Graphische Testfallbeschreibung mit UML – Expertenwissen in Bildern?

Dipl.-Ing. Volker Knollmann[1], Prof. Dr.-Ing. Karsten Lemmer, Institut für Verkehrsführung und Fahrzeugsteuerung (IFS), Deutsches Zentrum für Luft- und Raumfahrt e. V. (DLR), Braunschweig

Kurzfassung

Das *UML Testing Profile* stellt eine Möglichkeit zur formalen, graphischen Notation statischer und dynamischer Eigenschaften von Testfällen und Testsequenzen dar. Der Einsatz des Profils zur Testfallbeschreibung für ein reales System lies viele Vorteile (Übersichtlichkeit, Wartbarkeit, Aufbau von Bibliotheken) erkennen, zeigte aber auch, dass eine gründliche Einarbeitung in die Sprache und eine sorgfältige Strukturierung der entstehenden Modelle unerlässlich sind. Die UML-Testfallbeschreibungen sind sowohl Grundlage für eine Testautomatisierung als auch Bestandteil eines größeren, integralen System- und Testmodells, das eine engere Verzahnung von Entwicklungs- und Testprozess ermöglicht.

1 Motivation

Die Entwicklung neuer (Software-)Systeme wird heute mehr denn je von effizienten Teststrategien geprägt. Dass dabei der Umfang der Testmaßnahmen nicht immer dem Umfang und der Komplexität der zu testenden Systeme angemessen ist (und somit erhebliche Restfehler im System verbleiben), ist sicher jedem, der ein modernes Handy oder ein modernes Betriebssystem verwendet, bereits aufgefallen.

Die Entwicklung sicherheitskritischer Systeme – beispielsweise Stellwerks- oder Zugsicherungsrechner in der Eisenbahndomäne, Flugsicherungsrechner („Flight Control System“) in Flugzeugen oder Bremsassistenten in Automobilen – unterliegt daher strengen Vorgaben, um die Qualität neuer Produkte sicherzustellen.

Die erheblichen technischen Herausforderungen, die sich bei der Entwicklung und Einführung neuer Technologien ergeben, haben sich in den letzten Jahren auch bei der Planung und Inbetriebnahme von Bahnstrecken mit den europaweit vereinheitlichtem Sicherungssystem *ETCS* (European Train Control System) gezeigt. Da ein Schwerpunkt von ETCS die Kooperation von Geräten verschiedener Hersteller ist, wurde dem Thema *Interoperabilität* und dem Test der Schnittstellen von Anfang an erhebliche Bedeutung beigemessen.

Für unabhängige Interoperabilitätsuntersuchungen entstand daher am Institut für Verkehrsführung und Fahrzeugsteuerung (IFS) des Deutschen Zentrums für Luft- und Raumfahrt e. V. in Braunschweig das Simulationslabor RailSiTe® (Rail Simulation and Testing). Mit Hilfe des Labors lassen sich unter anderem reale Komponenten einem Hardware-in-the-Loop-Test (HiL-Test) unterziehen. Außerdem lässt sich, beispielsweise für theoretische Untersuchungen, das ETCS-System vollständig auf handelsüblichen Rechnern simulieren. Die stark modularisierte und verteilte Systemarchitektur des Labors erlaubt darüber hinaus viele weitere Nutzungsszenarien, wie beispielsweise auch die Untersuchung neuer Streckenprojektierungen.

Der Aufbau wie auch die Nutzung des RailSiTe erfordern regelmäßige Testkampagnen, die verschiedenen Ansprüchen gerecht werden müssen:

- Die Testbeschreibung muss nachvollziehbar, verständlich und leicht editierbar sein.
- Die Tests müssen in einer Notation vorliegen, die eine automatische Ausführung und Auswertung ermöglicht bzw. unterstützt.
- Testfälle müssen sich in Bibliotheken sammeln und für nachfolgende Projekte wiederverwenden lassen.
- Für Tests, die die Funktionalität des Labors an sich betreffen, muss ein Tracing auf die zugrunde liegenden funktionalen Anforderungen möglich sein.

Als möglichen Lösungsansatz stellen Ebrecht und Meyer zu Hörste dazu in [1] eine XML-basierte Testfallbeschreibung vor. Die Idee dazu resultierte aus der Arbeit mit den offiziellen ETCS-Testsequenzen ([2]), die als Tabelle in einer Microsoft®-Word®-Datei vorliegen und die das Geschwindigkeitsprofil (v-s-Diagramm) für einen virtuellen Testzug als Bitmapdatei definieren[2].

Während XML-Dokumente für automatisierte Testausführung und Testauswertung aufgrund ihrer strengen Syntax und Semantik sehr gut geeignet sind, ist die (manuelle) Erstellung und Bearbeitung komplexerer Testszenarien direkt in XML nicht mehr handhabbar. Eine mögliche Lösung dieses Problems,

[1] Volker.Knollmann@dlr.de

[2] Die Testbeschreibungen sind zwar auch als Datenbank verfügbar, die maßgebliche Version stellt jedoch das Word-Dokument dar.

die in diesem Artikel vorgestellt wird, ist die Verwendung graphischer Notationen wie beispielsweise UML (Unified Modeling Language). Diese Art von von Testbeschreibung lässt sich zum einen komfortabel editieren und zum anderen ist eine Überführung in andere (z. B. XML-basierte) Beschreibungsformen möglich.
In den folgenden Abschnitten soll daher das "UML 2 Testing Profile" (kurz U2TP, [3]) vorgestellt und sein Einsatz in der Praxis veranschaulicht werden.

2 Das UML 2 Testing Profile

2.1 Allgemeines und Aufbau des Profils

Das U2TP definiert auf Basis der UML-Notationselemente und einiger weniger neuer Symbole einen Satz von Diagrammen zur Testfallbeschreibung. Wie für ein UML-Profil üblich, spezifiziert auch das U2TP zu diesem Zwecke verschiedene Stereotypen, Tags und Symbole sowie deren Syntax und semantische Bedeutung. Es vereinbart somit einen Katalog von Elementen, die von allen Benutzern des Profils einheitlich interpretiert werden, und ermöglicht dadurch die eindeutige Beschreibung eines bestimmten Sachverhaltes.

Wie jede andere Systembeschreibung muss auch die Spezifikation von Tests sowohl statische als auch dynamische Aspekte – also Struktur und Verhalten – berücksichtigten. Das U2TP ist dem entsprechend in folgende Teile gegliedert, auf die in den nächsten Abschnitten genauer eingegangen wird:

Test Architecture: Beschreibung des Aufbaus der Testumgebung, Zusammenwirken der einzelnen Komponenten, Konfiguration für verschiedene Testanordnungen

Test Behaviour: Beschreibung des Testverhaltens, der Testschritte, der Sollergebnisse jedes Schrittes und der Reaktionen bei Abweichung der Ergebnisse vom Sollverhalten

Test Data: Definition von Testdaten, Partitionierung der Testdaten in Untereinheiten, Zugriffsmethoden auf die Testdaten

Time Concepts: Spezifikation von Timern und Timeouts, um zeitliche Anforderungen an den Testablauf beschreiben zu können

Die Konzepte des U2TP beziehen sich rein auf *BlackBox*-Tests und lassen sich auf allen Testebenen vom Modul- bis zum Systemtest anwenden. Bei der U2TP-Spezifikation handelt es sich um eine reine Notations- und Semantikdefinition; sie ist daher werkzeug- und plattformneutral. Ebenso ist die Verwendung des U2TP nicht auf eine bestimmte Domäne beschränkt.

2.2 Strukturbeschreibung

Eine Testumgebung zeichnet sich vor allem durch den Prüfling (SUT, *System Under Test*) und die mit ihm interagierenden Komponenten (*Test Components*) aus. Die Testkomponenten stellen die notwendige Funktionalität zur Stimulation des Prüflings zur Verfügung und nehmen auch dessen Ausgaben auf.

Der Prüfling und die Testkomponenten werden als UML-Klassen in entsprechenden Klassen- bzw. Strukturdiagrammen beschrieben und durch die entsprechenden Stereotypen `≪SUT≫` bzw. `≪TestComponent≫` klassifiziert.

Darüber hinaus definiert das U2TP auch Schnittstellen für Klassen, die der Steuerung des Testablaufs dienen. Es handelt sich dabei zum einen um den sogenannten *Arbiter*, der die Rückmeldungen verschiedener Testkomponenten über den Erfolg einzelner Testschritte zu einem Gesamturteil über den vollständigen Test zusammenfasst. Zum anderen wird das Interface für eine *Scheduler*-Klasse eingeführt, die den zeitlichen Ablauf des Tests steuert und ggf. Testkomponenten instantiiert oder startet.

Scheduler und *Arbiter* sind im U2TP nur durch ihre Schnittstelle und ihr grundsätzliches Verhalten beschrieben. Eine tatsächliche Implementierung muss durch den Anwender vor dem Hintergrund der jeweiligen Testumgebung vorgenommen werden.

Zusammenstellung der Testkonfiguration
Die einzelnen Bestandteile der Testumgebung werden durch die bisher genannten Maßnahmen nur „lose", also alleinstehend, definiert. Dadurch wird es möglich, Bibliotheken verschiedenster Testkomponenten aufzubauen und aus deren Elementen dann einen speziellen Testaufbau zu kombinieren.

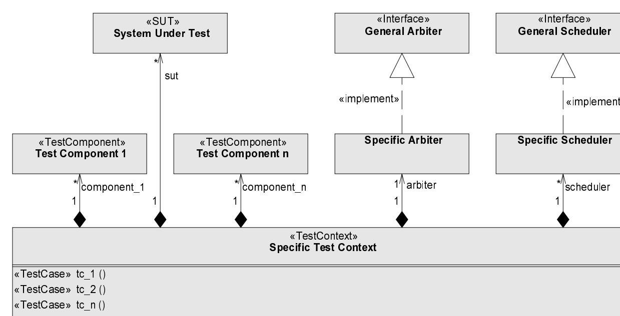

Bild 1: Zusammenstellung mehrerer Komponenten zu einer individuellen Testkonfiguration (*Test Context*)

Die Zusammenstellung einer solchen individuellen Testkonfiguration erfolgt mit Hilfe des sog. *Test Context* (**Bild 1**). Dabei handelt es sich um eine Klasse, die Instanzen der jeweils zu verwendenden Testkomponenten enthält (als Attribute bzw. Assoziation) und auch die inneren Beziehungen zwischen diesen Instanzen festlegt (in einem Strukturdiagramm). Die Methoden einer solchen mit `≪TestContext≫` stereotypisierten Klasse sind die einzelnen Testfälle (klassifiziert durch `≪Testcase≫`), die durch Aufruf von außen ausgelöst werden können und somit das

Interface zwischen Tester und Testumgebung darstellen.

2.3 Verhaltensbeschreibung

Die dynamischen Aspekte einer Testfallbeschreibung werden vor allem durch die Interaktion zwischen den Testkomponenten und dem Prüfling bestimmt. Sie werden mit Hilfe der in der UML gängigen Verhaltensdiagramme (vornehmlich Sequenz-, Aktivitäts- oder Zustandsdiagramme) beschrieben. Üblicherweise sind für diesen Anwendungsfall Sequenzdiagramme die geeignetste Notation, da sich in ihnen die Abfolge des Datenaustausches an den Prüflingsschnittstellen und auch zeitliche Randbedingungen übersichtlich darstellen lassen.

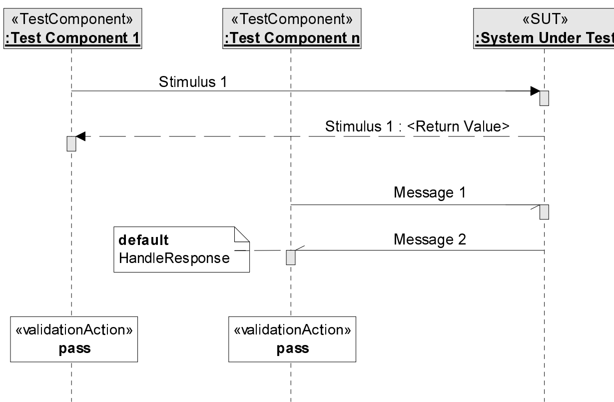

Bild 2: Exemplarische Interaktion zwischen Testkomponenten und SUT in einem Sequenzdiagramm

Der obere Teil von **Bild 2** zeigt, wie die erste Testkomponente den Prüfling durch einen synchronen Aufruf stimuliert und dieser mit einem Rückgabewert antwortet. Im zweiten Teil wird eine asynchrone Kommunikation mittels Austausch von Nachrichten dargestellt.
Für den Fall, dass ein Prüfling während der Testausführung vom Sollverhalten abweicht, muss entschieden werden, wie der weitere Testablauf zu gestalten ist. Dafür sieht das U2TP den Einsatz sog. *defaults* vor, der ebenfalls in Bild 2 dargestellt ist. Semantisch kann das verstanden werden als: „Wenn eine andere Nachricht als `Message 2` eintrifft, rufe das Unterprogramm `HandleResponse` auf!". Das *default*-Konzept wurde aus der Sprache TTCN-3 (*Testing and Test Control Notation Version 3*, [4]) übernommen.
Ein *default* verweist also auf eine separate Verhaltensbeschreibung, die den weiteren Testablauf festlegt und für gewöhnlich als weiteres Sequenz- oder Zustandsdiagramm ausgeführt ist. Folgende Möglichkeiten bestehen:

- Wiederholung des fehlerauslösenden Testschritts (*repeat*)
- Ignorieren des Fehlers und Fortsetzen der Testausführung (*continue*)
- Abbruch des Testablaufs

Durch die Trennung von „Normalablauf" und Fehlerbehandlung werden die Hauptdiagramme übersichtlicher gehalten und dadurch die Les- und Wartbarkeit des UML-Modells verbessert.
Selbstverständlich muss durch einen *default* auch die Wertung des Testergebnisses verändert werden. Wie oben bereits erwähnt erfolgt die Beurteilung des gesamten Tests durch den Arbiter. Die am Test beteiligten Komponenten melden ihm zur Laufzeit ihre „lokalen" Testergebnisse, aus denen der Arbiter dann das Gesamtergebnis ermittelt. Im U2TP sind bereits einige Ergebnisvarianten vordefiniert:

pass: Der Prüfling hat sich spezifikationskonform verhalten.

fail: Das Gegenteil von *pass*

inconclusive: Es ist nicht entscheidbar, ob das Verhalten spezifikationskonform ist oder nicht

error: Es liegt ein Fehler in der Testumgebung vor

Ein einfacher Arbiter könnte beispielsweise eine *worst-case*-Entscheidung vornehmen. Das Verhalten eines entsprechenden Arbiters zeigt **Bild 3**:

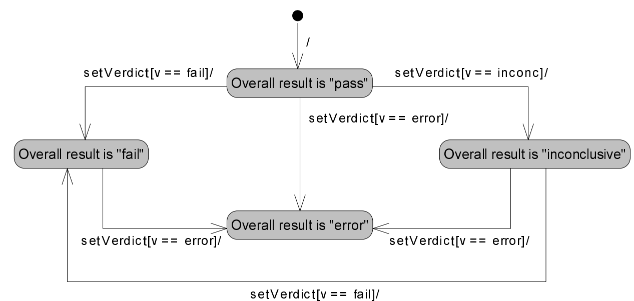

Bild 3: Zustandsdiagramm eines einfachen Arbiters, der eine *worst-case*-Entscheidung vornimmt. Das `v` in den *Guard-Conditions* sei der Parameter der `setVerdict`-Methode des Arbiters.

Würde also eine Testkomponente bei Abweichung des Prüflings vom Sollverhalten ein lokales Urteil *fail* an den Arbiter übertragen, könnte eine andere Komponente niemals mehr ein Gesamtergebnis *pass* hervorrufen. Selbstverständlich sind aber auch andere Arbiter-Implementierungen denkbar, zum Beispiel die Einführung einer Toleranzschwelle, die trotz einer gewissen Anzahl lokaler *fail*-Entscheidungen insgesamt dennoch ein *pass*-Urteil fällt. Aufgrund der vielen Möglichkeiten und Anwendungsfälle definiert das U2TP an dieser Stelle auch nur ein allgemeines Interface und keine genaue Implementierung eines Arbiters.
Innerhalb der Verhaltensdiagramme wird die Übergabe eines lokalen Urteils an den Arbiter entweder durch explizite Modellierung des Aufrufs seiner `setVerdict`-Methode erfasst oder durch die Einführung einer Aktion, die mit `«validationAction»` stereotypisiert ist (Bild 2).

2.4 Testdaten und zeitliche Randbedingungen

Die für die Ausführung von Tests notwendigen Daten (z. B. Sollergebnisse oder Nachrichtentelegramme) werden im U2TP in sog. *Data Pools* organisiert, die wiederum in Untereinheiten (*Data Partitions*) aufgeteilt werden können. Die Notation in UML erfolgt, wie bei Datenstrukturen üblich, als Klassendiagramm, für das die entsprechenden Stereotypen `≪DataPool≫` und `≪DataPartition≫` vom U2TP zur Verfügung gestellt werden.
Für die Beschreibung zeitlicher Zusammenhänge und Abhängigkeiten werden im U2TP Schnittstellen für Timer und Aktionen für deren Steuerung / Interaktion definiert. Timer verfügen über die Grundfunktionen starten, stoppen und auslesen und können nach Ablauf einer wählbaren Zeit eine bestimmte Aktion auslösen (*TimeOutAction*). Darüber hinaus können Testkomponenten verschiedenen Zeitdomänen (sog. *time zones*) zugeordnet werden, um auf diese Weise synchron arbeitende Subsysteme zu gruppieren.

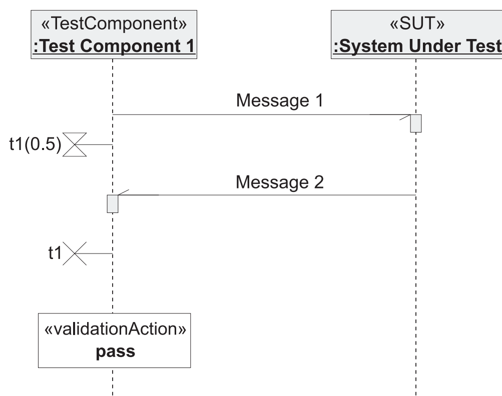

Bild 4: Exemplarisches Sequenzdiagramm, das den Einsatz von Timern zeigt

Den Einsatz von Timern verdeutlicht **Bild 4** anhand eines fiktiven Beispiels. Nach dem Senden der ersten Nachricht wird ein Timer mit einem Startwert von 0,5 s gestartet und bei Empfang der zweiten Nachricht wieder gestoppt. Sollte es zu einem Timeout kommen, ist die zweite Nachricht nicht rechtzeitig eingetroffen und eine *TimeOutAction* wird ausgelöst, die beispielsweise im *default* des *TestContext* verarbeitet werden könnte (zu *defaults* siehe Abschnitt 2.3).

3 Fallstudie

Die Leistungsfähigkeit der U2TP-Testbeschreibungen wurde im Rahmen einer Fallstudie am IFS erprobt. Der Prüfling bestand aus einem System zur Erzeugung bestimmter Transpondersignale, das Teil des oben beschriebenen Bahnlabors RailSiTe ist. Durch diese Laborkomponente kann einem realen Fahrzeugrechner im Rahmen eines HiL-Tests die Vorbeifahrt an einer sogenannten Balise (passiver Transponder) simuliert werden.

Dies geschieht, indem die in der Realität von der Balise abgestrahlten Hochfrequenzsignale durch das Simulationssystem erzeugt und über eine Antenne ausgesendet werden. Bei der Signalerzeugung müssen natürlich die zu übertragenden Daten und das von der (virtuellen) Zuggeschwindigkeit abhängige zeitliche Profil der Übertragung berücksichtigt werden. Die Balisensimulation ist über eine RS232-Verbindung mit dem Laborkern verbunden und bezieht von dort alle notwendigen Daten. Die Kommunikation erfolgt über ein verbindungsloses, auf einzelnen Nachrichtentelegrammen basierendes Protokoll ([5]).
Für diese Balisensimulation wurden eine Reihe von Integrations- und Systemtests mit Hilfe von U2TP erfasst, wobei hier vor allem auf die Tests der Protokollimplementierung eingegangen werden soll. Das Protokoll umfasst zwei Ebenen:

Sicherungsschicht: Diese Ebene stellt durch Längeninformationen, Sequenznummern und eine Prüfsumme den korrekten und vollständigen Empfang von Datenpaketen sicher.

Applikationsschicht: Die Applikationsdaten, also die Nutzdaten der Sicherungsschicht, stellen die eigentlichen PDUs (Protocol Data Units) dar, über die das RailSiTe und die Balisensimulation miteinander kommunizieren. Über sie werden beispielsweise Timing- oder Balisentelegrammdaten vom Labor an die Balisensimulation übertragen.

In **Bild 5** sind eine Reihe von Testfällen für die Sicherungsschicht des Protokolls als Klassifikationsbaum aufgeführt, der mit Hilfe des Classification Tree Editors (CTE) erstellt wurde (vgl. dazu auch [6] und [7]).

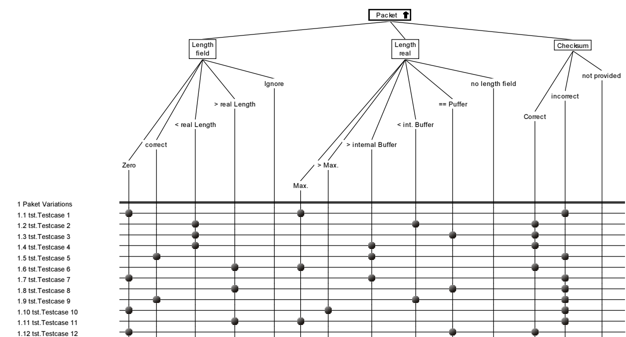

Bild 5: Klassifikationsbaum für eine Gruppe von Testfällen für die Balisensimulation (Ausschnitt)

Jede Zeile von Bild 5 entspricht dabei einem Testfall, der durch eine entsprechende Kombination von Paketlängen, gültigen / ungültigen Prüfsummen etc. gekennzeichnet ist.
Nach dieser systematischen Ableitung der grundsätzlichen Testfälle wurden diese dann mit Hilfe des U2TP spezifiziert und zu Testsequenzen zusammengestellt. Dazu wurden zunächst die Prüflingsschnittstellen und die Testumgebung, die aus einem normalen PC zur Stimulation der seriellen Schnittstelle besteht, genauer definiert. Im Sinne

des U2TP entspricht dies der *Test Architecture* mit dem Stimulations-PC als *Test Component* und dem Prüfling als *System Under Test*.
Die verschiedenen Datenpakete wurden einem *Data Pool* zugeordnet, der zusammen mit je einer Instanz des SUT und des Stimulations-PCs den *Test Context* bildet. Die einzelnen Testfälle sind Methoden dieser Klasse und spezifizieren die Abfolge der zu sendenden Pakete als Sequenzdiagramm (**Bild 6** und **Bild 7**).

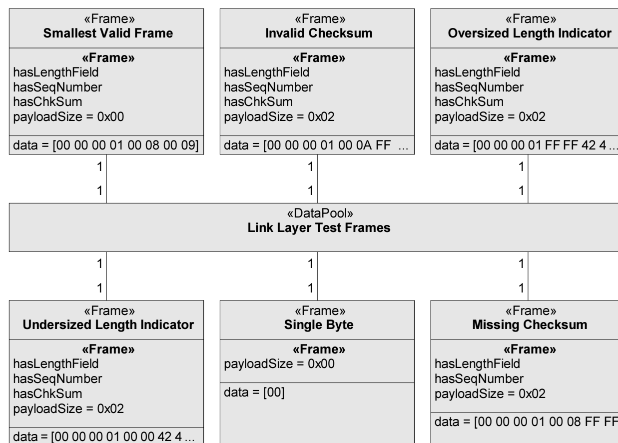

Bild 6: Definition eines *Data Pools*

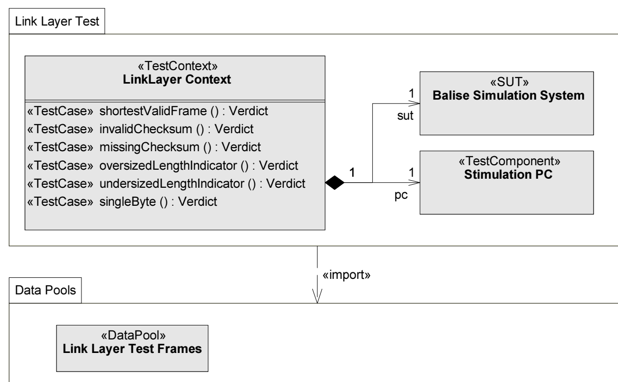

Bild 7: Einbindung des Datenpools und anderer Testkomponenten in einem Testkontext

In ähnlicher Weise wurden Testfälle für die Applikationsebene des Protokolls entworfen. Die Ableitung erfolgte in diesem Falle teils methodisch wie im Falle der Sicherungsschicht, teils aber auch basierend auf der Erfahrung des Testers. In einem weiteren *Data Pool* wurden Datenpakete mit verschiedenen PDUs und Variationen ihrer Parameter hinterlegt. Eine neue Kontextklasse instantiiert wiederum das SUT und einen Stimulations-PC und definiert verschiedene Testfälle als Methoden.
Wie **Bild 8** zeigt, ist es auch möglich, komplexe Testszenarien aus mehreren einfachen Testfällen zu erzeugen. Im gezeigten Falle werden die einfachen Schritte zur Übertragung eines langen bzw. kurzen Balisentelegramms an die Balisensimulation wiederverwendet, um die Speicherung und Funkübertragung einer vollständigen Balisentelegrammgruppe (Sequenz mehrerer Balisen) nachzustellen.

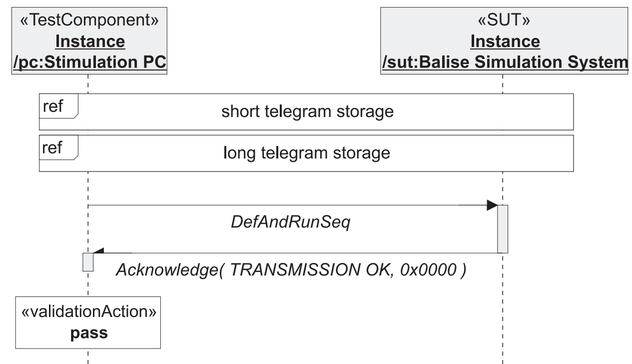

Bild 8: Testfallspezifikation als Sequenzdiagramm einschließlich Verweisen auf andere Testfälle / Diagramme

4 Ergebnisse der Fallstudie und Bewertung

Die Arbeit mit dem U2TP hat gezeigt, dass eine übersichtliche Darstellung von Testfällen und Testumgebungen mit UML möglich ist. Die Wiederverwendung von Testkomponenten in unterschiedlichen Konfigurationen (*Test Context*) des gleichen oder anderer Projekte erlaubt eine modulare Gestaltung der Tests. Darüber hinaus ermöglicht diese Modularisierung den Aufbau von Bibliotheken, aus denen zu einem späteren Zeitpunkt neue Testsysteme und Testbeschreibungen mit reduziertem Aufwand abgeleitet werden können.
Verglichen mit einer textuellen Darstellung - auch wenn sie strukturiert in Tabellenform vorliegt - ist die U2TP-Notation des Testverhaltens präziser, übersichtlicher und auch besser wartbar. Darüber hinaus wird in vielen textuellen Testbeschreibungen die Konfiguration des Testsystems nicht erfasst, was bei einer späteren Wiederholung oder Bewertung der Prüfung problematisch sein kann. Hier bietet das U2TP durch Rückgriff auf die mächtigen UML-Strukturdiagramme eine ausgezeichnete Abhilfe.
Durch ihren sehr viel formaleren Charakter (verglichen mit textuellen / tabellarischen Beschreibungen) bieten UML-Testfallbeschreibungen zudem die Grundlage für die Testautomatisierung und für reproduzierbare Ergebnisse.
Diese Vorteile werden durch eine gewisse Einarbeitungszeit in die Funktionsweise des U2TP im Speziellen und die UML im Allgemeinen erkauft. Außerdem verlangt die Vielzahl der Diagramme, die im Rahmen einer komplexeren Testbeschreibung entstehen, eine saubere Strukturierung und Namensgebung. Auch mit Hinblick auf die Wiederverwendung der Modellelemente in Bibliotheken ist diese Strukturierung von enormer Bedeutung, weil andernfalls das Auffinden von Komponenten stark erschwert wird. In diesem Zusammenhang kommt dem verwendeten UML-Tool entscheidende Bedeutung zu, weil dies entsprechende Konzepte unterstützen muss (Hierachisierung des Modells, (schreibgeschütztes) Einbinden von „Untermodellen“ in andere Modelle, Volltextsuche in Modellen, etc).

5 Ausblick

Eine folgerichtige Fortsetzung der Arbeiten mit U2TP ist die Fokussierung auf die Testautomatisierung. Dazu stehen verschiedene Wege offen, die alle auf einem Export der UML-Diagramme als XMI (*XML Metadata Interchange*, [8]) beruhen:

Konvertierung von XMI in ein Testskript: Die XMI-Beschreibung wird in eine gängige Skriptsprache wie beispielsweise Python oder Ruby übersetzt. Das Skript wird dann in der Testumgebung ausgeführt.

Direktes Ausführen von XMI: Die Testumgebung wird so adaptiert, dass sie die exportierten XMI-Daten direkt verarbeiten kann. Verglichen mit dem skriptbasierten Ansatz bietet diese Variante den Vorteil, dass kein vollständig lauffähiges Programm aus den XMI-Daten erzeugt werden muss.

Konvertierung in alternative Datenformate: Eine Übersetzung der XMI-Dateien in andere XML-basierte Formate (wie beispielsweise in [1] beschrieben) via XSLT (eXtensible Stylesheet Language Transformation) würde die direkte Verwendung in bestehenden Testaufbauten ermöglichen. Darüber hinaus beschreibt [3] auch ein Mapping der U2TP-Elemente zu TTCN-3 und zum JUnit-Framework, so dass U2TP-Testfälle durch aktuell verfügbare Tools zur Ausführung gebracht werden können.

Von allen Ansätzen erscheint der letztgenannte am vielversprechendsten und wird daher voraussichtlich Gegenstand der weiteren Arbeiten am IFS.
Neben der Testautomatisierung soll darüber hinaus auch die stärkere Einbindung des Testens in den Entwicklungsprozess untersucht werden. Das Ziel ist hierbei die Erstellung eines integralen Modells aus System- und Testbeschreibung mittels UML und geeigneter weiterer Profile. Auf diese Weise sollen unter anderem eine Nachverfolgung von Systemanforderungen bis auf die Testebene ermöglicht und Informationen zur Testabdeckung der Anforderungen bzw. der Implementierung abgeleitet werden.

6 Expertenwissen in Bildern?

U2TP stellt einen sinnvollen und funktionierenden Ansatz dar, um eine einheitliche, übersichtliche und formale Beschreibung von BlackBox-Tests auf allen Testebenen zu ermöglichen. Wie oben beschrieben kann eine U2TP-Beschreibung die Grundlage für Testautomatisierung und Bibliotheken von Testkomponenten bilden. Durch beides können Tests beschleunigt entwickelt bzw. ausgeführt und vorhandene Kapazitäten besser genutzt werden.
Das spezifische Expertenwissen des Testers liegt jedoch in der geschickten Definition und Kombination seiner Testfälle. Dieses Expertenwissen kann U2TP zwar in Bildern (sprich: Diagrammen) konservieren, aber es macht es nicht überflüssig. Der Tester wird also durch entsprechende UML-Testfallbibliotheken nicht ersetzt, sondern nur in Routineaufgaben, wie z. B. der Testfallbeschreibung, entlastet.

Literatur

[1] Ebrecht, Lars und Michael Meyer zu Hörste: *Formal Test Description – the central element for the automation of the process of testing*. In: Schnieder, Eckehardt und Géza Tarnai (Herausgeber): *FORMS/FORMAT 2004 – Formal Methods for Automation and Safety in Railway and Automotive Systems / Proceedings of Symposium FORMS/FORMAT 2004*, Braunschweig, 2004. Technische Universität Braunschweig.

[2] UNISIG: *Subset 076 - ETCS Test Plan*. AEIF, Brüssel, November 2005. Version 2.2.3. http://www.aeif.org/ccm/default.asp, Stand: 2006-08-01.

[3] Object Management Group: *UML Testing Profile*. Juli 2005. Version 1.0. http://www.omg.org/cgi-bin/apps/doc?formal/05-07-07.pdf, Stand: 2006-08-01.

[4] ETSI ES 201 873-1: *The Testing and Test Control Notation Version 3*. Juni 2005. Version 3.1.1.

[5] Willms, Martin: *Entwicklung von Hardware- und Softwarekomponenten für ein DSP-System zur Anbindung eines Zugfahrzeugrechners an ein Bahnsimulationslabor über Eurobalisen*. Diplomarbeit, FH Braunschweig / Wolfenbüttel, Januar 2006.

[6] Grochtmann, M. und K. Grimm: *Classification Trees for Partition Testing*. In: *Software Testing, Verification & Reliability*, Band 3, Seiten 63 – 82. 1993.

[7] Lehmann, Eckard und Joachim Wegener: *Test Case Design by Means of the CTE XL*. In: *Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000)*, Kopenhagen, Dänemark, Dezember 2000.

[8] Object Management Group: *MOF 2.0/XMI Mapping Specification, V2.1*. September 2005. http://www.omg.org/cgi-bin/doc?formal/2005-09-01, Stand: 2006-08-01.