# Automatische, modellbasierte Testdatengenerierung durch Einsatz evolutionärer Verfahren

N. Oster, C. Schieber, F. Saglietti, F. Pinte

Lehrstuhl für Software Engineering
Friedrich-Alexander-Universität Erlangen-Nürnberg
Martensstraße 3
91058 Erlangen
{oster, saglietti, pinte}@informatik.uni-erlangen.de

**Abstract:** Modellbasierte Testfallgenerierungsansätze sind inzwischen zwar weit verbreitet, meist jedoch auf die Erzeugung von Testszenarien beschränkt. Im Allgemeinen ist allerdings darüber hinaus ein nicht unerheblicher manueller Aufwand zur Ermittlung zugehöriger Eingabedaten notwendig. Dieser Artikel präsentiert ein Verfahren, das die vollautomatische Generierung vollständiger Testfallinformation aus Zustandsmaschinen ermöglicht und auf evolutionären Algorithmen sowie Modellsimulation basiert. Erste experimentelle Erfahrungen bei der Anwendung dieses Verfahrens werden berichtet.

## 1 Einführung

Bekanntlich entfällt in vielen Projekten ein wesentlicher Anteil der Entwicklungskosten auf das Testen [1], so dass der Bedarf an Verfahren zur Reduzierung des Testaufwands akut geworden ist. In den letzten Jahren stößt deshalb der Einsatz teilweise automatisierter, modellbasierter Testverfahren auf großes Interesse. Die bisherigen Ansätze hierzu haben allerdings den Nachteil, dass im Allgemeinen nur Testszenarien, nicht aber die zu ihrem Ablauf erforderlichen Eingabedaten automatisch erzeugt werden; die anschließende Ermittlung der auszuführenden Daten beruht allerdings meist noch auf manuellen Schritten, die einen nicht unerheblichen Aufwand verursachen.

Von dieser Lage motiviert wurde im Rahmen der „Softwareoffensive Bayern“ [2] das Projekt UnITeD („Unterstützung inkrementeller Testdaten“) gestartet mit dem Ziel, durch Einsatz evolutionärer Verfahren eine umfassende Testfallgenerierung, also unter Einbezug von Testszenarien und Eingabedaten, aus UML-Modellen vollautomatisch zu gestalten. Dabei sollen die im Rahmen dieses Projekts zu entwickelnden Verfahren sowohl den Komponententest als auch den Integrationstest unterstützen.

Dieser Artikel beschreibt die in UnITeD bereits entwickelten Ansätze (Kapitel 2) und präsentiert erste experimentelle Ergebnisse bei der Anwendung dieses Verfahrens (Kapitel 3). Abschließend wird ein Überblick über den weiteren Verlauf des Vorhabens gegeben (Kapitel 4).

# 2 Testdatengenerierung

## 2.1 Komponententest

Die Verwendung von UML-Modellen und insbesondere von Zustandsmaschinen [3], [4] zur Generierung von Testfällen auf Komponentenebene ist inzwischen weit verbreitet. Die Bestimmung der Testfälle, die zur Überdeckung eines Modells nach einem vorgegebenen Kriterium benötigt werden, wird meist durch statische Analyse erzielt. Dieses Vorgehen stößt allerdings bald an seine Grenzen, spätestens dann, wenn das Modell kontrollflusssteuernde Elemente (z.B. Variablen) enthält, die dynamisch veränderbar sind. In diesen Fällen kann statisch kaum ermittelt werden, welche Alternative einer Verzweigung oder wie oft eine Schleife durchlaufen wird. Aus diesem Grunde wurde ein heuristischer Ansatz [5], der an unserem Lehrstuhl, aufbauend auf einfacheren, bestehenden Methoden [6] entwickelt und bereits erfolgreich zum automatisierten Testen von Programmen eingesetzt wurde [7], von der Codeebene auf die Modellebene übertragen. Der dadurch entstandene modellbasierte Ansatz (Abb. 1) verwendet multikriterielle evolutionäre Verfahren, um aus Zustandsmaschinen Testfälle einschließlich der notwendigen Testdaten so zu generieren, dass eine möglichst hohe Überdeckung durch einen möglichst geringen Testumfang erreicht wird.

Dazu wird zunächst das zu testende Modell in der Notation des Eclipse Modeling Framework (EMF) [8] repräsentiert. Darauf aufbauend wird eine initiale Population von Testfallmengen generiert, die der genetische Algorithmus anschließend miteinander kombiniert, mutiert und auf Grund der jeweils erreichten Modellüberdeckung bewertet [5]. Erreicht eine der generierten Testfallmengen ein vorgegebenes Überdeckungsmaß oder wird ein vorgegebenes Abbruchkriterium (z.B. eine vorgegebene Anzahl von Iterationen) erreicht, wird der genetische Algorithmus beendet, andernfalls iterativ fortgesetzt.

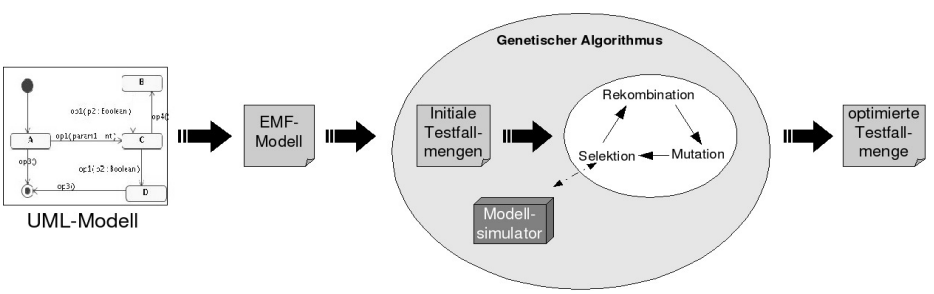


Abbildung 1: Vorgehensweise zur Testfallgenerierung in UnITeD

**Generierung der initialen Population:** Die initiale Population entsteht, indem zufällige Sequenzen der modellierten Operationen und Signale zusammen mit zufälligen Testdaten generiert werden, sodass jede der entstehenden Sequenzen in der zugrunde liegenden Zustandsmaschine auftreten kann. Die Testdaten und eventuell vorhandene Guards werden dabei zunächst nicht berücksichtigt. Jede Sequenz stellt einen Testfall dar und wird im genetischen Algorithmus durch ein Gen repräsentiert. Eine Menge von Genen bildet ein Individuum, eine Menge von Individuen schließlich eine Population, auf welcher der genetische Algorithmus arbeitet.

**Mutationsoperatoren:** Ein evolutionäres Verfahren umfasst drei wesentliche Schritte: Selektion, Rekombination und Mutation. Bei der Verwendung genetischer Algorithmen zur Testfallgenerierung und -optimierung ergeben sich für den Mutationsoperator drei Strategien: die Mutation von Testdaten, die Mutation eines Testfalls durch Hinzufügen bzw. Löschen einer Operation oder eines Signals und die Mutation einer Testfallmenge durch Hinzufügen bzw. Löschen eines Testfalls.

**Bestimmung der Modellüberdeckung:** Die von einer Testfallmenge T erreichte Überdeckung Ü(T) wird durch Ü(T) = #C(M,T) / #C(M) berechnet, wobei #C(M) die Anzahl der gemäß einem vorgegebenen Kriterium C im Modell M zu überdeckenden Elemente (Zustände, Transitionen, u.a. [9]) und #C(M,T) die Anzahl der durch T tatsächlich überdeckten Elemente bezeichnen. Letztere werden mit Hilfe eines Modellsimulators ermittelt; hierzu werden die Testfälle einzeln simuliert und die jeweils überdeckten Modellelemente vom Simulator ermittelt. Die von der jeweiligen Testfallmenge T erzielte Überdeckung Ü(T), sowie deren Umfang #T werden vom genetischen Algorithmus zur Bestimmung der Fitness des aktuellen Kandidaten T verwendet.

## 2.2 Integrationstest

Über den Komponententest hinaus wird im vorgestellten Projekt auch ein entsprechendes Verfahren zur Automatisierung des Integrationstests eingesetzt. Hierfür wird das Verhalten einzelner Komponenten mittels individueller Zustandsmaschinen derart modelliert, dass diese über die komponentenspezifische Funktionalität hinaus auch die möglichen Interaktionen zwischen den modellierten Komponenten wiedergeben.

Zu diesem Zweck wurde bereits eine Reihe spezieller Schnittstellenüberdeckungen definiert und hierarchisiert [9]. Um im Einzelfall eine rationale Entscheidung bezüglich einer angemessenen Integrationsteststrategie zu erleichtern, wurde darüber hinaus eine konservative Schätzung der Anzahl jeweils erforderlicher Testfälle ermittelt.

## 3 Experimentelle Ergebnisse für den Komponententest

In diesem Abschnitt werden erste experimentelle Resultate einer auf Java basierenden Umsetzung des vorgestellten Verfahrens präsentiert, die aus Zustandsmaschinen Testszenarien sowie die dazu benötigten Eingabedaten generiert. Die aktuelle Umsetzung ist auf Zustandsmaschinen beschränkt, die keine Nebenläufigkeit enthalten.

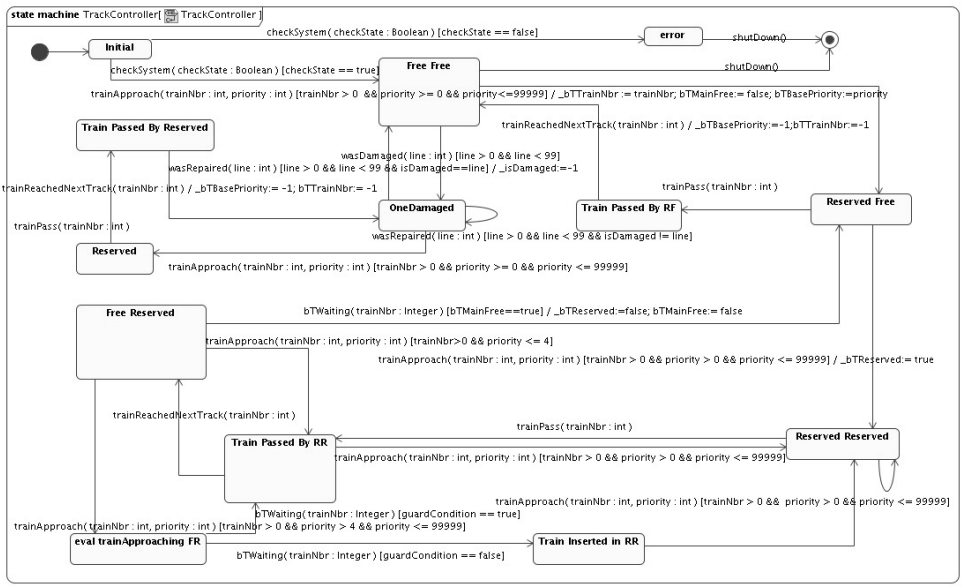


Abbildung 2: Beispiel der Zustandsmaschine TrackController (TC)

Drei Zustandsmaschinen wurden zur Erhebung der in Tabelle 1 präsentierten Ergebnisse herangezogen. Allen Messungen liegt dabei als Ziel die Transitionsüberdeckung, also das Durchlaufen aller Kanten einer Zustandsmaschine zugrunde.

Die erste Zeile aus Tabelle 1 beschreibt die Ergebnisse für die Zustandsmaschine TrackController (TC), die in Abb. 2 wiedergegeben ist und aus 9 Zuständen und 25 Transitionen besteht. Für diese Zustandsmaschine erreichte das vorgestellte Verfahren bereits nach 20 Generationen eine 100%-ige Überdeckung. Die Anwendung des genetischen Verfahrens wurde dennoch über diese Zielmarke hinaus fortgesetzt und erst nach 150 Generationen bzw. 1016 Sekunden beendet, um die Anzahl der erforderlichen Testfälle weiter zu reduzieren. Es wurden schließlich 7 Testfälle generiert, also ein Testfall weniger als im Falle statisch zu ermittelnder Pfade.

| | #Zust | #Trans | StatPfad | Cov | NwdGen | #Gen | #TF | DauerTFG |
|---|---|---|---|---|---|---|---|---|
| TC | 9 | 25 | 8 | 100% | 20 | 150 | 7 | 1016 |
| $Z_{konto}$ | 8 | 19 | 9 | 100% | 110 | 150 | 5 | 482 |
| $Z_{call}$ | 9 | 17 | 8 | 94,1% | -- | 200 | 3 | 353 |

#Zust : Anzahl der enthaltenen Zustände
#Trans : Anzahl der enthaltenen Transitionen
StatPfad : Anzahl statisch ermittelter zu überdeckender Pfade
Cov : erreichte Überdeckung
NwdGen : Anzahl notwendiger Generationen für 100% Überdeckung
#Gen : Anzahl der durchgeführten Iterationen
#TF : Anzahl der generierten Testfälle
DauerTFG : Dauer der Testfallgenerierung in Sekunden

Tabelle 1: Ergebnisse der Anwendung des vorgestellten Verfahrens

Bei der Zustandsmaschine $Z_{konto}$ konnte durch den genetischen Ansatz die Anzahl der Testfälle beinahe halbiert werden. Dieses Unterbieten lässt sich dadurch erklären, dass die statische Analyse sich auf die Betrachtung nur relativ einfacher Pfade beschränkt, während der genetische Algorithmus den Testumfang durch Erzeugung relativ komplexer Abläufe, die jeweils eine höhere Anzahl an Modellelementen überdecken, zu minimieren bestrebt ist.

Die weiteren Ergebnisse aus Tabelle 1 zeigen, dass das vorgestellte Verfahren bei zwei der drei Zustandsmaschinen eine hundertprozentige Überdeckung erreichen konnte (während eine Transition von $Z_{call}$ im Rahmen der verfügbaren Zeit nicht überdeckt werden konnte). Gleichzeitig konnte die Anzahl der generierten Testfälle gegenüber der statisch Ermittelten stets unterschritten werden.

Weiterhin erkennt man in Tabelle 1, dass die Anzahl der Generationen, die zur Erreichung einer 100%-igen Überdeckung notwendig sind, stark variiert. Um eine grobe Aufwandsschätzung zu ermöglichen, enthält die letzte Spalte die Dauer der Testfallgenerierung für das jeweilige Beispiel. Diese kann aufgrund verschiedener Faktoren (z.B. Rechenleistung, Systemauslastung, Komplexität des Modells) stark schwanken und soll daher nur als Anhaltspunkt dienen.

Um eine umfangreichere Evaluierung zu ermöglichen, ist eine Erprobung dieser Technik in einem realen medizintechnischen Umfeld geplant.

## 4 Zusammenfassung und Ausblick

In diesem Artikel wurde ein allgemeines Verfahren zur Automatisierung strukturellen, modellbasierten Testens präsentiert, das sich sowohl für den Komponententest als auch für den Integrationstest eignet. Der vorgestellte Ansatz verwendet genetische Algorithmen und Modellsimulation, um die Generierung der Testszenarien und der erforderlichen Testdaten vollständig zu automatisieren. Darüber hinaus wurden erste Ergebnisse vorgestellt, die bei der Anwendung des Verfahrens auf den Komponententest gewonnen wurden.

Die derzeitigen Forschungsarbeiten beschäftigen sich mit der Übertragung des vorgestellten Ansatzes auf andere Verhaltensdiagramme (z.B. Aktivitäten) der UML, damit auch für diese Modelle Testdaten automatisch generiert werden können. Außerdem liegt ein weiterer Forschungsschwerpunkt auf der Bewertung des Fehleraufdeckungspotentials der generierten Testfallmengen. Dabei soll insbesondere untersucht werden, welcher Einfluss von Art, Überdeckungsgrad und Abstraktionsniveau des zugrundeliegenden Modells auf das Fehleraufdeckungspotential entsprechend generierter Testläufe zu erwarten ist.

## Literaturverzeichnis


[1] S. R. Schach. Object-oriented and Classical Software Engineering. McGraw-Hill, 2005.

[2] Software-Offensive Bayern, http://www.software-offensive-bayern.de/

[3] J. Offutt, S. Liu, A. Abdurazik and P. Ammann. Generating test data from state-based specifications. Software Testing, Verification and Reliability, 13(1), 2003.

[4] L. C. Briand, J. Cui and Y. Labiche. Towards Automated Support for Deriving Test Data from UML Statecharts. Lecture Notes in Computer Science, vol. 2863, Springer 2003.

[5] N. Oster. Automatische Generierung optimaler struktureller Testdaten für objekt-orientierte Software mittels multi-objektiver Metaheuristiken. Dissertation, University Erlangen-Nuremberg 2007

[6] Phil McMinn. Search-based software test data generation: A survey. Software Testing, Verification and Reliability, 14(2), 2004

[7] N. Oster and F. Saglietti. Automatic Test Data Generation by Multi-Objective Optimisation. Lecture Notes in Computer Science, vol. 4166, Springer 2006

[8] Eclipse Homepage, http://www.eclipse.org/modeling/mdt/

[9] F. Saglietti, N. Oster and F. Pinte. Interface Coverage Criteria Supporting Model-Based Integration Testing. In M. Platzner et al (Eds.): Workshop proceedings of the 20th International Conference on Architecture of Computing Systems (ARCS 2007), VDE 2007