Pragmatische Modellierung von Testfällen für Software im Automobil[1]

Zoltán Ádám Mann

sd&m AG
Carl-Wery-Str. 42
D-81739 München
zoltan.mann@sdm.de

Abstract: Der Artikel berichtet von der Entwicklung und dem Einsatz einer domänenspezifischen Sprache für die Spezifikation von Testfällen in einem Projekt bei einem Automobilhersteller. Mit diesem Ansatz werden Testfälle einmal auf fachlicher Ebene spezifiziert und dann auf verschiedenen technischen Plattformen ausgeführt. Dadurch wird Aufwand im Absicherungsprozess gespart und eine höhere Konsistenz der Testergebnisse erreicht.

1 Einleitung

Im Auftrag eines Automobilherstellers hat die sd&m AG in der Spezifikation und Absicherung einer verteilten softwarebasierten Fahrzeugfunktion mitgewirkt. Diese Fahrzeugfunktion wird durch eine Vielzahl von (Client-)Steuergeräten verwirklicht, die von unterschiedlichsten Zulieferern, teilweise auch mit verschiedenen Technologien implementiert werden. Wir haben eine Java-Simulation der verteilten Fahrzeugfunktion für eine Standard-PC-Umgebung angefertigt, die später auch zur Steuerung der Testautomatisierung in der Absicherung der Serienimplementierung benutzt wurde. Der hier vorgestellte Ansatz ist die Essenz dieser Projekterfahrung.

Im Hinblick auf die Testfälle hatten wir folgende Aufgaben. Wir mussten die Testfälle in einer menschenlesbaren Form für das Systemlastenheft spezifizieren und dann in der Java-Simulation implementieren. Diese Testfallimplementierung mussten wir dann ergänzen für das Testen von (1) einer generischen Implementierung, die vom Automobilhersteller den Zulieferern als Referenzimplementierung zur Verfügung gestellt wird (Standard-Core); (2) von Steuergeräten mit Anbindung über CAN-Bus (Controller Area Network); (3) von Steuergeräten mit Anbindung über MOST-Bus (Media Oriented System Transport) und (4) des Gesamtsystems.

Die Testfälle (insgesamt etwa 120) ließen sich in drei Gruppen unterteilen: Testfälle für einzelne Knoten des Systems (Client-Tests), Testfälle für das Gesamtsystem, sowie Testfälle, die nur zum Testen der Simulation nötig waren aber nicht zum Testen der

[1] Veröffentlicht in: Informatik 2006 – Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V., Lecture Notes in Informatics (LNI, Volume P-94), Band 2, pp. 231-235.

Serienimplementierung. Unter den Client-Tests gab es weitere Unterscheidungen je nach Anbindungstechnologie (siehe Abbildung 1).

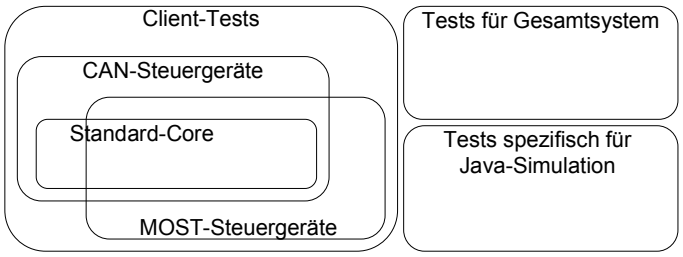

Abbildung 1: Gruppierung der Testfälle

Im Folgenden konzentrieren wir uns auf die Client-Tests. Diese sind fast ausschließlich Black-Box-Tests, die aus Senden von Nachrichten an das Client-Steuergerät und Abwarten der zeitgerechten Antwortnachricht vom Client-Steuergerät bestehen.

Die Standard-Core-Implementierung wurde vom PC aus über CAN angesprochen, so dass das Testen des Standard-Cores und der CAN-Steuergeräte sehr ähnlich war. Das Zusammenspiel mit anderen Funktionalitäten des Steuergerätes war bei der Standard-Core-Implementierung jedoch nur begrenzt möglich. So konnten einige Testfälle in der Absicherung des Standard-Cores nicht oder nicht vollständig ausgeführt werden. Zwischen CAN- und MOST-Steuergeräten gibt es einen wesentlichen Unterschied in der Art, wie das Steuergerät technisch anzusprechen ist, aber auf logischer Ebene sind die Testfälle in den beiden Fällen ähnlich. Andererseits sind die technischen Unterschiede zwischen den beiden Bussystemen – z.B. Kommunikationssicherheit, Größe der einzeln übertragbaren Datenblöcke und Broadcast-Fähigkeit – so gravierend, dass sie sich auch auf die Applikationsebene auswirken. So musste z. B. das korrekte Verhalten der Applikation beim Verlorengehen von Nachrichten für CAN aber nicht für MOST getestet werden, da MOST eine sichere Kommunikation garantiert, CAN aber nicht. Für MOST musste aber – im Gegensatz zu CAN – getestet werden, wie die Applikation mit großen Datenblöcken umgeht. Die Anzahl solcher technikspezifischer Testfälle war jedoch gering; die meisten Testfälle waren sowohl für CAN- als auch für MOST-Steuergeräte sinnvoll.

Die meisten Client-Testfälle waren relativ simpel und untereinander sehr ähnlich. Es gab einige, die deutlich komplizierter waren als der Rest, die aber aus den gleichen Bausteinen aufgebaut waren wie die einfachen. Die wichtigste Herausforderung bestand darin, ähnliche Testfälle für eine Reihe unterschiedlicher Plattformen (in Java simulierte Clients, CAN-Steuergeräte, MOST-Steuergeräte, Standard-Core) zu implementieren.

2 Domänenspezifische Sprache für Client-Testfälle

Diese Überlegungen haben zur Verwendung modellbasierter Methoden geführt. Das Ziel war dabei, die Testfälle mit einer abstrakten, plattformunabhängigen Notation zu definieren, damit sie nur einmal spezifiziert aber für mehrere Plattformen ausgeführt

werden können. Im Gegensatz zu anderen Ansätzen, die üblicherweise als modellbasiertes Testen bezeichnet werden [Pr05], handelt es sich also hierbei nicht um das Testen des Modells einer Anwendung. In unserem Fall hätte das bedeutet, dass ein formales und einheitliches Modell der Kommunikation mit den Steuergeräten vorliegt. Ein solches Modell wäre für unsere Zwecke vorteilhaft gewesen, existierte aber nicht. Um dennoch nicht auf die Vorteile einer modellbasierten Lösung verzichten zu müssen, haben wir den hier präsentierten pragmatischen Weg gewählt. Unser Ansatz bestand darin, eine maßgeschneiderte Sprache zu entwickeln, die einerseits die einfache Spezifikation der jeweiligen Testfälle ermöglicht, andererseits aber auch mit einfachen Mitteln in die Java-Simulation eingebunden werden kann. Aus dieser Überlegung haben wir uns für eine textuelle domänenspezifische Sprache [LS06] entschlossen.

Die Bausteine der Tests sind das Senden und Empfangen von Nachrichten. Im einfachsten Fall sendet der Tester eine Nachricht an das SUT (System Under Test) und erwartet eine Antwort in einer gegebenen Zeit. So ein Baustein kann mit verschiedenen Parametern parametrisiert werden, wie die zu sendende Nachricht, die zu erwartende Nachricht sowie die minimale und maximale Antwortzeit. Die zu erwartende Nachricht muss nicht vollständig spezifiziert werden: Es reicht, die Datenblöcke in der Nachricht anzugeben, deren Wert für den Erfolg des Testfalls maßgeblich ist. Es gibt auch weitere ähnliche Bausteine. Z. B. kann man angeben, dass nach dem Senden einer Nachricht in einer gegebenen Zeit keine Antwortnachricht kommen darf. Oder dass der Tester zwar keine Nachricht sendet, aber in einer gegebenen Zeit eine Nachricht vom SUT kommen muss. Abbildung 2 skizziert die Grammatik der Sprache in EBNF-Notation.

```
<testcase> := <first-block> <block>*
<first-block> := <param-spec> <command>
<block> := <param-spec-opt> <command>
<param-spec> := <inquiry-spec>
                <reply-spec>+
                <min-time-spec>
                <max-time-spec>
<param-spec-opt> := <inquiry-spec>?
                    <reply-spec>*
                    <min-time-spec>?
                    <max-time-spec>?
<inquiry-spec> := "messageToSend=new MessageInquiry("
                  <inquiry-params> ");"
<inquiry-params> := <id> | …
<reply-spec> := "functionIdInExpectedReply=" <id> ";" | …
<min-time-spec> := "minimalWaitTime=" <number> ";"
<max-time-spec> := "maximalWaitTime=" <number> ";"
<command> := "sendMessageAndWaitForReply();"
           | "sendMessageNoReply();"
           | "waitForReply();"
           | "waitNoReply();"
```

Abbildung 2: Grammatik der Modellierungssprache für Client-Testfälle

Aus diesen Bausteinen können auch kompliziertere Testfälle aufgebaut werden, wie z. B. in Abbildung 3 dargestellt. Dieser Testfall prüft, dass das SUT eine Anfrage mit genau drei Antwortnachrichten im Abstand von ungefähr 200 Millisekunden beantwortet. Man hätte die Modellierungssprache auch so auslegen können, dass man für eine Nachricht

gleich mehrere Antwortnachrichten mit separaten Timingparametern spezifizieren kann. Der präsentierte Ansatz löst aber diese Aufgabe mit einfacheren Konstrukten.

```
//Inquiry + 1st reply
messageToSend=new MessageInquiry(TEST_ID);
functionIdInExpectedReply=TEST_FUNC_ID;
minimalWaitTime=0;
maximalWaitTime=100;
sendMessageAndWaitForReply();

//2nd reply
minimalWaitTime=175;
maximalWaitTime=225;
waitForReply();

//3rd reply
waitForReply();

//wait one more period; no further reply should arrive
waitNoReply();
```

Abbildung 3: Beispiel für einen Testfall

Die einfache Syntax und die rein fachlichen Schlüsselwörter der Sprache machen die Testfälle leicht verständlich. Auch können Kommentare angegeben werden (wie auch in Abbildung 3 zu sehen). Damit ergibt sich eine menschenlesbare, jedoch formale Spezifikation der Testfälle, die ins Systemlastenheft aufgenommen werden kann.

3 Modellsteuerung

In einem üblichen modellbasierten Vorgehen würde man die Testfälle zuerst modellieren und dann entweder durch eine Transformation Code erzeugen oder das Modell direkt mit einem Interpreter verarbeiten. Im Projekt haben wir einen pragmatischeren Weg verfolgt: Die Modellsprache ist als eine Erweiterung von Java definiert, so dass die Testfälle eins-zu-eins in die Java-Simulation eingebettet werden können.

Es wäre natürlich auch möglich, die Testfälle mit einer Standardmodellsprache zu spezifizieren, z.B. mit TTCN oder mit Hilfe des UML Test-Profils. Die hier vorgestellte Lösung hat jedoch den Vorteil, dass die Testfälle dadurch nahtlos in die Java-Simulation – die für die Steuerung der Testautomatisierung gesetzt war – integriert werden konnten. Auch war es möglich, die wenigen Testfälle, die nicht in das Schema der meisten anderen gepasst haben, einfach in Java auszuprogrammieren, ohne die Modellierungssprache komplexer zu machen. Zudem hat die Integration in Java die Verwendung von Standard-Tools, wie z. B. JUnit, ermöglicht.

Technisch wurde das wie folgt umgesetzt. Alle Testfälle sind Java-Klassen abgeleitet von `ClientTest`, in dem die Attribute `minimalWaitTime`, `messageToSend` usw. sowie die Methoden `waitForReply()` usw. definiert sind. Das ermöglicht die Benutzung dieser Konstrukte in der Spezifikation der Testfälle. Die Implementierung

dieser Methoden hängt von der eingesetzten Plattform ab, deshalb delegieren diese Methoden die Arbeit an einen Adapter. Ein `ClientTest` kann mit verschiedenen Adaptern parametrisiert werden, je nach benutzter Plattform. Die Gegebenheiten der Plattform sind im Adapter gekapselt; die Testfälle sind technikfrei (siehe Abbildung 4).

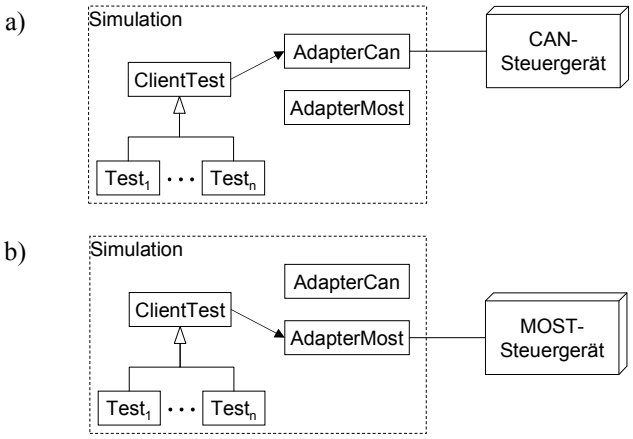

Abbildung 4: Die gleichen Testfälle können für verschiedene Plattformen benutzt werden

Wenn man nur das Endergebnis betrachtet, unterscheidet sich dieser Ansatz nicht wesentlich von einer herkömmlichen Architektur, in der auf die Trennung der Zuständigkeiten Wert gelegt wurde. Die Verfolgung eines modellbasierten Ansatzes hat auf natürliche Weise zu einer klaren Architektur geführt. Zudem ist die Option offen, die Testfälle aus der Simulation zu extrahieren und zu transformieren oder interpretieren, um eine losere Kopplung zu erreichen. Die meisten modellbasierten Ansätze gehen davon aus, dass die Modelle vom Code getrennt werden müssen, damit sie plattformunabhängig sein können. In diesem Fall ist das nicht nötig, da die Plattformunabhängigkeit sich darauf bezieht, dass die Testfälle nicht CAN- oder MOST-spezifisch notiert werden. Die Benutzung von Java ist neutral.

Durch diesen pragmatischen Ansatz konnten wir die wesentlichen Vorteile der Modellorientierung – Vermeidung der mehrfachen Durchführung stereotyper Programmierschritte, Trennung zwischen Technik und Fachlichkeit, einfache Handhabung mehrerer Plattformen (siehe auch [KM05]) – ohne einen hohen Aufwand für die Implementierung von komplexen Transformatoren oder Interpretern nutzen.

Literaturverzeichnis

[KM05] Kempa, M.; Mann, Z. Á.: Model Driven Architecture. Informatik Spektrum, August 2005, pp. 298-302.

[LS06] Ludwig, F.; Salger, F.: Werkzeuge zur domänenspezifischen Modellierung, OBJEKTspektrum 03/2006, pp. 16–20.

[Pr05] Pretschner, A.: Model-based testing, International Conference on Software Engineering, 2005, pp. 722–723.