Kapitel 4

Das Testsystem

4.1 Übersicht

Das zu konstruierende System für den dynamischen Softwaretest einer im Reverse Engineering (RE) befindlichen Software-Komponente soll folgende Funktionen haben bzw. unterstützen:

1. Systematische Testfallentwicklung einschließlich der Testdatenauswahl und der Erzeugung des Testprogrammes bzw. Testskriptes.

2. Speicherung und Verwaltung der Testspezifikationen, der Soll-Daten, der Ist-Daten der Testausführung sowie der Testergebnisse zum Zweck der Verarbeitung und der Testdokumentation.

3. Testauswertung durch Vergleich von Soll- und Ist-Daten, Aufbereitung der Testergebnisse für die Testdokumentation und Bewertung der Vollständigkeit des Tests.

4. Organisation der Testdurchführung durch Unterstützung der Ausführung einzelner Schritte der Tests, einzelner Tests und kompletter Regressionstests.

Es gibt bereits eine Reihe von Werkzeugen bzw. Werkzeugpaketen, die einzelne Punkte unterstützen, bzw. sich als vollständige Testsysteme empfehlen. Die Evaluierung solcher Werkzeuge, die im Rahmen dieser Arbeit nicht sehr umfangreich bzw. vollständig sein konnte, hat ergeben, dass keines der Werkzeugpakete wirklich alle diese Punkte erfüllt und dass viele der Werkzeuge aus technischen Gründen nicht auf das XCTL-Projekt anwendbar waren.

Eines der Ausschlusskriterien bei der Werkzeugsuche war die Betriebssystemauswahl, für die die Werkzeuge zur Verfügung stehen. Das zu testende XCTL-System ist eine 16-Bit Anwendung für Windows 3.1 und soll erst im Zuge des Reverse Engineering auf die Win32-Plattform portiert werden und dann auf Windows 95/98 laufen. Sehr viele der Werkzeuge aber erwarten aus unterschiedlichen Gründen Testanwendungen, die 32-Bit-Anwendungen sind. Sie fallen damit für das Projekt aus, da das Testsystem gerade die Reverse Engineering-Arbeiten durch einen Regressionstest begleiten und deren Korrektheit sicher stellen soll.

Das Ergebnis der Evaluierung war also, dass ein Testsystem für das XCTL-Programm aus einer Reihe von fertigen und selbstgeschriebenen Einzelwerkzeugen zusammengestellt werden muss. Im Folgenden wird ein Überblick über die im entwickelten Testsystem verwendeten Werkzeuge, Methoden und Technologien gegeben.

Testfallentwicklung Für die systematische Testfallentwicklung wird der *Classification Tree Editor (CTE)* verwendet. Der CTE ist ein graphisches Werkzeug für die Anwendung der *Klassifikationsbaummethode*, einer Methode für die systematische Klassifizierung der Eingabedaten eines Testobjekts sowie für die Bestimmung von Testfällen aus dieser Klassifikation. Zur Klassifikationsbaummethode s.u. im Abschnitt 4.5.

Die Testdatenauswahl unterstützt der CTE durch den Export einer Testfallspezifikation, welche – in halbformaler, textueller Form – die einen Testfall konstituierenden Klassen wiedergibt. Mein erster Ansatz bei der Erzeugung der Testprogramme war, diese Testfallspezifikationen als Vorlage für das manuelle Erarbeiten der Testprogramme zu verwenden. Dieser Vorgang erwies sich als sehr aufwendig und fehleranfällig. Fehler waren a) die üblichen Probleme bei manuellen Arbeiten, wie Verwechslungen, *Copy-and-Past*-Fehler, Übersichts- und Konzentrationsprobleme, sowie b) inkonsistente Testdaten. So wurden z.B. gleiche Klassen durch unterschiedliche Testdaten in den Testprogrammen repräsentiert. Ein weiteres Problem war der hohe Aufwand im Fall der Überarbeitung des Klassifikationsbaums.

Diese Probleme haben mich dazu veranlasst, nach einer automatisierten Lösung für die Testprogrammerzeugung zu suchen und ein entsprechendes Werkzeug zu schreiben. Ausgangspunkt dieser Lösung war ein Feature des CTE, das es ermöglicht die Elemente eines Klassifikationsbaums mit Attributen zu versehen. Voraussetzung war ebenso, dass die Dateien, die der CTE erzeugt, Textdateien sind, deren Struktur gut verständlich ist. Zu diesem Werkzeug s. Abschnitt 4.6.

Datenverwaltung Als Grundlage der Datenverwaltung wurde der Datenstandard XML (*Extensible Markup Language*) gewählt. Er ermöglicht die strukturierte Speicherung von Daten und Meta-Daten sowie die Verwendung einer ganzen Reihe vorhandener Werkzeuge zur Verarbeitung und Auswertung der Daten. Die leitende Vorstellung war dabei, dass die Daten jeden Typs, die im Testsystem erzeugt oder verwendet werden, nach Möglichkeit direkt Bestandteil der Testdokumentation sein sollten. So sind z.B. die Soll- und die Ist-Daten eines Testfalles durch den Vergleicher (s.u.) auswertbar, können aber ebenso mit Hilfe von Standard-Werkzeugen wie Internetbrowsern als Textdokumente angezeigt und ausgedruckt werden und können so Bestandteil der (Verhaltens-)Spezifikation der XCTL-Software werden. Fast alle während des Tests entstehenden Daten bilden zusammen auf diese Weise eine interaktive Online-Testdokumentation.

Die eigentliche Datenverwaltung findet in Dateien und Verzeichnissen statt und wird im Abschnitt 4.4 erläutert.

Testauswertung Die Testauswertung muss unter zwei Aspekten erfolgen: a) der Korrektheit des Tests und b) seiner Vollständigkeit. Für a), die Überprüfung

ob die Ausgaben einer Testausführung bestimmten Erwartungen entsprechen, ist ein Vergleicher geschrieben worden, der unter 4.7 beschrieben wird. Für b), die Ermittlung der Codeüberdeckung zur Beurteilung der Vollständigkeit des Tests, sollte auf ein vorhandenes Werkzeug zurückgegriffen werden. Es konnte aber keines gefunden werden, das die Codeüberdeckung von 16-Bit-Anwendungen bestimmen kann. Bei Objektcode-instrumentierenden Werkzeugen ist das verständlich, dagegen ist das bei Quellcode-Instrumentierung schwerer nachzuvollziehen. Dieser Punkt wird im vorliegenden Testsystem also noch nicht realisiert.

Testorganisation Zur Organisation der Testdurchführung habe ich das Standard-Softwareentwicklungswerkzeug *make* gewählt. Dies ermöglicht die Formulierung von Abhängigkeiten zwischen den als Dateien vorliegenden Resultaten der einzelnen Schritte der Testdurchführung und die Festlegung der Aktionen, die während der einzelnen Schritte notwendig sind. Damit lassen sich einzelne Zwischenergebnisse erstellen, einzelne Testfälle und komplette Regressionstests durchführen. Der zu organisierende Testablauf wird im folgenden Kapitel 4.2 erläutert, die eigentliche Organisation im Kapitel 4.4.

4.2 Der Testablauf im Überblick

Der prinzipielle Testablauf unter dem Gesichtspunkt der verwendeten Werkzeuge und der Ein- und Ausgaben, die diese verwenden bzw. erzeugen, wird in Abb. 4.1 dargestellt.

Am Anfang steht der *Classification Tree Editor*. Mit seiner Hilfe werden die Eingabedaten klassifiziert, wird die Testdatenauswahl durchgeführt und werden die Testfälle ausgewählt. Außerdem wird die Testskriptgenerierung vorbereitet. Das Ergebnis dieser Schritte ist eine CTE-Datei, die den mit Attributen versehenen Klassifikationsbaum enthält.

Im zweiten Schritt, der Testprogrammerzeugung, werden zuerst mit Hilfe des Programmes `cte2cpp` aus dem Klassifikationsbaum für jeden Testfall bzw. für jede Testsequenz ein Testskript generiert[1]. Da es nicht möglich war, eine allgemeingültige Form solcher Testskripte festzulegen, verwendet das Programm einen für den konkreten Test angepassten Testskriptgenerator. Im vorliegenden Testsystem sind die Testskripte C++-Dateien (CPP). Diese müssen zur Testausführung übersetzt werden. In diesen Übersetzungsprozess gehen im konkreten System weitere C++-Dateien für den Testrahmen mit ein und ebenso das Testobjekt, sprich: die zu testende Komponente selbst. Resultat dieses Schrittes ist ein ausführbares Programm (EXE).

Im eigentlichen Testschritt wird das Testprogramm ausgeführt und Informationen über den Testlauf werden in einer LOG-Datei protokolliert. Im Falle einer entsprechenden Instrumentierung der Testanwendung entstehen während dieses Schrittes ebenso Informationen über die Codeüberdeckung (TRACE). Im aktuellen Testsystem ist dieser Punkt noch nicht realisiert.

Im letzten Schritt werden die Ist-Ausgaben der Testausführung (LOG) mit den erwarteten Ausgaben (BAS) verglichen. Dazu dient das Programm `run_test`. Die Ergebnisse des Vergleiches werden in einer Ergebnisdatei (RES)

[1] Das Werkzeug ist trotz seines Namen nicht auf die Erzeugung von Testskripten in einer bestimmten Programmiersprache festgelegt.

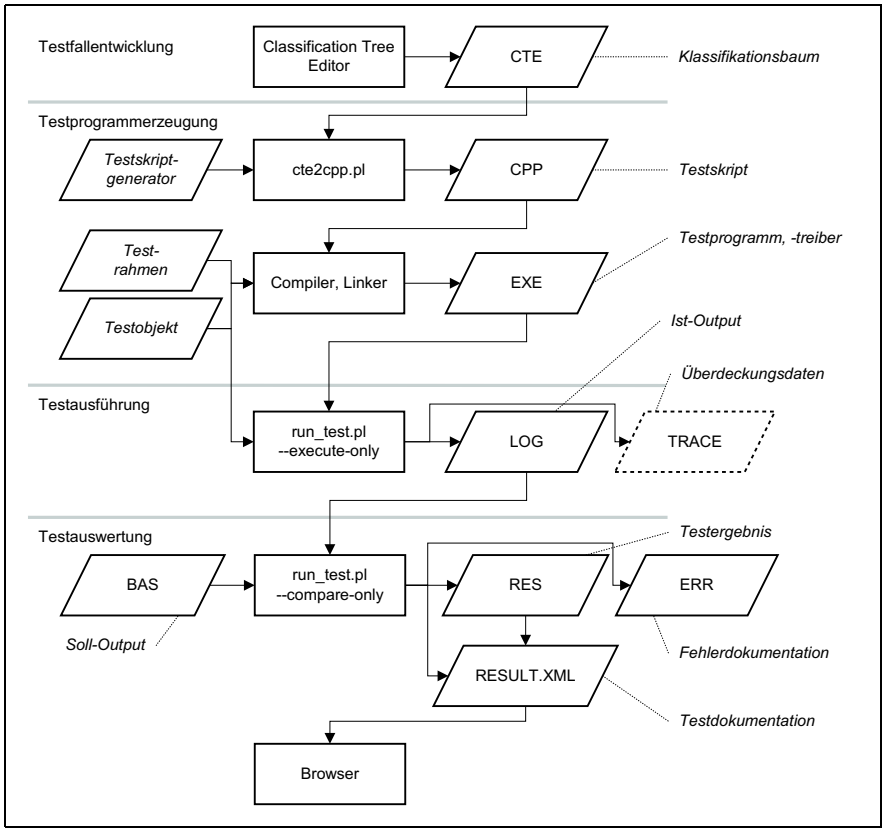

Abbildung 4.1: Werkzeuge und Datenfluss im Testsystem

gespeichert, und für jeden Testfall bzw. Testschritt, in dem eine Abweichung zwischen Ist und Soll auftritt, wird eine Fehlerbeschreibungsdatei (ERR) angelegt. Abschließend werden die Testergebnisse in die Gesamtdokumentation (RESULT.XML) des Tests integriert. Diese kann dann mit einem Browser (Internet Explorer) gelesen werden.

4.3 Begriffe

Bevor die Details des Testsystems erläutert werden, müssen zunächst einige Begriffe definiert werden.

Definition: Build *Ein* build *ist eine reproduzierbare, lauffähige Version der zu testenden Anwendung. Getestet wird stets ein bestimmter* build.

Ein *build* ist zu charakterisieren durch eine bestimmte Version der Quelltext-Dateien, durch die Übersetzungsparameter und die Umgebung in der er erzeugt wurde. Im XCTL-Projekt ist dies gegeben durch die Angabe des *CVS-Repositories*, in dem die Version gespeichert ist, des *CVS-Modules* und eines *CVS-Tags*, das die Version im Repository markiert, sowie die Angabe zum verwendeten Compiler und Betriebssystem.

Definition: 0-Build *Der nullte* build *ist die Version des zu überarbeitenden*

Programmes, die Ausgangspunkt des Reverse Engineering ist.

Im Idealfall wurden an dieser Version noch keinerlei RE-Schritte durchgeführt. Unter Umständen kann es aber notwendig sein, Änderungen durchzuführen, die den Test überhaupt erst ermöglichen. Im Fall der Motorenkomponente des XCTL-Programmes waren solche Änderungen zur Integration einer Hardwaresimulation notwendig. Zu diesen Änderungen s. Kapitel 3.3.

In diesem Testsystem, das in einem RE-Projekt angewendet werden soll, werden zwei Typen von Soll-Werten unterschieden:

Definition: Soll-Werte Typ 1 (baseline) *Der erste Soll-Werte-Typ enthält die Ausgaben, die das zu testende Programm in seinem letzten als gültig definierten Zustand* erzeugt hat[2].

Die Ausgangs-*baseline* ist durch das Verhalten des zu überarbeitenden Programmes in seiner Ausgangsversion (*0-build*) gegeben. Nach der Erstellung der möglichst vollständigen *baseline*-Daten können einzelne Bearbeitungsschritte kontrolliert am RE-Objekt durchgeführt werden. Durch einen Regressionstest ist überprüfbar, welche Auswirkungen ein Schritt auf das Programmverhalten hatte. Hat der Bearbeitungsschritt das Verhalten in beabsichtigter Art- und Weise verändert, wird die *baseline* diesem neuen Verhalten angepasst.

Daraus folgt, dass *baseline*-Daten in verschiedenen Versionen vorliegen bzw. mindestens in zwei Versionen: der Ausgangsversion, die mit dem *0-build* erzeugbar ist und die Version, die mit dem aktuell als gültig betrachteten *build* erzeugt wird. Unter Umständen ist es sinnvoll einzelne Zwischenschritte in der Entwicklung der *baseline* aus Dokumentations- und Nachvollziehbarkeitsgründen aufzubewahren.

Definition: Soll-Werte Typ 2 (targetline) *Der zweite Soll-Werte-Typ enthält die Ausgaben des Programms, die es* erzeugen sollte.

Dieser Soll-Daten-Typ ermöglicht es Erkenntnisse des RE-Prozesses über das eigentliche Soll-Verhalten schrittweise in die Soll-Daten einzuarbeiten ohne das konservierte Ist-Verhalten zu verlieren.

Demzufolge gibt es auch zwei verschiedene Typen von Fehlern:

Definition: Fehler Typ 1 *Fehler vom Typ 1 sind Abweichungen zwischen den Ist-Ausgaben des Testobjektes und den* baseline-*Daten.*

Definition: Fehler Typ 2 *Fehler des Typs 2 sind Unterschiede zwischen den Ist-Ausgaben des Testobjektes der* targetline.

Definition: Testfall *Ein Testfall ist eine Menge von Daten, die einem Testobjekt bei dessen Ausführung zur Verfügung gestellt werden. Für jeden Testfall wird das Testobjekt neu erzeugt. Mehrere Testfällen sind von einander unabhängig.*

Ist ein Testobjekt z.B. eine Methode, so sind die Testdaten zum einen die konkreten Parameter die der Methode übergeben werden, zum anderen aber auch der Zustand des Objektes zu dem die Methode gehört und eventuell der Zustand des Systems von dem das Objekt ein Teil ist.

Um ein Objekt oder ein System in einen bestimmten Zustand zu versetzen, ist es häufig notwendig andere Methoden vorher aufzurufen. So muss z.B.

[2] Die Bezeichnung *baseline* entstammt dem Testwerkzeugpaketes der Firma Rational.

ein Motor erst in Bewegung gesetzt werden, um sinnvoll das Stoppen testen zu können. Wenn die vorbereitenden Aktionen auch Teil des Tests sein sollen, ist es sinnvoll eine solche Abfolge als speziellen Testfall, als Testsequenz, zu bezeichnen.

Definition: Testsequenz, Testschritt *Eine Testsequenz ist eine Abfolge von Testschritten. Für jede Testsequenz wird das Testobjekt neu erzeugt. Mehrere Testsequenzen sind von einander unabhängig. Die einzelnen Testschritte werden – analog zu den Testfällen – jeweils durch eine Menge von Testdaten charakterisiert, auf die das Testobjekt angewendet wird. Sie sind aber – im Gegensatz zu den Testfällen – voneinander anhängig, da jeder Schritt die Bedingungen der nachfolgenden Schritte beeinflusst.*

Die Begriffe Testsequenz und Testschritt sind unter anderem aus deren Verwendung im *Classification Tree Editor* abgeleitet. Wenn im folgenden von Testfällen oder von Testsequenzen die Rede ist, sind i.d.R. *Testfälle und Testsequenzen* gemeint.

Für jeden Testfall und jede Testsequenz gibt es eine im Testsystem eindeutige Kennzeichnung (Test-ID).

Definition: Testpaket *Ein Testpaket ist eine Gruppe von zusammengehörigen Testfällen bzw. Testsequenzen, die alle das gleiche Testobjekt unter gleichen Aspekten testen.*

4.4 Datenverwaltung und Testorganisation

In diesem Abschnitt soll die konkrete Organisation der Daten und der Testdurchführung erläutert werden. Die Daten werden in Dateien und Verzeichnissen verwaltet. Die Organisation der Testdurchführung wird über *Makefiles* realisiert.

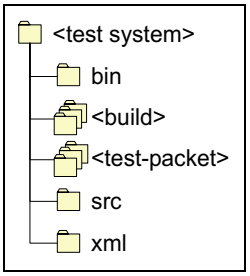

Abbildung 4.2: Die Verzeichnisse des Testsystems

Die Verzeichnisse des Testsystems sind in Abb. 4.2 zu sehen. Es gibt ein Verzeichnis für jedes *Testpaket* und eins für jeden zu testenden *Build*. Das `bin`-Verzeichnis enthält einige Werkzeuge des Testsystems, das `src`-Verzeichnis die Quelltexte der Testrahmen, die von den Testskripten bzw. Testprogrammen verwendet werden und das `xml`-Verzeichnis enthält einige Hilfsdateien für die Testdokumentation, wie XSL-Stylesheets, Dokumenttypdefinitionen und ähnliches.

Testpaket Für jedes Testpaket gibt es ein Dateiverzeichnis mit dem Kurzzeichen des Paketes als Namen. Im Augenblick gibt es drei Testpakete: `m_init`,

`m_layer` und `m_rpl`. Für detaillierte inhaltliche Erläuterungen zu den einzelnen Paketen s. Kapitel 5.

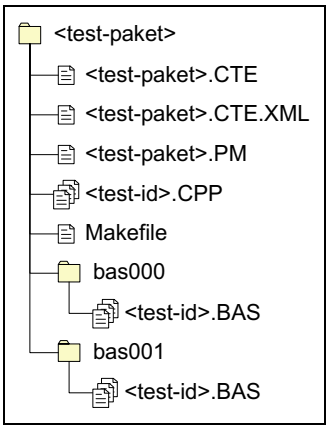

Abbildung 4.3: Die Dateien eines Testpaketes

Kern eines Testpakets ist eine CTE-Datei, d.h. ein attributierter Klassifikationsbaum mit einer Spezifikation für alle Testfälle. Für jeden Testfall des Paketes enthält das Verzeichnis ein Testskript mit der Test-ID des Testfalles als Namen. Eventuell können auch weitere Dateien je Testfall dort auftauchen, wenn diese für die Ausführung des Testskriptes notwendig sind. In den vorliegenden Testpaketen sind die Testskripte C++-Dateien.

Ein *Makefile* spezifiziert für jeden Testfall, wie das dazugehörige Testskript zu generieren ist. Jede Test-ID ist ein Ziel in diesem *Makefile*, so dass die Kommandozeile `make <test-id>` das entsprechende Testskript und evtl. weitere Dateien erzeugt. In den vorliegenden Testpaketen wird für die Skriptgenerierung das Werkzeug `cte2cpp` verwendet. Dieses verwendet einen Skriptgenerator, der in den vorliegenden Fällen paketspezifisch ist und deshalb im Testpaket liegt (`<test-paket>.PM`). Details zur Skriptgenerierung sind im Abschnitt 4.6 zu finden.

Neben den Testfallspezifikationen und den Testskripten enthält das Testpaket die Soll-Daten für die Testfälle. Da Soll-Daten in verschiedenen Versionen vorliegen, enthält das Verzeichnis für jede Soll-Daten-Version ein Unterverzeichnis, in dem für jeden Testfall eine Soll-Daten-Datei vorhanden ist. Im aktuellen System gibt es zwei Versionen der Soll-Daten: `bas000` und `bas001`. Die Soll-Daten-Dateien enthalten sowohl die Angaben zur *baseline* als auch zur *targetline*. Zu Struktur und Format dieser Daten s. Abschnitt 4.7.

Zur Integration der Testfallspezifikationen in die Testdokumentation befindet sich desweiteren eine transformierte Version des CTE-Datei im Verzeichnis (`<test-paket>.CTE.XML`). Diese dient der textuellen Präsentation des attributierten Klassifikationsbaums und der Testfälle und kann mit dem Werkzeug `cte2xml` erzeugt werden (s. Anhang C).

Build Neben den Testpaketen enthält das Testsystem für jeden *build* ein Dateiverzeichnis. Dieses enthält die entsprechende Version der zu testenden Anwendung oder Komponente. Im vorliegenden Fall die `motors.dll` und die

Komponenten von denen sie abhängt (`splib.dll`, `bwcc.dll`, `bc450rtl.dll`, `ctl3d.dll`, `msim.dll`).

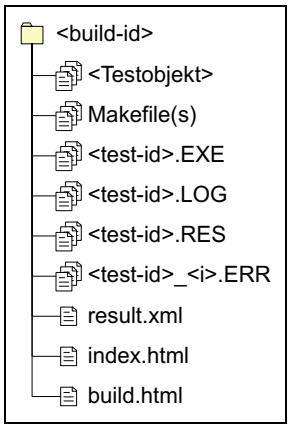

Abbildung 4.4: Die Dateien für einen Build

Da die Testskripte in den vorliegenden Testpaketen C++-Dateien sind, und diese vor Ausführung übersetzt werden müssen, enthält das Verzeichnis für jeden Testfall das übersetzte Testprogramm (`EXE`). Die Testausführung produziert für jeden Testfall ein Testprotokoll mit den Ist-Daten (`LOG`). Der Vergleich mit den Soll-Daten erzeugt für jeden Testfall eine Ergebnisdatei (`RES`) und eventuell notwendige Fehlerbeschreibungsdateien (`ERR`). Der Vergleicher wird unter Kapitel 4.7 beschrieben. Dieser generiert auch die Rahmendateien für die Testdokumentation: `result.xml` ist eine Zusammenführung aller Ergebnisdateien und `index.html` der Einstiegspunkt in die Dokumentation des Tests eines *builds*.

Ebenfalls zur Testdokumentation gehört die Datei `build.html`. Sie muss von Hand erstellt werden und liegt im Augenblick als nicht weiter strukturierte HTML-Datei vor. Sie soll alle Informationen enthalten, die notwendig sind, den *build* zu reproduzieren und den Test unter gleichen Bedingungen zu wiederholen. Dazu gehört:

- welche Quelltexte wurden verwendet,
- mit welchem Compiler und
- unter welchem Betriebssystem wurden diese übersetzt,
- auf welchem Betriebssystem und
- in welcher Umgebung fand der Test statt, und
- mit welcher Soll-Daten-Version wurde verglichen.

Die Durchführung des Tests wird über *Makefiles* organisiert. Der zu organisierende Ablauf ist der in Abb. 4.1 dargestellte. Die einzelnen Dateien werden in der folgenden Tabelle 4.5 kurz erläutert. Das realisierte System lässt an einigen Stellen noch etwas Eleganz vermissen, funktioniert aber gut.

Die Makefiles ermöglichen das Ausführen einzelner Schritte der Testdurchführung bis hin zu Regressionstests aller automatisierbaren Tests. Unter anderem sind folgende Kommandos möglich:

`make <test-id>.CPP`
Erstellt das CPP-Testskript.

`make <test-id>.EXE`
Erstellt das ausführbare Testsprogramm.

`Makefile`	Enthält Ziele für jeden Test und jedes Testpaket und `all` für alle automatisierbaren Tests. Delegiert die Aufgaben an die paketspezifischen Makefiles (s.u.).
`Makefile.tests`	Enthält die Namen aller Testpakete und für jedes Testpaket eine Liste der automatisiert und der manuell durchzuführenden Tests.

`Makefile.m_init`
`Makefile.m_layer`
`Makefile.m_rpl`
	Je ein *Makefile* für jedes Testpaket. Sie definieren die Ziele `<test-id>.CPP`, `<test-id>.EXE`, `<test-id>.LOG`, `<test-id>.RES` und `<test-id>`.

`Makefile.inc`	Enthält eine Reihe von Variablendefinitionen (Include-Verzeichnisse, Compiler-Parameter, etc.) sowie einige wiederholt benötigte Regeln.
`Makefile.bcc`	Enthält Regeln zum Erzeugen von 16-Bit Testprogrammen mit dem Borland-C++-Compiler.
`Makefile.vcc`	Enthält Regeln zum Erzeugen von 32-Bit Testprogrammen mit dem Mircosoft-C++-Compiler.
`Makefile.mdgl`	Makefile zum Erzeugen des Programmes `mgld.exe`, einem Hilfsprogramm für den Test der Dialoge der Motorenkomponente.

Abbildung 4.5: *Makefiles* zu Testdurchführung

`make <test-id>.LOG`
Führt das Testprogramm aus.

`make <test-id>.RES`
Führt die Testauswertung durch, falls `<test-id>.BAS` vorhanden ist.

`make <test-id>`
Identisch zu `make <test-id>.RES`.

`make <test-paket>`
Führt alle automatisierbaren Tests des gegebenen Testpaketes durch.

`make all`
Führt alle automatisierbaren Tests aller Testpakete durch.

4.5 Die Klassifikationsbaum-Methode

Die Klassifikationsbaum-Methode ist eine Methode zur systematischen Bestimmung von Testfällen.

Ausgangspunkt ist die Festlegung eines Testobjektes und die Identifikation aller relevanten Aspekte der Eingabedaten des Testobjektes, d.h. derjenigen Aspekte, von denen erwartet wird, dass sie das Verhalten des Testobjektes beeinflussen. Jeder dieser Aspekte wird als Klassifikation der Eingabedaten des Testobjektes betrachtet, wird also verwendet um die Eingabedaten des Testobjektes in Äquivalenzklassen zu zerlegen. Solche Klassifikationen müssen stets wenigsten eine Klasse enthalten; sinnvoll sind in der Regel aber nur Klassifikationen mit mehr als einer Klasse. Die gefundenen Äquivalenzklassen können wiederum bezüglich relevanter Aspekte klassifiziert werden. Dadurch entsteht ein Baum aus Klassen und Klassifikationen – ein Klassifikationsbaum.

Ein Testfall wird dann mit Hilfe dieses Klassifikationsbaums gebildet, in dem die entworfenen Klassen mit einander kombiniert werden – jeweils höchstens eine aus jeder Klassifikation. Durch die Kombination der Klassifikationen des Baumes entsteht also eine neue Klassifikation, die den gesamten Eingabebereich des Testobjektes in Äquivalenzklassen zerlegt. Jede der nichtleeren Klassen dieser Kombination stellt potentiell einen Testfall dar. Ein Testfall wird demzufolge durch eine Menge von Blättern (Klassen) des Klassifikationsbaums konstituiert.

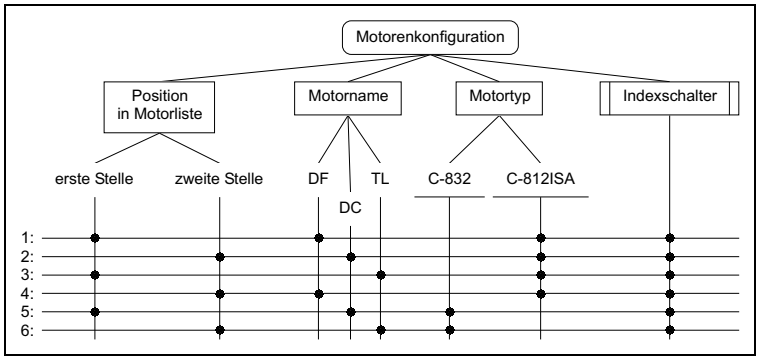

Abbildung 4.6: Beispiel für einen Klassifikationsbaum (1)

Die Zahl der möglichen Testfälle wird schnell sehr groß, so dass der Tester eine sinnvolle Auswahl treffen muss. Als Minimalkriterium kann hier gelten, dass wenigsten jede Klasse des Klassifikationsbaums in einer Testfalldefinition verwendet werden sollte. Ob dies ausreicht oder ob mehr Testfälle notwendig sind, muss der Tester auf Grund seiner Kenntnis des Testobjektes beurteilen.

Die Vollständigkeit des Tests hängt also auch bei dieser Methode stark vom Wissen und Können des Testers ab. Das beginnt bei der Konstruktion des Baumes, bei der alle relevanten Aspekte gefunden werden müssen und endet mit der Auswahl einer Menge von Testfällen. Die Beurteilung der Vollständigkeit des Tests sollte daher durch weitere Kriterien, wie z.B. Quellcodeüberdeckungsmaße, ergänzt werden. Außerdem sollte ein Testsystem so gestaltet werden, dass die spätere Einarbeitung neuer Testaspekte keinen zu großen Aufwand verursacht. Die Testentwicklung sollte von vornherein zyklisch angelegt werden.

Für den Klassifikationsbaum und die Testfallbestimmung wurde eine graphi-

sche Repräsentation entwickelt. Auf Grundlage dieser arbeitet der *Classification Tree Edior*, ein graphisches Werkzeug zur Testfallentwicklung.

Die Abb.en 4.6–4.8 zeigen einen beispielhaften Klassifikationsbaum. Das hypothetische Testobjekt `Motorenkonfiguration` bildet die *Wurzel* des Baums. Diese kann als eine spezielle Klasse angesehen werden – die Klasse aller Daten, die gerade Eingabedaten für das Testobjekt sind. Die Eingabedaten des Testobjektes werden nach verschiedenen Gesichtspunkten klassifiziert. *Klassifikationen* werden als Texte in Rechtecken dargestellt. Im Beispiel sind das u.a. `Motorname`, `Motortyp` und auch `Indexschalter`. *Klassen* werden als Text ohne Rechtecke dargestellt. So ist z.B. die Klassifikation `Motorname` in die Klassen `DF`, `DC` und `TL` zerlegt. Solche Klassen können, wie die Wurzel, entsprechend verschiedener Aspekte klassifiziert werden, wie das im Beispiel bei der Klasse `C-812ISA` der Fall ist.

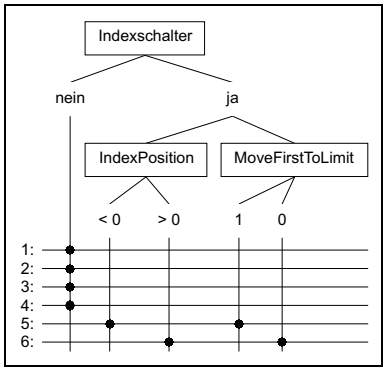

Abbildung 4.7: Beispiel für einen Klassifikationsbaum (2)

Aus Gründen der Übersicht kann es erforderlich werden, einen Klassifikationsbaum auf mehrere Abbildungen zu verteilen. Zu diesem Zweck wurden in der graphischen Repräsentation *Verfeinerungen (refinements)* eingeführt. *Verfeinerungen für Klassifikationen* werden durch zusätzliche senkrechte Linien gekennzeichnet, wie im Beispiel bei `Indexschalter` zu sehen. *Verfeinerungen für Klassen* sind an einer Unterstreichung zu erkennen, wie bei den Klassen `C-832` und `C-812ISA`. Verfeinerungen stellen keine Erweiterung des Modells des Klassifikationsbaums dar, sondern sind lediglich Platzhalter für Teile des Baumes, die an anderer Stelle abgebildet werden.

Der graphischen Repräsentation der *Testfallauswahl* dient die Kombinationstabelle. Jeder Testfall wird durch eine Zeile repräsentiert, in der durch Markierungen gekennzeichnet wird, welche Klassen den Testfall konstituieren. Der Testfall 1 im Beispiel ist demnach der Test für einen Motor der in der Motorenliste an erste Stelle steht, den Namen DF hat, vom Typ C-812ISA ist und keinen Indexschalter hat; die typabhängigen Parameter dieses Testfalles sind: `BoardID` aus dem Bereich 1–4 und `RamAddr` unspezifiziert.

Erfahrungen mit größeren Problemen haben gezeigt, dass die graphische Zerlegung gut funktioniert, aber die Übersicht über die ebenfalls zerlegte Kombinationstabelle beeinträchtigt wird.

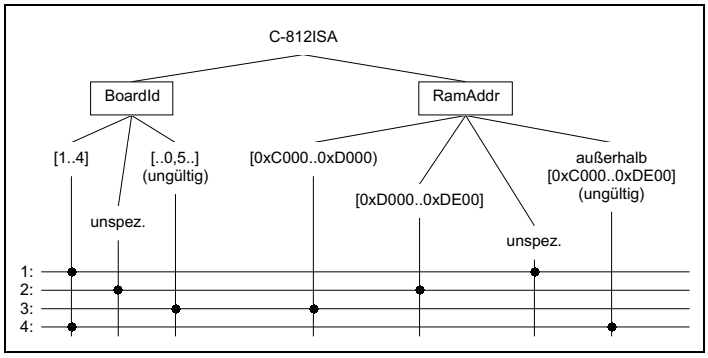

Abbildung 4.8: Beispiel für einen Klassifikationsbaum (3)

4.6 Testdatenauswahl und Testskriptgenerierung

Die Testdatenauswahl wird vom CTE durch den Export einer Testfallspezifikation unterstützt, einem halbformalen Text, der zu einem Testfall bzw. zu einem Testschritt angibt, welche Klassen der Eingabendaten ihn konstituieren. Das Beispiel aus dem Abschnitt 4.5 fortsetzend, zeigt Abb. 4.9 die Testfallspezifikation für den Testfall 1.

```
- Position in Motorliste : erste Stelle
- Motorname : DF
- Motortyp : C-812ISA
    - BoardId : [1..4]
    - RamAddr : unspez.
- Indexschalter : nein
```

Abbildung 4.9: Beispiel-Testfallspezifikation des CTE

Eine solche Testfallspezifikation kann die Testdatenauswahl bei der manuellen Erstellung von Testprogrammen unterstützen. Aus genannten Gründen habe ich aber nach einer automatisierbaren Form der Testskripterzeugung gesucht. Ausgangspunkt dieser Suche war die Möglichkeit mit Hilfe des CTE sowohl die einzelnen Elemente des Klassifikationsbaums als auch die Testfälle, -sequenzen und -schritte mit Attributen versehen zu können, sowie die leicht verständliche Struktur der CTE-Dateien.

Ausgehend von diesen Möglichkeiten waren die Überlegungen folgende:

Grundlage der Testsskripterzeugung sind die Testdaten. Das sind unter anderem Eingabedaten aus Dateien, Namen und Parameter von Funktionen, die aufzurufen sind, sowie deren Reihenfolge. In unserem konkreten Fall sind die Testdaten zum einen Parameter in den Konfigurationsdateien und zum anderen die Namen und Parameter der Funktionen der `m_layer.h`-Schnittstelle.

Die Angaben über die Testdaten stecken, bisher nur implizit, in den Klassen des Klassifikationsbaums, in dem ja gerade die Eingabedaten klassifiziert werden. Ein Testfall bzw. ein Testschritt wird durch eine Teilmenge eben dieser Klassen konstituiert. Wenn diese Klassen die erforderlichen Informationen über

die Testdaten in geeigneter Form bereitstellen, muss es möglich sein, daraus ein Skript zu erzeugen.

Hat man nun die Möglichkeit die Knoten des Klassifikationsbaums mit Attributen zu versehen, kann die erforderliche Information aus der Menge aller Attribute aller Klassen, die einen Testfall bzw. einen Testschritt konstituieren, gewonnen werden.

Die den Testfall bildenden Klassen sind Blätter des Klassifikationsbaums, d.h. über eine Art Vererbungsmechanismus können entlang des Pfades vom Blatt zur Wurzel des Baumes weitere Informationen gewonnen werden, die als Attribute an den Elementen dieses Pfades vorhanden sind.

Als letzte Überlegung schien es sinnvoll, einfache Berechnungen mit den Attributen zu ermöglichen, um z.B. Abhängigkeiten zwischen den Klassen verschiedener Klassifikationen des Baumes formulieren zu können.

Ist der Weg der Beschaffung der Informationen für einen Testfall bzw. einen Testschritt festgelegt, bedarf es nur noch einer Konvention für bestimmte Attribute um ein Skript generieren zu können, d.h. einer Spezifikation von Namen bestimmter Attribute, die ein Skriptgenerator für die Erzeugung des Skriptes verwendet.

Im Folgenden soll zuerst die Ermittlung der Attribute eines Testfalls oder eines Testschrittes erläutert und dann das konkrete System der Skriptgenerierung beschrieben werden.

Ermittlung der Attribute Bei der Ermittlung der Attribute eines Testfalls oder eines Testschrittes gelten folgende Regeln:

1. Elemente des Klassifikationsbaums erben Attribute ihrer Vorfahren. Ererbte Attributdefinitionen können überschrieben werden.

2. Attribute eines Testfalls selbst überschreiben gleichnamige Attribute an den konstituierenden Klassen. An einem Testschritt sind außerdem die Attribute der Testsequenz sichtbar. Diese werden wiederum durch gleichnamige Attribute des Testschrittes überschrieben.

3. Attribute, deren Name mit einem %-Zeichen oder mit einem $-Zeichen beginnen, werden als Variablen verwendet.

4. Variablennamen werden in allen Attributwerten rekursiv durch ihren Wert ersetzt.

5. Variablennamen werden ebenso in Attributnamen ersetzt, außer wenn diese selbst Variablen sind.

6. Der Sichtbarkeitsbereich von %-Variablen erstreckt sich auf alle Klassen, die einen Testfall konstituieren. Der von $-Variablen ist auf die einzelne Klasse beschränkt[3].

7. Attribute deren Werte mit einem &-Zeichen beginnen, werden nach der Ersetzung aller Variablen als Perl-Ausdrücke evaluiert.

8. Die Vereinigung aller Attribute und aller %-Variablen der Klassen, die einen Testfall konstituieren, muss widerspruchsfrei sein.

[3] Die Praxis hat gezeigt, dass man auf die $-Variablen wohl verzichten kann.

9. Die Attributdefinitionen müssen schleifenfrei sein.

Anhand des folgenden Beispiels sollen diese Regeln erläutert werden.

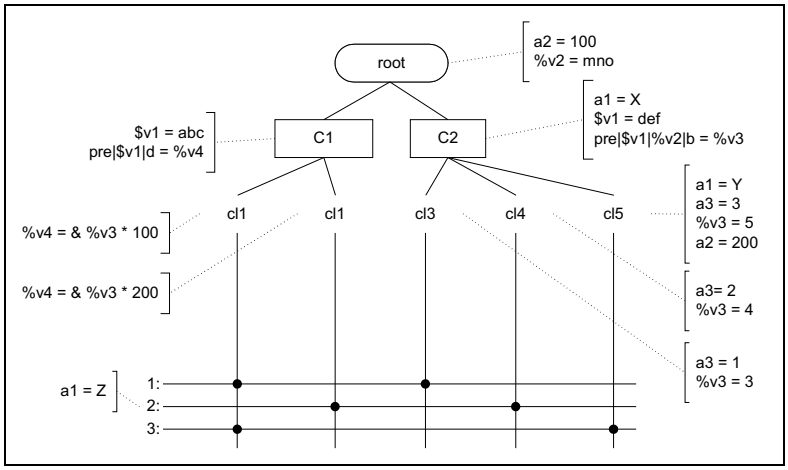

Abbildung 4.10: Beispiel für einen attributierten Klassifikationsbaum

Im Testfall 1 der Abbildung, der durch die Klassen `cl1` und `cl3` konstituiert wird, gilt u.a.

`a3=1` Die Definition kommt direkt von der Klasse `cl3`.

`a1=X` Die Klasse `cl3` erbt diese Definition von der Klassifikation `C2`.

`pre|def|mno|b=3` Die Definition `pre|$v1|%v2|b = %v3` wird über die Klasse `cl3` und die Klassifikation `C2` erreicht. Im Kontext von `cl3` wird dann zunächst versucht die $-Variablen aufzulösen; `$v1` wird bei `C2` gefunden. Dann wird im Kontext von Testfall 1 nach Definitionen für die %-Variablen gesucht; die für `%v2` wird bei `root` und die für `%v3` bei `cl3` selbst gefunden.

`pre|abc|d=300` Über die Klasse `cl1` und die Klassifikation `C1` steht die Definition `pre|$v1|d = %v4` zur Verfügung. `$v1` wird, im Kontext von `cl1` suchend, bei `C1` aufgelöst. Für `%v4` gibt es eine Definition bei `cl1`. Diese wiederum verwendet in ihrer Definition `%v3`, deren Wert über die andere den Testfall konstituierende Klasse `cl3` gefunden wird. Der Wert der Definition von `%v4` wird evaluiert und ergibt 300.

Am Testfall 2 gilt Folgendes:

`a1=Z` Hierbei überschreibt die Attributdefinition am Testfall selbst die über `cl4` und `C2` erreichbare Definition (s. Regel 2).

Für Testfall 3 ist folgendes wichtig:

`a2=100` **und** `a2=200` Hier liegt eine Verletzung der Regel 8 vor. Über `cl1`, `C1` und `root` ist für `a2` die Definition `a2=100` zu finden, über `cl5`, die die ererbte Definition überschreibt, aber steht die Definition `a2=200` zur Verfügung. Dies ist eine unerlaubte Mehrdeutigkeit.

Zu beachten ist, dass die unterschiedlichen Definitionen von `$v1` nicht mehrdeutig sind, da die Sichtbarkeit von $-Variablen eingeschränkt ist (s. Regel 6).

`a1=Y` Die Überschreibung des Attributs `a1` durch die Klasse `cl5` führt dagegen nicht zu Mehrdeutigkeiten.

Skriptgenerierung Das in Perl geschriebene System zur Testskriptgenerierung hat folgende, in Abb. 4.11 dargestellte, Struktur. Die Details der Implementation der einzelnen Komponenten sind im Anhang C dokumentiert.

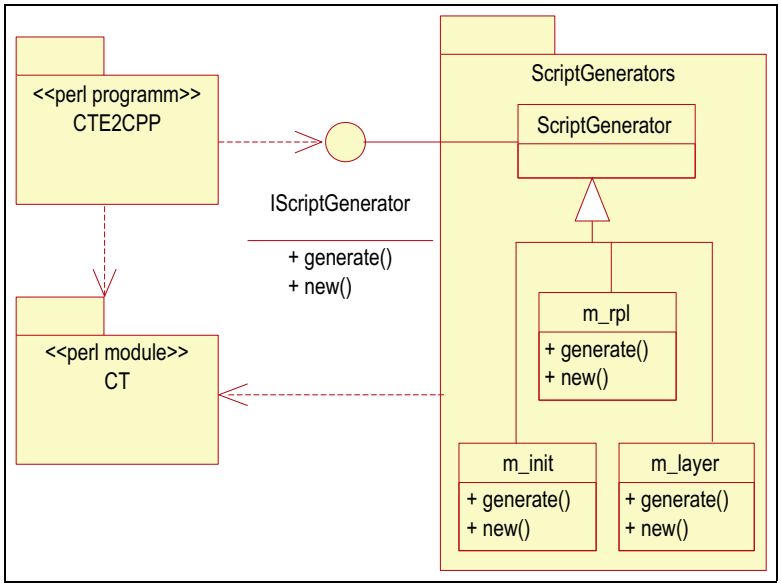

Abbildung 4.11: Struktur der Implementation der Testskriptgenerierung

Das Paket `CT` ist eine Implementation des Klassifikationsbaummodells. Es realisiert den Import von CTE-Dateien, einen XML-Export zu Dokumentationszwecken und die im vorherigen Abschnitt beschriebene Evaluierung der Attribute. Dieses Paket wird von dem Perl-Programm `CTE2CPP` verwendet, das die Skriptgenerierung für einen Testfall bzw. eine Testsequenz aus einer CTE-Datei durchführt. Da für Testskripte keine allgemeingültige Form gefunden wurde, verwendet das Programm einen in der CTE-Datei oder der Kommandozeile zu spezifizierenden Skriptgenerator. Von diesem wird nur die Implementation der Methoden `new` und `generate` erwartet (Interface `IScriptGenerator`). Im aktuellen System gibt es für jedes Testpaket eine Generator-Klasse: `m_init`, `m_layer` und `m_rpl`. Um gleiche Funktionalitäten in den Skriptgeneratoren wiederverwenden zu können, wurde die Klasse `ScriptGenerator` eingeführt von der die eigentlichen Skriptgeneratoren abgeleitet werden.

Die Verbindung zwischen einer CTE-Datei und dem dazugehörigen Skriptgenerator erfolgt über das Attribut `ScriptGenerator` des Wurzel-Objektes des Klassifikationsbaums.

Zur Identifikation muss jeder Testfall bzw. jede Testsequenz ein Attribut `test_id` besitzen, das die entsprechende Test-ID festlegt.

Wie bereits erwähnt, bedarf es einiger Festlegungen bzgl. der Attribute, die die konkreten Generatoren für die Skriptgenerierung verwenden. Hierbei werden so genannte *hierarchische Attribute* benutzt – Attribute deren Namen in Stufen zerlegt sind, um in den vom CTE angebotenen Name-Wert-Paaren komplexere Strukturen ablegen zu können. **Die Skriptgeneratoren verwenden folgende Attribute:**

`ini|<datei>|<abschnitt>|<parameter> = <wert>`
: Testdaten, die als Konfigurationsparameter in INI-Dateien abzulegen sind, werden in `ini`-Attributen spezifiziert. Dabei steht z.B. die Attributdefinition

```
ini|hardware.ini|Motor0|Name = Tilt
```

für folgenden Eintrag in der Konfigurationsdatei `hardware.ini`:

```
[Motor0]
Name=Tilt
```

Der spezielle Wert `NULL` steht für einen Parameter, der nicht in der Konfigurationsdatei vorkommt.

`step|<label> = <code>`
: Um die einzelnen Aktionen zu kodieren, die während eines Testschrittes ausgeführt werden sollen, werden die `step`-Attribute verwendet. Hierbei steht `<label>` für irgendeine Zeichenkette, anhand derer mehrere `step`-Attribute sortiert werden können. `<code>` steht für ein Zeichenkette, die ins Testskript übernommen wird. Z.B. führen folgende Attributdefinitionen

```
step|1 = retval = mlGetDistance(%axis, var);
step|2 = log << var;
step|3 = log << retval;
```

unter der Voraussetzung, dass `%axis` entsprechend definiert ist, zu folgenden Zeilen im Skript.

```
retval = mlGetDistance(DF, var);
log << var;
log << retval;
```

In der Praxis hat es sich als günstig erwiesen, hier etwas abstraktere Attributwerte zu verwenden. So kann man z.B. mit C++-Makros arbeiten. Meist sind für einen einzelnen Testschritt neben dem Aufruf einer Funktion oder Methode noch weitere Eintragungen im Testskript erforderlich, wie z.B. für die Protokollierung der Ein- und Ausgaben. Auch kann ein Testschritt eine *build*-abhängige Formulierung erfordern. Diese Angaben können bei der Verwendung von Makros getrennt und übersichtlicher in externen Dateien abgelegt werden. Änderungen im Testrahmen, bei der Protokollierung oder notwendige Änderungen für einen neuen *build* erfordern dann keine Änderungen in den Testfallspezifikationen.

`assert|<label> = <ausdruck>`
: Der letzte Attributtyp dient nicht der Angabe von Testdaten sondern der Formulierung von Zusicherungen. Nach ersetzen aller Variablen wird `<ausdruck>` als Perl-Ausdruck evaluiert. Im Fall, das das Ergebnis zu *false* evaluiert, d.h. zu `""`, `0` oder `"0"`, wird während der Skriptgenerierung eine Warnung ausgegeben. `<label>` dient dabei ausschließlich der Kennzeichnung der Zusicherung (wird in der Warnung mit ausgegeben) sowie der Unterscheidung verschiedener Zusicherungen.

Aufgabe dieser Attribute ist es den Testfallentwickler beim Entwurf konsistenter Testfälle zu unterstützen. Beispielsweise kann es vorkommen, dass einige Klassen des Klassifikationsbaums sich gegenseitig ausschließen. Für einen Testfall, der solche Kombinationen enthält, sind keine Testdaten zu bestimmen. Zum Teil lassen sich solche Widersprüche durch entsprechende Konstruktion des Klassifikationsbaums verhindern, oft aber nur mit einigem Aufwand und auf Kosten der Größe des Baumes. Sind solche sich ausschließenden Klassen im Baum vorhanden, ist eine automatische Überprüfung und ein entsprechender Hinweis für den Testentwickler sehr hilfreich.

Das `CT`-Modul stellt zu Berechnungszwecken folgende **vordefinierte Variablen** zur Verfügung.

`%test|step_nr`
: Diese Variable enthält die Position eines Testschrittes innerhalb einer Testsequenz. Der erste Testschritt einer Sequenz hat die Nummer 1. Diese Variable kann sinnvoll sein, wenn z.B. Attribute des ersten Testschrittes anders zu bestimmen sind als die der folgenden. Für Testfälle ist diese Variable nicht definiert.

`%test|pred_step|<variable>`
: Um Attribute eines Testschrittes bestimmen zu können, ist es öfters notwendig auf Informationen über die vorherigen Testschritte zugreifen zu können. Für diesen Zweck wurde diese Variable(ngruppe) eingeführt. `<variable>` steht dabei für den Namen einer %-Variable ohne das %-Zeichen. Der Wert der `%test|pred_step|<variable>`-Variable wird bestimmt, in dem in allen vorherigen Testschritten nach einer evaluierten Definition für die `%<variable>`-Variable gesucht wird. Die erste Definition die rückwärts gehend gefunden wird, wird verwendet. Wird keine Definition gefunden wird die Variable auf `undef` gesetzt. Die Verwendung soll an einem Beispiel verdeutlicht werden.

  Testschritt 1
  ```
  %Position: & (%test|step_nr == 1) ?
                  -2000 :
                  (defined(%test|pred_step|newPosition) ?
                       %test|pred_step|newPosition :
                       %test|pred_step|Position)
  ini|hardware.ini|Motor0|Position: %Position
  ```
  Testschritt 2
  ```
  %Position: <wie in Testschritt 1>
  %Distance = 2000
  %Direction = -1
  %newPosition = & %Position + (%Direction * %Distance)
  step|1: STEP_MOVE_TO_POSITION(%newPosition);
  ```
  Testschritt 3
  ```
  %Position: <wie in Testschritt 1>
  %Distance = 5000
  %Direction = 1
  %newPosition = & %Position + (%Direction * %Distance)
  step|1: STEP_MOVE_TO_POSITION(%newPosition);
  ```

Beispielhaftes Ergebnis

```
INI_ENTRY("hardware.ini", "Motor0", "Position", "-2000");
STEP_MOVE_TO_POSITION(-4000);
STEP_MOVE_TO_POSITION(1000);
```

Zum Abschluss dieses Kapitels soll nun noch ein **kleines Beispiel** vollständig dargestellt werden. Abb. 4.12 zeigt einen attributierten Klassifikationsbaum. Diese Abbildung führt gleichzeitig vor Augen, dass die Anzeige der Attribute in der graphischen Repräsentation der Klassifikationsbäume bei größeren Bäumen und einer größeren Anzahl von Attributen an Platzgrenzen stößt. Deshalb ist nach einer alternativen Darstellungsform gesucht worden, die im folgenden als *Textform* bezeichnet wird.

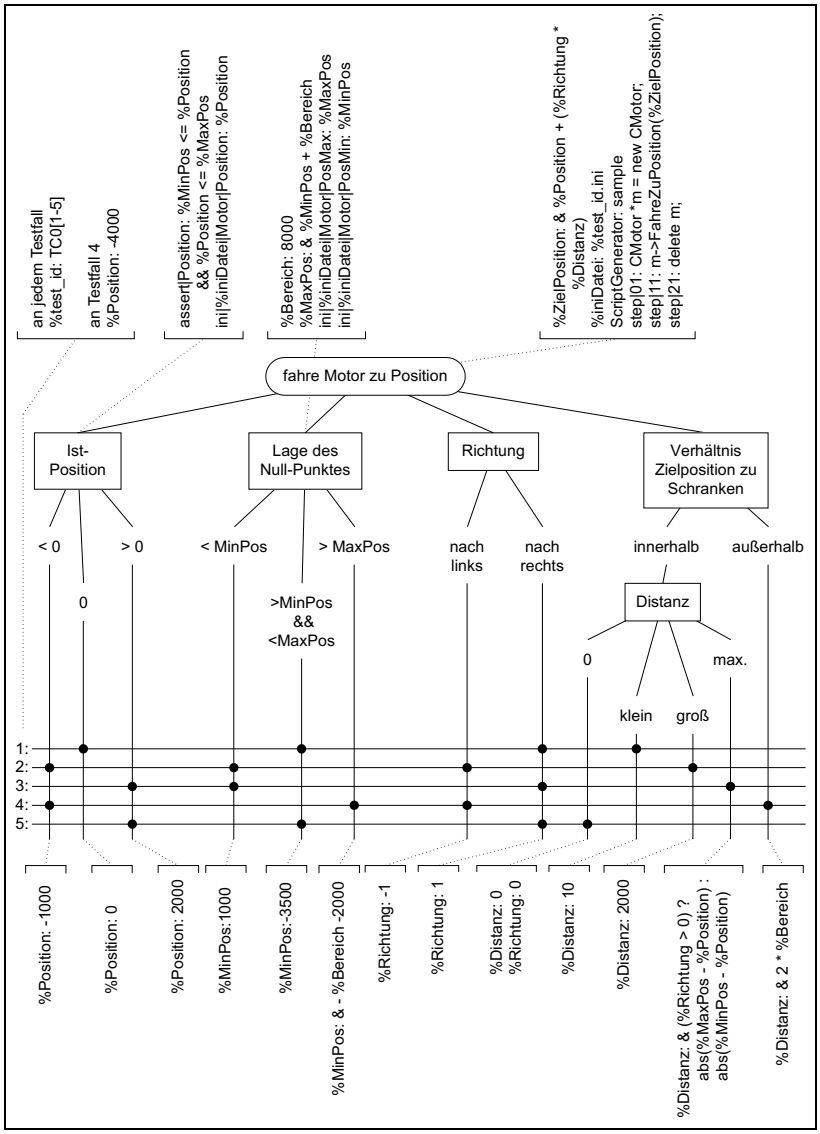

Abbildung 4.12: Beispiel für einen attributierten Klassifikationsbaum 2

In dem konstruierten Beispiel geht es darum, das Fahren eines Motors zu einer Position zu testen – eine Funktion die durch die Methode `FahreZuPosition` einer Klasse `CMotor` implementiert wird. Als Test-relevante Aspekte sind gefunden worden: die Lage des Koordinatensystem gegeben durch dessen Nullpunkt; die Ausgangs- bzw. Ist-Position des Motors; die Bewegungsrichtung; das Verhältnis der Zielposition zu gegebenen Softwareschranken sowie die zurückzulegende Distanz.

Es sei nun das zu testende System derart gestaltet, dass die Aspekte "Ist-Position" und "Lage des Nullpunktes" durch Konfigurationsparameter in einer ini-Datei abgebildet werden müssen. Diese Parameter heißen `Position` sowie `PositionMin` und `PositionMax` für den linken und rechten Rand des verfahrbaren Bereichs. Die anderen beiden Aspekte sind Aspekte des einzigen Parameter der zu testenden Methode, der die Zielposition angibt.

Zur Bestimmung von Testdaten für den `Positions`-Parameter ist an den Klassen unterhalb der Klassifikation `Ist-Position` jeweils eine Variable `%Position` definiert worden. Damit wird aus dem jeweiligen Wertebereichen, für den die Klassen stehen, ein Repräsentant ausgewählt. Die Variable `%Position` wird an der Klassifikation `Ist-Position` verwendet, um ein so genanntes `ini`-Attribut zu definieren[4]. Gleichzeitig wird dort eine Zusicherung formuliert, die überprüft, ob der Positions-Parameter stets innerhalb des gültigen Wertebereiches liegt[5].

Die Lage des Nullpunktes wird indirekt durch die beiden Schranken des verfahrbaren Bereiches gegeben. Diese werden durch die beiden Variablen `%MinPos` und `%MaxPos` bestimmt, deren Werte den Parametern `PositionMin` und `PositionMax` zugewiesen werden. Das geschieht in der Klassifikation `Lage des Null-Punktes`. Die untergeordneten Klassen definieren mit `%MinPos` die linke Schranke; die rechte ergibt sich aus der festgelegten Breite des verfahrbaren Bereichs (`%Bereich`) und kann daraus berechnet werden.

Der Aspekt "Verhältnis Zielposition zu Schranken" ist in zwei Klassen aufgeteilt: in Zielpositionen, die innerhalb der Schranken liegen und solche die außerhalb liegen. Die dem Aspekt untergeordneten Klassen definieren jeweils eine Variable namens `%Distanz`, die in der Wurzel des Baumes zusammen mit der `%Richtungs`-Variablen verwendet wird, um die Zielposition zu berechnen.

An der Wurzel des Baumes wird wie bereits erwähnt die Zielposition berechnet. Desweiteren werden dort drei `step`-Attribute definiert, die der Skriptgenerator verwendet um den variablen Teil des Testskriptes zu generieren. An dieser Stelle wird die Variable `ZielPosition` als Parameter für die zu testende Methode verwendet. Außerdem wird an der Wurzel die Variable `%iniDatei` definiert, die sowohl in den Definitionen der `ini`-Attribute Verwendung findet als auch in einem der `step`-Attribute. Diese Variable hat den Zweck für jeden Testfall eine separate Konfigurationsdatei zu erzeugen und diese dann dem Testobjekt zur Verfügung zu stellen; deshalb wird in ihrer Definition die Testfall-spezifische Variable `%test_id` verwendet.

Nicht zuletzt ist an der Wurzel dort auch das Attribut `ScriptGenerator` zu finden, das verwendet wird, die zum Testpaket passende Generator-Implemetation zu finden.

[4] Ein Attribut das ein Skriptgenerator verwenden kann um die Einträge in der ini-Datei zu produzieren; s. S. 92.

[5] Zu Attributen für Zusicherungen s. S. 92

Im Folgenden wird die Textform des Klassifikationsbaums aus Abb. 4.12 wiedergegeben. Danach werden einige der Testfälle bezüglich der konkreten Attributwerte betrachtet.

Klassifikationsbaum (Textform), vgl. Abb. 4.12.

fahre Motor zu Position

Attribute

%ZielPosition: & %Position + (%Richtung * %Distanz)
%iniDatei: %test_id.ini
ScriptGenerator: sample
step|01: CMotor *m = new CMotor('%iniDatei');
step|11: m->FahreZuPosition(%ZielPosition);
step|21: delete m;

Ist-Position

Attribute

assert|Position: %MinPos <= %Position && %Position <= %MaxPos
ini|%iniDatei|Motor|Position: %Position

< 0

Beschreibung

Links des Null-Punktes.

Attribute

%Position: -1000

0

Beschreibung

Auf Null-Punkt.

Attribute

%Position: 0

> 0

Beschreibung

Rechts des Null-Punktes.

Attribute

%Position: 2000

Lage des Null-Punktes

Attribute

%Bereich: 8000
%MaxPos: & %MinPos + %Bereich
ini|%iniDatei|Motor|PositionMax: %MaxPos
ini|%iniDatei|Motor|PositionMin: %MinPos

< MinPos

Beschreibung

Null-Punkt liegt links der linken Schranke.

Attribute

%MinPos: 1000

> MaxPos

Beschreibung

Null-Punkt liegt rechts der rechten Schranke.

Attribute

%MinPos: & - %Bereich -2000

> MinPos && < MaxPos
Attribute
%MinPos: -3500

Richtung

nach links
Attribute
%Richtung: -1

nach rechts
Attribute
%Richtung: 1

Verhältnis Zielposition zu Schranken

innerhalb

Distanz

Beschreibung
Entfernung der Zielposition

0
Attribute
%Distanz: 0
%Richtung: 0

klein
Attribute
%Distanz: 10

groß
Attribute
%Distanz: 2000

max.
Beschreibung
genaues Anfahren der Schranke in Fahrtrichtung
Attribute
%Distanz: & (%Richtung > 0) ? abs(%MaxPos - %Position) :
abs(%MinPos - %Position)

außerhalb
Attribute
%Distanz: & 2 * %Bereich

Testfall 1 wird durch folgende vier Klassen konstituiert:

- / Ist-Position / 0
- / Lage des Null-Punktes / > MinPos && < MaxPos
- / Richtung / nach rechts
- / Verhältnis Zielposition zu Schranken / innerhalb / Distanz / klein

Daraus ergibt sich die in Abb. 4.13 dargestellte Menge von Attributen, die an diesem Testfall sichtbar sind. Die Ebenen der hierarchischen Attribute (`ini`, `step`, `assert`) sind in der Abbildung durch Einrückung dargestellt. Im ersten Schritt der Attributbestimmung werden die Definitionen der Attribute der konstituierenden Klassen eingesammelt und im zweiten Schritt folgt dann im Kontext dieser Menge von Definitionen die Bestimmung der Werte der Attribute.

Betrachtet man im Anschluss an die Evaluierung die Attribute und ihre

Attribut	**Definition**	**Wert**
%Bereich	8000	8000
%Distanz	10	10
%MinPos	-3500	-3500
%MaxPos	& %MinPos + %Bereich	4500
%Position	0	0
%Richtung	1	1
%ZielPosition	& %Position + (%Richtung * %Distanz)	10
%test_id	TC01	TC01
%iniDatei	%test_id.ini	TC01.ini
assert		
Position	%MinPos <= %Position && %Position <= %MaxPos	-3500 <= 0 && 0 <= 4500
ini		
TC01.ini		
Motor		
Position	%Position	0
PositionMax	%MaxPos	4500
PositionMin	%MinPos	-3500
step		
01	CMotor *m = new CMotor('%iniDatei');	CMotor *m = new CMotor('TC01.ini');
11	m->FahreZuPosition(%ZielPosition);	m->FahreZuPosition(10);
21	delete m;	delete m;

Abbildung 4.13: Konstituierende Attribute des Testfall 1

Werte, sieht man leicht, dass sich daraus der variable Teil eines Testskriptes oder einer ini-Datei generieren lässt. Eine sehr direkte Übersetzung könnte z.B. wie in Abb. 4.14 aussehen. Die ersten 6 Zeilen schreiben die erforderlichen Parameter in die Konfigurationsdatei und die letzten drei Zeilen führen den Test durch und sind direkt durch Übernahme der Werte der `step`-Attribute gebildet worden.

```
WritePrivateProfileString("Motor",
                          "Position", "0", "TC01.ini");
WritePrivateProfileString("Motor",
                          "PositionMax", "4500", "TC01.ini");
WritePrivateProfileString("Motor",
                          "PositionMin", "-3500", "TC01.ini");
CMotor *m = new CMotor("TC01.ini");
m->FahreZuPosition(10);
delete m;
```

Abbildung 4.14: Beispielhafter Testskriptausschnitt für Testfall 1

Das Zusicherungsattribut `assert|Position` enthält als Wert einen logischen Ausdruck, der im Fall von Testfall 1 zu *wahr* evaluiert. Im Testfall 2 dagegen evaluiert er zu *falsch*, da dort folgende Attributdefinitionen gültig

sind: `%Position:-1000`, `%MinPos:1000`, `%MaxPos:& %MinPos + %Bereich` also somit `%MaxPos:9000` und `assert|Position:1000 <= -1000 && -1000 <= 9000` gilt. Ein Skriptgenerator sollte also bei diesem Testfall eine Warnung über das Verletzten der Zusicherung `Position` geben.

Bei Testfall 4 liegt eine ähnliche Situation vor. Hier gelten u.a. folgende Definitionen: `%MinPos:-10000` und `%MaxPos:-2000`. Über die Klasse "/Ist-Position/<0" ist die Definition `%Position:-1000` erreichbar. Diese 3 Definitionen würden wiederum die Zusicherung fehlschlagen lassen. Aber in diesem Testfall wird die Definition von `%Position` durch die Definition eines gleichnamigen Attributs am Testfall selbst überschrieben. Also gilt am Testfall 4 `%Position:-4000` und damit wird die Zusicherung erfüllt.

In Testfall 5 liegt ein Beispiel für eine mehrdeutige Attributdefinition vor. Hier soll getestet werden, wie sich das Motoren-Objekt verhält, wenn die Zielposition gleich der Istposition ist. Daher ist die Klasse ".../Distanz/0" in der Menge der konstituierenden Klassen enthalten. Das eine solche Bewegung keine Richtung hat, wollte der Testfallentwickler durch die Variablendefinition `%Richtung:0` darstellen. Über die Klasse "/Richtung/nach rechts" steht am Testfall 5 aber auch die Definition `%Richtung:1` zur Verfügung. Das ist ein Widerspruch, der zu einem Fehler bei der Skriptgenerierung führt. Wie solche Widersprüche aufzulösen sind, soll hier nicht weiter diskutiert werden. Möglichkeiten währen: die Marke an der Klasse "/Richtung/nach rechts" entfernen, die Definition `%Richtung:0` löschen oder die Einführung einer neuen Klasse "/Richtung/keine" die dann die Definition `%Richtung:0` tragen könnte.

4.7 Testauswertung (Soll/Ist-Vergleich)

In diesem Abschnitt soll der *Vergleicher* beschrieben werden, d.h. das Werkzeug zur Testauswertung, das durch Abgleich der Ist-Daten mit den Soll-Daten die Korrektheit des Programmverhaltens bestimmt.

Der Vergleich findet in diesem System *offline* statt, d.h. die zu überprüfenden Ausgabedaten (Ist-Daten) des zu testenden Systems werden gesammelt und *nach* dem eigentlichen Testlauf mit den Soll-Daten verglichen. Als erstes sollen dazu die Struktur der Informationen erläutert werden die a) über die Ist-Daten während des Testlaufes gesammelt werden müssen und b) über die Soll-Daten vorliegen müssen. Dann wird das konkrete Datenformat erläutert, in dem die Daten gespeichert werden, und abschließend ein Überblick über die Implementation des Vergleichers gegeben.

Ist-Daten Das Verhalten des Testobjektes wird durch die Eingabedaten bestimmt und ist anhand der Ausgabedaten, die vom Testobjekt erzeugt werden, überprüfbar. Demzufolge ist es, für die automatische Bestimmung der Korrektheit, ausreichend, während des Testlaufs die Ausgabedaten in der Reihenfolge ihrer Erzeugung im Testprotokoll abzuspeichern. Für die manuelle Aus- und Bewertung dagegen ist dies zu wenig. Für die Nachvollziehbarkeit des Testablaufes sollen deshalb zusätzlich die Eingabedaten im Protokoll festhaltbar sein. Außerdem soll es möglich sein, zusätzliche Informationen und Kommentare einbetten zu können, sowie die Ausgaben zu strukturieren.

```
<test-log>  : <test-id> <title> <date> (<test-step>)*
<test-step> : <number> <title>
              (<input> | <output> | <infos>)*
<input>     : <type>? <src>? text
<output>    : <type>? text
<infos>     : <comment> <error> <warn> <extension>
```

Abbildung 4.15: Struktur der Ist-Daten (Testprotokoll)

In Abb. 4.15 wird die aus diesen Überlegungen folgende Struktur der Soll-Daten schematisch dargestellt. Ein Testprotokoll (`<test-log>`) enthält als Metainformationen die Test-ID des Testfalles bzw. der -sequenz, deren Namen bzw. Titel, das Datum der Testausführung sowie eine Liste von Abschnitten, die die Ausgaben gliedern (`<test-step>`). Die einzelnen Abschnitte besitzen ebenfalls wiederum einen Titel und eine Nummer zur Identifikation und enthalten dann die Folge der Ein- und Ausgaben des Programmes sowie Zusatzinformationen.

Die Eingaben (`<input>`) besitzen neben dem Text der eigentlichen Daten noch Informationen über die Quelle dieser Daten sowie deren Typ. Als Quellen kommen, neben dem Testtreiber, auch Komponenten aus der Umgebung der zu testenden Anwendung infrage, wie z.B. Hardware- oder Simulationskomponenten. Die Ausgaben (`<output>`) des Testobjektes enthalten ebenso die Daten als Text und Typ-Angaben. Verschiedene Ausgabe-Typen sind z.B.: Ausgabe ist Rückgabewert einer Funktion, ist Inhalt eines Var-Parameters oder ist der Inhalt einer Datei.

Zusatzinformationen werden unterschieden in Kommentare, Fehler- und Warnmeldungen. Die Kommentare kann z.B. der Testrahmen einbetten um

die Lesbarkeit des Protokolls zu verbessern. Fehlermeldungen und Warnungen können unter Umständen von der Umgebung des Testprogrammes erzeugt werden, für die die Ausgaben des Testobjektes Eingaben darstellen, und die diese bzgl. ihrer Korrektheit testen können. Im entworfenen Testsystem z.B. testet die Simulationskomponente die Steuerungsparameter die das Testobjekt erzeugt auf Einhaltung bestimmter Wertebereiche, bzw. auf Einhaltung einfacher Protokollbedingungen. Werden diese verletzt, werden Fehler protokolliert.

Weitere nicht direkt zum Testsystem gehörige Zusatzinformationen sollen ebenfalls in das Protokoll integrierbar sein (`<extension>`). Im konkreten Testsystem z.B. protokolliert die Motorensimulation Informationen über den Motorenzustand und über die Interpretation der Kommandos (s. Abschnitt 3.4.5).

Der konkrete Mechanismus zur Einbettung solcher Erweiterungen ist bei der Weiterentwicklung des Testsystems nochmals zu überdenken. Sie werden im aktuellen Testsystem nicht maschinell ausgewertet, dienen im Moment nur der Dokumentation und dem besseren Verständnis der getesteten Vorgänge. Das selbe gilt für die Angaben zu Quelle und Type der Ein- und Ausgaben.

Soll-Daten Zur Spezifikation der Soll-Daten werden prinzipiell die gleichen Informationen benötigt, wie zum Festhalten des Ist-Verhaltens. Einige Erweiterungen sind jedoch notwendig um a) *baseline*- und *targetline*-Informationen zusammen ablegen zu können, und b) Unschärfen in den Soll-Daten zulassen zu können. Die erweiterte Struktur ist in der folgenden Abb. 4.16 zu sehen.

```
<test-bas>  : <test-id> <title> <date>? (<test-step>)*
<test-step> : <number> <title>
              (<input> | <output> | <output-seq> | <infos>)*
<input>     : <src>? text
<output>    : <type>? <comp>? <context>? (text | <regexp>)*
<output-seq>: <min-occur> <max-occur>
              (<input> | <output> | <output-seq> | <infos>)*
<infos>     : <comment> <error> <warn> <extension>
```

Abbildung 4.16: Struktur der Soll-Daten

Um Punkt a) zu realisieren, wurde die Definition von `<output>` um das Attribut `<context>` erweitert. Dieses erhält entweder den Wert `baseline` oder den Wert `targetline`, falls der entsprechende Ausgabewert nur zu einem der Soll-Daten-Typen gehört.

Bei Punkt b) muss zwischen zwei verschiedenen Arten der Unschärfe unterschieden werden. Erstens der Unschärfe, die entsteht, wenn ein Ausgabewert aus einem bestimmten Wertebereich zu erwarten ist. Zweitens kann es sein, dass bestimmte Steuerungsabläufe Zyklen durchlaufen, bei denen die Anzahl der Durchläufe nicht exakt vorhersehbar ist und die somit Zyklen in den Ausgabedaten produzieren, deren Länge variieren darf.

Für Unschärfen der ersten Art wurde wiederum die Definition von `<output>` erweitert, und zwar um das `<comp>`-Attribut und um in die Daten einbettbare `<regexp>`-Informationen. Diese ermöglichen es, einen Abschnitt der Soll-Ausgabedaten bzw. einen Teil davon in Form eines regulären Ausdrucks zu spezifizieren. Damit sind Mengen von zulässigen Werten beschreibbar. Auch

numerische Wertebereiche lassen sich so spezifizieren, auch wenn dafür sicher günstigere Varianten denkbar sind, die aber hier nicht realisiert werden konnten.

Unschärfen des zweiten Typs werden durch die Definition von Ausgabe-Sequenzen (`<output-seq>`) ermöglicht. Ausgabe-Sequenzen dürfen überall dort erscheinen, wo auch normale Ausgaben auftreten. Zwei Attribute (`min-occur`, `max-occur`) spezifizieren den Bereich, in dem die erwartete Anzahl der Zyklen liegt. Ausgabe-Sequenzen können wiederum alles enthalten, was ein `<test-step>` enthalten kann, einschließlich weiterer Ausgabe-Sequenzen.

Datenformat Konkret werden die Soll- und Ist-Daten in XML-Dateien abgelegt, die entsprechend den oben beschriebenen Anforderungen strukturiert sind. Abb. 4.17 zeigt die entsprechende Dokumenttypdefinition (DTD), in der aber, wie oben erwähnt, noch kein Mechanismus zur Einbettung von Zusatzinformationen integriert ist.

```
<!ELEMENT test-log (title, date, test-step*)>
<!ATTLIST test-log id CDATA #REQUIRED>

<!ELEMENT test-bas (title, date, test-step*)>
<!ATTLIST test-bas id CDATA #REQUIRED>

<!ELEMENT test-step (title,
                     (input|output|output-seq|comment|error)*)>
<!ATTLIST test-step number CDATA #IMPLIED>

<!ELEMENT input  (#PCDATA)>
<!ATTLIST input  type (file) #IMPLIED>
<!ATTLIST input  src (hw|sim) #IMPLIED>

<!ELEMENT output (#PCDATA)>
<!ATTLIST output type (return|varparam|file) #IMPLIED>
<!ATTLIST output context (all|baseline|targetline) 'all'>
<!ATTLIST output comp (regexp) #IMPLIED>

<!ELEMENT output-seq (input|output|output-seq|comment|error)*>
<!ATTLIST output-seq min-occur CDATA #REQUIRED>
<!ATTLIST output-seq max-occur CDATA #REQUIRED>

<!ELEMENT title   (#PCDATA)>
<!ELEMENT date    (#PCDATA)>
<!ELEMENT comment (#PCDATA)>
<!ELEMENT error   (#PCDATA)>
```

Abbildung 4.17: Dokumenttypdefinition der Soll- und Ist-Daten

Für die direkte Visualisierung der Soll- und Ist-Daten habe ich einen XSL-Stylesheet geschrieben, d.h. die Beschreibung einer Transformation der Daten aus dem eben beschriebenen Modell in ein Visualisierungs-Modell, konkret in ei-

ne HTML-Struktur. Diesen standardisierten Stylesheet verwenden u.a. Internet-Browser um XML-Daten anzuzeigen. Aufgrund des Standes der Implementation des XML- und XSL-Standards konnte die Visualisierung bisher nur mit dem *Microsoft Internet Explorer* getestet werden

Implementation des Vergleichers Der im Perl implementierte Vergleicher hat eine einfache Komponentenstruktur wie in Abb. 4.18 dargestellt.

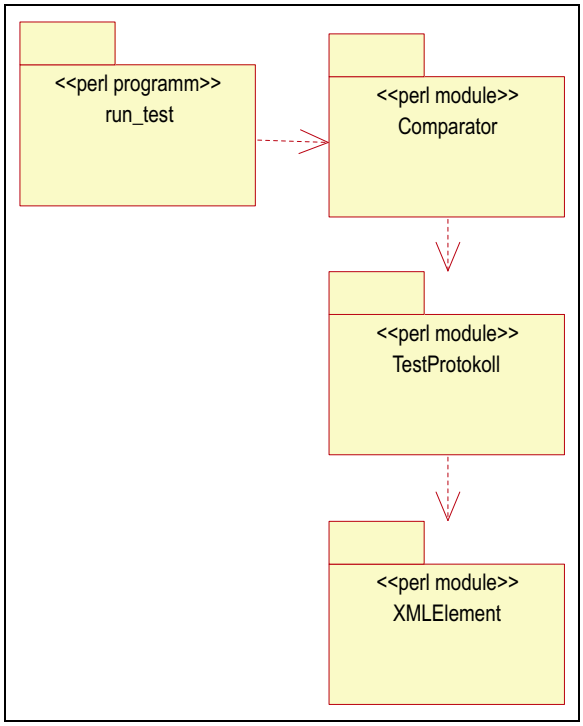

Abbildung 4.18: Struktur der Implementation des Vergleichers

Für das Einlesen von XML-Daten wird das Modul `XMLElement` verwendet. Dieses Modul nutzt einen in Perl integrierten XML-Parser und erzeugt mit dessen Hilfe aus XML-Dateien allgemeine Objektstrukturen, die die Auswertung der Daten ermöglichen. Dieses Modul entstand in früheren Projekten und wird hier (nur) wiederverwendet. Daher wird auf dieses Modul nicht weiter eingegangen.

Das Modul `TestProtokoll` spezialisiert die allgemeinen Objektstrukturen des `XMLElement`-Moduls für die Anforderungen für das Auswerten der Soll- und Ist-Datendateien. Es realisiert im Wesentlichen die Extraktion der für den Vergleich relevanten Ausgabedaten sowie deren Transformation in einen regulären Ausdruck.

Der eigentliche Vergleich wird durch das `Comparator`-Modul durchgeführt. Es werden stets die Informationen eines Testsschrittes (`<test-step>`) als Einheit getestet. Das erfolgt, in dem 1. aus den Ist-Daten die Ausgabedaten (`<output>`) extrahiert werden und 2. aus den extrahierten Ausgabedaten (`<output>`) der Soll-Daten ein regulärer Ausdruck gebildet wird. Im 3. Schritt

wird dann überprüft, ob die Ist-Daten innerhalb des durch den regulären Ausdruck beschriebenen Bereiches liegen. Ist dies der Fall, gilt der Testschritt als erfolgreich. Ein Test gilt als erfolgreich, wenn alle seine Testschritte erfolgreich waren.

Sind in den Soll-Daten Unterschiede zwischen der *baseline* und der *targetline* verzeichnet, erfolgt der eben beschriebene Vergleich zweimal. Die Ergebnisse der Überprüfung der einzelnen Testschritte werden in einer *Ergebnisdatei* für den Testfall bzw. die Testsequenz gesammelt.

Ist ein einzelner Testfall nicht erfolgreich, werden die Unterschiede zwischen Soll- und Ist-Daten analysiert und in einer *Fehlerbeschreibungsdatei* festgehalten. Die Analyse basiert im derzeitigen Testsystem auf dem Werkzeug *diff*. Bei einfachen Soll-Daten-Strukturen funktioniert das auch sehr gut, bei komplexeren Soll-Daten, z.B. wenn dort mehrfach geschachtelte Ausgabe-Sequenzen auftauchen, sind die Ergebnisse noch nicht zufriedenstellend.

Das Programm `run_test` ist für die Auswertung der Kommandozeilenparameter, den Start des Vergleiches sowie für die abschließende Integration der Testergebnisse in die Gesamtdokumentation zuständig.

Eine detaillierte Dokumentation der einzelnen Komponenten ist in Anhang C zu finden.

4.8 Das Testsystem am Beispiel

In diesem Abschnitt sollen die einzelnen Testaktivitäten und die dabei verwendeten und entstehenden Dokumente an einem Beispiel erläutert werden. Dabei dient die Abb. 4.1 auf Seite 80 als Leitfaden. Um ein möglichst realistisches Beispiel zu wählen, wird das Testpaket `m_rpl` verwendet, das die Funktion „F2 Grundstellung anfahren“ der Motorenkomponente testet, und dessen Dokumentation auf Seite 165 zu finden ist.

4.8.1 Testfallentwicklung

Erstellung des Klassifikationsbaums Die erste Testaktivität ist die Entwicklung der Testfälle. Zuerst ist ein Testobjekt auszuwählen, im Beispiel der Dialog „Grundstellung anfahren“. Als nächstes werden die Eingabedaten untersucht und alle Aspekte, die das Verhalten des Testobjektes beeinflussen, zusammengetragen[6]. Im Falle des vorliegenden Testobjektes sind diese Aspekte zum einen eine Reihe von Konfigurationsparametern und zum anderen die möglichen Anwender-Eingaben auf dem Dialog und deren Reihenfolge. Die Klassifikation der Eingabedaten führt zu einem Klassifikationsbaum. Der zum Beispiel gehörende Klassifikationsbaum ist auf Seite 167 abgebildet. Eine Textform dieses Baumes ist ebenfalls dort zu finden.

Testdatenauswahl 1 Nachdem die relevanten Eingabedaten klassifiziert wurden, müssen für die Ausführung konkreter Testfälle konkrete Testdaten ausgewählt werden. D.h, aus den gebildeten Klassen muss jeweils ein Repräsentant ausgewählt werden. Im vorliegenden Klassifikationsbaum gibt es z.B. die Klasse „..|InitialAngle|<0“. Um eine konkrete Konfigurationsdatei erzeugen zu können

[6] Für Details zur Klassifikationsbaumemethode s. Abschnitt 4.5 auf S. 86.

muss entschieden werden, welcher Wert aus der Klasse „ganze Zahlen kleiner Null“ als Testdatum verwendet werden soll.

Diese Auswahl muss für jede Klasse in einen Klassifikationsbaum genau einmal durchgeführt werden, da das Testdatum für alle Testfälle gleich sein sollte. Theoretisch kann man natürlich auch in jedem Testfall einen anderen Repräsentanten verwenden. Das führt aber dazu, dass implizit – nicht im Klassifikationsbaum erkennbar – spezielle Werte getestet werden, und so der Dokumentations- und Übersichtsaspekt der Methode ausgehebelt wird. Wenn der Tester bestimmte Werte einer Klasse testen will, sollte er diese Klasse im Klassifikationsbaum in weitere Klassen zerlegen und diesen die erforderlichen Testdaten zuweisen.

Die Testdatenauswahl orientiert sich also direkt am Klassifikationsbaum, d.h., es ist sinnvoll diese Auswahl auch direkt am Klassifikationsbaum unterzubringen. Dazu werden die Knoten des Klassifikationsbaums mit Attributen versehen. So wurde z.B. für die oben erwähnte Klasse „..|InitialAngle|<0“ ein Testdatum von -100 ausgewählt und die Klasse im Klassifikationsbaum mit dem Attribut `%InitialAngle=-100` versehen.

Im hier entwickelten Testsystem soll aus dem Klassifikationsbaum und der formalisierten Testdatenauswahl ein Testprogramm oder Testskript *generiert* werden. Dazu bedarf es vor der Testdatenauswahl einiger Vorüberlegungen hinsichtlich Form und Inhalt des zu generierenden Skriptes um die Testdaten in adäquater Art und Weise zu formalisieren.

Vorbereitung der Testskriptgenerierung Um später den Test durchführen zu können muss für jeden Testfall bzw. jede Testsequenz ein Testskript erzeugt werden, d.h. eine formale Beschreibung des Testablaufs, mit deren Hilfe die Testausführung automatisch erfolgt. Form und Inhalt eines solchen Skriptes sind für die verschiedenen Testobjekte unterschiedlich. Daher ist in dem Programm zur Testskriptgenerierung (`cte2cpp.pl` s. Anhang C) die Verwendung eines Testpaket-spezifischen Generators vorgesehen.

Im Beispiel-Testpaket ist das Testobjekt ein Dialog, der durch das Testprogramm „ferngesteuert“ werden soll, d.h., das Testprogramm sendet an den Dialog Nachrichten, die Anwender-Eingaben simulieren.

Da die Motorsteuerung unabhängig von den anderen Komponenten des XCTL-Systems getestet werden soll, und die Motorsteuerung eine DLL ist, musste noch ein einfaches Hilfsprogramm (`mdlg.exe`) geschrieben werden, das es ermöglicht, die einzelnen Dialoge der Motorsteuerung zu aktivieren und zu bedienen.

Die Testskripte sollen im vorliegenden Beispiel in C++ geschrieben sein und folgende allgemeine Struktur haben. In den Skripten wird sehr stark mit Präprozessor-Makros gearbeitet, um kurze und übersichtliche Skripte zu erhalten und um gemeinsame Definitionen in den Skripten zentral sammeln zu können.

```
1 #include "m_rpl.inc"
2
3 INI_BEGIN(ini_data)
4 INI_ENTRY(<Datei>, <Abschnitt>, <Name>, <Wert>)
5 INI_END
6
```

```
 7 TMAIN
 8 {
 9   T_INIT_FRAME(<Test-Id>, <Titel>);
10   TSTEP_RUN_MDLG(<Test-Id>, ini_data);
11
12   //-----------------------------------------
13
14  Anweisungen fuer die einzelnen Testschritte
15
16   //-----------------------------------------
17
18   T_CLEANUP_AND_RETURN;
19 }
```

Zeile 1: ermöglicht das Einfügen von Definitionen die in allen zu generierenden Skripten gemeinsam verwendet werden.

Zeilen 3-5: Makros die eine Struktur erzeugen, die alle Konfigurationsparameter für den Testfall enthält. D.h., ein konkretes Skript enthält für jeden Parameter eine Zeile analog zu Zeile 4.

Zeile 9: Makro, das die Initialisierung des Testrahmens, der Protokollierung und ähnliche vorbereitende Aufgaben realisiert.

Zeile 10: Makro, das die Erzeugung der Konfigurationsdatei entsprechend den oben gegebenen Informationen realisiert und das das Hilfsprogramm `mdgl.exe` startet und damit den zu testenden Dialog öffnet.

Zeile 14: Platzhalter für eine Reihe von Zeilen, die die einzelnen Aktionen der durchzuführenden Testsequenz realisieren. Für die verschiedenen Aktionen (s. im Klassifikationsbaum) sind jeweils Makros definiert worden, z.B.

`TSTEP_SELECT_MOTOR(<Schritt-Nr>,<Achse>)`
für die Aktion „Wähle Motor“,

`TSTEP_CLICK_RPL(<Schritt-Nr>)`
für die Aktion „Click auf RPL“,

`TSTEP_WAIT_FOR_MESSAGE(<Schritt-Nr>,<Text>,<Zeit>)`
für die Aktion „Warten|Worauf|Meldung“ und

`TSTEP_CLICK_CLOSE(<Schritt-Nr>)`
für die Aktion „Schließen|Wie|X“.

Zeile 18: Makro, das die ordnungsgemäße Beendigung des Tests und der Protokollierung realisiert.

Der Testskriptgenerator In den Testskripten sind also zwei Teile von den im Klassifikationsbaum untersuchten Eingabedaten abhängig: a) die Zeilen, die an stelle von Zeile 4 die Konfigurationsparameter angeben, und b) die Zeilen, für die Zeile 14 als Platzhalter dient und die Art und Parameter der einzelnen Aktionen beschreiben. Die dafür notwendigen Informationen korrespondieren gerade mit den im Klassifikationsbaum strukturierten Eingabedaten der Testfälle.

In der im Testsystem implementierten Basisklasse für Skriptgeneratoren ist die erforderlichen Funktionalität zum generieren der ini-Datenstrukturen und

der Testschritte bereits enthalten (`ScriptGenerators::ScriptGenerator`, s. Anhang C). Der zu schreibende Skriptgenerator (`m_rpl`) muss also nur von dieser Basisklasse abgeleitet werden und in der zu implementierenden `generate()`-Methode diese Funktionalität verwenden um das Skript zu erzeugen. Zur Einbindung des im folgenden abgebildeten Perl-Paketes in die Implementation des Testsystems s. Abb. 4.11 auf Seite 91.

```
 1 use strict;
 2 use ScriptGenerators::ScriptGenerator;
 3
 4 ############################################################
 5 package ScriptGenerators::m_rpl;
 6 @ScriptGenerators::m_rpl::ISA =
 7       qw(ScriptGenerators::ScriptGenerator);
 8 ############################################################
 9
10 sub generate {
11   my ($self, $testseq) = @_;
12
13   # test that $testseq is really an object of CT::TestSequence
14   warn('CT::TestSequence expected') unless
15        (ref($testseq) eq 'CT::TestSequence');
16
17   # check the assertion attributes of all test steps and
18   # print a warning if one fails
19   $self->evaluate_assertions_for_testseq($testseq);
20
21   # open the file to be generated, name it <test-id>.CPP
22   my $file_name = ($testseq->atts->{'%test_id'}
23                    || "unknown_test_id") . '.CPP';
24   open (OUT, ">$file_name")
25        || die qq(can't open $file_name ($!));
26
27   $self->testobj($testseq);
28   $self->output(*OUT);
29
30   # get the first test step in the sequence
31   my $first_step = $testseq->step(0);
32
33   # header
34   print OUT qq(\n#include "m_rpl.inc"\n\n);
35
36   # generate ini data from the attributes of the
37   # first test step (using a method from the base class)
38   $self->generate_ini_structure('ini_data',
39                                 $first_step->atts_evaluated);
40
41   # test steps in "int main()"
42   my $test_id = $testseq->atts->{'%test_id'};
43
```

```
44   print OUT qq(TMAIN\n{\n);
45   print OUT qq(\tT_INIT_FRAME\($test_id, "),
46               $testseq->name,
47               qq("\);\n\n);
48   print OUT qq(\tTSTEP_RUN_MDLG($test_id, ini_data);\n\n);
49
50   print OUT qq(\t//---------------------------------\n\n);
51   # generate the code given in the step-attributes at each
52   # test step in the sequence
53   $self->generate_teststeps_code($testseq->steps);
54   print OUT qq(\t//---------------------------------\n\n);
55
56   print OUT qq(\n\tT_CLEANUP_AND_RETURN;\n);
57   print OUT qq(}\n\n);
58
59   close OUT;
60 }
```

Der Skriptgenerator mit dem Namen `ScriptGenerators::m_rpl` ist ein Perl-Paket[7] und ist in der Datei `m_rpl.pm` im Verzeichnis des Testpakets `m_rpl` implementiert.

In Zeile 5-7 wird das Paket `ScriptGenerators::m_rpl` deklariert und festgelegt, dass es von `ScriptGenerators::ScriptGenerator` abgeleitet wird. Der Rest enthält die Implementation der Methode `generate` die als nulltes Argument das Generator-Objekt selbst erhält (`$self`) und als eigentliches Argument ein Objekt vom Typ `CT::TestSequence` erwartet (`$testseq`).

Als Erstes wird überprüft, ob das Argument vom richtigen Typ ist (Zeile 13-15). Dann werden alle Zusicherungen dahingehend getestet, ob sie innerhalb der Testsequenz erfüllt werden (Zeile 36-39). Als Drittes wird die Datei angelegt und geöffnet, in die das zu generierende Skript geschrieben werden soll (`OUT`, Zeile 21-25).

Ab Zeile 33 beginnt die eigentliche Generierung. In Zeile 36-39 werden mit Hilfe einer an der Basisklasse definierten Methode die Konfigurationsparameter ausgewertet und in die gewünschte Makro-Struktur überführt. Die Informationen über die Konfigurationsparameter werden den `ini`-Attributen des ersten Testschrittes entnommen. Hier werden also die Zeilen entsprechend Zeile 3-5 des Skript-Gerüstes auf S. 105 erzeugt. In Zeile 53 wird dann der Skript-Teil generiert, der die einzelnen Testschritte kodiert, d.h. hier wird der Inhalt von Zeile 14 des Skript-Gerüstes erzeugt.

Für eine detaillierte Erläuterung der in der `generate`-Methode verwendeten Methoden

- `evaluate_assertions_for_testseq`,
- `generate_ini_structure` und
- `generate_teststeps_code`

der Basisklasse `ScriptGenerators::ScriptGenerator` s. Anhang C. Die drei Methoden werten jeweils einen Attribut-Typ aus: die Zusicherungs- oder `assert`-Attribute, die `ini`-Attribute und die `step`-Attribute. Die Attribute sind im Einzelnen auf Seite 92f. erläutert. Diese drei Attribut-Typen können

[7] Pakete sind in Perl das Mittel zu Definition von Klassen.

nun zur Festlegung der Testdatenauswahl im attributierten Klassifikationsbaum verwendet werden.

Um es nochmals zusammenzufassen: der konkrete Skriptgenerator für das Testpaket `m_rpl` verwendet für die Erzeugung des Skriptes die `ini`-Attribute des *ersten* Testschrittes der Sequenz, um die für den Testlauf notwendigen Konfigurationsparameter zu bestimmen, und verwendet dann die `step`-Attribute *aller* Testschritte um die eigentliche Testsequenz zu kodieren.

Testdatenauswahl 2 Nach dem nun geklärt ist, welche Attribute der Skriptgenerator bei der Erzeugung der Testskripte verwendet, kann die Testdatenauswahl durchgeführt werden und in den entsprechenden Attributen der Elemente des Klassifikationsbaums gespeichert werden. Die Attribute des Klassifikationsbaums sind nur schlecht in der graphischen Repräsentation darzustellen. Darum ist in der Dokumentation des Testpakets auf Seite 165ff. auch eine textuelle Form des Baumes enthalten, in der an den Elementen des Baumes sowohl die Attribute als auch erläuternde Kommentare zu finden sind.

Um ein Beispiel für die Testdatenauswahl und die Attributierung zu geben, betrachte man die Klassifikation „Konfiguration|Beschleunigung“. Diese Klassifikation steht für den Testaspekt, dass ein Motor sich mit einer in einem Konfigurationsparameter gegeben Beschleunigung bewegen soll. Die Testdaten müssen also gerade den Inhalt dieses Parameters bestimmen. Aus diesem Grund ist die Klassifikation mit einem `ini`-Attribut versehen worden:

`ini|%ini_file|%ini_section|Acceleration = %Acceleration`.

In den untergeordneten Klassen wird die Variable `%Acceleration` mit den entsprechenden Werten der Testdatenauswahl belegt. Dort wurde festgelegt, dass eine „kleine“ Beschleunigung durch den Wert 10 repräsentiert werden kann, eine „normale“ durch den Wert 5000 und eine „große“ Beschleunigung durch den Wert 20000. Die ebenfalls in der Definition des Klassifikations-Attributs verwendeten Variablen `%ini_file` und `%ini_section` werden an anderen Stellen des Klassifikationsbaums definiert: `%ini_file` am Wurzelknoten und `%ini_section` in der Klassifikation „Konfiguration|Position in Liste“. An jedem Testfall, der eine der Klassen „kleine“, „normale“ oder „große“ Beschleunigung enthält, steht demzufolge (unter der Annahme, dass `%ini_file=hardware.ini` und `%ini_section=Motor0` gilt) eine der Attributdefinitionen

`ini|hardware.ini|Motor0|Acceleration = 10`,
`ini|hardware.ini|Motor0|Acceleration = 5000` oder
`ini|hardware.ini|Motor0|Acceleration = 20000`

zur Verfügung. Auf analoge Weise sind Testdaten für alle Klassen des Baumes auszuwählen und in Attributen abzulegen.

Testfallbestimmung Parallel zur Testdatenauswahl erfolgt mit Hilfe des Klassifikationsbaums die Festlegung der Testfälle. Dazu werden die Klassen des Baumes mit einander kombiniert[8].

Die folgenden Testaktivitäten sollen an einer konkreten Testsequenz erläutert werden. Die Testsequenz `RPL001` besteht aus 6 Testschritten, die jeweils durch folgende Klassen konstituiert werden[9]:

[8] Zur Klassifikationsbaumemethode s. Abschnitt 4.5 auf S. 86.

[9] Die Darstellung der konstituierenden Klassen erfolgt in der Form, in der sie der *Classification Tree Editor* liefert.

Schritt 1

```
- Konfiguration
   - Position in Liste: erster
   - Achse: DF
   - InitialMove: 1
   - Position: rechts RP
   - InitialAngle: 0
   - MotorType: C-812
   - Geschwindigkeit: normal
   - Beschleunigung: normal
   - IndexLine: 0
```

Schritt 2

```
- Aktion: Waehle Motor
   - Name: DF
```

Schritt 3

```
- Aktion: Click auf 'RPL'
```

Schritt 4

```
- Aktion: Warten
   - Worauf: Meldung
      - Art: Index XX
```

Schritt 5

```
- Aktion: Warten
   - Worauf: MessageBox
      - Antwort: X
```

Schritt 6

```
- Aktion: Schliessen
   - Wie: OK
```

Im ersten Schritt der Testsequenz werden die Parameter der für den Test zu verwendenden Konfigurationsdatei bestimmt. Der Motor, für den der Referenzpunktlauf getestet werden soll, ist also der erste in der Motorenliste, hat den Namen DF, ist vom Typ C-812 u.s.w.

Die nächsten Schritte legen die Eingaben des Anwenders fest: wähle zuerst den Motor DF in der Listbox aus, starte dann den Referenzpunktlauf durch Anklicken des entsprechenden Buttons, warte auf die Meldung „Index DF“, warte auf eine MessageBox und schließe diese über den x-Button und beende abschließend den Dialog mit Hilfe des OK-Buttons.

Soll-Daten-Bestimmung Im Falle eines *Forward Engineering*-Projektes kann nach der Bestimmung eines Testfalls und der Auswahl der Testdaten mit Hilfe der Spezifikation des zu testenden Programmes festgelegt werden, welche Ausgaben das Programm während der Testausführung zu erzeugen hat.

Die hier gewählte Vorgehensweise ist aber, wie im Kapitel 1 erläutert, dem *Reverse Engineering*-Zustand des XCTL-Projektes angepasst. D.h., dass die erste Instanz der Soll-Daten aus der ersten Testausführung des zu testenden Programmes gewonnen wird. Daher geht diese Erläuterung auf die Soll-Daten Abschnitt zur Testauswertung ein, der nach der Testausführung erfolgt.

4.8.2 Testprogrammerzeugung

Testskriptgenerierung Das Ergebnis der im vorhergehenden Abschnitt beschriebenen Testfallentwicklung ist ein CTE-Dokument, das einen attributierten Klassifikationsbaum und eine Reihe von Testfallspezifikationen enthält.

Außerdem ist im Zuge der Testfallentwicklung festgelegt worden, welcher Testskriptgenerator verwendet werden soll. Die Verbindung zwischen einem CTE-Dokument und dem zu verwendenden Generator wird in der Regel über das `ScriptGenerator`-Attribut des Wurzelknotens des Klassifikationsbaums hergestellt.

Die Skriptgenerierung wird durch das Programm `cte2cpp` umgesetzt. Details zu Arbeitsweise und Implementierung sind im Abschnitt 4.6 auf Seite 88 zu finden. Hier soll nur ein Überblick gegeben werden.

Während der Skriptgenerierung werden zuerst für jeden Testschritt der Sequenz die Attributdefinitionen zusammengetragen und evaluiert. Attribute eines Testschrittes sind alle Attribute von den Klassen des Klassifikationsbaums, die den Testschritt konstituieren. Im Folgenden soll trotz des etwas größeren Umfanges für jeden Testschritt der Beispiel-Testsequenz `RPL001` die Menge aller Attribute mit ihren Definitionen und den evaluierten Werten dargestellt werden[10]:

Schritt 1

Attribut	**Definition**	**Wert**
%Acceleration	5000	
%DeltaPosition	& (-(%DistanceToZero) + 7000)	-4000
%DistanceToZero	6000	
%IndexLine	0	
%IndexPosition	& -%DistanceToZero	-6000
%InitialAngle	0	
%MaxVelocity	8000	
%MotorType	C-812ISA	
%Velocity	7000	
%axis	DF	
%ini_file	hardware.ini	
%ini_section	Motor0	
%ini_section_a	Motor1	
%ini_section_b	Motor2	
%ini_section_c	Motor3	
%ini_section_d	Motor4	
%selected_axis	DF	
%test_id	RPL001	
ScriptGenerator	m_rpl	
ini		
hardware.ini	(wegen %ini_file = hardware.ini)	
MOTORSIM		
LogFile	%test_id.LOG	RPL001.LOG
LogLevel	2	
StatusWindow	1	
dll	msim.dll	

[10] Die Werte werden nicht wiederholt, falls sie mit ihren Definitionen gleich sind.

hwswDistance	4000	
Motor0	(%ini_section = Motor0)	
Acceleration	%Acceleration	5000
BoardId	%BoardId	2
DeltaPosition	%DeltaPosition	-4000
DistanceToZero	%DistanceToZero	6000
IndexLine	%IndexLine	0
InitialAngle	%InitialAngle	0
InitialMove	%InitialMove	1
MaxVelocity	%MaxVelocity	8000
Name	%axis	DF
PositionMax	15000	
PositionMin	-5000	
RemoveLimit	3000	
Type	%MotorType	C-812ISA
Velocity	%Velocity	7000
Motor1	(%ini_section_a = Motor1)	
BoardID	1	
DistanceToZero	500	
InitialMove	1	
Name	& ('%axis' eq 'DF') ? 'AR' : 'DF'	AR
PositionMax	1000	
PositionMin	-1000	
RemoveLimit	2000	
Type	C-812ISA	
Motor2	(%ini_section_b = Motor2)	
BoardID	3	
DistanceToZero	1000	
InitialMove	1	
Name	& ('%axis' eq 'TL') ? 'AR' : 'TL'	TL
PositionMax	1000	
PositionMin	-1000	
RemoveLimit	2000	
Type	C-812ISA	
Motor3	(%ini_section_c = Motor3)	
BoardID	4	
DistanceToZero	1500	
InitialMove	1	
Name	& ('%axis' eq 'CC') ? 'AR' : 'CC'	CC
PositionMax	1000	
PositionMin	-1000	
RemoveLimit	2000	
Type	C-812ISA	
Motor4	(%ini_section_d = Motor4)	
BoardID	0	
DistanceToZero	2000	
InitialMove	1	
Name	& ('%axis' eq 'DC') ? 'AR' : 'DC'	DC
PositionMax	1000	
PositionMin	-1000	
RemoveLimit	2000	
Type	C-832	
test_id	RPL001	

Schritt 2

Attribut	Definition	Wert
%ini_file	hardware.ini	
%select_motor	DF	
%selected_axis	%select_motor	DF
%test		
step_nr	2	
%test_id	RPL001	
ScriptGenerator	m_rpl	
step		
1	TSTEP_SELECT_MOTOR(%test|step_nr, %select_motor);	TSTEP_SELECT_MOTOR(2, DF);
test_id	RPL001	

Schritt 3

Attribut	Definition	Wert
%ini_file	hardware.ini	
%test		
step_nr	3	
%test_id	RPL001	
ScriptGenerator	m_rpl	
step		
1	TSTEP_CLICK_RPL(%test|step_nr);	TSTEP_CLICK_RPL(3);
test_id	RPL001	

Schritt 4

Attribut	Definition	Wert
%ini_file	hardware.ini	
%test		
pred_step		
selected_axis	DF	
step_nr	4	
%test_id	RPL001	
%time	100	
ScriptGenerator	m_rpl	
step		
1	TSTEP_WAIT_FOR_MESSAGE(%test|step_nr ,Index %test|pred_step |selected_axis, %time);	TSTEP_WAIT_FOR_MESSAGE(4 ,Index DF, 100);
test_id	RPL001	

Schritt 5

Attribut	Definition	Wert
%ini_file	hardware.ini	
%test		
step_nr	5	
%test_id	RPL001	

%time	100
ScriptGenerator	m_rpl
step	
1	TSTEP_WAIT_FOR_MSGBOX_CLOSE(
	%test|step_nr, 5,
	%time); 100);
test_id	RPL001

Schritt 6

Attribut	Definition	Wert
%ini_file	hardware.ini	
%test		
step_nr	6	
%test_id	RPL001	
ScriptGenerator	m_rpl	
step		
1	TSTEP_CLICK_OK(	TSTEP_CLICK_OK(
	%test|step_nr);	6);
test_id	RPL001	

Aus diesen durch die Attribute zur Verfügung gestellten Informationen erzeugt der oben beschrieben Skriptgenerator dann folgendes Skript in der Datei `RPL001.CPP`. Zeile 4-54 des Skriptes werden, wie gesagt, aus den `ini`-Attributen des ersten Testschrittes erzeugt und Zeile 65-73 jeweils aus den `step`-Attributen der Test-Schritte 2 bis 6.

Um das Beispiel von Seite 109 fortzusetzen: der Testschritt 1 wird u.a. von der Klasse „Konfiguration|Beschleunigung|normal“ konstituiert. Daher besteht die Definition des Testschrittes u.a. aus den Attributen

`ini|%ini_file|%ini_section|Acceleration = %Acceleration` und
`%Acceleration = 5000` („normal“) sowie
`%ini_file = hardware.ini` und
`%ini_section = Motor0`.

Aus diesen Angaben erzeugt der Skriptgenerator dann Zeile 9 des Skriptes.

```
 1 #include "m_rpl.inc"
 2
 3 INI_BEGIN(ini_data)
 4 INI_ENTRY("hardware.ini", "MOTORSIM", "LogFile", "RPL001.LOG")
 5 INI_ENTRY("hardware.ini", "MOTORSIM", "LogLevel", "2")
 6 INI_ENTRY("hardware.ini", "MOTORSIM", "StatusWindow", "1")
 7 INI_ENTRY("hardware.ini", "MOTORSIM", "dll", "msim.dll")
 8 INI_ENTRY("hardware.ini", "MOTORSIM", "hwswDistance", "4000")
 9 INI_ENTRY("hardware.ini", "Motor0", "Acceleration", "5000")
10 INI_ENTRY("hardware.ini", "Motor0", "BoardId", "2")
11 INI_ENTRY("hardware.ini", "Motor0", "DeltaPosition", "-4000")
12 INI_ENTRY("hardware.ini", "Motor0", "DistanceToZero", "6000")
13 INI_ENTRY("hardware.ini", "Motor0", "IndexLine", "0")
14 INI_ENTRY("hardware.ini", "Motor0", "InitialAngle", "0")
15 INI_ENTRY("hardware.ini", "Motor0", "InitialMove", "1")
16 INI_ENTRY("hardware.ini", "Motor0", "MaxVelocity", "8000")
```

```
17 INI_ENTRY("hardware.ini", "Motor0", "Name", "DF")
18 INI_ENTRY("hardware.ini", "Motor0", "PositionMax", "15000")
19 INI_ENTRY("hardware.ini", "Motor0", "PositionMin", "-5000")
20 INI_ENTRY("hardware.ini", "Motor0", "RemoveLimit", "3000")
21 INI_ENTRY("hardware.ini", "Motor0", "Type", "C-812ISA")
22 INI_ENTRY("hardware.ini", "Motor0", "Velocity", "7000")
23 INI_ENTRY("hardware.ini", "Motor1", "BoardID", "1")
24 INI_ENTRY("hardware.ini", "Motor1", "DistanceToZero", "500")
25 INI_ENTRY("hardware.ini", "Motor1", "InitialMove", "1")
26 INI_ENTRY("hardware.ini", "Motor1", "Name", "AR")
27 INI_ENTRY("hardware.ini", "Motor1", "PositionMax", "1000")
28 INI_ENTRY("hardware.ini", "Motor1", "PositionMin", "-1000")
29 INI_ENTRY("hardware.ini", "Motor1", "RemoveLimit", "2000")
30 INI_ENTRY("hardware.ini", "Motor1", "Type", "C-812ISA")
31 INI_ENTRY("hardware.ini", "Motor2", "BoardID", "3")
32 INI_ENTRY("hardware.ini", "Motor2", "DistanceToZero", "1000")
33 INI_ENTRY("hardware.ini", "Motor2", "InitialMove", "1")
34 INI_ENTRY("hardware.ini", "Motor2", "Name", "TL")
35 INI_ENTRY("hardware.ini", "Motor2", "PositionMax", "1000")
36 INI_ENTRY("hardware.ini", "Motor2", "PositionMin", "-1000")
37 INI_ENTRY("hardware.ini", "Motor2", "RemoveLimit", "2000")
38 INI_ENTRY("hardware.ini", "Motor2", "Type", "C-812ISA")
39 INI_ENTRY("hardware.ini", "Motor3", "BoardID", "4")
40 INI_ENTRY("hardware.ini", "Motor3", "DistanceToZero", "1500")
41 INI_ENTRY("hardware.ini", "Motor3", "InitialMove", "1")
42 INI_ENTRY("hardware.ini", "Motor3", "Name", "CC")
43 INI_ENTRY("hardware.ini", "Motor3", "PositionMax", "1000")
44 INI_ENTRY("hardware.ini", "Motor3", "PositionMin", "-1000")
45 INI_ENTRY("hardware.ini", "Motor3", "RemoveLimit", "2000")
46 INI_ENTRY("hardware.ini", "Motor3", "Type", "C-812ISA")
47 INI_ENTRY("hardware.ini", "Motor4", "BoardID", "0")
48 INI_ENTRY("hardware.ini", "Motor4", "DistanceToZero", "2000")
49 INI_ENTRY("hardware.ini", "Motor4", "InitialMove", "1")
50 INI_ENTRY("hardware.ini", "Motor4", "Name", "DC")
51 INI_ENTRY("hardware.ini", "Motor4", "PositionMax", "1000")
52 INI_ENTRY("hardware.ini", "Motor4", "PositionMin", "-1000")
53 INI_ENTRY("hardware.ini", "Motor4", "RemoveLimit", "2000")
54 INI_ENTRY("hardware.ini", "Motor4", "Type", "C-832")
55 INI_END
56
57 TMAIN
58 {
59    T_INIT_FRAME(RPL001, "RPL001");
60
61    TSTEP_RUN_MDLG(RPL001, ini_data);
62
63    //-----------------------------------------
64
65    TSTEP_SELECT_MOTOR(2, DF);
66
```

```
67    TSTEP_CLICK_RPL(3);
68
69    TSTEP_WAIT_FOR_MESSAGE(4 ,Index DF, 100);
70
71    TSTEP_WAIT_FOR_MSGBOX_CLOSE( 5, 100);
72
73    TSTEP_CLICK_OK(6);
74
75    //-----------------------------------------
76
77    T_CLEANUP_AND_RETURN;
78 }
```

Übersetzung des Testskriptes Der zweite Schritt während der Testprogrammerzeugung ist im vorliegenden Testpaket die Übersetzung der C++-Dateien in ein ausführbares Programm. In diesen Vorgang gehen neben dem Testskript selbst, noch weitere Informationen ein. Dazu zählen das Testobjekt selbst, im vorliegenden Fall die Motorsteuerung in Form einer DLL, sowie der Testrahmen. Der Testrahmen enthält unter anderem die Definitionen der Makros, die im Testskript verwendet werden, sowie Hilfsmittel zum Protokollieren der Ein- und Ausgaben.

Dieser Übersetzungsschritt fällt natürlich nur an, wenn die Sprache in der die Testskripte geschrieben werden eine Übersetzung notwendig macht.

4.8.3 Testausführung

Die Testausführung ist die einfachste der Testaktivitäten: das übersetzte Testprogramm muss ausgeführt werden. Während der Ausführung erzeugt das Testprogramm eine Protokoll- oder Log-Datei, die Ein- und Ausgaben des Tests enthält. Für den Fall, dass das zu testende Programm entsprechend instrumentiert ist, werden dabei auch Informationen über die Quelltextüberdeckung gewonnen.

Die Ist-Daten Die genaue Struktur der während der Testausführung aufgezeichneten Ist-Daten ist im Abschnitt 4.7 erläutert worden. Die Ist-Daten-Datei, die von der Testsequenz `RPL001` erzeugt wird, ist etwa 6500 Zeilen groß. Daher soll hier nur ein Ausschnitt exemplarisch abgebildet und erläutert werden. Die Ist- und auch die Soll-Daten werden im hier entworfenen Testsystem als XML-Daten gespeichert. Das folgende Beispiel zeigt den Ausschnitt, der das Ende des Testschrittes 3 und den Anfang des Testschrittes 4 wiedergibt.

```
<test-log id='RPL001'>
  <title>RPL001</title>
  <date>Tue Oct 16 17:15:20 2001</date>
  ...
  <test-step number='3'>
     <title>start reference point run</title>
     ...
     <output>C812ISA get 0x025f0800 </output>
     <input src='sim'>fc</input>
```

```
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA put 0x025f03fc 32 = 2</output>
<output>C812ISA put 0x025f03ff 32 = 2</output>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA put 0x025f03fc 4d = M</output>
<output>C812ISA put 0x025f03ff 4d = M</output>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA put 0x025f03fc 41 = A</output>
<output>C812ISA put 0x025f03ff 41 = A</output>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA put 0x025f03fc 2d = -</output>
<output>C812ISA put 0x025f03ff 2d = -</output>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA put 0x025f03fc 34 = 4</output>
<output>C812ISA put 0x025f03ff 34 = 4</output>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA put 0x025f03fc 30 = 0</output>
<output>C812ISA put 0x025f03ff 30 = 0</output>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA put 0x025f03fc 30 = 0</output>
<output>C812ISA put 0x025f03ff 30 = 0</output>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA put 0x025f03fc 30 = 0</output>
<output>C812ISA put 0x025f03ff 30 = 0</output>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
```

```
    <output>C812ISA put 0x025f03fc 30 = 0</output>
    <output>C812ISA put 0x025f03ff 30 = 0</output>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fc</input>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fc</input>
    <output>C812ISA put 0x025f03fc 30 = 0</output>
    <output>C812ISA put 0x025f03ff 30 = 0</output>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fc</input>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fc</input>
    <output>C812ISA put 0x025f03fc 30 = 0</output>
    <output>C812ISA put 0x025f03ff 30 = 0</output>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fc</input>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fc</input>
    <output>C812ISA put 0x025f03fc 30 = 0</output>
    <output>C812ISA put 0x025f03ff 30 = 0</output>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fc</input>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fc</input>
    <output>C812ISA put 0x025f03fc 30 = 0</output>
    <output>C812ISA put 0x025f03ff 30 = 0</output>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fc</input>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fc</input>
    <output>C812ISA put 0x025f03fc 0d</output>
    <output>C812ISA put 0x025f03ff 0d</output>
    <cmd-seq>
      <cmd c='C812ISA'>move absolute 2, -400000000</cmd>
      <out>{ETX}</out>
    </cmd-seq>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fe</input>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fe</input>
    <output>C812ISA get 0x025f03fe </output>
    <input src='sim'>03</input>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fc</input>
</test-step>
<test-step number='4'>
    <title>wait for message 'Index DF' 100</title>
    <output>C812ISA get 0x025f0800 </output>
    <input src='sim'>fc</input>
    <output>C812ISA get 0x025f0800 </output>
```

```
      <input src='sim'>fc</input>
      <output>C812ISA put 0x025f03fc 32 = 2</output>
      <output>C812ISA put 0x025f03ff 32 = 2</output>
      <output>C812ISA get 0x025f0800 </output>
      <input src='sim'>fc</input>
      <output>C812ISA get 0x025f0800 </output>
      <input src='sim'>fc</input>
      <output>C812ISA put 0x025f03fc 54 = T</output>
      <output>C812ISA put 0x025f03ff 54 = T</output>
      <output>C812ISA get 0x025f0800 </output>
      <input src='sim'>fc</input>
      <output>C812ISA get 0x025f0800 </output>
      <input src='sim'>fc</input>
      <output>C812ISA put 0x025f03fc 53 = S</output>
      <output>C812ISA put 0x025f03ff 53 = S</output>
      <output>C812ISA get 0x025f0800 </output>
      <input src='sim'>fc</input>
      <output>C812ISA get 0x025f0800 </output>
      <input src='sim'>fc</input>
      <output>C812ISA put 0x025f03fc 0d</output>
      <output>C812ISA put 0x025f03ff 0d</output>
      <cmd-seq>
        <cmd c='C812ISA'>tell status 2</cmd>
        <info c='c812'>time 1191 position -241/4759
                       velocity 1550</info>
        <out>02S0000{CR}{LF}
        {ETX}{ETX}</out>
      </cmd-seq>
      ...
  </test-step>
  ...
</test-log>
```

Eine Log-Datei enthält im wesentlichen eine Sequenz der Ein- und Ausgaben des zu testenden Programmes die den Testfall ausmachen. Ausgaben des Programmes werden durch `<output>`-Elemente dargestellt, Eingaben durch `<input>`-Elemente. Die Ein- und Ausgaben sind entsprechend den Testschritten gruppiert, wodurch es möglich wird, später beim Vergleich die Korrektheit der Testschritte einzeln zu bewerten. Die Ausgaben im Beispiel kodieren Schreib- und Leseoperationen der Software auf dem C-812er Motorcontroller. Die Eingaben sind die Antworten der Hardware (bzw. der Simulation) auf diese Anforderungen. Eine genaue Beschreibung dieser Protokollierung ist in diesem Fall in der Dokumentation der Motorensimulation zu finden (s. Abschnitt 3.4.5 auf S. 65).

Die im oben gegebenen Beispiel-Protokoll enthaltene Kommunikations-Sequenz beinhaltet das Senden des Motorbefehls `2MA-400000000`, der den zweiten Motor des C-812er Controllers anweist, auf die absolute Position -400000000 zu fahren. Danach – zu Beginn des Testschrittes 4 – wird der Befehl `2TS` gesendet, mit dessen Hilfe der Status des Motors abgefragt wird. Die Simulation bettet in das Protokoll zusätzliche Informationen ein, die die

Interpretation vereinfachen sollen. Diese Zusatzinformationen sind z.B. in den `<cmd-seq>`-Elementen zu finden.

Neben den eigentlichen Ein- und Ausgabe-Daten enthält das Protokoll noch einige Verwaltungsinformationen wir z.B. die Test-ID und das Datum der Testausführung.

4.8.4 Testauswertung

Aufgabe der abschließenden Auswertung der Testergebnisse ist die Überprüfung, ob die während der Testausführung erzeugten und in der Log-Datei gespeicherten Ist-Daten den erwarteten Ergebnissen entsprechen. Dazu müssen natürlich Soll-Daten vorliegen. Die hier gewählte Vorgehensweise erzeugt die erste Version der Soll-Daten auf die selbe Art wie die Ist-Daten – durch eine Testausführung. Die dabei gewonnenen Daten werden einer Bewertung unterzogen und zu Soll-Daten erklärt. Für eine genauere Erläuterung der Vorgehensweise s. Kapitel 2.

Die Soll-Daten Die Soll-Daten haben im Prinzip die gleiche Struktur wie die Ist-Daten, mit einigen Erweiterungen zum Beschreiben von zulässigen Unschärfen. So erlaubt eine dieser Erweiterungen die Beschreibung von Ausgabe-Wiederholungen.

In der Testsequenz `RPL001` wird z.B. zu Beginn des 4 Testschrittes, d.h. nach dem der Motor in Bewegung gesetzt wurde, der Motor so lange nach seinem Status befragt, bis dieser das Erreichen des Index-Schalters signalisiert. Die Anzahl der notwendigen Statusabfragen kann nun je nach Geschwindigkeit der Rechners und in Abhängigkeit vom Versuchsaufbau zwischen 80 und 120 schwanken. Um dies in den Soll-Daten auszudrücken, wurde die Sequenz von Ein- und Ausgaben, die in dieser Wiederholung auftauchen, durch ein `<output-seq>`-Element gruppiert und mit den Attributen `min-occur="80"` und `max-occur="120"` versehen, wie im folgenden Ausschnitt sehen ist.

```
<test-step number="4">
  <title>wait for message 'Index DF' 100</title>

  <comment>Status auslesen, bis Index erreicht</comment>
  <output-seq min-occur="80" max-occur="120">
    <output>C812ISA get 0x025f0800 </output>
    <input src="sim">fc</input>
    <output>C812ISA get 0x025f0800 </output>
    <input src="sim">fc</input>
    <output>C812ISA put 0x025f03fc 32 = 2</output>
    <output>C812ISA put 0x025f03ff 32 = 2</output>
    <output>C812ISA get 0x025f0800 </output>
    <input src="sim">fc</input>
    <output>C812ISA get 0x025f0800 </output>
    <input src="sim">fc</input>
    <output>C812ISA put 0x025f03fc 54 = T</output>
    <output>C812ISA put 0x025f03ff 54 = T</output>
    <output>C812ISA get 0x025f0800 </output>
    <input src="sim">fc</input>
    <output>C812ISA get 0x025f0800 </output>
```

```
    <input src="sim">fc</input>
    <output>C812ISA put 0x025f03fc 53 = S</output>
    <output>C812ISA put 0x025f03ff 53 = S</output>
    <output>C812ISA get 0x025f0800 </output>
    <input src="sim">fc</input>
    <output>C812ISA get 0x025f0800 </output>
    <input src="sim">fc</input>
    <output>C812ISA put 0x025f03fc 0d</output>
    <output>C812ISA put 0x025f03ff 0d</output>
    <cmd-seq>
      <cmd c="C812ISA">tell status 2</cmd>
      <info c="c812">time 1191 position -241 velocity 1550</info>
      <out>02S0000{CR}{LF}
      {ETX}{ETX}</out>
    </cmd-seq>
    ...
  </output-seq>
  ...
</test-step>
```

Der Vergleich Mit Hilfe des Programmes `run_test.pl` wird nun der Vergleich der Ist- mit den Soll-Daten durchgeführt. Dieses relativ einfache Programm wird im Abschnitt 4.7 erläutert. Es überprüft für jeden Testschritt einzeln, ob die in den Ist-Daten gespeicherte Folge von Programmausgaben (d.h. die Folge von `<output>`-Elementen) den in den Soll-Daten beschriebenen Folgen entspricht.

Während der Durchführung des Vergleiches wird eine Ergebnis-Datei (im Beispiel `RPL001.RES`) erzeugt, in der für jeden Testschritt das Ergebnis festgehalten wird. Diese Ergebnis-Datei ist im Folgenden vollständig abgebildet. Sie enthält die Ergebnisse des Tests `id='RPL001'` aus dem Paket `pkg='m_rpl'`. Der Test wurde mit dem Build `build='BUILD001'` durchgeführt und der Vergleich erfolgte mit der Soll-Daten-Version `baseline='bas001'`. In Testschritt 2 ist ein Fehler aufgetreten. Alle anderen Testschritte wurden erfolgreich durchgeführt. Für jeden fehlerhaften Testschritt erzeugt der Vergleicher eine Fehlerbeschreibungsdatei, für die Visualisierung der Differenzen zwischen Soll- und Ist-Daten (s.u.).

```
<?xml version="1.0" encoding="UTF-8" ?>
<?xml-stylesheet type="text/xsl" href="../xml/testres.xsl"?>
<test-seq-result pkg='m_rpl' id='RPL001'
                 baseline='bas001' build='BUILD001'>
  <title>RPL001</title>
  <test-step-result number='1'>
    <title>run 'mdlg.exe rp hardware.ini'</title>
    <result context='baseline' type='pass'/>
  </test-step-result>
  <test-step-result number='2'>
    <title>select motor 'DF'</title>
    <result context='baseline' type='fail' err='1'/>
  </test-step-result>
```

```
  <test-step-result number='3'>
    <title>start reference point run</title>
    <result context='baseline' type='pass'/>
  </test-step-result>
  <test-step-result number='4'>
    <title>wait for message 'Index DF' 100</title>
    <result context='baseline' type='pass'/>
  </test-step-result>
  <test-step-result number='5'>
    <title>wait for msgbox 100</title>
    <result context='baseline' type='pass'/>
  </test-step-result>
  <test-step-result number='6'>
    <title>click 'OK'</title>
    <result context='baseline' type='pass'/>
  </test-step-result>
</test-seq-result>
```

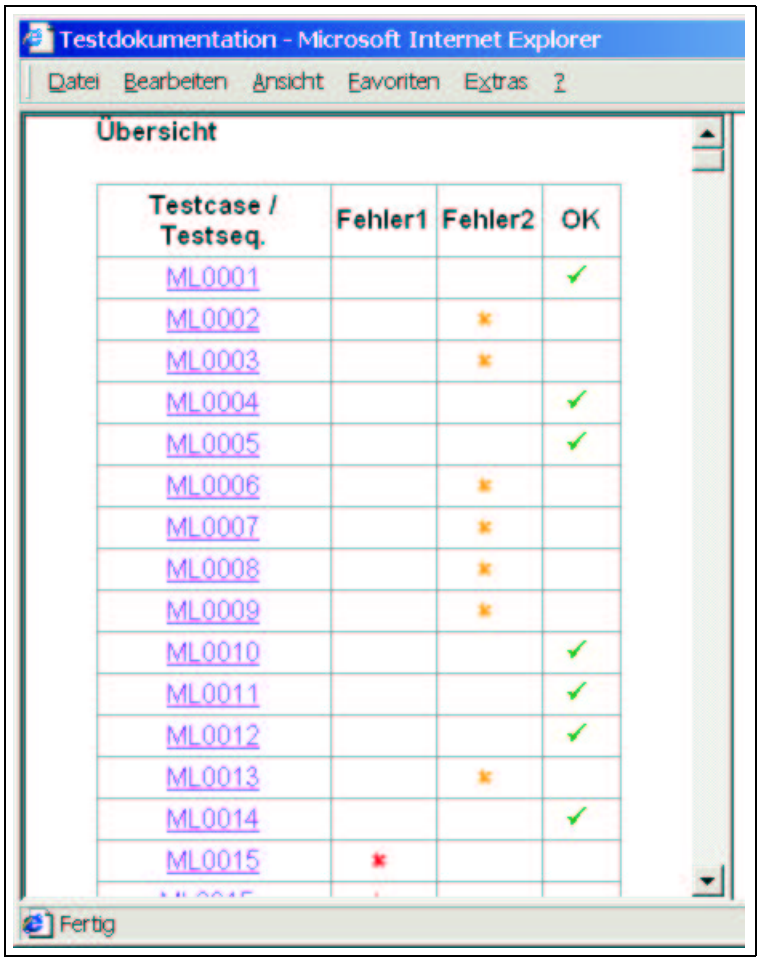

Abbildung 4.19: Testdokumentation (Übersicht)

Nach Abschluss des Vergleichs erzeugt das Programmes `run_test.pl` noch zwei Dateien, die der Integration der Testergebnisse in eine Testdokumentation dienen. Auf die sehr einfache Struktur dieser Dateien namens `result.xml` und `index.html` soll hier nicht weiter eingegangen werden. `index.html` stellt das Startdokument für das Lesen der Testdokumentation dar.

Für alle während des Tests erstellten XML-Dateien sind XSL-Stylesheets geschrieben worden, die für die Visualisierung der Dokumente mit Hilfe von Standardbrowsern wie z.B. dem Internet Explorer verwendet werden können. Die Abbildungen 4.19 bis 4.21 sollen einen Überblick über diese Dokumentation geben.

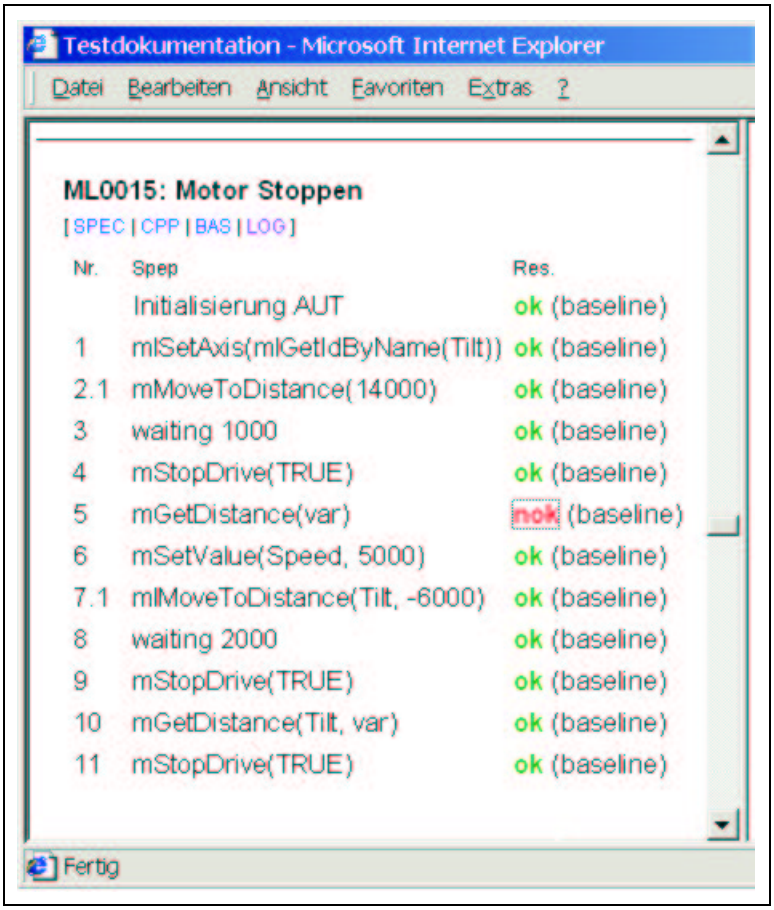

Abbildung 4.20: Testdokumentation (Testsequenzinformationen)

Am Anfang der Dokumentation steht eine Übersicht über alle durchgeführten Testfälle bzw. -sequenzen (Abb. 4.19). Diese Übersicht listet die Testfälle auf und markiert an jedem, ob der Test erfolgreich war, oder ob eine Abweichung zum letzten Ist-Verhalten (Fehler1) oder zum eigentlichen Soll-Verhalten (Fehler2) aufgetreten ist. So ist in der Abbildung zu erkennen, dass der Testfall `ML0001` erfolgreich war, dass das Verhalten während des Testfalls `ML0002` zwar dem letzten akzeptierten Verhalten folgt, dieses aber noch nicht den Anforderungen entspricht, und schließlich dass im Testfall `ML0015` eine

Abweichung vom bisher beobachteten Verhalten vorliegt.

Zu den einzelnen Testfällen oder -sequenzen erreicht man über Links detailliertere Informationen, wie in Abb. 4.20 für die fehlgeschlagene Testsequenz `ML0015` zu sehen ist. Dort werden die Testschritte aufgelistet und markiert welche davon erfolgreich waren und welche nicht. Von hieraus kann man die Testspezifikation (SPEC), das generierte Testskript (CPP), die Soll-Daten (BAS) und die Ist-Daten (LOG) lesen. Außerdem gibt es für jeden Fehler eine Fehlerbeschreibung. Für den fehlerhaften Testschritt 5 der Testsequenz `ML0015` zeigt Abb. 4.21 eine solche Beschreibung.

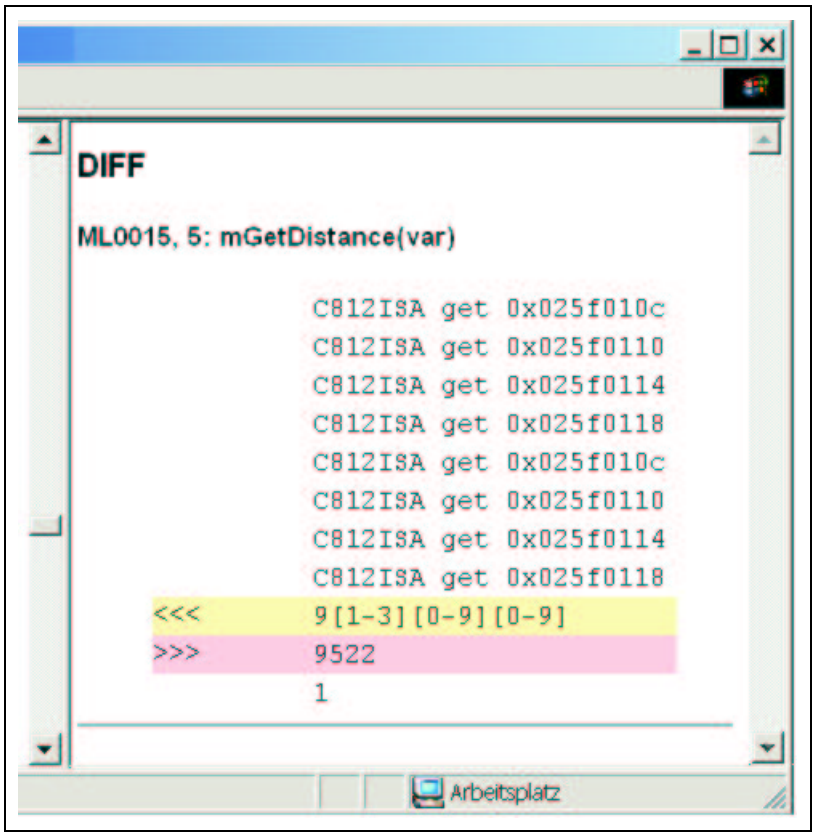

Abbildung 4.21: Testdokumentation (Fehlerbeschreibung)

In dieser Fehlerbeschreibung wird die Sequenz der vom zu testenden Programm erzeugten Ausgaben angezeigt, und Abweichungen von den Soll-Daten farbig angezeigt. Im vorliegenden Beispiel enthält die Ausgabe-Sequenz an vorletzter Stelle eine `9522`. In den Soll-Daten sind aber, mit Hilfe des regulären Ausdrucks `9[1-3][0-9][0-9]` nur Werte zwischen `9100` und `9399` zu gelassen worden.