

Qualitätsorientierte Analyse von Modellierungsansätzen

Stephan Kleuker
Fachhochschule Osnabrück
Fakultät Ingenieurwissenschaften und Informatik
- Software-Entwicklung -
Postfach 1940
49009 Osnabrück
s.kleuker@fh-osnabrueck.de

Abstract: Die systematische Modellierung von Geschäftsabläufen verfolgt als wesentliche Ziele die Wiederverwendbarkeit und die schnellere Realisierung unter Nutzung vorhandener Bausteine und Generierungsrichtlinien. Um diese Ziele zu erreichen, müssen die Ergebnisse verschiedenen Qualitätsansprüchen genügen, deren Elementarste und Wichtigste die funktionale Korrektheit ist. In der Informatik und Wirtschaftsinformatik wurden verschiedene Ansätze, wie die Geschäftsprozessmodellierung, Model Driven Software Development und Model Checking entwickelt, die ausgehend von der Modellierung hin zum lauffähigen System führen. Nach einer kurzen Vorstellung der Ansätze werden diese bezüglich ihrer Anwendungsmöglichkeiten und der garantierten Qualität der Ergebnissysteme verglichen sowie Synergieeffekte unter Nutzung klassischer Testverfahren vorgestellt.

1 Einleitung

Modellbildung hilft in unterschiedlichen Bereichen Lösungen für Probleme zu konzipieren, die ohne ein Modell kaum konstruierbar wären. In der Software-Entwicklung werden sehr unterschiedliche Arten zur Modellbildung mit unterschiedlichen Kernzielen genutzt. Hier werden drei verschiedene Ansätze vorgestellt, die alle die Entwicklung qualitativ hochwertiger Softwaresysteme unterstützen. Es wird analysiert, wann welche Modellierungsart aktuell eingesetzt wird, sowie welche Chancen und Risiken hinter diesen Ansätzen verborgen sind. Ausgehend von der Betrachtung klassischer Qualitätssicherungsansätze werden Integrationsmöglichkeiten der vorgestellten Ideen zur Steigerung der Korrektheit von Software diskutiert.

Diese Arbeit wird im Rahmen des Projekts „Korrekte verteilte Java-Applikationen“ des Projektträgers AiF durch das BMBF gefördert.

2 Ideen und Stand der analytischen Qualitätssicherung

Die Qualitätssicherung (QS) als Teil der Softwareentwicklung spielt für den Projekterfolg eine immer größere Rolle, da unzuverlässige Softwaresysteme von Kunden immer weniger akzeptiert werden. Da allerdings die Mittel für die Qualitätssicherung nicht beliebig erhöht werden können, stellt sich die Frage, welche Ansätze der Qualitätssi-

cherung unter welchen Randbedingungen die größten Erfolgswahrscheinlichkeiten haben.

Innerhalb der Qualitätssicherung gibt es verschiedene Teilgebiete, die z. B. konstruktive Maßnahmen, wie Guidelines und die Auswahl passender Schulungen, und analytische Maßnahmen, wie das klassische Testen und den Einsatz von Metriken [Hen96], umfassen. Weitere Klassifizierungen der QS-Maßnahmen sind möglich, wie es z. B. in [Lig02] [Kle09a] [Hof08] gezeigt wird.

Die analytischen Verfahren können generell wie folgt klassifiziert werden:

- Funktionale Tests: Ansätze zur Prüfung, ob ein in Entwicklung befindliches System die geforderte Funktionalität liefert, dies sind z. B.:
 - Äquivalenzklassenanalyse: Ansatz zur systematischen Auswahl möglichst weniger Testfälle, die möglichst viele potenzielle Fehler aufdecken können
 - Überdeckungstests: Ansätze zur Prüfung, ob bestimmte Anweisungen, Ablaufpfade oder Boolesche Bedingungen garantiert in Testfällen berücksichtigt werden
- Prüfung nichtfunktionaler Anforderungen, z. B.:
 - Performance-Analyse (Lasttest): Prüfung, ob ein in Entwicklung befindliches System auch unter bestimmten Lasten die erwartete Reaktionsgeschwindigkeit beibehält
 - Speichernutzung: Prüfung, ob ein in Entwicklung befindliches System auch unter bestimmten Lasten Grenzen für die maximale Speichernutzung beibehält
 - Codequalitätsanalyse durch Metriken: Berechnung von speziellen Kennzahlen einer Software, um Indikatoren für die Qualität des Programmcodes hinsichtlich Wartbarkeit und Änderbarkeit zu erhalten
 - Usability-Prüfung: Analyse, ob ein in Entwicklung befindliches System nutzbar ist
- Manuelle Prüfverfahren: Menschen prüfen (Teil-)Produkte eines in Entwicklung befindlichen Systems nach bestimmten Kriterien, die typischerweise nicht automatisch geprüft werden können, wie z. B. die Lesbarkeit von Dokumenten und die vollständige Umsetzung von Anforderungen

Während im Umfeld der Embedded-Software z. B. im Automobilbereich Testwerkzeuge im Preisbereich von 100 T€ liegen können, wurden in den letzten Jahren im Java-Umfeld, ausgehend von JUnit im Open Source Bereich einige sehr gute Werkzeuge zum Test der Funktionalität, der Performance und des Speicherhaltens entwickelt, die auch in Unternehmen frei eingesetzt werden können. Praxisberichte zeigen dabei, dass die Verbreitung von Testwerkzeugen und die damit zusammenhängende Automatisierungsmöglichkeit von Testabläufen allerdings noch sehr unterschiedlich sind.

Testen selbst wird ein wesentlicher Beitrag zur Qualitätssicherung bleiben, trotzdem gilt die zentrale Regel, dass man mit Tests nur die Existenz von Fehlern aufdecken und niemals die Korrektheit von Software garantieren kann. Es stellt sich die Frage, wie man ein

Mehr an Korrektheit in der Software-Entwicklung garantieren kann, um so die potenziell immensen Kosten für aufwändige Tests und nachträgliche Korrekturen zu reduzieren.

Der Einsatz von Modellierungstechniken verspricht einen wesentlichen Beitrag zum „Mehr“ zu leisten.

3 Modellierungsansätze

3.1 Vergleichskriterien für Modellierungsansätze

Obwohl die vorgestellten Modellierungsideen vordergründig unterschiedliche Ziele verfolgen, können Sie alle einen wesentlichen Beitrag zur Entwicklung korrekter Software leisten. Aus diesem Grund kann man sie bezüglich verschiedener Aspekte vergleichen, die in der folgenden Tabelle 1 beschrieben sind.

Tabelle 1: Bewertungskriterien für Modellierungsansätze

Kriterium	Beschreibung
intuitive Verständlichkeit	wie schnell sind die Modelle von Nicht-Experten des Modellierungsansatzes zu verstehen
Erlernbarkeit	wie lange dauert es, potenziellen Nutzern dieses Ansatzes den Ansatz verständlich zu machen
Verbreitung	wie stark ist der Ansatz in der Praxis verbreitet
Präzision	welche Garantie gibt es, dass Erkenntnisse des Modells garantiert in die Praxis übertragbar sind

3.2 Geschäftsprozessmodellierung

Software-Projekte werden in Unternehmen nicht zum Selbstzweck durchgeführt. Sie sind Teil des Unternehmens, das sich konkrete Ziele gesetzt hat. Diese Ziele können von kurzfristiger Gewinnerreichung bis hin zum anerkannten Marktführer für Qualitätsprodukte gehen. Ein Unternehmen besteht dabei aus sehr unterschiedlichen Prozessen, die ineinander greifen müssen. Beispiele für Prozesse sind: Unternehmensführung, Vertrieb, Personalmanagement, Entwicklung, Support und Gebäudeverwaltung [Wal01]. Diese Prozesse können durch passende Modellierungssprachen beschrieben werden [Gad03]. Dies ist z. B. durch ereignisgesteuerte Prozessketten [Sei06] oder die Aktivitätsdiagramme der UML [OWS03] möglich. Dabei muss ein Prozess folgende Fragen beantworten.

- Was soll in diesem Schritt getan werden?
- Wer ist verantwortlich für die Durchführung des Schritts?
- Wer arbeitet in welcher Rolle in diesem Schritt mit?
- Welche Voraussetzungen müssen erfüllt sein, damit der Schritt ausgeführt werden kann?

- Welche Teilschritte werden unter welchen Randbedingungen durchgeführt?
- Welche Ergebnisse kann der Schritt abhängig von welchen Bedingungen produzieren?
- Welche Hilfsmittel werden in dem Prozessschritt benötigt?
- Welche Randbedingungen müssen berücksichtigt werden?
- Wo wird der Schritt ausgeführt?

Die Prozessmodellierung kann auch für einzelne Prozesse hilfreich in der Anforderungsanalyse sein, wenn diese Prozesse mit neuer Software unterstützt werden sollen. Die dargestellten Abläufe sind durch das neue System zu unterstützen. In einem weiteren Schritt kann man versuchen, aus Modellen direkt die Software zu generieren, z. B. kann aus der Prozessbeschreibung für ein Dokument folgen, welche Personen in welchen Rollen dieses zu bearbeiten haben. Es ist dann möglich, ein Dokumentenverwaltungssystem zu erzeugen, das die jeweils nächste Rolle informiert, dass sie für dieses Dokument tätig werden soll. Um die Generierung zu vereinheitlichen, wurde die Business Process Execution Language (BPEL) [bpe] entwickelt, mit der das Zusammenspiel innerhalb von Prozessen unter Beantwortung der genannten Fragen beschrieben werden kann. Da die BPEL eher für die maschinelle Verarbeitung entwickelt wurde, werden wieder graphische Spezifikationssprachen zur Visualisierung der Abläufe genutzt. Neben den genannten Modellierungssprachen spielt hier auch die Business Process Model Notation (BPMN) [bpm] eine Rolle.

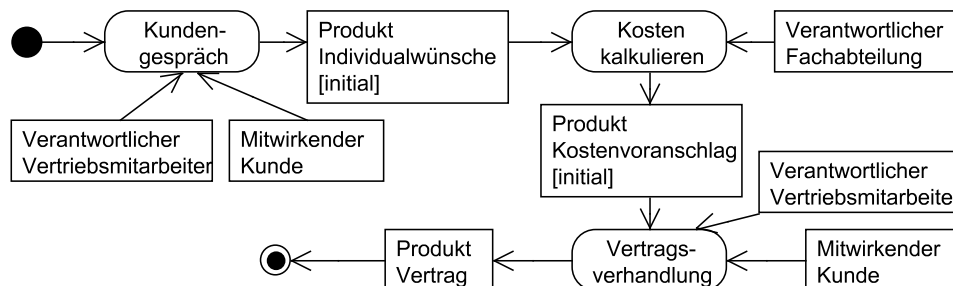


Abbildung 1: Ausschnitt aus Beispielprozess

Abbildung 1 zeigt einen Ausschnitt eines Beispielprozesses, der den gewünschten Ablauf einer Vertragsverhandlung beschreibt. Man erkennt die Arbeitsschritte in abgerundeten Rechtecken, die resultierenden Produkte in ihrem aktuellen Zustand und die beteiligten Personen. Die Nutzung weiterer graphischer Symbole könnte hier die Lesbarkeit erhöhen.

Der Vorteil solcher Modellierungsansätze ist, dass sie für viele Leute unmittelbar verständlich sind und so die Diskussion über gewünschte Softwaresysteme zielgerecht vorantreiben können. Der beschriebene Ablauf könnte z. B. zur Realisierung eines Planungssystems gehören, mit dem ein Unternehmen seine Auslastung kalkulieren kann. Jeder der Arbeitsschritte müsste mit einem solchen System kommunizieren. Die Realisierung des Systems ist allerdings noch sehr weit von einer Implementierung entfernt.

Tabelle 2: Bewertung der Geschäftsprozessmodellierung

Kriterium	Auswertung für Geschäftsprozessmodellierung
intuitive Verständlichkeit	meist sehr schneller Zugang auch für Nicht-IT-Experten; Verständlichkeit kann mit Modellgröße drastisch abnehmen
Erlernbarkeit	sehr schnell erlernbar; eventuelle Probleme, wenn zu viele Sprachmöglichkeiten gegeben
Verbreitung	sehr hoch, in jedem Buch zur Anforderungsanalyse als wesentliches Hilfsmittel beschrieben; neben der Software-Entwicklung auch in vielen anderen Bereichen verbreitet
Präzision	abhängig vom gewählten Ansatz; häufig besteht aber für nicht an der Modellierung beteiligte Personen die Möglichkeit zur Fehlinterpretation, da Details fehlen

3.3 Model Driven Software Development

Hier wird ein abstraktes Modell des zu entwickelnden Systems erstellt, das typischerweise auf einem vorher definierten Metamodell basiert. Danach wird das abstrakte Modell mit typischerweise auch im Projekt zu entwickelnden Transformationsregeln in ein weniger abstraktes Modell, z. B. direkt in ein ausführbares Programm, übersetzt. Diese Transformation kann in mehreren Schritten passieren, wobei Konkretisierungen z. B. in der Wahl des Betriebssystems, der Infrastruktur, der Programmiersprache und der Persistenz-Verwaltung bestehen können. Typische Vertreter sind die Model Driven Architecture (MDA) [GPR06] [WKB04] und das verwandte Model Driven Software Development (MDSD) [SVE07]. In einer vergleichbaren Variante werden spezielle Sprachen für bestimmte Applikationsbereiche betrachtet (Domain Specific Languages, DSL) [KK06], die für konkrete Applikationen genutzt werden, die dann in lauffähige Software transformiert werden. Erste Ansätze zum Model Driven Test, der Testerstellung parallel zur Entwicklung, werden bereits betrachtet [EGL07].

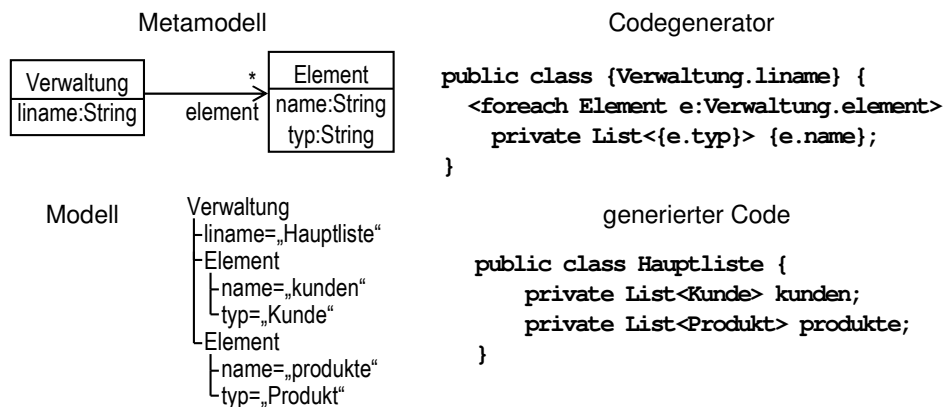


Abbildung 2: Beispiel für modellgetriebene Code-Generierung

Abbildung 2 zeigt eine Variante des modellgetriebenen Ansatzes, mit dem Klassen generiert werden, die eine beliebige Anzahl von Listen verwalten können. Im Metamodell wird festgelegt, welche Bausteine es in der Modellierung geben darf (Verwaltung, Element), welche Eigenschaften diese Bausteine haben und wie sie in Beziehung stehen; hier dürfen beliebig viele Elemente in Beziehung zu einem Verwaltungsbaustein stehen. Rechts daneben steht der Codegenerator, der auf alle Teile des Metamodells zugreifen darf. Das konkrete Modell nutzt die Bausteine und gibt konkrete Werte an, die dann zur Generierung lauffähigen Quellcodes genutzt werden.

Tabelle 3: Bewertung des Model Driven Software Development

Kriterium	Auswertung für Model Driven Software Development
intuitive Verständlichkeit	stark abhängig von der individuell geschaffenen Modellierungsgrundlage; durch Einsatz graphischer Elemente mit großen Potenzial
Erlernbarkeit	Abhängig vom einzuhaltenden Formalismus des Modellierungsansatzes müssen generell die Modellierungsregeln präzise eingehalten und ihre Auswirkungen im Detail verstanden werden
Verbreitung	aktueller Entwicklungszweig, der typischerweise zu zwar ähnlichen aber trotzdem auf Problembereiche und teilweise Unternehmen angepassten Ansätzen führt; nach erstem Hype ist Bedeutung für das Software Engineering noch unklar
Präzision	Durch die klare Vorgabe der Modellierungsmöglichkeiten und der automatischen Übersetzung werden Fehlermöglichkeiten sehr stark reduziert; Fehler des syntaktisch korrekten Modells oder Fehler der Übersetzungsregeln können verdeckt bleiben

3.4 Model Checking

Immer wichtiger wird in den Embedded Systems und auch der klassischen Software-Entwicklung der Einsatz formaler Methoden, wie z. B. Model Checking. Beim Model Checking werden die Spezifikation und die Anforderungen beschrieben und durch den Model Checker geprüft, ob die Anforderungen von der Spezifikation erfüllt werden. Aus der theoretischen Informatik, z. B. dem Halteproblem [VW04], ist bekannt, dass dieser Ansatz nicht immer möglich ist. Aus diesem Grund müssen Spezifikations- und Anforderungssprache in der Ausdrucksmächtigkeit eingeschränkt werden. Die Anforderungen an das System werden in einer passenden Logik [Pnu77] formalisiert. Danach kann ein Model Checking-Algorithmus [BCM90] vollautomatisch prüfen, ob das Modell die Anforderungen erfüllt. Das so verifizierte Modell kann dann in weiteren Entwicklungsschritten genutzt werden. Dieser Ansatz ist 2007 mit dem Turing Award für Clarke, Emerson und Sifakis [CES86] ausgezeichnet worden. Typische Vertreter von Model Checking-Werkzeugen sind der Stanford Model Verifier (SMV) [@smv], das für verteilte Realzeitsysteme konzipierte Uppaal [@upp], das 2001 mit dem renommierten ACM Software System Award ausgezeichnete SPIN [Hol04] und der auf Java-Bytecode basierende Java PathFinder [@jpf]. Eine Übersicht befindet sich in [DKW08].

Abhängig von der Wahl der Spezifikationssprache ist der Unterschied zu Programmiersprachen teilweise gering. Spezifikationssprachen unterstützen aber meist eine höhere

Abstraktion. Das folgende Beispiel zeigt die Möglichkeit, Nichtdeterminismus in PROMELA, der Sprache von SPIN, auszudrücken.

```
byte eingabe=0;
bool terminiert=false;

active proctype Nichtdet(){
  do
    :: eingabe < 100 -> eingabe=eingabe+1;
    :: true -> break;
  od;
  terminiert=true;
}
```

In einem Prozess `Nichtdet()` gibt es eine `do`-Schleife, deren Teilelemente die Form „Wächter \rightarrow Aktion“ haben. Bei der Ausführung der Schleife wird zunächst geprüft, welche Wächter nach wahr ausgewertet werden können und dann nichtdeterministisch einer dieser Wächter ausgewählt, um die zugehörige Aktion auszuführen. Falls in dieser Aktion kein `break` steht, kehrt die Ausführung nach der Aktion wieder zum Anfang der Schleife zurück. Nach der Terminierung des Prozesses hat `eingabe` einen ganzzahligen Wert aus dem Intervall $[0,99]$.

Beim Model Checking werden dann alle möglichen Ausführungen betrachtet; es werden alle Zustände bestimmt, die das System erreichen kann. Das Model Checking wird bezüglich einer Anforderung durchgeführt, die in einer formalen Logik spezifiziert wird. Beispiele für solche Anforderungen sind:

- [] `eingabe<120`: Für alle Ausführungspfade, also schrittweisen Ausführungen, gilt, dass `eingabe` immer ([]) einen Wert kleiner als 120 hat. Die Anforderung ist für das Beispiel erfüllt.
- <> `eingabe>20`: Für alle Ausführungspfade gilt, dass `eingabe` irgendwann(<>) einen Wert größer 20 annimmt. Die Anforderung ist nicht erfüllt.
- []<> `(eingabe%2==0 || terminiert)`: Für alle Ausführungspfade gilt, dass immer mal wieder der Wert von `eingabe` gerade oder der Prozess terminiert ist. Die Anforderung ist erfüllt. Man erkennt, dass `terminiert` als Hilfsvariable eingeführt wurde, um eine Eigenschaft prüfbar zu machen.

Die Anforderungslogiken haben dabei eine gewisse Mächtigkeit, so dass nicht alle Anforderungen formalisierbar sind. Weiterhin hat es sich in der Lehre gezeigt, dass der Zugang zu einfachen Formeln zwar intuitiv ist, es bei komplexeren Anforderungen aber einer intensiven Schulung in der Logik bedarf, um informelle Anforderungen korrekt umzusetzen.

Verschiedene Spezifikationsprachen bieten sehr unterschiedliche Sprachkonstrukte an [Kle09b], die in der Spezifikationsentwicklung genutzt werden können. Dabei sollte man wie bei allen vorgestellten Ansätzen darauf achten, dass die Erkenntnisse aus dem Modell in die Praxis übertragbar sind. PROMELA kann z. B. für klassische sequenzielle Programme aber auch für verteilte Systeme eingesetzt werden. Uppaal ermöglicht z. B. die zusätzliche Betrachtung von Zeitaspekten.

Offen bleibt die Frage, wie Ergebnisse der Modellierung in die Praxis übersetzt werden können. Hier bieten sich Transformationen an, alternativ gibt es auch Ansätze, das Model Checking direkt in Teilmengen von Programmiersprachen, wie C [cbm] und Java durchzuführen.

Tabelle 4: Bewertung des Model Checkings

Kriterium	Auswertung für Model Checking
intuitive Verständlichkeit	abhängig vom genauen Ansatz; typischerweise für Personen, die Programmieren können, verständlich
Erlernbarkeit	der Schritt vom kleinen Beispiel zur größeren Anwendung ist enorm groß und setzt ein sehr präzises Denken voraus
Verbreitung	hat im Bereich der Hardware- und hardwarenahen Entwicklung einige Verbreitung (z. B. Siemens und Bosch); stärkerer Einfluss auf die klassische Software-Entwicklung wird immer wieder erwartet, ist aber in den letzten 15 Jahren kaum erkennbar; meist nur optional in Lehrplänen der Hochschulen
Präzision	bis zu gewissen Modellgrößen echter Nachweis von Korrektheit; durch Modell und Anforderung werden die untersuchten Eigenschaften von zwei Seiten beschrieben, was Fehlerquellen einschränkt, aber nicht garantiert eliminiert

4 Integrationsmöglichkeiten

Die Vorstellung der Ansätze hat bereits angedeutet, dass eine strikte Trennung der Ideen unnötig ist. Teilweise werden Verknüpfungen der Ansätze bereits versucht, wobei Erkenntnisse aus den anderen Modellierungsansätzen häufiger ignoriert werden.

Der Schritt in die Formalisierung der Geschäftsprozessmodellierung wird durch die BPEL deutlich, aber auch Ansätze im Modellierungsumfeld von ARIS und SAP zielen darauf, dass Modelle ausführbar werden. Der Schritt zur Ausführbarkeit benötigt aber immer eine sehr präzise Beschreibung von Parametern, so dass die intuitive Verständlichkeit der Modellierung, also die für den Ansatz zentrale Eigenschaft für den Erfolg, verloren geht.

Auch modellgetriebene Ansätze können mit Model Checking-Ideen enger integriert werden. Es stellt sich die Frage nach einer Modellierungssprache, die unmittelbar von einem Model Checker unterstützt wird oder nach einer Metasprache, deren Modelle in Modelle des modellgetriebenen Ansatzes und des Model Checkers transformiert werden können. Wichtig ist, dass ein Model Checking-Einsatz eine präzise Semantik fordert, die für den modellgetriebenen Ansatz einen zusätzlichen Qualitätsaspekt liefern kann. Weiterhin können temporale Ideen der Anforderungslogiken des Model Checkings in Anforderungssprachen, die mit der in der UML eingesetzten Object Constraint Language (OCL) [WK02] verwandt sind, übertragen werden.

Eine kritische Analyse der Modellierungsansätze zeigt weiterhin, dass zwar mit der Zunahme der Präzision die Möglichkeit für Fehler abnimmt, allerdings Fehler trotzdem möglich bleiben. Hier eröffnet sich wieder ein weites Integrationsfeld, bei dem Modellierungsansätze mit Ideen der klassischen Qualitätssicherung verknüpft werden. Angelehnt an den bereits in der Erforschung befindlichen Ansätzen zur modellgetriebenen Testfallerzeugung stellt sich die Frage, wie man die Erstellung von Testfällen in die vorgestellten Modellierungsansätze einbauen kann. Das Ziel muss es sein, eine minimale Anzahl von Tests zu haben, mit denen Eigenschaften geprüft werden können, deren Korrektheit während der Entwicklung nicht garantiert werden kann. Dabei spielt die Testbarkeit eine wichtige Rolle. Ähnlich, wie man Programme erweitern muss, damit alle Bereiche testbar werden, stellt sich die Frage, wie man solche notwendigen Erweiterungen in die Modellierungsansätze einbauen kann.

Eine für die Forschung in diesem Bereich weitere wichtige Erkenntnis ist, dass es nicht eine einzige ultimative Lösung geben wird, da diese unterschiedliche Konzepte in möglichst einfacher Form unterstützen müsste, dadurch aber selbst komplex, unverständlich und für Werkzeuge nicht effizient handhabbar würde.

5 Fazit

Die Modellbildung ist ein weiterer wichtiger Schritt zur effektiven Entwicklung qualitativ hochwertiger Software. Bekannte Modellierungsansätze unterscheiden sich wesentlich im intuitiven Zugang für Nutzer, in der formalen Präzision und der möglichen Garantie der Korrektheit. Da die Ziele der Modellnutzung in den betrachteten Ansätzen in wesentlichen Bereichen übereinstimmen, steht die aktuell betrachtete Frage zur Integration der Ideen ein großes wissenschaftliches und wirtschaftliches Potenzial dar. Praktische Erfahrungen mit den verschiedenen Ansätzen haben gezeigt, dass eine intuitiv nutzbare und formal fundierte Modellierungssprache für beliebige Anwendungsgebiete nicht existiert. Aus diesem Grund sollten sich Integrationsbestrebungen auf fachliche Anwendungsgebiete konzentrieren und die erreichten Kenntnisse intensiv austauschen.

Auch mit dem aufgezeigten großen Potenzial zur Verbesserung der Qualität der basierend auf Modellen entwickelten Systeme, muss die klassische Qualitätssicherung mit in die Entwicklung eingebunden werden. Trotz der Möglichkeit die Erfüllung von Anforderungen zu verifizieren, bleibt unabhängig vom Ansatz ein Restfehlerpotenzial basierend auf fehlerhaften Modellierungen, Anforderungsformulierungen oder fehlerhafter Transformationsalgorithmen übrig. Aus dieser Überlegung wächst für die modellbasierten Ansätze die Forderung in ihren Entwicklungsprozessen die Generierung von Hinweisen auf das passende QS-Verfahren und die Vereinfachung des Einsatzes von Testansätzen zu integrieren.

Literaturverzeichnis

[@bpe] Web Services Business Process Execution Language, <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>, zuletzt abgerufen am 1.9.2009

- [@bpm] BPMN, <http://www.omg.org/spec/BPMN/>, zuletzt abgerufen am 1.9.2009
- [@cbm] The CBMC Homepage, <http://www.cprover.org/cbmc/>, zuletzt abgerufen am 1.9.2009
- [@jpf] Java PathFinder, <http://javapathfinder.sourceforge.net/>, zuletzt abgerufen am 1.9.2009
- [@oaw] openArchitectureWare.org - Official openArchitectureWare Homepage, <http://www.openarchitectureware.org/>, zuletzt abgerufen am 1.9.2009
- [@smv] SMV Model Checker Free Download, <http://www.kenmcmil.com/smv.html>, zuletzt abgerufen am 1.9.2009
- [@upp] UPPAAL, <http://www.uppaal.com/>, zuletzt abgerufen am 1.9.2009
- [@mda] MDA, <http://www.omg.org/mda/>, zuletzt abgerufen am 1.9.2009
- [BCM90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang. Symbolic Model Checking: 1020 States and Beyond, Proceedings of the 5th Annual Symposium on Logic in Computer Science, Seiten 428–439, 1990
- [CES86] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, ACM Transactions on Programming Languages and Systems, 8(2), Seiten 244–263, April 1986
- [DKW08] V. D’Silva, D. Kröning, G. Weißenbacher, A Survey of Automated Techniques for Formal Software Verification, IEEE TCAD, 27(7), Seiten 1165-1178, Juli 2008
- [EGL07] G. Engels, B. Güldali, M. Lohmann, Towards Model-Driven Unit Testing, in: Satellite Events at the MoDELS 2006, LNCS 4364, Seiten 182 - 192, Springer, Berlin Heidelberg, 2007
- [Gad03] A. Gadatsch, Grundkurs Geschäftsprozess-Management, 3. Auflage, Vieweg, Wiesbaden, 2003
- [GPR06] V. Gruhn, D. Pieper, C. Röttgers, MDA, Springer, Berlin Heidelberg, 2006
- [Hen96] B. Henderson-Sellers, Object-Oriented Metrics, Measures of Complexity, Prentice Hall, USA, 1996
- [Hof08] D. W. Hoffmann, Software-Qualität, Springer-Verlag, Berlin Heidelberg, 2008
- [Hol04] G. Holzmann, The SPIN model checker, Addison-Wesley – Pearson Education, Boston, 2004
- [KK06] M. Klar, S. Klar, Einfach Generieren, Hanser, München Wien, 2006
- [Kle09a] S. Kleuker, Grundkurs Software-Engineering mit UML, Vieweg+Teubner, Wiesbaden, 2008
- [Kle09b] S. Kleuker, Formale Modelle der Softwareentwicklung, Vieweg+Teubner, Wiesbaden, 2009
- [Lig02] P. Liggesmeyer, Software-Qualität. Testen, Analysieren und Verifizieren von Software, Spektrum Akademischer Verlag, Heidelberg Berlin Oxford, 2002
- [Pnu77] A. Pnueli, The Temporal Logic of Programs, Proceedings of the 18th IEEE Symposium on Foundations of Computer Science, Seiten 46–57, 1977
- [OWS03] B. Oestereich, C. Weiss, C. Schröder, T. Weilkiens, A. Lenhard, Objektorientierte Geschäftsprozessmodellierung mit der UML, dpunkt, Heidelberg, 2003
- [Sei06] H. Seidlmeier, Prozessmodellierung mit ARIS, 2. Auflage, Vieweg, Wiesbaden, 2006
- [SVE07] T. Stahl, M. Völter, S. Efftinge, A. Hasse, Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management, 2. Auflage. d.punkt, Heidelberg, 2007
- [VW04] G. Vossen, K.-U. Witt, Grundkurs Theoretische Informatik, 3. Auflage, Vieweg, Wiesbaden, 2004
- [Wal01] E. Wallmüller, Software-Qualitätsmanagement in der Praxis, 2. Auflage, Hanser, München Wien, 2001
- [WK02] J. Warner, A. Kleppe, The Object Constraint Language, Addison-Wesley, USA, 2002
- [WKB04] J. Warner, A. Kleppe, W. Bast, MDA Explained, Addison-Wesley, USA, 2004