Berücksichtigung zeitlicher Anforderungen bei der Generierung von Testfällen für den Integrationstest

Witali Aswolinskiy Yingfan Lei Mike Heidrich
Fraunhofer-Einrichtung für Systeme der Kommunikationstechnik ESK
Hansastraße 32
80686 München

Kurzfassung

Verteilte Systeme bestehen aus nebenläufigen Komponenten, die miteinander über Nachrichten oder Ereignisse kommunizieren. Ob das System sich korrekt verhält, bestimmt nicht nur der Inhalt und die Reihenfolge von Ereignissen, sondern auch ihr zeitlicher Abstand. Die klassische Lösung mittels Pfadsuche in Zustandsdigrammen für den Integrationstest zustandsbehafteter Komponenten basiert auf der Konstruktion eines globalen Zustandsraumes. Bei der Pfadgenerierung bleibt jedoch ein wesentliches Problem ungelöst: zur Generierungszeit sind manche Pfade nicht bestimmbar, ohne Berücksichtigung der aktuellen Werte der Attribute, die zur Laufzeit den Prozess steuern. Die in dieser Arbeit vorgestellte Lösung umfasst die zufallsbasierte bzw. evolutionäre Bestimmung der an das System zu sendenden Ereignisse und ihres Abstands. Die generierten Testfälle prüfen, ob eine Implementierung die modellierten Anforderungen erfüllt.

1. Einleitung und Motivation

Nicht nur die Reihenfolge von Ereignissen bzw. Funktionsaufrufen, sondern auch das Timing, d.h. der Abstand zwischen den Ereignissen bzw. die Antwortzeit bei den Funktionsaufrufen, spielt beim Testen von verteilten Software-Systemen eine wichtige Rolle. In dieser Arbeit betrachten wir vorgegebene Zeitrahmen für die Abstände zwischen Ereignissen als zeitliche Anforderungen. Diese sollten frühzeitig getestet werden, da Änderungen in der Architektur des Systems mit dem Fortschreiten des Projektes immer kostspieliger werden.

Als die kleinste zu testende Einheit wird die Komponente gewählt. Komponententests sind Unittest und prüfen die Interna einer einzelnen Komponente. Beim Integrationstest werden Fehler im Zusammenspiel der Komponenten gesucht [1]. Wenn die Anforderungen im Modell festgehalten werden, können Testfälle aus dem Modell heraus generiert werden [2]. In [9] werden mehrere Werkzeuge und Methoden vorgestellt.

Beim Test von Komponenten, die durch Zustandsdiagramme definiert werden, ist die Transitions-überdeckung durch All-Round-Trips ein häufiges Ziel [1] [3] [8]. Das Zustandsdiagramm wird als Karte benutzt um Pfade durch den Zustandsautomaten zu finden. Eine Transition entspricht einem Schritt auf dem Pfad. In einem Testfall werden die Schritte eines solchen Pfads ausgeführt. Je nach Art des Systems entspricht ein Schritt einem Methodenaufruf oder dem Senden eines Ereignisses und endet in einem neuen Zustand der Komponente.

Diese Technik kann auch für den Integrationstest verwendet werden. In [9] wurde gezeigt, wie aus den Zustandsdiagrammen kommunizierender Komponenten ein Modell globalen Verhaltens abgeleitet und getestet werden kann.

Bisher wurden jedoch Probleme beim automatisierten Test dieser zeitlichen oder im Allgemeinen nicht-funktionaler Anforderungen durch Integrationstests nicht analysiert. Der vorliegende Beitrag schlägt hierfür eine Lösung vor.

Der Artikel ist wie folgt strukturiert. Abschnitt 2 erklärt, warum die klassische Testgenerierung mit Pfadsuche im Zustandsdiagramm für den Komponententest geeignet, aber für den Integrationstest eine andere Strategie nötig ist. Abschnitt 4 beschreibt unser Verfahren zur Testgenerierung für den Integrationstest. Abschließend folgt unsere Schlussfolgerung in Abschnitt 5.

2. Problemstellung bei der Generierung von Testfällen für den Integrationstest zeitlicher Anforderungen

Zur Veranschaulichung unseres Konzeptes wird in dieser Arbeit durchgängig ein U-Bahn-System als Fallbeispiel verwendet. Das U-Bahn-System besteht aus den Komponenten `Tuersteuerung`, `Zugsteuerung` und `Fahrsteuerung`, die mit Zustandsdiagrammen modelliert wurden und den Komponenten `Notfallsystem` und `Welt`, die jeweils ein Ereignis senden und empfangen können. In Abbildung 1 ist das Zustandsdiagramm der Komponente `Zugsteuerung` beispielhaft dargestellt.

Während der Transition vom Zustand `FremdobjektReaktion` in den Zustand `Faehrt`, wird das Ereignis `Vollbremsung` nur dann ausgelöst, wenn ein Fremdobjekt auf den Gleisen nahe dem Zug festgestellt wurde (siehe Abbildung 1). Genauso könnte nicht nur das Senden eines Ereignisses, sondern auch der Übergang in

einen anderen Zustand vom Inhalt eines Attributes abhängen. Der tatsächliche Zustand einer Komponente hat also einen logischen Anteil, der durch das Zustandsdiagramm beschrieben ist, und einen physikalischen Anteil, der sich aus den Werten der Attribute der Komponenten ergibt [6]. Wenn Testdaten generiert werden, ist der physikalische Zustandsanteil ein Problem: Zur Generierungszeit des Testfalls kann der Durchlauf eines ausgewählten Pfades nicht garantiert werden. Im U-Bahn-Beispiel müsste der Testtreiber ein Fremdobjekt in der Nähe des Zuges auslösen können, um die Transition mit dem Wächter (Guard) „if nahe=true“ in UML `[nahe=true]` zu schalten. Das Wissen, was „in der Nähe“ bedeutet, hat jedoch nur der Tester. Der Testgenerator kann allein vom Modell ausgehend dieses Konzept nicht erfassen.

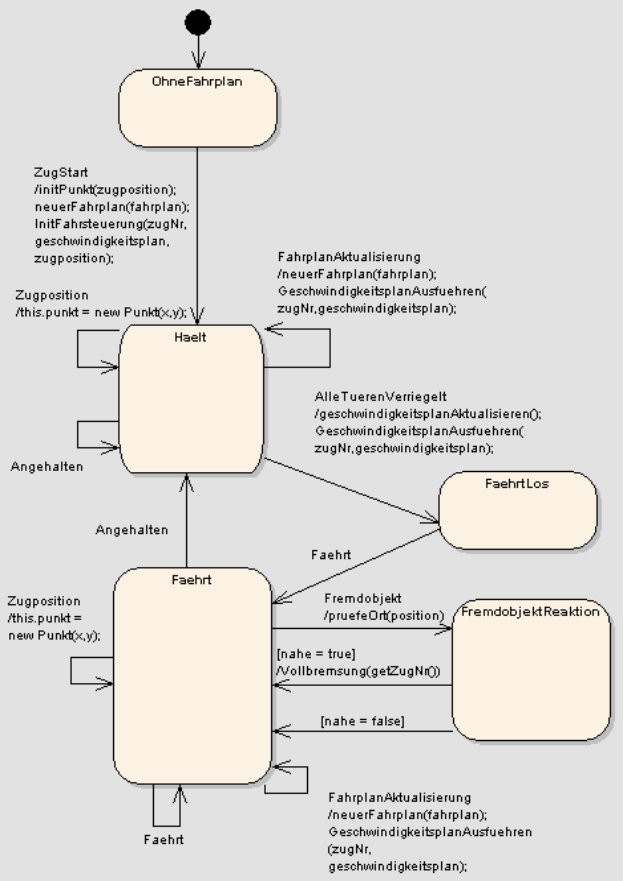

Abbildung 1: Zustandsdiagramm für die Komponente Zugsteuerung

Beim Komponententest gibt es für dieses Problem eine pragmatische Lösung: Die Guards können ausgespielt werden, indem die Attribute der Komponente, die die Guards prüfen, direkt im Testfall gesetzt werden. So könnte beim Komponententest von `Zugsteuerung` die im Guard geprüfte Variable `nahe` auf `true` bzw. `false` gesetzt werden, um die jeweilige Transition zu schalten. Diese Lösung funktioniert nur, wenn die Implementierung es zulässt, dass die Attribute (wie `nahe` in Abbildung 1) unmittelbar vor der Überprüfung im Guard gesetzt werden können. Zur Betrachtung solcher Fälle, erstellte nun unser Generator über Pfadsuche für die Komponente `Zugsteuerung` neun Testfälle. Einer davon ist in Abbildung 2 dargestellt.

```
public void testTC9() {
    ZugStart zugStart;
    AlleTuerenVerriegelt alleTuerenVerriegelt;
    Faehrt faehrt;
    Fremdobjekt fremdobjekt;
    zugStart = new ZugStart(new Fahrplan(),
            new Zugposition(0.8098153f, 83.84069f, -862961229, -2083185189,
                5998744566032698367L, 315947201), 783532894);
    zugStart.setZugNr(getTC().getZugNr());
    send(zugStart);
    waitFor("Haelt");
    assertInState(ZugsteuerungStates.Haelt);
    alleTuerenVerriegelt = new AlleTuerenVerriegelt(35596331);
    alleTuerenVerriegelt.setZugNr(getTC().getZugNr());
    send(alleTuerenVerriegelt);
    waitFor("FaehrtLos");
    assertInState(ZugsteuerungStates.FaehrtLos);
    faehrt = new Faehrt(1647255870);
    faehrt.setZugNr(getTC().getZugNr());
    send(faehrt);
    waitFor("Faehrt");
    assertInState(ZugsteuerungStates.Faehrt);
    fremdobjekt = new Fremdobjekt(new Punkt(-234334934, -1981757031));
    send(fremdobjekt);
    waitFor("FremdobjektReaktion");
    assertInState(ZugsteuerungStates.FremdobjektReaktion);
    checkObserverFaults();
}
```

Abbildung 2: Ein Testfall für den Komponententest

Beim Integrationstest würde jedoch der Eingriff in das Innere der Komponente zu Inkonsistenz führen. Der Grund ist die Kommunikation der getesteten Komponenten. Die Informationen in einer Komponente werden verändert, ohne dass die anderen Komponenten davon wissen. Der Inhalt der Komponente bleibt valide, jedoch wird die Information in der Gesamtheit der Komponenten inkonsistent.

Für den Integrationstest komplexer Systeme ist eine andere Strategie erforderlich. In unserer Lösung prüfen die generierten Testfälle, ob der Zug innerhalb des modellierten Zeitrahmens bremst, wenn er bremst.

3. Modellierungskonzept für den Integrationstest zeitlicher Anforderungen

Der manuelle Test der genannten zeitlichen Anforderungen, ist aufwendig, da die Reihenfolge und der zeitliche Abstand der Ereignisse – das Timing, vom Zustand der Komponenten, dem Inhalt der Ereignisse, dem Zustand der Systemumgebung und der Kombination all dieser Faktoren abhängt.

Im Folgenden werden die Konzepte der Modellierung und der Testfallgenerierung für diese Zeitanforderungen vorgestellt.

3.1 Modellierung der Zeitanforderungen anhand des Fallbeispiels

Bevor ein Zug losfahren darf, müssen alle Türen verriegelt sein. An der Erfüllung dieser Anforderung sind drei Komponenten beteiligt: `Zugsteuerung`, `Fahrsteuerung` und `Tuersteuerung`. Die Anforderung kann durch ein Zustandsdiagramm modelliert werden. Abbildung 3 zeigt den StatechartObserver für die Verriegelung. Die Ereignisfolge `Angehalten-Faehrt` ist verboten und die Folge `Angehalten-AlleTuerenVerriegelt-Faehrt` ist erlaubt.

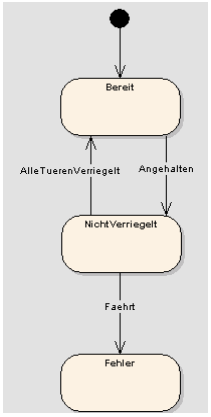

Abbildung 3: Modellierung der Reihenfolge der Ereignisse

Diese Modellierung unterstützt Zustandsmonitioring, d.h. die aktivierten Zustände und damit die entsprechenden Ereignisse sind während der Laufzeit beobachtbar. Unser Konzept ist inspiriert durch `StateChartObserver` [5] und `Statechart Assertions` [4] .

Wenn die unerlaubte Ereignisreihenfolge `Angehalten-Faehrt` auftritt, signalisiert der `StatechartObserver` das Versagen des Systems durch den Übertritt in den Fehlerzustand.

Im U-Bahn-System ist auch eine Anforderung denkbar, bei der der Zug innerhalb von 100 ms bremsen muss, wenn in seiner Nähe ein Fremdobjekt auf den Gleisen festgestellt wird. Sie ist in Abbildung 4 modelliert. Der erlaubte Zeitrahmen von 100 ms für die Einleitung der Vollbremsung wird durch `after(100)` ausgedrückt.

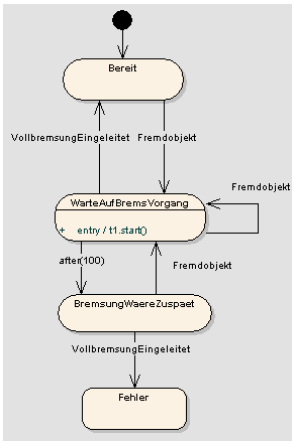

Abbildung 4: StateChartObserver für Vollbremsung

3.2 Generierung der Testfälle für die betrachteten Zeitanforderungen

Das grundsätzliche Ziel beim Integrationstest von Zeitanforderungen ist es, Situationen zu forcieren, in denen Zeit- und Reihenfolgebedingungen gelten. Je häufiger solche Situationen vorkommen, desto größer ist die Chance, dass ein Fehler auftritt. Hierzu müssen die Reihenfolge und der Zeitabstand der Ereignisse so bestimmt werden, dass möglichst viele Transitionen stattfinden – es soll also gutes Timing gefördert werden, bei dem die Ereignisse von den Komponenten bearbeitet werden und nicht von ihnen abprallen.

Ereignissequenzen können per Zufall, analytisch oder evolutionär erzeugt werden. Am einfachsten ist die Generierung nach dem Zufallsprinzip. Wenn man jedoch dem Zufall überlässt, welche Ereignisse in welcher Reihenfolge an die Komponenten versendet werden, ist die Wahrscheinlichkeit bei komplexen Zustandsdiagrammen gering, dass die Komponenten zur Kommunikation angeregt werden.

Das Problem bei der Zufallsgenerierung kann umgangen werden, wenn der Teststreiber zur Laufzeit über die Zustandsänderungen der Komponenten unterrichtet wird und somit weiß, welches Ereignis als nächstes an die Komponente geschickt werden soll, sodass diese in den nächsten Zustand übergeht. Aber auch dieses Vorgehen hat Nachteile. Die zusätzliche Prozessorlast verändert das Timing und in der Zeit, in der das nächste zu sendende Ereignis bestimmt wird, kann kein Ereignis gesendet werden.

Bei dem evolutionären Ansatz wird der Zufall iterativ und inkrementell in Richtung des besseren Timing gelenkt. Ereignissequenzen, die die Komponenten anregen und `StatechartObserver` aktivieren, dienen jeweils als Vorlage für den darauffolgenden Generierungsschritt. So entstehen inkrementell Ereignissequenzen, die nicht nur

vom getesteten System akzeptiert werden, sondern auch Situationen fördern, in denen Fehler auftreten können.

Am U-Bahn-System wurden der zufallsbasierte und der evolutionäre Ansatz erprobt. Das überraschende Ergebnis war, dass das zufallsbasierte Verfahren besser als das evolutionäre Verfahren abschnitt. Mit der zufallsbasierten Generierung der Ereignissequenzen wurden schneller Fehlersituationen herbeigeführt als mit der evolutionären.

Dass die Vorteile des evolutionären Verfahrens nicht ausgeschöpft wurden, ist zum einen durch die logisch-physikalische Natur der getesteten Komponenten bedingt (vgl. Abschnitt 3.1) und zum anderen durch die Komplexität des U-Bahn-Systems.

Das implementierte evolutionäre Verfahren optimierte das Timing, d.h. die Ereignisse wurden so gesendet, dass Transitionen stattfinden konnten. Dass sie tatsächlich stattfanden, hing aber auch von den Guards und den Werten der Attribute der Komponenten ab.

Die Zugkomponenten sind einfach: Zug- und Fahrsteuerung umfassen jeweils fünf bzw. vier Zustände. Auch mit dem zufallsbasierten Verfahren gelangten die Testfälle bis in den Zustand `Faehrt` der Zugsteuerung, in dem das Fremdobjekt auftreten kann. Die Zugsteuerung reagiert auf ein Fremdobjekt nur, wenn der Zug sich in der Nähe befindet. Ob er sich in der Nähe befindet, wird durch die Fahranweisungen im Ereignis `ZugStart` und `Weg` bestimmt. Das implementierte evolutionäre Verfahren veränderte diese Ereignisse nach der ersten, zufallsbasierten Erzeugung nur unwesentlich. Beim zufallsbasierten Verfahren wurden diese Ereignisse immer wieder neu generiert – damit wurde ein größerer Datenraum abgedeckt als beim evolutionären Test. Die Wahrscheinlichkeit, dass ein Zug sich in der Nähe eines Fremdobjektes befand, war beim zufallsbasierten Test größer.

4. Schlussfolgerung

Modellbasierter Test nicht-funktionaler Anforderungen verspricht für den Integrationstest komplexer, reaktiver Systeme zielgerichtete Transitionsüberdeckung. Wichtig ist es dabei, Strategien zu entwickeln, um den Datenraum möglichst gut zu erkunden.

Mit unserer Lösung, die auf Zustandsmonitoring basiert, können wir sowohl die Reihenfolge der Ereignisse/Funktionsaufrufe als auch die modellierten zeitlichen Anforderungen, z.B. die Antwortzeit testen.

Um eine allgemeingültigere Aussage zu machen, welche Teststrategien (zufallsbasiert, evolutionär) besser geeignet sind, müssen in Zukunft noch weitere Fallstudien untersucht werden.

5. Literatur

[1] Binder, R. V. 1999. Testing Object-Oriented Systems – Models, Patterns and Tools, Addison-Wesley

[2] Broy, Manfred; Jonsson, Bengt; et al. 2005. Model-based Testing of Reactive Systems. Springer

[3] Chow, T. S. 1978. Testing Software Design Modelled by Finite-State Machines. IEEE transactions on Software Engineering

[4] Drusinsky, Doron. Modeling and verification using UML statecharts: Aworking guide to reactive system design, runtime monitoring, and execution-based model checking; Elsevier: Amsterdam, 2006.

[5] Graf, Susanne; Ober, Ileana; Ober, Iulian. Model checking of UML models via a mapping to communicating extended timed automata. 2004

[6] Gross, Hans-Gerhard. Component-based software testing with UML; Springer: Berlin, 2005.

[7] Hartmann, Jean; Imoberdorf, Claudio; Meisinger, Michael. 2000. UML-based Integration Testing, Proceedings of the 2000 ACM SIGSOFT, International Symposium on Software Testing and Analysis

[8] Lee, D, Yannakakish, M. 1996. Principles and Methods of Testing Finite State Machines – A Survey. Proc. IEEE

[9] Utting, Mark; Legeard, Bruno. 2007. Practical model-based testing: A tools approach; Elsevier/Morgan Kaufmann Publ.: Amsterdam.