Integration von Model-Driven Development und formaler Verifikation in den Softwareentwicklungsprozess - eine Fallstudie mit einem 3D-Tracking-System

Christian Ammann
Fachhochschule Osnabrück
Postfach 1940, 49009 *Osnabrück*
c.ammann@fh-osnabrueck.de

Zusammenfassung

Bei modellgetriebener Softwareentwicklung werden Modelle entwickelt und aus diesen ausführbare Software generiert. Durch die Verknüpfung mit formaler Verifikation können Fehler in den Modellen gefunden und so der Ansatz der modellgetriebenen Softwareentwicklung verbessert werden. Diese Arbeit untersucht anhand von zwei Fallstudien, wie aktuelle Forschungsergebnisse im Bereich der Modellierung und Verifikation auf Verhaltensebene in den Softwareentwicklungsprozess integriert werden können.

1 Einleitung

Die modellgetriebene Softwareentwicklung (MDD)[1] ist eine Methode, um die Qualität von Software zu erhöhen und die Anzahl von Fehlern zu reduzieren. Der MDD-Ansatz besagt, dass zumindest für Teile eines Systems Modelle entwickelt und diese automatisiert in lauffähige Software übersetzt werden.

Die erzeugten Modelle können Fehler enthalten. Deshalb ist die Verwendung eines *Model Checkers*[2] zur Prüfung von Modelleigenschaften sinnvoll. MDD wird daher so erweitert, dass neben den Übersetzungsschablonen, die ein Modell in ausführbare Software übersetzen, auch Schablonen entwickelt werden, die das Modell in eine *Model Checker Eingabesprache* transformieren. Durch die Überprüfung mit einem *Model Checker* können Fehler im Modell entdeckt werden, die sich auch entsprechend auf die generierte Software auswirken würden. Die Integration von *Model Checking* in die modellgetriebene Softwareentwicklung ist deshalb ein wichtiger Beitrag zur Erhöhung der Korrektheit und somit der Qualität von Software.

Das Gesamtziel dieser Arbeit ist es zu untersuchen, wie die bestehenden Forschungsergebnisse bezüglich der Kombination von *Model Checking* und MDD in den klassischen SW-Engineering-Prozess integriert werden können. Dazu wird zunächst in Abschnitt 2 der Stand der aktuellen Forschung betrachtet und daraus vier mögliche Lösungsansätze abgeleitet, um MDD und *Model Checking* zu kombinieren und in die Praxis zu integrieren. Diese werden in einer ersten Fallstudie in Abschnitt 3 auf ihre Vor- und Nachteile hin untersucht. Das Ergebnis wird in Abschnitt 4 genutzt, um eine Erweiterung für ein 3D-Tracking-System umzusetzen. Abschließend wird das Ergebnis der Arbeit in Abschnitt 5 diskutiert und ein Ausblick auf zukünftige Forschungsvorhaben gegeben.

Diese Arbeit ist Teil des Forschungsprojektes *KoverJa*[3], das vom *Bundesministerium für Bildung und Forschung* gefördert wird.

2 Ideen und Stand der Technik

Die Verknüpfung von MDD und *Model Checking* ist bereits Gegenstand aktueller Forschung. Jones et al.[4] beschreiben beispielsweise *Mobile Health Systems* (MHS). Dabei handelt es sich um mobile Geräte zum Messen von menschlichen Körperfunktionen wie z.B. Blutdruck. MHS werden direkt am Körper des Patienten getragen und sind untereinander, mit einem Arzt etc. vernetzt. Die Autoren model-

lieren MHS mit Klassendiagrammen und schlagen SPIN[5] zur Verifikation vor. Eine weitere Möglichkeit, um Modelle zu erzeugen und zu übersetzen, stellen *Domain Specific Languages* (DSL) dar. In [6] werden Klassendiagramme als Graphentransformationssysteme mit einer DSL modelliert und von dem *Model Checker Alloy* verifiziert.

Es zeigt sich, dass der Schwerpunkt der Untersuchungen in der Literatur auf der strukturellen Ebene von Softwaresystemen liegt, also den Beziehungen von Klassen und ihrer Kommunikation untereinander. Deshalb wurde bereits im Rahmen von *KoverJa* ein ähnlicher Ansatz gewählt, um strukturelle Aspekte des in 4 präsentierten 3D-Tracking-Systems zu modellieren, nach *Java* und *Promela* zu übersetzen und anschließend zu verifizieren. Dabei werden mittels *open ArchitectureWare* eine DSL und entsprechende Übersetzungsschablonen entwickelt. Es zeigt sich:

- Die durch den *Model Checker* gefundenen Fehler können nicht durch Anpassen des Modells behoben werden, da sie sich auf Verhaltens- und nicht auf struktureller Ebene befinden (beispielsweise Zugriffe auf ungültige Speicherbereiche). Stattdessen müssen die Übersetzungsschablonen modifiziert werden.
- Die entwickelte DSL ist spezialisiert auf die 3D-Tracking-Software und nicht wiederverwendbar zur Beschreibung anderer Softwaresysteme.

Diese Probleme machen einen Einsatz von MDD und *Model Checking* in der Praxis schwierig. Anstelle von Klassendiagrammen, die nur strukturelle Aspekte abbilden oder einer DSL, die nur zum Modellieren eines speziellen Softwaresystems verwendet werden kann, kommen deshalb in dieser Arbeit UML-Statecharts zum Einsatz. Diese können neben strukturellen Aspekten auch auf Verhaltensebene modellieren und haben den Vorteil einer weiten Verbreitung. Ferner wurde im Bereich der Verifikation von UML-Statecharts bereits erfolgreich Grundlagenforschung betrieben.

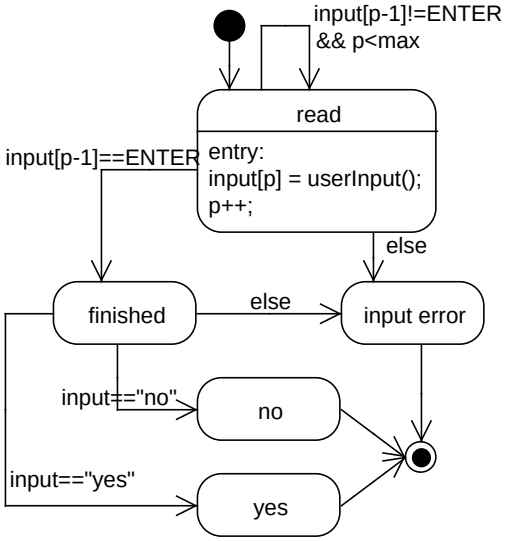

Abbildung 1: Beispielanwendung zum Einlesen einer Benutzereingabe

3 Fallstudie 1: Benutzereingabe

Abbildung 1 zeigt einen deterministischen Automaten als UML-Statechart, der die Tastatureingaben eines Benutzers entgegen nimmt (über die Funktion *userInput()*), diese in einem *char* Array namens *input* speichert und nach Drücken von *return* überprüft, ob der Benutzer *yes* oder *no* eingegeben hat. Die Variable p ist ein Zähler vom Typ *byte*, *max* eine Konstante mit dem Wert 4.

Dieses Beispiel wird untersucht, da der verwendete UML-Statecharts-Funktionsumfang ausreichend ist, um damit im nächsten Abschnitt die 3D-Tracking-Software umzusetzen. Dabei fällt auf, dass UML-Statechart-Elemente wie Nebenläufigkeit und *Trigger-Events* nicht gebraucht werden. Es ist nur eine Untermenge von UML-Statecharts relevant, die eine Möglichkeit bietet, Benutzereingaben entgegen zu nehmen und Transitionsübergänge von *Guards* abhängig macht.

Im Rahmen dieser Arbeit werden vier ver-

schiedene Ansätze zur Modellierung und Verifikation des Beispiels in Abbildung 1 auf Laufzeit und Speicherverbrauch hin untersucht und vorgestellt. Ziel ist es beim Verifizieren den kompletten Zustandsraum abzusuchen, um Fehler, wie zum Beispiel Zugriffe auf Elemente eines Arrays, die außerhalb seines Defnitionsbereiches liegen (*ArrayOutOfBoundsException*), zu finden.

Bei der ersten Messung (JPF+SCE) wird der Java Pathfinder (JPF) verwendet[7]. JPF ist ein Tool, das *Java* Bytecode verifizieren kann und besitzt eine Erweiterung zum Untersuchen von in Java implementierten UML-Statecharts (SCE). Der Vorteil ist, dass die *Java* UML-Statechart-Modelle gleichzeitig Software und *Model Checker Eingabesprache* sind. Somit ist nur eine Modelltransformation von UML-Statecharts nach *Java* notwendig. SCE unterstützt die volle Funktionalität von UML-Statecharts, was sich negativ auf die Verifikation auswirken kann. Mit der zweiten Messung (JPF+SCE0) wird deshalb eine eigene Implementierung der SCE vorgenommen, die nur die notwendige Untermenge der UML-Statecharts umsetzt.

Von Merz et al.[8] wird ein Tool zur automatischen Codegenerierung und Verifikation von UML-Statecharts namens *Hugo* entwickelt. Bei der dritten Messung (Hugo+SPIN) wird das Modell von *Hugo* nach *Promela* übersetzt und von SPIN verifiziert. Auch *Hugo* hat den Nachteil, dass alle Eigenschaften von UML-Statecharts unterstützt werden (Nebenläufigkeit, etc) und dadurch entsprechende Artefakte im *Promela* Code vorhanden sind, die sich negativ auf die Größe des Zustandsvektors auswirken. Die vierte Messung (oAW+SPIN) benutzt zum Modellieren des UML-Statecharts eine mit oAW erzeugte DSL und übersetzt diese nach *Promela*. Dabei wird nur die notwendige UML-Statechart-Untermenge verwendet. Die folgende Tabelle zeigt die Ergebnisse der vier Messungen:

Messung	RAM-Verbrauch	Laufzeit
JPF+SCE	55MB	707s
JPF+SCE0	8MB	155s
Hugo+SPIN	443MB	10.3s
oAW+SPIN	62MB	2.27s

Da die Kombination von oAW und SPIN einen Kompromiss aus Speicherverbrauch und Laufzeit darstellt, wird sie zur Umsetzung der zweiten Fallstudie verwendet.

4 Fallstudie 2: 3D-Tracking

Mit den gewonnen Erkenntnissen aus dem letzten Abschnitt wird eine Erweiterung für *assyControl* implementiert. *assyControl* ist ein 3D-Tracking-System, das von der *soft2tec GmbH* und der *Otto Kind AG* entwickelt wird. Es überwacht die Arbeitsschritte in der industriellen Produktion und warnt Angestellte, wenn Fehler beim Zusammenbau von Komponenten gemacht werden.

Die Erweiterung für *assyControl* soll eine einfachere Möglichkeit bieten, das System neu zu konfigurieren, wenn sich die Positionen von Objekten an einem Arbeitsplatz verändert haben (ein Angestellter ist beispielsweise Linkshänder und passt den Arbeitsplatz zu Beginn seiner Schicht entsprechend an). Nach dem Umbau des Arbeitsplatzes soll das System während eines ersten Durchlaufs automatisch die neuen Positionen von Objekten auf der Werkbank anhand der Bewegungen des Angestellten erkennen.

Die Erweiterung von *assyControl* wird als UML-Statechart mit einer DSL modelliert und nach *Java* sowie *Promela* transformiert. Bei der Verifikation kann zunächst aufgrund der Komplexität nicht der gesamte Zustandsraum durchsucht werden (M1). SPIN behandelt jedes *Promela Statement* als eigenen Zustand. Mittels der *d_step*{...} Instruktion können mehrere Statements umschlossen und zu einem Zustand zusammen gefasst werden. Deshalb werden manuell *d_step* Instruktion eingefügt und die Anzahl der Zustände dadurch reduziert (M2). Auch Hilfsvariablen in *Promela* sind Teil des Zustandsvektors. In einem zweiten Schritt

werden deshalb die Hilfsvariablen nach dem Verlassen eines *d_step* Bereiches zurückgesetzt und so die Größe des Zustandsraumes verringert (M3). Die Messungen der Verifikationsläufe hinsichtlich Größe des Zustandsraums und erreichter Zustände bringen die folgenden Ergebnisse:

Name	Zustandsraum	erreichte Zustände
M1*	17.000.000	1.217.791
M2*	17.000.000	2.884.019
M3	6.798.196	2.971.144

Bei M1 und M2 wird die Verifikation vorzeitig beendet, da der maximal zur Verfügung stehende Hauptspeicher (1.5GB) verbraucht ist. Erst mit M3 kann die *assyControl* Erweiterung vollständig verifiziert werden.

5 Zusammenfassung

Obwohl es eine breite Basis von Forschungsergebnissen gibt, ist es schwierig *Model Checking* und MDD in die Praxis zu integrieren. Bestehende Lösungen betrachten oft strukturelle Aspekte eines Softwaresystems und sind nicht generisch einsetzbar. Bei den Ansätzen zum Modellieren und Verifizieren auf Verhaltensebene wie UML-Statecharts wird deren gesamter Funktionsumfang berücksichtigt, was sich negativ auf Laufzeit und Speicherverbrauch der formalen Verifikation auswirkt. Für zukünftige Arbeiten wäre es daher sinnvoll ein System zu entwickeln, das flexibel nur die notwendige Teilmenge von UML-Statecharts berücksichtigt bzw. in dem sich nicht gebrauchte UML-Statechart-Eigenschaften deaktivieren lassen. Ferner ist es wünschenswert den erzeugten *Promela Code* automatisch zu optimieren (beispielsweise durch Hinzufügen von *d_step* Instruktionen oder Zurücksetzen von Hilfsvariablen), so dass ein Anwender keine tieferen Kenntnisse über *Model Checking* haben muss.

Literatur

[1] T. Stahl, M. Völter, S. Efftinge, and A. Haase. *Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management*. dpunkt.verlag, zweite auflage edition, 2007. in german.

[2] S. Kleuker. *Formale Modelle der Softwareentwicklung*. Vieweg+Teubner Verlag / GWV Fachverlage GmbH, Wiesbaden, 2009. in german.

[3] KoverJa - Korrekte verteilte Java Applikationen. `http://www.edvsz.fh-osnabrueck.de/kleuker/CSI/KoverJa`; accessed 25-February-2010; in german.

[4] V.M. Jones. Model driven development of m-health systems (with a touch of formality). In *Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference*, pages 5 pp.–584, March 2006.

[5] G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.

[6] Z. Demirezen, M. Mernik, J. Gray, and B. Bryant. Verification of DSMLs using graph transformation: a case study with Alloy. In *MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 1–10, New York, NY, USA, 2009. ACM.

[7] T. Gvero, M. Gligoric, S. Lauterburg, M. d'Amorim, D. Marinov, and S. Khurshid. State extensions for java pathfinder. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 863–866, New York, NY, USA, 2008. ACM.

[8] S. Merz. Model checking and code generation for uml state machines and collaborations. In *In G. Schellhorn and W. Reif. 5 th Workshop on Tools for System Design and Verification (FM-TOOLS*, pages 59–64, 2002.