Dr. Horst Kargl

(E-Mail: horst.kargl@sparxsystems.at)
beschäftigt sich seit 2000 mit objektorientierter Modellierung. Bevor er 2008 zu SparxSystems wechselte, war er wissenschaftlicher Mitarbeiter an der TU Wien und forschte in mehreren Projekten zu den Themen e-Learning, Semantic Web sowie modellgetriebene Software-Entwicklung. Seine momentanen Aufgaben sind Schulungen im Bereich Modellierungstechnik mit Enterprise Architect sowie dessen Erweiterung durch Add-Ins. Des Weiteren ist er beim Forschungsprojekt AMOR (Semantische Versionierung von Modellen) Vertreter des Unternehmens SparxSystems Central Europe und leitet die Integration der Forschungsergebnisse.

Flexibilität beim modellbasierten Testen

Zeit ist Geld. Vor allem bei der Entwicklung von komplexen Systemen steigt der Bedarf an Zeit und steigen somit auch die Finanzmittel, die man benötigt, um Fehler zu einem späten Zeitpunkt im Entwicklungsprozess zu beheben. Durch detailliertes Modellieren können Fehlentscheidungen in der Architektur schon frühzeitig erkannt und behoben werden. Die Dynamik von komplexen Systemen kann zwar modelliert werden, doch Modelle zum Anschauen auf Papier sind geduldig. Erstellte Modelle können Fehler enthalten, besonders kritisch und kostenintensiv sind dabei logische Fehler im konzeptionellen Modell.
Um dem entgegenzuwirken und möglichst früh möglichst viele Fehler zu erkennen, können Modelle selber ausgeführt und dabei getestet werden.

Motivation

„Der Mensch ist nicht dafür geschaffen, komplexe Systeme zu bauen“, schreiben Harry M. Sneed et. al. im Buch „Der Systemtest“ [Sne09]. Der Mensch ist fehlbar und so auch alles, was er erschafft. Seine Aufgabe ist es daher, diese Fehlerquote so gering wie möglich zu halten. Fehler in Softwaresystemen sind oft schwer aufzufinden und können zu kritischen Problemen führen. Zum Beispiel führten falsche Einheiten zum Scheitern der Marsmission Mars Climate Orbit 1998, der Satellit verglühte in der Marsatmosphäre und viel Geld mit ihm.

Das Entwickeln von komplexen Systemen setzt eine detaillierte und geprüfte Planung voraus. Die Komponenten dieser Systeme sind meist genormte Bausteine mit definiertem Verhalten, welche zu einem größeren System zusammengesetzt werden sollen. Ein Plan hilft konzeptionelle Abhängigkeiten und die Funktionsweise des Gesamtsystems in den Griff zu bekommen. Wenn dieses System ein Kraftwerk oder auch nur ein Haus ist, werden die Pläne so detailliert gestaltet, damit das zu erstellende System ohne wesentliche weitere Fragen gebaut werden kann. Im Fall von kritischen Systemen wird oft so detailliert geplant, dass die Funktionsweise vorab als Modell simuliert und damit getestet werden kann.

Wenn es sich bei dem zu erstellenden System um Software handelt, sind die Freiheitsgrade bezüglich standardisierter Komponenten und deren Zusammenspiel oft weit größer, als dies in anderen Disziplinen der Fall ist. Deshalb sollte vor jeder Realisierung eines Software-Systems ebenfalls eine Planung erfolgen. Da jedoch die meisten konzeptionellen Modelle für Software nur bis zu einem gewissen Detaillierungsgrad erstellt werden, ist eine genaue Überprüfung nur schwer und eine Simulation von abstrakten Modellen nur selten möglich. Dieser Artikel beschreibt einen Ansatz, der es möglich macht, konzeptionelle Softwaremodelle auf verschiedenen Granularitätsniveaus zu erstellen und zu simulieren und somit das dynamische Verhalten des zu erstellenden Systems evaluieren und testen zu können.

Testen von Software

Bei Softwaresystemen rechnet man mit einer durchschnittlichen Fehleranzahl von 3 Fehlern auf 1000 Anweisungen [Sne09]. In sicherheitskritischer Software kommen schlussendlich wesentlich weniger Fehler vor, weil sie umfangreicher getestet wird. Dabei wird in der Regel zwischen White-Box-Tests (Komponenten-/Unit-Test), Gray-Box-Tests (Integrationstest) und Black-Box-Tests (System- und Abnahmetest) unterschieden [Sne09].

Die Ausgangsbasis für funktionale Tests sind Anforderungen an das System. Aus den meist natürlich sprachlich vorliegenden Anforderungen werden Testfälle abgeleitet. Für diese Testfälle sind die nötigen Testdaten herzuleiten, welche als Input für die einzelnen Tests dienen. Die Ableitung der textuellen Spezifikation der Testfälle kann bei Einhaltung sprachlicher Regeln durch Analyse der textuellen Anforderungen automatisiert werden [Sne09]. Von textuellen Anforderungen zum Code ist es allerdings ein großer Sprung. Konzeptionelle Modelle schaffen Abhilfe.

Modellbasiertes Testen

Bevor eine Zeile Code geschrieben wird, sollte ein konzeptionelles Modell des Systems vorhanden sein. In manchen Industriebereichen ist die Dokumentation der erstellten Software durch solche Modelle sogar die Voraussetzung für eine Abnahme durch den Kunden (z. B. in der Automobilindustrie durch Automotive SPICE). Beim modellbasierten Testen [Hau03] geht man von einem konzeptionellen Modell (z.B. UML) aus und leitet davon die Testfälle ab. Wenn dieser Vorgang automatisch durchgeführt wird, spricht man auch von

modellgetriebenem Testen (Model Driven Testing, MDT). Dafür werden dieselben Technologien eingesetzt, wie sie auch beim modellgetriebenen Softwareentwickeln eingesetzt werden (Model Driven Engineering oder auch Model Driven Development). Durch diesen Ansatz können aus formal spezifizierten, meist grafischen Modellen, weit mehr testrelevante Informationen aus dem Modell extrahiert werden. Im Fall von MDT können z. B. benötigte Eingabedaten und erwartete Zieldaten automatisch aus dem Modell abgeleitet werden [Hau03]. Das Hauptziel von MDT ist es nun, aus dem konzeptionellen Modell alle nötigen Informationen für das Testdesign und dessen Ausführung abzuleiten [Bla04].

Ein weiterer modellbasierter Testansatz verwendet Modelle nicht, um daraus Testdaten und Testfälle abzuleiten, sondern zur Simulation des modellierten Systems. Hierfür muss noch keine Zeile Code geschrieben worden sein, um das konzeptionelle Modell des Systems bereits testen zu können. Dieser Ansatz kommt der Tatsache entgegen, dass frühzeitig gefundene Fehler im Entwicklungsprozess wesentlich günstiger zu korrigieren sind als später im Projektverlauf. Es ist allerdings zu bedenken, dass sich die Simulation auf das konzeptionelle Modell und auf die Realisierung der Simulation beziehen. Im entwickelten System könnten andere Rahmenbedingungen herrschen als bei der Simulation. Dieses Problem gibt es aber beispielsweise auch bei reinen Source-Code-Tests: Lokal durchgeführte Unit-Tests können fehlerfrei laufen, Fehler können aber dennoch im Endsystem auftreten, da das Produktivsystem möglicherweise eine andere Rechnerarchitektur besitzt.

Verschiedene Domänen nutzen für die konzeptionelle Modellierung unterschiedliche Tools und Modellierungssprachen. Im Bereich Embedded Systems wird sehr gerne MATLAB/Simulink, Statemate und LabVIEW eingesetzt. Für softwarelastige Projekte wird als Sprache oft UML mit diversen Modellierungstools verwendet.
Dieser Artikel beschäftigt sich im Weiteren mit dem modellbasierten Testen mittels Simulation von UML mit dem Modellierungswerkzeuges Enterprise Architect [SPX].

Testen durch Modellsimulation

Wenn ein System durch ein konzeptionelles Modell vollständig beschrieben ist, kann die Funktionalität durch Überprüfung dieses Modells sichergestellt werden. Dabei kann zwischen statischen und dynamischen Tests unterschieden werden. Das zu untersuchende Modell wird als *Model Under Test* (MUT) bezeichnet.

Statische Tests überprüfen die syntaktische Korrektheit des Modells sowie gegebenenfalls zuvor definierte Modellierungsrichtlinien. Modellierungsrichtlinien sind wie Programmierrichtlinien zu verstehen und beschreiben, welche Modellelemente in welchem Kontext verwendet werden dürfen, um ein unternehmensweit einheitliches Bild der Modelle zu gewährleisten. Des Weiteren können auch logische Fehler im Modell bereits durch statische Überprüfung gefunden werden, wenn z. B. für Verzweigungen eines Prozesses keine bzw. keine eindeutigen Bedingungen angegeben wurden. Diese Fehler können natürlich auch beim dynamischen Test durch eine Simulation erkannt werden. Das Auffinden von syntaktischen Fehlern ist von der Modellierungssprache abhängig. Im Enterprise Architekt ist eine Menge von Syntaxregeln schon vordefiniert, kann aber beliebig erweitert werden. Die Überprüfung von Modellierungsrichtlinien ist unternehmens- bzw. sogar projektspezifisch und wird ebenfalls selbst definiert. Die Definition diverser Regeln basiert auf dem EA-Objektmodell und kann in diversen Programmier- bzw. Script-Sprachen durchgeführt werden; auch OCL Regeln können beim Validieren berücksichtigt werden.

Dynamische Tests überprüfen das Laufzeitverhalten des Systems und zeigen Engpässe und ungünstige Konfigurationen auf. Je nach verwendeter Modellierungssprache stehen verschiedene Modelle zur Beschreibung von Verhalten zur Verfügung. Die UML bietet z. B. Aktivitäts-, Zustands- und Sequenzdiagramme. In SysML, (System Modelling Language), welche eingesetzt wird, um hardwarenahe Systeme zu modellieren, gibt es zusätzlich das Zusicherungsdiagramm (*Pragmatic Diagram*). Das Zusicherungsdiagramm beschreibt den Zusammenhang von Einschränkungs-Blöcken (constraint blocks), wobei jeder constraint block eine Funktion beschreibt. Zur Simulation des SysML-Zusicherungsdiagrammes bietet SparxSystems eine Lösung für den EA ab der Version 7.5 an. Für die Simulation werden Input-Daten für Funktionen in constraint blocks definiert. Das Ergebnis ist ein Funktionsplot. Für die Simulation von Aktivitätsdiagrammen (AD) und Zustandsautomaten (StM) hat die Firma Lieber-Lieber, Partner von Sparxsystems Central Europe, eine Lösung als Plug-In implementiert: die Model Simulation Engine (MSE). Im nächsten Abschnitt wird beschrieben, wie Enterprise Architect in Kombination mit der MSE benützt werden kann.

Simulationsansätze: Wenn Modelle Input-Daten benötigen und Output-Daten während der Simulation produzieren, muss für deren Ausführung ein Set an Input-Daten zur Verfügung gestellt werden. Dies kann während der Simulation manuell und interaktiv geschehen oder automatisch durch die Bereitstellung von Testdaten. Diese Testdaten müssen den Anforderungen und Einschränkungen des Modells entsprechen. Zudem können für ein gewisses Set an Input-Daten, erwartete Output-Daten definiert werden. Wenn nun die Simulation des Modells mit den Input-Daten gestartet wird, sollte die Simulation nach endlicher Zeit terminieren und die zuvor definierten „erwarteten“ Output-Daten generiert haben. Durch Vergleich von Input- und Output-Daten können *Unit-Tests* für das Modell erstellt werden. Handelt es sich beim Modell um ein „kontinuierliches Modell“ (das Modell hat keinen definierten Endzustand und läuft kontinuierlich weiter), können ebenfalls Input- und Output-Daten definiert werden. Der *Model-Unit-Test* wird allerdings nicht nach Terminierung des Modells angewendet, sondern bei einem zuvor definierten *Break-Point*. Die so definierten *Model-Unit-Tests* können wie *Code-Unit-Tests* dazu verwendet werden, das Modell kontinuierlich zu validieren während es einer evolutionären Weiterentwicklung unterzogen wird.

Um eine Analyse und somit die Bewertung des *Model Under Test* zu erlauben, werden alle Aktionen im simulierten Modell aufgezeichnet (getraced). Durch diesen *Trace* ist es nun möglich, Metriken für die Simulation des *Model Under Test* zu berechnen. Interessante Informationen sind z. B. wann und wie oft ein Element (Aktivität, Aktion, Zustand) ausgeführt wurde und welche Bereiche des Modells nie aktiv waren. Durch eine grafische Visualisierung, durch Einfärben von bereits aktiven Modellierungselementen und der Verwendung verschiedenkräftiger Farben für oftmaliges Aufrufen, kann eine benutzerfreundliche Analyse des Modells gewährleistet werden. Damit kann das Modell während der Simulation kontinuierlich überprüft werden. Eventuell vorhandene Bedingungen am Modell (*model constraints*) führen dazu, dass bei deren Verletzung zur Simulationszeit

eine Exzeption erzeugt wird und dadurch die Simulation unterbrochen wird.

Die Model Simulation Engine

Die UML bietet ihren Benutzern jede Freiheit beim Modellieren. Modelle können unvollständig, verschieden granular und informal (natürlich sprachlich) definiert sein. Um Modelle simulieren zu können, müssen diese jedoch einen gewissen Grad an Formalität und Vollständigkeit besitzen. Ein Simulator ist vergleichbar mit einem Interpreter einer Programmiersprache, wie zum Beispiel VisualBasic, JavaScript, Python, etc. Diese Sprachen müssen einer formalen Syntax entsprechen, um gelesen, und eine formale Semantik- Definition besitzen, um eindeutig interpretiert werden zu können. Bei der Interpretation von Modellen verhält es sich identisch, daher können nicht alle Freiheitsgrade der konzeptionellen Modellierung ausgenutzt werden. Um Modelle simulieren zu können, muss die Syntax der Modellierungssprache von der Simulation Engine gelesen werden können. Syntaktische Korrektheit kann nun mit den oben erwähnten statischen Tests gewährleistet werden, ohne statische Überprüfung wird bei der Simulation ein Fehler (eine *Exception*) produziert. Um eingelesene Modelle interpretieren zu können, muss eine eindeutige formale Semantik vorliegen. UML 2.1 definiert dies durch ihr formales Metamodell. Es gibt allerdings dezidierte semantische Variationspunkte in der UML-Spezifikation, daher muss bei der Simulation definiert werden, wie einzelne Modellelemente interpretiert werden sollen.

Simulation von Aktivitäts- und Zustandsdiagrammen

UML AD werden verwendet, um Prozesse zu beschreiben. Dabei können diese Prozesse allgemeine Workflows oder detaillierte Algorithmen definieren. **UML StM** werden eingesetzt, um die Zustände eines Objektes oder eines gesamten Systems zu beschreiben. Je detaillierter das Modell, desto „einfacher“ die Simulation, da jegliche Informationen darüber, wie Daten erstellt und verarbeitet werden, bereits im Modell vorhanden sind. Dies schränkt allerdings die Freiheitsgrade der Modellierung weiter ein und bedeutet mehr Aufwand bei der Modellierung. Wenn diese Modelle anschließend zur Generierung von Code verwendet werden, spricht allerdings nichts gegen sehr detaillierte Modelle. Da jedoch nicht alle Modelle zur Codegenerierung verwendet werden, bietet die Model Simulation Engine die Möglichkeit, abstrakte Modelle in Zusammenhang mit externen Bibliotheken (den Mock-Objekten) zu simulieren.

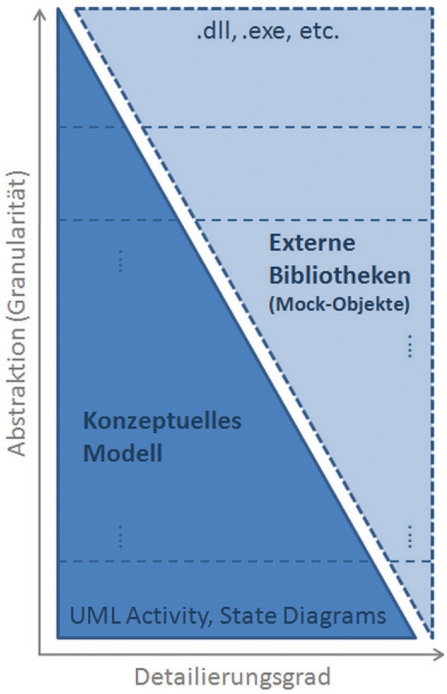

Abb. 1: Modellierungsdetails

Externe Bibliotheken (Mock-Objekte)

Abbildung 1 zeigt den Zusammenhang zwischen der Abstraktion eines Modells und die für die Simulation benötigten externen Bibliotheken. Je abstrakter das konzeptionelle Modell ist, desto aufwendiger müssen oftmals die externen Bibliotheken definiert sein. Beispielsweise kann der Prozess, Geld bei einem Geldautomat abzuheben, aus drei Schritten bestehen: „Karte einstecken“, „Betrag wählen“, „Geld entnehmen“. Um diesen Prozess zu simulieren, muss die Simulations-Engine ermitteln können, ob die Aktivität „Karte einstecken“ korrekt durchgeführt wurde oder ob ein Fehler aufgetreten ist. Diese Information muss durch ein externes Programm (dem Mock-Objekt) zur Verfügung gestellt werden. Derselbe Prozess kann nun weniger abstrakt mit mehr Details beschrieben werden. Dabei kann z. B. das Überprüfen der Karte, Überprüfen des PIN etc. genau modelliert werden.

Ganz ohne Mock-Objekt kommt man in den wenigsten Fällen aus. Im detaillierteren Fall des Prozesses „Geld abheben“ muss z. B. eine PIN-Eingabe abgefragt bzw. simuliert werden. Im ersten Fall wäre die Simulation interaktiv (PIN-Eingabe durch Benutzer), im zweiten Fall voll automatisch (das Mock-Objekt simuliert den Benutzer und stellt einen PIN zur Verfügung). Ein weiteres Mock-Objekt wäre dann erforderlich, welches den Server in der Bank repräsentiert.

Daraus ist ersichtlich, dass die Komplexität eines Systems immer erhalten bleibt, sie kann lediglich nach dem *Divide-and–Conquer*-Prinzip in Mock-Objekte und Model aufgeteilt und dadurch handhabbar gemacht werden. Der Autor spricht in diesem Zusammenhang gerne vom „Komplexitätserhaltungssatz“, wobei die Mock-Objekte im Idealfall trivial gehalten werden können.

Um mit diesem Ansatz effektiv und kostengünstig simulieren zu können, bietet es sich an, Mock-Objekte als generische, konfigurierbare und dadurch leicht wiederverwendbare Bibliotheken zu gestalten. Mock-Objekte können z. B. als Fassade für ganze externe Systeme fungieren, um Input-Daten bereitzustellen und Daten für das *Model Under Test* zu fordern. Durch die Konfigurationsmöglichkeit von Mock-Objekten wird die Dynamik der externen Systeme beschrieben. Gelieferte Daten werden z. B. aus einem definierten Bereich nach einer bestimmten Verteilung ausgewählt. Für den Prozess „Geld abheben“ repräsentiert ein Mock-Objekt den Bank-Server, der auf die PIN-Eingabe reagiert und je nach Konfiguration den PIN als valide bzw. invalide klassifiziert. Die Verteilung könnte z. B. 70% true und 30% false betragen. Damit wird durch das Mock-Objekt simuliert, dass 30% der PIN-Eingaben fehlschlagen. Ein solches Mock-Objekt ist trivial und beinhaltet wenig Implementierungslogik, erfüllt allerdings seinen Zweck, die erforderlichen Daten für das MUT zu liefern. Bei oftmaliger Wiederholung wird, durch die Verteilung im Mock-Objekt, verschieden auf das MUT reagiert. Damit kann das dynamische Verhalten des MUT überprüft werden, dies ist vor allem interessant, wenn mehrere Mock-Objekte Daten liefern.

Mock-Objekte

Wie im oberen Abschnitt bereits erwähnt, bilden Mock-Objekte eine Fassade, um externes Verhalten zu simulieren, d.h. auf Daten und Signale zu reagieren und dafür Input-Daten bzw. Signale zu liefern. Mock-Objekte können im einfachsten Fall triviale Implementierungen beinhalten wie das Liefern von vordefinierten Daten (z. B. das Liefern von true und false) nach einer vordefinierte Verteilung oder komplexere

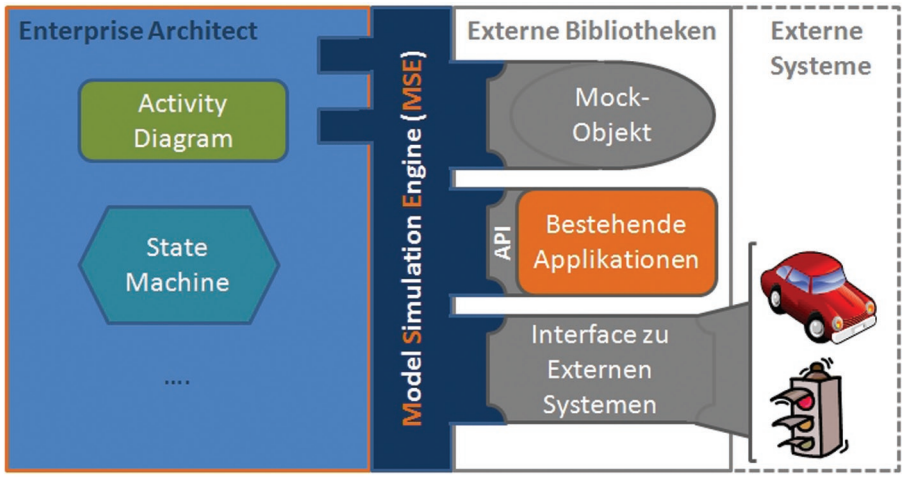

Abb. 2: Model Simulation Engine und Mock-Objekt

Berechnungen für die aus dem MUT gelieferten Daten/Signale durchführen.

Je nach Modellierungsdomäne werden Mock-Objekte trivial oder mit mehr Logik ausgestattet sein (siehe Abbildung 2). Zu bedenken ist dabei, dass Mock-Objekte ebenfalls Softwaresysteme sind und somit getestet werden müssen, um logische Fehler in deren Verhalten auszuschließen. Wenn sich das Mock-Objekt nicht so verhält wie das reale System, kann es, trotz valider Simulation des MUT, nach dessen Realisierung zu einem fehlerhaften System kommen (siehe Abbildung 2).

Ganze Applikationen können ebenso als Mock-Objekte genutzt werden. Das hat den Vorteil, dass die vollständig funktionsfähige Implementierung einer Anwendung verwendet wird. Das MUT interagiert nun mit dieser Applikation und simuliert das noch zu realisierende System. Wenn diese Applikation nicht direkt über eine API verfügt, ist es notwendig, eine Wrapper-API zu schreiben, um die einzelnen Funktionen der Applikation durch das MUT ansprechen zu können (Abbildung 2, Externe Bibliotheken, Mitte).

Eine weitere Möglichkeit besteht in der Nutzung externer Systeme als Mock-Objekte. Das bietet den weiteren Vorteil, dass direkt auf externe Hardware zugegriffen werden kann, d. h. die erstellten Modelle können verwendet werden, um Daten/Signale mit externen Systemen auszutauschen. Wenn das externe System keine Steuerlogik enthält, kann nun das Modell zum Simulieren des Verhaltens dieser Steuerlogik verwendet werden. Damit kann ohne großen Programmieraufwand externe Hardware angesprochen und gesteuert werden. Lediglich ein Interface für das externe System muss erstellt werden (Abbildung 2, Externe Bibliotheken, unten). Die Auswirkungen des simulierten Modells sind direkt im externen System ersichtlich, die Logik wird aber im Modell simuliert und kann daher einfach verändert und alle modellbasierten Tests können angewendet werden.

Zu beachten sind allerdings zeitkritische Steuerungen, da die Simulation des Modells nicht annähernd so performant sein kann, wie eine in Hardware gegossene Logik. Dies kann zu Synchronisationsproblemen beim Zusammenspiel von Simulation und realem System führen.

Um zeitkritische konzeptionelle Modelle simulieren zu können, ist es möglich, die Simulationszeit durch ein künstliches Verzögern zu normalisieren. Dabei wird die kleinste kritische Zeiteinheit zu einer im Modell simulierbaren Zeitspanne ausgedehnt, alle von dieser Zeitspanne abhängigen Zeiten werden relativ dazu verlängert. Das Problem der zeitkritischen Interaktion mit realen Systemen kann dadurch allerdings nicht gelöst, sondern es kann lediglich das logische Verhalten zeitkritischer Modelle simuliert werden.

Bei zeitunkritischen Systemen spielt die Simulationszeit eine untergeordnete Rolle und kann oftmals vernachlässigt werden. Um Änderungen am Modell visuell verfolgen zu können, ist eine zu performante Si-

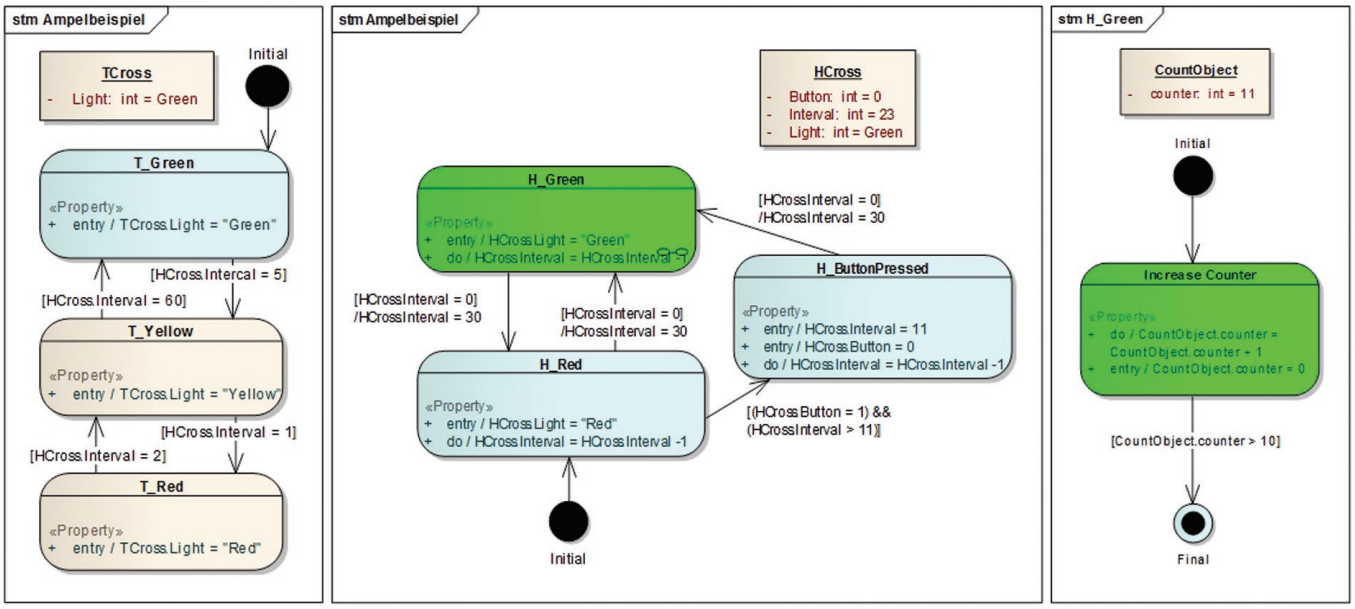

Abb. 3: Ampelsteuerung

mulation sogar ungeeignet und muss künstlich verlangsamt werden.

Um ein Debugging der Simulation gewährleisten zu können, ist es erforderlich, *Break-Points* zu setzen, an denen die Simulation automatisch unterbrochen wird und z. B. *step-by-step* weiter geführt werden kann. Durch das *Tracen* aller durchgeführten Schritte kann auch an einen früheren Zustand der Simulation zurückgesprungen werden, *step-by-step* oder direkt zu einem zuvor definierten *Break-Point*. Bis jetzt haben wir besprochen, wie die Model Simulation Engine für das Testen des Modells eingesetzt werden kann. Das Exekutieren von Modellen kann allerdings, wie schon angedeutet, auch zum Steuern eines externen Systems verwendet werden. Dabei übernimmt das Modell die Rolle des Codes und die Model Simulation Engine die Rolle des Interpreters dieser Sprache oder des Controllers einer Applikation.

Beispiel Ampelsteuerung

Einführend wurde ein Modell, wie Geld von einem Geldautomat abgehoben werden kann, skizziert und beschrieben, wie dieses Modell simuliert werden kann. In diesem Abschnitt wird ein konkretes Modell einer Ampelsteuerung gezeigt und diskutiert. Die Ampelsteuerung besteht aus zwei Zustandsautomaten (**siehe Abbildung 3**), einen für die Fußgängerampel (*HCross*) und einen für die Verkehrsampel (*TCross*). Beide Automaten werden mittels Guard-Bedingungen synchronisiert. Der Zustand *H_Green* besitzt einen Unterzustand *IncreaseCounter* (**Abbildung 3, rechts**), welcher die Logik des Modells detaillierter beschreibt (trivialerweise fungiert dieser Zustand lediglich als Zähler). Durch das Bilden von Unterzuständen bzw. das Erweitern des Modells um weitere Zustände, kann nun ein abstraktes Modell Schritt für Schritt um Details angereichert werden.

Somit kann die Logik modelliert werden, um zu den Daten zu gelangen, die sonst vom Mock-Objekt, durch z. B. triviale Funktionen, bereitgestellt werden würden.

Als Alternative zum Unterzustand *IncreaseCounter* ist auch die Verwendung eines Mock-Objekts möglich, welches die Funktionen *setCounter(int)* und *increaseCounter()* zur Verfügung stellt. Durch eine weitere Funktion *sleepFor(int)* ist es möglich, die Ausführungszeit zu verzögern, indem eine Zeit, z. B. in ms, angegeben wird, welche die Methode *increaseCounter()* verzögert, bevor sie beendet wird. Die Farben im Modell zeigen an, welche Zustände momentan

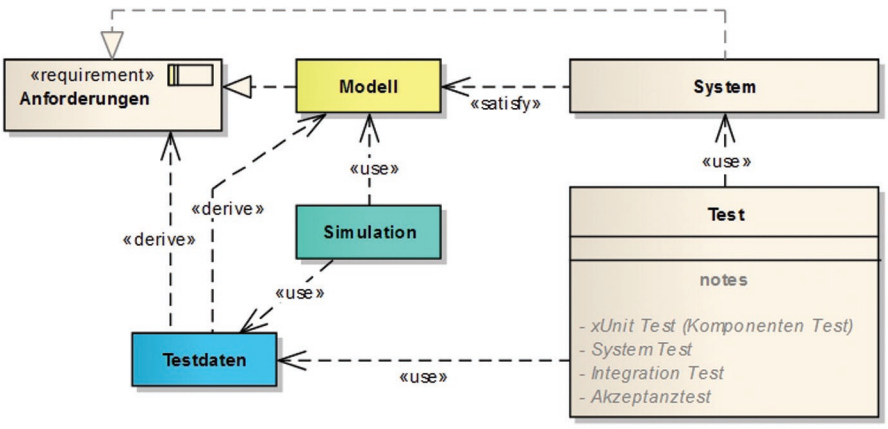

Abb. 4: Simulationsaufbau

aktiv sind (grüne Hintergrundfarbe), welche Zustände bereits aktiv waren (blaue Hintergrundfarbe) und welche Zustände noch nicht aktiv waren (rosa Hintergrundfarbe).

Durch die Definition und Konfiguration von initialen Objekten, den Test-Objekten (in Abbildung 3: *TCross, HCross, CounterObject*), werden Daten für die Simulation bereitgestellt. Beide Zustandsautomaten verändern nun diese Objekte. Durch das Definieren von „Zielobjekten" (Kopie des initialen Objekts und Konfiguration mit eventuell unterschiedlichen Werten) kann nun ein *Unit-Test* für dieses Modell erstellt werden. Aufgrund der Färbung der Zustände kann zudem untersucht werden, ob alle Zustände erreicht wurden und gegebenenfalls zusätzliche Test-Objekte erstellen werden müssen bzw. ob Fehler im konzeptionellen Modell vorhanden sind.

Das Ampelbeispiel ist ein kontinuierliches Modell. Es gibt keinen definierten Endzustand. Ein konkreter *Model-Unit-Test* kann nun durch Kopie des *HCross*- und *TCross*-Objektes definiert werden. Als *Break-Point* wird eine Bedingung (*model constraint*) angebracht, welche die Simulation nach 120 Zählschritten unterbricht und die beiden *HCross*- und *TCross*4-Objekte vergleicht. Nach z. B. 120 Zählschritten müssen beide Objekte wieder dem Anfangszustand entsprechen. Dies geschieht allerdings alle 60, 120, 180, etc. Zählschritte.

Fazit

Das Testen basierend auf Modellen ist kein neues Thema, es gibt bereits einige Ansätze und Tools, welche erlauben, konzeptionelle Modelle auszuführen und somit zu simulieren. Ein konkreter Vertreter ist z. B. der *NModel* Ansatz von [Jac08]. Hierbei wird allerdings von programmierten Modellen und nicht von grafischen Modellen ausgegangen.

Interessant ist die Flexibilität, welche eine Model Simulation Engine für grafische Modelle mit ihren Schnittstellen zu externen Bibliotheken bietet. Durch diesen Ansatz besteht die Möglichkeit, Modelle beliebigen Abstraktionsniveaus durch Bereitstellung von Mock-Objekten zu simulieren. Durch die Definition von Input-Daten und daraus resultierenden „erwünschten" Output-Daten können *Model-Unit-Tests* realisiert werden. Somit kann in sich weiterentwickelnden Modellen die Konsistenz sichergestellt werden. Durch das <I>Tracen<I> von Simulationsschritten und durch das Überprüfen von Modelleinschränkungen können Modelle kontinuierliche getestet werden, auch wenn sie keine dezidierten Input-Daten brauchen bzw. Output-Daten liefern.

Abbildung 4 zeigt den Zusammenhang von Modell, System, Testdaten und der Simulation. Das Modell berücksichtigt die geforderten Anforderungen. Das System wird basierend auf dem Modell implementiert und berücksichtigt demnach ebenfalls die Anforderungen. Testdaten werden aufgrund von Anforderungen und Einschränkungen im Modell ermittelt. Die Simulation verwendet die Testdaten und das Modell, um das Modell zu validieren.

spricht ordnungsgemäß **Abbildung 4** und alle Mock-Objekte entsprechen den externen Systemen, für die sie stehen, besitzt der Hersteller nach dem Testen (Simulieren) des Modelles ein verifiziertes konzeptionelles Modell. Das reale System sollte sich nun so verhalten wie das simulierte Modell, mit den bereits oben erwähnten Einschränkungen bei Echtzeitsystemen. Da nun bereits Testdaten für die Simulation des Modells vorhanden sind, besteht die Möglichkeit, dieselben Testdaten, mit denen auch das Modell getestet

wurde, zum Testen des realen Systems heranzuziehen. Das Erstellen von ausführbaren Modellen kommt so einem *early prototyping* nahe. Konzeptionelle (logische) Fehler im System können schon bei der Erstellung der Modelle erkannt und behoben werden, je nach Qualität und Aufwand für die Erstellung von Mock-Objekten können auch plattformspezifische Probleme frühzeitig erkannt und in der Planung berücksichtigt werden.

Die Model Simulation Engine (MSE) der Firma LieberLieber deckt den Kern der Simulationen von AD und StM ab und bietet die Möglichkeit, beliebige Mock-Objekte in die Simulation einzubinden. Das Ableiten von Testdaten wird momentan nicht unterstützt. Mehr Informationen bezüglich des aktuellen Entwicklungsstands der MSE erhalten Sie bei der Firma www.LieberLieber.com bzw. www.SparxSystems.at. Ein Demonstrationsvideo der Simulation zum Steuern eines ferngesteuerten Autos finden Sie unter: http//blog.lieberlieber.com/2009/03/19/uml-execution-with-ea/.

Referenzen

[Bla04] Blackburn, M., Busser, R. & Nauman, A: Why Model-Based TestAutomation is Different and What You Should Know to Get Started. International Conference on Practical Software Quality and Testing, Washington, USA, 2004.

[Sne09] Harry M. Sneed et al.: Der Systemtest, Hanser, München, 2009.

[Sne88] Harry M. Sneed: Software Qualitätssicherung, Rudolf Müller, Köln, 1988.

[Hau03] Rudolf Hauber et al.: Modelbasiertes Testen, OBJEKTspektrum, 03/2003.

[All08] Thomas Allweyer: BPMN - Business Process Modeling Notation, Books on Demand GmBH, Norderstedt, 2008.

[Poh07] Klaus Pohl: Requirements Engineering – Grundlagen, Pronzipien, Techniken, dpunkt Verlag, Heidelberg, 2007.

[Jac08] Jonathan Jacky, Margus Veanes, Colin Campbell, Wolfram Schulte: Model-based Software Testing and Analysis with C#, Cambridge University Press, 2008.

[SPX] SparxSystems Central Europe Software GmbH www.SparkySystems.at