Modellbasierte Testentwicklung

Aktivitätsdiagramme als verständliche und wiederverwendbare Testspezifikationen im expecco-Testsystem

Claus Gittinger, eXept Software AG

Email: info@exept.de
Tel: 07143-883040
Fax: 07143-8830444

http://www.exept.de
http://www.expecco.de

Motivation und Einführung

Die stetig wachsende Komplexität von Hard- und Softwaresystemen, sowie steigender Funktionsumfang bei stets kürzer werdenden Produktzyklen führte im letzten Jahrzehnt zum zunehmenden Einsatz von modellbasierten Technologien in der Entwicklung. Insbesondere sind hier Verfahren zu erwähnen, die auf formalen Spezifikationen wie UML-Diagrammen basieren.

Dagegen werden für das Testen oft textbasierte Script- oder kompilierte Programmiersprachen eingesetzt. Diese stellen hohe fachliche Anforderungen an die Testentwickler: müssen sie doch neben dem Domänenwissen über das zu testende System zusätzlich über fundierte Kenntnisse der Programmiersprache(n) verfügen, in denen die Tests implementiert sind. Häufig sind das systemnahe Sprachen wie C oder C++, bei denen sich der Testentwickler noch um Speicherverwaltung und andere Implementierungsdetails kümmern muss. In den vergangenen Jahren führte das zu einem stetig zunehmenden Aufwand für Entwicklung, Pflege und Weiterentwicklung von Tests und Testszenarien.

Expecco ist ein System, in welchem Testszenarien als Aktivitätsdiagramme in graphischer Form dargestellt und interpretativ abgearbeitet werden. Somit werden die vielfältigen Ausdrucksmöglichkeiten von UML-basierenden Modellbeschreibungen mit den kurzen Turn-Around-Zeiten interpretativer Systeme kombiniert, wodurch viele Beschränkungen traditioneller Testsysteme überwunden werden.

Folgende Anforderungen werden erfüllt:

- Tests werden auf einer hohen Abstraktionsebene als UML-Elemente graphisch formuliert, so dass sich Testentwickler auf die eigentliche Testerstellung konzentrieren können, ohne ständig mit den Problemen und Beschränkungen traditioneller Programmiersprachen konfrontiert zu werden.

- Aktivitätsdiagramme werden als einziges Mittel verwendet, um die Abläufe des Testsystems zu beschreiben. Dadurch sind Domänenexperten, Produktplaner bzw.

Ersteller von Anforderungsprofilen in der Lage, Testszenarien zu verstehen, zu modifizieren und zu erstellen – ohne dass Programmierkenntnisse vorausgesetzt werden.

- Da Testbeschreibungen unmittelbar interpretiert werden, kann der Ablauf von Tests im Detail angezeigt, und live animiert werden. Laufende Tests können angehalten, im Einzelschrittmodus ausgeführt und sogar während des Ablaufs modifiziert werden. Eine langwierige Generierung von Testprogrammen entfällt.

- Über umfangreiche Bibliotheken stehen sowohl Basismechanismen als auch firmen- bzw. projektspezifische (Aktivitäts-) Bausteine zur Verfügung.

- Einheitliche Beschreibung von Web-Applikations-Test (Capture/Replay), Funktions- und Lasttests.

Testszenarien werden als Test-Cases den einzelnen Use-Cases des zu testenden Systems zugeordnet. Über die direkte Ausführung im Testtool hinaus kann die Ausführung von Tests auch über ein Dashboard geplant, ferngesteuert sowie mit Anforderungen und Fehlermeldungen verknüpft werden.

Definition von “Modellbasiertem Testen”

Der Begriff *„Modellbasiertes Testen“* wird in der Literatur mit leicht unterschiedlichen Bedeutungen benutzt, daher erfolgt hier eine kurze Erläuterung unserer Interpretation.

Unter „modellbasiertem Testen“ wird im Allgemeinen die Generierung bzw. Abarbeitung von Testfällen/Testprogrammen aus einem Modell verstanden. Allerdings dient hier nicht selten das aus den Anforderungen entstandene „Entwicklermodell“, welches auch in der Entwicklungsabteilung in konkrete Software umgesetzt wird, als Grundlage zur Generierung von Testfällen.
Diese Vorgehensweise hat aber den fundamentalen Nachteil, dass sie niemals Fehler im Modell selbst erkennen kann. Sie prüft lediglich die korrekte Umsetzung des Modells in eine Implementierungssprache. Das ist dann sinnvoll, wenn die Umsetzung des Modells selbst nicht-generativ, sondern durch klassische Codierung erfolgt – wenn es also trotz sorgfältiger Modellierung noch zu Fehlern in der Implementierung kommen kann. Überflüssig wird diese Vorgehensweise aber, wenn die Umsetzung automatisch durch Generatoren aus dem Modell erfolgt. Denn hiermit wird dann im Endeffekt lediglich der Code-Generator, nicht aber das zu testende Produkt validiert. Diese Methodik ist also eher geeignet um Software-Tools wie Modellierungswerkzeuge, Generatoren oder Compiler selbst zu testen.

Wir plädieren daher für eine eigene, separate Modellierung der Tests selbst, und nennen dieses im Folgenden “*Testmodell*” - im Unterschied zum Entwicklermodell. Dieses Testmodell bildet die externen Schnittstellen sowie das extern messbare Verhalten des Entwicklermodells ab. Mit anderern Worten: während das Entwicklermodell die Internas des Systems beschreibt, wird mit dem Testmodell das möglicherweise vereinfachte Verhalten der Aussenwelt definiert bzw. simuliert. Analog zu ineinander passenden Puzzleteilen müssen die Testszenarien zu den Use-Cases des Entwicklermodells passen, und diese insofern komplementieren. Allerdings dürfen sie aus dem oben genannten Grund eben nicht automatisch aus einem gemeinsamen Modell erstellt werden, damit keine automatisch erfüllte Tautologie entsteht.

Selbstverständlich dürfen die Testmodelle gegebenenfalls zusammen mit den Entwicklermodellen in einem gemeinsamen physischen Container (Diagramm) definiert, abgelegt, versioniert und gewartet werden. Sie dürfen jedoch nicht zur Generierung von Zielsoftware dienen.

Interpretation vs. Generierung

Für die korrekte Durchführung eines Tests spielt es keine Rolle, ob der Testvorgang durch Generierung (Umsetzung in eine Programmiersprache) oder durch direkte Abarbeitung (d.h. Interpretation des Testmodells im Testsystem selbst) erfolgt.

Erfahrungsgemäß sind interpretative Systeme besser geeignet, Anforderungen wie kurze Turn-Around-Zeiten, Verfolgbarkeit, Kontrolle des Ablaufs (Einzelschritt, Haltepunkte, etc.) zu erfüllen. Da solche Anforderungen im Testbetrieb noch in weit höherem Maße gestellt werden als in der Entwicklung, haben wir uns in der hier beschriebenen Realisierung für ein interpretatives Abarbeiten des Modells entschieden.

Neben den kurzen Änderungszyklen (diese werden sofort, ohne Verzögerung wirksam) ermöglicht es eine sehr komfortable Entwicklungs- und Ausführungsumgebung für die Tests. So können Abarbeitung, Datenflüsse, Messwerte und Variableninhalte auf Wunsch graphisch im Modell live animiert und verfolgt werden. Ebenfalls werden Einzelschrittmodus, Haltepunkte usw. direkt in der Modellebene relativ einfach realisierbar. Moderne Implementierungsverfahren wie dynamische Just-In-Time Übersetzung sorgen dennoch für kurze Ausführungszeiten.

Welches Modell ?

Aus akademischer Sicht wäre sicher eine Modellierung mittels formaler, mathematischer Verfahren erstrebenswert. Allerdings scheitert dies bis heute an mehreren Problemen: einerseits erfordert sie ein sehr fundiertes, mathematisches Wissen, welches die wenigsten Entwickler und Tester aufbringen. Andererseits gelingt es bis heute nicht, Produkte der realen Welt, die also über mehr als nur demonstrativen Charakter verfügen, mit vertretbarem Aufwand formal zu spezifizieren. Im Übrigen sind manche der aktuell verwendeten Technologien (C, C++, Java, C#, etc.) mit ihrer ungenau definierten Semantik eher hinderlich. Leider konnten sich funktionale oder deklarative Systeme, die bewusst formal spezifiziert und validierbar angelegt wurden, nicht auf breiter Front durchsetzen. Somit werden viele Projekte in Programmierumgebungen realisiert, in denen falsche Zeiger, Zugriffe außerhalb definierter Datenbereiche, unerkannte Überläufe bei Additionen und insbesondere Seiteneffekte möglich sind.

Auf Entwicklerseite ist der Einsatz von UML (Unified Modelling Language) weit verbreitet. Dies ist eine aus verschiedenen Diagrammtypen bestehende graphische Darstellungsnorm, welche sich zur Beschreibung von Softwaresystemen sowohl informeller, also zur Dokumentation, als auch formeller Art, zur Generierung von Programmen eignet.

UML erlaubt es, sowohl statische als auch dynamische Aspekte eines Systems zu modellieren. Auf die Beschreibung der statischen Anteile des Modells (also der Klassen-diagramme) soll hier nicht weiter eingegangen werden: sie sind aus Sicht des Testers eher zweitrangig und können als Implementierungsdetail betrachtet werden. Dessen Fokus liegt primär auf dem externen Verhalten des zu testenden Systems, welches er zumeist als "*black box" betrachtet*.

Wichtig, da fehleranfälliger, sind für die Tester die dynamischen Aspekte des zu testenden Systems, die sich in UML durch Sequenz-, Zustands- und Aktivitäts-Diagramme darstellen lassen.

Sequenzdiagramme eignen sich primär zur Dokumentation und (mit Einschränkungen) zu Replay Tests, indem erwartete Reaktionen des Systems auf Stimuli beschrieben werden. Allerdings ist die Modellierung von kontextabhängigem, flexiblem Verhalten damit nur eingeschränkt und unter zusätzlicher Nutzung anderer Diagrammtypen möglich.

Zustandsdiagramme bieten weiterreichende Ausdrucksmöglichkeiten; es fehlen aber Sprachmittel zur komfortablen Beschreibung des Zeitverhaltens und von Parallelität,

Ausnahmebehandlungen usw., so daß Zustandsdiagramme ebenfalls zusätzliche Beschreibungen erfordern.

Aktivitätsdiagramme sind in der Lage alle Aspekte eines Systems semantisch vollständig zu beschreiben, vorrausgesetzt, das Verhalten der Ein- und Ausgänge (Pins), sowie die einem Bearbeitungsschritt zugrunde liegende Aktivität sind hinreichend genau festgelegt. Aktivitätsdiagramme können als Oberklasse von Flussdiagrammen und Petrinetzen betrachtet werden, und haben damit eine entsprechende mathematische Basis. So können Deadlock-Situationen, Abhängigkeiten, Vollständigkeit, Zyklen und andere Eigenschaften mit formalen Methoden relativ einfach validiert werden. Gleichzeitig eignen sie sich auch zur Modellierung komplexer, insbesondere parallel ausgeführter Vorgänge.

Alles in allem stellen Aktivitätsdiagramme derzeit sicher die attraktivste Form der Modellierung dynamischer Vorgänge dar. Dies wurde auch von den Standardisierungs-gremien erkannt – so wurden gerade die Aktivitätsdiagramme in der neueren UML2.0 Version stark erweitert und bieten nun weit mehr Möglichkeiten als in der Vorgängerversion.

Eigenschaften von Aktivitätsdiagrammen

Ein Aktivitätsdiagramm besteht aus einer Menge von (Verarbeitungs-) Schritten sowie Verbindungen ihrer Ein- und Ausgänge. Jedem Schritt ist eine Aktivität zugeordnet, welche durch den Datenfluss über die Verbindungen synchronisiert werden. Erst durch die Verfügbarkeit der zu einem Schritt benötigten Daten wird für diesen eine Aktivität erzeugt, welche seine Eingangswerte liest, die Aktion durchführt, und schließlich seinerseits Resultate an andere Verarbeitungsschritte weiterreicht. Die Daten können im einfachsten Fall einzelne Bits, bool'sche oder numerische Messwerte oder Triggerinformationen sein. Natürlich ist es auch möglich, komplexe Datenstrukturen, Tabellen, Dokumente, Datenbankhandles oder Objektinstanzen weiter zureichen. Wichtig zu bemerken ist, dass Daten sich nur in jeweils einer Bearbeitungsstation befinden sollten. Werden z.B. Formulare durch das System gereicht, kann dann sehr leicht automatisch sicher gestellt werden, dass Zugriffe (insbesondere Modifikationen) immer implizit synchronisiert erfolgen. Der Testentwickler muss sich also nicht mit Synchronisationsmitteln wie Semaphoren, Monitoren oder kritischen Regionen auseinander setzen.

Aktivitätsdiagramme werden sowohl von Programmierern als auch von nicht-Programmierern benutzt und verstanden werden. Sie werden daher auch zur Dokumenta-tion, zum Informationsaustausch und als Diskussionsbasis verwandt. Fragt man Entwickler nach einem einfach verständlichen Bild seines Programms, zeichnen viele intuitiv ein Aktivitätsdiagramm. Es ist daher naheliegend, diese nicht nur zusätzlich, zur Dokumentation der Programmlogik, sondern zur ausschließlichen semantischen Beschreibung von Systemen zu nutzen.

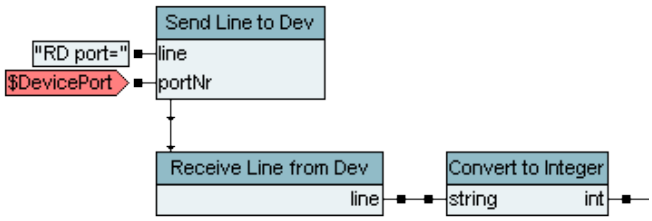
Das folgende einfache Beispiel soll exemplarisch zeigen wie verständlich und weniger fehleranfällig ein Aktivätsdiagramm im Vergleich zu klassischer, textueller Programmierung ist. Ein Programmfragment zum Auslesen eines Messwertes von einem Gerät könnte in einer Sprache wie C/C++ formuliert werden als:

```
char *lineBuffer; int voltage;
char lineBuffer[ MAX_LINE_SIZE ];
fprintf(file, "RD port=%d\n", DevicePort);
fgets(lineBuffer, MAX_LINE_SIZE, file);
sscanf(lineBuffer, "%d", &voltage);
```

in Java/C# und vergleichbaren:

```
int voltage;
file.put("RD port="+DevicePort.asString);
voltage = file.getLine().asInteger();
```

als Aktivitätsdiagram:



Eingefleischte Programmierer werden nun sicher einwenden, dass sie in ihrer Programmiersprache schneller entwickeln können; um so mehr, wenn sie über langjährige Erfahrung verfügen. Allerdings werden sie zur Dokumentation ihres Programms dann möglicherweise doch zusätzliche Schaubilder ähnlich dem obigren Diagramm zeichnen – insbesondere, wenn das Programm später von anderen Personen gewartet, erweitert oder portiert werden muss. Ausserdem könnte mancher einwenden, dass die Aufgabe anders oder mit weniger Codezeilen gelöst werden könnte (der C-Programmierer möglicherweise mit Makros).

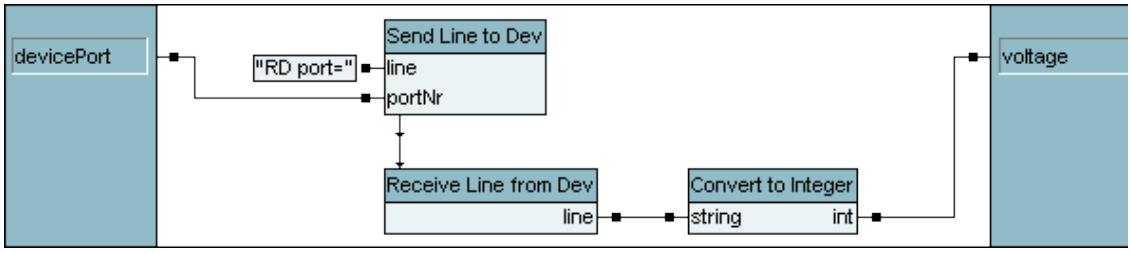
Es geht aber heutzutage bei der (Test-)Entwicklung vornehmlich darum, inwiefern sich Programme und Algorithmen kommunizieren lassen, ob sie leicht erweiterbar sind, und ob sie auch noch ein Jahr später verstanden werden. All dies insbesondere auch von nicht-Programierern und Domänenexperten, die gerade im Testumfeld besonders aktiv sind.

Um den Unterschied noch stärker herauszuheben, soll die Aufgabe durch eine in der realen Welt häufig vorkommende Problematik erschwert werden: der Code soll so erweitert werden, daß gleichzeitig Messwerte von zwei verschiedenen Geräten gelesen werden. Dazu sollen die Abfragen in einer Parallelverarbeitung gestartet und anschließend synchronisiert werden.

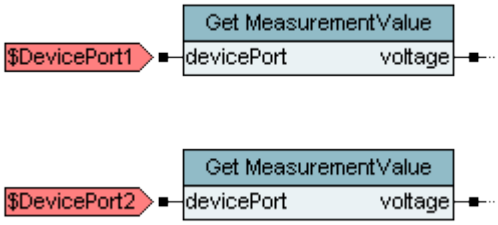
In C/C++ wird daraus dann:

```
static void
GetValue(struct portAndData data) {
  char *lineBuffer; int voltage;
  char lineBuffer[ MAX_LINE_SIZE ];
  fprintf(file, "RD port=%d\n", data.port);
  fgets(lineBuffer, MAX_LINE_SIZE, file);
  sscanf(lineBuffer, "%d", &voltage);
  data.voltage = voltage;
}
...
f1data.port = DevicePort1;
f2data.port = DevicePort2;
pthread_create(&pHandle1,NULL,(void *)GetValue,&f1data);
pthread_create(&pHandle2,NULL,(void *)GetValue,&f2data);
pthread_join(pHandle1, NULL);
pthread_join(pHandle2, NULL);
voltage1 = f1data.voltage;
voltage2 = f2data.voltage;
...
```

Analog zum C-Code wird auch im Aktivitätsdiagramm die Geräteabfrage als eigenständiger Verarbeitungsschritt “Get MeasurementValue” definiert (Refactoring):



und diese Unteraktivität dann zweimal im ursprünglichen Diagramm platziert:

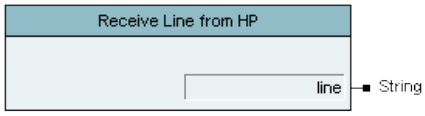


[Der aufmerksame Leser wird bemerken, daß obiger C-Code nicht zuverlässig ist, da keine Synchronisation des Zugriffs auf die “file”-Variable erfolgt. Analog setzt das Aktivitätsdiagramm vorraus, daß eine solche Synchronisation in den Send bzw. Receive Bausteinen über eine exklusive Resourcedefinition erfolgt. Es wurde hier darauf verzichtet, um das Beispiel klein und leicht verständlich zu halten.]

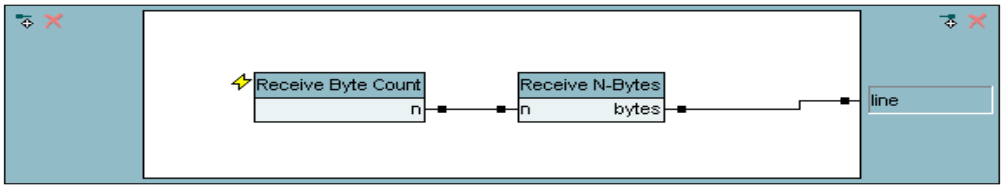
[Da die Java-Version eventuell noch umfangreicher wird, überlassen wir diese dem Leser zur Übung.]

Elementare und Zusammengesetzte Aktionen

Die einzelnen Schritte können ihrerseits wieder als (Unter-)Aktivitätsdiagramm oder als Elementarfunktion realisiert werden. Ein Unteraktivitätsdiagramm fasst mehrere Aktionen in einem einzelnen Baustein zusammen, womit ein Äquivalent zu Unterprogrammen oder Software-ICs zur Verfügung steht. Einmal definierte Aktionsbausteine können an verschiedenen Stellen in anderen Aktivitätsdiagrammen platziert und damit wiederverwendet werden. In obigem Beispiel könnte die Aktion:



aus zwei einfacheren Komponenten zusammengesetzt sein:

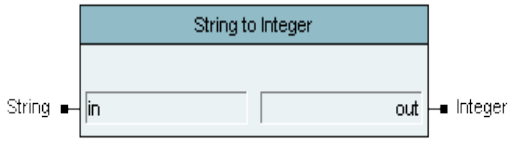


Diese Rekursion muss natürlich irgendwann enden. Nach wiederholtem “hineinzoomen” in die Bausteine erreichen wir schliesslich sogenannte “*Elementare Aktionen*”, welche als atomare Funktionseinheiten betrachtet werden können. Ein elementarer Schritt kann als

eingebaute Grundfunktion, Aufruf einer existierenden Bibliotheksfunktion (DLL-Call), als Remote-Procedure-Call (SOAP, COM, DCOM), oder als Aufruf eines externen Programms realisiert werden. Ausserdem ist es möglich, elementare Aktionen als Skript in einem eingebauten oder externen Skriptinterpreter auszuführen (Shell, Perl, Python, etc.).

Der integrierte Skriptinterpreter erlaubt es, Elementarfunktionen in der bekannten JavaScript-Syntax zu definieren. Diese werden zunächst zu Bytecode, und zur Ausführung automatisch dynamisch weiter in schnellen Maschinencode übersetzt. Buildvorgänge (make) oder Übersetzeraufrufe werden vom Tester nicht erwartet. Die oben beschriebene Übersetzung erfolgt während der Eingabe des Diagramms, so daß dieses sofort, aus dem Editor heraus, ausführbar ist.

Ein einfacher Elementarblock zur Konvertierung von Zeichenketten wird definiert als:




```
execute() {
    var s = in.value;
    out.value( s.asInteger() );
}
```

Bausteine zur Definition von Testszenarien

Selbstverständlich kann ein Testszenario nur dann vollständig ausgeführt werden, wenn sämtliche (und insbesondere *Low-Level*-) Funktionen in Form von Elementarbausteinen realisiert wurden und zur Modellierung komplexerer Diagramme zur Verfügung stehen. Sind diese Basisfunktionen erst einmal entwickelt worden, können weitere, auch komplexe, Testszenarien schnell erstellt, modifiziert oder erweitert werden, ohne dass hierzu Programmierkenntnisse nötig sind. Umgekehrt können Tests auch bereits sehr früh im Entwicklungszyklus formuliert und ausgeführt werden. Fehlende Funktionalität kann dann aber lediglich durch Platzhalterbausteine simuliert werden. Diese Testbeschreibungen dienen dann schon sehr früh im Projektzyklus zur Dokumentation der durchzuführenden Tests.

In der Praxis wird sowohl nach der Bottom-Up-, als auch der Top-Down-Vorgehensweise entwickelt. Bei der Bottom-Up-Methode werden komplexere Sequenzen aus elementaren Bausteinen zusammengesetzt. Dem gegenüber steht die Top-Down-Entwicklung, bei der ausgehend von den übergeordneten Testszenarien die Aktionen schrittweise verfeinert werden.

Erweiterte Sprachmittel in Aktivitätsdiagrammen

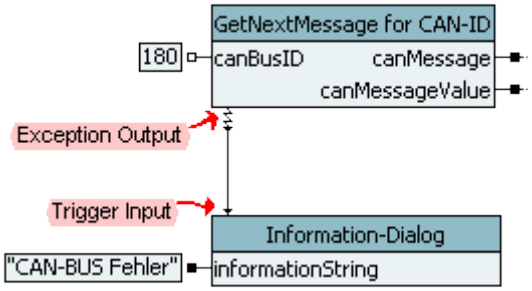
Auch in ihrer neuesten Ausprägung werden viele semantische Eigenschaften von Aktivitätsdiagrammen nicht definiert bzw. als sogenannte Stereotypes dem Generator bzw. Interpreter überlassen. Damit Testfälle vollständig und ausschließlich durch Aktivitätsdiagramme beschrieben werden können, wurden diverse Erweiterungen / Konkretisierungen realisiert:

1. **Datenübernahmeverhalten an Eingängen**
   Eingänge können sowohl als Parameter- als auch als reguläre Pins definiert werden. Parameterpins halten typischerweise statische Konfigurationsparameter und entsprechen statischen oder globalen Daten traditioneller Programmiersprachen. Reguläre Pins hingegen realisieren einen synchronisierten Datenfluss über eine Warteschlange. Anstehende Daten lösen in der Reihenfolge ihres Eintreffens neue Aktivitäten aus, welche dann diese Werte aus der Warteschlange “konsumieren”. Da mehrere Aktivitäten gleichzeitig ausgeführt werden können, ergibt sich hierdurch auch eine einfach verständliche Definition von Parallelverarbeitung.

2. **Pufferung und Synchronisation der Datenflüsse**
   Sowohl die maximale Anzahl gepufferter Werte in einer Warteschlange, als auch die maximale Anzahl parallel ausgeführter Aktivitäten kann über entsprechende Attribute definiert werden. Womit Produzer-Consumer-Beziehungen gesteuert werden.

3. **Ausnahme-Ausgang (*Exception-Pin*)**
   Dieser erhält im Falle eines Fehlers in einem Bearbeitungsschritt detaillierte Informationen über die stattgefundene Ausnahme. Sie kann (wie andere Datenpakete) weitergereicht, ausgewertet oder zur Fehlerbereinigung bzw. Benachrichtigung verwandt werden.

   Wie in anderen Programmiersystemen werden unbehandelte Fehler weiter gereicht – hier an das umgebende Diagramm, welches dann seinerseits eine Fehlerbehandlung durchführt. Wird ein Fehler auf diese Weise bis zur Testfallebene weiter gereicht, so gilt der Testfall als defekt.



4. **Abbruch-Eingang (*Cancel-Pin*)**
   Über diesen Eingang können bereits laufende Aktionen abgebrochen (storniert) werden.

5. **Zeitüberwachungs-Eingang**
   Hierdurch können Aktivitäten nach einem statisch vorgegebenen oder dynamisch berechneten Zeitintervall abgebrochen oder benachrichtigt werden. Da dies auch durch Kombination eines Zeitverzögerungsbausteins mit obigem Abbruch-Eingang realisiert werden könnte, dient diese primär der besseren Darstellung und Übersichtlichkeit.

6. **(Re-) Konfigurierbarkeit und Parametrisierung über Variablen**
   Neben den dynamisch, über Pins weitergereichten Daten muss es auch statisch vorbelegbare Variablen geben, welche zur Steuerung des Testablaufs oder zur Konfiguration (Port- oder IP-Adressen, Geräteadressen, Testumgebungen etc.) dienen können. Solche Variablen werden in Umgebungen gesammelt, die hierarchisch organisiert sind (sogenanntes *lexical scoping*, wobei ein Diagramm oder Unterdiagramm jeweils einen Scope darstellt).

7. **Virtuelle Aktivitäten**
   Analog zu polymorphen Interfaces bei Klassendefinitionen ist es sinnvoll, wenn auch Aktivitäten in abstrakter Form definierbar und später (auch dynamisch polymorph) konkretisierbar bzw. redefinierbar sind. Damit wird es z.B. sehr einfach möglich, Schnittstellenbausteine in generischen Testszenarien auszutauschen bzw. dynamisch an konkrete Aufgaben anzupassen.

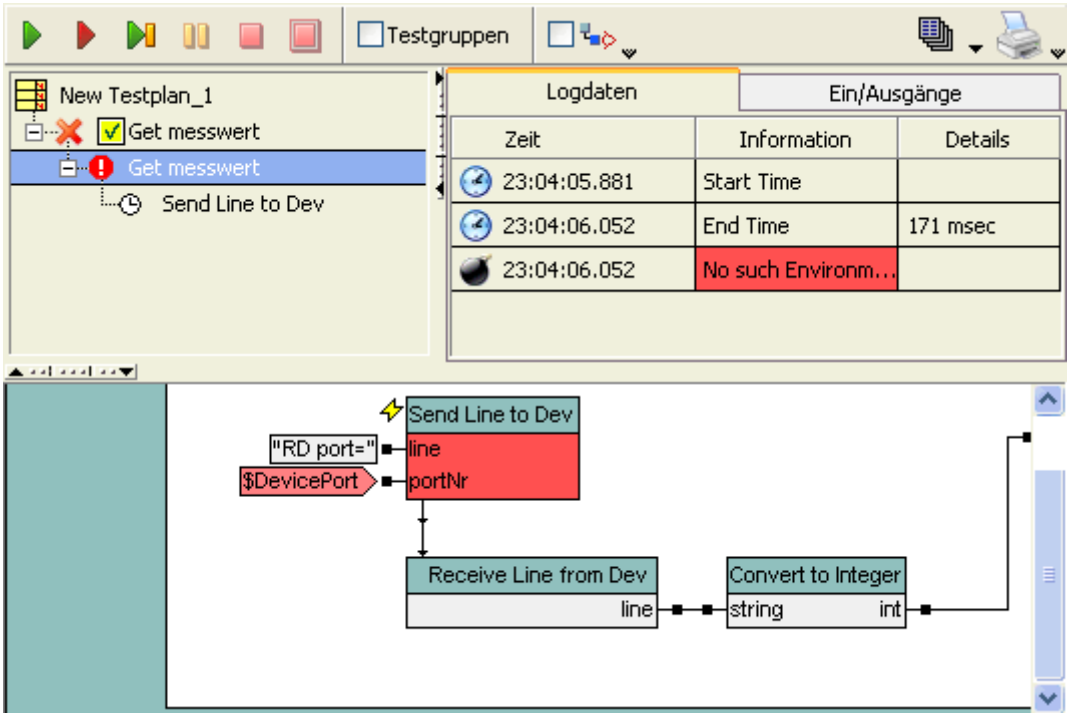
8. **Betriebsmittel (*Resource*-) Verwaltung**
   Zur Durchführung von Tests werden in vielen Fällen Betriebsmittel wie Messgeräte, Prüfsignalgeneratoren oder Prüflinge (*Boards*, *Ports*, etc.) benötigt. Diese Betriebsmittel sollten ebenfalls Teil des Modells sein und zur Synchronisation von Testabläufen dienen. Auch Datenbanklocks, kritische Regionen und andere Synchronisationselemente lassen sich als Betriebsmittel modellieren.

9. **Reflektive Elemente**
   Teile des Testsystems selbst können über Bausteine angesprochen und modifiziert werden. So können z.B. benutzerspezifische Konfigurationsdateien automatisch eingelesen, und in Umgebungsvariablen abgelegt werden, ohne dass dazu eine explizite Programmierung notwendig wäre.

Analog dazu gibt es Bausteine, welche den Zustand der Ablaufumgebung, der Tests, der Netzwerke und sogar einzelner Bausteine reflektiert. Somit ist es sogar möglich, neue Bausteine dynamisch zu erzeugen (dies wurde in einem Projekt genutzt, um bestehende Testpläne, welche in formalisiertem Englisch vorlagen, einzulesen, und automatisch in Testpläne und Diagramme umzusetzen).
Ebenso erfolgen Reservierung und Freigabe von Betriebsmitteln über benutzerdefinierte Aktionen. So kann z.B. ein Test, welcher ein bestimmtes Messgerät benötigt, dieses von sich aus vom Operator anfordern (indem er eine email an den Operator schickt, und auf dessen Antwort wartet bevor der eigentliche Test anläuft).

Interpretation vs. Generation

Obschon bereits oben erwähnt, soll hier nochmal betont werden, daß es sich hier um ein interpretatives System handelt (das allerdings durch einen Just-In-Time Compiler unterstützt wird). Der Wegfall eines Übersetzungsschrittes bietet viele Vorteile: Turn-Around-Zeiten im Sekundenbereich, die Möglichkeit von Änderungen im laufenden Test (tatsächlich sogar, ohne diesen überhaupt anzuhalten). Des weiteren ist die verbesserte Fehlersuche zu erwähnen. Verfolgbarkeit, Visualisierung der Ausführung und Daten, Haltepunkte und Einzelschrittausführung sind sowohl auf Diagramm- als auch Elementarblockebene möglich.

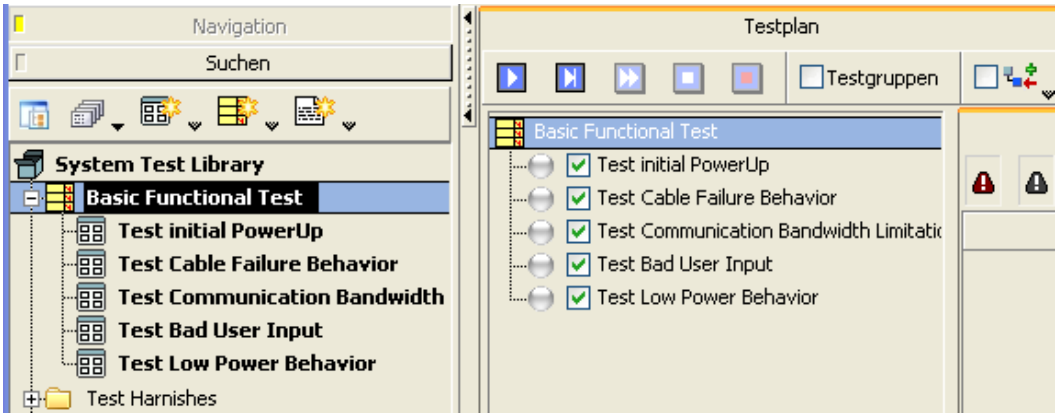


[Testplan mit Live-Animation des gerade ausgeführten Schrittes und der übergebenen Daten. Hier wird ein Fehler gemeldet, da ein Parameter nicht korrekt definiert wurde.]

Es sollte nicht vergessen werden, dass die Entwicklung von Tests seinerseits eine Entwicklertätigkeit darstellt, und viele Gemeinsamkeiten mit der Entwicklung des eigentlichen Produkts aufweist. Allerdings ist die Entwicklung von Testszenarien oft sehr explorativer Natur, indem der Testentwickler versucht, Schwachstellen und Grenzsituationen zu finden und gezielt zu reproduzieren. Für diese Art der Entwicklung sind interpretative, explorative Entwicklungsumgebungen traditionell prädestiniert.

Organisation von Testplänen

Testpläne bestehen aus einer Sammlung einzelner Testfälle, welche ihrerseits entweder wieder als eine Menge von Untertests oder als Aktivitätsdiagramm beschrieben werden. Typischerweise werden jedem Use-Case des Entwicklermodells ein oder mehrere Testfälle zugeordnet, welche ihrerseits jeweils durch eine Testaktivität definiert sind.



Automatische Erzeugung von Traces und Testberichten

Da sämtliche Datenflüsse zwischen den Diagrammelementen optional vermerkt werden, kann diese Information sehr einfach visualisiert, in Dateien oder Datenbanken geschrieben werden, oder zur automatischen Erzeugung eines detaillierten Testberichts dienen. Dazu erzeugt ein Generator entsprechende Berichte im HTML, XML oder PDF Format. Ausserdem können solche Traces auch als Grundlage von automatischen Ist-Soll-Vergleichen heranziehen. Natürlich können Logeinträge auch explizit aus Elementar- oder Diagrammcode mittels entsprechenden Schnittstellen abgesetzt werden.

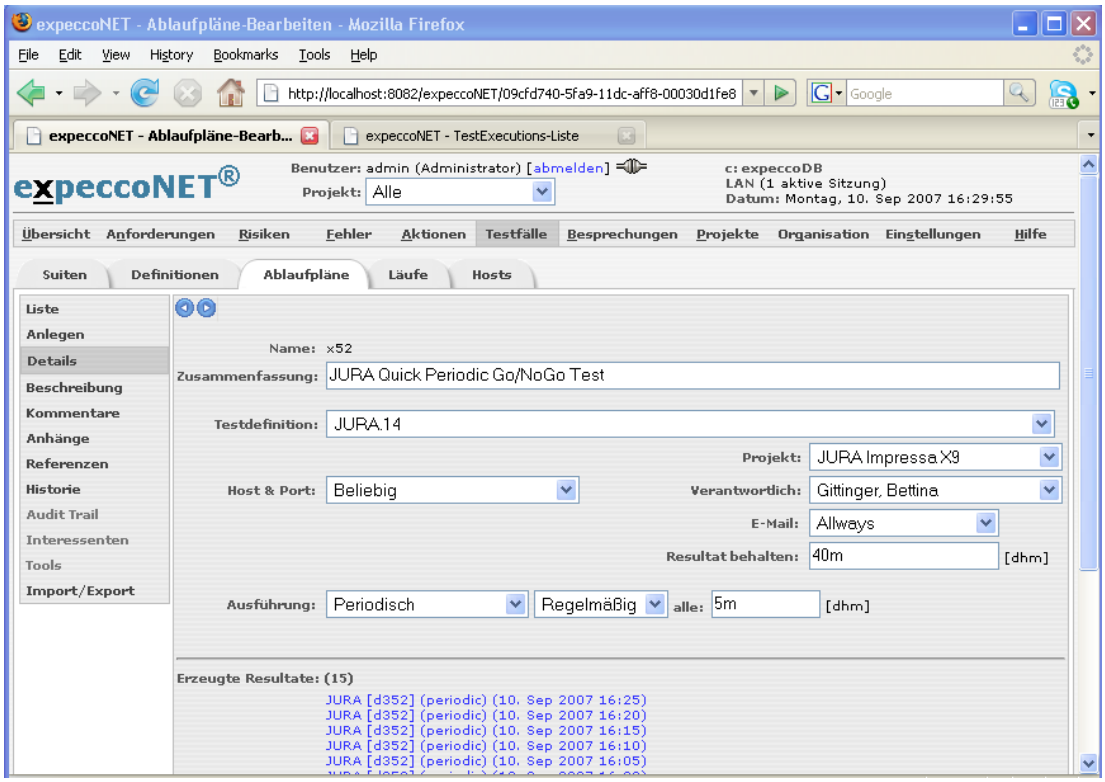
Import aufgezeichneter Szenarien

Verschiedene Fremdformate können mittels Import zur automatischen und halb-automatischen Erzeugung von Aktivitätsdiagrammen dienen. Besonders erwähnenswert ist hierbei der Import von Aufzeichnungen einer Web-Sitzung, wie sie z.B. das freie "*Selenium*"-Tool liefert. So aufgezeichnete Webaktivitäten können in wiederverwendbare Teilaktivitäten (Login, Bestellung, usw.) zerlegt werden oder durch semantische Tests erweitert werden.

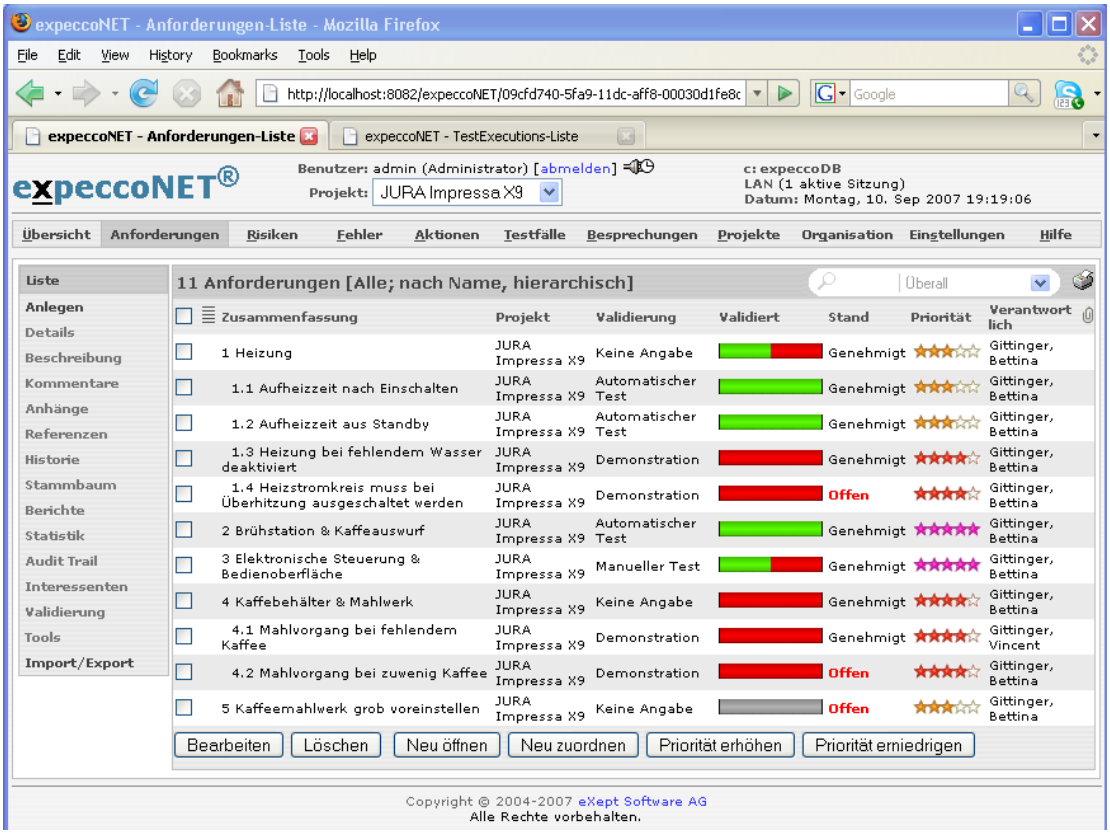
Eine klassisches Beispiel ist hier der Webshop, dessen Datenbank auch im Fehlerfall oder nach Abbrüchen und Ausfällen konsistent sein muss. Eine einmal aufgezeichnete Interaktion eines Benutzers kann durch Hinzufügen eines entsprechenden Datenbankprüfbaustein sehr einfach erweitert werden um die tatsächlich durchgeführte Aktion im Webserver zu überprüfen.

Testmanagement und Automatisierung

Die Testausführung kann sowohl manuell als auch vollautomatisch (ohne Bediener-GUI) erfolgen. Zur automatischen Ausführung werden die Testpläne mittels der Web-basierten QM Plattform *expeccoNET* in einer zentralen Datenbank verwaltet, versioniert und insbesondere zur automatischen Ausführung parametrisiert. Hier kann die Ausführung einzelner Testpläne zeitgesteuert (z.B. „jeden Sonntag, über Nacht") oder durch ein externes Ereignis (z.B. „Änderung im Modul X") angestoßen werden.



Die so gesammelten Daten werden automatisch archiviert und dienen zur Analyse, Ermittlung des Projekfortschrittes, zur Erstellung von Statistiken oder zur Generierung von Prüfberichten (Ausgangskontrolle). Aus der Zuordnung von Testfällen zu Projekt-anforderungen entsteht eine zeitnahe Übersicht des aktuellen Entwicklungs- und Fehlerstandes:



Erfahrungen und Zusammenfassung

Das expecco-Testsystem wird in verschiedenen Projekten zur automatischen Testausführung, sowie zur Installation und Konfiguration von Geräten im Bereich Telekommunikation, Anlagen- und Maschinenbau, Medizintechnik sowie Automotive eingesetzt.

Testfälle sind selbst dokumentierend und die zugrunde liegenden Konzepte werden schnell verstanden – dies insbesondere auch von Nicht-Programmierern. Somit wurde in allen Anwendungsfällen die Kommunikation zwischen den Domänenexperten und Testern einerseits und den Entwicklern andererseits spürbar verbessert, insbesondere da diese innerhalb weniger Stunden in der Lage waren, Testszenarien zu verstehen, zu modifizieren und zu erweitern. Gleichzeitig konnten diese den Entwicklern sofort als Feedback zur Reproduktion von Fehlern dienen, wodurch die mittlere Fehlerbehebungszeit spürbar reduziert wurde.

Literatur

Andreas Spillner, Tilo Linz, Basiswissen Softwaretest, August 2005

Mario Winter: Qualitätssicherung für objektorientierte Software: Anforderungsermittlung und Test gegen die Anforderungsspezifikation, 2000

Wolfgang Prenninger, Alexander Pretschner: Abstractions for Model-Based Testing
Electronic Notes in Theoretical Computer Science, Vol116 pp58-71, 2005

Mario Friske, Holger Schlingloff: Von Use Cases zu Test Cases
TU Braunschweig Report, 2005

Harald Störrle, Jan Hendrik Hausmann:
Towards a Formal Semantics of UML 2.0 Activities
Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, pp117-128

Larry Apfelbaum, John Doyle: Model Based Testing
Software Quality Week Conferences, Mai 1997

S.R. Dalal, A. Jain et al.: Model Based Testing in Practice
International Conference on Software Engineering, 1999; ACM Press, pp285-294

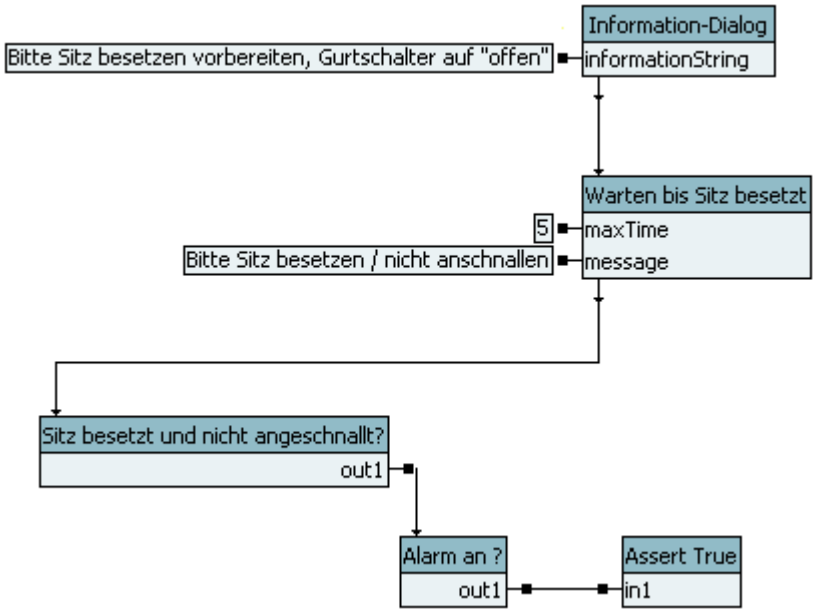
Robert V. Binder: Object Oriented Testing: Myth and Reality
Object Magazine, Mai 1995

Harry Robinson: Intelligent Test Automation
Software Testing and Quality Engineering Magazine, 2000

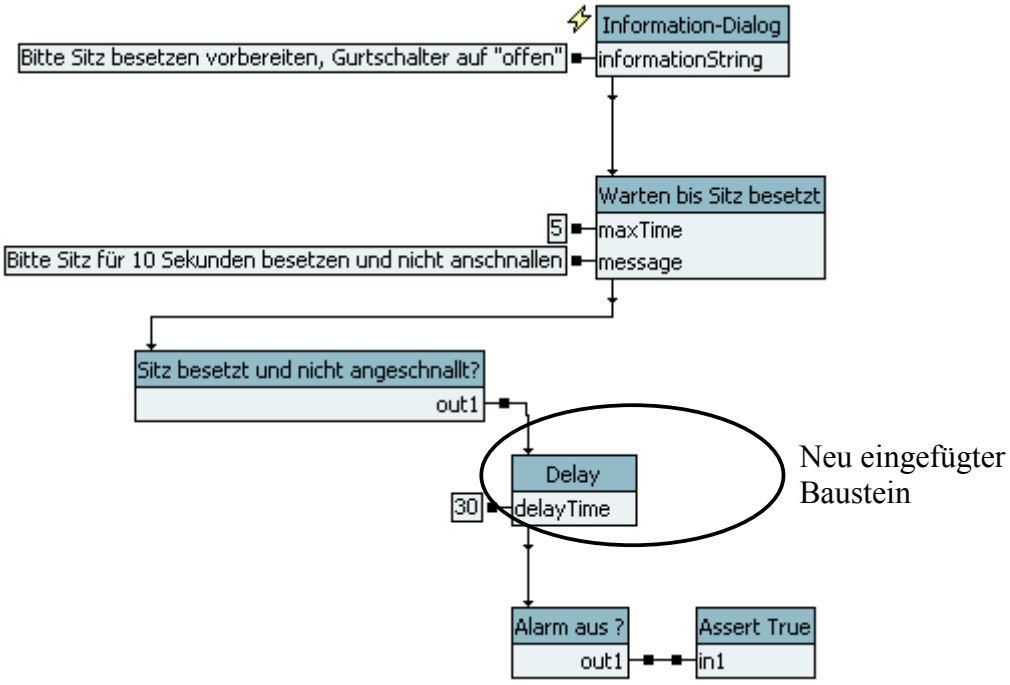
Andreas Bartsch, Stefan Vogel: expecco Whitepaper, www.expecco.de

Anhang: Beispiel für einfache, konkrete Testfälle

Die folgenden zwei Testfälle sollen das hier Beschriebene verdeutlichen. Der erste Testfall überprüft die Software des Sicherheitsgurtes: falls der Beifahrersitz besetzt ist, und der Gurt nicht angelegt wurde, soll ein Signalton erfolgen. Bei vorliegenden Elementarbausteinen zur CAN-Bus Abfrage, könnte ein solcher Testfall in etwa folgende Form haben:



Es ist offensichtlich, dass auch nicht-Programmierer derart formulierte Testszenarien sehr leicht verstehen, und erweitern können. Ein weiterer, leicht daraus formbarer Testfall ergibt sich zum Beispiel durch die Prüfung, ob jener Alarm auch nach einer gewissen Zeit wieder beendet wird (damit die abgelegte Aktentasche auf dem Beifahrersitz nicht zu einem Nervenzusammenbruch führt):



Die gezeigten Bausteine sind ihrerseits entweder als Elementarbausteine oder Unterdiagramme realisiert.

Elementarbausteine können konkret mittels einer Programmier- oder Scriptsprache, sowie durch Aufruf eines bestehenden Services (SOAP, RPC oder DLL-Aufruf) implementiert werden. Der oben abgebildete *Delay*-Baustein wurde aus einer Standardbibliothek mit häufig benötigten Elementarbausteinen in das Diagramm eingefügt; seine Realisierung erfolgte in einem Javascript Dialekt: