UPPAAL-Modelle als ausführbare Spezifikation in Java

Michael Goedicke, Moritz Balz, Michael Striewe
Universität Duisburg-Essen
{michael.goedicke, moritz.balz, michael.striewe}@s3.uni-due.de

Abstract: Zustandsautomaten können als Modelle umfassend spezifiziert, simuliert und validiert werden. Werden sie im Programmcode umgesetzt, bleibt der formale Bezug zu diesen Modellen jedoch in der Regel nicht vollständig erhalten. Dieser Beitrag untersucht, wie Modellentwicklung am Beispiel von UPPAAL und die Umsetzung im Java-Programmcode gleichzeitig geschehen können, indem eine Methode zur Spezifikation von Zustandsautomaten im Programmcode skizziert wird. Dieser Programmcode kann sowohl als Zustandsautomat ausgeführt, als auch ad hoc in ein UPPAAL-Modell transformiert werden.

1 Einleitung

Modellgetriebene Technologien und Werkzeuge unterstützen die abstrakte Spezifikation von Zustandsautomaten von der Modellierung über Simulation bis zum Model Checking. So können bereits zum Entwicklungszeitpunkt der Modelle Fehleranalysen erfolgen und eine korrekte Funktionsweise sichergestellt werden. Durch die Hierarchisierung von Automaten kann zudem eine Strukturierung der Softwarearchitektur erreicht werden.

Auf einer weniger hohen Abstraktionsebene erfolgt allerdings die Umsetzung im Programmcode in Programmiersprachen wie etwa Java. Gleich, ob der Code von Hand geschrieben oder auf Basis des Modells generiert wird, bleibt die Semantik des Modells im ausführbaren Code nicht erhalten. Das Wissen über den Bezug zwischen Modell und Code ist nur indirekt vorhanden, im günstigen Fall über eine (oft nicht-formale) Dokumentation, im schlechteren Fall nur über das implizite Wissen der Entwickler [TG03]. Diese Problematik setzt der Verwendung von Modellen in der Softwareentwicklung enge Grenzen: Der häufig verwendete Ansatz der Code-Generierung aus Modellen macht eine Rückkopplung ebenso unmöglich wie die manuelle Entwicklung. Bekannte Techniken des Model Round-Trip Engineering [SK04] versuchen, Inkonsistenzen zu benennen und durch geeignete Manipulationen zu beheben und damit bereits mehr zu leisten als reine Codegenerierung und Reverse-Engineering. Auch dann sind allerdings Inkonsistenzen aufgrund der unterschiedlichen Abstraktionsebenen von Code und Modellen nicht vollständig zu vermeiden.

Ausgangspunkt für diesen Beitrag ist ein in einem Industrieprojekt entwickelter Lastgenerator [BSGT03], dessen Steuerung der Lasterzeugung über Zustandsautomaten realisiert und zu dem Zweck in UPPAAL [LPY97] modelliert und dokumentiert wurde. Um den Zusammenhang zwischen Programmcode und Modell zu erhalten stellt sich hier die Auf-

gabe, eine Umsetzung von Zustandsautomaten im Programmcode zu finden, die sowohl direkt ausführbar ist als auch als formale Spezifikation dient. Diese soll alle Informationen enthalten, die zur Laufzeit-Interpretation und gleichzeitig zu Modellierung und Model Checking zur Entwicklungszeit notwendig sind: Informationen über Abläufe, Variablenbelegungen sowie Vor- und Nachbedingungen sollen permanent explizit und konsistent verfügbar sein, so dass semantische Inkonsistenzen nicht entstehen können.

2 Ansatz

Da mit vorhandenen Mitteln der modellbasierten Entwicklung das Problem nicht zu beheben ist, soll versucht werden, die Beziehung zwischen Modell und Programmcode langfristig zu erhalten. Eine parallele aber ansonsten unabhängige Pflege beider Ebenen ist nicht realistisch. Daher soll eine Struktur für den in jedem Fall zur Ausführung benötigten Programmcode gefunden werden, die zur Laufzeit als Zustandsautomat interpretiert wird und gleichzeitig eine Rückkopplung ins Modell erlaubt. Die statischen Bestandteile des Modells – Zustände, Übergänge und Bedingungen, die für das Schalten einer Transition erfüllt sein müssen [PGS01, Kap. 4] – sollen dazu als Klassenstrukturen definiert und die Klassen mit Metadaten über Nachfolgezustände und Bedingungen angereichert werden. In den Transitionen wird Java-Code ausgeführt, der anwendungsspezifisch und beliebig komplex ist, so dass er als Black Box nicht betrachtet werden soll. Nicht umsetzbar ist daher die statische Prüfung von Nachbedingungen, die das Setzen und Verändern von Variablen abbilden. Diese sollen allerdings trotzdem definiert werden, so dass zur Entwicklungszeit die Validerung des Modells und zur Laufzeit eine dynamische Überprüfung zum Debugging des Modells möglich ist.

Es wird also das Ziel verfolgt, das UPPAAL-Modell so vollständig im Java-Code abzubilden, dass es semantisch interpretiert werden kann. So wird erreicht, dass Modell und Programmcode nicht getrennt gepflegt werden und der Zusammenhang erhalten bleibt. Das Modell soll aus dem Java-Code ausgelesen und in UPPAAL eingespeist werden können, um dort auf einer höheren Abstraktionsebene validiert zu werden. Entwickelt werden sollen für diesen Zweck neben der erforderlichen Spezifikation drei Werkzeuge:

- Eine Ausführungskomponente, die zur Laufzeit die Strukturen interpretiert und Zustandsübergänge entsprechend erfüllter Bedingungen aktiviert oder untergeordnete Automaten aufruft.
- Ein Java-nach-UPPAAL-Konverter, der die Klassenstrukturen und Metadaten des Java-Codes interpretiert und das UPPAAL-Modell aufbaut.
- Ein Konverter, der UPPAAL-Modelle in Java transformiert. Die Herausforderung ist hierbei die Validierung, dass nur unterstützte Funktionalität im Modell verwendet wird und somit keine Informationen verloren gehen. Diese Komponente ist optional, da sie für den Erhalt des Zusammenhangs zwischen Programmcode und Modell nicht benötigt wird. Sie dient lediglich der Vereinfachung der erstmaligen Erzeugung von Programmcode-Strukturen aus einem gegebenen Modell.

3 Umsetzung

3.1 Relevante UPPAAL-Funktionalität

UPPAAL wurde für das genannte Projekt aufgrund der Fähigkeiten zur Erstellung zeitbehafteter Modelle gewählt. Dieser Aspekt bleibt für die konzeptionelle Vorstellung in diesem Beitrag zunächst unberücksichtigt, eine spätere Ergänzung ist allerdings sinnvoll. Daher wurde darauf verzichtet, für diesen Beitrag eine einfachere Modellsprache zu verwenden. Zudem eignen sich UPPAAL-Modelle aufgrund der übersichtlichen Darstellung der ausgereiften Werkzeuge zur Dokumentation und Kommunikation von Modelle. Verwendet wird in diesem Ansatz zunächst die Grundfunktionalität von UPPAAL, also die Definition von Automaten mit Zuständen und Transitionen [LPY97, Kap. 4.1] sowie die Verwaltung von Variablen. Letztere werden in UPPAAL für einen Automaten global definiert. Auf die gleiche Art stehen dabei global definierte Funktionen zum Aufruf an den Transitionen zur Verfügung, die komplexere Variablenänderungen bewirken. Das Senden von Signalen über Kanäle bleibt ebenfalls zunächst unberücksichtigt.

Aus der Vernachlässigung zeitbehafteter Modelle ergibt sich implizit die weitere Einschränkung, dass die Aufenthaltsdauer in einem Zustand 0 ist, so dass innerhalb eines Zustandes keine Veränderung von Variablen vorgenommen werden kann. Eine Prüfung der Invarianten eines Zustandes ist in diesem vereinfachten Modell daher hinfällig. Im Model Checking im engeren Sinne, in dem der Zustandsraum durch die Belegung der Variablen aufgespannt wird, ist eine Veränderung der Variablenbelegung innerhalb eines Zustandes aber ohnehin nicht möglich, da jede Änderung einer Transition und damit einem Wechsel des Zustandes entspricht, während ein einzelner Zustand stets nur einem Zeitpunkt des Systems und niemals einer Zeitspanne entspricht ([PGS01, Kap. 4.2]). Als weitere Einschränkung wird aus Gründen der Übersicht zunächst auf die Umsetzung der in UPPAAL definierbaren komplexen Datentypen verzichtet und die Vorgehensweise anhand der vordefinierten primitiven Datentypen für Wahrheitswerte, Ganzzahlen und Fließkommazahlen erläutert.

3.2 Umsetzung und Ausführung im Java-Code

Die entsprechend strukturierten UPPAAL-Modelle sollen im Java-Code so repräsentiert werden, dass der Bezug zum Modell erhalten bleibt. Erforderlich sind dafür deklarative Strukturen, hier in Form von Klassen, Methoden und Java-Annotationen [GJSB05, Kap. 9.7]. Dabei wurde Wert auf die Beibehaltung der Typsicherheit gelegt, so dass die Programmierung ohne Kompromisse für die Angabe der Spezifikation erfolgen kann. Die Strukturen sind im Einzelnen:

- Aus obiger Beschreibung der Struktur von Zustandsautomaten in UPPAAL ergibt sich, dass der Aufruf von Java-Code, der eine Zustandsänderung bewirken soll, ausschließlich über die Formulierung entsprechender Methodenaufrufe für das Schalten von Transitionen erfolgen kann. Es wird davon ausgegangen, dass die Schnittstel-

le zu diesem als Black Box betrachteten Code in einer Klasse gekapselt und eine Instanz davon – bezeichnet als **Actor** – an die Ausführungskomponente übergeben wird. Dadurch wird gleichzeitig vermieden, dass versehentlich Programmaufrufe über die Grenzen verschiedener Automaten hinweg getätigt werden, da jeder Automat einen ihm zugeordneten Actor haben kann und eine saubere Trennung gemäß der Architektur damit erzwungen wird.

- UPPAAL-**Zustände** werden als Klassen abgebildet, die das vorgegebene State-Interface implementieren. State-Klassen müssen über das Interface einen generischen Typ definieren, der dem Typ der Actor-Implementierung entspricht.
- **Transitionen** werden als Methoden innerhalb des States abgebildet, von dem sie ausgehen. Jeder Transitions-Methode wird als Parameter die Actor-Instanz übergeben, so dass im Methodenrumpf Methoden im Actor aufgerufen werden können. Über Annotationen werden die Klasse des Ziel-States sowie Mengen von Vor- und Nachbedingungen angegeben. Letztere können sowohl konstante Werte als auch Wertebereiche für bestimmte Variablen erwarten. Nachbedingungen werden zur Laufzeit nicht zur Steuerung des Ablaufs genutzt, allerdings kann die Ausführungskomponente mit den Angaben die vom Actor gesetzten Variablen nach dem Feuern der Transition validieren.
- Aus UPPAAL wird die Verwaltung der **Variablen** adaptiert: Diese werden nicht von der Ausführungskomponente verwaltet, sondern in einem separaten Variablencontainer, das für den jeweiligen Anwendungsfall spezifisch implementiert wird und die Variablen über get-Methoden zur Verfügung stellt. Für das Setzen der Variablen ist der Actor verantwortlich, da die Wertezuweisung im Programmcode erfolgen muss, der nicht interpretiert werden kann.

In der Ausführungskomponente wird die Abarbeitung des Automaten über eine Methode angestoßen, die die zu verwendenden Instanzen von Actor und Variablencontainer sowie den Startknoten als Parameter übernimmt. Diese wird in Java über folgende Signatur definiert, wobei <A> der generische Typ des Actors und <V> der generische Typ des Variablencontainers ist:

```
void <A, V> start(Class<State<A>> start, A actor, V vars)
```

Die Ausführungskomponente baut aus diesen Informationen einen Zustandsautomat mit Instanzen der State-Klassen auf. Dabei wird die Gültigkeit der Transitionsmethoden im Bezug auf die Parameter (Actor-Typ) validiert. Dann beginnt im Startzustand die Prüfung der Vorbedingungen und das Feuern der Transitionen.

4 Beispiel

Gegeben sei der Zustandsautomat aus Abbildung 1 mit fünf Zuständen. Für die Umsetzung in Java sollen der Actor als Objekt vom Typ MyActor und der Variablencontainer

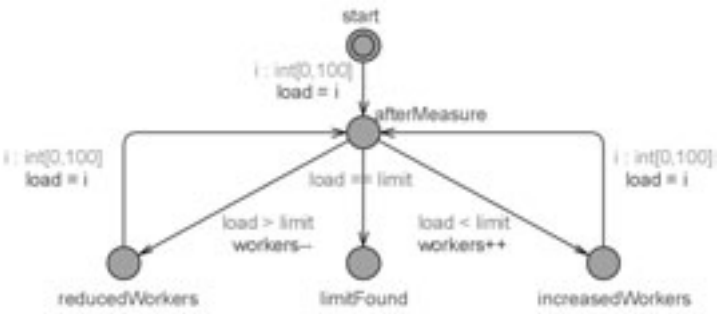

Abbildung 1: Einfacher Zustandsautomat mit fünf Zuständen.

repräsentiert durch ein Objekt vom Typ MyVariables als gegeben angenommen werden. Die States müssen demnach wie in folgendem Codeausschnitt definiert werden:

```
public class StartState implements State<MyActor> {
   @Transition(AfterMeasure.class)
   @Updates(ranges = {
      @RangeUpdate(var="load", min=0, max=100)
   })
   public void startToAfterMeasure(MyActor a) {
      a.doMeasure();
   }
}

public class AfterMeasure implements State<MyActor> {
   @Transition(ReducedWorkers.class)
   @Condition("load > limit")
   @Updates(constants = {
      @ConstantUpdate(var="workers", value="workers-1")
   })
   public void reduceWorkers(MyActor a) {
      a.reduceWorkers();
   }
   // . . .
}
```

Die im Modell eingetragene nicht-deterministische Auswahl einer Ganzzahl im Intervall von 0 bis 100 bildet die Durchführung einer Messung ab, die durch den Actor vollständig gekapselt und deren Ergebnis erst zur Laufzeit bestimmbar ist. Die Spezifikation der gültigen Spanne für Messwerte erlaubt es, das Modell sowohl statisch als auch dynamisch zu überprüfen, ohne die Implementierung der Messung zu kennen.

Der Start der Ausführungskomponente in Java entspricht folgenden Zeilen, wobei die Parameter der Konstruktoren nur beispielhaft sind:

```
MyVariables v = new MyVariables();
MyActor a = new MyActor(v);
StateMachine.start(StartState.class, a, v);
```

5 Fazit und Ausblick

Als Fazit kann festgestellt werden, dass für den gewählten Teilbereich der Funktionalität von UPPAAL alle Modell-relevanten Informationen als Klassenstrukturen und Annotationen im Programmcode abgelegt werden können. Eine Extraktion und Weiterverwendung als Modell ist jederzeit möglich, so dass dieses Ziel vollständig erreicht wurde.

Ein Großteil der Informationen wird auch verwendet, um das Modell auszuführen, so dass die Angaben obligatorisch sind und die Weiterentwicklung des Modells nicht vernachlässigt werden kann. Eine Ausnahme sind die Updates, deren Angabe freiwillig ist und vernachlässigt werden kann, solange das Modell nicht in UPPAAL verifiziert wird. Somit ist dieses Ziel größtenteils, aber nicht vollständig erreicht worden. Fehlerhafte Updates werden beim Simulieren in UPPAAL allerdings erkannt und Inkonsistenzen aufgedeckt. Da die Einstiegspunkte in den anwendungsspezifischen Programmcode bekannt sind, können die fehlenden Informationen durch Analyse dieses im Ansatz als Black Box definierten Codes mit geeigneten Methoden extrahiert werden.

Die gewonnenen Erkenntnisse werden verwendet, um ein Werkzeug zu entwickeln, das in Echtzeit die parallele Entwicklung an Code und Modell unterstützt und dabei trotz unterschiedlicher Repräsentationen die Synchronisierung erhält, ohne automatische Transformationen oder Manipulationen vornehmen zu müssen.

Nach der Umsetzung soll zudem die Möglichkeit der Übertragung des Ansatzes auf weitere Problemfelder, in denen Modelle im Programmcode abgelegt werden können, untersucht werden.

6 Related Work

Auf formaler Ebene kann die Konsistenzsicherung zwischen Modellen durch Triple-Graph-Grammatiken [Kö05] ausgedrückt werden. Wird der Programmcode in Form seines Syntaxbaumes als Graph aufgefasst, definieren die in Abschnitt 3.2 Umsetzungen das Mapping zwischen den beiden Graphen.

Durch die Erhaltung der permanenten Konsistenz zwischen Code und Modell lässt sich der gewählte Ansatz auch gegenüber anderen Ansätzen zu ausführbaren Spezifikationen abgrenzen. Anders als im Konzept der “executable UML” [MB02], bei dem Inkonsistenzen zwischen verschiedenen Repräsentationen durch toolbasierte automatische Transformation oder die Definition einer primären Repräsentation, aus der andere Darstellungeformen – inklusive Programmcode – generiert werden, ausgeglichen werden sollen, soll in diesem Ansatz die Vorrangstellung einer Repräsentation genau vermieden werden.

Anders als in der Java Modeling Language (JML) [LBR99] wird mit dem gewählten Ansatz nicht versucht, eine Spezifikationsnotation für alle denkbaren Systeme zu finden, sondern nur für das ausgewählte Teilproblem der Zustandsautomaten. Dadurch kann für diesen Teilbereich weitgehende Vollständigkeit erreicht werden.

Literatur

[BSGT03] Reinhard Bordewisch, Bärbel Schwärmer, Michael Goedicke und Peter Tröpfner. Lastsimulation für Anwendungsumgebungen in vernetzten IT-Architekturen. *Mitteilungen der GI-Fachgruppe MMB*, (43), 2003.

[GJSB05] James Gosling, Bill Joy, Guy Steele und Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[Kö05] Alexander Königs. Model Transformation with Triple Graph Grammars. In *Model Transformations in Practice Satellite Workshop of MODELS 2005*, Montego Bay, Jamaica, 2005.

[LBR99] Gary T. Leavens, Albert L. Baker und Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe und Ian Simmonds, Hrsg., *BehavioralSpecifications of Businesses and Systems*, Seiten 175–188. Kluwer, 1999.

[LPY97] Kim G. Larsen, Paul Pettersson und Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct 1997.

[MB02] Stephan J. Mellor und Marc J. Balcer. *Executable UML*. Addison-Wesley, 2002.

[PGS01] Doron A. Peled, David Gries und Fred B. Schneider, Hrsg. *Software reliability methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[SK04] Shane Sendall und Jochen Küster. Taming Model Round-Trip Engineering. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development*, 2004.

[TG03] Matthias Tichy und Holger Giese. Seamless UML Support for Service-based Software Architectures. In N Guefi, E Artesiano und G Reggio, Hrsg., *Proceedings of the International Workshop on scientiFic engIneering of Distributed Java applIcations (FIDJI) 2003, Luxembourg*, Jgg. 2952 of *Lecture Notes in Computer Science*, Seiten 128–138. Springer-Verlag, November 2003.