilder
{
    class Program
    {
        static void Main(string[] args)
        {
            if ((args.Length < 1) || (!File.Exists(args[0])))
            {
                Console.WriteLine("Please supply a project file name as first argument.");
                Environment.Exit(0);
            }

            var pb = new ProjectBuilder(args[0]);
            var result = pb.Build(Console.Out);

            Console.WriteLine("Error: {0:D}", pb.ErrorCount);
            Console.WriteLine("Warning: {0:D}", pb.WarningCount);

            foreach (var e in pb.ErrorList) {
                var ausgabe = "{4}, Error: {0}, {1}, {2}, {3}";
                if (String.IsNullOrEmpty(e.Action))
                    ausgabe = "{4}, Error: {0}, {2}, {3}";

                Console.WriteLine(ausgabe, e.Module, e.Action, e.Message, e.Line, e.LineOfOutput);
            }

            Console.Read();

            if (result)
                Environment.Exit(0);
            else
                Environment.Exit(1);
        }
    }
}
```

ProjectBuilder.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Threading;
using CoDeSys;
using System.Text.RegularExpressions;

namespace ProjectBuilder
{
```

```
12      class ProjectBuilder
13      {
14          /// <summary>
15          /// Fehler oder Warnung, die aus der Ausgabe von EPAS-4 ausgelesen wurde
16          /// </summary>
17          public class ErrorItem
18          {
19              /// <summary>
20              /// Modul in dem der Fehler aufgetreten ist
21              /// </summary>
22              public string Module { get; internal set; }
23
24              /// <summary>
25              /// Action in welcher der Fehler aufgetreten ist, gegebenenfalls leer
26              /// </summary>
27              public string Action { get; internal set; }
28
29              /// <summary>
30              /// Zeile des Fehlers
31              /// </summary>
32              public int Line { get; internal set; }
33
34              /// <summary>
35              /// Fehlerbeschreibung wie in der EPAS-4 Ausgabe
36              /// </summary>
37              public string Message { get; internal set; }
38
39              /// <summary>
40              /// Zeilennummer der Ausgabe in welcher der Fehler ausgegeben wurde
41              /// </summary>
42              public int LineOfOutput { get; internal set; }
43          }
44
45          private string project;
46
47          private static Regex messageRegex = null;
48
49          /// <summary>
50          /// Anzahl der Fehler beim Buildvorgang (bei negativen Wert ist der
51          /// Buildvorgang nicht korrekt beendet wurden)
52          /// </summary>
53          public int ErrorCount { get; private set; }
54
55          /// <summary>
56          /// Liste aller Meldungen (Fehler/Warnungen) die beim Buildvorgang ausgegeben wurden
57          /// </summary>
58          public List<ErrorItem> ErrorList { get; private set; }
59
60          /// <summary>
61          /// Anzahl der Warnungen beim Buildvorgang (bei negativen Wert ist der
62          /// Buildvorgang nicht korrekt beendet wurden)
63          /// </summary>
64          public int WarningCount { get; private set; }
65
66          public ProjectBuilder(string project)
67          {
68              this.project = project;
69          }
70
71          /// <summary>
72          /// Statet den Übersetzungsvorgang das Projektes und wertet die Fehlermeldungen aus.
73          /// </summary>
74          /// <param name="output">TextWriter in dem die Statusmeldungen ausgegeben werden
75          /// sollen (optional).</param>
76          /// <returns>Gibt zurück ob das Übersetzungsvorgang erfolgreich beendet
77          /// wurden ist.</returns>
78          public bool Build(TextWriter output = null)
79          {
80              ErrorList = new List<ErrorItem>();
81              ErrorCount = -1;
82              WarningCount = -1;
83
84              // EPAS-4/CoDeSys COM-Objekt erzeugen
85              // Leider benutzt EPAS-4 die gleiche TypeLib wie CoDeSys. Wenn
86              // CoDeSys nach EPAS installiert wurde, statet jetzt leider CoDeSys.
87              var codesys = new CCoDeSys();
88
89              try
90              {
91                  // übergebenes Projekt öffnen
92                  codesys.OpenProject(project);
93              }
94              catch (Exception e)
95              {
```

```
96                  output.WriteLine(e.Message);
97                  return false;
98              }
99
100             // Übersetzung starten
101             var lastline = Build(codesys, output, ErrorList);
102             ParseResult(lastline);
103
104             // Projekt schließen
105             codesys.CloseProject();
106
107             return (ErrorCount == 0);
108         }
109
110         /// <summary>
111         /// Startet den Übersetzungsvorgang für ein Projekt und wertet bei Bedarf die
112         /// Fehlermeldungen aus.
113         /// </summary>
114         /// <param name="codesys">ICoDeSys Objekt in dem das Projekt geladen ist.</param>
115         /// <param name="output">TextWriter in dem die Statusmeldungen ausgegeben
116         /// werden sollen (optional).</param>
117         /// <param name="errorList">Liste in der die Fehlermeldungen eingetragen
118         /// werden sollen (optional).</param>
119         /// <returns>Gibt die letzte Zeile der Ausgabe zurück,
120         /// in der die Anzahl der Fehler und Warnungen ausgegeben wird.</returns>
121         public static string Build(ICoDeSys codesys, TextWriter output = null,
122                                    IList<ErrorItem> errorList = null)
123         {
124             int lineCounter = 0;
125             string lastline = null;
126
127             LogAction(codesys,
128                       c =>
129                           // Projekt übersetzen
130                           BuildProject(c.GetBatch()),
131
132                       line =>
133                       {
134                           lastline = line;
135
136                           if (output != null)
137                               output.WriteLine(line);
138
139                           if (errorList != null)
140                               // eventuelle Fehler auswerten
141                               ParseMessages(line, ++lineCounter, errorList);
142                       });
143
144             return lastline;
145         }
146
147         /// <summary>
148         /// Führt eine CoDeSys Aktion aus und wertet das Log aus. Für jede gelesene Logzeile
149         /// </summary>
150         /// <param name="codesys">ICoDeSys Objekt auf dem die Aktion
151         /// ausgeführt werden soll.</param>
152         /// <param name="action">Aktion die ausgeführt werden soll.
153         /// Bekommt das ICoDeSys Objekt als Parameter.</param>
154         /// <param name="loglineCallback">Callback das für jede Logzeile
155         /// aufgerufen werden soll.</param>
156         private static void LogAction(ICoDeSys codesys, Action<ICoDeSys> action,
157                                       Action<string> loglineCallback)
158         {
159             // temporäre Logdatei erstellen um die Ausgabe von EPAS zu bekommen
160             var logfile = Path.GetTempFileName();
161             codesys.GetBatch().SetLogging(true, logfile);
162
163             // Logdatei auslesen und parallel dazu einen Thread starten
164             // um das Projekt zu übersetzen
165             var t = new Thread(() => action(codesys));
166             t.Start();
167
168             TailLogfile(logfile, t, loglineCallback);
169
170             t.Join();
171
172             // temporäre Logdatei löschen
173             File.Delete(logfile);
174         }
175
176         /// <summary>
177         /// Diese Funktion liest die übergebene Logfile solange der
178         /// Thread t läuft aus und gibt sie auf der Console aus.
179         /// </summary>
```

```
180         /// <param name="logfile">Logdatei die ausgegeben werden soll</param>
181         /// <param name="t">Thread der läuft, solange die Logfile ausgelesen wird.</param>
182         /// <param name="callback"></param>
183         /// <returns>Letzte Zeile der Ausgabe für die eventuelle Auswertung</returns>
184         private static void TailLogfile(string logfile, Thread t, Action<string> callback)
185         {
186             // Logdatei öffnen und EPAS-4 die Möglichkeit lassen
187             // (FileShare.ReadWrite) seine Logeinträge zu schrieben.
188             var stream = new FileStream(logfile, FileMode.OpenOrCreate,
189                                         FileAccess.Read, FileShare.ReadWrite);
190             var reader = new StreamReader(stream);
191
192             string line;
193             // Reihenfolge der Bedingungen ist wichtig: Könnte sonst sein,
194             // dass line nicht initialisiert wird.
195             while ((line = reader.ReadLine()) != null || t.IsAlive)
196             {
197                 if (line != null)
198                 {
199                     callback(line);
200                 }
201                 else
202                     // Wenn es nichts zu lesen gab, 100ms abwarten.
203                     // (Polling einschränken, sonst 100% CPU-Auslastung)
204                     Thread.Sleep(100);
205             }
206
207             reader.Close();
208             stream.Close();
209         }
210
211         /// <summary>
212         /// Wertet die Ausgaben von EPAS-4 aus und liest alle Fehler/Warnungen aus
213         /// </summary>
214         /// <param name="line">aktuelle Zeile der EPAS-4 Ausgabe</param>
215         /// <param name="lineCounter">aktuelle Zeilennummer der EPAS-4 Ausgabe</param>
216         /// <param name="errorList">Liste in die, die eventuellen Fehler/Warnungen
217         /// eingetragen werden sollen</param>
218         private static void ParseMessages(string line, int lineCounter, IList<ErrorItem> errorList)
219         {
220             if (messageRegex == null)
221             {
222                 // bei erster Ausführung den regulären Ausdruck compilieren
223                 // (erhöht die Geschwindigkeit bei wiederholten Aufrufen)
224                 messageRegex = new Regex(@"^(?<Module>[^. ]+)(?:.(?<Action>[^ ]+))? " +
225                                          @"\((?<Line>[0-9]+)\): (?<Message>.*)");
226             }
227
228             var match = messageRegex.Match(line);
229             if (match.Success)
230             {
231                 var error = new ErrorItem();
232                 error.Module = match.Groups["Module"].Value;
233                 error.Action = match.Groups["Action"].Value;
234                 error.Line = int.Parse(match.Groups["Line"].Value);
235                 error.Message = match.Groups["Message"].Value;
236                 error.LineOfOutput = lineCounter;
237                 errorList.Add(error);
238             }
239         }
240
241         /// <summary>
242         /// Wertet die letzte Zeile der EPAS-4 Ausgabe aus und ermittelt, wie
243         /// viele Fehler und Warnungen aufgetreten sind.
244         /// </summary>
245         /// <param name="lastline">letzte Zeile der EPAS-4 Ausgabe</param>
246         private void ParseResult(string lastline)
247         {
248             if (lastline != null)
249             {
250                 // Format der Zeile: <Zahl> Fehler, <Zahl> Warnung(en)
251                 var m = Regex.Match(lastline, @"^(\d) .*, (\d)");
252                 if (m.Success)
253                 {
254                     ErrorCount = int.Parse(m.Groups[1].ToString());
255                     WarningCount = int.Parse(m.Groups[2].ToString());
256                 }
257             }
258         }
259
260         /// <summary>
261         /// Wertet die letzte Zeile der EPAS-4 Ausgabe aus und gibt die Fehler anzahl zurück.
262         /// </summary>
263         /// <param name="lastline">letzte Zeile der EPAS-4 Ausgabe</param>
```

```
264         /// <returns>Anzahl der Fehler</returns>
265         private static int ParseErrors(string lastline)
266         {
267             if (lastline != null)
268             {
269                 // Format der Zeile: <Zahl> Fehler, <Zahl> Warnung(en)
270                 var m = Regex.Match(lastline, @"^(\d) .*, (?:\d)");
271                 if (m.Success)
272                 {
273                     return int.Parse(m.Groups[1].ToString());
274                 }
275             }
276
277             throw new ArgumentException();
278         }
279
280         /// <summary>
281         /// Startet den vollständigen Übersetzungsvorgang eines EPAS-4
282         /// Projektes. Dafür wird erst eine Bereinigung ausgeführt und
283         /// danach der Übersetzungsvorgang angestoßen.
284         /// </summary>
285         /// <param name="batch">IBatch Objekt vom geladenen Projekt.</param>
286         private static void BuildProject(IBatch batch)
287         {
288             try
289             {
290                 // Hier werden nicht die Shortcuts Clean() und Build() von IProject genutzt,
291                 // da bei deren Verwendung nach Programmende EPAS-4.exe nicht beendet wird.
292                 batch.Execute("project clean");
293                 batch.Execute("project build");
294             }
295             catch { }
296         }
297     }
298 }
```

A.2 Network PutExecuter

PUT_TestManagementCom

```
1   PROGRAM PUT_TestManagementCom
2   VAR
3           pxTmp                   : POINTER TO BOOL;
4           psTmp                   : POINTER TO STRING[40];
5           State                   : (sIdle, sStarting, sRunning, sSaving) := sIdle;
6           Timer                   : TON;
7   END_VAR
8
9   VAR_INPUT
10          Start                   : BOOL := FALSE;
11  END_VAR
12
13  VAR_OUTPUT
14          TestDone                : BOOL := FALSE;
15          TestSuccess             : BOOL := FALSE;
16          TestCountSuccess        : UDINT := 0;
17          TestCountFailure        : UDINT := 0;
18          Progress                : LREAL := 0.0;
19          Running                 : BOOL := FALSE;
20  END_VAR
21
22  CASE State OF
23
24          sIdle:
25                  IF Start THEN
26                          (* Variablen für eine Testausführung initialisieren *)
27                          TestDone := FALSE;
28                          TestSuccess := FALSE;
29                          Timer(IN := FALSE);
30                          Progress := 0.0;
31
32                          Start := FALSE;
33                          State := sStarting;
34                          Running := TRUE;
35                  END_IF
36
37          sStarting:
38                  IF NOT PUT_SR_PacUnitTest.xInit THEN
```

```
39                          (* Tests starten *)
40                          pxTmp := ADR(PUT_SR_PacUnitTest.xStart);
41                          pxTmp^ := TRUE;
42                          State := sRunning;
43                  END_IF
44
45          sRunning:
46                  Progress := PUT_SR_PacUnitTest.lrProgress;
47
48                  IF (PUT_SR_PacUnitTest.xTestSuiteDone) THEN
49                          TestSuccess := PUT_SR_PacUnitTest.xOverallSuccess;
50                          TestCountSuccess := PUT_SR_PacUnitTest.udiTestsFailed;
51                          TestCountFailure := PUT_SR_PacUnitTest.udiTestsSucceeded;
52
53                          (* wenn Tests beendet sind, dann Ergebnisdatei speichern *)
54                          psTmp := ADR(PUT_SR_PacUnitTest.sOperatorName);
55                          psTmp^ := 'TestManagementCom';
56                          pxTmp := ADR(PUT_SR_PacUnitTest.xSaveResult);
57                          pxTmp^ := TRUE;
58                          State := sSaving;
59                  END_IF
60
61          sSaving:
62                  IF (NOT PUT_SR_PacUnitTest.xSaveResult) THEN
63                          (* 500ms auf Dateioperationen warten *)
64                          Timer(IN := TRUE, PT := T#500ms);
65                  END_IF
66
67                  IF Timer.Q THEN
68                          (* alles fertig *)
69                          pxTmp := ADR(PUT_SR_PacUnitTest.xInit);
70                          pxTmp^ := TRUE;
71                          TestDone := TRUE;
72                          State := sIdle;
73                          Running := FALSE;
74                  END_IF
75
76  END_CASE
77  END_PROGRAM
```

PUT_TestTcpController

```
1   PROGRAM PUT_TestTcpController
2   VAR CONSTANT
3           MAX_RULES                       : INT := 2;
4
5           CONNECTION_TIMEOUT              : UDINT := 60;
6           MAX_TIME_RATIO_FOR_AC           : INT := 3;
7
8           (* values for increasing the cycle check *)
9           OVERRUN_TIME                    : DINT := 100;
10          WDOG_TIME                       : DINT := 1000;
11  END_VAR
12
13  VAR
14          InitDone                        : BOOL := FALSE;
15
16          I                               : INT;
17
18          Rules                           : ARRAY[1..MAX_RULES] OF ST_ETCP_AC_AccessRule
19                                                  := (lwIP := 0, lwMask := 0, ICountOK := -1);
20
21          Frame                           : ST_ETCP_ServerFrame;
22
23          MemPort                         : UINT;
24          MaxConnections                  : INT := g_ETCP_SERVER_FRAME_MAX_CONNECTIONS;
25
26          Receive                         : ARRAY[1..g_ETCP_SERVER_FRAME_MAX_CONNECTIONS]
27                                                  OF ARRAY[1..RECEIVE_BUFFER_SIZE] OF BYTE;
28          pReceive                        : POINTER TO ARRAY[1..RECEIVE_BUFFER_SIZE] OF BYTE;
29
30          Send                            : ARRAY[1..g_ETCP_SERVER_FRAME_MAX_CONNECTIONS]
31                                                  OF ARRAY[1..SEND_BUFFER_SIZE] OF BYTE;
32          pSend                           : POINTER TO ARRAY[1..SEND_BUFFER_SIZE] OF BYTE;
33
34          Notify                          : ARRAY[1..g_ETCP_SERVER_FRAME_MAX_CONNECTIONS]
35                                                  OF NotifyState := nDisabled;
36
37          SendResult                      : BOOL;
38
39          StartTrig                       : R_TRIG;
```

```
40          Running                         : BOOL := FALSE;
41
42          (* Empfangsnachrichten: *)
43          MessageLength                   : INT;
44          lBytesProcessed                 : INT;
45
46          Command                         : STRING(255);
47          CommandLength                   : INT;
48
49          (* Tests fertig *)
50          TestDoneTrig                    : R_TRIG;
51          TestDone                        : BOOL := FALSE;
52          TestDoneText                    : STRING(255);
53  END_VAR
54
55  VAR_INPUT
56          Enable                          : BOOL := FALSE;
57          Port                            : UINT := 4688;
58  END_VAR
59
60
61  IF NOT InitDone THEN
62          Init();
63  END_IF
64
65  StartTrig(CLK := Enable);
66  IF StartTrig.Q THEN
67          Start();
68  END_IF
69
70  IF Running THEN
71          Run();
72  END_IF
73
74  END_PROGRAM
```

Init

```
1   Frame.conf_iMaxConnections := MaxConnections;
2
3   (* connections array, initialize the send and receive buffers of the connections *)
4   FOR I := 1 TO Frame.conf_iMaxConnections DO
5           Frame.ConArr[I].bACisUsing := TRUE;
6           Frame.ConArr[I].fbCon.Send_lwSendBufferAdr := ADR(Send[I][1]);
7           Frame.ConArr[I].fbCon.Receive_lwReceiveBufferAdr := ADR(Receive[I][1]);
8           Frame.ConArr[I].fbCon.Receive_lReceiveBufferSize := RECEIVE_BUFFER_SIZE;
9   END_FOR
10
11  (* set the socket options in the rules, we set here equal socket
12     options for all rules, but you could also set them on a per rule basis. *)
13  FOR I := 1 TO MAX_RULES DO
14          (* set the socketoptions in the rules *)
15          Rules[I].stSockOpts.bModify_TCPNoDelay := TRUE;
16          Rules[I].stSockOpts.bTCPNoDelay := TRUE;
17          Rules[I].stSockOpts.bModify_TCPResuseAddr := TRUE;
18          Rules[I].stSockOpts.bTCPReuseAddr := TRUE;
19          Rules[I].stSockOpts.bModify_TCPKeepAlive := TRUE;
20          Rules[I].stSockOpts.bTCPKeepAlive := TRUE;
21  END_FOR
22
23  F_ETCP_AC_CreateAccessRule(sIP := '0.0.0.0', sMask := '0.0.0.0', stRule := Rules[1]);
24  Rules[1].iCountOK := 10;
25
26  (* create catch-all-deny rule *)
27  Rules[2].lwIP := 0;
28  Rules[2].lwMask := 0;
29  Rules[2].iCountOK := -1;
30
31  (* initialize the server frame *)
32  Frame.conf_iMaxTimeRatioForAC := MAX_TIME_RATIO_FOR_AC;
33  Frame.conf_pRuleArr := ADR(Rules);
34
35  (* calculate the kill value, so that a connection whose peer did
36     not receive for ~60 seconds is closed *)
37  Frame.conf_ulKillConIfSendFullForXCycles := CONNECTION_TIMEOUT / LzsTaskGetInterval();
38  Frame.conf_iRuleArrSize := MAX_RULES;
39
40  (* increase the cycle check time *)
41  CycleCheckTimeSet(OVERRUN_TIME, WDOG_TIME);
42
43  InitDone := TRUE;
```

Start

```
Running := TRUE;
```

Run

```
Frame.dyn_bEnable := Enable;
Frame.dyn_uiPort := Port;

F_ETCP_ServerFrame1(Frame);

WHILE(Frame.out_bServerFrameDoWhile) DO
        F_ETCP_ServerFrame2(Frame);

        (* beim Beenden der Verbindung den Stauts des Notify zurücksetzen *)
        IF(Frame.out_bClosedByPeer) THEN
                Notify[Frame.out_iActiveCon] := nDisabled;
        END_IF

        IF(Frame.out_bSendFullClosed) THEN
                Notify[Frame.out_iActiveCon] := nDisabled;
        END_IF

        (* Datenempfang *)
        IF (Frame.out_bData) THEN
                pReceive := ADR(Receive[Frame.out_iActiveCon][1]);
                lBytesProcessed := 0;

                (* sind noch empfangene Daten im Puffer *)
                WHILE ((Frame.dyn_alReceivePos[Frame.out_iActiveCon] - lBytesProcessed) > 0) DO

                        (* erstes Newline im Empfangspuffer finden *)
                        MessageLength := FindFirstNewline(Input := pReceive^,
                                InputLength := DINT_TO_INT(
                                        Frame.dyn_alReceivePos[Frame.out_iActiveCon] -
                                        lBytesProcessed));

                        IF (MessageLength = 0) THEN
                                (* Datenauswertung stoppen, keine weiteren kompletten Nachrichten *)
                                EXIT;
                        END_IF

                        (* String einlesen *)
                        CommandLength := F_ETCPUDP_ReadSTRING(pBuffer := pReceive,
                                                              pString := ADR(Command),
                                                              iMaxLength := MIN(256, MessageLength),
                                                              bStopAtZero := TRUE);
                        Command := LowerCase(Command);

                        SendResult := TRUE;

                        (* Protokoll *)
                        IF (Command = 'start') THEN
                                (* wenn Tests noch laufen, warten *)
                                IF (PUT_TestManagementCom.Running OR TestDone) THEN
                                        EXIT;
                                END_IF

                                (* Tests starten *)
                                IF (Notify[Frame.out_iActiveCon] <> nEnabled) THEN
                                        Notify[Frame.out_iActiveCon] := nSingle;
                                END_IF

                                SendResult := SendResponse(Response := 'ok$n',
                                        Buffer := Send[Frame.out_iActiveCon],
                                        BufferPosition := Frame.dyn_alSendPos[Frame.out_iActiveCon]);
                        ELSIF (Command = 'quit') THEN
                                (* Verbindung beenden *)
                                Notify[Frame.out_iActiveCon] := nDisabled;
                                Frame.ConArr[Frame.out_iActiveCon].fbCon.Close();
                                EXIT;

                        ELSIF (Command = 'notify') THEN
                                (* Benachrichtigung bei fertigen Tests aktivieren *)
                                Notify[Frame.out_iActiveCon] := nEnabled;

                                SendResult := SendResponse(Response := 'ok$n',
                                        Buffer := Send[Frame.out_iActiveCon],
                                        BufferPosition := Frame.dyn_alSendPos[Frame.out_iActiveCon]);

                        ELSE
```

```
76                                  (* Fehler *)
77                                  SendResult := SendResponse(Response := 'unknown command$n',
78                                          Buffer := Send[Frame.out_iActiveCon],
79                                          BufferPosition := Frame.dyn_alSendPos[Frame.out_iActiveCon]);
80                          END_IF
81
82                          (* kein Platz im Sendepuffer *)
83                          IF (NOT SendResult) THEN
84                                  Frame.dyn_bSendWasFull := TRUE;
85                          END_IF
86
87                          (* Nachricht wurde ausgewertet *)
88                          lBytesProcessed := lBytesProcessed + CommandLength;
89                          pReceive := pReceive + INT_TO_DWORD(CommandLength);
90                  END_WHILE
91
92                  (* verbleibenden Daten an den Pufferanfang verschieben *)
93                  MoveRemainingData(ProcessedBytes := lBytesProcessed,
94                                    ReceivedBytes := Frame.dyn_alReceivePos[Frame.out_iActiveCon],
95                                    ReceiveBuffer := Receive[Frame.out_iActiveCon]);
96          END_IF
97
98          (* Nachrichten an verbundenen Client schicken *)
99          IF(Frame.out_bOpen) THEN
100                 TestDoneTrig(CLK := PUT_TestManagementCom.TestDone);
101
102                 IF (TestDoneTrig.Q) THEN
103                         (* Tests fertig, Nachricht zusammen bauen *)
104                         TestDone := TRUE;
105                         TestDoneText := 'tests dont: ';
106                         TestDoneText := CONCAT(TestDoneText,
107                                         UDINT_TO_STRING(PUT_TestManagementCom.TestCountSuccess));
108                         TestDoneText := CONCAT(TestDoneText,
109                                         ' succeded, ');
110                         TestDoneText := CONCAT(TestDoneText,
111                                         UDINT_TO_STRING(PUT_TestManagementCom.TestCountFailure));
112                         TestDoneText := CONCAT(TestDoneText,
113                                         ' failed$n');
114                 END_IF
115
116                 IF (TestDone AND (Notify[Frame.out_iActiveCon] = nEnabled OR
117                                   Notify[Frame.out_iActiveCon] = nSingle)) THEN
118                         (* Tests fertig, Nachricht senden*)
119                         SendResult := SendResponse(Response := TestDoneText,
120                                 Buffer := Send[Frame.out_iActiveCon],
121                                 BufferPosition := Frame.dyn_alSendPos[Frame.out_iActiveCon]);
122
123                         IF (SendResult) THEN
124                                 (* Benachrichtigungsstatus zurücksetzen *)
125                                 IF (Notify[Frame.out_iActiveCon] = nSingle) THEN
126                                         Notify[Frame.out_iActiveCon] := nDisabled;
127                                 ELSE
128                                         Notify[Frame.out_iActiveCon] := nDone;
129                                 END_IF
130                         ELSE
131                                 Frame.dyn_bSendWasFull := TRUE;
132                         END_IF
133                 END_IF
134
135                 IF (CheckNotify(Notify)) THEN
136                         (* alle Benachrichtigungen fertig *)
137                         TestDone := FALSE;
138                 END_IF
139         END_IF
140
141         F_ETCP_ServerFrame3(Frame);
142 END_WHILE
```

CheckNotify

```
1   FUNCTION CheckNotify : BOOL
2   VAR_INPUT
3           Notify  : ARRAY [1..g_ETCP_SERVER_FRAME_MAX_CONNECTIONS] OF NotifyState;
4   END_VAR
5   VAR
6           I       : INT;
7   END_VAR
8
9   FOR I := 1 TO g_ETCP_SERVER_FRAME_MAX_CONNECTIONS DO
10          IF (Notify[I] = nSingle OR Notify[I] = nEnabled) THEN
11                  CheckNotify := FALSE;
```

```
12                      EXIT;
13              END_IF
14      END_FOR
15
16      FOR I := 1 TO g_ETCP_SERVER_FRAME_MAX_CONNECTIONS DO
17              IF (Notify[I] = nDone) THEN
18                      Notify[I] := nEnabled;
19              END_IF
20      END_FOR
21
22      CheckNotify := TRUE;
23      END_FUNCTION
```

FindFirstNewline

```
1       FUNCTION FindFirstNewline : INT
2       VAR_INPUT
3               Input           : ARRAY[1..RECEIVE_BUFFER_SIZE] OF BYTE;
4               InputLength     : INT;
5       END_VAR
6       VAR
7               I               : INT;
8       END_VAR
9
10      FindFirstNewline := 0;
11
12      FOR I := 1 TO InputLength DO
13              IF (Input[I] = 13 OR Input[I] = 10) THEN
14                      FindFirstNewline := I;
15                      RETURN;
16              END_IF
17
18              IF (Input[I] = 0) THEN
19                      RETURN;
20              END_IF
21      END_FOR
22      END_FUNCTION
```

LowerCase

```
1       FUNCTION LowerCase : STRING(255)
2       VAR_INPUT
3               InputString     : STRING(255);
4       END_VAR
5       VAR
6               Result          : STRING(255);
7               pString         : POINTER TO BYTE;
8       END_VAR
9
10      Result := InputString;
11      pString := ADR(Result);
12
13      WHILE pString^ > 0 DO
14              (* A-Z in a-z umwandeln *)
15              IF pString^ >= 65 AND pString^ <= 90 THEN
16                      pString^ := pString^ + 32;
17              END_IF
18
19              pString := pString + 1;
20      END_WHILE
21
22      LowerCase := Result;
23      END_FUNCTION
```

MoveRemainingData

```
1       FUNCTION MoveRemainingData : BOOL
2       VAR_INPUT
3               ProcessedBytes  : DWORD;
4       END_VAR
5       VAR_IN_OUT
6               ReceivedBytes   : DINT;
7               ReceiveBuffer   : ARRAY[1..RECEIVE_BUFFER_SIZE] OF BYTE;
8       END_VAR
9
10      (* verbleibenden Daten an den Pufferanfang verschieben *)
```

```
IF (ReceivedBytes - ProcessedBytes) > 0 THEN
        F_ETCPUDP_MemCopy(dest := ADR(ReceiveBuffer[1]),
                                          src := ADR(ReceiveBuffer[ProcessedBytes + 1]),
                                          LEN := ReceivedBytes - ProcessedBytes);
END_IF

ReceivedBytes := ReceivedBytes - ProcessedBytes;
END_FUNCTION
```

SendResponse

```
FUNCTION SendResponse : BOOL
VAR_INPUT
        Response        : STRING(255);
END_VAR
VAR_IN_OUT
        Buffer          : ARRAY [1..SEND_BUFFER_SIZE] OF BYTE;
        BufferPosition  : DINT;
END_VAR
VAR
        Length          : INT;
END_VAR


IF BufferPosition + LEN(Response) > SEND_BUFFER_SIZE THEN
        (* kein Platz mehr im Sendepuffer *)
        SendResponse := FALSE;
ELSE
        (* Nachricht senden *)
        Length := F_ETCPUDP_WriteSTRING(pBuffer := ADR(Buffer[BufferPosition]),
                                        pString := ADR(Response),
                                        iMaxLength := LEN(Response));

        BufferPosition := BufferPosition + Length;
        SendResponse := TRUE;
END_IF
END_FUNCTION
```

NotifyState

```
TYPE NotifyState :
(
        nDisabled,
        nEnabled,
        nSingle,
        nDone
);
END_TYPE
```

TcpConstants

```
VAR_GLOBAL CONSTANT
        RECEIVE_BUFFER_SIZE             : INT := 5000;
        SEND_BUFFER_SIZE                : INT := 5000;
END_VAR
```

B Literaturverzeichnis

[Bec99] Beck, Kent: *Kent Beck's Guide to Better Smalltalk*. Cambridge : Cambridge University Press, 1999. – ISBN 9780521644372

[Cor10] Corporation, Microsoft: *The Component Object Model*. `http://msdn.microsoft.com/de-de/library/ms694363.aspx`. Version: 12 2010

[Dij70] Dijkstra, Edsger W.: *Notes on Structured Programming*. `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF`. Version: April 1970. – circulated privately

[Fow10] Fowler, Martin: *MF Bliki: Xunit*. `http://martinfowler.com/bliki/Xunit.html`. Version: 2010

[HT99] Hunt, Andrew ; Thomas, David: *The pragmatic programmer: from journeyman to master*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999. – ISBN 0–201–61622–X

[Jef01] Jeffries, Ron: *Extreme Programming Installed*. Boston : Addison-Wesley, 2001. – ISBN 0201708426

[MBTS04] Myers, G.J. ; Badgett, T. ; Thomas, T.M. ; Sandler, C.: *The art of software testing*. John Wiley & Sons, 2004 (Business Data Processing: A Wiley Series). – ISBN 9780471469124

C Abbildungsverzeichnis

3.1 Oberfläche von EPAS-4: Multi-Dokument-Interface für die Bearbeitung von mehreren Bausteinen parallel 16
3.2 Visualisierung zur Steuerung der Tests. Oben werden die verschiedenen Testfälle und ihr Erfolg oder Misserfolg aufgelistet und unten befinden sich Buttons zur Steuerung. 23

4.1 EPAS-5 Workbench zum Starten der Entwicklungsumgebung, allen Hilfsprogrammen und sogar der alten Entwicklungsumgebung 27
4.2 Oberfläche von EPAS-5 auf Basis von Visual Studio 2008 28
4.3 Zentrale Verwaltung von Bibliotheken 30

5.1 Oberfläche von TwinCAT 3 auf Basis von Visual Studio 2010 34

D Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Bachelorarbeit selbständig verfasst und nur unter Zuhilfenahme der angegebenen Quellen erstellt zu haben.

Berlin, 14. Januar 2011 _____ _____

Ort, Datum Alexander Sulfrian